securityandtrust.lu

FAQAS Framework
SUM
Software User Manual

O. Cornejo, F. Pastore, E. Viganò

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

ITT-1-9873-ESA-FAQAS-SUM

Issue 4, Rev. 2

November 23, 2021

UNIVERSITÉ DU LUXEMBOURG

EUROPEAN SPACE AGENCY. CONTRACT REPORT.
The work described in this report was done under ESA contract. Responsibility for the contents resides in the author or organisation that prepared it.

Revisions

Issue Number	Date	Authors	Description
ITT-1-9873-ESA-FAQAS-SUM Issue 1 Rev. 1	March 31th, 2021	Oscar Cornejo, Fabrizio Pastore	Initial release.
ITT-1-9873-ESA-FAQAS-SUM Issue 2 Rev. 2	April 14th, 2021	Fabrizio Pastore, Oscar Cornejo	ESA comments addressed.
ITT-1-9873-ESA-FAQAS-SUM Issue 3 Rev. 1	September 20th, 2021	Fabrizio Pastore, Oscar Cornejo, Enrico Viganò	Added Chapter 5. Added Chapter 8. Added Chapter 13. Added Chapter 6. Added Chapter 9. Added Chapter **??**.
ITT-1-9873-ESA-FAQAS-SUM Issue 3 Rev. 2	October 7th, 2021	Fabrizio Pastore, Oscar Cornejo, Enrico Viganò	Revised version.
ITT-1-9873-ESA-FAQAS-SUM Issue 3 Rev. 3	October 15th, 2021	Fabrizio Pastore, Oscar Cornejo, Enrico Viganò	Updated SEMuS commands according to improvements. Added libUtil example.
ITT-1-9873-ESA-FAQAS-SUM Issue 4 Rev. 1	October 29th, 2021	Fabrizio Pastore, Oscar Cornejo, Enrico Viganò	Addressed A6 (Section 5.1) and A13 (Section 12.1). Also, updated Chapter13; added Chapters **??** and **??**.
ITT-1-9873-ESA-FAQAS-SUM Issue 4 Rev. 2	November 15th, 2021	Fabrizio Pastore, Oscar Cornejo, Enrico Viganò	Added Section 9.3 to address request from GSL for CI/CD use of the tool.

Contents

Chapter 1

Introduction

This document is the Software User Manual (SUM) of the software delivered by the ESA activity ITT-1-9873-ESA (i.e., 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 (DAMAt), a toolset implementing data-driven test generation.

This document follows the structure described in ECSS-E-ST-40C Annex B. Chapters' titles indicate the name of the software referred to.

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.
- MLFS: Mathematical Library for Flight Software
- SUT: Software under test, i.e, the software that should be mutated by means of mutation testing.
- WP: Work package
- HPC: High Performance Computing

Chapter 3

External View of the Software

The FAQAS-framework is delivered as a compressed archive consisting of source files and installers. The following bulletpoints provide a description of the archive's structure once uncompressed:

- `MASS/`
  - `SRCMutation/`: contains the source files of the component that performs code-driven mutations.
  - `llvm-build.sh`: build script that compiles the SRCMutation component
  - `PythonWrappers/`: contains Python script wrappers that facilitate code-driven mutations.
  - `MASS/`: contains all the executable files and scripts that implement the methodology for code-driven mutation testing supported by the FAQAS-Framework. They are listed below.
    * `FAQAS-Setup`: contains the necessary Bash scripts to install the FAQAS-Framework.
    * `FAQAS-GenerateCodeCoverageMatrixes`: contains the Bash scripts providing procedures to collect code coverage from the SUT.
    * `FAQAS-GenerateMutants`: contains a Bash script that invokes the `SRCMutation` component to generate mutants.
    * `FAQAS-CompileOptimizedMutants`: contains the scripts (in Python and Bash) that provide the procedures to compile mutants and filter equivalent and redundant mutants based on trivial compiler optimizations.
    * `FAQAS-CompileAndExecuteMutants`
      · `FAQAS-GeneratePrioritizedTestSuite`: contains the Python and Bash scripts that provide the procedures to generate prioritized and reduced test suites from the SUT.
      · `FAQAS-CompileAndExecute`: contains the Python and Bash scripts that provide the procedures to compile and execute the mutants against the SUT test suite. It also provides the procedures to determine the mutation stopping criterion (i.e., mutant sampling).
      · `FAQAS-IdentifyEquivalentAndRedundantMutants`: contains the Python and Bash scripts that provides the procedures to identify equivalent mutants based on code coverage.
    * `FAQAS-MutationScore`: contains the Python and Bash scripts that provide the procedures to compute the mutation score and provide summarized information about the code-driven mutation testing process.

- DAMAt/
  - damat_pipeline
    - DAMAt_configure.sh: this script defines the necessary variables for the execution of *DAMAt*. They shall be set by the engineer.
    - DAMAt_probe_generation.sh: this script set the variables necessary to generate the data mutation API and execute the python script generateDataMutator.py to generate them.
    - DAMAt_mutants_launcher.sh: this script starts the *DAMAt* pipeline.
    - generateDataMutator.py: this is the script that generates the *DAMAt* mutation API.
    - DDB_TEMPLATE_header.c and DDB_TEMPLATE_footer.c: these are templates used to generate the *DAMAt* API by generateDataMutator.py
    - DAMAt_compile.sh: this is a stub of the script used to compile a mutant, which shall be completed by the engineer.
    - DAMAt_run_tests.sh: this is a stub of the script used to run the tests, which shall be completed by the engineer.
    - data_analysis: a folder containing five python scripts used for the generation of the final results:
      - beautify_results.py: this script renders the raw results from the execution of the tests in a more readable format.
      - get_coverage.py: this script analizes the results of the fault model coverage.
      - get_operator_coverage.py: this script analizes the results of the operator coverage.
      - get_stats.py: this script produces statistics from the mutants' execution.
      - get_final_results.py: this script produces a summary of the execution of *DAMAt*.
    - pipeline_scripts: a folder containing the four scripts that make up the *DAMAt* pipeline:
      - DAMAt_obtain_coverage.sh: this script obtains fault model coverage data in order to execute only the tests that cover each mutant.
      - get_mutant_test_list.py: this script produces the list of test against which avery mutant shall be executed.
      - DAMAt_compile_and_run_mutants.sh: this scripts compile each mutant and run it against the SUT test suite.
      - DAMAt_data_analysis.sh: this script executes all the data analysis steps at the end of the execution of the *DAMAt* pipeline
    - fault_model.csv: an example of a *DAMAt* fault model in csv format.
    - tests.csv : an example of list of test cases and nominal times in csv format.
    - automated_probe_insertion: this folder contains the necessary scripts for automating the probe insertion.
      - DAMAt_probe_insertion.sh: this script backups the file to instrument and then executes DAMAt_insert_probes.py.
      - DAMAt_insert_probes.py: this script replaces specific comments in the file to instrument with the mutation probes.
      - DAMAt_probe_removal.sh: this script restores the original file from the template.
      - DAMAt_insertion_test.sh: this script runs the test cases for the automated probe insertion procedure.

        - `test_files`: this folder contains the necessary files for executing the test cases.
    - `mutator`: contains the testing environment for the *DAMAt* API.
        * `src`: contains the source code of the unit tests for *DAMAt* and the script for launching them.
            - `runTests.sh`: this script execute all the unit test cases.
            - `cleanTests.sh`: this script clean the results of previous tests.
            - `tests`: this folder contains the source code for the unit test cases.
            - `generateDataMutator.py`: this is the script that generates the *DAMAt* mutation API.
            - `DDB_TEMPLATE_header.c` and `DDB_TEMPLATE_footer.c`: these are templates used to generate the *DAMAt* API by `generateDataMutator.py`.
            - `CoverageReportHeader.csv`: contains a template used by `FMcoverage.py` and `FMcoverage2.py`.
            - `getCoverage.sh`, `FMcoverage.py` and `FMcoverage2.py`: these scripts evaluate the coverage when requested by a unit test.
            - `executeTest.sh`, `executeTest_gcc.sh`: these scripts are used by `runTests.sh` to compile and execute the unit tests.
            - `executeTestCoverage.sh`, `executeTestCoverage2.sh` and `executeTestCoverage_gcc.sh`: these scripts are used by `runTests.sh` to compile and execute the unit tests and evaluate the Fault Model coverage.
            - `executeTestJustOnce.sh` and `executeTestProbability.sh`: these scripts are used by `runTests.sh` to compile and execute unit tests that need particular compilation macros enabled.

- `SEMuS/`
    - `underlying_test_generation/`: contains the source files of the component that invokes KLEE-SEMu.
    - `pre_semu/`: contains the source files of the component that prepares the meta-mutant file to be processed by KLEE-SEMu.
    - `ktest_to_unittest/`: contains the source files that implements the component that parses the output of KLEE (i.e., KLEE tests) and converts it to readable C unit test cases.
    - `case_studies/`: contains the configuration files, SUT source codes, and SEMuS launchers for the case studies ASN.1 and MLFS.
        * `scripts/`: contains the configuration files and launchers of the case study.
        * `util_codes/`: folder containing the generated test templates for the case study.
        * `WORKSPACE/`: folder containing the SUT source code and list of live mutants.
    - `Dockerfile`: text document file that contains all the commands necessary to build a Docker container with all the dependencies of SEMuS.
    - `cd_semu_docker.sh`: bash script that automates the execution of the toolset through the use of Docker.
    - `install_requirements.sh`: build script that installs SEMuS' requirements.
    - `requirements.txt`: list of Python packages to be parsed by `install_requirements.sh`.

Chapter 4

MASS - Operations Environment

4.1 Hardware Configuration - Single Machine

The *MASS* toolset uses a host computer with the Linux operating system. *MASS* needs the following hardware requirements to be executed on single machines:

- x86_64 PC architecture
- 4 096 MB of RAM
- Intel i3 (or equivalent) processor

4.2 Hardware Configuration - High Performance Computing (HPC)

The *MASS* toolset can be also executed on High Performance Computing (HPC) infrastructures. An HPC infrastructure is a supercomputer where the computing resources of several single computer nodes are combined to achieve a high level of performance thanks to parallel execution and distribution of tasks. For replicability, in the following, we describe the main characteristics of the HPC of the University of Luxembourg, where MASS experiments had been performed.

- HPC CPU processors: Intel Xeon E5-2680 v4 (2.4 GHz)
- HPC nodes memory: 112 gb
- Operative system: CentOS 7 x86_64 GNU/Linux
- Job scheduler: `slurm 20.11.7`

Instead, these are the minimum hardware configurations to be set on HPCs for a single job:

- CPU per task: 2
- Memory per CPU: 4 096 MB of RAM

4.3 Software Configuration

MASS integrates the `SRCMutation` component from `SRCIRor` mutation analysis tool[1]. The installation of `SRCMutation` makes part of *MASS* installation (see 7.1).

In order to perform its tasks, *MASS* also requires a few external components listed in the following:

- clang-3.8
- r-base (package binom)
- jq
- Python-3.7 or higher (packages numpy and scipy)
- bash 4.4 or higher

Note that these components shall be installed by the user before the actual MASS installation.

[1] https://github.com/TestingResearchIllinois/srciror

Chapter 5

SEMUS - Operations Environment

5.1 Hardware Configuration - Single Machine

The *SEMuS* toolset uses a host computer with the Linux operating system. *SEMuS* needs the following hardware requirements to be executed on single machines:

- x86_64 PC architecture
- 4 096 MB of RAM
- Intel i3 (or equivalent) processor

A6

Execution with HPC is feasible, however, given that test generation is fast it can be achieved also with a single machine. Also, even in case of relying on an HPC, it is first necessary to generate test templates, which requires manual intervention and thus it is easier to be performed with a standard development environment (i.e., a single machine).

5.2 Software Configuration

SEMuS integrates *SEMu*, a symbolic execution-based mutation analysis engine, and thus it is necessary to install it before running *SEMuS*. *SEMuS* includes a Dockerfile that provides the necessary commands to install *SEMu* and *SEMuS*.

In order to perform its tasks, *SEMuS* requires these components previously installed:

- `Docker` (`docker-ce` `docker-ce-cli` `containerd.io`)

Chapter 6

DAMAt - Operations Environment

6.1 Hardware Configuration - Single Machine

The *DAMAt* toolset uses a host computer with the Linux operating system. *DAMAt* needs the following hardware requirements to be executed on single machines:

- x86_64 PC architecture
- 4 096 MB of RAM
- Intel i3 (or equivalent) processor

6.2 Software Configuration

In order to perform its tasks, *DAMAt* also requires a few external components listed in the following:

- `Python 3.6.8` or higher
- `GNU bash, version 4.2.46` or higher

Chapter 7

MASS - Operations Manual

7.1 Set-up and Initialization

MASS depends on LLVM for source code mutation. For this reason, a full LLVM-3.8.1 installation is necessary preceding the installation of the `SRCMutation` component. For this procedure, a Bash script is provided.

The following shell command installs the corresponding dependencies and the `SRCMutation` component.

```
$ ./llvm-build.sh
```

7.1.1 Dependencies

- Linux packages: `clang 3.8`, `r-base`, `jq`, `Python 3.7` or higher
- R packages: `binom`
- Python packages: `numpy`, `scipy`

7.2 Getting started

7.2.1 Initialization of the MASS workspace

MASS creates a workspace folder where all the steps from the methodology are stored.

An installation Bash script is provided for the creation of this workspace, the script can be found on `$FAQAS/MASS/FAQAS-Setup/install.sh`

To use the installation script the shell variable `INSTALL_DIR` has to be set:

```
$ export INSTALL_DIR=/opt/DIRECTORY
```

If the `INSTALL_DIR` directory must be binded inside a container. In addition, also the shell variable `EXECUTION_DIR` has to be set. This step is optional.

For instance, *MASS* has been installed on `/opt/MASS_WORKSPACE` (i.e., the `INSTALL_DIR`), and *MASS* will be executed inside a container, but on a different directory such as `/home/user/MASS_WORKSPACE` (i.e., the `EXECUTION_DIR`). The use of both environment variables enable this differentiation.

After setting the corresponding environment variables, the following commands are necessary to create the *MASS* workspace folder:

```
$ cd $FAQAS/MASS/FAQAS-Setup
$ ./install.sh
```

Once the installation folder has been created, the folder shall contain the following structure and files:

- `Launcher.sh`: *MASS* single launcher; the script executes all the steps of the methodology in one command.
- `mass_conf.sh`: *MASS* configuration file; the file has to be configured before being able to execute *MASS*.
- `mutation_additional_functions.sh`: Bash script that must be filled by the application engineer before executing *MASS*.
- `MASS_STEPS_LAUNCHERS/`: folder containing all the single launchers for each step of the *MASS* methodology.
  - `MASS_STEPS_LAUNCHERS/PrepareSUT.sh`: launcher for the script that prepares the SUT and collects information about the SUT test suite.
  - `MASS_STEPS_LAUNCHERS/GenerateMutants.sh`: launcher for the generation of mutants.
  - `MASS_STEPS_LAUNCHERS/CompileOptimizedMutants.sh`: launcher for the trivial compiler optimization step.
  - `MASS_STEPS_LAUNCHERS/OptimizedPostProcessing.sh`: launcher for the post-processing of the trivial compiler optimization step.
  - `MASS_STEPS_LAUNCHERS/GeneratePTS.sh`: launcher for the generation of prioritized and reduced test suites.
  - `MASS_STEPS_LAUNCHERS/ExecuteMutants.sh`: launcher for the execution of mutants against the SUT test suite.
  - `MASS_STEPS_LAUNCHERS/IdentifyEquivalents.sh`: launcher for the identification of equivalent mutants based on code coverage.
  - `MASS_STEPS_LAUNCHERS/MutationScore.sh`: launcher for the computation of the mutation score and final reporting.
  - `MASS_STEPS_LAUNCHERS/PrepareMutants_HPC.sh`: HPC launcher that prepares the mutants workspace for the execution on HPCs.
  - `MASS_STEPS_LAUNCHERS/ExecuteMutants_HPC.sh`: HPC launcher that executes mutants on HPCs.
  - `MASS_STEPS_LAUNCHERS/PostMutation_HPC.sh`: HPC launcher that assesses past mutant executions, and decides whether more mutant executions are needed.

7.2.2 MASS Configuration

There are three Bash scripts that should be edited by the engineer to configure *MASS*. These three scripts enable *MASS* to correctly identify the SUT paths (e.g., source code folder, test suite folder), the SUT compilation commands, the SUT test suite execution commands, and the configuration of *MASS* itself (e.g., trivial compiler optimizations flags, mutant selection strategy, sampling rate).

Table 7.1 provides a summary of *MASS* configuration files, their parameters, and a brief description. A detailed description of the MASS configuration files follows.

Table 7.1: *MASS* parameters to be configured.

Script Name	Parameter	Optional	Description
mass_conf.sh	SRCIROR	No	Installation directory of the FAQAS-framework.
	APP_RUN_DIR	No	Workspace directory of *MASS*.
	BUILD_SYSTEM	No	Specifies the building system type.
	PROJ	No	Path of the SUT root directory.
	PROJ_SRC	No	Path of the SUT source directory.
	PROJ_TST	No	Path of the SUT test directory.
	PROJ_COV	No	Path of the directory with SUT coverage information.
	PROJ_BUILD	No	Path of the directory where the compiled binary is stored.
	COMPILED	Yes	Filename of the compiled file/library.
	ORIGINAL_MAKEFILE	No	Path to the original build script.
	COMPILATION_CMD	No	Compilation command of the SUT.
	ADDITIONAL_CMD	Yes	Additional compilation commands of the SUT.
	ADDITIONAL_CMD_AFTER	Yes	Command to be executed after each test case execution.
	TCE_COMPILE_CMD	No	Compilation command for TCE analysis.
	CLEAN_CMD	No	Clean command for the SUT.
	COVERAGE_NOT_INCLUDE	Yes	Folder(s) not to be included during coverage analysis.
	GC_FILES_RELATIVE_PATH	Yes	Relative path to the location of GCOV files.
	HPC	No	Specifies whether *MASS* will be executed on a HPC.
	FLAGS	No	TCE flags to be tested.
	PRIORITIZED	No	Specifies whether *MASS* should be executed with a prioritized test suite.
	SAMPLING	No	Specifies the mutant sampling technique.
	RATE	Yes	Specifies the mutant sampling rate.
PrepareSUT.sh	None	No	Commands shall be provided manually.
mutation_additional_functions.sh	run_tst_case	No	Implementation of the Bash function run_tst_case that executes the test case passed as a parameter.

7.2.2.1 Edit mass_conf.sh

Within file `$INSTALL_DIR/mass_conf.sh` there are multiple environment variables that must be set; they are shown in Listing 7.1.

```
1  # set SRCIROR path
2  export SRCIROR=
3
4  # set workspace directory path where MASS files can be stored (i.e., $INSTALL_DIR)
5  export APP_RUN_DIR=
6
7  # specifies the building system, available options are "Makefile" and "waf"
8  export BUILD_SYSTEM=
9
10 # directory root path of the SUT
11 export PROJ=
12
13 # directory source path of the SUT
14 export PROJ_SRC=
15
16 # directory test path of the SUT
17 export PROJ_TST=
18
19 # directory coverage path of the SUT
20 export PROJ_COV=
```

```
21
22  # directory path where the compiled binary is stored
23  export PROJ_BUILD=
24
25  # list of folders not to be included during coverage analysis, name folders shall be separated by
      '\|'
26  export COVERAGE_NOT_INCLUDE=
27
28  # filename of the compiled file or library
29  export COMPILED=
30
31  # path to the original build script
32  export ORIGINAL_MAKEFILE=
33
34  # compilation command of the SUT, the command shall be specified as a Bash array, e.g., (). Special
      characters shall be escaped.
35  export COMPILATION_CMD=
36
37  # additional commands for compiling the SUT (e.g., setup of workspace), the command shall be
      specified as a Bash array, e.g., (). Special characters shall be escaped.
38  export ADDITIONAL_CMD=
39
40  # command to be executed after each test case execution (optional), the command shall be specified as
      a Bash array, e.g., (). Special characters shall be escaped.
41  export ADDITIONAL_CMD_AFTER=
42
43  # compilation command for TCE analysis, the command shall be specified as a Bash array, e.g., ().
      Special characters shall be escaped.
44  export TCE_COMPILE_CMD=
45
46  # command to clean installation of the SUT, the command shall be specified as a Bash array, e.g., ().
      Special characters shall be escaped.
47  export CLEAN_CMD=
48
49  # relative path to location of gcov files (i.e., gcda and gcno files)
50  export GC_FILES_RELATIVE_PATH=
```

Listing 7.1: Excerpt of mass_conf.sh file.

Furthermore, the following specific *MASS* variables must be set (See Listing 7.2):

```
1  # specify if MASS will be executed on an HPC, possible values are "true" or "false"
2  export HPC=
3
4  # TCE flags to be tested, the flags shall be specified as a Bash array, e.g., ("-O0" "-O1").
5  export FLAGS=
6
7  # set if MASS should be executed with a prioritized and reduced test suite, possible values are "true
     " or "false"
8  export PRIORITIZED=
9
10 # set sampling technique, possible values are "uniform", "stratified", "fsci", and "no"
11 # note: if "uniform" or "stratified" is set, $PRIORITIZED must be "false"
12 export SAMPLING=
13
14 # set sampling rate if whether "uniform" or "stratified" sampling has been selected
15 export RATE=
```

Listing 7.2: *MASS* specific variables. Excerpt of mass_conf.sh file.

7.2.2.2 Edit PrepareSUT.sh

To configure *MASS* to work with the SUT, the engineer should also edit the Bash file `$INSTALL_DIR/MASS_STEPS_LAUNCHER/PrepareSUT.sh`. The following actions shall be performed by the engineer:

1. Provide commands to generate a compilation database file `compile_commands.json` of the

SUT. Note that the paths defined within the database file must be full paths. The compilation database file provides the necessary compilation commands of each source for the source mutation step of the methodology.

2. Provide commands to compile the SUT;
3. Provide additional commands to prepare the SUT workspace (optional);
4. Provide commands to execute the SUT test suite iteratively over each test case; more precisely, the engineer should do the following:
   - After the command executing a a test case add a call to `$MASS/FAQAS-GenerateCodeCoverageMatrixes/FAQAS-CollectCodeCoverage.sh` script;
   - the script `FAQAS-CollectCodeCoverage.sh` shall be invoked with two arguments: (i) the test case name, and (ii) the time taken to run the test case in seconds.

7.2.2.3 Mutation Script Configuration

The mutation script configuration file is the Bash file `mutation_additional_functions.sh`. In it, the engineer is expected to implement the Bash function `run_tst_case`. This function shall receive as argument the name of the test case to be executed. It should execute the command for running the specified test case. The function shall return 0, if the test case passes; it shall return 1 if the test case fails.

7.2.2.4 Build Script for Compiler Optimizations

The SUT engineer shall provide a build script for the SUT. Such script shall be placed in the same folder where the original build script resides (a different name shall be used). The script shall have the following characteristics:

- The script shall not contain any debugging flag within compilation/linking commands;
- The script shall not contain any code coverage flag within compilation/linking commands;
- The script shall contain a placeholder for the compiler optimization option, specifically the placeholder `TCE`;
- The script shall contain a 'sort' command in the list of source files to be compiled, to ensure that source files are always compiled in the same order;

  In the case of a Makefile, it can be achieved with the following command:

  ```
  SRC=$(sort $(wildcard $(SourceFolder/*.c))
  ```

- The script shall be named the same as the original build script, but with an ending '.template'.

7.2.3 Running MASS on Single Machines

MASS can be executed in two modes, *single machine* and *shared resources mode*. The single machine mode provides the advantage of running *MASS* in an unsupervised mode, executing the methodology on one step. Instead, the shared resources facilities mode gives the possibility of

running multiple steps in parallel and executing a higher number of mutants, in a similar time frame, with respect to the single machine mode. In this section we describe the *single machine*, Section 7.2.4 covers the *shared resources mode*.

7.2.3.1 One Step Launcher

The single machine mode gives the possibility of running *MASS* in one step, by executing all the eight steps of the framework with one command. The one step launcher will execute the following steps sequentially:

1. PrepareSUT
2. GenerateMutants
3. CompileOptimizedMutants
4. OptimizedPostProcessing
5. GeneratePTS
6. ExecuteMutants
7. IdentifyEquivalents
8. MutationScore

To execute the one step launcher, the following command shall be provided:

```
$ ./Launcher.sh
```

7.2.3.2 Multiple Step Launchers

The single machine mode also gives the possibility to run *MASS* by executing all the eight steps of the framework through independent commands. The multiple steps of the methodology and their respective commands are described in the following.

1. **Prepare the SUT** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/PrepareSUT.sh
```

2. **Generate Mutants** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/GenerateMutants.sh
```

3. **Compile Optimized Mutants** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/CompileOptimizedMutants.sh
```

4. **Compile Optimized Mutants Post-Processing** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/OptimizedPostProcessing.sh
```

5. **Generate Prioritized and Reduced Test Suites** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/GeneratePTS.sh
```

6. **Execute mutants** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/ExecuteMutants.sh
```

7. **Identify Equivalent Mutants based on Code Coverage** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/IdentifyEquivalents.sh
```

8. **Computer Mutation Score** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/MutationScore.sh
```

7.2.4 Running MASS on Shared Resources Facilities

Given that resources from HPC infrastructures has to be requested for every performed tasks, it is not possible to run all the steps from *MASS* in one step. However, since resources can be requested accordingly, *MASS* can perform multiple steps simultaneously, enhancing the capabilities of the toolset. With an HPC, for example, *MASS* could analyze more mutants than if *MASS* was executed on a single machine.

The multiple steps of the methodology and their respective commands are described in the following.

1. **Prepare the SUT** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/PrepareSUT.sh
```

2. **Generate Mutants** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/GenerateMutants.sh
```

3. **Compile Optimized Mutants**: A parameter shall be passed to the launcher script to indicate the chosen optimization level. If six levels of optimizations are defined, then numbers between zero and five can be provided.

```
$ level=0
$ ./MASS_STEPS_LAUNCHER/CompileOptimizedMutants.sh $level
```

4. **Compile Optimized Mutants Post-Processing** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/OptimizedPostProcessing.sh
```

5. **Generate Prioritized and Reduced Test Suites** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/GeneratePTS.sh
```

6. **Prepare mutants** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/PrepareMutants_HPC.sh
```

7. **Execute mutants**: The launcher script receives two parameters: the first parameter is the number of the mutant to be executed, defined as $1..M$, being M the total number of mutants. The total number of mutants can be derived from the folder `$INSTALL_DIR/hpc-src-mutants`. The second parameter defines if the test suite has to be executed in a reduced fashion or not. The possible values are "true" and "false".

```
$ nr_mutant=1
$ reduced="false"
$ ./MASS_STEPS_LAUNCHER/ExecuteMutants_HPC.sh $nr_mutant $reduced
```

8. **Post-mutation execution**: The launcher script receives two numbers as parameters, a minimum and a maximum value, that represent the range of mutants to assess. The assessment consists of evaluating if more mutant executions are needed.

```
$ min=1
$ max=700
$ ./MASS_STEPS_LAUNCHER/PostMutation_HPC.sh $min $max
```

9. **Identify Equivalent Mutants based on Code Coverage** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/IdentifyEquivalents.sh
```

10. **Computer Mutation Score** step can be executed with the following command:

```
$ ./MASS_STEPS_LAUNCHER/MutationScore.sh
```

7.2.5 MASS results

After the execution of *MASS* the results are stored in dedicated folders. Such folders are defined as follows:

1. Prepare SUT: the results of this step are stored in the folder `COV_FILES`. Besides the coverage files, it contains (i) the coverage matrices, and (ii) the timeout file.
2. Generate Mutants: the results of this step are stored in the folder `src-mutants`, it contains all mutant sources.
3. Compile Optimized Mutants: the results of this step are stored in the folder `COMPILED` that contains one folder for each optimisation level.
4. Compile Optimized Mutants Post-Processing: the results of this step are stored in the folder `COMPILED`. Specifically, it generates four lists: (i) list of all mutants, (ii) list of nonequivalent and nonredundant mutants, (iii) list of equivalent mutants, and (iv) list of redundant mutants.
5. Generate Prioritized and Reduced Test Suites: the results of this step are stored in the folder `PRIORITIZED`, which contains one file with the prioritized and reduced test suites, and one file with the prioritized test suites.
6. Execute Mutants: the results of this step are stored in the folder `MUTATION`. This step produces several files and folders:
   - `main.csv`: complete mutation traces

- `sampled_mutants`: list of sampled mutants to be executed
- `all_live`: list of live mutants
- `all_killed`: list of killed mutants
- `traces_live`: live mutants traces
- `traces_killed`: killed mutants traces
- `mutant folder`: there is one folder for each executed mutant, this folder contains the mutant code coverage information and test cases execution logs.

7. Identify Equivalent Mutants based on Code Coverage: the results of this step are stored in the folder `DETECTION`.
8. Compute Mutation Score: the results of this step are stored in folder `RESULTS`. Particularly, the file `MASS_RESULTS` contains the final output of *MASS*.

7.3 Normal Termination

Each methodology step of *MASS* is executed when invoked and computes a result. There is no software interruption foreseen during the computation and the procedure terminates by returning the result. If the engineer decides to interrupt *MASS* execution, it can be done by sending a signal interrupt `SIGINT` to the running process.

7.4 Error Conditions

There is no error condition handling in the FAQAS-framework. If all preconditions are met, there should not be any error.

7.5 Recover Runs

If for any reason the execution of *MASS* is interrupted, an engineer can restart the process from a specific task if all preconditions are met. This is possible since each *MASS* step work by processing data that is permanently stored by previous steps.

Chapter 8

SEMUS - Operations Manual

8.1 Set-up and Initialization

SEMuS depends directly on *SEMu* for test inputs generation, and *MASS* for mutant generation.

For this reason, we provide a Dockerfile containing all the commands necessary to download, install and prepare a full installation of *SEMu* and MASS. The Dockerfile can be found at `SEMuS/Dockerfile`. The installation will take place the first time the Docker image is created through the Bash script `cd_semu_docker.sh`, no further installation is required for now.

8.1.1 Dependencies

- Linux packages:
  - `docker-ce`
  - `docker-ce-cli`
  - `containerd.io`

8.2 Getting started

8.2.1 Initialization of the *SEMuS* workspace

SEMuS needs a dedicated folder structure for running the test generation on a case study. For this reason, the engineer shall create the following directory structure under the folder `case_studies`, assuming the name of the case study is stored in the variable `$SUT`:

- `case_studies/$SUT/`:
  - `scripts/`
  - `util_codes/`
  - `WORKSPACE/`

Table 8.1: SEMuS parameters to be configured.

Parameter	Description
FAQAS_SEMU_CASE_STUDY_TOPDIR	Root folder of the case study
FAQAS_SEMU_CASE_STUDY_WORKSPACE	SEMuS workspace for the case study
FAQAS_SEMU_OUTPUT_TOPDIR	SEMuS output folder, to be placed inside the workspace
FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR	Root folder for storing the generated mutants
FAQAS_SEMU_REPO_ROOTDIR	Root folder of the case study source code
FAQAS_SEMU_ORIGINAL_SOURCE_FILE	Path of the source file under analysis
FAQAS_SEMU_COMPILE_COMMAND_SPECIFIED_SOURCE_FILE	Name of the source file under analysis
FAQAS_SEMU_GENERATED_MUTANTS_DIR	Folder for storing the generated mutants for the specified source file
FAQAS_SEMU_BUILD_CODE_FUNC_STR	Bash function for building the source file under analysis, to be specified in string format
FAQAS_SEMU_BUILD_LLVM_BC	Bash function for building the source file to LLVM bitcode
FAQAS_SEMU_META_MU_TOPDIR	Root folder for the meta mutant
FAQAS_SEMU_GENERATED_META_MU_SRC_FILE	Path of the source file (i.e., C file) of the meta mutant
FAQAS_SEMU_GENERATED_META_MU_BC_FILE	Path of the source file (i.e., LLVM bitcode file) of the meta mutant
FAQAS_SEMU_GENERATED_META_MU_MAKE_SYM_TOP_DIR	Folder for storing intermediate files for the generated inputs
FAQAS_SEMU_GENERATED_TESTS_TOPDIR	Folder for storing the generated inputs
FAQAS_SEMU_TEST_GEN_TIMEOUT	Timeout in seconds for the test generation process
FAQAS_SEMU_HEURISTICS_CONFIG	Configuration array for SEMu heuristics
FAQAS_SEMU_TEST_GEN_MAX_MEMORY	Maximum test generation memory in MB
FAQAS_SEMU_STOP_TG_ON_MEMORY_LIMIT	Parameter to stop test generation when the memory limit is reached
FAQAS_SEMU_TG_MAX_MEMORY_INHIBIT	Parameter to stop forking states when the memory limit is reached

The *SEMuS* repository we provide, contains already the configured files and the source code of two open-source case studies, the ASN.1 and MLFS. After the creation of the folder structure, the engineer shall copy the scripts for (1) creating the mutants, (2) running the toolset, (3) configuring *SEMuS*. The scripts shall be copied from the ASN case study folder. The commands for copying these files are provided below:

```
$ cd case_studies/ASN/scripts
$ cp create_mutants.sh run.sh docker_run.sh faqas_semus_config.sh ../../$SUT/scripts
```

At this step, the engineer shall also provide a compilation database file of the SUT, to be placed inside `$SUT/scripts` with the name `compile_commands.json`. Note that the paths defined within the database file must be full paths. The compilation database file provides the necessary compilation commands of each source for the source mutation and the test generation steps of the methodology.

Next, the engineer shall copy the necessary scripts for generating the test templates for guiding the test generation. This can be done through the following commands:

```
$ cd case_studies/ASN/util_codes
$ cp generate_direct.py generate_template_config.json ../../$SUT/util_codes
```

Last, the engineer shall place inside the folder `case_studies/$SUT/WORKSPACE/DOWNLOADED` (1) the SUT source code, and (2) the list of live mutants (provided by *MASS* output).

8.2.2 *SEMuS* Configuration

SEMuS configuration comes in two parts, that is, (1) configuring the Bash script `faqas_semus_config.sh`, and (2) configuring the JSON test templates.

The Bash script enables *SEMuS* to correctly identify the SUT paths (e.g., source code folder), the SUT compilation commands, output folder, and the configuration of *SEMuS* itself (e.g., configuration of the heuristics, maximum memory, test generation timeout).

The different JSON files indicate for each function under test, the values that should be printed out, so *SEMuS* can determine if a mutant has been killed.

Table 8.1 provides a summary of the Bash *SEMuS* configuration file and a brief description. A detailed description of the SEMuS configuration file follows in Section 8.2.2.1.

Table 8.2: Test template generator parameters to be configured.

Parameter	Description
TYPES_TO_INTCONVERT	Specify how to convert a type to int.
TYPES_TO_PRINTCODE	Specify how to print a type.
OUT_ARGS_NAMES	Specify the names of function arguments that are used as function output.
IN_OUT_ARGS_NAMES	Specify the names of function arguments that are used both as function input and output
TYPE_TO_INITIALIZATIONCODE	Specify the initialization statement of a type.
VOID_ARG_SUBSTITUTE_TYPE	Specify the underlying type for a void pointer.
TYPE_TO_SYMBOLIC_FIELDS_ACCESS	Specify, for pointer parameters, the number of elements it points to.

Table 8.2 provides a summary of the JSON *SEMuS* configuration file and a brief description. A detailed description of the JSON configuration file follows in Section 8.2.2.2.

8.2.2.1 Edit faqas_semus_config.sh

Within file `faqas_semus_config.sh` there are multiple environment variables that must be set; they are shown in Listing 8.1.

```
1
2  # Root folder of the case study
3  FAQAS_SEMU_CASE_STUDY_TOPDIR=
4
5  # SEMuS workspace for the case study
6  FAQAS_SEMU_CASE_STUDY_WORKSPACE=
7
8  # SEMuS output folder, to be placed inside the workspace
9  FAQAS_SEMU_OUTPUT_TOPDIR=
10
11 # Root folder for storing the generated mutants
12 FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR=
13
14 # Root folder of the case study source code
15 FAQAS_SEMU_REPO_ROOTDIR=
16
17 # Full path of the source file under analysis
18 FAQAS_SEMU_ORIGINAL_SOURCE_FILE=
19
20 # Relative path of the source file under analysis
21 FAQAS_SEMU_COMPILE_COMMAND_SPECIFIED_SOURCE_FILE=
22
23 # Folder for storing the generated mutants for the specified source file
24 FAQAS_SEMU_GENERATED_MUTANTS_DIR=
25
26 # Bash function for building the source file under analysis, to be specified in string format
27 FAQAS_SEMU_BUILD_CODE_FUNC_STR='FAQAS_SEMU_BUILD_CODE_FUNC() { }'
28
29 # Bash function for building the source file to LLVM bitcode
30 FAQAS_SEMU_BUILD_LLVM_BC() { }
31
32 # Root folder for the meta mutant
33 FAQAS_SEMU_META_MU_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/meta_mu_topdir
34
35 # Path of the source file (i.e., C file) of the meta mutant
36 FAQAS_SEMU_GENERATED_META_MU_SRC_FILE=
37
38 # Path of the source file (i.e., LLVM bitcode file) of the meta mutant
39 FAQAS_SEMU_GENERATED_META_MU_BC_FILE=
40
41 # Folder for storing intermediate files for the generated inputs
42 FAQAS_SEMU_GENERATED_META_MU_MAKE_SYM_TOP_DIR=
43
44 # Folder for storing the generated inputs
45 FAQAS_SEMU_GENERATED_TESTS_TOPDIR=
46
47 # Timeout in seconds for the test generation process
48 FAQAS_SEMU_TEST_GEN_TIMEOUT=
49
50 # Configuration array for SEMu heuristics. The accepted values of 'PSS' are 'RND' for random and 'MDO'
       for minimum distance to output
51 FAQAS_SEMU_HEURISTICS_CONFIG='{
```

```
        "PL":
        "CW":
        "MPD":
        "PP":
        "NTPM":
        "PSS":
}'

# Maximum test generation memory in MB
FAQAS_SEMU_TEST_GEN_MAX_MEMORY=

# Parameter to stop test generation when the memory limit is reached
FAQAS_SEMU_STOP_TG_ON_MEMORY_LIMIT=

# Parameter to stop forking states when the memory limit is reached
FAQAS_SEMU_TG_MAX_MEMORY_INHIBIT=
```

Listing 8.1: faqas_semus_conf.sh file.

Listing 13.1 provides an example of faqas_semus_conf.sh file configured for the ASN case study.

```

FAQAS_SEMU_CASE_STUDY_TOPDIR=../

FAQAS_SEMU_CASE_STUDY_WORKSPACE=$FAQAS_SEMU_CASE_STUDY_TOPDIR/WORKSPACE

FAQAS_SEMU_OUTPUT_TOPDIR=$FAQAS_SEMU_CASE_STUDY_WORKSPACE/OUTPUT

FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/mutants_generation

FAQAS_SEMU_REPO_ROOTDIR=$FAQAS_SEMU_CASE_STUDY_WORKSPACE/DOWNLOADED/casestudy

FAQAS_SEMU_ORIGINAL_SOURCE_FILE=$FAQAS_SEMU_REPO_ROOTDIR/test.c

FAQAS_SEMU_COMPILE_COMMAND_SPECIFIED_SOURCE_FILE=./test.c

FAQAS_SEMU_GENERATED_MUTANTS_DIR=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/test

FAQAS_SEMU_BUILD_CODE_FUNC_STR='
FAQAS_SEMU_BUILD_CODE_FUNC()
{
    local in_file=$1
    local out_file=$2
    local repo_root_dir=$3
    local compiler=$4
    local flags="$5"
    # compile
    $compiler $flags -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -Wshadow -Wundef -
    fdiagnostics-show-option -D_DEBUG -I $repo_root_dir -O0 $in_file -o $out_file $flags
    return $?
}
'

FAQAS_SEMU_BUILD_LLVM_BC()
{
    local in_file=$1
    local out_bc=$2
    eval "$FAQAS_SEMU_BUILD_CODE_FUNC_STR"
    FAQAS_SEMU_BUILD_CODE_FUNC $in_file $out_bc $FAQAS_SEMU_REPO_ROOTDIR clang '-c -emit-llvm'
    return $?
}

FAQAS_SEMU_META_MU_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/meta_mu_topdir

FAQAS_SEMU_GENERATED_META_MU_SRC_FILE=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/test.MetaMu.c

FAQAS_SEMU_GENERATED_META_MU_BC_FILE=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/test.MetaMu.bc

FAQAS_SEMU_GENERATED_META_MU_MAKE_SYM_TOP_DIR=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/"MakeSym-TestGen-
    Input"

FAQAS_SEMU_GENERATED_TESTS_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/test_generation

# timeout in seconds
```

```
FAQAS_SEMU_TEST_GEN_TIMEOUT=7200

# This is the config for SEMU heuristics. The accepted values of 'PSS' are 'RND' for random and 'MDO'
    for minimum distance to output
FAQAS_SEMU_HEURISTICS_CONFIG='{
        "PL": "0",
        "CW": "4294967295",
        "MPD": "0",
        "PP": "1.0",
        "NTPM": "5",
        "PSS": "RND"
}'

# max Test Generation memory in MB
FAQAS_SEMU_TEST_GEN_MAX_MEMORY=2000

# Set to 'ON' to stop test generation when the memory limit is reached
FAQAS_SEMU_STOP_TG_ON_MEMORY_LIMIT='OFF'

# Set this to 'ON' so the the states the sate fork is disabled when the memory limit is reached, to
    avoid going much over it
FAQAS_SEMU_TG_MAX_MEMORY_INHIBIT="ON"
```

Listing 8.2: faqas_semus_conf.sh file for ASN case study.

8.2.2.2 Edit generate_template_config.json

Within file `generate_template_config.json` there are multiple variables that must be set; they are shown in Listing 8.3. These JSON variables are `TYPES_TO_INTCONVERT`, `TYPES_TO_PRINTCODE`, `OUT_ARGS_NAMES`, `IN_OUT_ARGS_NAMES`, `TYPE_TO_INITIALIZATIONCODE`, `TYPE_TO_SYMBOLIC_FIELDS_ACCESS`, `VOID_ARG_SUBSTITUTE_TYPE`, `ARG_TYPE_TO_ITS_POINTER_ELEM_NUM`.

The option TYPES_TO_INTCONVERT specifies a type as key and the type conversion template as value, where the placeholder for the expression to convert should be specified as the string `''`, an example would be `"TYPES_TO_INTCONVERT": "flag": "(int)"`.

The option TYPES_TO_PRINTCODE specifies a type as key and the object printing code as value, where the placeholder for the object to print must be specified as the string `''`. Do not forget to escape the backslashes and double quoted in printf format, an example would be `"TYPES_TO_PRINTCODE": "struct XY *": "printf(\" \" AQAS-SEMU-TEST_OUTPUT: X=%d, Y=%s n\" \"  ->x, ->y)"`.

The option OUT_ARGS_NAMES specifies the names of function arguments that are used as function output (passed by reference for output only) e.g. `"OUT_ARGS_NAMES": ["pErrCode"]`.

The option IN_OUT_ARGS_NAMES Specify the names of function arguments that are used both as function input and output (passed by reference) e.g. `"IN_OUT_ARGS_NAMES": ["inoutArg"]`.

The option TYPE_TO_INITIALIZATIONCODE specifies a type as key and the pre `klee_make_symbolic` statement initialization code as value, the placeholder for the object to initialize must be specified as the string `''`
e.g. `"TYPE_TO_INITIALIZATIONCODE": "struct head": ".next = malloc(sizeof(struct head));
n.next->next = NULL;"`.

The option TYPE_TO_SYMBOLIC_FIELDS_ACCESS specifies how to make an object symbolic (specially useful for objects that are initialized, like pointers). The object type is the dict key and a dict of field accesses and their type is the dict value. The placeholder for the object to make sym-

bolic must be specified as the string '' e.g. "TYPE_TO_SYMBOLIC_FIELDS_ACCESS": "struct head": ".data": "char [3]", ".next->data": "char [3]".

The option VOID_ARG_SUBSTITUTE_TYPE specifies the underlying type for a void pointer (the data type pointed by the void pointer). For instance, if the function is to be called with an `int` array for a void pointer parameter, then set `VOID_ARG_SUBSTITUTE_TYPE` to `int`. Set the value to the empty string (`""`) to let the user specify at runtime on case by case. Set the value to `null` (JSON equivalent to None) to let the user change the types `void` directly in the generated templates, e.g. `VOID_ARG_SUBSTITUTE_TYPE: "char"`.

The option ARG_TYPE_TO_ITS_POINTER_ELEM_NUM specifies, for pointer parameters, the number of elements it points to (must be > 0). This will give the flexibility to set the number of elements the pointer points to. The default value is 1, for non specified types,
e.g. `ARG_TYPE_TO_ITS_POINTER_ELEM_NUM: "int *": 2, "char *": 6`, this let and array of 2 for int pointer and array of 6 for char pointer, and an array of 1 for unsigned pointer.

```
{
    "TYPES_TO_INTCONVERT": {},
    "TYPES_TO_PRINTCODE": {},
    "OUT_ARGS_NAMES": [],
    "IN_OUT_ARGS_NAMES": [],
    "TYPE_TO_INITIALIZATIONCODE": {},
    "TYPE_TO_SYMBOLIC_FIELDS_ACCESS": {},
    "VOID_ARG_SUBSTITUTE_TYPE": "",
    "ARG_TYPE_TO_ITS_POINTER_ELEM_NUM": {}
}
```

Listing 8.3: generate_template_config.json file.

Listing 8.4 provides an example of generate_template_config.json file configured for the ASN case study.

```
{
    "TYPES_TO_INTCONVERT": {"flag": "(int){}"},
    "TYPES_TO_PRINTCODE": {},
    "OUT_ARGS_NAMES": ["pErrCode"],
    "IN_OUT_ARGS_NAMES": [],
    "TYPE_TO_INITIALIZATIONCODE": {},
    "TYPE_TO_SYMBOLIC_FIELDS_ACCESS": {},
    "VOID_ARG_SUBSTITUTE_TYPE": "",
    "ARG_TYPE_TO_ITS_POINTER_ELEM_NUM": {}
}
```

Listing 8.4: generate_template_config.json file for ASN case study.

8.2.3 Running *SEMuS*

To run *SEMuS*, it is necessary to generate the test templates as first step; this can be achieved through the following command:

```
$ case_studies/$SUT/util_codes/generate_direct.py ../WORKSPACE/DOWNLOADED/casestudy/test.c direct \
                    " -I../WORKSPACE/DOWNLOADED/casestudy/" -c generate_template_config.json
```

The previous command will generate inside the directory `case_studies/$SUT/util_codes` one folder for each source under analysis, and inside of these folders, one template for each function under test.

The test generation process can be started with the command shown in Listing 8.2.3.

Note the following:

- the first time the command is invoked, Docker will install all the dependencies of *SEMuS* (i.e., *SEMu* and *MASS*).
- the environment variable `ENV_FAQAS_SEMU_SRC_FILE` has to be set before invoking; this variable indicates the source file for which the test generation will be targeted

```
ENV_FAQAS_SEMU_SRC_FILE=test.c ./docker_run.sh [<starting-step>] [<mutants-list-file> <output-dir-for-pre-semu-and-semu>]
```

Where:

- starting-step: is the step of the pipeline from which to start, possible values are `{mutation, compile, presemu, semu, unittests}`
  - mutation: the process starts from the mutant generation step
  - compile: the process starts from the mutant compilation step
  - presemu: the process starts from the preparation of the meta mutant step (i.e., Pre-SEMu)
  - semu: the process starts from the test generation step itself
  - unittests: the process starts from the conversion of KLEE tests to unit tests step
- mutants-list-file: is the file containing the list of mutants to use during the phases pre-semu and semu.
- output-dir-for-pre-semu-and-semu: directory to store the output of pre-semu and semu phases, when the mutants list is specified.

An example for launching the test generation from the mutation generation step follows:

```
$ ENV_FAQAS_SEMU_SRC_FILE=test.c scripts/docker_run.sh mutation WORKSPACE/DOWNLOADED/live_mutants WORKSPACE/OUTPUT/live_mutants_output
```

8.2.4 *SEMuS* results

We provide a script that summarizes *SEMuS* results, the command for launching this script follows:

```
$ ./generateReport.sh
```

This script generates a `AnalysisReport.csv` file located in the path `case_studies/$SUT/WORKSPACE/OUTPUT`; the report indicates the total number of analyzed mutants, the number of killed and live mutants, and the list of mutant names, including the status of the mutant (i.e., KILLED/LIVE).

However, more detailed results, including intermediate files, can be found at the folder `case_studies/$SUT/WORKSPACE/OUTPUT`:

- `mutants_generation`: this folder contains the mutant sources generated by *MASS*
- `live_mutants_output/mutants_generation`: this folder stores the source files and the compiled objects of the meta mutant files (e.g., `*.MetaMu.c` and `*.MetaMu.bc`)
- `live_mutants_output/test_generation`: this folder contains the outputs of *SEMuS* concerning the test generation step, this directory contains one folder for each test template. Furthermore, it also contains the following folders:

- `direct/TEMPLATE/FAQAS_SEMu-out/semu`: *SEMu* output (e.g., KLEE tests files, execution traces)
- `direct/TEMPLATE/FAQAS_SEMu-out/produced-unittests`: Unit test cases converted from *SEMu* output

8.3 Normal Termination

Each step of *SEMuS* is executed when invoked and a result is generated. There is no software interruption foreseen during the computation and the procedure terminates by returning the result. If the engineer decides to interrupt *SEMuS* execution, it can be done by sending a signal interrupt `SIGINT` to the running process.

8.4 Error Conditions

There is no error condition handling in the FAQAS-framework. If all preconditions are met, there should not be any error.

8.5 Recover Runs

If for any reason the execution of *SEMuS* is interrupted, an engineer can restart the process from a specific step if all preconditions are met. This is possible since each *SEMuS* step work by processing data that is permanently stored by previous steps.

Chapter 9

DAMAt - Operations Manual

9.1 Set-up and Initialization

9.1.1 Dependencies

- `Python 3.6.8` or higher
- `GNU bash, version 4.2.46` or higher

9.1.2 Initialization of the *DAMAt* workspace

All the scripts for the *DAMAt* pipeline are contained in the `DAMAt-pipeline` folder. All *DAMAt* steps will take place inside this folder, which shall be placed by the engineer in a path of their choosing.

The folder shall contain the following structure and files:

- `DAMAt_configure.sh`: this script defines the necessary variables for the execution of *DAMAt*. They shall be set by the engineer.
- `DAMAt_probe_generation.sh`: this script sets the variables that are necessary to generate the data mutation API and execute the python script `generateDataMutator.py` to generate them.
- `DAMAt_mutants_launcher.sh`: this script starts the *DAMAt* pipeline.
- `generateDataMutator.py`: this is the script that generates the *DAMAt* mutation API.
- `DDB_TEMPLATE_header.c` and `DDB_TEMPLATE_footer.c`: these are templates used to generate the *DAMAt* API by `generateDataMutator.py`
- `DAMAt_compile.sh`: this is a stub of the script used to compile a mutant, which shall be completed by the engineer.
- `DAMAt_run_tests.sh`: this is a stub of the script used to run the tests, which shall be completed by the engineer.
- `data_analysis`: a folder containing five python scripts used for the generation of the final results:

  - `beautify_results.py`: this script renders the raw results from the execution of the tests in a more readable format.
  - `get_coverage.py`: this script analyzes the results of the fault model coverage.
  - `get_operator_coverage.py`: this script analyzes the results of the operator coverage.
  - `get_stats.py`: this script produces statistics from the mutants' execution.
  - `get_final_results.py`: this script produces a summary of the execution of *DAMAt*.
- `pipeline_scripts`: a folder containing the four scripts that make up the *DAMAt* pipeline:
  - `DAMAt_obtain_coverage.sh`: this script obtains fault model coverage data in order to execute only the tests that cover each mutant.
  - `get_mutant_test_list.py`: this script produces the list of test against which every mutant shall be executed.
  - `DAMAt_compile_and_run_mutants.sh`: this scripts compile each mutant and run it against the SUT test suite.
  - `DAMAt_data_analysis.sh`: this script executes all the data analysis steps at the end of the execution of the *DAMAt* pipeline
- `fault_model.csv`: an example of a *DAMAt* fault model in csv format.
- `tests.csv` : an example of list of test cases and nominal times in csv format.
- `automated_probe_insertion`: this folder contains the necessary scripts for automating the probe insertion.
  - `DAMAt_probe_insertion.sh`: this script backups the file to instrument and then executes `DAMAt_insert_probes.py`.
  - `DAMAt_insert_probes.py`: this script replaces specific comments in the file to instrument with the mutation probes.
  - `DAMAt_probe_removal.sh`: this script restores the original file from the template.
  - `DAMAt_insertion_test.sh`: this script runs the test cases for the automated probe insertion procedure.
  - `test_files`: this folder contains the necessary files for executing the test cases.

9.1.3 Writing a list of all test cases

The user shall provide the list of all the test cases with corresponding nominal time in csv format, using the `tests.csv` file as an example. In the first column of the csv file, there shall be an identifier for the test case. The second column shall contain the nominal execution time in ms.

```
1 test_01,11002
2 test_02,13456
3 test_03,58347
```

This file will be used by the *DAMAt* pipeline for two purposes:

1. the nominal execution time will be used to set a timeout for each test case.
2. the whole list will be used to generate a list for each mutant containing only the test cases that cover it.

9.1.4 Setting variables for the *DAMAt* pipeline

The user must set the following variables inside the `DAMAt_configure.sh` script

```

tests_list=$DAMAt_FOLDER/tests.csv

fault_model=$DAMAt_FOLDER/fault_model.csv

buffer_type="unsigned char"

singleton="TRUE"

padding=0
```

The variable `test_list` shall be set to the path of the csv containing the test names and execution times.

The variable `fault_model` shall be set to the path of the csv containing the fault model of the SUT that the engineer shall define for the SUT.

The variable `buffer_type` shall contain the type of the elements of the buffer that will be targeted by the mutation.

The variable `singleton` shall be set to `TRUE` or `FALSE`. If set to `TRUE`, the Fault Model will be initialized in a singleton variable, to avoid memory issues.

The variable `padding` shall be set to an integer number representing the number of bytes to skip at the beginning of the target buffer. Normally it shall be set to `0`, but it can be used to skip the header of the buffer if needed.

9.1.5 Setting up the compilation of the mutants

The user shall modify `DAMAt_compile.sh` to include the commands for the compilation of the mutants. Every mutant is identified by an integer called "MutationOpt". To enable the data-driven mutation, the engineer shall compile the SUT with this macro enabled:

`-DMUTATIONOPT=<MutationOpt>`.

To use the singleton mode, the engineer shall compile the SUT with this macro enabled:

`-D_FAQAS_SINGLETON_FM TRUE"`.

A way to do it is to include the following lines in the SUT makefile:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -DMUTATIONOPT=$ENV{MUTATIONOPT}")
# comment the following line if you do not want to use the singleton option.
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -D_FAQAS_SINGLETON_FM=$ENV{_FAQAS_SINGLETON_FM}")
```

and then export the corresponding variables in the `DAMAt_compile.sh` script as shown in the `DAMAt_compile.sh` stub.

The contents of the `DAMAt_compile.sh` stub are portrayed in the following.

```

#!/bin/bash

mutant_id=$1
singleton=$2
```

```
6
7  ##############################################################################
8  #enabling extended pattern matching features:
9  shopt -s
10 #options for enabling aliases:
11 shopt -s expand_aliases
12 ##############################################################################
13
14
15 echo "-----------------------------------"
16 echo "-----------------------------------"
17 echo "Mutant opt: "$mutant_id
18 echo "-----------------------------------"
19 echo "-----------------------------------"
20
21 # The user shall complete the following section:
22 ##############################################################################
23 #exporting the operation counter
24 export MUTATIONOPT=$mutant_id
25
26 if [$singleton == "TRUE"]; then
27 export _FAQAS_SINGLETON_FM=$singleton
28 fi
29
30 # here the engineer must invoke the compilation of the SUT, we provided a simple example.
31
32 compilation_folder="/home/SUT"
33
34 pushd $compilation_folder
35
36 make install-debug
37
38     if [ $? -eq 0 ]; then
39         echo $x " compilation OK"
40     else
41         echo $x " compilation FAILED"
42     fi
43
44 popd
```

Once completed, the `DAMAt_compile.sh` script shall take as input the value of the MutationOpt that refers to the mutant being currently compiled and of the singleton variable, and then compile the SUT accordingly.

9.1.6 Setting up the execution of the test suite against the mutants

The user shall modify `DAMAt_run_tests.sh` stub to include the commands for the execution of the mutants.

Every mutant is identified by an integer called "MutationOpt". This script shall take as input the mutationOpt and a list of tests in the format described in section 9.1.3.

The user shall complete the stub by substituting the generic execution function with a command or a series of commands that execute a specific test case using the test identifier specified in the csv as input. The user shall use the `timeout` command as shown in the example included in the stub to set a timeout.

The output of the the new function shall be `0` if the test passes, different from `0` if it fails and `124` in case of timeout.

The contents of the `DAMAt_run_tests.sh` stub are portrayed in the following.

```
1
2  #!/bin/bash
```

```
3
4  mutant_id=$1
5  tests_list=$2
6  DAMAt_FOLDER=$3
7  results_dir=$DAMAt_FOLDER/results
8
9  mutant_dir=$results_dir/run_"$mutant_id"
10 execution_log=$mutant_dir/"$mutant_id"_execution.out
11 coverage_file=$mutant_dir/"$mutant_id"_coverage.csv
12 results_file=$mutant_dir/main.csv
13
14 ################################################################################
15 #date in milliseconds
16 start_time=$(($(date +%s%N)/1000000))
17
18 mkdir $results_dir
19 echo "this is the results folder: $results_dir"
20 mkdir $mutant_dir
21 echo "this is the folder of this mutant $mutant_dir"
22 touch $execution_log
23 echo "this is the logfile of the tests' execution $execution_log"
24 touch $coverage_file
25 echo "this is the file with the results $results_file"
26 touch $results_file
27 echo "this is the coverage file: $coverage_file"
28 export FAQAS_COVERAGE_FILE=$coverage_file
29 export _FAQAS_SINGLETON_FM
30
31 ################################################################################
32
33 while IFS="," read -r p || [ -n "$p" ];do
34
35     mutant_start_time=$(($(date +%s%N)/1000000))
36
37     # obtaining test number to be executed
38     read tst <<< $(awk -F',' '{print $1}' <<< "$p")
39
40     # obtaining corresponding timeout for the test case
41     TIMEOUT=$(echo "$p" | awk -F',' '{$2=($2*4)/1000; printf("%.0f\n", $2);}')
42
43     echo "************************************************************"
44     echo "************************************************************"
45     echo "Running mutant $mutant_id against test case "$tst
46     echo "************************************************************"
47     echo "************************************************************"
48     echo -n "${mutant_id};COMPILED;${tst};" >> $results_file
49
50
51 ################################################################################
52 # here the engineer shall call the execution of the current test case,
53 # we provided a simple example
54
55   timeout $TIMEOUT bash execute_test_case.sh $tst
56
57
58 ################################################################################
59 #the exec return code should be 0 if the test case passes, 1 if the test case fails, and 124 in case of
       a timeout
60 EXEC_RET_CODE=$?
61
62
63   mutant_end_time=$(($(date +%s%N)/1000000))
64   mutant_elapsed="$(($mutant_end_time-$mutant_start_time))"
65
66
67
68   if [ $EXEC_RET_CODE -ge 124 ]; then
69      echo "Test return code: [$EXEC_RET_CODE]"
70      echo "Mutant timeout by $tst"
71      echo -ne "TIMEOUT;KILLED_${EXEC_RET_CODE};${mutant_elapsed}\n" >> $results_file
72
73   else
74       if [ $EXEC_RET_CODE -eq 0 ]; then
75         echo "Test return code: [$EXEC_RET_CODE]"
76         echo -ne "PASSED;LIVE;${mutant_elapsed}\n" >> $results_file
```

```
77
78        else
79          echo "Test return code: [$EXEC_RET_CODE]"
80          echo "Mutant killed by $tst"
81          echo -ne "FAILED;KILLED;${mutant_elapsed}\n" >> $results_file
82        fi
83    fi
84
85
86  ##############################################################################
87
88    #create a different coverage file for every test
89
90    NEW_COVERAGE_FILE=$mutant_dir/coverage_"$tst".csv
91    cp $coverage_file $NEW_COVERAGE_FILE
92    >$FAQAS_COVERAGE_FILE
93
94  done < $tests_list
95
96  rm -rf ~/Obsw/Test/System/testresults/*
97
98  end_time=$(($(date +%s%N)/1000000))
99  elapsed="$(($end_time-$start_time))"
100
101 echo "elapsed time $elapsed [ms]"
```

9.2 Executing *DAMAt*

9.2.1 Running *DAMAt* on a Single Machine

DAMAt works in six steps:

1. The user prepares a fault model specification tailored to the SUT.
2. *DAMAt* generates a mutation API with the functions that modify the data according to the provided fault model.
3. The user manually modifies the SUT by introducing mutation probes (i.e., invocations to the mutation API) into its source code.
4. *DAMAt* generates and compiles mutants.
5. *DAMAt* executes the test suite against all the mutants.
6. *DAMAt* generates mutation analysis results.

9.2.1.1 Step 1

The engineer is expected to write a fault model in the csv format with the definition of all the mutation operators they want to apply. An example of fault model is contained in the `DAMAt` folder and in listing 9.2.1.1.

```
1 FaultModel,DataItem,Span,Type,FaultClass,Min,Max,Threshold,Delta,State,Value
2 fault_model_01,0,1,BIN,BF,3,3,NA,NA,-1,1
3 fault_model_01,0,1,BIN,BF,4,4,NA,NA,-1,1
4 fault_model_01,0,1,BIN,BF,5,7,NA,NA,-1,1
5 fault_model_02,12,2,DOUBLE,VAT,NA,NA,3.6,0.1,NA,NA
6 fault_model_02,12,2,DOUBLE,FVAT,NA,NA,3.6,0.1,NA,NA
7 fault_model_02,14,2,DOUBLE,VAT,NA,NA,33.53,0.01,NA,NA
```

```
fault_model_02,14,2,DOUBLE,FVAT,NA,NA,33.53,0.01,NA,NA
fault_model_02,14,2,DOUBLE,VBT,NA,NA,24,1,NA,NA
fault_model_02,14,2,DOUBLE,FVBT,NA,NA,24,1,NA,NA
fault_model_03,0,1,HEX,IV,NA,NA,NA,NA,NA,0x51
fault_model_03,0,1,HEX,IV,NA,NA,NA,NA,NA,0x52
fault_model_03,0,1,HEX,IV,NA,NA,NA,NA,NA,0x53
fault_model_03,0,1,HEX,IV,NA,NA,NA,NA,NA,0x54
fault_model_03,0,1,HEX,IV,NA,NA,NA,NA,NA,0x56
```

For details on the available mutation operators and how they can be configured see the D2 document.

9.2.1.2 Step 2

In this step, *DAMAt* generates a mutation API with the functions that modify the data according to the provided fault model.

The engineer shall run the following command in the terminal inside the `DAMAt_pipeline` folder:

```
bash DAMAt_probe_generation.sh
```

This procedure will produce three files:

1. `FAQAS_dataDrivenMutator.h`: the mutation API.
2. `FAQAS_mutants_table.csv`: a csv table with the mutationOpt and definition of all the generated mutants.
3. `function_calls.out` function templates for the mutation probes to insert in the SUT.

a copy of the first two files must remain in the DAMAt folder for the correct execution of the data analysis section of the pipeline.

9.2.1.3 Step 3

The engineer will manually instrument the SUT by:

1. copying the `FAQAS_dataDrivenMutator.h` file in the same folder as the file containing the target function that they chose to instrument.
2. including the mutation API inthe chose file by adding

```
  #include "FAQAS_dataDrivenMutator.h"

```

3. inserting function calls in the chosen function based on the prototypes contained in `function_calls.out`.

An example of a probe is included below, where `v` is the vector representing the buffer:

```
mutate_FM_fault_model_01( &v );
```

9.2.1.4 Step 4, Step 5 and Step 6

The other steps of the *DAMAt* procedure are carried out automatically by the pipeline. The pipeline can be started by running the `DAMAt_mutants_launcher.sh` script with the following command, executed inside the `DAMAt_pipeline` folder:

```
bash DAMAt_mutants_launcher.sh
```

Before running all the generated mutants, a special mutant (MutationOpt=-2) will be executed to gather coverage information. All the subsequent mutants will only be executed against tests that cover them to save execution time. The lists of test cases executed against every mutant can be found in the folder `.../DAMAt/testlists`, which will be automatically generated.

These steps can also be executed separately by manually exporting the relevant variables and then running the pipeline scripts one by one. An example of the necessary commands can be found in the following.

```

# variable to export
export DAMAt_FOLDER=$(pwd)
export tests_list=$DAMAt_FOLDER/tests.csv
export fault_model=$DAMAt_FOLDER/fault_model.csv
export buffer_type="unsigned char"
export padding=2
export singleton="TRUE"
export PIPELINE_FOLDER=$DAMAt_FOLDER/pipeline_scripts
export RESULTS_FOLDER=$DAMAt_FOLDER/results
export TESTS_FOLDER=$DAMAt_FOLDER/testlistsRESULTS_FOLDER=$DAMAt_FOLDER/results
export PIPELINE_FOLDER=$DAMAt_FOLDER/pipeline_scripts
export RESULTS_FOLDER=$DAMAt_FOLDER/results
export TESTS_FOLDER=$DAMAt_FOLDER/testlistsRESULTS_FOLDER=$DAMAt_FOLDER/results

# step 4 and step 5
bash $PIPELINE_FOLDER/DAMAt_obtain_coverage.sh $tests_list $DAMAt_FOLDER $singleton
bash $PIPELINE_FOLDER/DAMAt_compile_and_run_mutants.sh $DAMAt_FOLDER $singleton
# step 6
bash $PIPELINE_FOLDER/DAMAt_data_analysis.sh $DAMAt_FOLDER $tests_list
```

9.2.2 *DAMAt* results

After the execution of *DAMAt*, the results are stored in the `.../DAMAt/results` folder that will be automatically generated. For every mutant the pipeline will create a subfolder called `run_<mutationOpt>` containing:

- A `main.csv` file with the results of the mutant's execution against the test suite
- A `coverage_<test case>.csv` file containing the raw operator coverage data for that mutant-test case couple.
- A `readable_coverage_<test case>.csv` file containing the readable operator coverage data for that mutant-test case couple.
- A `<mutationOpt>_exectuion.out` file containing an execution log for the mutant.

In addition to that, a `logs` folder shall be created, containing the full compilation and execution logs for every mutant.

The `.../DAMAt/results` folder contains also files relative to the metrics defined to characterize the results of the full execution of *DAMAt*:

1. Fault model coverage is the percentage of fault models covered by the test suite.
2. Mutation operation coverage 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.
3. 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.

These metrics measure the frequency of the following scenarios:

1. the message type targeted by a mutant is never exercised.
2. the message type is covered by the test suite but it is not possible to perform some of the mutation operations.
3. the mutation is performed but the test suite does not fail.

The file generated by the final steps are:

- `mutation_sum_up.csv`: a file containing the previously described three metrics.
- `final_mutants_table.csv`: a file containing the definition and status of every mutant.
- `mutation_score_by_data_item.csv`: a file containing the mutation score by data item.
- `mutation_score_by_fault_class.csv`: a file containing the mutation score by fault class.
- `mutation_score_by_fault_model.csv`: a file containing the mutation score by fault model.
- `test_coverage.csv`: a file containing the tests covering the different fault models.
- `readable_data.csv`: a file containing a more readable version of the execution data.
- `raw_data.csv` and `raw_data_sorted.csv`: these files contain all the execution data.
- `readable_operator_coverage.csv`: a file containing a more readable version of the operator coverage data.
- `operator_coverage.csv`: a file containing a raw version of the operator coverage data.
- `readable_FM_coverage.csv`: a file containing a more readable version of the fault model coverage data.
- `FM_coverage.csv`: a file containing a raw version of the fault model coverage data.

9.3 Automated probe insertion

The `automated_probe_insertion` folder contains some additional scripts to facilitate the insertion and removal of the mutation probes from the instrumented file. These scripts can be run before and after the execution of the *DAMAt* pipeline to instrument the source code and then restore it to its original state. This can be useful, among other things, to integrate *DAMAt* into a continuous integration/continuous delivery pipeline.

The contents of the folder are the following:

- `DAMAt_probe_insertion.sh`: this script backups the file to instrument and then executes `DAMAt_insert_probes.py`.
- `DAMAt_insert_probes.py`: this script replaces specific comments in the file to instrument with the mutation probes.
- `DAMAt_probe_removal.sh`: this script restores the original file from the template.
- `DAMAt_insertion_test.sh`: this script runs the test cases for the automated probe insertion procedure.
- `test_files`: this folder contains the necessary files for executing the test cases.

9.3.1 Probe comments

In place of the mutation probes, the user shall add *probe comments* to the source code of the file to instrument. The *probe comments* are regular *C/C++* comments, so they do not influence the normal compilation and functioning of the source code and can be kept in the code during every step of a CD/CI pipeline. Comments following a specific structure will be automatically replaced with the correct mutation probes and the header containing the mutation API will be included, while the rest of the file will not be modified in any way.

An example of the structure of the *probe comments* is reported in Listing 9.3.1.

Every *probe comment* must contain, between brackets and separated by a comma, the name of the fault model that the mutation probe will implement, and the name of the buffer to mutate. In Listing 9.3.1 they are, respectively, `Fm_1` and `v_1`.

```

  printf("Hello, World!");

  [ ... ]

// mutation_probe(Fm_1, v_1)
```

9.3.2 Commands

The user needs to execute the `DAMAt_probe_insertion.sh` script before executing the *DAMAt* pipeline, and the `DAMAt_probe_removal.sh` afterwards.

The structure of the necessary commands for this procedure is reported in Listing 9.3.2. The scripts need only four parameters:

1. `<target_folder>`: the folder of the file to instrument.
2. `<target_file>`: the name of the file.
3. `<template>`: see Section 9.3.3
4. `<path>`: the path to the *DAMAt* mutation API. It will be used in the include statement.

```

# to intrument the source code
bash DAMAt_probe_insertion.sh "<target_folder>"  "<target_file>" "<template>" "<path>"

# to restore the source code
bash DAMAt_probe_removal.sh "<target_folder>"  "<target_file>"
```

9.3.3 Probe Templates

The `<template>` parameter determines the structure of the mutation probe. In most cases, the insertion of a mutation probe does not require additional lines of code except for the probe itself. In those cases the user must set the `<template>` parameter to *standard*. The scripts will replace the *probe comments* with the correct calls to the mutation API, following this structure: `mutate_FM_$fault_model($buffer);`.

The string `$fault_model` will be automatically replaced with the fault model indicated in the probe comment, and the same will happen to `$buffer`.

In some particular cases, however, there could be the need for additional code, for example, to convert a `struct` into an array to be fed to the mutation probe. In those cases, the user shall write a *probe template* and must set the path to that template as the `<template>` parameter.

A *probe template* is a simple text file containing the call to the mutation API and any additional line of code.

As for the previous case, the strings `$fault_model` and `$buffer` will be automatically replaced with the ones indicated in the probe comment.

An example of *probe template* is provided in Listing 9.3.3.

```

//start of the  mutation probe

unsigned long long int $buffer[6];

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

mutate_FM_$fault_model($buffer );

target_data_structure->action = (uint8_t) $buffer[0];
target_data_structure->table_id = (uint8_t) $buffer[1];
target_data_structure->length = (uint16_t) $buffer[2];
target_data_structure->checksum = (uint16_t) $buffer[3];
target_data_structure->seq = (uint16_t) $buffer[4];
target_data_structure->total = (uint16_t) $buffer[5];

//end of the mutation probe
```

9.4 Normal Termination

If the engineer decides to interrupt *DAMAt* execution, it can be done by sending a signal interrupt `SIGINT` to the running process.

9.5 Error Conditions

There is no error condition handling in the FAQAS-framework. If all preconditions are met, there should not be any error.

9.6 Recover Runs

If for any reason the execution of *DAMAt* is interrupted, an engineer can restart the process from a specific task if all preconditions are met, manually executing the rest of the steps.

Chapter 10

DAMTE - Operations Manual

10.1 Set-up and Initialization

10.1.1 Dependencies

- `Python 3.6.8` or higher
- `GNU bash, version 4.2.46` or higher
- KLEE 2.3
- LLVM 9.0.1

10.2 Executing DAMTE

To execute DAMTE, first the engineer should generate the data-driven mutation testing FAQAS API (i.e., FAQAS_dataDrivenMutator.h) through the following command:

```
$ python DAMTE/mutator/src/generateDataMutator.py <BufferType> <FaultModel.csv> <TestAssessment>
```

The procedure is similar to the one specified for *DAMAt*, the only difference is that the engineer should specify the argument `<TestAssessment>` which for mutation testing shall be 0.

The probes to be inserted shall be contained in the generated `FAQAS_dataDrivenMutator.h`. Note that the environment variable `TEST_ASSESSMENT` has to be set to 0 (i.e., `TEST_ASSESSMENT=0`) to enable test generation.

The next step consists of preparing a test template for guiding the test generation with KLEE. An example of test template follows in Listing 10.1:

```
    // a little hack - this is next element, we use it check for overwrite and missing 0 termination
    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;

```

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

Listing 10.1: Test template to enable data-driven mutation testing

Note that the arguments of the function, for which test generation is targeted, shall be declared symbolic with KLEE API `klee_make_symbolic`.

It is important to include the `FAQAS_dataDrivenMutator.h` with an include statement in the same source as the test template defined above.

Then, the test template plus the mutated functions shall be compiled with the following command:

```
1  $ clang -I klee/include -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone source.c
```

This command will generate a bitcode file that should be passed to KLEE with the following command:

```
1  $ klee --libc=uclibc --posix-runtime --external-calls=all
```

If test cases are generated, the results can be then checked with `ktest-tool` command:

```
1  $ ktest-tool klee-last/test000001.ktest
```

The output of `ktest-tool` command will represent the results in binary format.

Chapter 11

Reference Manual

11.1 Code-driven mutation analysis toolset (MASS)

MASS supports the following commands:

- `PrepareSUT`: command to prepare the SUT and collect information about the SUT test suite.
- `GenerateMutants`: command to generate mutants from the SUT source code.
- `CompileOptimizedMutants`: command to compile the mutants with the multiple optimisation levels.
- `OptimizedPostProcessing`: command to disregard equivalent and redundant mutants based on compiler optimisations.
- `GeneratePTS`: command to generate the prioritized and reduced test suites.
- `ExecuteMutants`: command to execute mutants against the SUT test suite.
- `IdentifyEquivalents`: command to identify equivalent mutants based on code coverage.
- `MutationScore`: command to compute the mutation score and final reporting.
- `PrepareMutants_HPC`: command to prepare the mutants workspace for the execution on HPCs.
- `ExecuteMutants_HPC`: command to execute mutants on HPCs.
- `PostMutation_HPC`: command to assess past mutant executions, and to decide whether more mutant executions are needed.

For more information about how to operate each command, please refer to Section 7.2.3.2.

11.2 Code-driven test generation toolset (*SEMuS*)

SEMuS supports the following commands:

- `call_generate_direct.sh`: command to generate the test templates for guiding the test generation process.
- `docker_run.sh` or `run.sh`: command to launch the test generation process.
- `generateReport.sh`: command to generate the final *SEMuS* report.

For more information about how to operate each command, please refer to Section 8.2.3.

11.3 Data-driven mutation analysis toolset (*DAMAt*)

- `DAMAt_probe_generation.sh`: command to generate the data mutation probes.
- `DAMAt_mutants_launcher.sh`: command to execute the mutants against the SUT test suite.

For more information about how to operate each command, please refer to Section 9.2.1.

11.4 Data-driven test generation toolset (DAMTE)

Chapter 12

MASS - Tutorial

12.1 Introduction

This tutorial instructs on how to use *MASS* on a typical case. Since *MASS* provides two modes of execution (i.e., running *MASS* on a single machine and running *MASS* on shared resources facilities), we provide an example for both.

Both examples use the Mathematical Library for Flight Software as case study to exemplify the use of FAQAS-framework.

The Mathematical Library for Flight Software (MLFS) implements mathematical functions ready for qualification[1]. MLFS is born from the need of having a mathematical library ready for qualification for flight software. Well known mathematical libraries such as `libm` and `newlib` are not completely validated with respect to specific input ranges, errors and performance, and so, they do not comply with ECSS criticality category B. The set of functions provided by MLFS are limited to the functions typically needed in flight software. A13

We provide the source code of the Mathematical Library for Flight Software version 1.2 together with the MASS framework. The source code can be found at the location `MASS/examples/mlfs`.

12.2 Running MASS on a Single Machine

12.2.1 Mathematical Library for Flight Software Example

The first step regards installing the *MASS* framework, please refer to Section 7.1.

The second step consists of creating and installing a workspace folder for running *MASS* on the MLFS example. For this case, the workspace folder will be created on `/opt/MLFS`. Note that variable `$FAQAS` represents the installation folder of the FAQAS-framework.

```
$ cd $FAQAS/MASS/FAQAS-Setup
$ export INSTALL_DIR=/opt/MLFS
$ ./install.sh
```

[1] `https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software`

The third step consists of configuring the *MASS* configuration file `mass_conf.sh`. In the following, we provide all the excerpts that require manual editing. Listing 12.1 contains the necessary configuration for the MLFS case study.

```
# set FAQAS path
export SRCIROR=/opt/srcirorfaqas

...

# set directory path where MASS files can be stored
export APP_RUN_DIR=/opt/MLFS

# specifies the building system, available options are "Makefile" and "waf"
export BUILD_SYSTEM="Makefile"

# directory root path of the software under test
export PROJ=$HOME/mlfs

# directory src path of the SUT
export PROJ_SRC=$PROJ/libm

# directory test path of the SUT
export PROJ_TST=$HOME/unit-test-suite

# directory coverage path of the SUT
export PROJ_COV=$HOME/blts_workspace

# directory path of the compiled binary
export PROJ_BUILD=$PROJ/build-host/bin

# filename of the compiled file/library
export COMPILED=libmlfs.a

# path to original Makefile
export ORIGINAL_MAKEFILE=$PROJ/Makefile

# compilation command of the SUT
export COMPILATION_CMD=(make all ARCH=host EXTRA_CFLAGS="-DNDEBUG" \&\& make all COVERAGE="true" ARCH=
    host_cov EXTRA_CFLAGS="-DNDEBUG")

# compilation additional commands of the SUT (e.g., setup of workspace)
export ADDITIONAL_CMD=(cd $HOME/blts/BLTSConfig \&\& make clean install INSTALL_PATH="$HOME/
    blts_install" \&\& cd $HOME/blts_workspace \&\& $HOME/blts_install/bin/blts_app --init)

# command to be executed after each test case (optional)
export ADDITIONAL_CMD_AFTER=(rm -rf $HOME/blts_workspace/*)

# compilation command for TCE analysis
export TCE_COMPILE_CMD=(make all ARCH=host EXTRA_CFLAGS="-DNDEBUG")

# command to clean installation of the SUT
export CLEAN_CMD=(make cleanall)

# relative path to location of gcov files (i.e., gcda and gcno files)
export GC_FILES_RELATIVE_PATH=Reports/Coverage/Data
```

Listing 12.1: *MASS* variables. Excerpt of mass_conf.sh file.

Also *MASS* variables shall be configured within the same file. Particularly, we will run *MASS* with the setup contained in Listing 12.2.

```
### MASS variables

# TCE flags to be tested
export FLAGS=("-O0" "-O1" "-O2" "-O3" "-Ofast" "-Os")

# specify if MASS will be executed on a HPC, possible values are "true" or "false"
export HPC="false"

# set if MASS should be executed with a prioritized and reduced test suite
export PRIORITIZED="true"

# set sampling technique, possible values are "uniform", "stratified", "fsci", and "no"
```

```
# note: if "uniform" or "stratified" is set, $PRIORITIZED must be "false"
export SAMPLING="fsci"

# set sampling rate if whether "uniform" or "stratified" sampling has been selected
export RATE=""
```

Listing 12.2: *MASS* specific variables. Excerpt of mass_conf.sh file.

The fourth step consists of configuring the `PrepareSUT` configuration file (`MASS_STEPS_LAUNCHERS/PrepareSUT.sh`, the actions provided in this file enables *MASS* to collect the code coverage; within this file the following actions must be performed by the engineer (see 12.3), notice that we use the tool `bear` for the generation of the `compile_commands.json`:

```
#!/bin/bash

cd /opt/MLFS
. ./mass_conf.sh

# 1. Compile SUT

cd $PROJ

# generate compile_commands.json and delete build
bear make all && rm -rf build* && sed -i 's: libm: /home/mlfs/mlfs/libm:' compile_commands.json && mv
    compile_commands.json $MUTANTS_DIR
eval "${COMPILATION_CMD[@]}"

# 2. Prepare test scripts
# example

cd $HOME/blts/BLTSConfig
make clean install INSTALL_PATH="$HOME/blts_install"

# Preparing MLFS workspace (e.g., where test cases data is stored)
cd $HOME/blts_workspace
$HOME/blts_install/bin/blts_app --init

# 3. Execute test cases
# Note: execution time for each test case should be measured and passed as argument to FAQAS-
    CollectCodeCoverage.sh

# example
for tst in $(find $HOME/unit-test-suite -name '*.xml');do
    cd $HOME/blts_workspace

    tst_filename_wo_xml=$(basename -- $tst .xml)

    start=$(date +%s)
    $HOME/blts_install/bin/blts_app -gcrx $tst_filename_wo_xml -b coverage --nocsv -s $tst
    end=$(date +%s)

    # call to FAQAS-CollectCodeCoverage.sh
    # parameter should be test case name and the execution time
    source $MASS/FAQAS-GenerateCodeCoverageMatrixes/FAQAS-CollectCodeCoverage.sh $tst_filename_wo_xml "
    $(($end-$start))"
done
```

Listing 12.3: PrepareSUT.sh file.

The fifth step consists of defining the function `run_tst_case` within the file `mutation_additional_functions.sh`. An example of its implementation is provided in Listing 12.4.

```
run_tst_case() {

    tst_name=$1
    tst=$PROJ_TST/$tst_name.xml

    echo $tst_name $tst

    # run the test case
```

```
    cd $PROJ_COV
    $HOME/blts_install/bin/blts_app -gcrx $tst_name -b coverage --nocsv -s $tst

    # define if test case execution passed or failed
    summaryreport=$tst_name/Reports/SessionSummaryReport.xml
    originalreport=$HOME/unit-reports/$summaryreport

    test_cases_failed=`xmllint --xpath "//report_summary/test_set_summary/test_cases_failed/text()"
     $summaryreport`
    o_test_cases_failed=`xmllint --xpath "//report_summary/test_set_summary/test_cases_failed/text()"
     $originalreport`

    echo "comparing with original execution"
    echo $test_cases_failed $o_test_cases_failed

    if [ "$test_cases_failed" != "$o_test_cases_failed" ]; then
        return 1
    else
        return 0
    fi
}
```

Listing 12.4: 'run_tst_case' Bash function for the MLFS. Excerpt of mutation_additional_functions.sh file.

The sixth step consists of providing a template for the build script for the trivial compiler optimizations step. In particular, we replaced the optimization flag in the original build script:

```
CFLAGS = -c -Wall -std=gnu99 -pedantic -Wextra -frounding-math -fsignaling-nans -g O2 -fno-builtin $(
    EXTRA_CFLAGS)
```

Listing 12.5: Excerpt from Makefile.

with TCE, creating a new template for the build script:

```
CFLAGS = -c -Wall -std=gnu99 -pedantic -Wextra -frounding-math -fsignaling-nans TCE -fno-builtin $(
    EXTRA_CFLAGS)
```

Listing 12.6: Excerpt from Makefile.template.

The seventh step consists of launching the one step launcher (see Section 7.2.3):

```
 $ /opt/MLFS/Launcher.sh
```

The following results shall be reported at the end of the execution:

```
##### MASS Output #####
## Total mutants generated: 28071
## Total mutants filtered by TCE: 6918
## Sampling type: fsci
## Total mutants analyzed: 461
## Total killed mutants: 369
## Total live mutants: 92
## Total likely equivalent mutants: 53
## MASS mutation score (%): 90.44
## List A of useful undetected mutants: /opt/MLFS/RESULTS/useful_list_a
## List B of useful undetected mutants: /opt/MLFS/RESULTS/useful_list_b
## Number of statements covered: 1973
## Statement coverage (%): 100
## Minimum lines covered per source file: 2
## Maximum lines covered per source file: 138
```

Listing 12.7: *MASS* output.

12.3 Running MASS on HPC Infrastructure

12.3.1 Mathematical Library for Flight Software Example

This tutorial was implemented by executing *MASS* on the UL HPC[2] infrastructure. The examples shown in the following uses the SLURM job scheduler[3] and the GNU parallel utility[4]. Both software are not mandatory for executing *MASS*, and can be replaced by similar software if executed in a different environment.

MASS has been executed on a Singularity container[5], for enabling reproducibility and parallelism for experiments.

To run *MASS* on a HPC infrastructure, the first step regards installing the *MASS* framework, to do so please refer to Section 7.1.

The second step, consists of creating and installing a workspace folder to run *MASS* on the MLFS example. In this case, the workspace directory will be created on `$FAQAS/MASS_MLFS/MASS_WORKSPACE`, and the execution directory on `/opt/MLFS`, since `$FAQAS/MASS_MLFS/MASS_WORKSPACE` will be binded to `/opt/MLFS` inside the Singularity container. Note that variable `$FAQAS` represents the installation folder of the FAQAS-framework.

```
$ cd $FAQAS/MASS/FAQAS-Setup
$ export INSTALL_DIR=$FAQAS/MASS_MLFS/MASS_WORKSPACE && export EXECUTION_DIR=/opt/MLFS
$ ./install.sh
```

The third step consists of configuring the *MASS* configuration file `mass_conf.sh`. In the following, we provide all the excerpts that require manual editing. Listing 12.8 contains the necessary configuration for the MLFS case study. Notice that all paths defined here must refer to the container.

```
# set FAQAS path
export SRCIROR=/opt/srcirorfaqas

...

# set directory path where MASS files can be stored
export APP_RUN_DIR=/opt/MLFS

# specifies the building system, available options are "Makefile" and "waf"
export BUILD_SYSTEM="Makefile"

# directory root path of the software under test
export PROJ=$HOME/mlfs

# directory src path of the SUT
export PROJ_SRC=$PROJ/libm

# directory test path of the SUT
export PROJ_TST=$HOME/unit-test-suite

# directory coverage path of the SUT
export PROJ_COV=$HOME/blts_workspace

# directory path of the compiled binary
export PROJ_BUILD=$PROJ/build-host/bin

# filename of the compiled file/library
export COMPILED=libmlfs.a

# path to original Makefile
```

[2] https://hpc.uni.lu
[3] https://slurm.schedmd.com/overview.html
[4] https://www.gnu.org/software/parallel/
[5] https://sylabs.io

```
export ORIGINAL_MAKEFILE=$PROJ/Makefile

# compilation command of the SUT
export COMPILATION_CMD=(make all ARCH=host EXTRA_CFLAGS="-DNDEBUG" \&\& make all COVERAGE="true" ARCH=
    host_cov EXTRA_CFLAGS="-DNDEBUG")

# compilation additional commands of the SUT (e.g., setup of workspace)
export ADDITIONAL_CMD=(cd $HOME/blts/BLTSConfig \&\& make clean install INSTALL_PATH="$HOME/
    blts_install" \&\& cd $HOME/blts_workspace \&\& $HOME/blts_install/bin/blts_app --init)

# command to be executed after each test case (optional)
export ADDITIONAL_CMD_AFTER=(rm -rf $HOME/blts_workspace/*)

# compilation command for TCE analysis
export TCE_COMPILE_CMD=(make all ARCH=host EXTRA_CFLAGS="-DNDEBUG")

# command to clean installation of the SUT
export CLEAN_CMD=(make cleanall)

# relative path to location of gcov files (i.e., gcda and gcno files)
export GC_FILES_RELATIVE_PATH=Reports/Coverage/Data
```

Listing 12.8: *MASS* variables. Excerpt of mass_conf.sh file.

Also, *MASS* variables shall be configured within the same file. We will run *MASS* with the setup provided in Listing 12.9.

```
### MASS variables

# TCE flags to be tested
export FLAGS=("-O0" "-O1" "-O2" "-O3" "-Ofast" "-Os")

# specify if MASS will be executed on a HPC, possible values are "true" or "false"
export HPC="true"

# set if MASS should be executed with a prioritized and reduced test suite
export PRIORITIZED="true"

# set sampling technique, possible values are "uniform", "stratified", "fsci", and "no"
# note: if "uniform" or "stratified" is set, $PRIORITIZED must be "false"
export SAMPLING="fsci"

# set sampling rate if whether "uniform" or "stratified" sampling has been selected
export RATE=""
```

Listing 12.9: *MASS* specific variables. Excerpt of mass_conf.sh file.

The fourth step consists of configuring the `prepareSUT` configuration file (`MASS_STEPS_LAUNCHERS/PrepareSUT.sh`; within this file the actions from Listing 12.10 must be performed by the engineer:

```
#!/bin/bash

# This file should be prepared by the engineer!
cd /opt/MLFS
. ./mass_conf.sh

# 1. Compile SUT

cd $PROJ

# generate compile_commands.json and delete build
bear make all && rm -rf build* && sed -i 's: libm: /home/mlfs/mlfs/libm:' compile_commands.json && mv
    compile_commands.json $MUTANTS_DIR
eval "${COMPILATION_CMD[@]}"

# 2. Prepare test scripts

cd $HOME/blts/BLTSConfig
make clean install INSTALL_PATH="$HOME/blts_install"

```

```
# Preparing MLFS workspace (e.g., where test cases data is stored)
cd $HOME/blts_workspace
$HOME/blts_install/bin/blts_app --init

# 3. Execute test cases
# Note: execution time for each test case should be measured and passed as argument to FAQAS-CollectCodeCoverage.sh

# example
for tst in $(find $HOME/unit-test-suite -name '*.xml');do
    cd $HOME/blts_workspace

    tst_filename_wo_xml=$(basename -- $tst .xml)

    start=$(date +%s)
    $HOME/blts_install/bin/blts_app -gcrx $tst_filename_wo_xml -b coverage --nocsv -s $tst
    end=$(date +%s)

    # call to FAQAS-CollectCodeCoverage.sh
    # parameter should be test case name and the execution time
    source $MASS/FAQAS-GenerateCodeCoverageMatrixes/FAQAS-CollectCodeCoverage.sh $tst_filename_wo_xml "$(($end-$start))"
done
```

Listing 12.10: *MASS* PrepareSUT.sh file.

The fifth step consists of defining the function `run_tst_case` within the file `mutation_additional_functions.sh` (see Listing 12.11).

```
run_tst_case() {

    tst_name=$1
    tst=$PROJ_TST/$tst_name.xml

    echo $tst_name $tst

    # run the test case
    cd $PROJ_COV
    $HOME/blts_install/bin/blts_app -gcrx $tst_name -b coverage --nocsv -s $tst

    # define if test case execution passed or failed
    summaryreport=$tst_name/Reports/SessionSummaryReport.xml
    originalreport=$HOME/unit-reports/$summaryreport

    test_cases_failed=`xmllint --xpath "//report_summary/test_set_summary/test_cases_failed/text()" $summaryreport`
    o_test_cases_failed=`xmllint --xpath "//report_summary/test_set_summary/test_cases_failed/text()" $originalreport`

    echo "comparing with original execution"
    echo $test_cases_failed $o_test_cases_failed

    if [ "$test_cases_failed" != "$o_test_cases_failed" ]; then
        return 1
    else
        return 0
    fi
}
```

Listing 12.11: Implementation of the run test case Bash function for the MLFS.

The sixth step consists of providing a template for the build script for the trivial compiler optimizations step. In particular, we replaced the following command:

```
CFLAGS = -c -Wall -std=gnu99 -pedantic -Wextra -frounding-math -fsignaling-nans -g O2 -fno-builtin $(EXTRA_CFLAGS)
```

with the following one:

```
CFLAGS = -c -Wall -std=gnu99 -pedantic -Wextra -frounding-math -fsignaling-nans TCE -fno-builtin $(EXTRA_CFLAGS)
```

The seventh step consists of executing the steps PrepareSut and GenerateMutants on the HPC. For this purpose, the SLURM launcher from Listing 12.12 it is provided.

Notice that the workspace has been defined in `$FAQAS/MASS_MLFS/MASS_WORKSPACE`, but it is being binded in `/opt/MLFS`. The step PrepareSut is being executed on the line 27, and the step GenerateMutants on the line 29.

The Singularity container is represented on the file `blts.sif`, which contains a singularity image file. The same sif file will be used throughout all the steps of the methodology.

Also notice that the following SLURM launcher, and all the examples below can be send to the job scheduler with the command `sbatch`.

```
1  #!/bin/bash -l
2
3  #SBATCH -J PrepSUT-GenMut
4  #SBATCH --mem-per-cpu=4096
5  #SBATCH -N 1
6  #SBATCH --ntasks-per-node=1
7  #SBATCH -c 2
8  #SBATCH --time=12:00:00
9
10 RUNDIR=/tmp/MLFS/run
11
12 echo "== Creating MLFS home folder"
13 mkdir -p $RUNDIR
14 cp -r $FAQAS/MASS_MLFS/mutant $RUNDIR
15
16 MLFS_HOME=$RUNDIR/mutant
17
18 echo "== Loading Singularity"
19 module load tools/Singularity
20
21 echo "== Loading container..."
22 singularity instance start --bind $MLFS_HOME:/home/mlfs --bind $FAQAS/MASS_MLFS/unit-test-suite:/home/
       mlfs/unit-test-suite --bind $FAQAS/MASS_MLFS/unit-reports:/home/mlfs/unit-reports --bind $FAQAS/
       MASS_MLFS/MASS_WORKSPACE:/opt/MLFS --bind $FAQAS/srcirorfaqas:/opt/srcirorfaqas $FAQAS/MASS_MLFS/
       blts.sif mlfs_instance
23
24 singularity instance list
25
26 echo "Running singularity instance"
27 srun -N 1 -n 1 -c 2 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
       PrepareSUT.sh
28
29 srun -N 1 -n 1 -c 2 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
       GenerateMutants.sh
30
31 echo "== Stoping singularity instance"
32 singularity instance stop mlfs_instance
```

Listing 12.12: Example of the SLURM launcher for PrepareSUT and GenerateMutants steps.

The eighth step consists of compiling all the generated mutants against the different compilation flags. Since this step can be parallelized, we propose a SLURM launcher implemented with GNU parallels for the CompileOptimizedMutants step. An example is shown in Listing 12.13. This launcher shall be executed with the parameters `-min` and `-max`, indicating the minimum and the maximum of the Bash array that defines the trivial compiler optimizations to be used. For instance, if FLAGS has been defined as `FLAGS=("-O0" "-O1" "-O2" "-O3" "-Ofast" "-Os")` then min should be 0, and max should be 5.

```
1  #!/bin/bash -l
2
3  #SBATCH -J MLFSComp
4  #SBATCH --time=02:00:00
5  ##SBATCH --partition=batch
6  #SBATCH --mem-per-cpu=8192
7  #SBATCH -N 1
```

```
#SBATCH --ntasks-per-node=6
#SBATCH -c 2

SRUN="srun --exclusive -n1 -c ${SLURM_CPUS_PER_TASK:=1} --cpu-bind=cores"

# Parse the command-line argument
while [ $# -ge 1 ]; do
    case $1 in
        -h | --help) usage; exit 0;;
        -n | --dry-run) CMD_PREFIX=echo;;
        --min) shift; MIN=$1;;
        --max) shift; MAX=$1;;
        *) TASK="$*";;
    esac
    shift;
done

# Use the UL HPC modules
if [ -f  /etc/profile ]; then
    .  /etc/profile
fi

module load tools/Singularity

########################
# Data preparation

RUNDIR=$FAQAS/MASS_MLFS

TASK="cp -r $RUNDIR/mutant /dev/shm/mut_{} && \
    singularity instance start --bind /dev/shm/mut_{}:/home/mlfs --bind $RUNDIR/MASS_WORKSPACE:/opt/
     MLFS --bind $FAQAS/srcirorfaqas:/opt/srcirorfaqas $RUNDIR/blts.sif instance_{} \
    sleep 5 && \
    singularity exec instance://instance_{} /bin/bash /opt/MLFS/CompileOptimizedMutants.sh {}"

########################

# Create logs directory
mkdir -p logs

PARALLEL="parallel --delay .2 -j ${SLURM_NTASKS} --joblog logs/state.parallel.log --resume"

${CMD_PREFIX} ${PARALLEL} "${SRUN} ${TASK} 2>&1 | tee logs/parallel_{}.log && ${SRUN} singularity
     instance stop instance_{}" ::: $(seq ${MIN} ${MAX})
```

Listing 12.13: Example of the SLURM launcher for CompileOptimizedMutants step.

The ninth step consists of executing the launcher that processes all the compiled mutants in the previous step, and the launcher that generate prioritized and reduced test suites. Listing 12.14 introduces a SLURM launcher example for this purpose.

```
#!/bin/bash -l

#SBATCH -J OptimizedAndGenPTS
#SBATCH --mem-per-cpu=4096
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH -c 4
#SBATCH --time=12:00:00

#RUNDIR=/tmp/MLFS/run

echo "== Creating MLFS home folder"
#mkdir -p $RUNDIR
#cp -r $FAQAS/MASS_MLFS/mutant $RUNDIR

#MLFS_HOME=$RUNDIR/mutant

echo "== Loading Singularity"
module load tools/Singularity

echo "== Loading container..."
singularity instance start --bind $FAQAS/MASS_MLFS/MASS_WORKSPACE:/opt/MLFS --bind $FAQAS/srcirorfaqas
```

```
    :/opt/srcirorfaqas $FAQAS/MASS_MLFS/blts.sif mlfs_instance

singularity instance list

echo "Running singularity instance"
srun -N 1 -n 1 -c 4 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
    OptimizedPostProcessing.sh

srun -N 1 -n 1 -c 4 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
    GeneratePTS.sh

echo "== Stoping singularity instance"
singularity instance stop mlfs_instance
```

Listing 12.14: Example of the SLURM launcher for OptimizedPostProcessing and GeneratePTS steps.

The tenth step consists of preparing the execution of mutants in the HPC. This can be done following the commands contained in Listing 12.15.

```
#!/bin/bash -l

#SBATCH -J PrepMutExec
#SBATCH --mem-per-cpu=4096
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH -c 4
#SBATCH --time=12:00:00

MASS_WORKSPACE=$FAQAS/MASS_MLFS/MASS_WORKSPACE

echo "== Loading Singularity"
module load tools/Singularity

echo "== Loading container..."
singularity instance start --bind $MASS_WORKSPACE:/opt/MLFS --bind $FAQAS/srcirorfaqas:/opt/
    srcirorfaqas $FAQAS/MASS_MLFS/blts.sif mlfs_instance

singularity instance list

echo "Running singularity instance"
srun -N 1 -n 1 -c 4 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
    PrepareMutants_HPC.sh

echo "== Stoping singularity instance"
singularity instance stop mlfs_instance
```

Listing 12.15: Example of the SLURM launcher for preparing mutants for its execution on the HPC.

The eleventh step consists of executing the mutants on the HPC. Since executing mutants can be parallelized, we propose a SLURM launcher implemented with GNU Parallel. Listing 12.16 provides an example of this step. The script receives three parameters: (i) min – the lower boundary mutant number, (ii) max – the upper boundary mutant number, and (iii) range – the mutant batch number. For instance, the parameters `-min 1 -max 100 -range 1` indicate that the script executes mutants number 1 to 100, and that they belong to the mutant batch #1.

```
#!/bin/bash -l

#SBATCH -J MLFSPar
#SBATCH --time=2-00:00:00
#SBATCH --mem-per-cpu=4096
#SBATCH -N 1
#SBATCH --ntasks-per-node=14
#SBATCH -c 2

SRUN="srun --exclusive -n1 -c ${SLURM_CPUS_PER_TASK:=1} --cpu-bind=cores"

######################## Let's go ##############################
# Parse the command-line argument
while [ $# -ge 1 ]; do
```

```
    case $1 in
        -h | --help) usage; exit 0;;
        -n | --dry-run) CMD_PREFIX=echo;;
        --min) shift; MIN=$1;;
        --max) shift; MAX=$1;;
        --range) shift; RANGE=$1;;
        *) TASK="$*";;
    esac
    shift;
done

# Use the UL HPC modules
if [ -f  /etc/profile ]; then
    .  /etc/profile
fi

module load tools/Singularity

#######################
# Data preparation

RUNDIR=$FAQAS/MASS_MLFS

# to be set, "false" or "true"
REDUCED="false"

mkdir -p $RUNDIR/MASS_WORKSPACE/HPC_MUTATION/runs

TASK="mkdir -p $RUNDIR/MASS_WORKSPACE/HPC_MUTATION/runs/run_{}/test_runs && \
    cp -r $RUNDIR/mutant /dev/shm/mut_{} && \
    singularity instance start --bind /dev/shm/mut_{}:/home/mlfs --bind $RUNDIR/MASS_WORKSPACE/
    HPC_MUTATION/runs/run_{}/test_runs:/home/mlfs/test_runs --bind $FAQAS/srcirorfaqas:/opt/
    srcirorfaqas --bind $RUNDIR/MASS_WORKSPACE:/opt/MLFS --bind $FAQAS/MASS_MLFS/unit-test-suite:/home
    /mlfs/unit-test-suite --bind $FAQAS/MASS_MLFS/unit-reports:/home/mlfs/unit-reports $FAQAS/
    MASS_MLFS/blts.sif instance_{} && \
    sleep 5 && \
    singularity exec instance://instance_{} /bin/bash /opt/MLFS/ExecuteMutants_HPC.sh {} $REDUCED"

#######################

# Create logs directory
mkdir -p logs

PARALLEL="parallel --delay .2 -j ${SLURM_NTASKS} --joblog logs/state.${RANGE}.parallel.log --resume"

${CMD_PREFIX} ${PARALLEL} "${SRUN} ${TASK} 2>&1 | tee logs/parallel_{}.log && ${SRUN} singularity
    instance stop instance_{} && rm -rf /dev/shm/mut_{}" ::: $(seq ${MIN} ${MAX})
```

Listing 12.16: Example of the SLURM launcher for the execution of mutants on the HPC.

To simplify the execution of several mutant batch, the Bash script of Listing 12.17 is provided. Given that the ExecuteMutants script from Listing 12.16 is stored at `$FAQAS/MASS_MLFS/HPC_LAUNCHERS/ExecuteMutants/ExecuteMutants_HPC.sh`, the following script executes two batch of mutants of 336 mutants each.

```
#/bin/bash

LAUNCHER=$FAQAS/MASS_MLFS/HPC_LAUNCHERS/ExecuteMutants/ExecuteMutants_HPC.sh
JOBNAME=MLFS

min=0
max=600
chunksize=336

j=1
for i in $(seq $min $chunksize $max); do
    sbatch \
        -J ${JOBNAME}_$j \
        ${LAUNCHER} --min $((i+1)) --max $((i+chunksize)) --range $j;
    j=$((j+1))
done
```

Listing 12.17: Example of a Bash launcher for the ExecuteMutants_HPC.sh script.

The twelfth step consists of processing the executed mutants, and verifying if its necessary to execute a new batch of mutants. An example SLURM script is provided in Listing 12.18.

```
#!/bin/bash -l

#SBATCH -J PostMutExec
#SBATCH --mem-per-cpu=4096
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH -c 4
##SBATCH --time=12:00:00
#SBATCH --time=02:00:00

MASS_WORKSPACE=$FAQAS/MASS_MLFS/MASS_WORKSPACE

echo "== Loading Singularity"
module load tools/Singularity

echo "== Loading container..."
singularity instance start --bind $MASS_WORKSPACE:/opt/MLFS --bind $FAQAS/srcirorfaqas:/opt/
    srcirorfaqas $FAQAS/MASS_MLFS/blts.sif mlfs_instance

singularity instance list

echo "Running singularity instance"
srun -N 1 -n 1 -c 4 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
    PostMutation_HPC.sh 1 672

echo "== Stoping singularity instance"
singularity instance stop mlfs_instance
```

Listing 12.18: Example of a SLURM launcher for the PostMutation step.

The thirteenth step consists of identifying equivalent mutants based on code coverage, and computing the final mutation score. Listing 12.19 provides an example of a SLURM launcher for executing both steps.

```
#!/bin/bash -l

#SBATCH -J IdEquiv
#SBATCH --mem-per-cpu=4096
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH -c 4
#SBATCH --time=1-00:00:00

RUNDIR=/tmp/MLFS/run

echo "== Creating MLFS home folder"
mkdir -p $RUNDIR
cp -r $FAQAS/MASS_MLFS/mutant $RUNDIR

MLFS_HOME=$RUNDIR/mutant

echo "== Loading Singularity"
module load tools/Singularity

echo "== Loading container..."
singularity instance start --bind $MLFS_HOME:/home/mlfs --bind $FAQAS/MASS_MLFS/MASS_WORKSPACE:/opt/
    MLFS --bind $FAQAS/srcirorfaqas:/opt/srcirorfaqas $FAQAS/MASS_MLFS/blts.sif mlfs_instance

singularity instance list

echo "Running singularity instance"
srun -N 1 -n 1 -c 4 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
    IdentifyEquivalents.sh

srun -N 1 -n 1 -c 4 --exclusive singularity exec instance://mlfs_instance /bin/bash /opt/MLFS/
    MutationScore.sh

echo "== Stoping singularity instance"
singularity instance stop mlfs_instance
```

Listing 12.19: Example of a Bash launcher for the steps IdentifyEquivalents and MutationScore.

The output of the last step should provide the final output of MASS.

```
##### MASS Output #####
## Total mutants generated: 28071
## Total mutants filtered by TCE: 6914
## Sampling type: fsci
## Total mutants analyzed: 672
## Total killed mutants: 550
## Total live mutants: 122
## Total likely equivalent mutants: 86
## MASS mutation score (%): 93.85
## List A of useful undetected mutants: /opt/MLFS/DETECTION/test_runs/useful_list_a
## List B of useful undetected mutants: /opt/MLFS/DETECTION/test_runs/useful_list_b
## Number of statements covered: 1955
## Statement coverage (%): 100
## Minimum lines covered per source file: 2
## Maximum lines covered per source file: 138
```

Listing 12.20: *MASS* output.

Chapter 13

SEMUS - ASN.1 Tutorial

13.1 Introduction

This tutorial instructs on how to use *SEMuS* on the ASN.1 case study provided by ESA.

13.2 Running *SEMuS*

For running *SEMuS* the user needs to set up the toolset, for this task please refer to Section 8.1.

The objective of this tutorial is to generate test inputs that kill the mutants non detected by the ASN.1 test suite. For this reason, we consider a precondition of this tutorial to have applied *MASS* to the ASN.1 case study, and thus to have the list of live mutants.

We consider as a target of the test generation the autogenerated code from ASN.1, i.e., `test.c`.

SEMuS is distributed with a set of scripts already configured, meaning that the user does not need to edit them. All the code concerning the ASN.1 case study can be found on `faqas_semu/case_studies/ASN/`

In general, the steps for generating tests for the ASN.1 case study are:

1. Configure the file `faqas_semus_config.sh` (Already filled by SnT).
2. Configure the `generate_template_config.json` file (Already filled by SnT).
3. Generate the test templates for the SUT functions.
4. Launch the test generation process for the case study.
5. Verifying the generated unit test cases.

13.2.1 Step 1: configuring *SEMuS*

The first step consists of configuring *SEMuS*; for doing so it is necessary to provide the paths for the SUT paths, the SUT compilation commands, the output folders, and the configuration of *SEMu* for guiding the symbolic search. Listing 13.1 provides an example of configuration file for the case study, in this case, we already provide a filled version of it, the file can be found at `faqas_semu/case_studies/ASN/s`

```
FAQAS_SEMU_CASE_STUDY_TOPDIR=../

FAQAS_SEMU_CASE_STUDY_WORKSPACE=$FAQAS_SEMU_CASE_STUDY_TOPDIR/WORKSPACE

FAQAS_SEMU_OUTPUT_TOPDIR=$FAQAS_SEMU_CASE_STUDY_WORKSPACE/OUTPUT

FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/mutants_generation

FAQAS_SEMU_REPO_ROOTDIR=$FAQAS_SEMU_CASE_STUDY_WORKSPACE/DOWNLOADED/casestudy

FAQAS_SEMU_ORIGINAL_SOURCE_FILE=$FAQAS_SEMU_REPO_ROOTDIR/test.c

FAQAS_SEMU_COMPILE_COMMAND_SPECIFIED_SOURCE_FILE=./test.c

FAQAS_SEMU_GENERATED_MUTANTS_DIR=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/test

FAQAS_SEMU_BUILD_CODE_FUNC_STR='
FAQAS_SEMU_BUILD_CODE_FUNC()
{
    local in_file=$1
    local out_file=$2
    local repo_root_dir=$3
    local compiler=$4
    local flags="$5"
    # compile
    $compiler $flags -g -Wall -Werror -Wextra -Wuninitialized -Wcast-qual -Wshadow -Wundef -
    fdiagnostics-show-option -D_DEBUG -I $repo_root_dir -O0 $in_file -o $out_file $flags
    return $?
}
'

FAQAS_SEMU_BUILD_LLVM_BC()
{
    local in_file=$1
    local out_bc=$2
    eval "$FAQAS_SEMU_BUILD_CODE_FUNC_STR"
    FAQAS_SEMU_BUILD_CODE_FUNC $in_file $out_bc $FAQAS_SEMU_REPO_ROOTDIR clang '-c -emit-llvm'
    return $?
}

FAQAS_SEMU_META_MU_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/meta_mu_topdir

FAQAS_SEMU_GENERATED_META_MU_SRC_FILE=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/test.MetaMu.c

FAQAS_SEMU_GENERATED_META_MU_BC_FILE=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/test.MetaMu.bc

FAQAS_SEMU_GENERATED_META_MU_MAKE_SYM_TOP_DIR=$FAQAS_SEMU_GENERATED_MUTANTS_TOPDIR/"MakeSym-TestGen-
    Input"

FAQAS_SEMU_GENERATED_TESTS_TOPDIR=$FAQAS_SEMU_OUTPUT_TOPDIR/test_generation

# timeout in seconds
FAQAS_SEMU_TEST_GEN_TIMEOUT=300

# This is the config for SEMU heuristics. The accepted values of 'PSS' are 'RND' for random and 'MDO'
    for minimum distance to output
FAQAS_SEMU_HEURISTICS_CONFIG='{
        "PL": "0",
        "CW": "4294967295",
        "MPD": "0",
        "PP": "1.0",
        "NTPM": "5",
        "PSS": "RND"
}'

# max Test Generation memory in MB
FAQAS_SEMU_TEST_GEN_MAX_MEMORY=2000

# Set to 'ON' to stop test generation when the memory limit is reached
FAQAS_SEMU_STOP_TG_ON_MEMORY_LIMIT='OFF'

# Set this to 'ON' so thae the states the sate fork is disabled when the memory limit is reached, to
    avoid going much over it
FAQAS_SEMU_TG_MAX_MEMORY_INHIBIT="ON"
```

Listing 13.1: faqas_semus_conf.sh file for ASN case study.

Concerning *SEMu* configuration, we can see that we have setup the tool to run for a maximum of 5 minutes, and to use a maximum of 2000 MB of memory.

13.2.2 Step 2 and 3: configuring the `generate_template_config.json` file and generating test templates

In this step, we need to configure the `generate_template_config.json`, remember that this JSON file provides detailed information about how to interpret SUT function to *SEMuS*. Listing 13.2 shows an example of configuration JSON file for the case study. It indicates that the parameter *pErrCode* acts as an output parameter (see *OUT_ARG_NAMES*). Also, it provides customized instructions to specify how to instantiate and initialize variables; for instance, the listing shows that the type `struct BitStream_t` shall be initialized with the support of the `BitStream_Init` function.

```
{
    "TYPES_TO_INTCONVERT": {"flag": "(int){}"},
    "TYPES_TO_PRINTCODE": {},
    "OUT_ARGS_NAMES": ["pErrCode"],
    "IN_OUT_ARGS_NAMES": [],
    "TYPE_TO_INITIALIZATIONCODE": {"struct BitStream_t": "static byte encBuff[
    T_POS_SET_REQUIRED_BYTES_FOR_ENCODING + 1];\n\tBitStream_Init(&{}, encBuff,
    T_POS_SET_REQUIRED_BYTES_FOR_ENCODING)"},
    "TYPE_TO_SYMBOLIC_FIELDS_ACCESS": {"struct BitStream_t": {}},
    "VOID_ARG_SUBSTITUTE_TYPE": "",
    "ARG_TYPE_TO_ITS_POINTER_ELEM_NUM": {}
}
```

Listing 13.2: JSON configuration file for ASN.1.

To generate the the test templates, the user shall execute the Python script `generate_direct.py` for the generation of test templates by passing as argument (1) the source file to be analyzed, (2) the include argument of the library, (3) the `generate_template_config.json` file.

```
$ ./generate_direct.py ../WORKSPACE/DOWNLOADED/casestudy/test.c direct " -I../WORKSPACE/DOWNLOADED/
    casestudy/" -c generate_template_config.json
```

The previous command will generate the following template for the `T_POS_SET_Encode` function.

```
#include <stdio.h>
#include <string.h>

#include "asn1crt.c"
#include "asn1crt_encoding.c"
#include "asn1crt_encoding_uper.c"

#include "klee/klee.h"

int main(int argc, char** argv)
{
    (void)argc;
    (void)argv;

    // Declare variable to hold function returned value
    _Bool result_faqas_semu;

    // Declare arguments and make input ones symbolic
    T_POS_SET pVal;
    struct BitStream_t pBitStrm;
    int pErrCode;
```

```
    _Bool bCheckConstraints;
    memset(&pVal, 0, sizeof(pVal));
    memset(&bCheckConstraints, 0, sizeof(bCheckConstraints));
    static byte encBuff[T_POS_SET_REQUIRED_BYTES_FOR_ENCODING + 1];
    BitStream_Init(&pBitStrm, encBuff, T_POS_SET_REQUIRED_BYTES_FOR_ENCODING);
    klee_make_symbolic(&pVal, sizeof(pVal), "pVal"); //T_POS_SET
    klee_make_symbolic(&bCheckConstraints, sizeof(bCheckConstraints), "bCheckConstraints"); //_Bool

    // Call function under test
    result_faqas_semu = T_POS_SET_Encode(&pVal, &pBitStrm, &pErrCode, bCheckConstraints);

    // Make some output
    printf("FAQAS-SEMU-TEST_OUTPUT: pErrCode = %d\n", pErrCode);
    printf("FAQAS-SEMU-TEST_OUTPUT: result_faqas_semu = %d\n", result_faqas_semu);
    return (int)result_faqas_semu;
}
```

13.2.3 Step 4: launching the test generation process

For the sake of this tutorial we consider the test generation for mutants of the function `T_POS_SET_Encode`. For this, the user shall provide a file text containing the name of the mutants, and will place it in folder `case_studies/ASN/WORKSPACE/DOWNLOADED/live_mutant`.

The engineer shall then execute one test generation command (i.e., invoke the script *run.sh*), the command follows:

```
$ scripts/run.sh mutation WORKSPACE/DOWNLOADED/live_mutant WORKSPACE/OUTPUT/live_mutants_output
```

13.2.4 Step 5: verifying the generated test cases

SEMuS' output is stored in the directory `WORKSPACE/OUTPUT/live_mutants_output`. The generated unit test cases are stored in the directory `live_mutants_output/test_generation/<test template name>/<function under test>/FAQAS_SEMU-out/produced-unittests`.

For example, after test generation, in the folder `produced-unittests` that has been created by *SEMuS* when testing the function `T_POS_SET_Encode`, we will find the following files:

- runtest.sh
- test000001.ktest.c
- test000001.ktest.c.expected
- test000002.ktest.c
- test000002.ktest.c.expected

The Bash script `runtest.sh` provides the necessary commands to execute the generated test case, the files with suffix `.ktest.c` are the test cases generated by *SEMuS*, while the files with extension *.expected* contain the output that is observed when executing the test case with the current version of the SUT.

The generated test case can be executed using the following command:

```
$ ./runtest.sh test000001.ktest.c
```

The command above will generate the text file `test000001.ktest.c.got`, which stores the system outputs generated during the execution of the test case. The script will also compare the observed output (i.e., the file with extension *.got*) with the output generated during test generation (i.e., the file with extension *.expected*) through a *diff* command. If the function under test was not modified, the `runtest.sh` script should not output any difference.

The command `runtest.sh` becomes handy in a CI/CD context; indeed, the test cases generated by *SEMuS* can be reused as is to determine regression fault. When a new version of the SUT is available, the engineers can simply replace the content of FAQAS_SEMU_REPO_ROOTDIR (i.e., the folder with the SUT) with the newer SUT version. The execution of command `runtest.sh` will thus show the presence of differences with respect to a previous version. If the function under test has not been updated in the new version, the presence of changes may indicate a regression.

Additionally, the user may generate the summary report of *SEMuS*, through the following command:

```
$ ./generateReport.sh
```

The output of this command can be found at `case_studies/$SUT/WORKSPACE/OUTPUT/AnalysisReport.csv`:

```
Number of analyzed mutants: 20
Number of killed mutants: 2
Number of live mutants: 18
test.mut.2489.1_1_2.SDL.T_POS_SET_Encode.c;KILLED
test.mut.2489.2_1_14.ICR.T_POS_SET_Encode.c;LIVE
test.mut.2489.1_2_14.ICR.T_POS_SET_Encode.c;LIVE
test.mut.2490.4_1_74.LVR.T_POS_SET_Encode.c;LIVE
test.mut.2491.2_1_10.LCR.T_POS_SET_Encode.c;LIVE
test.mut.2491.2_2_10.LOD.T_POS_SET_Encode.c;LIVE
test.mut.2491.1_1_6.LOD.T_POS_SET_Encode.c;LIVE
test.mut.2491.2_5_23.ROR.T_POS_SET_Encode.c;LIVE
test.mut.2491.4_3_23.ROR.T_POS_SET_Encode.c;LIVE
test.mut.2502.4_8_97.ICR.T_POS_SET_Encode.c;LIVE
test.mut.2502.5_9_97.ICR.T_POS_SET_Encode.c;LIVE
test.mut.2504.2_2_66.LOD.T_POS_SET_Encode.c;LIVE
test.mut.2504.6_1_32.ROR.T_POS_SET_Encode.c;LIVE
test.mut.2506.5_1_96.LVR.T_POS_SET_Encode.c;KILLED
test.mut.2510.5_9_94.ICR.T_POS_SET_Encode.c;LIVE
test.mut.2510.4_8_94.ICR.T_POS_SET_Encode.c;LIVE
test.mut.2512.6_1_36.ROR.T_POS_SET_Encode.c;LIVE
test.mut.2518.5_1_86.LVR.T_POS_SET_Encode.c;LIVE
test.mut.2521.5_1_90.LVR.T_POS_SET_Encode.c;LIVE
test.mut.2524.5_1_98.LVR.T_POS_SET_Encode.c;LIVE
```