securityandtrust.lu

FAQAS Framework

RB - Requirements Baseline

SSS - Software Systems Specifications

F. Pastore, O. Cornejo

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

ITT-1-9873-ESA-FAQAS-SSS

Issue 1, Rev. 1

November 5, 2020

UNIVERSITÉ DU LUXEMBOURG

Revisions

Issue Number	Date	Authors	Description
ITT-1-9873-ESA-FAQAS-D2 Issue 1 Rev. 1	November 4th, 2020	Fabrizio Pastore, Oscar Cornejo	Initial release.

Contents

Chapter 1

Scope and content

1.1 Introduction

This document is the deliverable SSS of the ESA activity ITT-1-9873-ESA. It concerns requirements specification for the *FAQAS framework* to be delivered by ITT-1-9873-ESA. Following the structure described in the SoW *AO9873-ws00pe_SOW.pdf*, it provides the structured requirements baseline for the FAQAS framework according to ECSS-E-ST-40C Annex B. Since the *FAQAS framework* implements two distinct functionalities, code-driven mutation testing and data-driven mutation testing, this document contains two separate chapters, each one concerning one of the two features; more precisely

Requirements are univocally identified with the paragraph id appearing on the left.

1.2 Applicable and reference documents

- D1 - Mutation testing survey
- D2 - Study of mutation testing applicability to space software

1.3 Terms, definitions and abbreviated terms

Chapter 2

Code-driven Mutation Testing

2.1 General description

2.1.1 Product perspective

2.1.1.1 The code-driven mutation testing component (in Section 2.1 referred to as *the system*) implements the Mutation Testing Process for code-driven mutation testing described in D2.

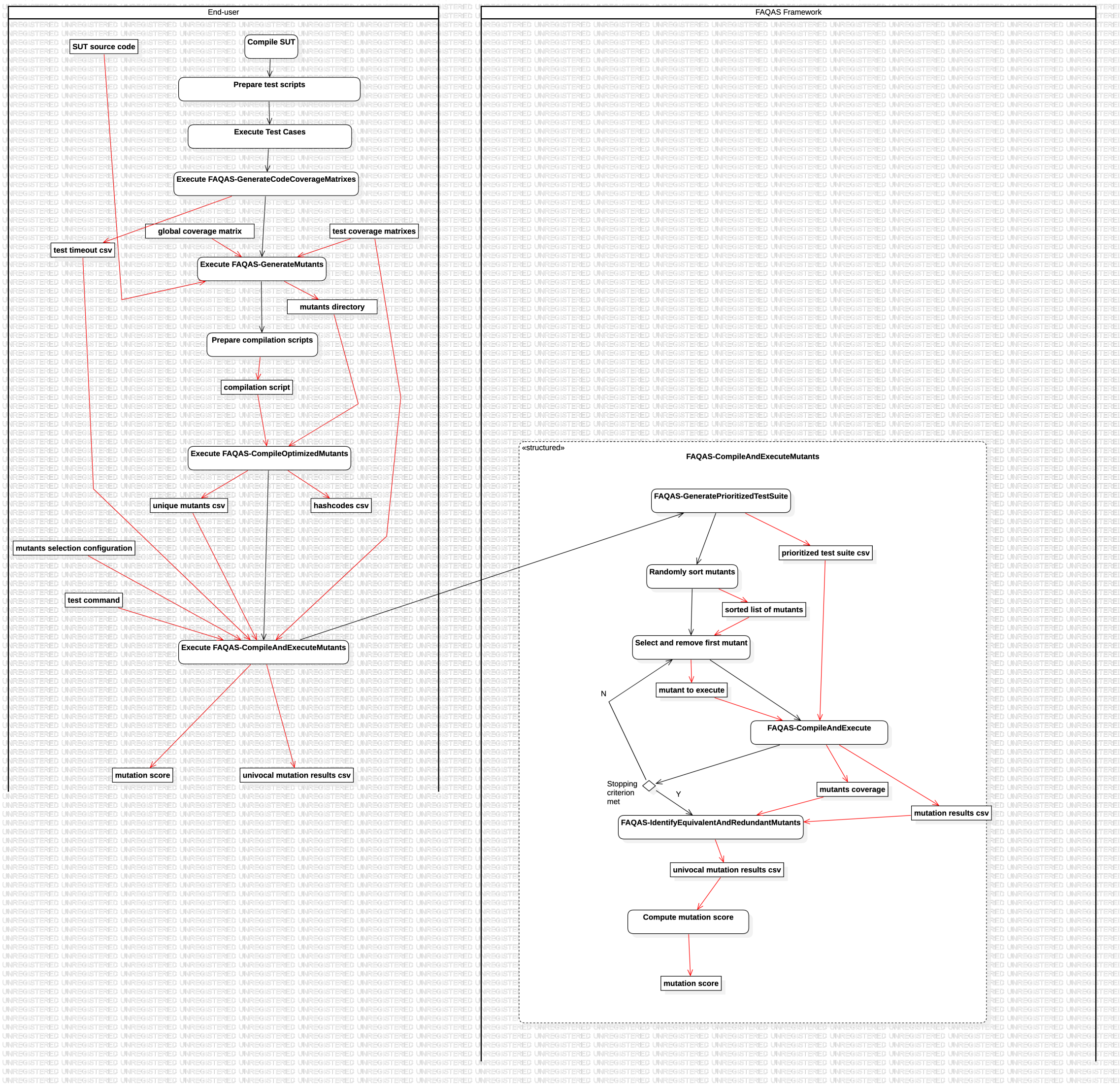

Figure 2.1: Overview of the code-driven mutation testing process to evaluate test suite effectiveness.

2.1.2 General capabilities

2.1.2.1 The code-driven mutation testing component shall implement the process for the evaluation of test suite effectiveness that is drafted in Figure 2.1. Figure 2.1 relies on UML activity diagram notation where the execution of specific software artefacts from the end user is made explicit. Each activity is described in Section 2.2.1.

2.1.3 General constraints

2.1.4 Operational environment

2.1.4.1 The system works with a Linux operating system and Bash shell.

2.1.5 Assumptions and dependencies

2.1.5.1 The system targets SUT built using either GCC Make [1] or WAF[2].

2.1.5.2 The system targets SUT compiled with GCC [3].

2.2 Specific requirements

2.2.1 Capabilities requirements

2.2.1.1 The gcov coverage information associated to each test case shall be stored in a separate directory.

2.2.1.2 The activity *Compile SUT* in Figure 2.1 concerns the compilation of the SUT with coverage options enabled.

2.2.1.3 The activity *Prepare test scripts* in Figure 2.1 concerns extending the test scripts to store the code coverage of each single test case separately. This is achieved by add add a call to a dedicated bash script provided by FAQAS (*FAQAS-CollectCodeCoverage*).

2.2.1.4 The activity *Execute test cases* in Figure 2.1 concerns the execution of the test cases following the practice for the SUT.

2.2.1.5 The activity *Execute FAQAS-GenerateCodeCoverageMatrix* in Figure 2.1 concerns the execution of a provided python program delivered with the FAQAS framework.

2.2.1.6 The activity *Execute FAQAS-GenerateCodeCoverageMatrix* in Figure 2.1 generates a set of files:

- one csv file referred to as *global coverage marix*, which indicates, for every lines of code of the SUT, the ID of the test cases that cover the line of code;
- a number of files referred to as *test coverage matrix*, one for each test case of the SUT. Each file indicates, for every line of code of the SUT, the number of times it has been covered during a single execution of the test case;

[1] https://gcc.gnu.org/onlinedocs/gccint/Makefile.html
[2] https://waf.io/
[3] https://gcc.gnu.org

- one file with the timeout after which we can consider a test case as non terminated (used in later stages). It is obtained by multiplying the test execution time times three.

2.2.1.7 Activity *Execute FAQAS-GenerateMutants* in Figure 2.1 concerns the execution of the program *FAQAS-GenerateMutants*.

2.2.1.8 *FAQAS-GenerateMutants* automatically generates a number of copies of each source file. Each copy contains one mutant.

2.2.1.9 *FAQAS-GenerateMutants* mutates source files with extension .c and .cpp.

2.2.1.10 *FAQAS-GenerateMutants* generates mutants by applying a set of mutation operators that can be selected by the end-users.

2.2.1.11 *FAQAS-GenerateMutants* implements the set of operators listed in Table 2.1

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

*Each pair in parenthesis shows how a program element is modified by the mutation operator. Th eleft element of the pair is replaced with the right element. 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).

2.2.1.12 *FAQAS-GenerateMutants* generates as output a directory tree (*mutants directory* in Figure 2.1) that follows the structure of the source directory tree of the SUT. However, every source file is replaced by a folder having the same name. The folder contains all the mutants generated for that file. Every mutant has a name that univocally identify it. The mutant name results from the conjunction of the following information: source file name, mutated function name, mutated line, mutation operator name, mutation operation, mutated "column" (i.e., char position from the beginning of the line).

2.2.1.13 Activity *Prepare compilation scripts* in Figure 2.1 concern the modification of the main compilation script for the SUT. The engineer is expected to perform the following manual activities:

- Remove debugging flags
- Remove coverage flags

- Add placeholder for compiler optimization option
- Add a 'sort' command in the source dependency list to ensure that source files are always compiled in the same order

2.2.1.14 Activity *Execute FAQAS-CompileOptimizedMutants* in Figure 2.1 concerns the execution of the program *FAQAS-CompileOptimizedMutants*.

2.2.1.15 The program *FAQAS-CompileOptimizedMutants* compiles every mutant multiple times; once for every compiler optimization option selected by the end-user. It implements pseudocode in Figure 2.2.

```
Require: OPT, the set of compiler optimization options specified by the end-user
Require: MutantsDir, path to the directory tree containing the mutants
Require: SUTsources, path of the folder containing the sources of the SUT
Require: CompilatonCommand, the command to execute to compile the original software
Ensure: hashcodes csv, a csv file containing for every mutant, for every option, the SHA512 hashcode of the generated executable
Ensure: unique mutants, a csv file the list of unique mutants. Unique mutants are mutants that are not equivalent and not redundant.
    See D2 for details.
for OPT in OPTS do
    for Mutant in MutantsDir do
        Compile Mutant with program FAQAS-CompileAndExecute
        Generate a SHA512 hash of the generated executable
        Put the generated SHA512 hash in the hashcodes csv file
    end for
end for
Process hashcodes csv and identify unique mutants
Save the list of unique mutants in the output unique mutants csv file
```

Figure 2.2: FAQAS-CompileOptimizedMutants: Algorithm for compiling mutants with multiple optimization options

2.2.1.16 Activity *Execute FAQAS-CompileAndExecuteMutants* in Figure 2.1 concerns the execution of the program *FAQAS-CompileAndExecuteMutants*.

2.2.1.17 The program *FAQAS-CompileAndExecuteMutants* iterates over three activities (implemented by separate executable program that are inkoved automatically without user intervention): *FAQAS-GeneratePrioritizedTestSuite*, *FAQAS-CompileAndExecute*, *FAQAS-IdentifyEquivalentAndRedundantMutants*.

2.2.1.18 The program *FAQAS-CompileAndExecuteMutants* takes as inputs the mutants selection configuration, the unique mutants csv, the path of the SUT source folder, the command to execute test cases, and the path to the folder containing the test coverage matrixes.

2.2.1.19 The program *FAQAS-CompileAndExecuteMutants* implements the four mutants selection strategies described in D2: *all mutants*, *proportional uniform sampling*, *proportional method-based sampling*, *uniform fixed-size sampling*, and *uniform FSCI sampling*.

2.2.1.20 The *mutants selection configuration* indicates the mutants selection strategy and a configuration value to specify the number of mutants to consider, which depends on the strategy; the value may indicate the percentage of mutants to sample (for *proportional uniform sampling*, *proportional method-based sampling*), the number of mutants to sample (for *uniform fixed-size sampling*), the size of the confidence interval (for *uniform FSCI sampling*).

2.2.1.21 The program*FAQAS-GeneratePrioritizedTestSuite* takes as input the test coverage matrixes and generate a file that specifies, for every line of the SUT, the prioritized list of test cases to execute (*prioritized test suite csv*). This file indicates the sequence of test cases to execute for every mutants concerning a specific line.

2.2.1.22 The activity *Randomly sort mutants* indicates that *FAQAS-CompileAndExecuteMutants* generate a randomly prioritized list of mutants to compile and execute from the *unique mutants csv*. In the case of *proportional method-based sampling*, the list contains a set of mutants selected by following the stratified sampling strategy.

2.2.1.23 The activity *Select and remove first mutant* indicates that *FAQAS-CompileAndExecuteMutants* select the first mutant in *sorted list of mutants* and remove it from the list.

2.2.1.24 The program *FAQAS-CompileAndExecute* compiles a mutant by running the makefile of the original program; then it executes the SUT test suite. It follows the algorithm in Figure 2.3.

```
Require: Mutant, path of the mutant to compile
Require: SUTsources, path of the folder containing the sources of the SUT
Require: CompilatonCommand, the command to execute to compile the original software
Require: TestCommand, the command to execute to execute a single test case
Require: TestCases, the prioritized list of test cases for the line of the mutant
Require: TestTimeout, the max execution time that can be taken by the test case
Ensure: Result KILLED or LIVE, based on test execution result (i.e., all test cases pass or one tst case fails)
put Mutant in place of the file it has been derived (original file), keep the original file in a safe place
execute CompilatonCommand inside SUTsources
for TestCase in TestCases do
    execute the TestCase by running TestCommand inside SUTsources
    if t thenhe TestCase fails (i.e., TestCommand terminates with an error code)
        set Result as KILLED
        break the for loop
    end if
    if a then the TestTimeout expires
        set Result as KILLED
        break the for loop
    end if
end for
move code coverage information in a subfolder of mutants coverage dir
restore original file
```

Figure 2.3: FAQAS-CompileAndExecute: Algorithm to compile and test mutants

2.2.1.25 The program *FAQAS-CompileAndExecute* collects the mutation results of every mutant in a file, *mutation results csv*. It contains for every mutant the indication of the mutation result (KILLED/LIVE).

2.2.1.26 The program *FAQAS-CompileAndExecute* compiles and execute mutants till a termination criteria is met. The termination criteria depends on the mutants selection strategy:

- *all mutants*: the list *sorted list of mutants* is empty
- *proportional uniform sampling*: a number of mutants matching the selected percentage has been executed
- *proportional method-based sampling*: the list *sorted list of mutants* is empty
- *uniform fixed-size sampling*: a number of mutants matching the selected value has been executed
- *uniform FSCI sampling*: the confidence interval computed from *mutation results csv* is smaller than the lenght specified by the user.

2.2.1.27 The program *FAQAS-IdentifyEquivalentAndRedundantMutants* relies on code coverage information stored in *mutants coverage dir* to identify equivalent and redundant mutants using the distance criterion D_C (see D2).

2.2.1.28 The program *FAQAS-IdentifyEquivalentAndRedundantMutants* generates a copy of *mutation results csv* (i.e., *univocal mutation results csv*) where only mutants that are considered non-equivalent and non-redundant are reported.

2.2.1.29 The activity *Compile mutation score* concerns the computation of the mutation score based on the mutation results reported in *univocal mutation results csv*.

2.2.2 System interface requirements

2.2.3 Adaptation and missionization requirements

2.2.4 Computer resource requirements

2.2.5 Security requirements

2.2.6 Safety requirements

2.2.7 Reliability and availability requirements

2.2.8 Quality requirements

2.2.9 Design requirements and constraints

2.2.10 Software operations requirements

2.2.11 Software maintenance requirements

2.2.12 System and software observability requirements

2.3 Verification, validation and system integration

2.3.1 Verification and validation process requirements

2.3.2 Validation approach

2.3.3 Validation requirements

2.3.4 Verification requirements

2.4 System models

Chapter 3

Data-driven Mutation Testing

3.1 General description

3.1.1 Product perspective

3.1.1.1 The data-driven mutation testing component implements the Mutation Testing Process for code-driven mutation testing described in D2.

3.1.2 General capabilities

3.1.3 General constraints

3.1.4 Operational environment

3.1.5 Assumptions and dependencies

3.2 Specific requirements

3.2.1 Capabilities requirements

3.2.2 System interface requirements

3.2.3 Adaptation and missionization requirements

3.2.4 Computer resource requirements

3.2.5 Security requirements

3.2.6 Safety requirements

3.2.7 Reliability and availability requirements

3.2.8 Quality requirements

3.2.9 Design requirements and constraints

3.2.10 Software operations requirements

3.2.11 Software maintenance requirements

3.2.12 System and software observability requirements

3.3 Verification, validation and system integration

3.3.1 Verification and validation process requirements

3.3.2 Validation approach

3.3.3 Validation requirements

3.3.4 Verification requirements

3.4 System models