securityandtrust.lu

FR

Final Report

F. Pastore, O. Cornejo, E. Viganò

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

ITT-1-9873-ESA-FAQAS-D2

Issue 5, Rev. 3

October 14, 2021

UNIVERSITÉ DU LUXEMBOURG

Revisions

Issue Number	Date	Authors	Description
ITT-1-9873-ESA-FAQAS-D2 Issue 1 Rev. 1	June 12th, 2020	Fabrizio Pastore, Oscar Cornejo	Initial release.
ITT-1-9873-ESA-FAQAS-D2 Issue 2 Rev. 1	June 25th, 2020	Fabrizio Pastore, Oscar Cornejo	Delivered document.
ITT-1-9873-ESA-FAQAS-D2 Issue 2 Rev. 2	July 6th, 2020	Fabrizio Pastore, Oscar Cornejo	Revised document.
ITT-1-9873-ESA-FAQAS-D2 Issue 3 Rev. 3	July 8th, 2020	Fabrizio Pastore, Oscar Cornejo	Revised document after review meeting.
ITT-1-9873-ESA-FAQAS-D2 Issue 3 Rev. 1	Nov 4th, 2020	Fabrizio Pastore, Oscar Cornejo	Document delivered for TR3, Prototyping and tool chain review. With respect to previous version, the following sections had been updated: • Section 1.1.4 • Section 2.1.6 • Section 2.3.1.5 • Section about evaluation of code-driven mutation analysis (now moved to D4) • Section about case studies for LXS (now moved to D4) • Section about case studies for ASN (now moved to D4) Modified text is in blue.
ITT-1-9873-ESA-FAQAS-D2 Issue 4 Rev. 1	Dec 16th, 2020	Fabrizio Pastore, Oscar Cornejo	Updated empirical results for code-driven mutation testing concerning ESAIL.
ITT-1-9873-ESA-FAQAS-D2 Issue 5 Rev. 1	Sept 20th, 2021	Fabrizio Pastore, Oscar Cornejo	Updated Section **??** to include the finalized version of the data-driven mutation analysis approach. Added Section **??** to describe the code-driven test suite augmentation approach. Extended Section 2.4 to reflect our considerations on the fasibility of data-driven test suite augmentation. Modified text is in blue.
ITT-1-9873-ESA-FAQAS-D2 Issue 5 Rev. 2	Sept 22nd, 2021	Fabrizio Pastore, Oscar Cornejo	Updated Section **??** to include a revised version of the text, in line with what published in IEEE Transactions on Software Engineering [**?**].
ITT-1-9873-ESA-FAQAS-D2 Issue 5 Rev. 3	Oct 6th, 2021	Fabrizio Pastore, Oscar Cornejo	Addressing comments from ESA. In particular, we addressed typos in Figures 2.13 to 2.17.

Delivered Items

In the following Table we provide a list of deliverable items released with this document. Each Item is identified by its path on the Alfresco system.

Deliverable	Description
Review Meeting 2/Deliverables/ASN1CC-CaseStudy/Specifications/ACN-UM-v-3-2.pdf	User manual of ASN1 Compiler
Review Meeting 2/Deliverables/ASN1CC-CaseStudy/Specifications/taste-documentation-current.pdf	Taste software documentation including ASN1 overview
Review Meeting 2/Deliverables/ASN1CC-CaseStudy/Software/asn1.container.tar.gz	Mutation testing Singularity container corresponding to the ASN1CC case study.
Review Meeting 2/Deliverables/ASN1CC-CaseStudy/Software/ASN1-container-instructions.md	Instructions to deploy the ASN1CC singularity container.
Review Meeting 2/Deliverables/GSL-CaseStudies/Specifications/gs-man-nanosoft-ms100-command-and-management-sdk-3.6.2-1-g67fe6e1.pdf	GSL software specifications
Review Meeting 2/Deliverables/LXS-CaseStudies/Specifications	Folder with LXS specifications
Review Meeting 2/Deliverables/LXS-CaseStudies/Specifications/FAQAS-LXS-MAN-001_1- SVF Software Installation and User Manual.pdf	SVF and ESAIL Software Installation and User Manual
Review Meeting 2/Deliverables/LXS-CaseStudies/Specifications/ESAIL-LXS-ICD-P-0184_2A ADCS IF SW External ICD.docx	ADCS IF SW External ICD
Review Meeting 2/Deliverables/LXS-CaseStudies/Specifications/ESAIL-LXS-SDD-P-0105_1B On-board Application Software Design Document	On-board Application Software Design Document
Review Meeting 2/Deliverables/LXS-CaseStudies/Specifications/MOC-applicable MIB egos-mcs-s2k-icd-0001-version7.0-FINAL	SCOS-2000 Database Import ICD
Review Meeting 2/Deliverables/LXS-CaseStudies/Specifications/ocp.dat	SCOS-2000 Database file containing the specifications of the nominal value ranges for ESAIL ADCS parameters.
LXS-CaseStudies/Specifications/ESAIL-VirtualMachine	Folder containing the ESAIL virtual machine. The ESAIL virtual machine contains both unit and system test cases. The location and instructions required to execute ESAIL system test cases are provided in FAQAS-LXS-MAN-001_1 Section 7. Concerning unit test cases, they are implemented using the check framework [?]. Unit test cases are stored inside the *Test* folder of the component under test. For example, for the AdcsController, they are stored in *./ApplicationLayer/AdcsController/Test/*.
Review Meeting 2/Deliverables/MLFS-CaseStudy/Software/mlfs.container.tar.gz	Mutation testing Singularity container corresponding to the MLFS case study.
Review Meeting 2/Deliverables/MLFS-CaseStudy/Software/MLFS-container-instructions.md	Instructions to deploy the MLFS singularity container.
Review Meeting 2/Deliverables/MLFS-CaseStudy/Specifications/E1356-GTD-SUM-01_I1_R2.pdf	Installation and execution instructions of the MLFS.
Review Meeting 2/Deliverables/MLFS-CaseStudy/Specifications/E1356-CS-SUM-01_I1_R5.pdf	Installation and execution instructions of the MLFS test suite.
Review Meeting 2/Deliverables/SnT-Software/FAQAS-DataDrivenMutator-Buffers.zip	Preliminary implementation of the data-driven mutation testing component
Review Meeting 2/Deliverables/SnT-Software/FAQAS-DataDrivenMutator-ASN1.tar.gz	Preliminary implementation of the data-driven mutation testing component for ASN1.
Review Meeting 2/Deliverables/SnT-Software/FAQAS-CodeDriven-Mutation.tar.gz	Preliminary implementation of the code-driven mutation testing component
Review Meeting 3/Deliverables/SnT-ScriptsAndPrograms-ForCodeDrivenMutationTesting.zip	Updated scripts for code-driven mutation testing.
Review Meeting 3/Deliverables/SnT-ScriptsAndPrograms-ForCodeDrivenMutationTesting.zip	Updated scripts for code-driven mutation testing.
Review Meeting 5/Delivered Software/ITT-1-9873-ESA-FAQAS-MASS.zip	Zipped repository for the code-driven mutation analysis toolset (MASS).
Review Meeting 5/Delivered Software/ITT-1-9873-ESA-FAQAS-SEMuS.zip	Zipped repository for code-driven mutation testing toolset (SEMuS).
Review Meeting 5/Delivered Software/ITT-1-9873-ESA-FAQAS-DAMAt.zip	Zipped repository for data-driven mutation analysis toolset (DAMAt).

Contents

Introduction

This document is the final report of the ESA activity ITT-1-9873-ESA, which concerns the development of a framework for the automated assessment and the automated improvement of test suites for space software[1].

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 [**?**] 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 [**?**, **?**].

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 [**?**]). 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* [**?**]. 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 [**?**]. 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 [**?**] or redundant with each other [**?**]. 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 [**?**] [**?**] or solutions that prioritize and select the test cases to be executed for each mutant [**?**]. Equivalent and redundant mutants can be detected by comparing the code coverage of the original program and its mutants [**?**, **?**, **?**, **?**]. However, these approaches have not been evaluated on industrial, embedded sys-

[1] In this report, we use the term space software to indicate software to be deployed on hardware that runs on-orbit.

tems 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*** [**?**, **?**], 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 [**?**]. Unfortunately, well known, code-driven mutation operators (e.g., the sufficient set [**?**, **?**]) 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 following contributions:

- A survey of the literature on mutation analysis/testing, which is summarized in section 1.
- 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

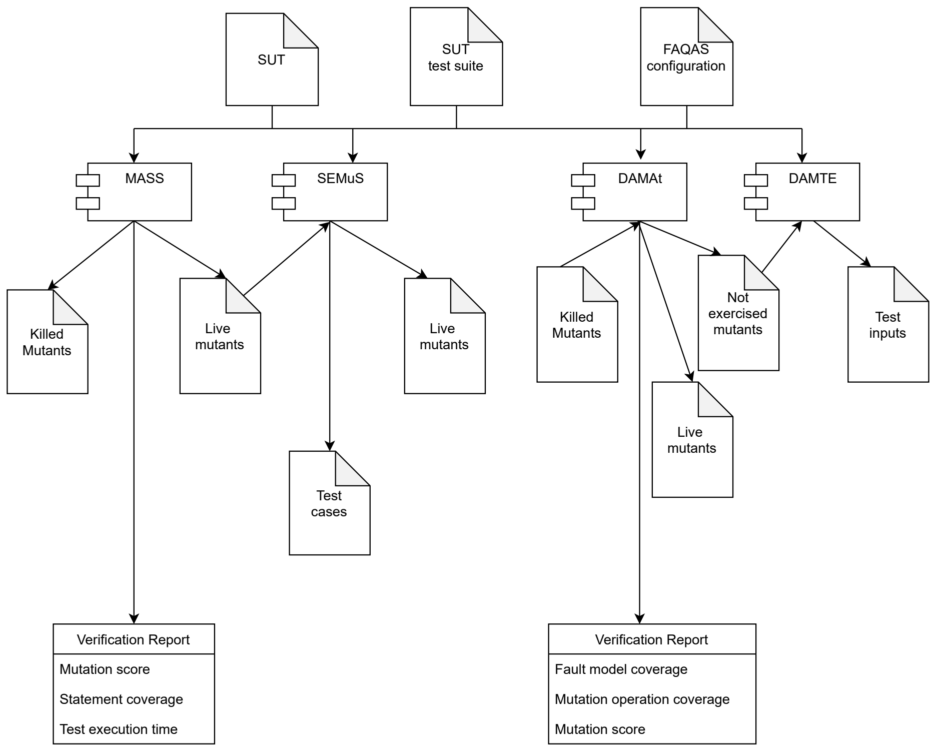


Figure 1: Overview of the FAQAS toolset

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.
- DAMTE, a tool-supported methodology for the generation of test inputs based on data-driven mutation analysis results.

• An extensive empirical evaluation demonstrating the feasibility, effectiveness, and scalability of the proposed approaches with space software.

• The definition of guidelines for the adoption of mutation analysis and testing strategies within ECSS activities. The proposed guidelines support both quality assurance activities described in ECSS standards and Independent Software Verification and Validation (ISVV) practices.

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 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.

Chapter 1

State-of-the-art

This chapter briefly discusses the applicability, in the context of space software, of existing solutions related to the four contributions of FAQAS: code-driven mutation analysis, code-driven mutation testing, data-driven mutation analysis, data-driven mutation testing.

1.1 Code-driven mutation analysis

Mutation analysis can drive the generation of test cases, which is referred to as ***mutation testing*** in the literature. A detailed overview of mutation testing and analysis solutions and optimizations can be found in recent surveys [?, ?].

1.1.1 Mutation Adequacy and Mutation Score computation

A mutant is said to be killed if at least one test case in the test suite fails when exercising the mutant. Mutants that do not lead to the failure of any test case are said to be live. Three conditions should hold for a test case to kill a mutant: *reachability* (i.e, the test case should execute the mutated statement), *necessity* (i.e., the test case should reach an incorrect intermediate state after executing the mutated statement), and *sufficiency* (i.e., the final state of the mutated program should differ from that of the original program) [?].

The mutation score, i.e., the percentage of killed mutants, is a quantitative measure of the quality of a test suite. Recent studies have shown that achieving a high mutation score improves significantly the fault detection capability of a test suite [?], a result which contrasts with that of structural coverage measures [?]. However, a very high mutation score (e.g., above 75%) is required to achieve a higher fault detection rate than the one obtained with other coverage criteria, such as statement and branch coverage [?]. In other words, there exists a strong association between a high mutation score and a high fault revelation capability for test suites.

The capability of a test case to kill a mutant also depends on the observability of the program state. To overcome the limitations due to observability, different strategies to identify killed mutants can be adopted; they are known as strong, weak, firm, and flexible mutation coverage [?]. With strong mutation, to kill a mutant, there shall be an observable difference between the outputs of the original and mutated programs. With weak mutation, the state (i.e., the valuations of the program variables in scope) of the mutant shall differ from the state of the original program, after the execution of the mutated statement [?]. With firm mutation, the state of the mutant shall differ from the state of the original program at execution points between the first execution of the mutated statement and the termination of the program [?]. Flexible mutation coverage consists of checking if the mutated code leads to object corruption [?]. For space software, we suggest to rely on strong mutation because it is the only criterion that truly assesses the fault detection capability of the test suite; indeed, it relies on a mutation score that reflects the percentage of mutants leading to test failures. With the other mutation coverage criteria, a mutant is killed if the state of the mutant after execution of the mutated statement differs from the one observed with the original code, without any guarantee that either the erroneous values in state variables will propagate or the test oracles will detect them.

1.1.2 Mutation Operators

Mutation analysis introduces small syntactical changes into the code (source code or machine code) of a program through a set of mutation operators that simulate programming mistakes.

The *sufficient set of operators* is widely used for conducting empirical evaluations [?, ?, ?, ?]. The original sufficient set, defined by Offutt et al., is composed of the following operators: Absolute Value Insertion (ABS), Arithmetic Operator Replacement (AOR), Integer Constraint Replacement (ICR), Logical Connector Replacement (LCR), Relational Operator Replacement (ROR), and Unary Operator Insertion (UOI) [?]. Andrews et al. [?] have also included the *statement deletion operator* (SDL) [?], which ensures that every pointer-manipulation and field-assignment statement is tested.

The sufficient set of operators enables an accurate estimation of the mutation score of a test suite [?]; furthermore, the mutation score computed with the sufficient set is a good estimate of the fault detection rate (i.e., the portion of real faults discovered) of a test suite [?, ?].

However, empirical work has shown that, to maximize the detection of real faults, a set of operators should be used in addition to the sufficient set: Conditional Operator Replacement (COR), Literal Value Replacement (LVR), and Arithmetic Operator Deletion (AOD) [?].

The SDL operator has inspired the definition of mutation operators (e.g., *OODL operators*) that delete portions of program statements, with the objective of replacing the sufficient set with a simpler set of mutation operators. The OODL mutation operators include the delete Arithmetic (AOD), Bitwise (BOD), Logical (LOD), Relational (ROD), and Shift (SOD) operators. Empirical results show that deletion operators produce significantly fewer equivalent mutants[1] [?, ?] and, furthermore, test suites that kill mutants generated with both SDL and OODL operators kill a very high percentage of all mutants (i.e., 97%) [?].

Another alternative to the sufficient set of operators is the generation of *higher order mutants*, which result from the application of multiple mutation operators for each mutation [?, ?, ?, ?]. However, higher order mutants are easier to kill than the first order ones (i.e., less effective to assess test suites limitations) [?, ?], and there is limited empirical evidence regarding which mutation operators should be combined to resemble real faults and minimize the number of redundant mutants [?].

1.1.3 Compile-time Scalability

The potentially large size of the software under test, combined with the large number of available mutation operators, may make the compilation of all mutants infeasible.

To reduce the number of invocations to the compiler to one, *mutant schemata* include all the mutations into a single executable [?]. With mutant schemata, the mutations to be tested are selected at run-time through configuration parameters. This may lead to a compilation speed-up of 300% [?].

Another solution to address compile-time scalability issues consists of *mutating machine code* (e.g., binary code [?], assembly language [?], Java bytecode [?], and .NET bytecode [?]), thus avoiding the execution of the compilation process after creating a mutant. A common solution consists of mutating the LLVM Intermediate Representation (IR) [?], which enables the development of mutants that work with multiple programming

[1] For example, statement deletion can lead to equivalent mutants only if statements are redundant, which is unlikely [?].

languages [?] and facilitates the integration of optimizations based on dynamic program analysis [?].

Unfortunately, the mutation of machine code may lead to mutants that are not representative of real faults (i.e., faults caused by human mistakes at development time) because they are impossible to generate from the source code [?]. For instance, a function invocation in the source code may lead to hundreds of machine code instructions (e.g., the function call *std::vector::push_back* leads to 200 LLVM IR instructions) and, consequently, some of the mutants derived from such instructions cannot be derived by mutating the source code. In the case of IR mutation, some of these impossible mutants can be automatically identified [?]; however, the number of generated mutants tend to be higher at the IR level than at the source code level, which may reduce scalability [?]. In addition, we have encountered three problems that prevented the application of mutation analysis tools based on LLVM IR to our case study systems. First, space software relies on compiler pipelines (e.g., RTEMS [?]) that include architecture-specific optimizations not supported by LLVM. Second, there is no guarantee that the executables generated by LLVM are equivalent to those produced by the original compiler. Third, efficient toolsets based on LLVM often perform mutations dynamically [?], which is infeasible when the software under test needs to be executed within a dedicated simulator, a common situation with space software and many other types of embedded software in cyber-physical systems.

1.1.4 Runtime Scalability

A straightforward mutation analysis process consists of executing the full test suite against every mutant; however, it may lead to scalability problems in the case of a large software under test (SUT) with expensive test executions. *Simple optimizations* that can be applied to space software consist of (S1) stopping the execution of the test suite when the mutant has been killed, (S2) executing only those test cases that cover the mutated statements [?], and (S3) rely on timeouts to automatically detect infinite loops introduced by mutation [?].

Split-stream execution consists of generating a modified version of the SUT that creates multiple processes (one for each mutant) only when the mutated code is reached [?, ?], thus saving time and resources. Unfortunately, it cannot be applied in the case of space software that needs to run with simulators because, in general, the hosting simulator cannot be forked by the hosted SUT.

Another feasible solution consists of *randomly selecting a subset of the generated mutants* [?, ?, ?]. Zhang et al. [?] empirically demonstrated that a random selection of 5% of the mutants is sufficient for estimating, with high confidence, the mutation score obtained with the complete mutants set. Further, they show that sampling mutants uniformly across different program elements (e.g., functions) leads to a more accurate mutation score prediction than sampling mutants globally in a random fashion. For large software systems that lead to thousands of mutants, random mutation analysis is the only viable solution. However, for very large systems such as the ones commonly found in industry, randomly selecting 5% of the mutants may still be too expensive.

Gopinath et al. estimate the number of mutants required for an accurate mutation score [?]. They rely on the intuition that, under the assumption of independence between mutants, mutation analysis can be seen as a Bernoulli experiment in which the outcome of the test for a single mutant is a Bernoulli trial (i.e., mutant successfully killed or not) and, consequently, the mutation score should follow a binomial distribution. They rely on Tchebysheff's inequality [?] to find a theoretical lower bound on the number of mutants required for an accurate mutation score. More precisely, they suggest that, with 1,000 mutants, the estimated mutation score differs from the real mutation score at most by 7 percentage points. However, empirical results show that the binomial distribution provides a conservative estimate of the population variance and, consequently, 1,000 mutants enable in practice a more accurate estimate ($> 97\%$) of the mutation score than expected.

In the statistics literature, the correlated binomial model [?], and related models [?, ?, ?] are used when Bernoulli trials are not independent [?]. In our work, based on the results achieved by Gopinath et al., we assume that the degree of correlation between mutants is limited and the binomial distribution can be used to accurately estimate the mutation score.

The statistics literature also provides a number of approaches for the computation of a sample size (i.e., the number of mutants, in our context) that enables estimates with a given degree of accuracy [?, ?, ?, ?]. For binomial distributions, the most recent work is that of Gonçalves et al. [?], that determines the sample size by relying on heuristics for the computation of confidence intervals for binomial proportions. A confidence interval has a probability p_c (the confidence level) of including the estimated parameter (e.g., the mutation score). Results show that the largest number of samples required to compute a 95% confidence interval is 1,568.

If used to drive the selection of mutants, both the approaches of Gopinath et al. and Gonçalves et al., which suggest sampling at least 1,000 mutants, may be impractical when mutants are tested with large system test suites.

An alternative to computing the sample size before performing an experiment is provided by sequential analysis approaches, which determine the sample size while conducting a statistical test [?]. Such approaches do not perform worst-case estimates and may thus lead to smaller sample sizes. For example, the sequential probability ratio test, which can be used to test hypotheses, has been used in mutation analysis as a condition to determine when to stop test case generation (i.e., when the mutation score is above a given threshold) [?]. In our context, we are interested in point estimation, not hypothesis testing; in this case, the sample size can be determined through a fixed-width sequential confidence interval (FSCI), i.e., by computing the confidence interval after every new sample and then stop sampling when the interval is within a desired bound [?, ?, ?]. Concerning the method used to compute the confidence interval in FSCI, the statistics literature [?] reports that the Wald method [?] minimizes the sample size but requires an accurate variance estimate. We will therefore resort to a non-parametric alternative, which is Clopper-Pearson [?]. Note that FSCI has never been applied to determine the number of mutants to consider in mutation analysis.

Other solutions to address *runtime scalability problems* in mutation analysis aim to *prioritize test cases* to maximize the likelihood of executing first those that kill the mutants [?, ?, ?]. The main goal is to save time by preventing the execution of a large subset of the test suite, for each mutant. Previous work aimed at prioritizing faster test cases [?] but this may not be adequate with system-level test suites whose test cases have similar, long execution times. Approaches that rely on data-flow analysis to identify and prioritize the test cases that likely satisfy the killing conditions [?] are prohibitively expensive and are unlikely to scale to large systems. Other work [?] combines three coverage criteria: (1) the number of times the mutated statement is exercised by the test case, (2) the proximity of the mutated statement to the end of the test case (closer ones have higher chances of satisfying the sufficiency condition) , and (3) the percentage of mutants belonging to the same class file of the mutated statement that were already killed by the test case. Criterion (3) is also used to reduce the test suite size, by only selecting the test cases above a given percentage threshold. Unfortunately, only criterion (1) seems applicable in our context; indeed, criterion (2) is ineffective with system test cases whose results are checked after long executions, while criterion (3) may be inaccurate when only a random, small subset of mutants is executed, as discussed above.

1.1.5 Detection of Equivalent Mutants

A mutant is equivalent to the original program when they both generate the same outputs for the same inputs. Although identifying equivalent mutants is an undecidable problem [?, ?], several heuristics have been developed to address it.

The simplest solution consists of relying on *trivial compiler optimisations* [?, ?, ?], i.e., compile both the mutants and the original program with compiler optimisations enabled and then determine whether their executables match. In C programs, compiler optimisations can reduce the total number of mutants by 28% [?].

Solutions that identify equivalent mutants based on *static program analysis* (e.g., concolic execution [?, ?] and bounded model checking [?]) show promising results (e.g., to automatically identify non-equivalent mutants for batch programs [?]) but they rely on static analysis solutions that cannot work with system-level test cases that execute with hardware and environment simulators in the loop. Indeed, (1) simulation results cannot be predicted by pure static analysis, (2) concolic execution tools, which rely on LLVM, cannot be run if the SUT executable should be generated with a specific compiler (see Section 1.1.3), (3) there are no solutions supporting the concolic execution of large software systems within simulation environments (state-of-the-art techniques work with small embedded software [**?**]), and (4) communication among components not based on direct method invocations (e.g., through network or databases) is not supported by existing toolsets.

Alternative solutions rely on *dynamic analysis* and compare data collected when testing the original software and the mutants [**?**, **?**, **?**, **?**]. The most extensive empirical study on the topic shows that nonequivalent mutants can be detected by counting the number of methods (excluding the mutated method) that, for at least one test case, either (1) have statements that are executed at a different frequency with the mutant, (2) generate at least one different return value, or (3) are invoked at a different frequency [**?**]. To determine if a mutant is non-equivalent, it is possible to define a threshold indicating the smallest number of methods with such characteristics. A threshold of one identifies non-equivalent mutants with an average precision above 70% and an average recall above 60%. This solution outperforms more sophisticated methods relying on dynamic invariants [**?**]. Also, coverage frequency alone leads to results close to the ones achieved by including all three criteria above [**?**]. However, such approaches require some tailoring because collecting all required data (i.e., coverage frequency for every program statement, return values of every method, frequency of invocation of every method) has a computational and memory cost that may break real-time constraints.

1.1.6 Detection of Redundant Mutants

Redundant mutants are either *duplicates*, i.e., mutants that are equivalent with each other but not equivalent to the original program, or *subsumed*, i.e., mutants that are not equivalent with each other but are killed by the same test cases.

Duplicate mutants can be detected by relying on the same approaches adopted for equivalent mutants.

According to Shin et al., subsumed mutants should not be discarded but analyzed to augment the test suite with additional test cases that fail with one mutant only [**?**]. The augmented test suite has a higher fault detection rate than a test suite that simply satisfies mutation coverage; however, with large software systems the approach becomes infeasible because of the lack of scalable test input generation approaches.

1.1.7 Benchmark of State-of-the-art, Code-driven Mutation Testing Toolsets

This section summarizes the outcome of an experiment performed to evaluate the applicability of state-of-the-art mutation testing tools in the space context, based on the case study systems of the project.

To carry out this preliminary evaluation of mutation testing tools, we selected a set of tools presented in the literature based on the following criteria:

- **Availability of source code.** To enable optimizations, the tool under analysis should be provided along with source code.
- **Applicability to C/C++ code.** The tool under analysis should be able to process C and C++ code.
- **Licence compatible with ESA Software Community Licence Permissive (ESA SCLP).** The licence of the tool under analysis, should enable redistributing the tool itself within the FAQAS framework, which is released under ESA SCLP.
- **Age.** To avoid problems due to support for recent libraries, we should prioritize tools that are recent and actively developed.

The first three criteria mentioned above constitute mandatory requirements. Tools not meeting these requirements are not selected for evaluation in our context because they cannot be integrated into the FAQAS framework.

Table 1.1: Summary of Data-Driven Mutation Testing Benchmarks.

Reference	**Approach/Tool Name**	**Evaluation**
Hariri & Shi 2018	SRCIRor	**Source code availability.** Yes, https://github.com/TestingResearchIllinois/srciror. **Applicability to C/C++ code.** Yes. **ESA SCLP Compatible.** Yes, released under NCSA, https://opensource.org/licenses/NCSA, which allows redistribution and re-licensing. **Age.** Aged, last update in September 2018. **Outcome. The tool is applicable in space context.**
Wang et al. 2017	Accmut	**Source code availability.** Yes, https://github.com/wangbo15/accmut/ **Applicability to C/C++ code.** Yes. **ESA SCLP Compatible.** Yes, released under NCSA, https://opensource.org/licenses/NCSA, which allows redistribution and re-licensing. **Age.** Aged, last update in January 2018. **Outcome. Depends on CLANG/LLVM, which prevents compilations for some sysems.**
Phan et al. 2018	MUSIC	**Source code availability.** Yes, https://github.com/swtv-kaist/MUSIC/ **Applicability to C/C++ code.** Yes. **ESA SCLP Compatible.** No. The software is licensed with proprietary licence. In private communication via e-mail, authors have shown to be available to relicensing, however this might not fit the budget of the project. **Age.** Recent, last update in July 2019.
Denisov & Pankevich 2018	Mull	**Source code availability.** Yes, https://github.com/mull-project/Mull **Applicability to C/C++ code.** Yes. **ESA SCLP Compatible.** Yes. Apache Licence 2.0, https://opensource.org/licenses/Apache-2.0. **Age.** Ongoing, last update in June 2020. **Outcome. The tool requires compilation with CLANG/LLVM, which leads to compilation errors with systems depending on RTEMS. Also, natively, Mull performs mutations on the fly through just-in-time compilation features, which is inapplicable if the SUT is executed within a simulator.**
Delgado et al. 2018	MuCPP	**Source code availability.** No, only executables are available https://ucase.uca.es/mucpp/
Jia & Harman 2008	Milu	**Source code availability.** Yes, https://github.com/yuejia/Milu/ **Applicability to C/C++ code.** Yes. **ESA SCLP Compatible.** Yes, released under NCSA licence, https://opensource.org/licenses/NCSA. **Age.** Aged, last update in April 2018. **Outcome. The tool generates a preprocessed source code that does not compile.**
Brannstrom et al. 2015	Dextool	**Source code availability.** Yes, https://github.com/joakim- brannstrom/dextool **Applicability to C/C++ code.** Yes. **ESA SCLP Compatible.** Yes, released under Mozilla public Licence 2.0, https://opensource.org/licenses/MPL-2.0. **Age.** Ongoing, last update in June 2020. **Outcome. Depends on CLANG/LLVM, which prevents compilations for some sysems.**
Delamaro et al. 2001	Proteum	**Source code availability.** Yes, https://github.com/magsilva/proteum. **Applicability to C/C++ code.** Yes. **Age.** Aged, last update December 2015.
Shariar and Zulkernine 2008	Function Calls Mutation	**Source code availability.** No.
Dans & Hierons 2001	Floating-point Mutation	**Source code availability.** No.

Table 1.1 provides the list of selected tools along with the evaluation results. We do not evaluate all the criteria when one of the mandatory requirements is not met. For what it concerns the compatibility with the ESA Software Community Licence Permissive, we consider the licenses NCSA and Apache Licence 2.0 compatible. Indeed, both the two licences allow for redistribution of the software, a condition that is sufficient to release a mutation testing tool as component of the FAQAS framework.

For our evaluation we then selected the five most recent tools that fulfill our mandatory requirements: SRCIRor, Mull, Dextool, Accmut, and Milu. Proteum has been discarded because its latest stable version dates back to December 2015; on May 2020 a few changes had been made on Proteum GitHub repository, however, the up to date version is indicated by its developer as not usable.

To evaluate the applicability of existing mutation testing tools to space software, we evaluated each mutation testing tool considered in our study against the same case study system of the project, i.e., the System Test Suite for ESAIL provided by LXS. We selected this case study system, because (1) it is the largest case study system of FAQAS in terms of lines of code, (2) the ESAIL system test suite requires that the full software is compiled and all the required libraries linked (this may complicate the use of tools that cannot parse all the source code), (3) the software under test (SUT) is executed within a system emulator (SVF) that requires the SUT to respect its real-time constraints. ESAIL consists of 924 source files (719 files with extension ".c" and 205 with extension ".h"). In total, it consists of 74,161 LOC. ESAIL is compiled with sparc-rtems4.8-gcc, a tailored version of the gcc compiler for sparc systems, the compiler is provided by Cobham Gaisler[2].

To draw a final outcome for or evaluation (see Table 1.1), we applied each selected tool to ESAIL and verified if the mutation testing tool could successfully create mutated version of ESAIL that can be compiled and executed within the SVF. Out of all the selected tools, only SRCIRor had been successfully applied to ESAIL.

1.1.8 Summary

We aim to rely on the sufficient set of operators since it has been successfully used to generate a mutation score that accurately estimates the fault detection rate for software written in C and C++, languages commonly used in embedded software. Further, since recent results have reported on the usefulness of both LVR and OODL operators to support the generation of test suites with high fault revealing power [**?**], the sufficient set may be extended to include these two operators as well.

To speed up mutation analysis by reducing the number of mutants, we should consider the SDL operator alone or in combination with the OODL operators. However, such heuristic should be carefully evaluated to determine the level of confidence we can expect.

Among compile time optimizations, only mutant schemata appear to be feasible with space software. Concerning scalability, simple optimizations (i.e., S1, S2, and S3 in Section 1.1.4) are feasible. Alternative solutions are the ones relying on mutant sampling and coverage metrics. However, to be applied in a safety or mission critical context, mutant sampling approaches should provide guarantees about the level of confidence one may expect. Currently, this can only be achieved with approaches requiring a large number of sampled mutants (e.g., $1,000$). Therefore, sequential analysis based on FSCI, which minimizes the number of samples and provides accuracy guarantees, appears to be the most appropriate solution in our context. Further, test suite selection and prioritization strategies based on code coverage require some tailoring to cope with real time constraints.

Equivalent mutants can be identified through trivial compiler optimizations and the analysis of coverage differences; however, it is necessary to define and evaluate appropriate coverage metrics. The same approach can be adopted to identify duplicate mutants. The generation of test cases that distinguish subsumed mutants is out of the scope of this work.

A high-level description of a possible mutation testing pipeline was proposed in a recent survey[3] [**?**]. It consists of the following sequence of activities: select (sample) mutants, compile mutants, remove equivalent and redundant mutants, generate test inputs that kill mutants, execute mutants, compute mutation score, reduce test suites and prioritize test cases. Unfortunately, such pipeline does not enable the integration of many optimizations proposed above, which further motivates our work. For example, it cannot support FSCI-based sampling, which requires mutants sampling to be coupled with mutants execution. Also, it does not envision the detection of equivalent and redundant mutants based on code coverage. Moreover, it only partially addresses scalability issues since test suite reduction and prioritization are performed after mutation analysis. Further, it includes a test input generation step that is not feasible in the context of CPS. Finally, it has never been implemented and therefore its feasibility has not been evaluated.

[2]https://www.gaisler.com/index.php/products/operating-systems/rtems

[3]The main objective of such pipeline was to walk the reader through the survey, not to propose a precise and feasible solution.

1.2 Data-driven Mutation Analysis

Data-driven mutation analysis evaluates the effectiveness of a test suite in detecting ***interoperability faults***. The CPS literature reports on four different interoperability types [**?**]: technical (which concerns communication protocols and infrastructure), syntactic (which concerns data format), semantic (which concerns the exchanged information, that is, errors in the processing of exchanged data), and cross-domain interoperability (which concerns interaction through business process languages such as BPEL [**?**]). P-1 For example, a technical interoperability problem may concern two components working with two different network protocols (e.g., TCP VS UDP), a syntactic interoperability problem is that of two components using different keywords to specify a field in a json [**?**] data file (e.g., ”temperature” and ”temp”), a semantic interoperability problem is that of a control software that does not take appropriate actions when the voltage of the board is above nominal range, cross domain interoperability is that of a web service not following the expected flow of remote calls. Technical and syntactic interoperability are provided by off-the-shelf hardware and libraries (not tested by CPS developers) while cross-domain interoperability concerns systems integrated in online services (e.g., energy plants) but is out of scope for the type of CPSs we target in this work, which are safety-critical CPSs like flight systems, robots, and automotive systems. In this paper, we thus focus on ***semantic interoperability*** faults, that is, faults that affect CPS components integration and are triggered (i.e., lead to failures) in the presence of specific subsets of the data that might be exchanged by CPS components. We thus aim to ensure that a test suite fails when the data exchanged by CPS components is not the one specified by test cases (e.g., through simulator configurations). Related work includes mutation analysis [**?**, **?**] and ***fault injection*** [**?**] techniques.

1.2.1 Mutation Analysis

Mutation analysis concerns the automated generation of faulty software versions (i.e., mutants) through automated procedures called mutation operators [**?**, **?**]. The effectiveness of a test suite is measured by computing the mutation score, which is the percentage of mutants leading to failures when exercised by the test suite.

Mutation operators introduce syntactical changes into the code of the SUT. The ***sufficient set of operators*** is implemented by most mutation analysis toolsets [**?**, **?**, **?**, **?**, **?**]. Unfortunately, these operators simulate faults concerning the implementation of algorithms (e.g., a wrong logical connector), which is usually tested in unit test suites that, by definition, do not exercise the communication among components, our target in this paper. Also, as stated in the Introduction, such operators can't be used to generate faulty data with simulated or off-the-shelf components. ***Higher-order*** mutation analysis [**?**], which simply combines multiple operators, has the same limitations.

Components integration is targeted by interface [**?**], integration [**?**], contract-based [**?**], and system-level mutation analysis [**?**]. The former three assess the quality of integration test suites by introducing changes that concern function invocations (e.g., switch function arguments) and inter-procedural data-flow (e.g., alter assignments to variables returned to other components); they can simulate integration faults in units integrated with API invocations but not interoperability problems concerning larger components communicating through channels (e.g., network). ***System-level mutation*** relies on operators for GUI components, which are out of our scope, and configuration files, by applying simple mutations, such as deleting a line of text, and are unlikely to lead to interoperability problems.

1.2.2 Fault injection

Fault injection techniques simulate the effect of faults by altering, at runtime, the data processed by the SUT [**?**]. Faults are introduced according to a fault model that describes the type of fault to inject, the timing of the injection, and the part of the system targeted by the injection. Different from data-driven mutation analysis, fault injection techniques aim to stress the robustness of the software, not assess the quality of its test suites.

Faults affecting components' communication, CPU, or memory can simulated by performing bit flips [**?**, **?**, **?**, **?**]. ***Communication faults*** are simulated also by duplicating or deleting packets, altering their sequence, or introducing incorrect identifiers, checksums, or counters [**?**, **?**]. Faults affecting ***signals*** can be simulated by shifting the signal or increasing the number of signal segments [**?**]. The largest set of faults affecting data exchanged through files or byte streams is simulated by Peach [**?**], which includes also protocol-specific fault injection procedures such as replacing host names with randomly generated ones. In general, although existing techniques may simulate a large set of faults they do not cover all the CPS interoperability faults (see Section 2.3.1.1).

Approaches performing fault injections other than bit flips require a model of the data to modify. The modelling formalisms adopted for this purpose are grammars [**?**, **?**, **?**, **?**], UML class diagrams [**?**, **?**], or block models [**?**, **?**]. Grammars are used to model textual data (e.g., XML), which is seldom exchanged by CPS components because of parsing cost. Block models enable specifying the representation to be used for consecutive blocks of bytes, which makes them applicable to a large set of systems; however, existing block model formalisms rely on the XML format, which is expensive to process and thus not usable with real-time systems [**?**, **?**]. The ***UML class diagram*** is a formalism that enables the specification of complex data structures and data dependencies [**?**, **?**]; however, it requires loading the data as UML class diagram instances, which is too expensive for real-time systems.

1.2.3 Summary

To summarize, the modification of the data exchanged by software components enables the simulation of communication and, therefore, semantic interoperability faults. Test suites can thus be assessed by relying on fault injection techniques to mutate data. However, existing fault injection techniques do not target mutation analysis; consequently, we lack methods for the specification of fault models and metrics for the assessment of test suites. Also, a larger set of procedures for the modification of data is needed. Finally, block models can effectively capture the structure of the data to modify but formalisms not relying on XML are needed. Our paper addresses such limitations.

1.3 Code-driven Mutation Testing

This section describes the approaches that can be adopted to automatically generate test cases that kill mutants. To kill a mutant we need test cases that (1) reach the mutation point (i.e., execute the mutated code), (2) cause corruptions in the program state right after the mutated code, and (3) manifest these corruptions into the program output (e.g., by producing an erroneous value in a state variable verified by a test assertion) thus leading to a failure [**?**]. These conditions are also known as the ***killing conditions*** of a mutant.

In the literature, there exist two groups of approaches for generating test cases that kill mutants: approaches based on constraint-programming, and approaches based on evolutionary computation. Below we introduces these two stream of approaches after introducing two well-known solutions for test generation driven by structural coverage, which are the basis for mutation testing solutions based on constraint programming.

1.3.1 Test Generation driven by program structure

Two alternative state-of-the-art solutions to generate test inputs that maximize structural coverage are CBMC, a bounded model checker, and KLEE [**?**], a symbolic execution engine. They are detailed below.

CBMC is an approach that implements ***Bounded Model Checking*** [**?**, **?**] (BMC), an approach for purely static software verification. The idea in BMC is to represent the software together with the properties to be verified as an instance of the propositional satisfiability problem (SAT). Such a representation captures the software behavior exactly, assuming that all the loop bodies in the software are repeated at most a fixed number of times. This approach has several advantages: the logical formulation is usually very compact compared to traditional model checking, where verification is reduced to a reachability problem in a graph representing the program state space; there are several high-performance SAT solvers [**?**, **?**] that can be used for solving the instances; and the satisfying assignments of an instance can be directly translated to meaningful counterexamples for correctness in the form of fault-inducing executions. Furthermore, it is widely recognized that BMC based approaches are particularly good at quickly finding short counterexamples when they exist.

KLEE [**?**] is an open source tool that implements ***Concolic Execution***, a technique that performs ***symbolic execution*** along a concrete execution path. KLEE was designed to automatically generate test cases that achieve high code coverage. The tool is implemented as a modified LLVM virtual machine that targets LLVM bytecode programs. KLEE also provides a symbolic POSIX library that enable analysis of programs that uses the system's environment. For example, KLEE can be executed with the flag `-sym-stdin N` which will make stdin symbolic with size `N`.

In FAQAS, we rely on KLEE to automatically produce the inputs that make the mutated version of the program generate a different output than the original version.

1.3.2 Test Generation based on Constraint Programming

Techniques based on constraint programming use some form of automated reasoning (e.g., Propositional Satisfiability or Constraint Solving [**?**]) to derive data that satisfy all the conditions necessary to kill a mutant [**?**].Existing approaches, differ for the strategy adopted to automatically generate these constraints from the program under test.

Offutt et al. [**?**], for example, automatically derive such constraints from the program by extracting the predicate expressions on the program's control flow graph. Then, such constraints are encoded to form a constraint system. In their approach, they propose three strategies for identifying infeasible constraint systems, the contradictions to such systems are the new test cases for the program under analysis.

```
 1  int midval (int x, int y, int z) {
 2    int midval;
 3
 4    midval = z;
 5    if (y < z) {
 6      if (x < y) {
 7        midval = y;
 8      }
 9      else if (x < z)
10   Δ else if (x <= z) {
11        midval = x;
12      }
13    }
14    else
15      if (x > y) {
16        midval = y;
17      } else if (x > z) {
18        midval = x;
19      }
20    return midval;
21  }
```

Listing 1.1: midval function returns the mid value between three integers.

In the following, we introduce an example of the application of Offutt's [?] approach by using the `midval` function presented in Listing 1.1. This function has a mutation on line 9, which has been mutated into line 10. According to their approach, the three killing conditions would be the following:

- Reachability $C_R : (y < z) \wedge (x \geq y)$
- Necessity $C_N : (x < z) \neq (x \leq z)$
- Sufficiency C_S : Output(P) $\neq$ Output(M)

C_R defines the condition required to reach the mutated statement, in this case the conjunction between the predicate of the first if condition and the negation of the second if condition. C_N defines the condition required to assure a different program state between the original and mutated version of the program right after the mutation point. Finally, C_S defines the condition necessary to demonstrate that both the original and the mutated program returns different values.

Holling et al. [?] proposed to use a ***symbolic execution*** approach to identify new test cases. Their idea is to first execute symbolically both the original and the mutated function, then to check if their return values are equivalent or not. Symbolic execution determines what inputs cause each part of a function to be covered during execution. To symbolically execute a function, it is necessary to replace the original inputs (i.e., concrete values) with symbolic ones. The ***symbolic values*** represent a set of possible concrete values that lead to a certain program path (i.e., path condition).

Holling's approach [?] to automatically identify equivalent mutants relies on the observation that two mutants are equivalent when there are no concrete values making the mutated function produce an output that is different from the one of the original function. If a value that makes the two functions generate distinct results can be found, the mutant is non-equivalent. To automate the generation of inputs, Holling et al. rely on KLEE [?].

We introduce an example of Holling's approach in Listing 1.2. The top part of Listing 1.2 shows the function `isPositive`, which checks if an integer number is positive or not. The bottom part of Listing 1.2 presents the mutated version of `isPositive`, where the relational operator $\geq$ has been replaced by the operator $>$. To automate the generation of inputs using KLEE, all the parameters need to be treated as symbolic values. This is achieved by function `make_symbolic` (see Listing 1.3) which converts concrete variables to symbolic ones by considering their memory address and size. In Listing 1.3, the parameter `numSymbolic` is made symbolic in Line 3. Then, the original and mutated functions are called using the symbolic arguments in Lines 5 and 6. Finally, we need to introduce an assertion that makes the symbolic execution engine ***look for inputs that make the output of the two functions different***. In Listing 1.3, this is achieved with an assertion that verifies that the return values of the two functions the same (see Line 8). Despite being counter-intuitive, this approach is effective because symbolic execution engines aim to identify inputs that falsify the assertions in the program. When the equality is falsified, then the two functions can produce a different output for a same input. The input that falsifies the equality can thus be used to improve the test suite enabling it to kill the mutant. In the example of Listing 1.3, KLEE will indicate that the return values of the original and mutated function differ when `num` is equal to zero. A new test case exercising function `isPositive` with `num=0` should thus be added to the test suite in order to kill the mutant.

```c
int isPositive(int num){
  if (num >= 0){
    return 1;
  } else {
    return 0;
  }
}

int MUT_isPositive(int num){
  if (num > 0){
    return 1;
  } else {
    return 0;
  }
}
```

Listing 1.2: isPositive and MUT_isPositive functions

```c
void generate_test_input () {
  int numSymbolic;
  make_symbolic(&numSymbolic, sizeof(numSymbolic), "numSymbolic");

  int original_ret = isPositive(numSymbolic);
  int transformed_ret = MUT_isPositive(numSymbolic);

  assert(original_ret == transformed_ret);
}
```

Listing 1.3: Holling's approach for test case generation.

```c
void generate_test_input () {
  int numSymbolic;
  make_symbolic(&numSymbolic, sizeof(numSymbolic), "numSymbolic");

  int original_ret = isPositive(numSymbolic);
  int transformed_ret = MUT_isPositive(numSymbolic);

  assert(original_ret != transformed_ret);
}
```

Listing 1.4: Test case generation with assertion that reflects the desired behaviour.

Similarly to Holling's approach, Riener et al. [?] proposed to use ***bounded model checking*** techniques to search for these counter examples. In their bounded model checking approach, the original program and the mutant are unrolled with respect to a certain maximum bound. In program unrolling, loops are re-written as a repeated sequence of similar independent statements. Then, both unrolled programs are encoded into a logic formula over the same input variables. To ensure that the mutation affects the output of the mutant, a propagation condition is encoded and added to the previous logic formula, the condition asserts that there exist at least one pair of different outputs under the assumption of equal inputs. In the last step, the formula is processed by a SMT-solver, if the solver finds a satisfying assignment, the inputs of the formula are translated into a new test case for the current program under analysis.

SEMu [?] is a recent mutantion testing framework based on dynamic symbolic execution that has been built on top of the KLEE Symbolic Virtual Machine [?]. SEMu uses a form of ***differential symbolic execution*** [?] to generate test inputs that kill mutants. The approach consists of modeling the mutant killing problem as a symbolic execution search in a scalable and cost-effective way. The SEMu framework is the building block of the test generation tool developed in FAQAS (i.e., ***SEMuS***). Different from the approaches presented above, SEMu can generate test inputs that kill mutants without the need of human intervention (e.g., to ad assertions); however, it targets only bash programs, it cannot generate unit test cases, for example.

In SEMu, the multiple program versions (i.e., the original and the mutated programs) are compiled within the same executable. Because mutants differ only slightly from the original program (minor syntactic differences), the technique performs one symbolic execution search for the unchanged source code, and then it forks the symbolic execution search every time it reaches a mutated statement.

The technique follows both the mutated and the original execution and compares the state (i.e., the value of the variables) at each step of the execution. More precisely, every time the execution reaches the statement after the mutation, the technique triggers the generation of test inputs for the forked and the original programs by comparing their symbolic states. To generate test inputs that kill the mutant, the technique encodes the three killings conditions, obtained from the symbolic execution search, into a single formula (the $kill$ formula) to be passed to a constraint solver.

To speed-up test generation, SEMu integrates a number of optimizations:

- ***Meta-mutation***: all mutants are encoded into a single program called meta-mutant (in FAQAS, we extended it at source code level), the mutant is selected through a branching statement named mutant choice statement.
- **Discarding non-infected mutant paths**: mutant paths failing to infect the program state are discarded immediately.
- **Heuristic search**: stop exploration of a path after K transitions, and solve the kill formula.
- **Infection-only strategy**: Generates test inputs by aiming only at mutant infection.

1.3.3 Test Generation based on Evolutionary Computation

Test generation approaches based on ***evolutionary computation*** typically rely on population-based meta-heuristic optimization algorithms [**?**]. They search for program inputs that could kill mutants under the guidance of a fitness function [**?**]. The main research contribution of these methods is the definition of fitness functions that capture the killing conditions of a mutant and identify test inputs that satisfy those conditions.

The ***fitness function*** captures the killing conditions of a mutant. For instance, Ayari et al. [**?**] proposed to use an evolutionary approach based on ***ant colony optimization*** (ACO) for automatic test input data generation on mutation testing. The ACO is an optimization algorithm inspired by the behavior of ants, it is based on the ants ability to find the shortest path between their nest and the food source. In the study by Ayari et al. [**?**], the approach takes an existing test case and produces a new test case by slightly modifying its inputs. The fitness function measures the distance between the mutated statement, and the statement reached by the new test case (e.g., the reachability condition). More precisely, the distance is defined as the number of basic blocks between the two statements in the program's control flow graph. Papadakis et al. [**?**], instead, rely on fitness functions that capture the distance between mutated statement and the statement covering the branches of the different mutations (e.g., the necessity condition).

Fraser and Arcuri [**?**] propose to use distinct distance metrics tailored to the specific operator used to generate the mutants. This tailoring is needed because the ***necessity killing condition*** relies on changes in the program state and the execution of a mutated statement does not guarantee that the program state had been changed (i.e., values on the stack are different at the mutation point). For example, the *deletion operator*, which removes a statement, may or may not change the program state, depending on the semantic of the removed sentence (e.g., a logging instruction does not alter the program state). In case the mutation effectively changes the program state the distance is set to 0, otherwise the given value is 1. In the case of the *insert unary operator*, which adds or subtracts 1 to a numerical value, the operator always change the program state, so the distance is set to 0 when the statement is reached.

Instead, in the case of the *replace variable operator*, which replaces a specific variable with all other variables of the same type in the program scope, the distance is set to 0 only if the values of the variables being exchanged are different before executing the statement, otherwise it is set to 1.

1.3.3.1 Generation of test oracles

The automated generation of test oracles is a research topic that goes beyond the specific needs of mutation testing [?, ?].

Fraser et al. provide an overview of existing approaches for the automated generation of test oracles that have been integrated into existing test case generation tools [?]. A common solution consists of the automated synthesis of assertions for the test case. These assertions reflect the output generated by the function under test when it is exercised with the automatically generated input. For example, Line 6 in Listing 1.5 shows the oracle that can be automatically generated for the function analyzed in Listing 1.3 (i.e., function *isPositive*). The oracle, in this case, consists of an assertion verifying that the value generated by function *isPositive* matches the value '1', which is the value observed during test generation for the input value '0'. This is the approach implemented by Riener et al. [?], who generate assertions that verify variables that present different values between the original and the mutated executions.

```
1 void test () {
2   int numSymbolic = 0;
3 
4   int ret = isPositive(numSymbolic);
5 
6   assert( ret == 1 );
7 }
```

Listing 1.5: Automatically generated oracle for function 'isPositive'.

Randoop [?] allows annotation of the source code to identify observer methods to be used for assertion generation. Orstra [?] generates assertions based on observed return values and object states. DiffGen [?] extends the Orstra approach to generate assertions from runs on two different program versions.

A well known limitation of automatically generated assertions that reflect the actual values observed during execution is that they need to be validated. More precisely, we need to ensure that the values expected by the assertions do not reflect a failure triggered by the test case (e.g, an erroneous value being returned). Such validation activity is typically performed manually by the engineers because it should be based on domain knowledge and system specifications. Specifications are generally written in natural language because, to reduce development costs, only few components of the system are specified using formal languages. For this reasons the automated verification of such assertions is infeasible.

Approaches that support engineers in the analysis of generated oracles exist and might be considered to speed up the process [?, ?]. For example, Staats et al. [?] identify the subset of variables to be verified by oracles in order to maximize the fault finding potential of the testing process. First, they generate a collection of mutants from the SUT. Second, the test suite (automatically generated) is run against the mutants using the original system as the oracle. Third, they select the variables to verify in test oracles by focussing on those variables that show different values in the original and the mutated version.

ZoomIn. [?] automatically identifies suspicious assertions. These are assertions that verify data that is generated by functions showing anomalies during test cases executions. Anomalies are detected by automatically deriving pre- and post-conditions of the functions of the SUT based on the data recorded during the execution of a manually implemented test suite. Anomalies consist of function executions that violate such pre- and post-conditions.

1.3.4 Summary

Among all the approaches for test input generation, SEMu is the most advanced one. However, it does not integrate solutions to automatically generate oracles. Solutions for generating oracles are at their infancy and cannot be considered ready for integration into the FAQAS toolset.

1.4 Data-driven Mutation Testing

Although the literature does not address the problem of automatically generating test cases for data mutation testing, in this section we provide references to work that can be reused for this purpose. We discuss both the generation of ***test inputs*** and ***test oracles***. Concerning the generation of test inputs, we group the applicable approaches according to the type of models used to specify the data to be generated: ***UML models***, ***grammars***, ***block models***, and ***no models***.

Automated test inputs generation aims to automatically generate data that can be altered through a mutation operator. In the context of system and integration testing, which is the target of data-driven mutation testing, test input data is provided through input interfaces. However, since data-driven mutation is applied to data exchanged different communication layers, including input interfaces and interfaces used for the communication among internal components, we may observe two possible situations. First, the data targeted by mutation testing coincide with the input data (i.e., it was not transformed internally). Second, the data targeted by data mutation is the result of a transformation of the input data. We thus need to consider both the two cases when discussing related work.

1.4.1 UML models

When the data exchanged on the communication layer targeted by mutation testing coincide with the input data, existing approaches based on constraint-solvers can be adopted. These approaches rely on a formal or semi-formal specification of the structure of the input data and the constraints among data fields. Some techniques target the generation of inputs whose data structure had been specified using a UML class diagram where the relations among data fields have been captured using OCL constraints. The data structure model is a UML class diagram and resembles the one reported in Figure **??**. The OCL language is instead used to capture all the constraints among data fields. Existing techniques in this category work by generating class diagram instances that satisfy a set of given OCL constraints by executing appropriate constraint solvers after having transformed the OCL constraints into other formalisms such as ***Alloy models*** [**?**], ***constraint satisfaction*** [**?**], ***SMT*** [**?**], or ***SAT*** problems [**?**].

Other approaches, instead, work with models specified in formats other than UML class diagrams: ***Java classes*** [**?**, **?**], ***constraint logic*** [**?**], ***Alloy*** [**?**], or ***Z specifications*** [**?**]. These techniques have been proven to be effective for testing software systems that process classical data structures like trees. Alloy is a modelling language for expressing complex structural constraints [**?**], which has been successfully used to generate test inputs for testing object-oriented programs [**?**]. Korat, instead, is a technique that enables the generation of data structures to test Java programs [**?**]. Given a bound to the input structures (i.e., the maximum number of instances for each class to be used), Korat exhaustively generates all the nonisomorphic structures that are valid. Some of the limitations of Korat include requiring the definition of an imperative predicate that evaluates the correctness of the generated structure, which could be complex in the case of complex data models, requiring the manual definition of an input bound for each non primitive attribute or association, which might be particularly expensive in case of complex data structure, not dealing with constraints defined over integers. A more efficient, black-box test generation approach is UDITA [**?**]. What contributes to the efficiency of UDITA is the combination of both generator methods and predicates. Generator methods are used to build instances of the data structure, while predicates are used to validate the generated instances. UDITA relies upon the Java Path Finder model checker [**?**] to generate all the instances that satisfy the given predicates. However, the implementation of these generator methods that define the complete structure of a complex data model instance and lead to realistic test inputs can be quite expensive.

A common limitation of solutions based on constraint-solvers is their scalability [**?**]. In the context of mutation testing, where existing test inputs are available, a possible solution to address the scalability problem may consist of generating new test inputs by ***regenerating only portions of existing test inputs***. For example, Di Nardo et al. [**?**] automatically generate test inputs for new requirements by adapting existing field data. This is achieved by combining model transformations with constraint solving. Despite empirical results show a huge performance gain with respect to traditional constraint-based approaches, the need for dedicated parsers to translate existing test inputs into class diagram instances may limit the applicability of the approach. Also, the available test inputs may not enable the generation of all the inputs data needed.

Other approaches address the scalability problem by relying on ***hybrid input generation approaches*** [**?**]. For example, PLEDGE [**?**] combines metaheuristic search and Satisfiability Modulo Theories (SMT [**?**]) to generate UML instance models from UML class diagrams annotated with OCL constraints. It works by using the Negation Normal Form (NNF [**?**]) to represent all the constraints derived from the UML data model. Different subformulas that build the NNF formula are then solved by combining metaheuristic search and SMT. Metaheuristic search is used to handle subformulas whose satisfaction involves structural tweaks to the instance model, i.e., additions and deletions of objects and links. SMT is used with subformulas involving only primitive attributes, i.e., attributes with primitive types.

When the data exchanged on the communication layer targeted by mutation is the result of a transformation of the input data, the only applicable solution consist in the application of approaches that generate inputs from scratch. By generating a large number of inputs that include instances for all the classes of the data model, these approaches may, in principle, lead to the generation of required data by internal components. However, without means to drive the generation of such data, the application of UML-based test input generation approaches in these situation is likely to be inefficient.

1.4.2 Grammars

When grammars are used to model the input, ***grammar-based test input generation*** approaches relying on the expansion of the production rules of the grammar can be adopted [**?**]. Available tools are shown in Table 1.2.

Table 1.2: List of tools for grammar-based inputs generation.

Name	**Grammar**	**Licence**	**Description**
GramTest [**?**]	BNF	Apache 2.0	Java-based tool that allows you to generate test cases based on BNF grammars. It covers all the production rules of the grammar.
Fuzzingbook Grammar Coverage-based Fuzzer [**?**]	BNF	MIT	Python tool that implements production rules coverage. It implements the Shortest Path Selection [**?**] optimization.
GP [**?**, **?**]	Annotated BNF	Proprietary	Tool for the automated generation of inputs from Stochastic Context Free Grammars. Implemented on top of genetic programming algorithms. https://selab.fbk.eu/kifetew/downloads/gplib-607.jar
RIDDLE [**?**]	BNF	Proprietary	Tool that adopts a grammar to describe the format of inputs; based on the grammar, random and boundary values are generated for tokens representing input parameters.
pFuzzer [**?**]	BNF	Tool that aims at producing valid inputs for input parsers `https://github.com/uds-se/pFuzzer`.	

Among existing approaches, ***Parser-Directed Fuzzing*** (hereafter, *pFuzzer*) aims at producing valid inputs for input parsers [**?**]. The challenge is to cover all the lexical and syntactical features of a certain language. The approach systematically produces inputs for the parser and tracks all the comparisons made; after every rejection, it satisfies the comparisons leading to rejections, effectively covering the input space. Evaluated on five subjects, from CSV files to JavaScript, the *pFuzzer* prototype covers more tokens than both lexical-based (AFL) and constraint-based approaches (KLEE).

Similarly to the case of UML-based test case generation approaches that generate inputs from scratch, these grammar-based approaches can be adopted when the communication layer targeted by mutation testing is either the input layer or an internal layer.However, they may suffer of inefficiency problems; in additions, grammar can be unlikely used to model the types of data processed by space software.

1.4.3 Block-models

Among the existing toolsets based on block-models, ***Peach*** is the only one which supports the automated generation of test data. However, it's implementation simply generates random data, the process is referred to as ***blind fuzzing***[4].

1.4.4 No models

The generation of test inputs without the need of data model is typically driven by code coverage. A representative solution is given by AFL, which has been introduced in Section **??**.

More in general, approaches that address the problem of automatically testing programs that process structured data can be adopted for this purpose [**?**, **?**, **?**]. For example, SUSHI [**?**] is a tool that aims to cover test objectives that depend on non-trivial data structure instances. It relies on symbolic execution to generate path conditions that capture the relationship between program paths and input data structures. The path conditions are then translated into fitness functions to enable testing based on meta-heuristic search. A solution for the search problem is a sequence of method invocations that instantiates the structured inputs to exercise the program paths identified by the path condition.

1.4.5 Automated Generation of Test Oracles

In the case of data-driven mutation testing, solutions for the ***automated generation of oracles*** that consist of assertions verifying the output of the software under test (see Section 1.3.3.1) might still be applied. However, considering that data-driven mutation testing is more likely adopted in the context of system-level testing, where inputs consist of complex, structured data, the generation of test oracles using techniques built for unit testing is likely infeasible in this context.

Possible solutions may consist of approaches that verify the correctness of the log files generated during the execution of the program [**?**, **?**]. Given a set of log files (or execution traces) generated during valid executions, these approaches can derive ***finite state automata*** (FSAs) that capture the sequences of events and data-flow observed in valid executions. The derived FSAs can be used to verify if new executions match the inferred models. More precisely, they enable the automated detection of invalid sequences of events and data-flows. Despite none of these approaches had been applied in the context of data-driven mutation testing, traces collected from multiple runs of a same test case might be used to derive an FSA that captures the behaviour of a single test case. By relying on multiple traces collected from a same test case we can leverage the generalization power of the inference engines used to generate the FSAs; these inference engines, for example, can automatically recognize and filter out variable elements such as timestamps. The inferred FSAs could thus then be used as oracles for newer executions of the generated test cases. Such approaches have shown to be effective in detecting both functional failures [**?**] and performance problems [**?**].

[4] https://wiki.mozilla.org/Security/Fuzzing/Peach#Creating_a_Data_Model

1.4.6 Evaluation of Data-driven Mutation Testing Toolsets

This section describes an evaluation we conducted to identify a data-driven mutation testing tool applicable to space context. In particular, we assessed the ***Peach Fuzzer toolset***.

Operator Name	Description
ArrayVarianceMutator	Change the length of arrays. Given L the original length of the array, the length is changed in range L-N to L+N.
ArrayReverseOrderMutator	Reverse the order of an array.
ArrayRandomizeOrderMutator	Put array elements in random order.
DWORDSliderMutator	Slides a DWORD through the blob.
BitFlipperMutator	Flips a given % of bits in blob. Default is 20%.
BlobMutator	Randomly grows a Blob block or shrinks it.
DataTreeRemoveMutator	Remove nodes from data tree.
DataTreeDuplicateMutator	Duplicate a node's value starting at 2x through 50x.
DataTreeSwapNearNodesMutator	Swap the data of two nodes that are near each other in the data model.
NumericalVarianceMutator	Produce numbers that are defaultValue - N to defaultValue + N.
NumericalEdgeCaseMutator	Replace with random numbers of appropriate correct size.
FiniteRandomNumbersMutator	Produce a finite number of random numbers for each *Number* element.
NumericalEvenDistributionMutator	Generate numbers evenly distributed through the total numerical space of the number range.
NullMutator	Does nothing, just test the data produced by the fuzzer.
PathValidationMutator	Does not mutate. Used to trace path of each test for path validation.
SizedVarianceMutator	Change the length of sizes to count - N to count + N.
SizedNumericalEdgeCasesMutator	Change the length of sizes to numerical edge cases.
SizedDataVarianceMutator	Change the length of sized data to count - N to count + N. Size indicator will stay the same.
SizedDataNumericalEdgeCasesMutator	Change the length of sizes to numerical edge cases.
StringCaseMutator	Change the case of a string.
UnicodeStringsMutator	Generate unicode strings.
ValidValuesMutator	Replace with random values other than the legal ones.
UnicodeBomMutator	Injects BOM markers into default value and longer strings.
UnicodeBadUtf8Mutator	Generate bad UTF-8 strings.
UnicodeUtf8ThreeCharMutator	Generate long UTF-8 three byte strings.
StringMutator	Generate a random unicode string, for each string node, one Node at a time.
XmlW3CMutator	Replace XML trees with invalid, non-well former, and valid (but random) XML trees.
PathMutator	Replace a path with an erroneous path generated according to 20 different rules.
HostnameMutator	Replace a hostname with an erroneous hostname generated according to 20 different rules.
IpAddressMutator	Replace an IP address with an erroneous IP address generated according to 20 different rules.
TimeMutator	Replace a time value with an erroneous value generated according to 3 different rules.
DateMutator	Replace a date with 60 predefined erroneous dates.
FilenameMutator	Replace a file name with an file name generated according to 10 different rules.
ArrayNumericalEdgeCasesMutator	This operator is not well documented in the source code of Peach.
BlobSpread	This operator is not well documented in the source code of Peach.

Table 1.3: Mutation Operators for the opensource version of Peach [**?**]

```
<Number name="lfh_CompSize" size="32" endian="little" signed="false"/>
<Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
<Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
    <Relation type="size" of="lfh_FileName"/>
</Number>
<Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
    <Relation type="size" of="lfh_FldName"/>
</Number>
<String name="lfh_FileName"/>
<String name="lfh_ExtraField"/>
```

Listing 1.6: Portion of a Peach data model.

Peach [**?**, **?**] is a fuzzing tool that relies on ***block models*** [**?**, **?**] to perform data mutations. In other words, Peach performs mutations by altering the data of an input according to a large, predefined set of rules. For example, Listing 1.6 introduces a portion of a data model describing the properties of the Zip data format [**?**].

Even though Peach is currently a proprietary software [**?**], the Mozilla Foundation maintains a community edition of the toolset [**?**], the community edition implements basic features such as the fuzzing capabilities. The proprietary version of Peach instead provides features for automatic generation of test cases and detailed reports about the potential security threats of a software [**?**]. The version we evaluated in this activity was the community edition provided by the Mozilla Foundation. We provide an overview of the mutation operators implemented by the Peach community edition in Table 1.3.

In the assessment of Peach, we defined three criteria to understand its applicability to the space context software. The first criteria concerns assessing if the community edition of Peach does work and if it can be installed properly. The second criteria concerns its portability. Finally, the third criteria concerns assessing its compatibility with FAQAS case study systems.

Regarding the first criteria, we tested Peach by applying it to the `unzip` program, and zip file mutants with the fuzzing capabilities of Peach. For this objective we reproduced the steps indicated in [**?**]. So, first, we generated a Peach Data Model for Zip data files, and then we specified a launcher that enables the complete mutation process.

Peach provides a monitoring infrastructure that enables the execution of the whole mutation process. The process consists of the following steps:

1. Specifying the data model for the a data type into an XML file.
2. Loading the data model into Peach.
3. Generating a new mutant (i.e, a mutated input).
4. Running the program taking as an input the generated mutant, and with the monitoring infrastructure enabled.
5. If the program crashes the process is stopped.
6. If the program does not crashes, the process goes back to step 3, performing a new mutation.

In particular, we were able to generate mutants for the Zip data format, but we could not run the monitoring infrastructure since it had dependencies with graphical environments that prevent us to execute it properly.

Regarding the second criteria, and specifically its portability. We seek to integrate it into the case studies as a component of their software to mutate data once it is sent, basically the idea would be to intercept the methods that exchange data, and apply Peach directly to the variable containing the data. During the evaluation, we discovered that Peach -mainly implemented in Python- can be used as a Python library, and that this library can be invoked to generate multiple mutants in a off-line mode.

Regarding the third criteria and its compatibility with our case studies, we conclude that its integration with embedded systems is unlikely to work, mainly because of the characteristics of our case study systems. For example, the ESAIL case study system runs within a real-time operative system (i.e., RTEMS by Edisoft) that does not possess a filesystem. Therefore, integrating Peach into ESAIL is not feasible because it might affect the real-time performance of the application, and also because it will be necessary to implement a solution to port the Peach toolset into the ESAIL infrastructure.

Chapter 2

The FAQAS approach

This chapter presents the theory and the methodology behind the toolset implemented by FAQAS.

2.1 Code-driven Mutation Analysis: MASS

2.1.1 Overview

Figure 2.1 provides an overview of the mutation analysis process that we propose, *Mutation Analysis for Space Software (*MASS*)*. 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 *MASS* is to assess the effectiveness of test suites with respect to detecting violations of functional requirements.

Different from the mutation analysis pipeline presented in related work [**?**], *MASS* enables the integration of all mutation analysis optimization techniques that are feasible in our context to address scalability and pertinence problems (see Section 1.1.8). *MASS* 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, *MASS* enables FSCI-based sampling by iterating between 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.

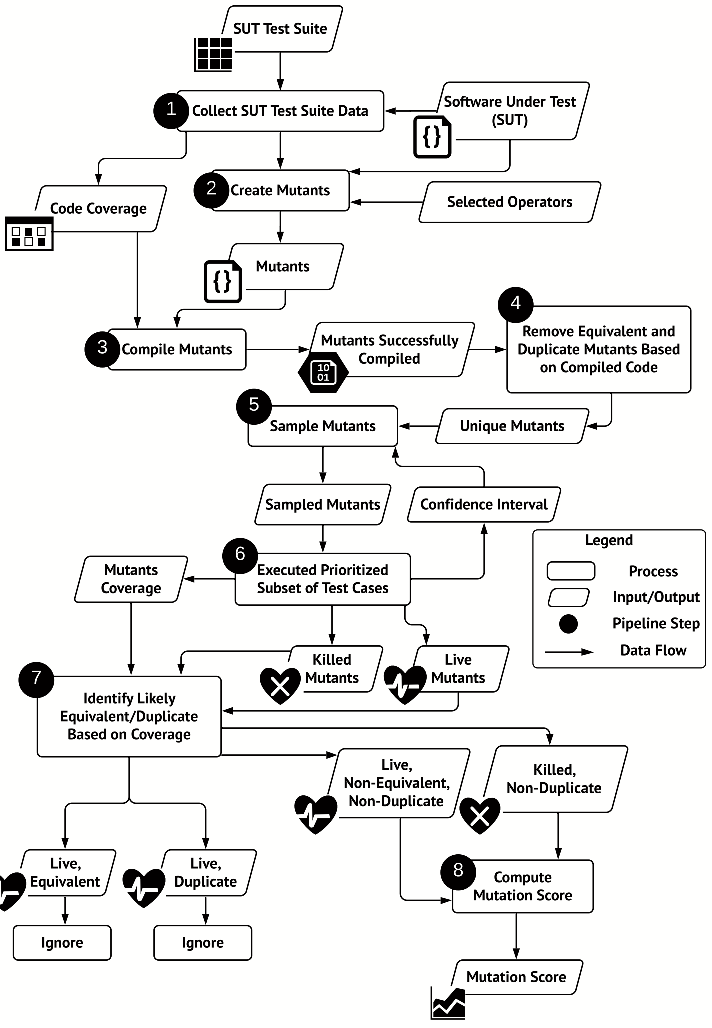


Figure 2.1: Overview of MASS

Table 2.1: 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	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&\&, \Vert\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&=, \vert=, \&=\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&, \vert, \&\&\} \wedge op_1 \neq op_2\}$
	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	$\{(v, \text{-}v), (v, v\text{-}), (v, {+}{+}v), (v, v{+}{+})\}$
OODL	AOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{+, -, *, /, \%\}\}$
	LOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&\&, \Vert\}\}$
	ROD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{>, >=, <, <=, ==, !=\}\}$
	BOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&, \vert, \wedge\}\}$
	SOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{», «\}\}$
Other	LVR	$\{(l_1, l_2) \mid (l_1, l_2) \in \{(0, -1), (l_1, -l_1), (l_1, 0),$ $(\mathit{true}, \mathit{false}), (\mathit{false}, \mathit{true})\}\}$

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

2.1.2 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 [?] and GDB [?], enabling the collection of coverage information for embedded systems without a file system [?].

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. In *MASS*, based on the considerations provided in Section 1.1.2, we rely on an extended sufficient set of mutation operators, which are listed in Table 2.2. In addition, in our experiments, we also evaluate the feasibility of relying only on the SDL operator, combined or not with OODL operators, instead of the entire sufficient set of operators.

To automatically generate mutants, we have extended SRCIRor [?] to include all the operators in Table 2.2. After mutating the original source file, our extension saves the mutated source file and keeps track of the mutation applied. Our toolset is available under the ESA Software Community Licence Permissive [?] at the following URL **https://faqas.uni.lu/**.

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.

Build systems (e.g., GNU make [?] driving the GCC [?] compiler) create one object file for each source file to be compiled and then link these object files together into the final executable. After the first build, in subsequent builds, build systems recompile only the modified files and link them to the rest. For this reason, our optimized compilation procedure, which modifies at most two source files for each mutant (i.e., the mutated file and the file restored to eliminate the previous mutation), can reuse almost all the compiled object files in subsequent compilation runs, thus speeding up the compilation of multiple mutants. The experiments conducted with our subjects have shown that our optimization is sufficient to make the compilation of mutants feasible for large projects. Other state-of-the-art solutions introduce additional complexity (e.g., change the structure of the software under test [?]) that does not appear to be justified by scalability needs.

Mutants that lead to compilation errors are discarded. Concerning compilation warnings, we assume the build system of the SUT has been properly configured; more precisely, if the system should compile without warnings, the compiler is expected to be configured to treat warnings as errors otherwise mutants that lead to warning are retained.

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. After each execution of the compiler, we compute the SHA-512 hash summary of the generated executable.To detect equivalent mutants, *MASS* compares the hash summaries of the mutants with that of the original executable. To detect duplicate mutants but avoid combinatorial explosion, *MASS* 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 [**?**]). 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. We compare hash summaries rather than executable files because it is much faster, an important consideration when dealing with a large number of mutants. 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. *MASS* 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 *MASS* 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***.

The strategies ***proportional uniform sampling*** and ***proportional method-based sampling*** were selected based on the results of Zhang et al. [**?**], who compared eight strategies for sampling mutants. The former was the best performing strategy and consists of sampling mutants evenly across all functions of the SUT, i.e., sampling r% mutants from each set of mutants generated inside the same function. The latter consists of randomly selecting r% mutants from the complete mutants set. This is included in our study because it is simpler to implement and showed to be equivalent to stratified sampling strategies, based on recent work [**?**].

The ***uniform fixed-size sampling*** strategy stems from the work of Gopinath et al. [**?**] and consists of selecting a fixed number N_M of mutants for the computation of the mutation score. Based their work, with 1,000 mutants, one can guarantee an accurate estimation of the mutation score.

In this paper, we introduce 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. Based on related work, we assume that the mutation score computed with a sample of mutants follows a binomial distribution (see Section 1.1.4). For this reason, 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 (see Section 1.1.4). Mutation analysis (i.e., sampling and testing a mutant) stops when the confidence interval is below a given threshold T_{CI} (we use $T_{CI} = 0.10$ in our experiments). More formally, given a confidence interval $[L_S; U_S]$, with L_S and U_S indicating the lower and upper bound of the interval, mutation analysis stops when the following condition holds:

$$(U_S - L_S) < T_{CI}. \tag{2.1}$$

Unfortunately, the assumption about the estimated mutation score following a binomial distribution may not hold when a subset of the test suite is executed for every mutant (which could happen in Step 6). Without going into the details behind the implementation of Step 6, which is described in Section 2.1.7, we can expect that a reduced test suite may not be able to kill all the mutants killed by the entire test suite, i.e., the estimated mutation score may be affected by negative bias. Consequently, over multiple runs, the mean of the estimated mutation score may not be close to the ***actual mutation score*** (i.e., the mutation score computed with the entire test suite exercising all the mutants for the SUT) but may converge to a lower value. To compute a correct confidence interval that includes the actual mutation score of the SUT, we thus need to take into account this negative bias.

To study the effect of negative bias on the confidence interval, we address first the relation between the actual mutation score and the mutation score computed with the reduced test suite when the entire set of mutants for the SUT is executed. A mutant killed by the entire test suite has a probability P_{KErr} of not being killed by the reduced test suite. The probability P_{KErr} can be estimated as the proportion of mutants (erroneously) not killed by the reduced test suite

$$P_{KErr} = \frac{|E_R|}{|M|} \tag{2.2}$$

with E_R being the subset of mutants that are killed by the entire test suite but not by the reduced test suite, and M being the full set of mutants for the SUT.

The mutation score for the reduced test suite (MS_R) can be computed as

$$MS_R = \frac{|K| - |E_R|}{|M|} = \frac{|K|}{|M|} - \frac{|E_R|}{|M|} = MS - \frac{|E_R|}{|M|} = MS - P_{KErr} \tag{2.3}$$

where K is the set of mutants killed by the whole test suite, M is the set of all the mutants of the SUT, and MS is the actual mutation score. Consequently, the actual mutation score can be computed as

$$MS = MS_R + P_{Err_R} \tag{2.4}$$

We now discuss the effect of a reduced test suite on the confidence interval for a mutation score estimated with mutants sampling. When mutants are sampled and tested with the entire test suite, the actual mutation score is expected to lie in the confidence interval $[L_S; U_S]$. In the presence of a reduced test suite, we can still rely on the Clopper-Pearson method to compute the confidence interval $CI_R = [L_R; U_R]$. However, we have to take into account the probability of an error in the computation of the mutation score MS_R; MS_R can be lower than MS and, based on Equation 2.4, we expect the actual mutation score to lie in an interval that is shifted with respect to the interval for MS_R:

$$CI = [L_R + P_{KErr}; U_R + P_{KErr}] \tag{2.5}$$

We can only estimate P_{KErr} since computing it would require the execution of all the mutants with the complete test suite, thus undermining our objective of reducing test executions. To do so, we can randomly select a subset M_R of mutants, on which to execute the entire test suite and identify the mutants killed by the reduced test suite. The size of the set M_R should be lower than the number of mutants we expect FSCI sampling to return, otherwise sampling would not provide any cost reduction benefit. Since, for every mutant in M_R, we can determine if it is erroneously reported as not killed by the reduced test suite R, we can estimate the probability P_{KErr} as the percentage of such mutants. As for the case of the mutation score, we assume that the binomial distribution provides a conservative estimate of the variance for P_{KErr}.

We can estimate the confidence interval for P_{KErr} using one of the methods for binomial distributions. We rely on the Wilson score method because it is known to perform well with small samples [?]. The value of P_{KErr} will thus lie within $CI_E = [L_E; U_E]$, with L_E and U_E indicating the lower and upper bounds of the interval.

Based on Equation 2.5, the confidence interval to be used with FSCI sampling in the presence of a reduced test suite should thus be

$$CI = [L_R + L_E; U_R + U_E] \tag{2.6}$$

The estimated mutation score is the value lying in the middle of the interval.

Since the width of the confidence interval CI (hereafter, $|CI|$) results from the sum of $|CI_R|$ and $|CI_E|$, mutation sampling with a reduced test suite may lead to the execution of a larger set of mutants.

Based on Equations 2.1 and 2.6, $|CI_R| \leq T_{CI} - |CI_E|$. Consequently, when $|CI_E| > T_{CI}$, the reduced test suite cannot lead to sufficiently accurate results. Also, a large $|CI_E|$ may prevent the identification of accurate results with a feasible number of mutants. For example, Clopper-pearson may require up to 1568 samples for a confidence interval below 0.05 [?]. We shall thus identify a threshold (T_{CE}) for the confidence interval $|CI_E|$ that enables accurate estimates with a small sample size (e.g., in the worst case, with less than 1000 samples, the sample size for related work). For this reason, starting from a minimal number of samples to estimate P_{KErr} (150 in our experiments), *MASS* keeps estimating P_{KErr} until it yields $|CI_E| \leq T_{CE}$. In our experiments we set $T_{CE} = 0.035$. To select T_{CE}, we have identified a reasonable minimal mutation score to be expected in space software (i.e., 65%) and identified, based on confidence interval estimation methods with finite population correction factor [?], the minimal value for $|CI_E|$ that requires a number of samples below 850 (i.e., $1000 - 150$).

When it is not possible to estimate $|CI_E| \leq T_{CE}$ or when the number of samples required to estimate $|CI_E| \leq T_{CE}$ is sufficient to accurately estimate the mutation score, the test suite can be prioritized but not reduced and the confidence interval is computed using the traditional Clopper-Pearson method, i.e., $[L_S; U_S]$.

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. Similarly to the approach of Zhang et al. [?], 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 (see Section 1.1.4). 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.

To reduce the number of test cases to be executed with a mutant, we should first execute the ones that more likely satisfy the necessity condition. This might be achieved by executing a test case that exercises the mutated statement with variable values not observed before. Unfortunately, in our context, the size of the

SUT and its real-time constraints prevent us from recording all the variable values processed during testing.

Therefore, we rely on code coverage to determine if two test case executions exercise the mutated statement with diverse variable values. Such coverage is collected by efficient procedures provided by compilers, thus having lower impact on execution performance than other types of dynamic analysis solutions (e.g., tracing variable values). Since, because of control- and data-flow dependencies, a different set of input values may lead to differences in code coverage, the latter helps determine if two or more test cases likely exercise a mutated statement with different variable values. To increase the likelihood that the observed differences in code coverage are due to the use of different variable values to exercise the mutated statement, we restrict the scope of code coverage analysis to the functions belonging to the component (i.e., the source file) that contains the mutated statement. Indeed, such functions typically present several control- and data-flow dependencies, thus augmenting the likelihood that a coverage difference is due to the execution of the mutated statement with a diverse set of values. Also, collecting code coverage for a small part of the system further reduces the impact of our analysis on system performance.

Based on related work, we have identified two possible strategies to characterize test case executions based on code coverage:

S1 Compare the sets of source code statements that have been covered by test cases [**?**].

S2 Compare the number of times each statement has been covered by test cases [**?**].

To determine how dissimilar two test cases are and, consequently, how likely they exercise the mutated statement with different values, we rely on widely adopted distance metrics. In the case of S1, we rely on the Jaccard and Ochiai index, which are two similarity indices for binary data and have successfully been used to compare program executions based on code coverage [**?**, **?**, **?**]. The Jaccard index is also known as ***intersection over union*** it measures similarity between sets, and is defined as the size of the intersection divided by the size of the union. The Jaccard distance measures dissimilarity between sample sets and results from subtracting the Jaccard coefficient from 1. The Ochiai index calculates cosine similarity with binary data, it is used in molecular biology and software engineering. Given two test cases T_A and T_B, the Jaccard (D_J) and Ochiai (D_O) distances are computed as follows:

$D_J(T_a, T_b) = 1 - \frac{|C_a \cap C_b|}{|C_a \cup C_b|}$ $\quad D_O(T_a, T_b) = 1 - \frac{|C_a \cap C_b|}{\sqrt{|C_a| * |C_b|}}$, where C_a and C_b are the set of covered statements exercised by T_a and T_b, respectively.

In the case of S2, we compute the distance between two test cases by relying on the euclidean distance (D_E) and the cosine similarity distance (D_C), two popular distance metrics used in machine learning. Euclidean distance is the straight-line distance between two points in Euclidean space; precisely, the Euclidean distance between two points is the length of the line segment connecting them. In our context, the two vectors consist of the number of times the program statements had been exercised by a test. Cosine similarity measures similarity between two vectors of an inner product space. It results from the inner product of the same vectors normalized to both have length 1, which matches the the cosine of the angle between them. It is widely adopted to measure cohesion within clusters in data mining. Given two vectors V_A and V_B, whose elements capture the number of times a statement has been covered by test cases T_A and T_B, the distances D_E and D_C can be computed as follows:

$D_E = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$

$D_C = 1 - \frac{\sum_{i=1}^{n} A_i * B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2} * \sqrt{\sum_{i=1}^{n} {B_i}^2}}$, where A_i and B_i refer to the number of times the i-th statement had been covered by T_A and T_B, respectively.

Figure 2.2 shows the pseudocode of our algorithm for selecting and prioritizing test cases. It generates as output a prioritized test suite (***PTS***). Based on the findings of Zhang et al. [**?**], we first select the test case that exercises the mutated statement the highest number of times (Line 3) and add it to the prioritized test suite

(Line 4). Then, in the next iterations, the test case selected is the one with the largest distance from the closest test case already selected (Lines 10 to 13). When two or more test cases have the same distance, we select randomly among the test cases that exercise the mutated statement the most.

The algorithm iterates as long as it identifies a test case showing a difference in code coverage from the already selected test cases (Line 14).

Test cases are then executed in the selected order. During execution, we collect code coverage information and identify killed and live mutants.

```
Require: TS, the test suite of the software under test
Require: Cov, coverage information, for each test case
Require: ms, the mutated statement
Ensure: PTS, a list of test cases to be executed, sorted by priority
 1: $TS_m$ ← subset of TS that cover the mutated statement ms, based on Cov
 2: PTS ← newlist //this list is initially empty
 3: PTS ← based on Cov select from $TS_m$ the test case $t$ that exercises ms more times
 4: PTS ← PTS ∪ $t$ //include first the test case selected above
 5: repeat
 6:     for each $n$ in the set ($TS_m$ - PTS) , which is the set of test cases not already added to PTS
 7:         for each $t$ in PTS
 8:             compute the distance between $t$ and $n$
 9:         identify $t_n$ i.e., the test case $t$ with the minimal $d$
10:     among all the $t_n$ identified, select the one with the highest distance $d$
11:     if $d > 0$ //there is at least a test case with a different coverage
12:         //note: $n$ is the test case in the set ($TS_m$ - PTS) closer to $t_n$
13:         PTS ← PTS ∪ $n$
14: until $d > 0$
```

Figure 2.2: Algorithm for prioritizing test cases

2.1.8 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 [?], we identify nonequivalent and nonduplicate mutants based on a threshold.

In our case, consistently with previous steps of *MASS*, 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.

Figure 2.3 shows the algorithm for detecting nonequivalent and nonduplicate mutants. It first identify among the list of killed mutants all the non-duplicate ones (Line 1). Then it identifies the non-equivalent mutants among the list of live mutants (Line 2). Finally, it further filters the list of non-equivalent mutants to keep only the ones that appear to be nonduplicate (Line 3).

```
Require: D, the distance function to use to identify equivalent/duplicate mutants
Require: KM, list of killed mutants
Require: LM, list of live mutants
Require: $Cov_O$, coverage information for all the test cases, for the original program
Require: $Cov_M$, coverage information for all the executed test cases, for every mutant
Require: TS, list of test cases
Require: TR, test results, for all the executions
Ensure: KND, a list of killed, non-duplicate mutants
Ensure: LNEND, a list of live, non-equivalent, non-duplicate mutants
KND ← identifyNonDuplicateMutants(KM, TS, TR, $Cov_M$)
LNE ← identifyNonEquivalentMutants(LND, TS, $Cov_M$, $Cov_O$)
LNEND ← identifyDuplicateMutants(LNE, TS, $Cov_M$)
procedure identifyNonDuplicateMutants(M, TS, TR, $Cov_M$)//$M$ is a list of mutants, TS, TR, and $Cov_M$ are defined above
    ND ← emptyset
    k1 ← extract and remove first element of M
    ND ← ND ∪ k1
    while M not empty do
        k2 ← extract and remove first element of M
        for mutant $k1$ in ND do
            duplicate = TRUE
            for test case $t$ in TS do
                if $t$ has different result in $k1$ and $k2$ then
                    duplicate = FALSE
                    break
                else
                    $cov_{k1t}$ ← extract coverage information for test case $t$ executed with mutant $k1$
                    $cov_{k2t}$ ← extract coverage information for test case $t$ executed with mutant $k2$
                    if $D(cov_{k1t}, cov_{k2t}) > T_R$ then
                        duplicate = FALSE
                        break
                    end if
                end if
            end for
            if duplicate == FALSE then
                break //No need to compare with all the mutants if we know that it is not duplicate
            end if
        end for
        if duplicate == FALSE then
            ND ← ND ∪ k2
        end if
    end while
    return ND
end procedure
procedure identifyNonEquivalentMutants(M, TS, $Cov_M$, $Cov_O$)//$M$ is a list of mutants, TS, and $Cov_M$, and $Cov_O$ are defined above
    NE ← emptyset
    while M not empty do
        m ← extract and remove first element of M
        for test case $t$ in TS do
            $cov_m$ ← extract coverage information for test case $t$ executed with mutant $m$
            $cov_o$ ← extract coverage information for test case $t$ executed with original program
            if $D(cov_m, cov_o) > T_E$ then
                equivalent = FALSE
                break
            end if
        end for
        if equivalent == FALSE then
            NE ← NE ∪ m
        end if
    end while
    return NE
end procedure
```

Figure 2.3: Algorithm for identifying non-equivalent and non-duplicate mutants

2.1.9 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):

$$MS = \frac{|KND|}{|LNEND| + |KND|} \tag{2.7}$$

2.2 Code-driven Mutation Testing: SEMuS

2.2.1 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. As demonstrated with our preliminary evaluation of existing mutation analysis tools, any analysis requiring compilation into LLVM format is unlikely applicable to space software because of two reasons: (1) our case studies 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 2.4 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.

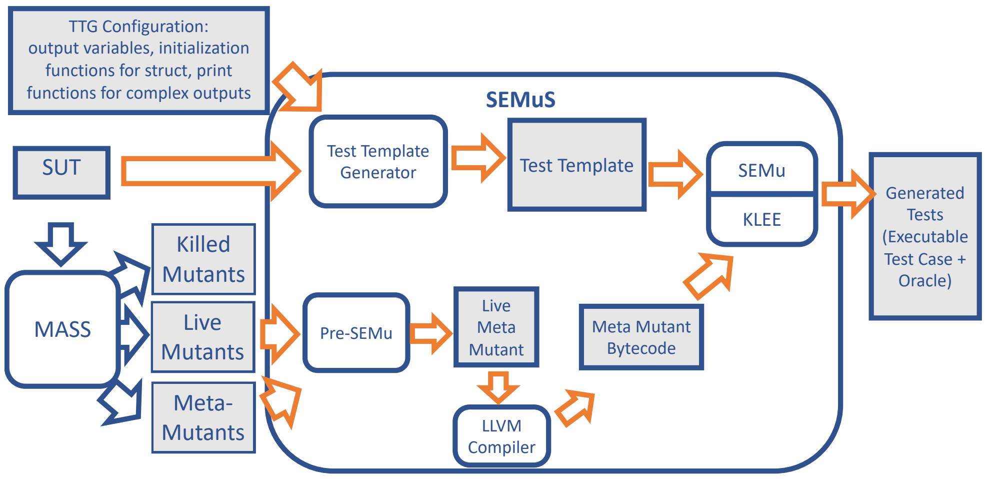

TTG Configuration: output variables, initialization functions for struct, print functions for complex outputs
SEMuS
SUT
Test Template Generator
Test Template
SEMu
KLEE
Generated Tests (Executable Test Case + Oracle)
Killed Mutants
MASS
Live Mutants
Meta-Mutants
Pre-SEMu
Live Meta Mutant
Meta Mutant Bytecode
LLVM Compiler

Figure 2.4: FAQAS-SEMuS Architecture and Workflow

2.2.2 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 2.1 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 2.1). Finally, it generates a number of invocations of the *printf* function that print the value of the software outputs and adds a return statement with the value returned by the function under test (e.g., `result_faqas_semu` in Listing 2.1). Such *printf* invocations are necessary because of the way SEMu determines that a mutant is killed; two are the cases in which the mutant is considered killed: (1) the main function returns a different return value, (2) different values are printed to the standard output. Consequently, it is necessary to print out all the values of the software outputs. To select which variables to print, for every source file under test, the engineer can specify, in the SEMuS configuration file, the arguments (typically pointers) that should be either considered output or considered both input and outputs. By default, the TTG considers all the function arguments as inputs, in case some argument (e.g., a pointer to a memory buffer) is used both as input and as output the engineer shall specify it as such; similarly, in case some some argument (e.g., a pointer to a struct) is used to store outputs, the engineer shall indicate that it is an output. Moreover, since, in the C language, the function *printf* cannot automatically determine how to print the different fields specified in a data structure (i.e., it prints only the memory address of the pointer), the engineer can also specify how to printout the different fields of a data structure.

Listing 2.2 shows an example configuration file for SEMuS. In addition to output arguments (i.e., *OUT_-ARGS_NAMES*), input/output arguments (i.e., *IN_OUT_ARGS_NAMES*), and customized printf instructions (i.e., *TYPES_TO_PRINTCODE*), the SEMuS configuration file enables the engineer to customize the generation of the test template further. Indeed, engineers can specify input argument types that should not be treated symbolically but that shall be initialized using a specific function of the SUT (i.e., *TYPE_TO_INITIALIZATION-CODE*); also, engineers can specify the fields, within such types (e.g., the attributes of a struct), that shall be treated symbolically. Moreover, since the template returns the value of the function under test, the engineer can specify how to convert the returned value to int, if necessary (see parameter *TYPES_TO_INTCONVERT*). Finally, in case the function under test receives a pointer to an array (which is typically passed as a pointer), the engineer can specify the size of such array (see parameter *ARG_TYPE_TO_ITS_POINTER_ELEM_NUM*); by default, SEMuS assumes that a pointer refers to a single element (i.e., it is a pointer to a variable not an array).

```c
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 2.1: SEMuS test template.

```json
{
"TYPES_TO_INTCONVERT": {},
"TYPES_TO_PRINTCODE": {"gs_timestamp_t": "printf(\"FAQAS-SEMU-TEST-OUTPUT: result_faqas_semu = tv_sec: %u, tv_nsec: %u\\n\", {}.tv_sec, {}.tv_nsec);"},
"OUT_ARGS_NAMES": ["pErrCode"],
"IN_OUT_ARGS_NAMES": ["base"],
"TYPE_TO_INITIALIZATIONCODE": {},
"TYPE_TO_SYMBOLIC_FIELDS_ACCESS": {},
"VOID_ARG_SUBSTITUTE_TYPE": "",
"ARG_TYPE_TO_ITS_POINTER_ELEM_NUM": {"char *": 6}
}
```

Listing 2.2: SEMuS configuration example.

2.2.3 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 supported by the *LLVM* compiler infrastructure.

```
flag T_INT_IsConstraintValid(const T_INT* pVal, int* pErrCode)
{
    flag ret = TRUE;
    (void)pVal;

    ret = ((*(pVal)) <= 50UL);
    *pErrCode = ret ? 0 :  ERR_T_INT;

    return ret;
}
```

Listing 2.3: Function T_INT_IsConstraintValid.

```
flag T_INT_IsConstraintValid(const T_INT* pVal, int* pErrCode)
{
    flag ret = TRUE;
    (void)pVal;

    ret = ((*(pVal)) <= 50UL);
    *pErrCode = ret ? 1 :  ERR_T_INT;

    return ret;
}
```

Listing 2.4: Mutant 1 of function T_INT_IsConstraintValid.

```
flag T_INT_IsConstraintValid(const T_INT* pVal, int* pErrCode)
{
    flag ret = TRUE;
    (void)pVal;

    ret = ((*(pVal)) <= 50UL);
    *pErrCode = ret ? (-1) :  ERR_T_INT;

    return ret;
}
```

Listing 2.5: Mutant 2 of function T_INT_IsConstraintValid.

Listings 2.3 provides the source code of function *T_INT_IsConstraintValid*, while Listings 2.4 and 2.5 provide two example mutants generated by MASS. Listing 2.6 provides an example ***meta-mutant*** including the same two mutants of Listings 2.4 and 2.5. To select the mutants to analyze at runtime, SEMu relies on three support functions that shall be invoked within the eta-mutant:

- `klee_semu_GenMu_Mutant_ID_Selector_Func`: function that takes two mutant IDs as arguments, representing a range of mutant IDs. It specifies where a portion of code containing mutants start.
- `klee_semu_GenMu_Mutant_ID_Selector`: global variable that contains the ID of the mutant to be activated durng the analysis with SEMu.
- `klee_semu_GenMu_Post_Mutation_Poin_Func`: function that takes two mutant IDs as arguments, it specifies where a portion of code containing mutants ends. It is used by SEMu to identify the portion of the code where to compare the state of the original and the mutated program and determine if the mutation has affected the program state (i.e., the ***necessity*** condition to kill a mutant). In other words, it enables SEMu to perform conservative pruning and remove the mutant states that are not infected.

In Listing 2.6, mutant 2 from Listing 2.5 appears on line 11 (indeed, the value *-1* is selected when the mutant ID is equal to 2). Mutant 1 from Listing 2.4 appears on line 12 (indeed, the value *1* is selected when the mutant ID is equal to 1). The original software is represented by the mutant ID zero; indeed the value *0*

```
 1 flag T_INT_IsConstraintValid(const T_INT* pVal, int* pErrCode)
 2 {
 3     flag ret = TRUE;
 4     (void)pVal;
 5 
 6     ret = ((*(pVal)) <= 50UL);
 7 
 8     klee_semu_GenMu_Mutant_ID_Selector_Func(1,2);
 9     *pErrCode = ret ?
10       ( klee_semu_GenMu_Mutant_ID_Selector==2 ?
11         ((-1)):
12         (klee_semu_GenMu_Mutant_ID_Selector==1?
13           (1):
14           (0)))
15           :  ERR_T_INT;
16     klee_semu_GenMu_Post_Mutation_Point_Func(0,0);
17     klee_semu_GenMu_Post_Mutation_Point_Func(1,2);
18 
19     return ret;
20 }
```

Listing 2.6: Meta-Mutant for function T_INT_IsConstraintValid.

(i.e., what appears in line 7 of the Listing 2.3) is selected on line 14 (i.e., when the mutant ID is neither *2* nor *1*).

2.2.4 KLEE-SEMu

KLEE-SEMu is the underlying test generation component, previously described in Section 1.3.2. 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.

An example of a KLEE test is presented in Listing 2.7. The field *args* report the entry point of the analysis; in this case, the test generation was performed for live mutants present in the function `T_INT_IsConstraintValid`, which SEMuS stores in a dedicated folder. The fields named *object* provide information about the outputs generated by KLEE (e.g., the generated test inputs). For each object, the KLEE test provides a *name* (usually the name of the symbolic variable), its *size*, and the actual *value* generated by KLEE through constraint solving (usually this is reported in binary form). Objects are numbered. Object number *0* reports information about the data structure used by KLEE, that is, the version of the structure. The other objects report information about the generated test inputs. Our example shows that one value of size 8 was generated for the variable `pVal`, the data field shows the binary representation of the `pVal` variable, in this case `pVal=0`.

```
ktest file : 'test000001.ktest'
args       : ['/MakeSym-TestGen-Input/direct/T_INT_IsConstraintValid/test.MetaMu.bc']
num objects: 2
object    0: name: b'model_version'
object    0: size: 4
object    0: data: b'\x01\x00\x00\x00'
object    1: name: b'pVal'
object    1: size: 8
object    1: data: b'\x00\x00\x00\x00\x00\x00\x00\x00'
```

Listing 2.7: Klee-test output

2.2.5 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, match 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.

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

Listing 2.8: Generated test case

Listing 2.8 shows an example of a test case generated for a mutant present in the function `T_INT_IsConstraintValid`. For instance, line 20 shows that the variable `pVal` is initially filled with zeros, then in line 21, it is filled with the value stored in the variable `pVal_faqas_semu_test_data`, which holds the binary output produced by KLEE. In line 25, the function under test is invoked with the concrete value of `pVal`.

The test case generated by the KTU prints the function return value and the value of every variable passed by reference, using the same instructions of the test template. KTU cannot generate test assertions because only engineers can know, based on specifications, what are the values to be expected at the end of the test case execution. However, the generated printf invocations still play the role of an oracle for regression testing,as explained in the next sections.

2.2.6 Test suite augmentation

The procedure for testing with SEMuS is shown in Figure 2.5. 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 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 2.5), 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.

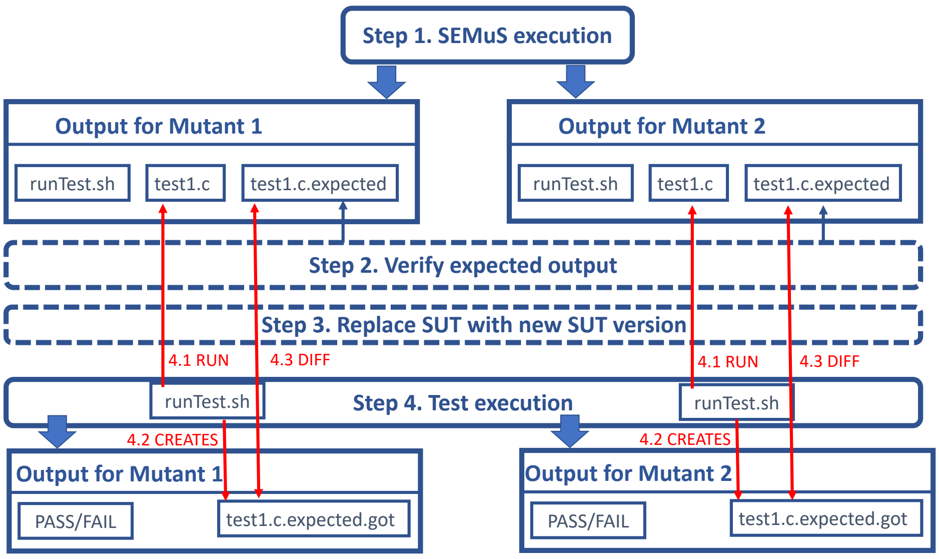


Figure 2.5: Workflow for test suite augmentation with SEMuS

2.2.7 Live mutants

SEMuS may not be capable of selecting test inputs that kill the mutant under analysis. We may distinguish two cases:

- SEMuS execution terminates and no test case is generated.
- SEMuS execution does terminate (i.e., it is killed after a timeout configured by the end-user, usually 15 minutes are sufficient to generate test cases).

In the first case, SEMuS has successfully exercised all the execution paths covering the mutated statements but did not identify inputs that satisfy the killing conditions. This may case indicate that the mutant is equivalent and may be discarded (however, the engineer shall verify that the test template is configured correctly). Also, we may be in such situations when some of the functions under test belong to libraries not compiled with LLVM that, consequently are not correctly processed by KLEE-SEMu; to detect these cases the engineer shall look for errors in the output generated by KLEE.

In the second case, SEMuS did not complete the analysis of the possible execution paths covering the mutated statement. Such cases may indicate that test generation is complex and probably an engineer may more efficiently select test inputs than the underlying constraint solving solution implemented by KLEE-SEMu.

2.3 Data-driven Mutation Analysis: DAMAt

2.3.1 Overview

In this Section, we propose ***data-driven mutation analysis***, 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. Also, we present a technique, ***data-driven mutation analysis with tables*** (DAMAt), to automate data-driven mutation analysis by relying on a fault model that captures, for a specific set of components, both the characteristics of the data to mutate (e.g., the size and structure of the messages generated by the ADCS) and the types of fault that may affect such data (e.g., a value out of the nominal range). The latter is formalized as a set of parameterizable mutation operators. To simplify adoption, we rely on fault models in tabular form where each row specifies, for a given data item, what mutation operator (along with its corresponding parameter values) to apply to which elements of the data item. At runtime, DAMAt modifies the data exchanged by components according to the provided fault model (e.g., replaces a nominal voltage value with a value out of the nominal range).

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.

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 2.6 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 2.7. 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, see Section 2.3.1.4). 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.

In the following sections we describe the structure of our fault model and each step of DAMAt .

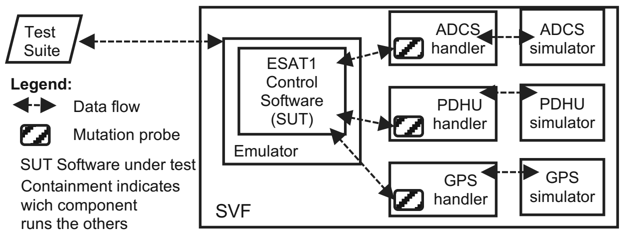


Figure 2.6: Data mutation probes integrated into ESAIL.

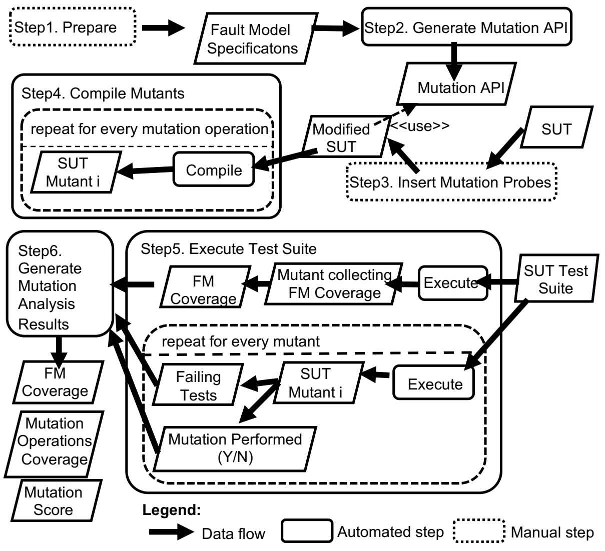


Figure 2.7: The DAMAt process.

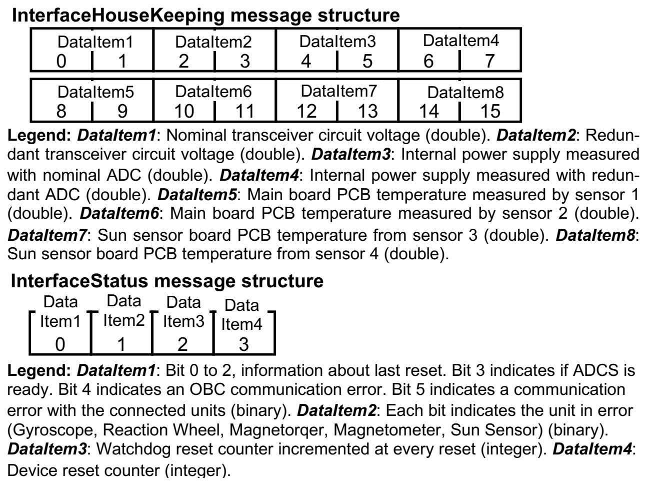
2.3.1.1 Fault Model Structure

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. In this paper, 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. For example, in the case of ESAIL we have defined one fault model for every message type that could be exchanged by the three components under test (i.e., ADCS, PDHU, and GPS). In total, for ESAIL, we have 14 fault models, 10 for the communication concerning ADCS (we have 10 different message types), 3 for PDHU, and 1 for GPS.

The DAMAt fault model 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. Also, more complex data structures (e.g., hierarchical ones like trees) are often flattened into data buffers in order to be exchanged by different components (e.g., through the network). When the CPS software is implemented in C or C++ (common CPS development languages) data buffers are implemented as arrays. Figure 2.8 shows three block diagrams representing (part of) the buffer structure used to exchange messages of type InterfaceHouseKeeping and InterfaceStatus in ESAIL.

A data buffer is characterized by a ***unit size*** that specifies the dimension, in bytes, of the single cell of the underlying array and a ***buffer size***, which specifies the total number of units belonging to the buffer. Each data buffer can contain one or more ***data items***; the size of data items may vary as they may span over multiple units. Also, each data item is interpreted by the CPS software according to a specific ***representation*** (e.g., integer, double, etc.). In ESAIL, the unit size is one byte and the data items may span over one or two buffer units (see Figure 2.8).

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



Legend: ***DataItem1***: Nominal transceiver circuit voltage (double). ***DataItem2***: Redundant transceiver circuit voltage (double). ***DataItem3***: Internal power supply measured with nominal ADC (double). ***DataItem4***: Internal power supply measured with redundant ADC (double). ***DataItem5***: Main board PCB temperature measured by sensor 1 (double). ***DataItem6***: Main board PCB temperature measured by sensor 2 (double). ***DataItem7***: Sun sensor board PCB temperature from sensor 3 (double). ***DataItem8***: Sun sensor board PCB temperature from sensor 4 (double).

Legend: ***DataItem1***: Bit 0 to 2, information about last reset. Bit 3 indicates if ADCS is ready. Bit 4 indicates an OBC communication error. Bit 5 indicates a communication error with the connected units (binary). ***DataItem2***: Each bit indicates the unit in error (Gyroscope, Reaction Wheel, Magnetorqer, Magnetometer, Sun Sensor) (binary). ***DataItem3***: Watchdog reset counter incremented at every reset (integer). ***DataItem4***: Device reset counter (integer).

Figure 2.8: Structure of data buffers in ESAIL.

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 2.2 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 [**?**, **?**, **?**]; the operators VAT, VBT, FVAT, FVBT, FVOR, IV, ASA, and HV are a contribution of this paper and were derived and conceptualised as a result of discussion with domain experts. Although other data representation types (e.g., null terminated strings) and operators (e.g., replacement of a random char in a string) might be envisioned, in this paper, we focus on operators that are necessary in the CPS context, based on our experience. For example, CPS components are unlikely to exchange strings.

Table 2.2: Data-driven mutation operators

Fault Class	Types	Parameters	Description
Value above threshold (VAT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	Replaces the current value with a value above the threshold T for a delta (Δ). It simulates a value that is out of the nominal case and shall trigger a response from the system that shall be verified by the test case (e.g., the system may continue working but an alarm shall be triggered). Not applied if the value is already above the threshold. ***Data mutation procedure:*** $v' = \begin{cases} (T+D) & \textit{if} v \leq T \\ v & \textit{otherwise} \end{cases}$
Value below threshold (VBT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	Replaces the current value with a value below the threshold T for a delta (Δ). It simulates a value that is out of the nominal case and shall trigger a response from the system that shall be verified by the test case (e.g., the system may continue working but an alarm shall be triggered). Not applied if the value is already below the threshold. ***Data mutation procedure:*** $v' = \begin{cases} (T-D) & \textit{if} v \geq T \\ v & \textit{otherwise} \end{cases}$
Value out of range (VOR)	I,L,F,D,H	MIN: minimum valid value MAX: maximum valid value Δ: delta, difference with respect to minimum/-maximum valid value	Replaces the current value with a value out of the range $[MIN; MAX]$. It simulates a value that is out of the nominal range and shall trigger a response from the system that shall be verified by the test case (e.g., the system may continue working but an alarm shall be triggered). Not applied if the value is already out of range. This was inspired by the *ARBC* operator [?]; however, DAMAt enables engineers to explicitly specify the delta. ***Data mutation procedure 1:*** $v' = \begin{cases} (MIN-D) & \textit{if} MIN \leq v \leq MAX \\ v & \textit{otherwise} \end{cases}$ ***Data mutation procedure 2:*** $v' = \begin{cases} (MAX+D) & \textit{if} MIN \leq v \leq MAX \\ v & \textit{otherwise} \end{cases}$
Bit flip (BF)	B	MIN: lower bit MAX: higher bit STATE: mutate only if the bit is in the given state (i.e., 0 or 1). VALUE: number of bits to mutate	A number of bits randomly chosen in the positions between MIN and MAX (included) are flipped. If STATE is specified, the mutation is applied only if the bit is in the specified state; the value -1 indicates that any state shall be considered for mutation. Parameter VALUE specifies the number of bits to mutate. This was inspired by the *BitFlipperMutator* operator [?]; however, DAMAt introduces the STATE parameter, which is not supported by related work. ***Data mutation procedure:*** the operator flips VALUE randomly selected bit if they are in the specified state.
Invalid numeric value (INV)	I,L,F,D,H	MIN: lower valid value MAX: higher valid value	Replace the current value with a mutated value that is legal (i.e., in the specified range) but different than current value. It simulates the exchange of data that is not consistent with the state of the system. It matches the *ARR* operator [?]. ***Data mutation procedure:*** Replace the current value with a different value randomly sampled in the specified range.
Illegal Value (IV)	I,L,F,D,H	VALUE: illegal value that is observed	Replace the current value with a value that is equal to the parameter *VALUE*. It matches the *ValidValuesMutator* operator [?]. ***Data mutation procedure:*** $v' = \begin{cases} \textit{VALUE} & \textit{if} v \neq \textit{VALUE} \\ v & \textit{otherwise} \end{cases}$
Anomalous Signal Amplitude (ASA)	I,L,F,D,H	T: change point Δ: delta, value to add/remove VALUE: value to multiply	The mutated value is derived by amplifying the observed value by a factor *V* and by adding/removing a constant value $\Delta$from it. It is used to either amplify or reduce a signal in a constant manner to simulate unusual signals. The parameter *T* indicates the observed value below which instead of adding we subtract . ***Data mutation procedure:*** $v' = \begin{cases} T + ((v-T) * \textit{VALUE}) + D & \textit{if } v \geq T \\ T - ((T-v) * \textit{VALUE}) - D & \textit{if } v < T \end{cases}$
Signal Shift (SS)	I,L,F,D,H	Δ: delta, value by which the signal should be shifted	The mutated value is derived by adding a value $\Delta$to the observed value. It simulates an anomalous shift in the signal. This was inspired by work on signal mutation [?]; however, DAMAt also enables engineers to rely on SS to increment (or decrement) counters and identifiers. ***Data mutation procedure:*** $v' = v + \Delta$
Hold Value (HV)	I,L,F,D,H	V: number of times to repeat the same value	This operator keeps repeating an observed value for V times. It emulates a constant signal replacing a signal supposed to vary. ***Data mutation procedure:*** $v' = \begin{cases} \textit{previous } v' & \textit{if counter} \leq V \\ v & \textit{otherwise} \end{cases}$
Fix value above threshold (FVAT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	It is the complement of VAT and implements the same mutation procedure as VBT but we named it differently because it has a differently purpose. Indeed, it is used to verify that test cases exercising exceptional cases are verified correctly. 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. ***Data mutation procedure:*** $v' = \begin{cases} v & \textit{if} v > T \\ (T-D) & \textit{otherwise} \end{cases}$
Fix value below threshold (FVBT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	It is the counterpart of FVAT for the operator VBT. ***Data mutation procedure:*** $v' = \begin{cases} v & \textit{if} v < T \\ (T+D) & \textit{otherwise} \end{cases}$
Fix value out of range (FVOR)	I,L,F,D,H	MIN: minimum valid value MAX: maximum valid value	It is the complement of VOR and implements the same mutation procedure as INV but we named it differently because it has a different purpose. Indeed, it is used to verify that test cases exercising exceptional cases are verified correctly. ***Data mutation procedure:*** $v' = \begin{cases} v & \textit{if} MIN \leq v \leq MAX \\ \textit{random}(\textit{MIN}, \textit{MAX}) & \textit{otherwise} \end{cases}$

Legend: I: INT, L: LONG INT, F: FLOAT, D: DOUBLE, B: BIN, H: HEX

Table 2.4: DAMAt fault modelling methodology

Data nature	Representation type	Dependencies	# of input partitions	Operators	Comments
numerical	I, L, F, D	stateless/stateful	2	[VAT,FVAT] or [VBT,FVBT]	Nominal below T Nominal above T
			3 or more	[VOR,FVOR]	
		stateful		INV	For valid range
				[VOR,FVOR]	For out of range
		signal		ASA, SS, HV	
categorical	I, H	N/A	N/A	IV	
	B	N/A	N/A	BF	
ordinal	I, H	N/A	N/A	ASA	
other	B	N/A	N/A	BF	

Legend: N/A not applicable.

2.3.1.2 Fault Modelling Methodology (Step 1)

The fault model shall enable the specification of all possible interoperability problems in the SUT while minimizing equivalent and redundant mutants. Equivalent mutants have the same observable output as the original SUT. Instead, redundant mutants have the same observable output as other mutants. We use the term ***observable output*** to refer to any output that can be verified by the test suite. The equivalent or redundant nature of a mutant depends on the equivalence relation for observable outputs (i.e., how to determine if two outputs are the same). In a testing context, such equivalence relation depends on the type of testing being performed. For example, system test cases, different than unit test cases, are unlikely to verify the values of all the state variables of the system and thus mutants that are nonequivalent for unit test suites might be considered equivalent for system test suites. For example, in satellite systems, the correctness of the GPS triangulation algorithm output is verified by unit test cases; system test cases, instead, verify if the software takes appropriate actions when the satellite is out of orbit. Consequently, slight changes in the coordinates communicated by the GPS component may not lead to any change in the observable output verified by the test suite.

We provide a set of guidelines for the definition of fault models that are summarized in Table 2.4. For guidance, we account for the nature of the data (i.e., numerical, categorical, ordinal, or binary) and their representation type. Also, for numerical data, we consider the data dependencies, that is how data values depend on the previously observed values; we identified three categories: stateless (i.e., there are no dependencies between consecutive values), stateful (i.e, values depend on previous ones), and signal (i.e., values derive from a function of independent variables like time). Data dependencies determine the granularity of the mutation (i.e., with data dependencies, small differences shall be noticed); for non numerical data, we do not provide mutation operators with different granularities and data dependencies can be ignored.

For ***stateless numerical data***, our guidelines are driven by input space partitioning concepts [**?**]. Indeed, given equivalence relations among outputs, it is unlikely that every change in ***stateless numerical data*** will result into nonequivalent mutants; however, we can partition the input domain into regions with equivalent values (partitions). Precisely, we rely on the ***interface-based input domain modeling*** approach [**?**]: for each data item we identify a number of input partitions (set of values or value ranges) according to the interface specifications of the interacting components. In our methodology, the number and type of mutation operators selected for stateless numerical data depend on the number of input partitions identified. With *two input partitions* (e.g., nominal and exceptional data values), engineers can rely either on the pair [VBT,FVBT] or the pair [VAT,FVAT]. With *three partitions*, engineers must configure one VOR and one FVOR operator. If a different delta (Δ) is considered for the upper and lower bounds, engineers may configure two pairs [VBT,FVBT] and [VAT,FVAT], for the lower and upper bound, respectively. In the presence of *more than three* partitions, engineers shall configure one [VOR,FVOR] pair for each extra partition above three (e.g., two pairs in the case of five partitions). The parameter Δ is used to determine the partition to which the mutated data belongs.

In the presence of ***stateful data***, replacement with random values in the valid range (i.e., the INV operator) will lead to nonequivalent mutants (e.g., because it leads to data values that are systematically different than the values expected for the current system state). Alternatively, the valid data range might be partitioned as for stateless data. However, to avoid redundant mutants, engineers should rely either on the INV operator or the partitioning of the valid data range. The effect of data outside the valid data range should instead be verified

Table 2.5: Portion of the fault model specification for ESAIL

#	Fault Model	Position	Span	Type	Op	MIN	MAX	T	DELTA	STATE	VALUE
1	IfHK	8	2	DOUBLE	VAT	-	-	33.53	0.01	-	-
2	IfHK	8	2	DOUBLE	FVAT	-	-	33.53	0.01	-	-
3	IfHK	8	2	DOUBLE	VBT	-	-	24	1	-	-
4	IfHK	8	2	DOUBLE	FVBT	-	-	24	1	-	-
5	IfHK	10	2	DOUBLE	VAT	-	-	6	1	-	-
6	IfHK	12	2	DOUBLE	VOR	-20	50	-	1	-	-
7	IfHK	14	2	DOUBLE	VOR	-20	50	-	1	-	-
8	IfStatus	0	1	BIN	BF	3	3	-	-	0	1
9	IfStatus	0	1	BIN	BF	3	3	-	-	1	1
10	IfStatus	0	1	BIN	BF	4	4	-	-	0	1
11	IfStatus	0	1	BIN	BF	4	4	-	-	1	1
12	IfStatus	0	1	BIN	BF	5	5	-	-	0	1
13	IfStatus	0	1	BIN	BF	5	5	-	-	1	1
14	IfStatus	1	1	BIN	BF	0	0	-	-	0	1
15	IfStatus	1	1	BIN	BF	0	0	-	-	1	1
16	IfStatus	1	1	BIN	BF	1	1	-	-	0	1
17	IfStatus	1	1	BIN	BF	1	1	-	-	1	1
18	IfStatus	1	1	BIN	BF	2	2	-	-	0	1
19	IfStatus	1	1	BIN	BF	2	2	-	-	1	1
20	IfStatus	1	1	BIN	BF	3	3	-	-	0	1
21	IfStatus	1	1	BIN	BF	3	3	-	-	1	1
22	IfStatus	1	1	BIN	BF	4	4	-	-	0	1
23	IfStatus	1	1	BIN	BF	4	4	-	-	1	1

Note: a ″-″ is used for parameters not required to configure a mutation operator.

by means of the [VOR, FVOR] pair.

For ***signal values***, depending on the shape of the expected signal, engineers should configure one operator among the ASA, SS, and HV. The configuration of more than one of these operators may lead to redundant mutants (e.g., because each of them triggers the same warning in the SUT).

With ***categorical data*** represented using *integers and hexadecimals*, engineers must configure one IV operator for each possible value; indeed, a change in the observed category shall trigger a different behaviour in the SUT. With categorical data in *binary form*, each bit indicates a specific class (e.g., the unit in error for the DataItem2 in the IFStatus message of Figure 2.8). To verify that the test suite can detect any possible category change, engineers must configure two BF operators for every bit (both MIN and MAX must coincide with the bit position), one operator must flip a bit when it is set (i.e., $STATE = 1$), and the last one when it is unset (i.e., $STATE = 0$).

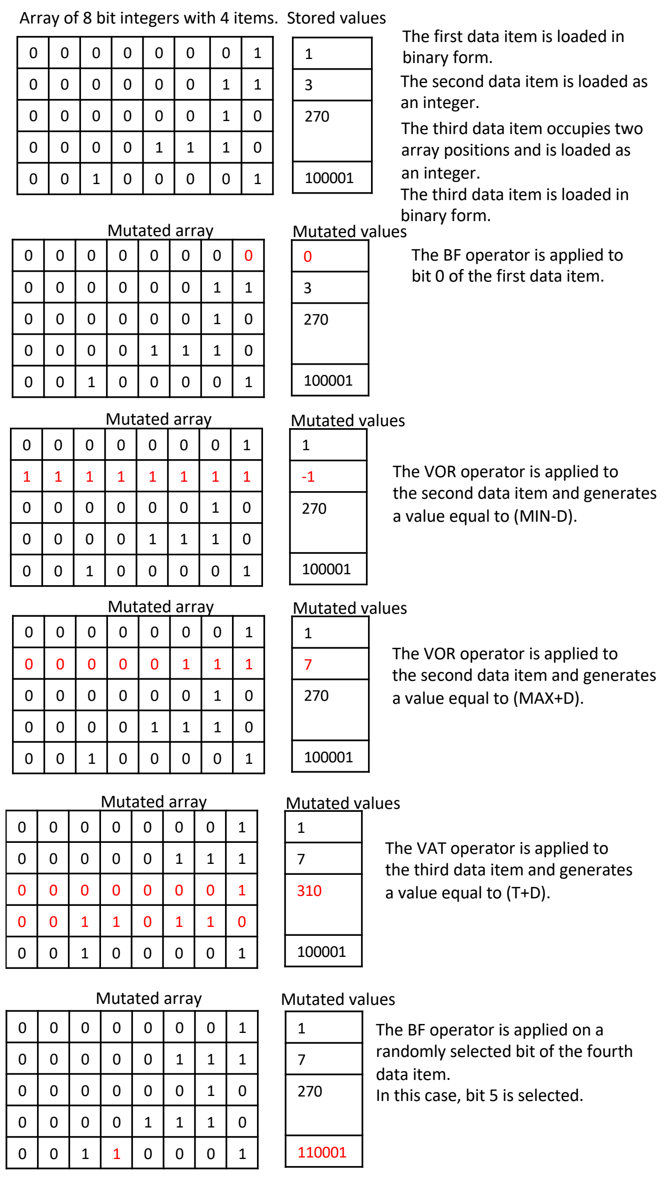
For ***ordinal data***, which is represented by means of either integers or hexadecimals, we suggest to apply the ASA operator with T being set to the middle point of the ordinal scale and Δset to the step distance between consecutive data (usually 1). For data in binary form (e.g., pictures), engineers must configure a BF operator to flip a number of bits that is sufficient to alter the semantics of the data (e.g., introduce sufficient noise in images).

Table 2.5 provides a specification in tabular form (i.e, the format processed by DAMAt) of two fault models configured for the IfKH (i.e., Interface House Keeping) and IfStatus (i.e, Interface Status) messages. In the fault models, each row captures the configuration of a mutation operator for a specific data item. For example, row number 5 indicates that DAMAt interprets as double the data inside the two buffer units starting at position 10 (units 10 and 11) and applies the VAT operator. Rows 1 and 2 show that, for a same numerical data item (i.e., the one covering units 8 and 9), we can apply both the VAT and VBT operators, using a different delta for each. Rows 2 and 4 show the FVAT and FVBT operators complementing the VAT and VBT operators in rows 1 and 3. They simulate the case in which data for the nominal cases is observed instead of data for exceptional cases, as visible in Table 2.2. Rows 8 to 23 show that different bits of a same data item can be targeted by different BF operators. Rows 8 to 13 concern binary categorical data with two categories each, thus we configured two BF each. Rows 14 to 23 concern binary categorical data with five categories; consequently, they present ten BF operators configured for the five categories.

Figure 2.9 provides a visual representation of an array of 8 bit unsigned integers (i.e., unsigned chars) that is modelled using the ***FMExample*** fault model in Table 2.6. It also provides an example of the mutated data generated by the six mutation operation instances derived from the fault model in Table 2.6.

Fault Model	Position	Span	Type	Op	MIN	MAX	T	DELTA	STATE	VALUE
IfHK	0	1	BIN	BF	0	0	-	-	-	-
IfHK	1	1	INT	VOR	0	5	-	1	-	-
IfHK	2	2	INT	VAT	-	-	300	10	-	-
IfHK	4	1	BIN	BF	0	6	-	-	-	-
IfStatus	0	1	BIN	BF	0	0	-	-	-	-

Table 2.6: Example of a data-driven fault model specified in a (CSV) table.



Array of 8 bit integers with 4 items. Stored values

0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	1	3
0	0	0	0	0	0	1	0	270
0	0	0	0	1	1	1	0	
0	0	1	0	0	0	0	1	100001

The first data item is loaded in binary form.
The second data item is loaded as an integer.
The third data item occupies two array positions and is loaded as an integer.
The third data item is loaded in binary form.

Mutated array Mutated values

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	3
0	0	0	0	0	0	1	0	270
0	0	0	0	1	1	1	0	
0	0	1	0	0	0	0	1	100001

The BF operator is applied to bit 0 of the first data item.

Mutated array Mutated values

0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	-1
0	0	0	0	0	0	1	0	270
0	0	0	0	1	1	1	0	
0	0	1	0	0	0	0	1	100001

The VOR operator is applied to the second data item and generates a value equal to (MIN-D).

Mutated array Mutated values

0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	1	7
0	0	0	0	0	0	1	0	270
0	0	0	0	1	1	1	0	
0	0	1	0	0	0	0	1	100001

The VOR operator is applied to the second data item and generates a value equal to (MAX+D).

Mutated array Mutated values

0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	1	7
0	0	0	0	0	0	0	1	310
0	0	1	1	0	1	1	0	
0	0	1	0	0	0	0	1	100001

The VAT operator is applied to the third data item and generates a value equal to (T+D).

Mutated array Mutated values

0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	1	7
0	0	0	0	0	0	1	0	270
0	0	0	0	1	1	1	0	
0	0	1	1	0	0	0	1	110001

The BF operator is applied on a randomly selected bit of the fourth data item.
In this case, bit 5 is selected.

Figure 2.9: Example of original data and data mutated according to the fault model in Table 2.6.

```
switch(message_type)
  case IfStatus:
    GetIfStatus(buffer);
    mutate_FM_IfStatus( buffer);
... break;
```

```
  ...
    case IfHouseKeeping:
      GetIfHouseKeeping(buffer);
      mutate_FM_IfHK( buffer );
      break;
```

Figure 2.10: Example of DAMAt mutation probes (in bold).

2.3.1.3 Automated Generation of Mutation API (Step 2) and Probe Insertion (Step 3)

DAMAt automatically generates a ***mutation API*** to perform mutations at runtime. The API implements a set of functions (called ***mutate_FM_<name>***) that mutate a data buffer according to the given fault model. These functions select the data item to mutate and the mutation procedure to apply based on the mutant under test (see Section 2.3.1.4).

The DAMAt mutation API works with C/C++ code; however it may be extended to deal with other programming languages. Since it is not possible to automatically determine which data buffer to mutate, DAMAt requires engineers to modify the source code of the CPS under test by introducing a mutation probe which consists of an invocation of the DAMAt function that mutates the data buffer according to a specific fault model. Note that 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. Figure 2.10 shows how the implementation of ESAIL has been modified to add the mutation probes. The SVF function was modified to handle the message requests sent to the ADCS by inserting one mutation probe for each message type to mutate, e.g., IfStatus and IfHouseKeeping in Figure 2.10. Function *mutate_FM_IfStatus* is part of the generated mutation API; it loads the fault model *IfStatus* into memory (our API relies on a tree data structure) and then invokes the function *mutate*. The function *mutate* performs data-driven mutation according to the provided fault model; the implementation of *mutate* is part of the DAMAt toolset.

The behavior of function *mutate* depends on the value of a unique identifier (i.e., the *MutantID*) associated at compile time to the mutant; the *Mutant ID* univocally identifies the performed mutation operation (each mutant executes one mutation operation, see Section 2.3.1.4). At a high level, *mutate* performs four activities. First, it checks if the mutation should be performed (i.e., if the data buffer is targeted by the mutation operation identified with the *Mutant ID*). Second, it casts the data item instance targeted by the mutant to a support variable of the type specified in the fault model. Third, it mutates the data stored in the support variable; for each mutation operator, we have implemented a distinct set of instructions for each data representation type. Fourth, before terminating, the function *mutate* writes the mutated data back to the data buffer.

2.3.1.4 Automated Generation of Mutants (Step 4)

Consistent with code-driven mutation analysis, DAMAt generates one mutant for each mutation procedure of the mutation operators configured in the fault model. Each mutant performs exactly one ***data mutation operation*** (i.e., a data mutation procedure configured for a specific data item). For example, the specification in row 6 of Table 2.5 makes DAMAt generate two mutants: each mutant modifies the value of the data item starting at position 12 but one mutant replaces the current value with the value 51 (i.e., $50 + 1$) while the other replaces the current value with the value -21 (i.e., $-20 - 1$).

The mutant generation is invisible to the end-user who does not need to modify the source code further; indeed, we rely on a C macro to specify, at compile time, which mutation operation must be performed by every mutant. Mutants are generated by compiling the SUT multiple times, once for each mutation operation. At runtime, the mutate function executes only the mutation operation selected for the mutant under test.

2.3.1.5 Mutants Execution (Step 5)

As for ***code-driven mutation analysis***, the test suite under analysis is executed iteratively with every data-driven mutant. At runtime, all the data items targeted by a mutant are mutated whenever the mutation preconditions hold (e.g., the STATE of the BF operator); we leave the mutation of a sampled subset of data item instances to future work [**?**, **?**].

To speed up the mutation analysis process, the test suite under analysis is first executed with a special mutant that, instead of mutating data items, keeps trace of the fault models loaded by each test case; in other words, it traces what are the data types covered by each test case. The collected information 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.

2.3.1.6 Mutation Analysis Results (Step 6)

Inspired by work on ***abstract mutation analysis*** [**?**], 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. 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. Since different component functionalities often require different message types, 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; for example, the FVOR operator leads to two mutation operations, which are applied only if the observed value is outside range. Otherwise the two mutation operations will not be covered, thus enabling the engineer to identify such shortcoming in the test suite.

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).

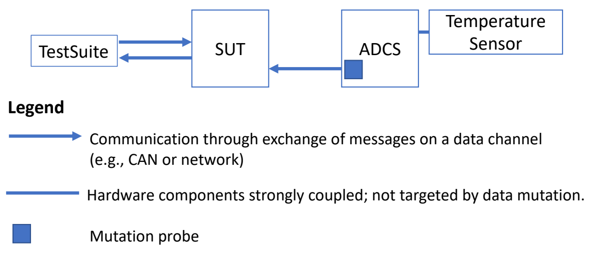

TestSuite
SUT
ADCS
Temperature Sensor
Legend
Communication through exchange of messages on a data channel (e.g., CAN or network)
Hardware components strongly coupled; not targeted by data mutation.
Mutation probe

Figure 2.11: Architecture of the running example system.

2.3.2 Running Examples

In this Section, we provide a set of running examples that show the results achieved by DAMAt in the presence of test suites affected by different problems. More precisely we aim to demonstrate, for representative cases, the absence of false alarms due to equivalent mutants and the usefulness of the mutation analysis metrics computed. Each running example is an artificial but realistic definition of test suites to illustrate the methodology. P-3
We consider the following representative scenarios:

1. The test suite exercises only one message exchange between the ADCS and the SUT; each test case covers a distinct input partition.

2. The test suite exercises only one message exchange between the ADCS and the SUT; multiple test cases cover a same input partition.

3. The test suite exercises multiple message exchanges between the ADCS and the SUT; each test case covers a distinct combination of input partitions.

2.3.2.1 Example set 1: One message exchange, distinct input partitions

Our first set of running examples concerns the presence of a single message exchange and test cases covering distinct input partitions.

Our examples concern an SUT that exchanges with the ADCS component one type of message. More precisely, the ADCS sends a message (hereafter, *TempMessage*) with the temperature collected by its sensor. The architecture of such system is shown in Figure 2.11. For simplicity, we assume that the ADCS periodically sends a TempMessage to the SUT. The ADCS is connected to the TemperatureSensor. The communication between the ADCS and the TemperatureSensor is not affected by data-driven mutation. We may assume the ADCS and the TemepratureSensor are simulated during the execution of the test suite. The SUT and the ADCS communicate through a channel (e.g., a CAN bus). The data mutation probe is inserted in the ADCS simulator; in practice we mutate the data generated by the ADCS. The test suite exercises the software under test by communicating through a data channel; the type of communication channel used by the test suite is not relevant for the purpose of the running example.

For our running example, the SUT has two state variables that are observable by the test suite, *temperature* and *temperature_alarm*. The state variable *temperature* reports the temperature returned by the sensor in the last message. The state variable *temperature_alarm* is set to 1 if the temperature returned by the sensor is above 100.

The test suite is comprised of two test cases. *Test 1* exercises the SUT with a temperature in the nominal case (i.e., temperature below 100), *Test 2* concerns the non nominal case (i.e., temperature above 100). The two test cases of the test suite exercise the same sequence of interactions, which are depicted in the UML Sequence diagram of Figure 2.12. The sequence diagram shows that the test case first starts the ADCS simulator and the SUT; then, it waits for the ADCS to send the TempMessage before requesting the values of

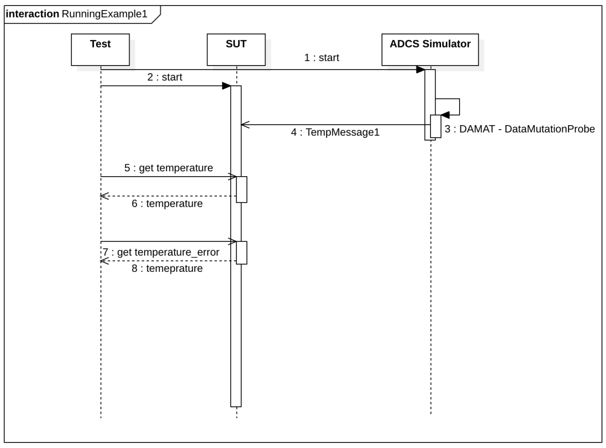


Figure 2.12: Interactions exercised by the test cases belonging to the running example set 1.

the variables *temperature* and *temperature_alarm*. Figure 2.12 also shows that, in this case, the mutation is performed within the ADCS simulator.

Figures 2.13 to 2.15 show our running examples. On the top, we report the fault model. In this case the fault model applies the operators VAT (value above threshold) and FVAT (fix value above threshold). The VAT operator is configured with a threshold of 100 and a delta of 10 (i.e., it replaces values below the threshold with the value 110). The FVAT operator is configured in the same manner (i.e., it replaces values above threshold with the value 90). The operator VAT leads to *MUTANT 1*, the operator FVAT leads to *MUTANT 2*; P-5 to summarize, in this example we have two mutants, each implements one distinct mutation operator.

In Figure 2.13, under *Exchanged data*, we report the sequence of data messages exchanged by the test cases of the test suite (one column for each test case). In the following rows, we report the oracles. For each oracle we indicate how it verifies if a state variable has been assigned with a correct value; we indicate " == " if it verifies the exact value taken by the state variable (e.g., *temperature == 50)*, ">=" if it verifies that the value is above a threshold (e.g., *temperature* >= *50*). Please note that, according to standard practice, test cases for non nominal cases verify that state variables contain the expected alarm and anomalous values (i.e., the test case passes only if the SUT report the anomaly).

For each mutant, we report the value of the data exchanged during the execution and the data of the state variables read by the oracle. In red we indicate data that has been mutated according to the fault model. If an oracle does not read the value of a variable we leave the cell empty. Failing oracles are highlighted in yellow. P-5 Note that for each mutant we provide multiple columns (named *Test 1* and *Test 2*), each provides the data exchanged during the execution of the test case. Finally, for each mutant, for each oracle, we report the test cases that detect the anomalous value (if any); also, we report if the mutant had been killed.

Please note that, following the DAMAt procedures, MUTANT 1 does not mutate the data exchanged by Test 2 because it is already above the threshold. MUTANT 2 does not mutate the data exchanged by Test 1 because it is already below the threshold.

TestSuite1 is an optimal test suite that verifies the expected value of all the state variables. For MUTANT 1, Test 1 fails because the assertion about temperature fails (expected 50 observed 110). For MUTANT 2, P-6 Test 2 fails because the assertion about temperature fails (expected 120 observed 90). We have a fault model coverage (*FMC*) of 100% because we have only one fault model and it is covered (i.e., the test suite exchanged the data message under analysis). We have a mutation operator coverage (*MOC*) of 100% because all the mutation operations associated with the configured mutation operators are applied (in other words, all the mutants perform at least one mutation of a data item instance). We have a mutation score (*MS*) of 100% because, for each mutant, at least one test case fails.

TestSuite2 verifies the expected value of the temperature for the nominal case and the presence of a temperature error for the non nominal case. For MUTANT 1, Test 1 fails because the assertion on temperature fails (expected 50 observed 110). For MUTANT 2, Test 2 fails because the assertion on temperature errors fail (expected 1 observed 0). Concerning FMC, MOC, and MS, the same considerations made for TestSuite1 hold. P-6

TestSuite3 verifies only temperature values while TestSuite4 verifies only temperature alarms. They all kill the mutants for the same reasons reported in the cases above. Concerning FMC, MOC, and MS, the same considerations made for TestSuite1 hold. P-6

TestSuite5 is a copy of TestSuite2 having an imprecise oracle (i.e., it verifies that temperature is above 50 instead of equal to 50). Consequently the test suite does not fail for MUTANT 1 and the mutation score is not 100%; consequently, DAMAt enables an engineer to detect the limitation of the test suite. P-6

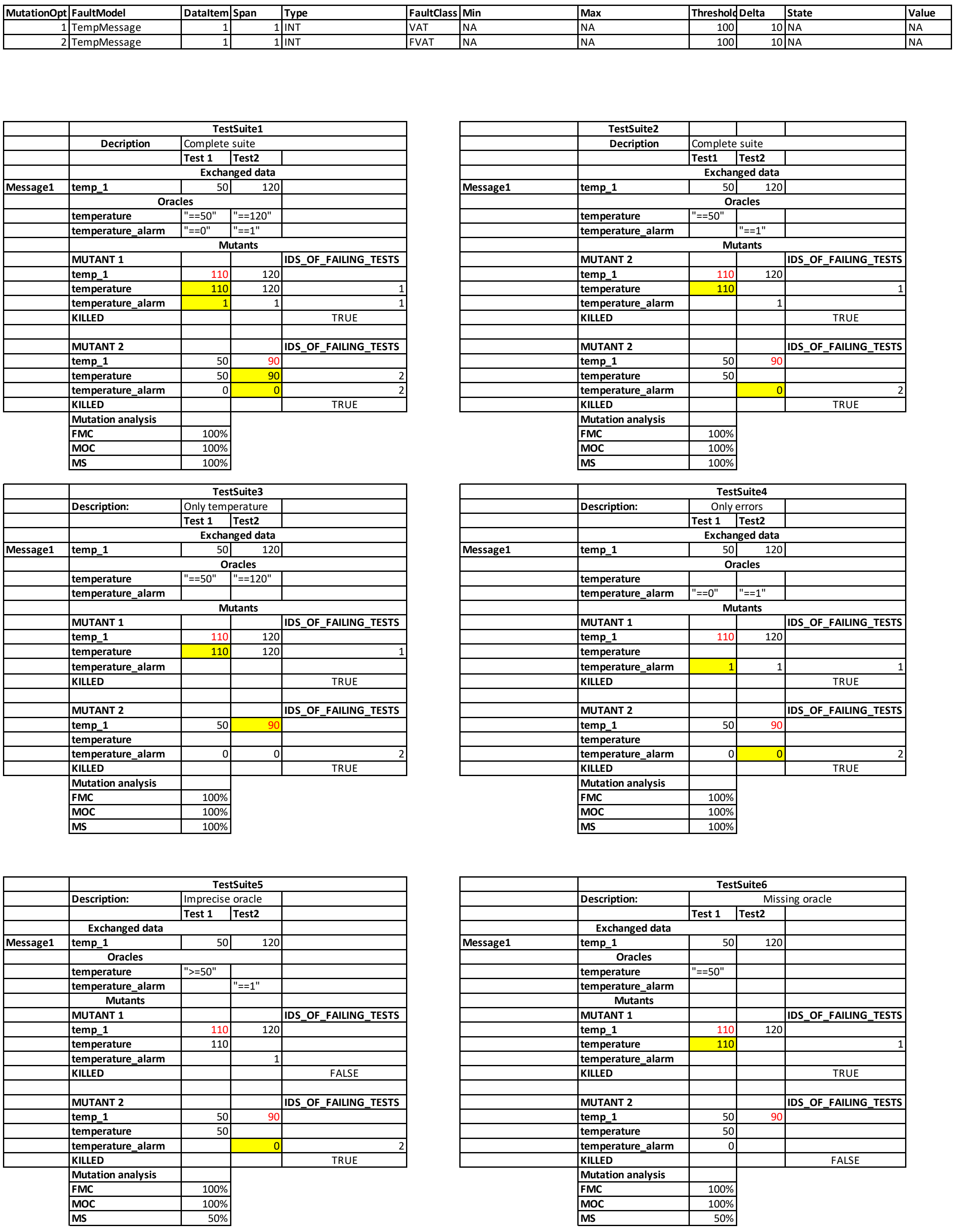
TestSuite6 is a copy of TestSuite2 without an oracle for Test2 (i.e., it does not verify the presence of temperature alarms). Consequently, the test suite does not fail for MUTANT 2 and the mutation score is not 100%; consequently, DAMAt enables an engineer to detect the limitation of the test suite.

TestSuite7 is a copy of TestSuite2 with multiple test cases covering the same input partitions; indeed, Test3 covers the same input partition of Test1 (i.e., it exercises the value 0, which is a nominal value like 50), Test4 covers the same input partition of Test2 (i.e., it exercises the value 101, which is a non nominal value like 120). All the mutants are killed by the test suite and we do not observe equivalent mutants.

TestSuite8 is a copy of TestSuite7 with the same limitations of TestSuite6 (i.e., Test2 does not contain an oracle). In this case, the mutation score is 100% because Test 4 contains an appropriate oracle and covers the same input partition not appropriately verified by Test 2. We do not observe equivalent mutants.

TestSuite9 is a copy of TestSuite7 without oracles for both Test2 and Test4; similarly to the case of TestSuite6, since the non-nominal partition is not appropriately verified, the mutation score is not 100%; consequently, DAMAt enables an engineer to detect the limitation of the test suite.

TestSuite10 covers the case of a test suite that does not exercise an input partition; precisely, it does not exercise non nominal values. In this case, DAMAt can detect that MUTANT 2 is never applied (to exemplify it, we do not report any red value for MUTANT 2 in Figure 2.15); consequently, the MOC is not 100% and the engineer can determine what is the limitation of the test suite. Indeed, since the mutant not covered is FVAT, the engineer can know that the test suite does not exercise the value above threshold for the temperature (FVAT applies the mutation only if a value above threshold is observed).



MutationOpt	FaultModel	DataItem	Span	Type	FaultClass	Min	Max	Threshold	Delta	State	Value
1	TempMessage	1	1	INT	VAT	NA	NA	100	10	NA	NA
2	TempMessage	1	1	INT	FVAT	NA	NA	100	10	NA	NA

	TestSuite1			
	Decription	Complete suite		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	"==50"	"==120"	
	temperature_alarm	"==0"	"==1"	
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature	110	120	1
	temperature_alarm	1	1	1
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature	50	90	2
	temperature_alarm	0	0	2
	KILLED			TRUE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	100%		

	TestSuite2			
	Decription	Complete suite		
		Test1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	"==50"		
	temperature_alarm		"==1"	
	Mutants			
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature	110		1
	temperature_alarm		1	
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature	50		
	temperature_alarm		0	2
	KILLED			TRUE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	100%		

	TestSuite3			
	Description:	Only temperature		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	"==50"	"==120"	
	temperature_alarm			
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature	110	120	1
	temperature_alarm			
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature			
	temperature_alarm	0	0	2
	KILLED			TRUE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	100%		

	TestSuite4			
	Description:	Only errors		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature			
	temperature_alarm	"==0"	"==1"	
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature			
	temperature_alarm	1	1	1
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature			
	temperature_alarm	0	0	2
	KILLED			TRUE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	100%		

	TestSuite5			
	Description:	Imprecise oracle		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	">=50"		
	temperature_alarm		"==1"	
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature	110		
	temperature_alarm		1	
	KILLED			FALSE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature	50		
	temperature_alarm		0	2
	KILLED			TRUE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	50%		

	TestSuite6			
	Description:	Missing oracle		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	"==50"		
	temperature_alarm			
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature	110		1
	temperature_alarm			
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature	50		
	temperature_alarm	0		
	KILLED			FALSE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	50%		

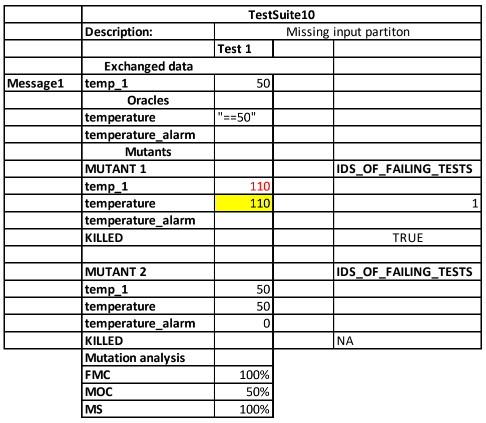
Figure 2.13: Running example set 1 - Part A.

	TestSuite7					
	Description:	Missing oracle				
		Test 1	Test2	Test 3	Test4	
	Exchanged data					
Message1	temp_1	50	120	0	101	
	Oracles					
	temperature	"==50"		"==0"		
	temperature_alarm		"==1"		"==1"	
	Mutants					
	MUTANT 1					IDS_OF_FAILING_TESTS
	temp_1	110	120	110	101	
	temperature	110		110		1,3
	temperature_alarm		1		1	
	KILLED					TRUE
	MUTANT 2					IDS_OF_FAILING_TESTS
	temp_1	50	90	0	90	
	temperature	50		0		
	temperature_alarm		0		0	2,4
	KILLED					TRUE
	Mutation analysis					
	FMC	100%				
	MOC	100%				
	MS	100%				

	TestSuite8					
	Description:	Missing oracle				
		Test 1	Test2	Test 3	Test4	
	Exchanged data					
Message1	temp_1	50	120	0	101	
	Oracles					
	temperature	"==50"		"==0"		
	temperature_alarm				"==1"	
	Mutants					
	MUTANT 1					IDS_OF_FAILING_TESTS
	temp_1	110	120	110	101	
	temperature	110		110		1,3
	temperature_alarm				1	
	KILLED					TRUE
	MUTANT 2					IDS_OF_FAILING_TESTS
	temp_1	50	90	0	90	
	temperature	50		0		
	temperature_alarm				0	4
	KILLED					TRUE
	Mutation analysis					
	FMC	100%				
	MOC	100%				
	MS	50%				

	TestSuite9					
	Description:	Missing oracle				
		Test 1	Test2	Test 3	Test4	
	Exchanged data					
Message1	temp_1	50	120	0	101	
	Oracles					
	temperature	"==50"		"==0"		
	temperature_alarm					
	Mutants					
	MUTANT 1					IDS_OF_FAILING_TESTS
	temp_1	110	120	110	101	
	temperature	110		110		1,3
	temperature_alarm					
	KILLED					TRUE
	MUTANT 2					IDS_OF_FAILING_TESTS
	temp_1	50	90	0	90	
	temperature	50		0		
	temperature_alarm					
	KILLED					FALSE
	Mutation analysis					
	FMC	100%				
	MOC	100%				
	MS	50%				

Figure 2.14: Running example set 1 - Part B.



	TestSuite10			
	Description:	Missing input partiton		
		Test 1		
	Exchanged data			
Message1	**temp_1**	50		
	Oracles			
	temperature	"==50"		
	temperature_alarm			
	Mutants			
	MUTANT 1			**IDS_OF_FAILING_TESTS**
	temp_1	110		
	temperature	110		1
	temperature_alarm			
	KILLED			TRUE
	MUTANT 2			**IDS_OF_FAILING_TESTS**
	temp_1	50		
	temperature	50		
	temperature_alarm	0		
	KILLED			NA
	Mutation analysis			
	FMC	100%		
	MOC	50%		
	MS	100%		

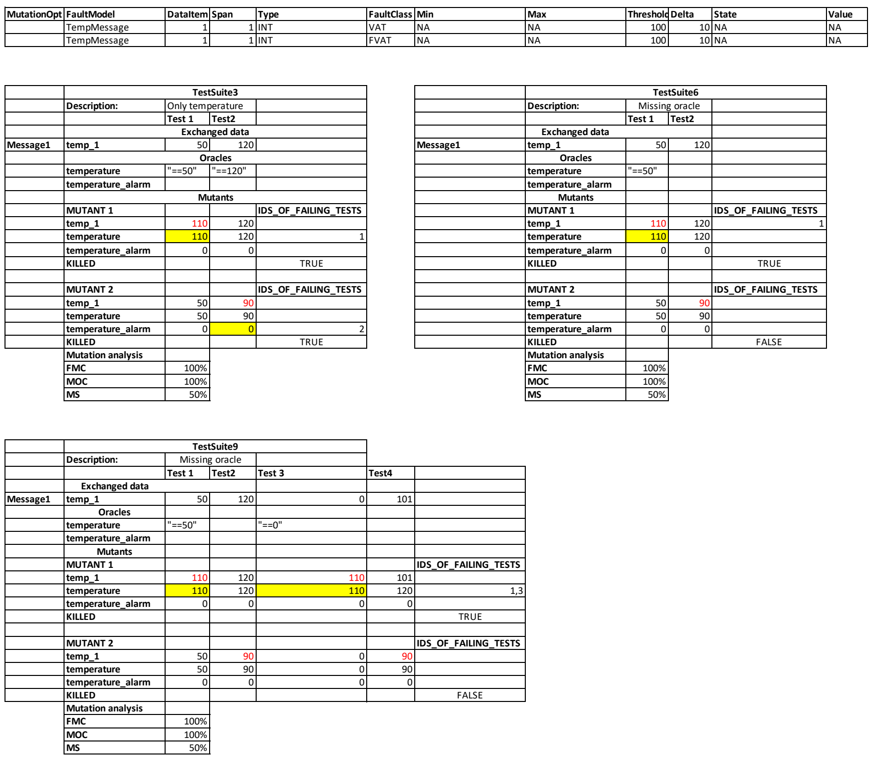
Figure 2.15: Running example set 1 - Part C.

2.3.2.2 Example set 2: One message exchange, distinct input partitions, faulty software

This running example (see Figure 2.16) covers the same case of Section 2.3.2.1, with the difference that we assume the software to be faulty. More precisely, we assume that the value of *temperature alarm* is always set to 0. We ignore the test suites number 1, 2, 4, 5, 6, 7, 8 because they would detect the presence of the fault before mutation analysis (i.e., the oracles "==1" would fail). For completeness, in Figure 2.16, we report the values of state variables also when they are not verified by oracles. We highlight failing oracles in yellow.

For TestSuite6 and TestSuite9, which do not detect the fault because they lack an oracle for the faulty variable, DAMAt would indicate that MUTANT 2 is not killed thus enabling the engineer to introduce an appropriate oracle (i.e., "temperature_alarm == 1") and thus discover the fault.

For TestSuite3, DAMAt would not help the engineer in detecting the fault because the test suite kills the mutant. What we observe in this case is a sort of masking effect due to the fact that two outputs are affected by the same data; indeed, DAMAt ensures that the effect of data mutation is propagated to at least one software output but it cannot verify that the effects of data mutation are propagated to all the software outputs that depend on the mutated data.



MutationOpt	FaultModel	DataItem	Span	Type	FaultClass	Min	Max	Threshold	Delta	State	Value
	TempMessage	1	1	INT	VAT	NA	NA	100	10	NA	NA
	TempMessage	1	1	INT	FVAT	NA	NA	100	10	NA	NA

	TestSuite3			
	Description:	Only temperature		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	"==50"	"==120"	
	temperature_alarm			
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	
	temperature	110	120	1
	temperature_alarm	0	0	
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature	50	90	
	temperature_alarm	0	0	2
	KILLED			TRUE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	50%		

	TestSuite6			
	Description:	Missing oracle		
		Test 1	Test2	
	Exchanged data			
Message1	temp_1	50	120	
	Oracles			
	temperature	"==50"		
	temperature_alarm			
	Mutants			
	MUTANT 1			IDS_OF_FAILING_TESTS
	temp_1	110	120	1
	temperature	110	120	
	temperature_alarm	0	0	
	KILLED			TRUE
	MUTANT 2			IDS_OF_FAILING_TESTS
	temp_1	50	90	
	temperature	50	90	
	temperature_alarm	0	0	
	KILLED			FALSE
	Mutation analysis			
	FMC	100%		
	MOC	100%		
	MS	50%		

	TestSuite9					
	Description:	Missing oracle				
		Test 1	Test2	Test 3	Test4	
	Exchanged data					
Message1	temp_1	50	120	0	101	
	Oracles					
	temperature	"==50"		"==0"		
	temperature_alarm					
	Mutants					
	MUTANT 1					IDS_OF_FAILING_TESTS
	temp_1	110	120	110	101	
	temperature	110	120	110	120	1,3
	temperature_alarm	0	0	0	0	
	KILLED					TRUE
	MUTANT 2					IDS_OF_FAILING_TESTS
	temp_1	50	90	0	90	
	temperature	50	90	0	90	
	temperature_alarm	0	0	0	0	
	KILLED					FALSE
	Mutation analysis					
	FMC	100%				
	MOC	100%				
	MS	50%				

Figure 2.16: Running example set 2.

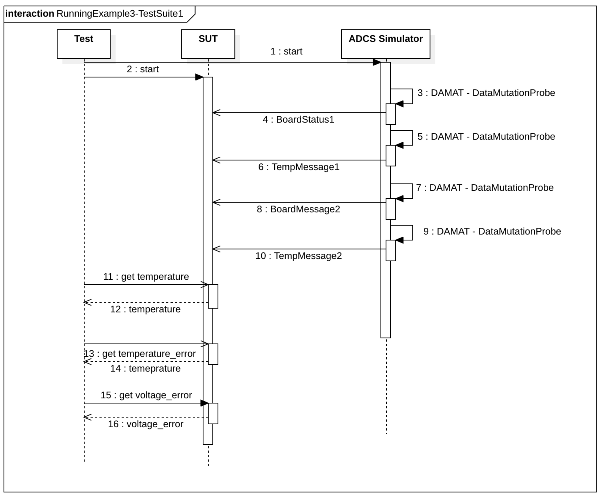


Figure 2.17: Interactions exercised by the test cases of TestSuite1 in the running example set 3.

2.3.2.3 Example set 3: Multiple message exchanges, distinct input partitions

For the third running example set, we consider a system that has the same architecture of Figure 2.11 but exchanges also messages of type *BoardStatus*. We assume that *BoardStatus* messages indicate the voltage of the board and the sensors. The SUT has an additional output state variable called *voltage_error*. A voltage error occurs when the voltage is out of the range (10;14). In the presence of a voltage error, the software shall not update the value of the *temperature* state variable.

Below we discuss two possible test suites, TestSuite1 and TestSuite2, which are affected by different type of limitations. More precisely, we rely on TestSuite1 to (1) show how DAMAt spot a test suite shortcoming difficult to determine manually and (2) demonstrate that DAMAt may unlikely lead to equivalent mutants (i.e., by showing that if a mutant is not killed it's because relevant assertions are missing). We rely on TestSuite2 to exemplify the case of lack of coverage of a fault model.

2.3.2.3.1 TestSuite1 Figure 2.17 shows the interactions exercised by the test cases in the test suite named TestSuite1 for the running example set 3. Each test case, after starting the ADCS simulator and SUT, waits till the ADCS has sent the following sequence of messages: one BoardStatus message, one TempMessage, one BoardStatus message, and one TempMessage. Then it requests and verifies the values of the state variables *temperature*, *temperature_alarm*, and *voltage_error*.

Figures 2.18 and 2.19 show the data exchanged in our running example. With respect to the example set 1, the fault model includes also one VOR and FVOR operator to mutate the BoardStatus message. In Figures 2.18 and 2.19, in the row named *PASS*, we explicitly indicate the result of each test case (i.e., PASS or FAIL).

TestSuite1 covers all the possible combinations of values for the sequence of messages reporting about messages voltage error absent (true or false), temperature alarm absent (true or false), voltage error absent (true or false), temperature alarm absent (true or false); indeed, in total, we have 16 test cases.

TestSuite1 kills MUTANT 1. MUTANT 1 applies VAT, that is, it sets the temperature value above the threshold when it is below the threshold. To not kill MUTANT 1, the test suite should not include any failing oracle. Since the failing oracles include the complete set of oracles that either verify the temperature being in the nominal range or verify the absence of a temperature alarm, we conclude that whenever MUTANT 1 is not killed, we cannot be in the presence of an equivalent mutant but in the presence of a test suite limitation.

TestSuite1 kills MUTANT 2. MUTANT 2 applies FVAT, that is, it sets the temperature value below the threshold when it is above the threshold. To not kill MUTANT 2, the test suite shall not include any of the failing oracles. Since the failing oracles include the complete set of oracles that either verify the temperature being out of the nominal range or verify the presence of a temperature alarm, we conclude that whenever MUTANT 2 is not killed, we cannot be in the presence of an equivalent mutant but in the presence of a test suite limitation.

TestSuite1 kills MUTANT 3. MUTANT 3 applies the first mutation procedure of VOR, that is, it sets the value above range if it is within range. To not kill MUTANT 2, the test suite shall not include any of the failing oracles. Since the failing oracles include the complete set of oracles that either verify the temperature being in the nominal range or verify the absence of a temperature alarm, we conclude that whenever MUTANT 3 is not killed, we cannot be in the presence of an equivalent mutant but in the presence of a test suite limitation.

TestSuite1 kills MUTANT 4. MUTANT 4 applies the second mutation procedure of VOR, that is, it sets the value below range if it is in range. MUTANT 4 leads to the same system outputs as MUTANT 3 thus we can make the same conclusion (no equivalent mutant possible).

TestSuite1 kills MUTANT 5. MUTANT 5 applies the first mutation procedure of FVOR, that is, it sets the value within range if it is above range. To not kill MUTANT 5, the test suite shall not include any of the failing oracles. All the failing oracles belong to test cases that verify a scenario in which a voltage error is present just before the last temperature message is collected; consequently, the lack of such oracles would indicate a major limitation of the test suites (i.e., it would not test if the software identifies the error condition simulated by the scenario under test). Similarly, MUTANT 5 is killed if all such test cases would be missing; even in this case we would be in the presence of a major test suite limitation (i.e., the test suite does not simulate an important error condition). We thus conclude that MUTANT 5 cannot lead to equivalent mutants.

TestSuite1 does not kill MUTANT 6. MUTANT 6 applies the second mutation procedure of FVOR, that is, it sets the value in range if it is below range. Since no values below range are observed during the execution of the test cases, DAMAt never applies this mutation operation. For this reason the mutation operation for the test suite is 83% (i.e., five out of six mutation operation are covered). Although TestSuite1 seems complete because it tests all the pairwise combinations of the conditions *voltage error absent* and *temperature alarm absent*, DAMAt enables us to determine that when defining the test suite we did not consider the fact that the input partitions for voltage error are three (i.e., voltage within range, voltage above range, and voltage below range) not two (i.e., voltage error absent, voltage error present).

2.3.2.3.2 TestSuite2 TestSuite 2 does not trigger messages of type BoardStatus; its interactions are depicted in Figure 2.20. For the remaining message type (i.e., TempMessage) it covers all the possible combinations of temperature alarms being present/absent for two TempMessages sent in sequence; in total, we have 4 test cases.

The fault model BoardStatus is not covered; therefore the fault model coverage (FMC) is 50%. Since MUTANTS 3 to 6 belong to BoardStatus they are not considered in the computation of the other mutation analysis metrics. MOC and MS are thus 100%; indeed all the mutants not belonging to BoardStatus are killed by the test suite.

MUTANT 1 could be live only if relevant oracles are missing; more precisely, only if oracles concerning the nominal temperature value are missing. MUTANT 1 is killed by TestSuite2.

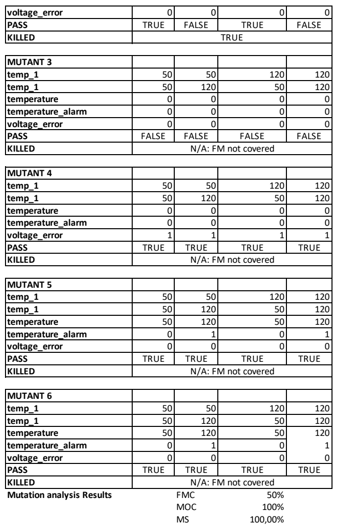
MUTANT 2 could be live only if relevant oracles are missing; indeed, only if oracles concerning the non nominal temperature value are missing. MUTANT 2 is killed by TestSuite2.

	FaultModel	DataItem	Span	Type	FaultClass	Min	Max	Threshold	Delta	State	Value
	TempMessage	1	1	INT	VAT	NA	NA	100	10	NA	NA
	TempMessage	1	1	INT	FVAT	NA	NA	100	10	NA	NA
	BoardStatus	1	1	DOUBLE	VOR	10	13	NA	1	NA	NA
	BoardStatus	1	1	DOUBLE	FVOR	10	13	NA	1	NA	NA

	TestSuite1																
	Decription	Complete suite															
	Requirements coverage																
		Test 1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Test11	Test12	Test13	Test13	Test15	Test16
Message1 (BoardStatus)	Voltage_Error_Absent	T	T	T	T	T	F	F	T	F	T	F	F	T	F	F	F
Message2 (TempMessage)	Temperature_Alarm_Absent	T	T	F	F	T	T	T	F	F	F	F	F	T	T	F	T
Message3 (BoardStatus)	Voltage_Error_Absent	T	T	T	T	F	T	F	F	T	F	T	F	F	T	F	F
Message4 (TempMessage)	Temperature_Alarm_Absent	T	F	T	F	T	T	T	T	T	F	F	F	F	F	T	F
		Test 1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Test11	Test12	Test13	Test13	Test15	Test16
	Exchanged data																
Message1 (BoardStatus)	voltage	12	12	12	12	12	20	20	12	20	12	20	20	12	20	20	20
Message2 (TempMessage)	temp_1	50	50	120	120	50	50	50	120	120	120	120	120	50	50	120	50
Message3 (BoardStatus)	voltage	12	12	12	12	20	12	20	20	12	20	12	20	20	12	20	20
Message4 (TempMessage)	temp_1	50	120	50	120	50	50	50	50	50	120	120	120	120	120	50	120
	Oracles																
	temperature	"==50"	"==120"	"==50"	"==120"	"==50"	"==50"	"==0"	"==120"	"==50"	"==120"	"==120"	"==0"	"==50"	"==120"	"==0"	"==0"
	temperature_alarm	"==0"	"==1"	"==0"	"==1"	"==0"	"==0"	"==0"	"==1"	"==0"	"==1"	"==1"	"==0"	"==0"	"==1"	"==0"	"==0"
	voltage_Error	"==0"	"==0"	"==0"	"==0"	"==1"	"==0"	"==1"	"==1"	"==0"	"==1"	"==0"	"==1"	"==1"	"==0"	"==1"	"==1"
	Outputs original software (for reference)																
	temperature	50	120	50	120	50	50	0	120	50	120	120	0	50	120	0	0
	temperature_alarm	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0	0
	voltage_Error	0	0	0	0	1	0	1	1	0	1	0	1	1	0	1	1
	Mutants																
	MUTANT 1																
	voltage	12	12	12	12	12	20	20	12	20	12	20	20	12	20	20	20
	temp_1	110	120	120	120	110	110	110	120	120	120	120	120	110	110	120	110
	voltage	12	12	12	12	20	12	20	20	12	20	12	20	20	12	20	20
	temp_1	110	120	110	120	110	110	110	110	110	120	120	120	120	120	110	120
	temperature	110	120	110	120	110	110	0	120	110	120	120	0	110	120	110	0
	temperature_alarm	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0
	voltage_error	0	0	0	0	0	0	1	1	0	1	0	1	1	0	1	1
	PASS	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE
	KILLED	TRUE															
	MUTANT 2																
	voltage	12	12	12	12	12	20	20	12	20	12	20	20	12	20	20	20
	temp_1	50	50	90	90	50	50	50	90	90	90	90	90	50	50	90	50
	voltage	12	12	12	12	20	12	20	20	12	20	12	20	20	12	20	20
	temp_1	50	90	50	90	50	50	50	50	50	90	90	90	90	90	50	90
	temperature	50	90	50	90	50	50	0	120	50	90	90	0	50	90	0	0
	temperature_alarm	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	voltage_error	0	0	0	0	1	0	1	1	0	1	0	1	1	0	1	1
	PASS	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
	KILLED	TRUE															
	MUTANT 3																
	voltage	15	15	15	15	15	20	20	15	20	15	20	20	15	20	20	20
	temp_1	50	50	120	120	50	50	50	120	120	120	120	120	50	50	120	50
	voltage	15	15	15	15	20	15	20	20	15	20	15	20	20	15	20	20
	temp_1	50	120	50	120	50	50	50	50	50	120	120	120	120	120	50	120
	temperature	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	temperature_alarm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	voltage_error	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	1
	PASS	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE
	KILLED	TRUE															
	MUTANT 4																
	voltage	9	9	9	9	9	20	20	9	20	9	20	20	9	20	20	20
	temp_1	50	50	120	120	50	50	50	120	120	120	120	120	50	50	120	50
	voltage	9	9	9	9	20	9	20	20	9	20	9	20	20	9	20	20
	temp_1	50	120	50	120	50	50	50	50	50	120	120	120	120	120	50	120
	temperature	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	temperature_alarm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	voltage_error	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	1
	PASS	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE
	KILLED	TRUE															
	MUTANT 5																
	voltage	12	12	12	12	12	13	13	12	13	12	13	13	12	13	13	13
	temp_1	50	50	120	120	50	50	50	120	120	120	120	120	50	50	120	50
	voltage	12	12	12	12	20	12	13	13	12	13	12	13	13	12	13	13
	temp_1	50	120	50	120	50	50	50	50	50	120	120	120	120	120	50	120
	temperature	50	120	50	120	50	50	50	50	50	120	120	120	120	120	0	0
	temperature_alarm	0	1	0	1	0	0	0	1	0	1	1	1	1	1	0	0
	voltage_error	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	PASS	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
	KILLED	TRUE															
	MUTANT 6																
	voltage	12	12	12	12	12	20	20	12	20	12	20	20	12	20	20	20
	temp_1	50	50	120	120	50	50	50	120	120	120	120	120	50	50	120	50
	voltage	12	12	12	12	20	12	20	20	12	20	12	20	20	12	20	20
	temp_1	50	120	50	120	50	50	50	50	50	120	120	120	120	120	50	120
	temperature	50	120	50	120	50	50	0	120	50	120	120	0	50	120	0	0
	temperature_alarm	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0	0
	voltage_error	0	0	0	0	1	0	1	1	0	1	0	1	1	0	1	1
	PASS	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
	KILLED	N/A: mutation opration not covered															
	Mutation analysis Results		FMC	100,00%													
			MOC	83,33%													
			MS	100,00%													

	TestSuite2				
	Decription	Lack of message			
	Requirements coverage				
		Test 1	Test2	Test3	Test4
Message1 (TempMessage)	Temperature_Alarm_Absent	T	T	F	F
Message2 (TempMessage)	Temperature_Alarm_Absent	T	F	T	F
		Test 1	Test2	Test3	Test4
	Exchanged data				
Message1 (TempMessage)	temp_1	50	50	120	120
Message2 (TempMessage)	temp_1	50	120	50	120
	Oracles				
	temperature	"==50"	"==120"	"==50"	"==120"
	temperature_alarm	"==0"	"==1"	"==0"	"==1"
	voltage_Error	"==0"	"==0"	"==0"	"==0"
	Outputs original software (for reference)				
	temperature	50	120	50	120
	temperature_alarm	0	1	0	1
	voltage_Error	0	0	0	0
	Mutants				
	MUTANT 1				
	temp_1	110	120	120	120
	temp_1	110	120	110	120
	temperature	110	120	110	120
	temperature_alarm	1	1	1	1
	voltage_error	0	0	0	0
	PASS	FALSE	TRUE	FALSE	TRUE
	KILLED	TRUE			
	MUTANT 2				
	temp_1	50	50	90	90
	temp_1	50	90	50	90
	temperature	50	90	50	90
	temperature_alarm	0	0	0	0

Figure 2.18: Running example set 3 - Part A.



voltage_error	0	0	0	0
PASS	TRUE	FALSE	TRUE	FALSE
KILLED	TRUE			
MUTANT 3				
temp_1	50	50	120	120
temp_1	50	120	50	120
temperature	0	0	0	0
temperature_alarm	0	0	0	0
voltage_error	0	0	0	0
PASS	FALSE	FALSE	FALSE	FALSE
KILLED	N/A: FM not covered			
MUTANT 4				
temp_1	50	50	120	120
temp_1	50	120	50	120
temperature	0	0	0	0
temperature_alarm	0	0	0	0
voltage_error	1	1	1	1
PASS	TRUE	TRUE	TRUE	TRUE
KILLED	N/A: FM not covered			
MUTANT 5				
temp_1	50	50	120	120
temp_1	50	120	50	120
temperature	50	120	50	120
temperature_alarm	0	1	0	1
voltage_error	0	0	0	0
PASS	TRUE	TRUE	TRUE	TRUE
KILLED	N/A: FM not covered			
MUTANT 6				
temp_1	50	50	120	120
temp_1	50	120	50	120
temperature	50	120	50	120
temperature_alarm	0	1	0	1
voltage_error	0	0	0	0
PASS	TRUE	TRUE	TRUE	TRUE
KILLED	N/A: FM not covered			
Mutation analysis Results		FMC	50%	
		MOC	100%	
		MS	100,00%	

Figure 2.19: Running example set 3 - Part B.

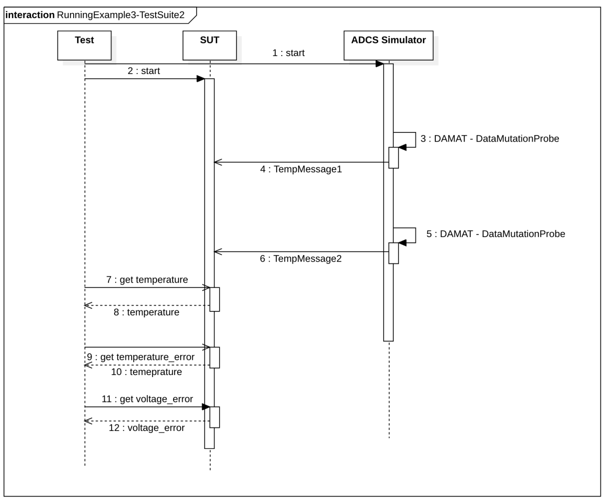


Figure 2.20: Interactions exercised by the test cases of TestSuite2 in the running example set 3.

2.4 Data-driven Mutation Testing: DAMTE

2.4.1 Overview

In this section we describe 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.

The ***test suite augmentation process*** concerns the definition of additional test cases to increase the mutation score. It consists of four activities ***Identify Test Inputs***, ***Generate Test Oracles***, ***Execute the SUT***, ***Fix the SUT***. Despite these activities match the ones performed in the case of code-driven mutation testing, they are triggered and implemented in a different manner, as described below.

In the presence of mutants not killed by test cases (i.e., when the ***mutation score*** is not equal to 100%), engineers are expected to manually investigate the underlying problems. Indeed, as reported in Section 2.3.1.6, two might be the reasons for a low MS: poor oracle quality and missing test input sequences (i.e., the software does not reach the state in which it could kill the mutant). For the first case (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. For the second case, 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 this reason, generating test oracles and fixing the SUT (in case a fault is discovered after test suite augmentation) shall be performed manually.

When mutation operators are not applied because of the lack of appropriate data to mutate (i.e., in the presence of fault model coverage and mutation operation coverage below 100%), engineers are expected to generate new test inputs for the SUT that enable the application of all the mutation operators. However, the methodology to adopt may vary based on the test objective and the system architecture. We discuss the case of the producer-consumer and client-server architecture, two common software architectures. We leave the discussion of other architectures (e.g., broker architecture and event-bus architecture) to future work.

In Figures 2.21 to 2.23, we exemplify the two architectures. In both the two cases, data-driven mutation may concern the generated data and occur either on the component that generates the data (Figure 2.21), or on the component that receives the data (Figure 2.22). For the client-server case, instead, data mutation may concern also the request for data and be performed either on the client or the server (Figure 2.23). For the producer-consumer case, static program analysis may be employed to automatically generate the missing data; to this end, we aim to rely on an ***extended data mutation probe***. For the client-server case, the ***extended data mutation probe*** may still be used but only to generate message requests; therefore, it would be useful only when data-driven analysis is performed on the request message. Since the steps required to perform test generation is the same in both the two cases, we provide an example based on the client-server case.

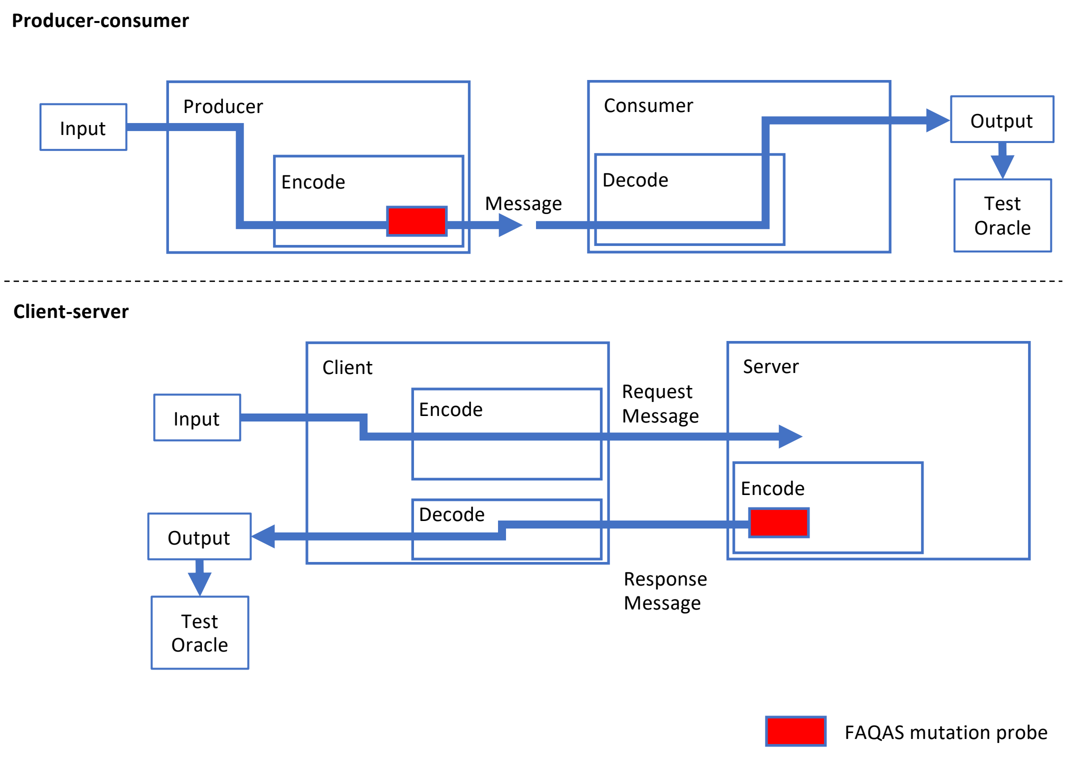


Figure 2.21: Data-driven mutation analysis for different architectures.

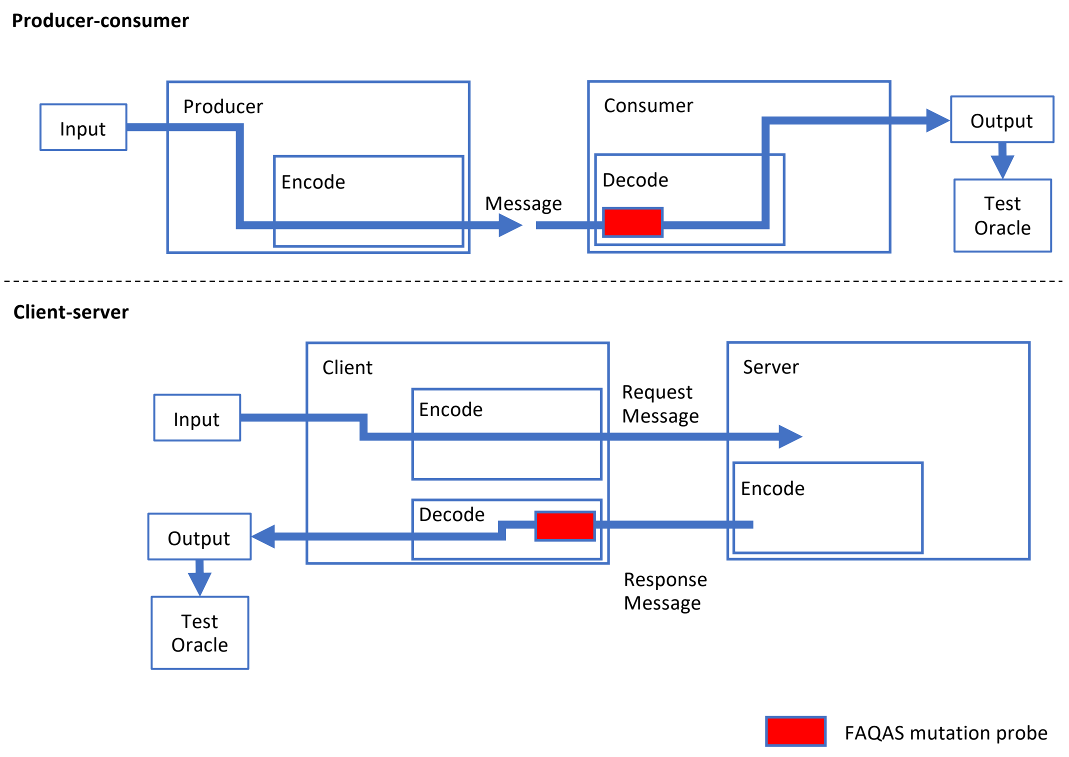


Figure 2.22: Data-driven mutation analysis for different architectures.

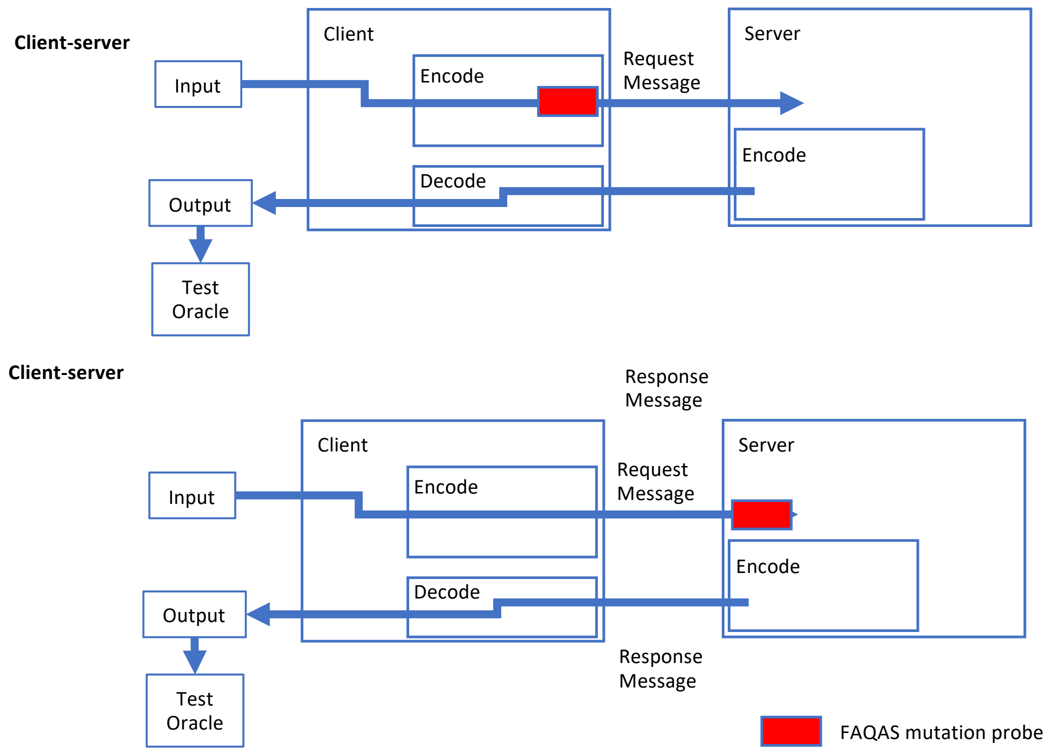


Figure 2.23: Data-driven mutation analysis for different architectures.

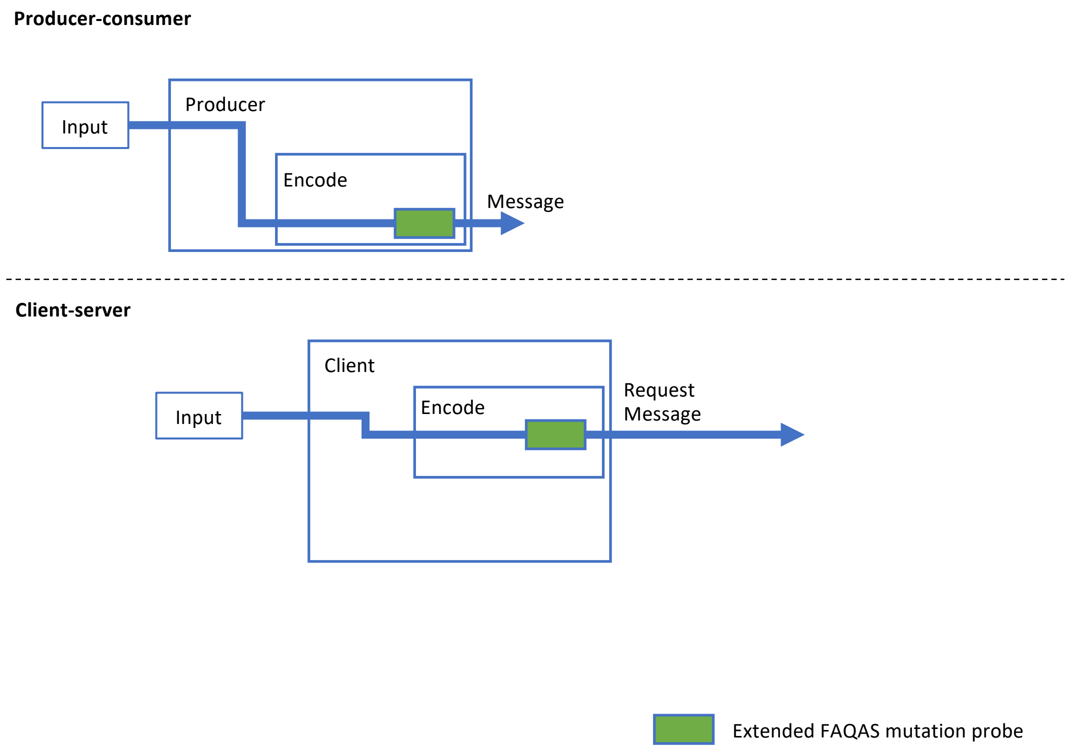


Figure 2.24: Data-driven mutation analysis for different architectures.

2.4.2 Test generation with a client-server system

For our example, we rely on the libParam case study provided by GSL. Listing 2.10 shows the mutation probe, which is inserted into function *gs_rparam_process_packet*, on the server side. The probe mutates the buffer *v_General*, which contains a copy of a message request (i.e., *request*). In the case of GSL, the FVAT operator configured to mutate *request- table_id* cannot be applied (i.e., MOC is not equal to 100%); this indicates that the test cases do not cover a scenario in which the client passes a *table_id* above the threshold. To generate such a test case we may rely on the extended probe combined with ***static program analysis***.

Listing 2.11 shows how the ***extended mutation probe*** might be inserted into the code of libParam. In practice, it requires the engineer to know the portion of code that handles the generation of a request message. Unfortunately, injecting the mutation probe is not sufficient to enable test generation but engineers need also to prepare a test template to enable test generation with KLEE. Listing 2.12 shows an example of such template based on existing libParam test cases; such test case requires the initialization of a number of state variables, which limits the possibility to automate its definition. For this reason, within FAQAS we did not find it feasible to automate data-driven mutation analysis with a tool but we aim to evaluate its manual feasibility in WP4.

Finally, when data-driven mutation is applied to the data generated by the server, test automation is made unfeasible by the fact that KLEE cannot work in the presence of a communication channel within the code to be analyzed. Such shortcoming is not observed when we mutate request data because the extended mutation probe is installed only on the client; the producer-consumer case is not affected by such shortcoming because, in this case, the probe is installed on the producer. Alternative test generation tools or extensions of KLEE shall be considered to overcome such limitations.

```

static void gs_rparam_process_packet(csp_conn_t * conn, csp_packet_t * request_packet)
{
    csp_packet_t * reply_packet = NULL;
    gs_rparam_query_t * reply;


    /* Handle endian */
    gs_rparam_query_t * request = (gs_rparam_query_t *) request_packet->data;


    request->length = csp_ntoh16(request->length);
    request->checksum = csp_ntoh16(request->checksum);


    FaultModel *fm_General = _FAQAS_General_FM();
    unsigned long long int v_General[6];

    v_General[0] = (unsigned long long int) request->action;
    v_General[1] = (unsigned long long int) request->table_id;
    v_General[2] = (unsigned long long int) request->length;
    v_General[3] = (unsigned long long int) request->checksum;
    v_General[4] = (unsigned long long int) request->seq;
    v_General[5] = (unsigned long long int) request->total;


    _FAQAS_mutate(v_General,fm_General);
```

Listing 2.9: Example of data-driven mutation probe for libParam

```

/**
   Get string.
   @note If the returned string is max length, the value buffer will not be 0 terminated.
   @param[in] node CSP address
   @param[in] table_id remote table id.
   @param[in] addr parameter address (remote table).
   @param[in] checksum checksum
   @param[in] timeout_ms timeout
   @param[out] value returned value (user allocated)
   @param[in] value_size size of \a value, i.e. size of parameter type in bytes.
   @return_gs_error_t
*/
static inline gs_error_t gs_rparam_get_string(uint8_t node, gs_param_table_id_t table_id, uint16_t addr,
                                               uint16_t checksum, uint32_t timeout_ms, char * value, size_t value_size)
{
    return gs_rparam_get(node, table_id, addr, GS_PARAM_STRING, checksum, timeout_ms, value, value_size);
}


gs_error_t gs_rparam_get(uint8_t node,
                         gs_param_table_id_t table_id,
                         uint16_t addr,
                         gs_param_type_t type,
                         uint16_t checksum,
                         uint32_t timeout_ms,
                         void * value,
```

```c
                         size_t value_element_size)
{
    return gs_rparam_get_array(node, table_id, addr, type, checksum, timeout_ms, value, value_element_size, 1);
}


gs_error_t gs_rparam_get_array(uint8_t node,
                               gs_param_table_id_t table_id,
                               uint16_t addr,
                               gs_param_type_t type,
                               uint16_t checksum,
                               uint32_t timeout_ms,
                               void * value,
                               size_t value_element_size,
                               size_t array_size)
{
    /* Calculate length */
    gs_rparam_query_t * query;
    const size_t query_payload_size = sizeof(query->payload.addr[0]) * array_size;
    const size_t query_size = RPARAM_QUERY_LENGTH(query, query_payload_size);
    const size_t reply_payload_element_size = value_element_size + sizeof(query->payload.addr[0]);
    const size_t reply_payload_size = reply_payload_element_size * array_size;
    const size_t reply_size = RPARAM_QUERY_LENGTH(query, reply_payload_size);

    query = alloca(reply_size);
    query->action = RPARAM_GET;
    query->table_id = table_id;
    query->checksum = csp_hton16(checksum);
    query->seq = 0;
    query->total = 0;
    for(unsigned int i = 0; i < array_size; i++) {
        query->payload.addr[i] = csp_hton16(addr + (value_element_size * i));
    }
    query->length = csp_hton16(query_payload_size);

    FaultModel *fm_General = _FAQAS_General_FM();
    unsigned long long int v_General[6];

    v_General[0] = (unsigned long long int) query->action;
    v_General[1] = (unsigned long long int) query->table_id;
    v_General[2] = (unsigned long long int) query->length;
    v_General[3] = (unsigned long long int) query->checksum;
    v_General[4] = (unsigned long long int) query->seq;
    v_General[5] = (unsigned long long int) query->total;


    _FAQAS_cover(v_General,fm_General);


    /* Run single packet transaction */
    if (csp_transaction2(CSP_PRIO_HIGH, node, GS_CSP_PORT_RPARAM, timeout_ms, query, query_size, query, reply_size, CSP_O_CRC32) <= 0) {
        return GS_ERROR_IO;
    }
 ...

 }
```

Listing 2.10: Example of extended data-driven mutation probe for libParam

```c

    // a little hack - this is next element, we use it check for overwrite and missing 0 termiation
    memset(alltypes_mem.string_A, 'Z', sizeof(alltypes_mem.string_A));
    alltypes_mem.string_A[0][1] = 0;

    char buf[GS_TEST_ALLTYPES_STRING_LENGTH + 10];

    // get max size - no 0 termination
    memset(alltypes_mem.string, 'B', sizeof(alltypes_mem.string));
    memset(buf, 'A', sizeof(buf));
    buf[GS_TEST_ALLTYPES_STRING_LENGTH + 1] = 0;

    csp_node CSP_NODE;
    unsigned long long int tableID;
    klee_make_symbolic(&CSP_NODE, sizeof(CSP_NODE), ""CSP_NODE);
    klee_make_symbolic(&tableID, sizeof(tableID), ""tableID);
    gs_rparam_get_string(&CSP_NODE, tableID, GS_TEST_ALLTYPES_STRING, GS_RPARAM_MAGIC_CHECKSUM, 1000, buf, GS_TEST_ALLTYPES_STRING_LENGTH);
```

Listing 2.11: Test template to enable data-driven mutation testing for libParam

Index