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 29, 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

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 the structure described in ECSS-E-ST-40C Annex B.

1.1 Applicable and reference documents

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

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 Unit Testing and Software Integration Testing

3.1 Organization

The unit tests are to be prepared by SnT in the form of Bash shell scripts. Testing will be conducted by SnT personnel.

If any software problems occur during testing a development issue shall be raised in GitLab Issue Tracker, which shall be amended by SnT personnel.

3.2 Resource Summary

Unit 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.

3.3 Responsibilities

The unit tests are to be prepared by SnT personnel.

The tests will be conducted by a SnT specialist who will also have the responsibility of reporting any occurring software issues.

3.4 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.

3.5 Personnel and Personnel Training Requirements

Unit testing can be performed by a single person using a general script. No special training is needed.

3.6 Risks and Contingencies

The unit testing campaign does not have any risk.

Chapter 4

Software Unit Testing and Integration Testing Approach

4.1 Unit/Integration Testing Strategy

OSCAR: Possibly integration testing might be the use of FAQAS-framework with the case studies?

4.2 Tasks and Items under Test

The items to be tested are all the mutation operators listed in Chapter 5.

4.3 Test Pass - Fail Criteria

The unit test of a procedure can be regarded as passed if all of the following is true:

- The test executed without failure (did not throw an error, etc.).
- All tests were executed.
- The general script shows all unit test cases as passed.

4.4 Manually and Automatically Generated Code

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

Chapter 5

Software Unit and Integration Test Case Specification

5.1 General

The FAQAS-framework unit test suite covers the source code mutation component. In particular, it provides one test case for each of the mutation operators of the FAQAS-framework. Table 5.1 shows the complete list of mutation operators.

The set of FAQAS-framework 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.

Table 5.1: 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	$\{(l_1, l_2) \mid (l_1, l_2) \in \{(0, -1), (l_1, -l_1), (l_1, 0),$ $(\mathit{true}, \mathit{false}), (\mathit{false}, \mathit{true})\}\}$

*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).

5.2 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.

The purpose of each test case is to verify that the component produces the expected results according to the SDD, that is, the corresponding syntactically altered version of the software according to operators defined in Table 5.1. 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 5.1: 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 5.2: ABS test case example.

Listings 5.1 and 5.2 introduces an example of source code and test case for the mutation operator ABS, respectively. As shown in Listing 5.2, 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 tests in a defined order.

Table 5.2 introduces the complete list of unit test cases that covers all the mutation operators of FAQAS-framework. Note that the mutator shall generate one mutant for each element of the replacement column.

Table 5.2: 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