



EVALUATION OF DAMAT TOOLSET

GomSpace

Document Title:	Evaluation of DAMAt toolset		
Revision number:	1.0	Date:	2021-10-22

Release Table:

Action	Name	Function	Date	Signature
Prepared / Owned by:	Nikhil Mehta	Software Engineer	2021-10-21	
Verified / Reviewed by:	Alastair Isaacs	Mission Lead	2021-10-22	

Document Change Log

Revision	Date	Name	Description
0.1	2021-10-11	NIME	DRAFT for internal release
0.2	2021-10-21	NIME	DRAFT for internal release
1.0	2021-10-22	ALIS	FINAL for release

TABLE OF CONTENTS

1	INTRODUCTION.....	1
1.1	PURPOSE.....	1
2	DOCUMENTS.....	2
2.1	REFERENCE DOCUMENTS.....	2
3	OVERVIEW	3
3.1	GENERAL	3
3.2	BACKGROUND	3
3.3	SOFTWARE LIBRARIES.....	4
3.4	OBJECTIVES OF EVALUATION	4
3.4.1	Effectiveness.....	4
3.4.2	Scalability.....	4
3.4.3	Efficiency.....	5
3.4.4	Applicability.....	5
4	EVALUATION.....	6
4.1	EASE OF USE.....	6
4.1.1	Documentation.....	6
4.1.2	Configuration.....	6
4.2	EFFECTIVENESS	7
4.3	SCALABILITY	7
4.4	EFFICIENCY.....	7
4.5	APPLICABILITY	7
5	CONCLUDING REMARKS	9
6	SEMUS EVALUATION	10
6.1	EASE OF USE.....	10
6.1.1	Documentation.....	10
6.1.2	Configuration.....	10
6.2	EFFECTIVENESS	10
6.3	SCALABILITY	10
6.4	EFFICIENCY.....	10
6.5	APPLICABILITY	10

1 Introduction

1.1 Purpose

This document reports the evaluation activities performed by GomSpace Luxembourg on the toolset implementing data-driven mutation analysis (DAMAt) delivered by the Interdisciplinary Centre for Security, Reliability and Trust (SNT) at the University of Luxembourg. This toolset represents a portion of the overall FAQAS framework, namely data-driven mutation analysis.

2 Documents

2.1 Reference Documents

Reference	Document Title	Issue	Date
RD-01	D2: Study of mutation testing applicability to space software	Issue 1 Rev 1	2020-06-24
RD-02	FAQAS Framework Software User Manual	Issue 1 Rev 1	2021-09-24

3 Overview

3.1 General

Data driven mutation testing is a formalization of the test suite assessment process based on the injection of faults in the data processed by software components. Data-driven mutation testing aims to assess test suites by simulating faults that affect the data produced, received, or exchanged by the software and its components. It is based on a fault model capturing the type of data faults that might affect the system.

The fault model is produced by software engineers based on their domain knowledge and experience. The considered faults might be due to programming errors, hardware problems, or critical situations in the environment (e.g.: channel noise). The data is automatically mutated by a set of operators that aim to replicate the faults in the fault model. A simple operator is the bit flip operator, which could be implemented through a procedure that flips a randomly selected bit of every field of the transmitted data.

Techniques to inject data faults in the data processed by software systems have been applied mostly to test software systems (e.g., to determine if the software is robust against errors in the data being processed) but not to assess the quality of test suites. For systems, or components, exchanging structured data (e.g., message sequences), a data fault may consist of either an invalid data structure (i.e., a structure that does not respect the data model of the system) or an illegal data value.

For systems, or components, processing signals, a data fault may result in signals that do not respect the characteristics observed in the original signal. Example of signal features are value, derivative, second derivative. Since data-driven mutation testing alters the data produced, received, or exchanged by the software or its components, it should be applied to evaluate test suites that trigger the execution and communication between multiple components (e.g.: system or integration test cases). Data-driven mutation testing is not meant to be applied to assess unit test suites.

For further details on the mutation analysis approach used by the DAMAt, see RD01.

3.2 Background

GomSpace is a manufacturer and operator of small satellites known as cubesats. As well as hardware, GomSpace also produces software for commanding and operating satellites. This software is based around the CubeSat Space Protocol (CSP), a protocol stack written in C and used for communication and commanding of the satellite.

The software libraries developed by GomSpace for use onboard satellites are tested using an automated test approach on ground. The test suites involved are prepared by software developers and aim to ensure a high quality of code before deployment and placement in the space environment. GomSpace does not currently employ mutation testing to evaluate the quality of these test suites.

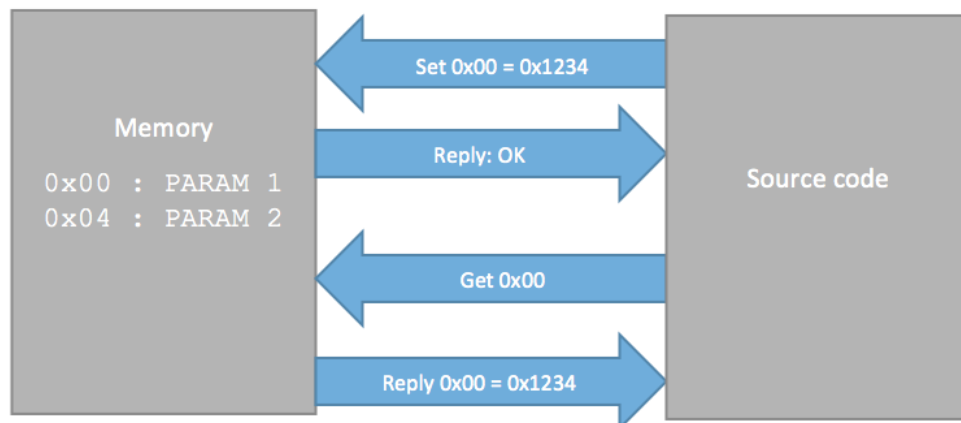
CubeSats are built, integrated, and tested on a shorter timescale than is normal in more traditional satellite manufacturing. This means that the time available for testing and quality control is limited, and test cases must therefore be as complete as possible while executing in a short period. To this end, GomSpace applies a process of continuous integration and deployment (CI/CD) that emphasizes short development and test cycles. To be successful, the DAMAt should be usable within this CI/CD framework, and capable of rapidly (i.e., on timescales as short as daily) evaluating changing code and test suites.

3.3 Software Libraries

The libparam is a light-weight parameter system designed for GomSpace satellite subsystems. It is based around a logical memory architecture, where every parameter is referenced directly by its logical address. A backend system takes care of translating addresses into physical addresses.

The features of this system include:

- Direct memory access for quick parameter reads.
- Simple data types: uint, int, float, double, string.
- Arrays of simple data types.
- Supports multiple stores per table, e.g. FRAM, MCU flash, file (binary or text).
- Remote client with support for most features (rparam).
- Packed GET, SET queries, supporting multiple parameter set/get in a single request.
- Data serialization and deserialization.
- Supports both little and big-endian systems.
- Commands for both local (param) and remote access (rparam).
- Parameter server for remote access over CSP.
- Compile-time configuration of parameter system



3.4 Objectives of Evaluation

The evaluation performed by GomSpace had the following objectives:

- To evaluate the effectiveness, scalability, efficiency, and applicability of FAQAS to space software
- To compare the effectiveness with state-of-the-art approaches in terms of the capability of spotting deficiencies in test suites
- To evaluate the effort required to put it in practice, its potential to uncover errors and the verification and validation activities to which it can be applied

3.4.1 Effectiveness

Effectiveness can be defined as the degree to which something – in this case the DAMAt – is successful in producing a desired result. The desired result in this case is an evaluation of the quality of the test suite based on the injection of faults in the data processed by software under test and highlighting of the mutants that survive the test suite; and to do so in a reasonable time.

3.4.2 Scalability

Scalability is the ability of a system to adapt to increased demands in term of performance, maintenance, and availability. In the case of DAMAt, the goal is to achieve easy horizontal scaling to perform mutation

testing on many software components. Horizontal scaling means adding more instances of software instead of increasing computing power.

3.4.3 Efficiency

The efficiency of test software can be measured as number of test cases executed divided by unit of time (for example number of executed test cases per minute). The goal is to have a quick feedback loop that allows the introduction of mutation testing as part of CI/CD pipelines. In this way a company could monitor quality of a test suite after every code or test case change.

3.4.4 Applicability

The applicability of software refers to how useful it is. The goal is to assess if DAMAt is a tool needed in the space industry to improve software reliability. As part of this assessment, GomSpace's libparam library will be placed under such mutation testing.

The need for reliable software and high-quality test suites is especially acute when working with the space environment. Objects in space cannot be directly accessed for repair or modification, and as a result a missed defect, whether in hardware or software, can be fatal to a mission. Traditionally, satellites have undergone a large degree of testing before launch to reduce as much as possible the risk of a defect.

However, as timelines are often shorter, cubesats and the newspace industry require a more rapid approach to testing and quality. There is thus a need for a large degree of automation in testing and more rapid iterations of software and testing, more akin to the software practices seen in other parts of the technology industry. The challenge for tools like DAMAt is to make sure that mission critical SUT is reliable when faults in data are introduced or if there is garbage data then the software has the capability to handle itself without leading to undesired failures or breakdowns. This ability will greatly improve the reliability of the software.

4 Evaluation

4.1 Ease of Use

4.1.1 Documentation

Before starting the DAMAt toolset, the project must be configured. The steps for this configuration are provided in the Software User Manual (SUM) (see RD02).

The SUM contains all necessary information in a well written style. This includes an explanation of the library structure and the purpose of contained files, a description of all configuration variables within those files and instructions for running the toolset. Each important file is provided with its own subsection, allowing the end-user to get a clear understanding of the framework configuration.

The evaluation did identify that some of the missing steps in the SUM were present in the readme of the project. Without which it was quite difficult to configure some of the steps, as there is no explanation in the document.

Example: the below steps are present in the Readme but not in the SUM:

This version of DAMAt comes with the additional folder "libparam_set_up"

This is how to set up DAMAt to work with libparam:

- 1) substitute "damat_pipeline/libparam_set_up/DAMAt_compile.sh" and "damat_pipeline/libparam_set_up/DAMAt_run_tests.sh" in place of the stubs that you find in the "damat-pipeline" folder.*
- 2) copy the "damat_pipeline/libparam_set_up/fault_model_param.csv" and "damat_pipeline/libparam_set_up/tests_param.csv" in the main folder.*
- 3) make sure that the variables in DAMAt_configure.sh are correctly defined.*

They should be:

```
tests_list=$DAMAt_FOLDER/tests_param.csv
fault_model=$DAMAt_FOLDER/fault_model_param.csv
buffer_type="unsigned long long int"
padding=0
singleton="TRUE"
```

- 4) copy "damat_pipeline/libparam_set_up/wscript" in the "libparam" folder.*
- 5) copy "damat_pipeline/libparam_set_up/csp_service_handler.c" in the "libparam/src" folder.*
- 6) the folder to copy the FAQAS_dataDrivenMutator.h, once generated is "libparam/include/gs/param"*

Finally, the SUM doesn't have sufficient information about fault models and how they are important to the process, analysis, and results. Only an example is present in the SUM. It would be useful to have some information explaining fault models, their significance, and some background of its correlation to probes, testcases etc.

4.1.2 Configuration

The DAMAt toolset offers many configuration options to the end-user. All these configuration options and parameters are well described in the SUM. The ease of configuring these parameters has increased compared to the previous MASS tool. For example: `DAMAt_FOLDER=$(pwd)` and using this variable further for configuration significantly saves time.

But at the same time we need to manually inject probes into the code which requires prior additional knowledge/understanding of inner working of the software under test (SUT). Moreover, the injected probes should always be used only for tests. In production, they shouldn't be present and there is necessity of automated process that handles probes management.

The way in which probes are testing the code is as follows: DAMAt generates a code that must be inserted into a SUT and that code is modifying the original behaviour. For that reason, mutation probes should never remain in the production code – communication between modules would be broken. This prevents us from running such data-driven mutation testing on a regular basis. The process requires manual

intervention and thus we are not able to use automation engines like Jenkins to run the test suite. Our suggestion would be to modify the configuration part. Instead of manually inserting generated probes, an engineer could rather leave a comment with an identifier. Then an additional tool could scan the original source code and replace such comments with corresponding probes. Such an improvement unblocks the possibility to integrate data-mutation testing with DAMAt into automatic Jenkins pipelines (or any other automation engine like Bamboo or Travis).

4.2 Effectiveness

DAMAt was used to evaluate the test suite used with libparam. The toolset executed successfully and produced a report with the number of created, killed, and surviving mutants. Analysis of the surviving mutants shows the toolset does identify valid (potential) test cases that the test suites currently miss. This implies the presence of missing test cases, often in areas that can be considered as challenging edge cases that are difficult for a developer or dedicated software tester to anticipate, and in some cases poorly written test cases. Based on the results generated, the DAMAt tool is effective at identifying potential defects and missing test cases that are unanticipated by our current test suites.

Based on the metrics (MutationID, Fault Model, Status, Application, Description) generated in the result file it is quite convenient to trace back the problems in the test suite by analysing the results with the probes inserted. As a result, it can lead to the test cases which needs an update or to functions for which test cases are missing. It is also worth mentioning that closer examination of the code inspired by this approach did seem to reveal actual defects in the software that were previously unknown.

Last but not least, it should be noted that manual insertion of probes, creation of fault model and analysis of the results are quite extensive and time-consuming processes.

4.3 Scalability

DAMAt can be containerized. The FAQAS team used Singularity as a container system, however it is also possible to create a Docker image that allows running DAMAt mutation tests in a docker container. Configuration files can be stored together with source code and mounted as volumes. In principle this makes it trivial to spawn a new instance (horizontal scaling). Also addition of probes as manual step would make it more difficult to scale especially when it is the case of microservices when there are large number of interacting microservices.

4.4 Efficiency

Currently it takes a few hours to perform DAMAt testing against libparam from scratch on a regular PC (two steps – DAMAt_probe_generation.sh, DAMAt_mutants_launcher.sh along with configuring and setting the pipeline). It also depends on how many entries are applicable in the fault model and the list of testcases

Keeping in mind how many operations are performed during such testing this is still a very good result – however it is too long to run such tests after any change to a codebase which would be necessary for integration with a CI/CD pipeline. This, at least without efficiency improvements, means this analysis would be reserved for more infrequent milestones (for example at the end of a SCRUM sprint, before a project review, after developing new tests etc.).

4.5 Applicability

The evaluation showed that the DAMAt would be considered as a useful addition to GomSpace's testing processes.

After checking the output report, GomSpace realised that the test suite of libparam does not address certain side effects of functions. This already demonstrates the ability of the toolset to identify improvements that would raise the quality of existing test suites. Overall, however, the number of killed mutants were moderate, which provides GomSpace to improve upon libparam test suite. But a note can be made that a prior knowledge of on which functions the mutation would be generated is quite handy. Otherwise, it is again time-consuming process to understand and identify the potential issues just based on the results derived after execution.

Applying this approach to other libraries maintained by GomSpace could allow managers to direct efforts towards improving test suites identified as lower quality (i.e., those with more surviving mutants) or to set certain gateway thresholds (i.e., a certain proportion of mutants must be caught before a test suite is considered high enough quality to proceed). This would lead to an overall improvement in both test suite and code quality. But keep in mind that there is manual intervention of adding lots of probes and building a fault model which can be a time-consuming process.

5 Concluding Remarks

The evaluation shows that the DAMAt toolset fulfils the expected role. It can also, according to the evaluation results, help to build more reliable software and be incorporated as a part of standard testing approaches at GomSpace.

The evaluation also concludes that:

- The setup process is generally well documented and practical to implement
- The user documentation should include all the missing steps that exists in separate readme file
- The data modelling and fault modelling is time consuming process along with addition of probes.
- The toolset takes too long to complete to form a part of the regular CI/CD pipeline, however it could be run less regularly to evaluate quality of test suites, and indirectly, code.
- The approach and toolset are useful enough to provide a meaningful benefit to the current GomSpace test and quality process

6 SEMUS Evaluation

There was not enough time to Evaluate SEMUS completely. The following evaluation is done in a small amount of time.

6.1 Ease of Use

6.1.1 Documentation

Before starting the SEMUS toolset, the project must be configured. The steps for this configuration are provided in the Software User Manual (SUM) (see RD02).

However, there are some typos in commands in SUM which can be corrected for error free configuration.

Example : `case_studies/LIBUTIL/util_codes/call_generated_direct.sh src/timestamp.c` instead it should be `case_studies/LIBUTIL/util_codes/call_generate_direct.sh src/timestamp.c`

6.1.2 Configuration

The SEMUS offers many configuration options to the end-user. All these configuration options and parameters are well described in the SUM. There are also scripts that are used for automatic generation of JSON files and Test templates.

6.2 Effectiveness

SEMUS was used to evaluate the test suite used with libutil based on the mutant generation from MASS. The toolset generates additional tests to see whether the mutants survived in MASS can be killed effectively. Analysis of the generated testcases for the mutants identified valid bug for timestamp.c as well as missing Test cases. This implies the presence of missing test cases, often in areas that can be considered as challenging edge cases that are difficult for a developer or dedicated software tester to anticipate and in some cases poorly written test cases.

6.3 Scalability

SEMUS can be containerized. The FAQAS team used Singularity as a container system, however it is also possible to create a Docker image that allows running SEMUS test generation in a docker container. Configuration files can be stored together with source code and mounted as volumes. In principle this makes it trivial to spawn a new instance (horizontal scaling). Also creation of JSON file and test template can be it more difficult to scale especially when it is the case of microservices when there are large number of interacting microservices.

6.4 Efficiency

Currently it takes a few hours to execute SEMUS against libutil from scratch on a regular PC. If you run it on Windows machine with WSL or vagrant, there are chances you might run in few troubles with respect to versioning of different libraries/container/virtual env etc.

However, when it is fully executed it does help with identifying the missing test cases in test suite and as a side-effect points out to potential bugs.

6.5 Applicability

The evaluation showed that the SEMUS would be considered as a useful addition to GomSpace's testing processes.

After checking the output report, GomSpace realised that the test suite of libutil does not address certain tests. This already demonstrates the ability of the SEMUS to identify improvements that would raise the quality of existing test suites

Applying this approach to other libraries maintained by GomSpace could allow managers to direct efforts towards improving test suites identified as lower quality (i.e., tests generated for surviving mutants). This would lead to an overall improvement in both test suite and code quality. But keep in mind that there is manual intervention of generating large amount of JSON and test templates which can be a time-consuming process in case of large software libraries.