securityandtrust.lu

FAQAS Framework
SUITP
Software Unit and Integration Test Plan

O. Cornejo, F. Pastore

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

ITT-1-9873-ESA-FAQAS-SUITP

Issue 1, Rev. 1

March 30, 2021

UNIVERSITÉ DU LUXEMBOURG

Revisions

Issue Number	Date	Authors	Description
ITT-1-9873-ESA-FAQAS-SUITP Issue 1 Rev. 1	March 31th, 2021	Oscar Cornejo, Fabrizio Pastore	Initial release.

Contents

Chapter 1

Introduction

This document is the combined Software Unit and Integration Test Plan of the software delivered by the ESA activity ITT-1-9873-ESA (i.e., the *FAQAS framework*). Its purpose is to describe the unit and integration tests to be done for the FAQAS-framework. The FAQAS framework consists of the following software: a toolset implementing code-driven mutation analysis (MASS), a toolset implementing code-driven test generation, a toolset implementing data-driven mutation analysis, a toolset implementing data-driven test generation.

This document follows a structure derived from tailoring of the structure proposed in ECSS-E-ST-40C Annex K. Tailoring has been adopted to clearly describe differences in the procedures adopted for the different components of the FAQAS-framework. More precisely, our tailoring adheres to the following directives:

- Sections *K.1 - Introduction*, *K.2 - Applicable and reference documents*, *K.3 - Terms, definitions and abbreviated terms*, which are generic, are reported following the structure suggested in ECSS-E-ST-40C.
- Sections *K.4 - Software overview*, *K.5 - Software unit testing and software integration testing*, and *K.6 - Control procedures for software unit testing / integration testing* are reported following the structure suggested in ECSS-E-ST-40C. They describe the procedures followed concern all the components of the FAQAS-framework.
- Sections *K.7 - Software unit testing and integration testing approach*, *K.8 - Software unit test / integration test design*, *K.9 - Software unit and integration test case specification*, and *K.10 - Software unit and integration test procedures* are repeated multiple times, once for each tested component of the FAQAS-framework[1].
- Section *K.11 - Software test plan additional information* appears once and covers al the components of the FAQAS-framework.

[1] Issue 1 contains only MASS

Chapter 2

Applicable and reference documents

- D1 - Mutation testing survey
- D2 - Study of mutation testing applicability to space software
- SSS - Software Systems Specifications
- SUM - Software User Manual
- SVS - Software Validation Specifications

Chapter 3

Terms, definitions and abbreviated terms

- FAQAS: activity ITT-1-9873-ESA
- FAQAS-framework: software system to be released at the end of WP4 of FAQAS
- D2: Deliverable D2 of FAQAS, *Study of mutation testing applicability to space software*
- KLEE: Third party test generation tool, details are provided in D2.
- SUT: Software under test, i.e, the software that should be mutated by means of mutation testing.
- WP: Work package

Chapter 4

Software overview

This document concerns the unit testing of the following components:

- code-driven mutation analysis toolset (MASS)
- code-driven test generation toolset (FAQAS-SEMu)
- data-driven mutation analysis toolset (DataDrivenMutator)
- data-driven test generation toolset

Chapter 5

Software Unit Testing and Software Integration Testing

5.1 Organization

Unit tests are prepared by SnT in the form of Bash shell scripts.

Since units are self-contained (i.e., they do not interact with other units but simply store results in files processed by other units), we avoid testing the integration of pairs of unit. Instead, we test the system as a whole, based on tutorial examples described in SUM. The system testing procedure is reported in the SVS - software validation specification.

Testing is conducted by SnT personnel.

If any software problems occur during testing a development issue shall be raised in GitLab Issue Tracker for the FAQAS-framework (see Section 6), which shall be amended by SnT personnel.

5.2 Resource Summary

Unit tests and integration tests make use of the FAQAS-framework. Tests will be performed in one hardware platform, a x86-64 desktop PC. Unit test execution time should not exceed one day for all target platforms combined.

5.3 Responsibilities

Unit tests tests are prepared by the SnT personnel.

The test cases are executed by an SnT specialist who has the responsibility of reporting any failure.

5.4 Tool, techniques and methods

Unit tests are specified in Bash files. Each unit test contains a launcher script that configures the environment for the correct execution of FAQAS-framework, and a source code example for its mutation. The launcher script will invoke a dedicated operator Bash script, that generates the corresponding mutants and assesses its results.

5.5 Personnel and Personnel Training Requirements

Unit testing is performed by a single person using a launcher script from the source folder of the SUT. Additional training is not needed.

5.6 Risks and Contingencies

The unit testing campaign is conducted in parallel with the development of the SUT and thus is not associated to any specific risk.

Chapter 6

Control procedure for Software Unit and Integration Testing

Technical and organizational problems shall be reported on in GitLab Issue Tracker for the FAQAS-framework, which is available at https://gitlab.uni.lu/fpastore/FAQAS. The FAQAS management will take care of conflict resolution.

Chapter 7

MASS - Software Unit Testing and Integration Testing Approach

7.1 Unit/Integration Testing Strategy

Integration testing is out of scope because of the motivations discussed in Section 5.1.

Unit testing aims to verify that the functional requirements of MASS units are correctly implemented; test inputs are identified through the category-partition method.

7.2 Tasks and Items under Test

Testing concerns the source code mutation component (hereafter, *SRCMutation*) of MASS. SRCMutation is the component with the most complicate implementation logic and thus require detailed unit testing. All the other components either filter or join data, their implementation is simpler than SRCMutation and thus their test automation is performed through system tests (described in SVS).

7.3 Feature to be tested

Testing concerns verifying the correct functional behavior of the mutation operators implemented by *SRCMutation*.

7.4 Feature not to be tested

Testing does not concern the verification of the capability of *SRCMutation* to parse any possible source file valid according to C/C++ language grammar. Since *SRCMutation* is implemented on top of CLANG, we assume parsing capabilities are inherited from CLANG.

7.5 Test Pass - Fail Criteria

Unit testing pass if all the following are true:

- Every test case is executed
- All test cases pass
- Exceptions and unexpected messages do not appear on screen and logs.

7.6 Manually and Automatically Generated Code

The FAQAS-framework does not contain any automatically generated code.

Chapter 8

MASS - Software Unit and Integration Test Case Design

8.1 General

The MASS unit test suite concerns the source code mutation component (SRCMutation). It address its functional requirements. A single test design has been identified, it is based on the category partition method and reported in the following sections.

8.2 MASS - Test Design - SRCMutation - Operators

8.2.1 Test design identifier

The test design identifier is *MASS-TD-SRCMutation-1*

With this test design we aim to ensure that each mutation operator for the FAQAS is implemented according to its requirements.

8.2.2 Features to be tested

Table 8.1 shows the specifications for the mutation operators implemented by SRCMutation.

The set of SRCMutation mutation operators is composed of the following: Absolute Value Insertion (ABS), Arithmetic Operator Replacement (AOR), Integer Constraint Replacement (ICR), Logical Connector Replacement (LCR), Relational Operator Replacement (ROR), Unary Operator Insertion (UOI), Statement Deletion Operator (SDL), and Literal Value Replacement (LVR). It also include OODL mutation operators: delete Arithmetic (AOD), Bitwise (BOD), Logical (LOD), Relational (ROD), and Shift (SOD) operators.

Each mutation operator, when applied to a statement, generates one or more mutated statements. Each mutation operator works by altering the value of a *term* in a statement, which could be either an operator (op), a value (v), or a literal (l). More precisely, each mutation operator replaces a term with a number of replacement terms, which are identified based on the rules in the description

Table 8.1: Implemented set of mutation operators.

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

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

column of Table 8.1. For each term to mutate, SRCMutation shall generate one mutant including each replacement term. If a statement includes more than one term to mutate, the mutation operator generates a set of mutants for each term to mutate. Table 8.2, shows, for every mutation operator the terms it mutates and the replacement terms (separated by comma). Replacement terms shall be used when defining test assertions (i.e., we shall verify that the term to mutate has been replaced any of the available replacements, one for each generated mutant).

Table 8.2: Terms to mutate and replacements per mutation oprator.

Operator	Term to mutate	Replacements
ABS	v	$-v$
AOR	$+$	$-, *, /, \%$
AOR	$-$	$+, *, /, \%$
AOR	$*$	$+, -, /, \%$
AOR	$/$	$+, -, *, \%$
AOR	$\%$	$+, -, *, /$
AOR	$+=$	$-=, *=, /=, \%=$
AOR	$-=$	$+=, *=, /=, \%=$
AOR	$*=$	$+=, -=, /=, \%=$
AOR	$/=$	$+=, -=, *=, \%=$
AOR	$\%=$	$+=, -=, *=, /=$
ICR	i	$1, -1, 0, i+1, i-1, -i$
LCR	&&	||
LCR	||	&&
LCR	&	$\|, \wedge$
LCR	|	$\&, \wedge$
LCR	$\wedge$	$\&, \|$
LCR	$\&=$	$\|=, \wedge=$
LCR	$\|=$	$\&=, \wedge=$
LCR	$\wedge=$	$\&=, \|=$
ROR	$>$	$>=, <, <=, ==, !=$
ROR	$>=$	$>, <, <=, ==, !=$
ROR	$<$	$>, >=, <=, ==, !=$
ROR	$<=$	$>, >=, <, ==, !=$
ROR	$==$	$>, >=, <, <=, !=$
ROR	$!=$	$>, >=, <, <=, ==$
ROR	`if(e )`	`if(!e)`
ROR	`while( e)`	`while(!e)`
SDL	s	`remove(s)`
UOI	v	$-v, v-, ++v, v++$
AOD	$a+b$	a, b
AOD	$a-b$	a, b
AOD	$a*b$	a, b
AOD	a/b	a, b
AOD	$a\%b$	a, b
LOD	$a\&\&b$	a, b
LOD	$a\|\|b$	a, b
ROD	$>$	a, b
ROD	$>=$	a, b
ROD	$<$	a, b
ROD	$<=$	a, b
ROD	$==$	a, b
ROD	$!=$	a, b
BOD	&	a, b
BOD	|	a, b
BOD	$\wedge$	a, b
SOD	$>>$	a, b
SOD	$<<$	a, b
LVR	0	-1
LVR	l	$-l, 0$
LVR	`true`	`false`
LVR	`false`	`true`

8.2.3 Approach refinements

Based on Table 8.2 we can derive the categories to be used for the category-partition method. They are *Operator* (i.e., the operator to be applied, which could be a specific *one* or *many* operators at once) and *Term to replace* (which depend on the operator and might be either *one* or *many* for each statement).

We provide the identified categories and class values in tabular form, in Table 8.3. Because of the presence of many constraints between the categories *Operator* and *Term to replace*, to simplify the reading, in Table 8.3, instead of listing dependencies between *Operator* and *Term to replace*, we simply provide all the feasible combinations. In Table 8.3, the keyword *many* may indicates that either (a) more than one term should be replaced in a same statement (if the keyword *many* appears under *Term to replace*) or (b) more than one operator shall be applied in a same statement (if the keyword *many* appears under *Operator*). In Table 8.3, column *Constraints* reports other standard category-partition constraints (in this case, all the value classes shall be tested once). Finally, we also provide corresponding test identifiers, which match the name of the test script file. Test cases are identified using the name of the operator acronym and the input type to be processed, for example the test case `ror_lt.sh` represents the ROR operator for the "less than" input.

Table 8.3: Organization matrix of unit test cases for source code mutation operators component.

Operator	Term to replace	Constraints	Test Case
ABS	v	[single]	abs_val.sh
ABS	*many*	[single]	
AOR	$+$	[single]	aor_plus.sh
AOR	$-$	[single]	aor_minus.sh
AOR	$*$	[single]	aor_mult.sh
AOR	$/$	[single]	aor_div.sh
AOR	%	[single]	aor_mod.sh
AOR	$+=$	[single]	aor_plus_assign.sh
AOR	$-=$	[single]	aor_minus_assign.sh
AOR	$*=$	[single]	aor_mult_assign.sh
AOR	$/=$	[single]	aor_div_assign.sh
AOR	$\%=$	[single]	aor_mod_assign.sh
AOR	*many*	[single]	
ICR	i	[single]	icr_val.sh
ICR	*many*	[single]	
LCR	&&	[single]	lcr_logic_or.sh
LCR	||	[single]	lcr_logic_and.sh
LCR	&	[single]	lcr_and.sh
LCR	|	[single]	lcr_or.sh
LCR	$\wedge$	[single]	lcr_xor.sh
LCR	$\&=$	[single]	lcr_and_assign.sh
LCR	$\|=$	[single]	lcr_or_assign.sh
LCR	$\wedge=$	[single]	lcr_xor_assign.sh
LCR	*many*	[single]	
ROR	$>$	[single]	ror_gt.sh
ROR	$>=$	[single]	ror_ge.sh
ROR	$<$	[single]	ror_lt.sh
ROR	$<=$	[single]	ror_le.sh
ROR	$==$	[single]	ror_eq.sh
ROR	$!=$	[single]	ror_neq.sh
ROR	`if(e )`	[single]	ror_if.sh
ROR	`while( e)`	[single]	ror_while.sh
ROR	*many*	[single]	
SDL	s	[single]	sdl.sh
SDL	*many*	[single]	
UOI	v	[single]	uoi.sh
UOI	*many*	[single]	
AOD	$a+b$	[single]	aod_plus.sh
AOD	$a-b$	[single]	aod_minus.sh
AOD	$a*b$	[single]	aod_mult.sh
AOD	a/b	[single]	aod_div.sh
AOD	$a\%b$	[single]	aod_mod.sh
AOD	*many*	[single]	
LOD	$a\&\&b$	[single]	lod_logic_and.sh
LOD	$a\|\|b$	[single]	lod_logic_or.sh
LOD	*many*	[single]	
ROD	$>$	[single]	rod_gt.sh
ROD	$>=$	[single]	rod_ge.sh
ROD	$<$	[single]	rod_lt.sh
ROD	$<=$	[single]	rod_le.sh
ROD	$==$	[single]	rod_eq.sh
ROD	$!=$	[single]	rod_neq.sh
ROD	*many*	[single]	
BOD	&	[single]	bod_and.sh
BOD	|	[single]	bod_or.sh
BOD	$\wedge$	[single]	bod_xor.sh
BOD	*many*	[single]	
SOD	$>>$	[single]	sod_sl.sh
SOD	$<<$	[single]	sod_sr.sh
SOD	*many*	[single]	
LVR	0	[single]	lvr_zero.sh
LVR	l	[single]	lvr_literal.sh
LVR	`true`	[single]	lvr_true.sh
LVR	`false`	[single]	lvr_false.sh
LVR	*many*	[single]	
many	*many*	[single]	

Chapter 9

MASS - Software Unit and Integration Test Case Specification

9.1 General

Table 9.1 provides the list of unit test cases derived based on the procedure described in Chapter 9. Column *Term to replace* indicates the operator appearing in the line to be mutated. When a test case is supposed to mutate many terms, we report all of them. Column *Replacements* provides, for each operator, the expected replacements. Note that *SRCMutation* shall generate one distinct mutant for each element of the replacement column.

TODO: Add missing test cases

Table 9.1: Unit test cases for SRCMutation.

Operator	Term to replace	Replacements	Test Case
ABS	v	$-v$	abs_val.sh
AOR	$+$	$\{-, *, /, \%\}$	aor_plus.sh
AOR	$-$	$\{+, *, /, \%\}$	aor_minus.sh
AOR	$*$	$\{+, -, /, \%\}$	aor_mult.sh
AOR	$/$	$\{+, -, *, \%\}$	aor_div.sh
AOR	$\%$	$\{+, -, *, /\}$	aor_mod.sh
AOR	$+=$	$\{-=, *=, /=, \%=\}$	aor_plus_assign.sh
AOR	$-=$	$\{+=, *=, /=, \%=\}$	aor_minus_assign.sh
AOR	$*=$	$\{+=, -=, /=, \%=\}$	aor_mult_assign.sh
AOR	$/=$	$\{+=, -=, *=, \%=\}$	aor_div_assign.sh
AOR	$\%=$	$\{+=, -=, *=, /=\}$	aor_mod_assign.sh
ICR	i	$\{1, -1, 0, i+1, i-1, -i\}$	icr_val.sh
LCR	&&	||	lcr_logic_or.sh
LCR	||	&&	lcr_logic_and.sh
LCR	&	$\{\|, \wedge\}$	lcr_and.sh
LCR	|	$\{\&, \wedge\}$	lcr_or.sh
LCR	$\wedge$	$\{\&, \|\}$	lcr_xor.sh
LCR	$\&=$	$\{\|=, \wedge=\}$	lcr_and_assign.sh
LCR	$\|=$	$\{\&=, \wedge=\}$	lcr_or_assign.sh
LCR	$\wedge=$	$\{\&=, \|=\}$	lcr_xor_assign.sh
ROR	$>$	$\{>=, <, <=, ==, !=\}$	ror_gt.sh
ROR	$>=$	$\{>, <, <=, ==, !=\}$	ror_ge.sh
ROR	$<$	$\{>, >=, <=, ==, !=\}$	ror_lt.sh
ROR	$<=$	$\{>, >=, <, ==, !=\}$	ror_le.sh
ROR	$==$	$\{>, >=, <, <=, !=\}$	ror_eq.sh
ROR	$!=$	$\{>, >=, <, <=, ==\}$	ror_neq.sh
ROR	`if(e )`	`if(!e)`	ror_if.sh
ROR	`while( e)`	`while(!e)`	ror_while.sh
SDL	s	`remove(s)`	sdl.sh
UOI	v	$\{$`-`v, v`-`$, $`++`$v, v$`++`$\}$	uoi.sh
AOD	$a+b$	$\{a, b\}$	aod_plus.sh
AOD	$a-b$	$\{a, b\}$	aod_minus.sh
AOD	$a*b$	$\{a, b\}$	aod_mult.sh
AOD	a/b	$\{a, b\}$	aod_div.sh
AOD	$a\%b$	$\{a, b\}$	aod_mod.sh
LOD	$a\&\&b$	$\{a, b\}$	lod_logic_and.sh
LOD	$a\|\|b$	$\{a, b\}$	lod_logic_or.sh
ROD	$>$	$\{a, b\}$	rod_gt.sh
ROD	$>=$	$\{a, b\}$	rod_ge.sh
ROD	$<$	$\{a, b\}$	rod_lt.sh
ROD	$<=$	$\{a, b\}$	rod_le.sh
ROD	$==$	$\{a, b\}$	rod_eq.sh
ROD	$!=$	$\{a, b\}$	rod_neq.sh
BOD	&	$\{a, b\}$	bod_and.sh
BOD	|	$\{a, b\}$	bod_or.sh
BOD	$\wedge$	$\{a, b\}$	bod_xor.sh
SOD	$>>$	$\{a, b\}$	sod_sl.sh
SOD	$<<$	$\{a, b\}$	sod_sr.sh
LVR	0	-1	lvr_zero.sh
LVR	l	$\{-l, 0\}$	lvr_literal.sh
LVR	`true`	`false`	lvr_true.sh
LVR	`false`	`true`	lvr_false.sh

9.2 Organization of the Test Cases

```
1 double function() {
2 int a = 4, b = 5;
3 return a / b;
4 }
```

Listing 9.1: C function example.

```
1  $MUTATOR --compilation "$FILE -o test" --operators ABS
2
3  EXPECTED="double function() {\ndouble a = 3;\nreturn -(a);\n}"
4
5  tst=`diff test.mut.3.1_1_8.ABS.function.c <(echo -e $EXPECTED) | wc -l`
6
7  if [ $tst -eq 0 ];then
8      echo -e "TEST abs_val PASSED"
9  else
10     echo -e "TEST abs_val FAILED"
11 fi
```

Listing 9.2: ABS test case example.

Listings 9.1 and 9.2 introduce an example of source code and test case for the mutation operator ABS, respectively. As shown in Listing 9.2, each test case (i) invokes the mutator component selecting the corresponding operator acronym (Line 1), (ii) defines the expected output for operator (Line 3), (iii) checks if there are differences between the actual output of the component and the expected output (Line 5), (iv) print out the test result (Lines 7-11).

All test cases are independent from each other; therefore there is no need for executing test cases in a specific order. Note that a mutation operator might produce one or more mutants for a single input, all outputs shall be as expected.

Chapter 10

MASS - Software Unit and Integration Test Procedures

10.1 General

A single procedure has been identified to perform unit testing of MASS, it is described in the following.

10.2 MASS Test Procedure

10.2.1 Test procedure Identifier

10.2.2 Purpose

This procedure aims to execute all the test cases reported in Table 9.1.

10.2.3 Procedure steps

10.2.3.1 log

Since test cases are executed within a console, the output of the test cases can be considered as is. In case failures need to be reported, the console output should be copy-pasted.

10.2.3.2 set up

SRCMutation comes with test cases already set-up. The only precondition is to compile the software by running the command *make*.

10.2.3.3 start

Test cases are executed by running the script `run_unit_tests.sh`.

10.2.3.4 test result acquisition

Test results are print out on the console.

10.2.3.5 shut down

Test cases are shut down through a SIGINT (i.e., pressing `CTRL-C`)

10.2.3.6 restart

Test cases can be restarted at any point by re-running the script `run_unit_tests.sh`.

10.2.3.7 wrap up

Test execution terminates autonomously, no further action is needed.

Chapter 11

Software Test Plan Additional Information

Traceability between test cases and procedures is straightforward and covered in respective sections.

Test scripts are automated and provided along with the software.