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

Scope and content

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*, and *K.9 - Software unit and integration test case specification* appear multiple times, one for each tested component of the FAQAS-framework[1].

1.1 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

[1] Issue 1 contains only MASS

Chapter 2

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 3

Software overview

This document concerns the testing of the following components

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

Chapter 4

Software Unit Testing and Software Integration Testing

4.1 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, which shall be amended by SnT personnel.

4.2 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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 5

MASS - Software Unit Testing and Integration Testing Approach

5.1 Unit/Integration Testing Strategy

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

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

5.2 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).

5.3 Feature to be tested

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

5.4 Feature not to be tested

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

5.5 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.

5.6 Manually and Automatically Generated Code

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

Chapter 6

MASS - Software Unit and Integration Test Case Design

6.1 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.

6.2 MASS - Test Design - SRCMutation - Operators

6.2.1 Test design identifier

The test design identifier is

6.3 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.

6.3.1 Features to be tested

Table **??** 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.

For each mutation operator, we aim to ensure

Table 6.1: 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).

6.4 Organization of the Test Cases

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.

Mutation operators We define one test case for each mutation operator The purpose of each test case is to verify that, for a selected CMC produces an output that matches the specifications. This implies that every line of

i.e., alters the line

according to the SDD, which implies that that is, the corresponding syntactically altered version of the software according to operators defined in Table **??**. Note that a mutation operator might produce one or more mutants for a single input, all outputs shall be as expected.

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

Listing 6.1: 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 6.2: ABS test case example.

Listings **??** and **??** introduce an example of source code and test case for the mutation operator ABS, respectively. As shown in Listing **??**, each test case (i) invokes the mutator component selecting the corresponding operator acronym, (ii) defines the expected output for operator, (iii) checks if there are differences between the actual output of the component and the expected output.

All test cases are independent from each other; therefore there is no need of executing test cases in a specific order.

Table **??** provides the complete list of unit test cases that covers all the mutation operators of FAQAS-framework. Column *Input* indicates the operator appearing in the line to be mutated. Note that the mutator shall generate one mutant for each element of the replacement column.

Table 6.2: Organization matrix of unit test cases for source code mutation operators component.

Operator	Input	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

Chapter 7

MASS - Software Unit and Integration Test Case Specification

7.1 General

Table **??** provides the complete list of unit test cases that covers all the mutation operators of FAQAS-framework. Column *Input* indicates the operator appearing in the line to be mutated. Note that the mutator shall generate one mutant for each element of the replacement column.

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.

7.2 Organization of the Test Cases

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

Listing 7.1: C function example.

```
$MUTATOR --compilation "$FILE -o test" --operators ABS

EXPECTED="double function() {\ndouble a = 3;\nreturn -(a);\n}"

tst=`diff test.mut.3.1_1_8.ABS.function.c <(echo -e $EXPECTED) | wc -l`

if [ $tst -eq 0 ];then
    echo -e "TEST abs_val PASSED"
else
    echo -e "TEST abs_val FAILED"
fi
```

Listing 7.2: ABS test case example.

Listings **??** and **??** introduce an example of source code and test case for the mutation operator ABS, respectively. As shown in Listing **??**, each test case (i) invokes the mutator component selecting the corresponding operator acronym, (ii) defines the expected output for operator, (iii) checks if there are differences between the actual output of the component and the expected output.

All test cases are independent from each other; therefore there is no need of 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.

Table 7.1: Organization matrix of unit test cases for source code mutation operators component.

Operator	Input	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