17th International Conference on Space Operations, Dubai, United Arab Emirates, 6 - 10 March 2023.

SpaceOps-2023, ID # 436

Integrated Framework for Software Testing and Verification based on Open Source Software

Ruediger Gad[a*], Francesco Croce[b], Jessica Cowley[a], Ales Simonic[c], Meriem Drira[a], Carlos Miranda[b]

[a] Terma GmbH, Bratustraße 7, 64293 Darmstadt, Germany, ruga@terma.com
[b] EUMETSAT, Generic Systems and Infrastructure Division, Monitoring and Control Applications and Tools, Eumetsat Allee 1, 64295 Darmstadt, Germany, Francesco.Croce@eumetsat.int
[c] IGNIS d.o.o., Slovenskogoriška cesta 39, 2250 Ptuj, Slovenia, ales.simonic@ignis-it.eu
* Corresponding Author

Abstract

In this paper, we present an integrated framework for supporting software testing and verification based on Open Source Software solutions. The paper describes the framework and OSS selection including, e.g., their role within test definition and execution, test results management, overall documentation production, and enhancements we implemented.

The context is an application based on a microservices architecture for back-end elements and generic Human Machine Interfaces for front-end clients. Both extensively adopt cloud and web technologies. To establish this context, this paper provides a summary of the application for which the testing framework was developed and is applied.

The paper concludes with considerations and lessons learned on the framework implementation, adoption, and its usage.

Keywords: Ground Segment Software, Testing, Verification, Automated Testing

1 Introduction

The context of this paper is formal testing and verification of complex ground segment software applications for EUMETSAT as part of a new generation of software elements development within the satellite Monitoring and Control Applications (M&C) functional domain. The overall architecture of the such new generation of applications is based on multi-systems multi-mission generic elements on top of which different specific high-end applications can be defined and developed in terms of back-end cloud native microservices hosted by Platform-as-a-Service (PaaS) and front-end Human Machine Interface (HMI) clients based on web technologies. A description of the applications development objectives and context can be found at [1].

The applications software development project follows a formal process regarding, e.g., traceability of requirements, delivery milestones, delivery verification, and documentation production. The process is loosely comparable to ECSS-E-ST-40C [2] with review milestones such as Software Requirements Review (SRR), Preliminary Design Review (PDR), etc.

The original technical and development lifecycle requirements asked for a high level of automatic testing at component and system level for both back-end and front-end elements. Unit testing was required to be ensured by the test unit suite technology for the software language in use by a specific element.

The complexity of the software and the formal requirements posed significant challenges for the testing and verification of the software. The aim of the testing framework presented in this paper is to support the formal process, e.g., by increasing the automation level, reducing duplication across information sources, and supporting the involved teams.

We aimed on re-using as much available solutions as possible. However, we could not find a readily available turn-key solution that fulfilled our needs. Thus, we implemented the testing framework based on other readily available solutions that each cover subsets of the overall testing framework functionality.

We focused on re-using as much Open Source Software (OSS) solutions as possible. We consider that using established OSS solutions eases the adoption of the testing framework as key concepts, such as test automation tools or assertion libraries, are widely used and well documented. Furthermore, we consider that focusing on OSS improves the future maintainability.

2 Background

The System Under Test (SUT) for which the testing framework was initially developed is a service-based application that consists of multiple backend services and frontend user interfaces [1, 3]. The backend services are

deployed using principles of containerization and orchestration. The frontend user interfaces provide interactive sophisticated interfaces to these services. The testing framework aims at testing and verifying backend and frontend functionality and thus has to offer service and user interface testing capabilities.

2.1 Key Concepts and Terminology

Below, we introduce what we consider as the most relevant concepts and terminology:

- Requirement Definitions
  This is a formal set of documents that define the requirements for the system that is to be developed. In our case, the requirements are maintained in DOORS as primary source and documents are generated from this. For the test framework presented in this paper, we import requirement definitions from DOORS. However, we are confident that other requirement management mechanisms can also be supported in the future if this may be needed.
- System Under Test (SUT)
  The System Under Test (SUT) is the system to be tested and verified. For the work presented in this paper, the SUT is a service-based application that consists of multiple backend services and multiple frontend Human Machine Interface (HMI) user interface applications. For component and system tests, the actual SUT may be the entire system with backend services and frontend HMIs or subsets of these.
- SUT Execution Environment
  This is the environment in which the SUT is executed. For backend services, this is a Kubernetes cluster. For frontend HMIs, this is a desktop environment.
- Test Framework
  The test framework provides a framework for implementing tests and for easing testing and verification of the SUT. In addition, the test framework provides tools for easing the development of tests, their execution, and document generation. The test framework, its driving needs, design, implementation, and lessons learned are the main topic of this paper.
- Test Plan
  A test plan is a concrete implementation of tests for testing a SUT. The test plan is implemented using the test framework. Note, a test plan as considered in this paper is for testing and verification on component or system level. There may be additional tests on unit level that aim more on supporting the actual development instead of verification.
- Test Plan Execution Environment
  This is the environment for executing the test plan. This can be different than the environment for running the SUT, e.g., for testing backend services, the services are run in a Kubernetes cluster and the test plan is executed from outside of the cluster.

2.2 Stakeholders

For the work presented in this paper, the key stakeholders involved in the software project process include, e.g.:

- Customer Technical Team
  The customer technical team is located at the customer. It is responsible for, e.g.,
  - the interaction with end-users of the software on the customer side,
  - the definition of technical requirements in coordination with the end-users of the software.
  - the acceptance verification of the software delivered by the software development team. I.e., the customer team has to verify that the content of a delivered software increment fulfils the expected requirements.
  - The customer technical team may also be responsible for rolling out the software into test or production systems.
- Contractor Software Development Team
  The software development team is responsible for, e.g.:
  - the actual software design and development. The software development team can be located, e.g., at a contractor that develops the software for a customer. Within the software development team, roles such as project manager, software developer, or quality assurance engineer may be present.
  - development of test procedures for the verification of the software,
  - production of documents associated with the software.

In different phases of the project lifecycle, different sub-teams may be responsible for different parts, e.g., the initial requirements definition and the subsequent verification on the customer side or the software development and the test development on the contractor side may be done by disjoint sub-teams. For the sake of simplicity, we will not differentiate between sub-teams but only refer to the overall teams.

3 Motivation and Goals

Motivations for the development and adoption of the testing framework are:

- ease the work of and the communication between the involved teams (contractor and customer),
- increase the degree of automation for running tests,
- increase automation of document generation and reduce manual effort for document production.

Furthermore, because of the complexity of the software, it is difficult to keep all aspects presently in mind. For this reason, requirements, their traceability to test items, the verification of test items and requirements, etc. need to be broken down into manageable units.

From the context etc. in which the presented work is situated, the following goals for the testing framework were derived:

- Integrated Support across the Verification Process
  Individual OSS tools could be largely re-used, e.g., for test artefact implementation, execution, or results processing. However, these tools only cover isolated demands and do not provide an integrated framework. Our framework combines, integrates, and amends these tools. The aim is to ease the process and workflow across the verification process, covering, e.g., development, execution, debugging, or review of test artefacts and document generation.
- Hierarchical Structure of Test Artefacts
  The software system in which scope the verification was done consists of multiple backend services and frontend HMIs. Requirements are structured hierarchically to ease the navigation. Consequently, the test plan shall also be hierarchically organized to ease management and navigation.
- Support for Automatic and Manual Execution of Test Artefacts
  Ideally, as many test artefacts as possible shall be executed automatically. Automatic test artefact execution supports, e.g., running the test plan in Continuous Integration (CI).
  However, it may not be possible to automate all test artefacts. Thus, it shall also be possible to include test artefacts that need to be run manually within the overall test plan to provide a centralized source.
- Tooling for Development, Execution, and Debugging of Automated Test Artefacts
  For easing development, execution, and debugging corresponding tooling is used.
- Document Generation for
  - Test Plan Definitions
  - Test Results
  - Documents shall support the review process of test artefacts and the verification process of the software under test.
- Linking of Requirements to Test Artefacts
  Linking of requirements to test artefacts supports the assessment of the completeness and correctness of implemented test artefacts and the interpretation of test results.
- Annotation of Test Artefacts with Documentation
  Documentation shall support the understandability and of the test plan.
- Import from Requirement Management Software such as DOORS
  In our project, requirements are maintained in DOORS. Importing the requirements enables treating them as integrated items. This enables, e.g., source code completion and highlighting showing details of requirements or including requirement details in the document generation.
- Test Campaign Execution Support (Manual and Automatic)
  The test plan may be executed fully automatically, e.g., as part of CI or semi-automatically, e.g., during Factory Acceptance Test (FAT) campaign. During FAT, it may be needed to re-execute different subsets of test artefacts or to gather logs etc. from test executions.
- Support for Backend Services and Frontend HMI Testing
- Parametric Test Definitions
  To enable better re-use of tests, e.g., across deployments (FAT vs. OSAT) or for applying generic tests to multiple specialized SUTs, the test framework shall provide parametric tests that allow configuration of test parameters via configuration files.

- Remote HMI Test Execution
  It shall be possible to run HMI tests in an execution environment different from the test execution environment. Use cases for this are, e.g., to execute tests on a testing desktop machine and the HMI within the intended production environment or to run the tested HMI as a different user (either on the same host or a different host).
- Selective Test Execution
  - By Tree Browser
    A tree browser allows easily navigating a complex hierarchical test plan implementation.
  - By Pattern Matching
    Pattern matching of test artefact ids allows easy execution of selective test items
  - By User Input
    It shall be possible to selectively skip test elements.
- Slow-motion HMI Test Execution
  For easing the understandability of test procedures and for debugging, it is desirable that it is possible to execute HMI tests in "slow motion". This is contrast to the normal HMI test execution, which typically happens too fast to analyze the exact actions. Slow motion execution will slow down the automated HMI actions to make it easier to follow the automated HMI execution.
- Screenshots for Test Result Documents
  Taking screenshots of HMI applications during test execution and including these screenshots in the generated documentation improves understandability of generated documents and aids in analyzing discrepancies.
- Output Document Production
  Document generation for documents like test procedures or test results is required to produce documents that can be used in formal review processes.
  - HTML
    HTML representations can be useful for easing the navigation of documents, e.g., via hyperlinks and for online viewing or pre-view of content.
  - Word Document Generation
    Word documents are typically used as final versions in the formal review processes.

4 Design and Implementation

4.1 Process and Interaction between Involved Stakeholders and Artefacts

Figure 1 shows an overview of the software development, testing, and verification process and the involved stakeholders, artefacts, and their interactions. The numbering on the arrows showing the interactions roughly indicate their chronological order. It is possible that subsets of the interactions are executed repeatedly, e.g., step 11, verification of test procedures, may lead to feedback from the customer team to the development team for re-working the test plan and going through another review before executing tests.

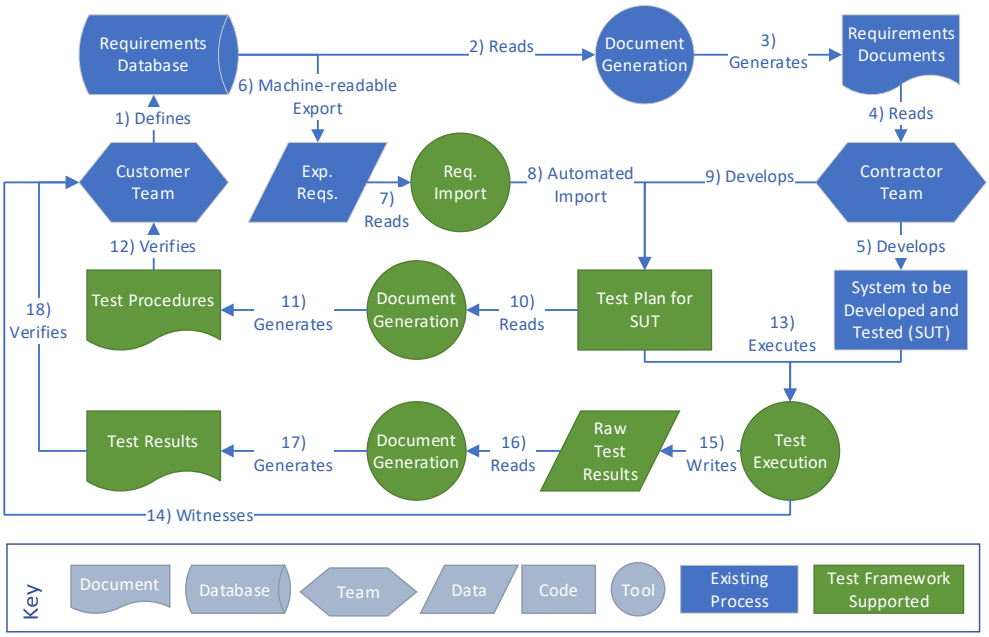

Figure 1 Software Development, Testing, and Verification Process Interactions Overview

Below, the interactions shown in Figure 1 are further detailed:

1. The customer team defines the requirements in a requirements management database. For the definition of requirements, it is also possible that the contractor team contributes. However, typically, the main contributor is the customer team. For the project for which the test framework was initially developed, requirements are managed with DOORS [4].
2. From the requirements database, requirements are read
3. and requirement documents are produced.
4. The requirement documents form the initial input to the contractor development team.
5. Based on the requirement documents, the software/system is designed and developed.
6. From the requirements database, a machine-readable export is produced.
7. The machine-readable requirements export is read by the import tool
8. and automatically imported into a test plan implementation.
9. The contractor team develops the test plan.
   The customer team may also contribute to the test plan development. However, for the initial development, typically, the main contributor to the test plan development is the contractor team. In later phases, the contributions from the customer team may increase. For the sake of conciseness, this is not shown in the figure.
10. The test plan is read by a document generation tool.
11. The document generation tool generates a document of test procedures from the test plan.
12. The document is reviewed by the customer team.
    Optionally, feedback can be provided to the contractor team and the test plan can be further refined and the changes reviewed again. This loop can be performed repeatedly as needed.
13. When the test procedures and test plan are sufficiently verified, the SUT and the test plan are executed.
14. For formal milestones such as Factory Acceptance Test (FAT), the execution is witnessed by the customer team.
15. From the execution, the raw results are written.
16. The raw test results are read by a document generation tool.
17. The document generation tool produces a document version of the test results.
18. The test results are reviewed by the customer team.

The test framework and the test plan are software deliverables that are delivered along with the software that is to be developed. This way, the test plan can be executed as needed, e.g., first for Factory Acceptance Test (FAT) and later for On-site Acceptance Testing (OSAT) or regression testing during maintenance.

4.2 Test Framework Design

Figure 2 shows an overview of the test framework design. The aim is to show the main elements of the test framework, how they interact, and to indicate the re-use of Open Source Software (OSS).

The figure follows the same color coding as Figure 1. In addition, it defines grey to indicate OSS elements and yellow to indicate elements that are specific for the SUT. Gray to green shading is used to indicate OSS elements that were patched.

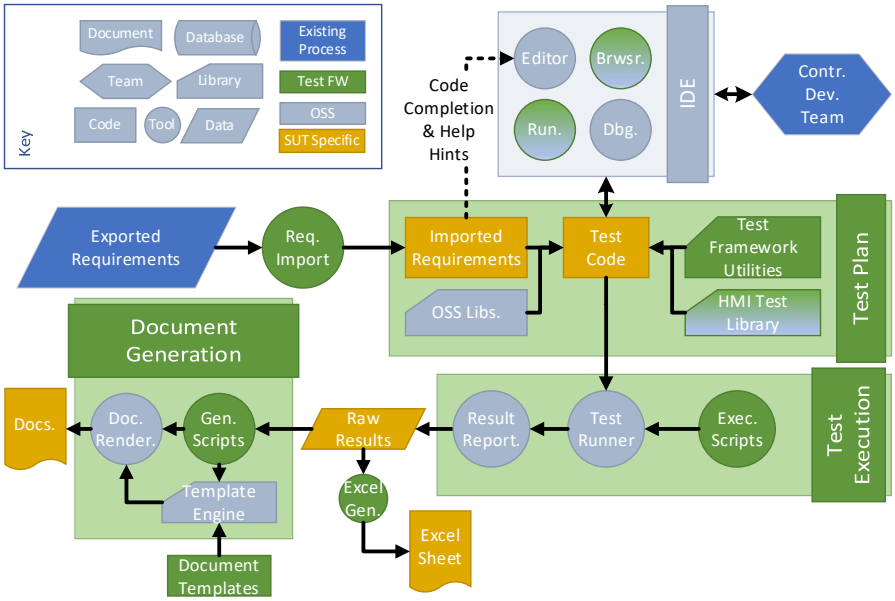

Figure 2 Test Framework Design Overview

The starting point for a testing and verification campaign are the requirements. The test framework provides a tool for importing requirements into the internal format.

The central element when testing a SUT is the test plan. Thus, the test framework libraries and tooling also put the test plan at the center. The test plan consists of:

- Imported Requirements
- Test Code Implementation
- Test Framework Utilities
- Optionally, a patched library for HMI testing (Playwright).
- Optionally, further OSS or custom libraries.

As graphical tool for test plan implementation, execution, browsing, and debugging, an Integrated Development Environment (IDE) is used (VSCode [5] plus patched versions of VSCode Test Explorer [6] and VSCode Test Explorer Mocha Adapter [7] plug-ins). The imported requirements are used to provide code completion and help hints in the IDE.

While a test plan or parts of it can be executed via the IDE, the main execution mechanism for document generation is via the command line. For this the test framework provides execution scripts that call the test runner (Mocha [8]) and reporter (Mochawesome [9]) to generate the raw result output.

From test execution raw results, test procedures and test results documents are generated via the document generation script. This uses a set of scripts provided as part of the test framework, which leverage a templating engine (Handlebars [10]) and document rendering tool (Pandoc [11]). In case of HMI tests using screenshots, the screenshots are also considered part of the raw test results.

In addition to generating the above documents, it is also possible to generate a Microsoft Excel spreadsheet from the test results. This spreadsheet provides a pre-filled tree view of the test plan with additional fields for taking notes. This Microsoft Excel sheet is intended for supporting note taking during the test execution.

4.2.1 Open Source Software Re-use and Custom Tool Implementations

Our testing framework is heavily based on OSS. Our aim was to re-use existing solutions as much as possible. Below, a more detailed view of the off-the shelf software items used in the testing framework is given:

- The testing framework is based on JavaScript using the nodejs/npm [12, 13] ecosystem
  Automated tests are implemented in JavaScript. In addition, the support tools we developed are also implemented in JavaScript using libraries and tools from the nodejs ecosystem. The aim is to provide a self-contained consistent framework.
- Mocha [8] is used as automated testing framework.
  - We developed a light-weight approach for hierarchically structuring the test plan into test artefacts at different abstraction levels (e.g., test suite, test case, test procedure, test step). For this we use existing Mocha functionality which we amend for easily using nested directories and file names for structuring Mocha test artefacts into a hierarchical tree structure that can be easily navigated and updated.
  - We added functionality for annotating test artefacts with human readable documentation using Markdown. The resulting annotations can be used in the document generation for producing rich documents.
  - We added support for keeping the results of multiple executions of subsets of test artefacts. This is required to enable more complex test campaigns while still allowing to auto-generate overall result documents from the collected results.
- Chai [14] is used as assertion library.
  This enables expressing pass/fail criteria using natural-language-like mnemonics, e.g., “expect(test_variable).to.equal(42)”.
- The Mochawesome [9] mocha test reporter is used for raw result generation.
  - We use Mochawesome for storing supplemental information, e.g., requirements associated to test artefacts etc., in the raw test results.
  - Mochawesome includes the test item code in the raw test results, which we leverage for extracting supplemental information such as annotations or test assertions.
- Post-processing Tools for Extracting and Merging Test Results
  - We developed tools for post-processing the Mochawesome raw test results such that they can be used for document generation. The post-processing uses the esprima library as tokenizer for extracting information from the test step source code contained in the raw test results.
- Handlebars [10] is used for document templating.
  - The layout of generated documents if defined via the Handlebars templating library. This eases the design of rich document layouts.
  - Handlebars renders to HTML documents, which can be used as-is or be further processed with Pandoc as mentioned below.
- Pandoc [11] is used for rendering Word documents.
  - The HTML documents generated by Handlebars as describe above, can be rendered to Word documents via Pandoc. Pandoc also supports custom styles for Word documents, e.g., to define heading font settings etc.
- VSCode [5] is used as Integrated Development Environment (IDE) for developing, browsing, running, and debugging the source code of a test plan implementation.
  - The VSCode Test Explorer [6] plug-in is used for navigating and executing test artefacts.
    - We extended the VSCode Test Explorer plug-in to support arbitrary deep nesting of test artefacts. Before, it only supported one nesting level.
  - VSCode Test Explorer Mocha Test Adapter [7]
    - For running tests from the IDE, the Test Explorer Mocha Adapter plug-in is used. We customized the plug-in to extend it to provide the same functionality as for the command line-based test execution, e.g., selective test execution based on user input or parametric test execution.
- DOORS Requirements Import
  - We implemented import of requirements from DOORS [4] into JavaScript. This enables source code highlighting showing details about requirements in the IDE, code completion for requirements, and including requirement details in generated documents.
- Microsoft Excel Sheet Generation from Test Artefacts

  - We developed a tool for generating Microsoft Excel sheets from test artefacts for supporting manual test campaigns. The generate Microsoft Excel sheet provides a view on the hierarchical test plan with pre-filled elements and placeholders to ease taking notes in a systematic way during (semi-)manual test campaigns.
- HMI Testing via Playwright [15]
  - HMI tests for Electron-based applications can be implemented in playwright.
  - HMI tests support screenshots. The test framework provides utility functions for taking screenshots such that screenshots will be automatically included in the auto-generated documents.
- Test Plan Versioning via Git
  - Our tooling supports Git elements (selected tags and commit ids) for handling different versions of test plan implementations.

In additions to the additions to the respective OSS parts mentioned above, further contributions provided by our framework are, e.g., selection of matching OSS items, integration of OSS items into an overall framework, designing the framework for re-using OSS, enhancing of OSS elements for enabling the required functionality, and addition of new implementations for automating common tasks.

4.3 Test Plan Hierarchy

The SUT we consider for testing is a complex system. To make it easier to navigate a test plan targeting the SUT, the test plan is hierarchically structure. Figure 3shows an overview of the different hierarchy structure levels and how they are used for building the test plan as a tree of these artefacts. The root of the tree is implicit.

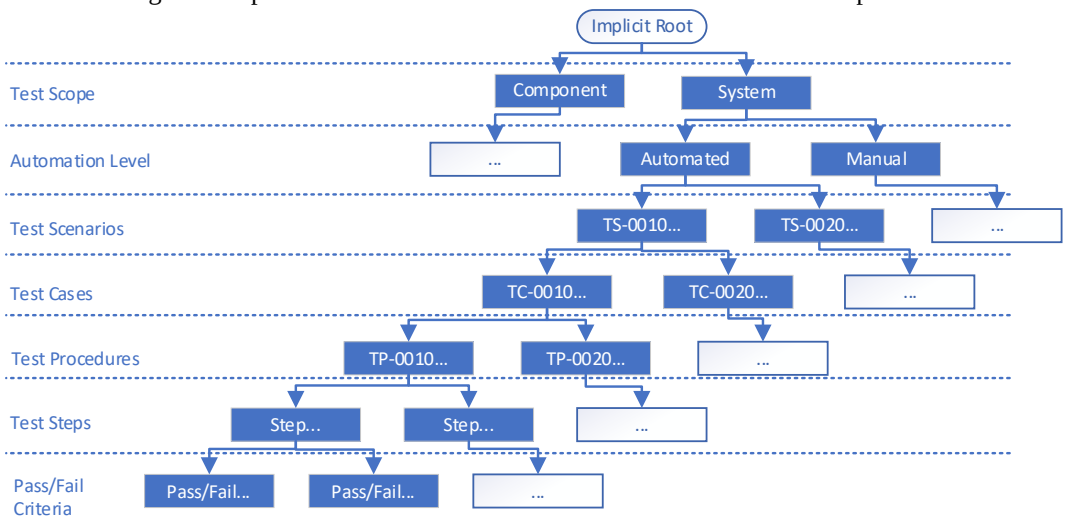

Figure 3 Test Plan Structure Overview

For making the navigation of the test plan tree easier, we use the convention of assigning numerical identifiers to test scenarios, test cases, and test procedures. At each level, a numeric id is expected to be unique. Reasons for this approach are:

- It has shown that the numerical ids can serve as convenient shorthand for identifying tests in the test plan, especially when navigating large test plan implementations.
- The lexicographic order corresponds to the hierarchical order based on the numbering.
- Related artefacts on the same level can be grouped with common id prefixes, e.g., TS-01xx and TS-02yy.

In addition to the numerical id part, a human readable text description can be appended. It is suggested to also include a human readable part to further support navigation and readability of test plan implementations.

4.4 Test Plan Implementation Structure

The structure of the test plan implementation was chosen to facilitate re-use of OSS components while enabling easy development and maintenance of the test plan source code. The most important element here is the hierarchical structure. This structure is manifested at the following levels:

- File System Structure
- Source Code Structure

- Call Level

From the source code structure, Mocha allows nesting of elements as shown below. For the sake of simplicity, import/require statements are omitted.

```
describe(‘First_Level”), function() {
 describe(“Second_Level_1”, function() {
     describe(“Third_Level_a”), function() {
           it(“Test_at_1_a”, function() {
               expect(1).to.equal(1);
           });
       });
       describe(“Third_Level_b”), function() {
           it(“Test_at_1_b”, function() {
               expect(1).to.equal(1);
           });
       });
   });
   describe(“Second_Level_2”, function() {
    describe(“Third_Level_c”), function() {
           it(“Test_at_2_c”, function() {
               expect(1).to.equal(1);
           });
       });
   });
});
```

This will produce the following hierarchical structure: This structure will already work in the IDE tree browser.

- First_Level
  - Second_Level_1
    - Third_Level_a
      - Test_at_1_a
    - Third_Level_b
      - Test_at_1_b
  - Second_Level_2
    - Third_Level_c
      - Test_at_2_c

The problem with this way of expressing a hierarchical structure is that the entire hierarchy is contained in a single source code file. For making the test plan implementation easier navigable and maintainable. It was desired that the structure is also reflected in the file system.

4.4.1 File System Structure

At filesystem level, elements such as test scope, test automation level, test scenarios, and test cases are intended to be directories. Test procedures are intended to be source code files. Test steps and pass/file criteria are intended to be contained within a test procedure source code file.

To ease refactoring and reduce duplication of identifiers, the file system elements (directories and files) are intended to define the id of a corresponding test item at the respective level, e.g., a test scenario is identified via its directory name. This way, it becomes possible to easily move and duplicate test artefacts by re-naming/moving or copying them.

Below, an example for a file system structure is given. For the sake of conciseness, the automation level and test scope are omitted.

- TS-0010_I/
  - test_scenario.js
  - TC-0010_A/
    - test_case.js
    - TP-0010_a.js
    - TP-0020_b.js
  - TC-0020_B/
    - test_case.js
    - TP-0010_x.js
    - TP-0020_y.js
- TS-0020_II
  - test_scenario.js

- …

The example shows the use of unique numerical identifiers Tx-yyyy for each level. For the example, the additional textual identifiers are roman numbers for TS, uppercase letters for TC, and lowercase letters for TP.

Following the convention, TPs are represented by JavaScript files. For items above TPs, a corresponding JavaScript file representing the scope, e.g., test_scenario.js or test_case.js, are present. These files can contain, e.g., documentation strings for the corresponding element at that level or, for TC, the associations to requirements.

4.4.2 Source Code Structure

At the source code level, a structure equivalent to the one given in the example above needs to be resembled. This is required to enable the use with the established tools, e.g., IDE tree browser.

This is achieved by importing the higher-level elements starting from the innermost level. Below, simplified source code examples are given for the first leaf of the file system example shown above (TS-0010_I/TC-0010_A/TP-0010_a) in reverse order starting with the TP:

```
const tc = require("./test_case");
const util = require('test-framework-util/util');
tc.add(util.get_test_procedure_name_from_path(__filename), function () {
  it("test-step", function () {
    expect(1).to.equal(1);
  });
});
```

Below, a simplified example for the test_case.js at the TC level is given:

```
const ts = require('../test_scenario')
const util = require('test-framework/util');
function add(name, f) {
  ts.add(util.get_x_name_from_path(__filename), function() {
    describe(name, f);
  });
}
module.exports = {add};
```

Below, a simplified example for the test_scenario.js at the TS level is given:

```
const automation_level = require('../test_automation');
function add(name, f) {
    automation_level.add(util.get_x_name_from_path(__filename), function() {
      describe(name, f);
  });
}
module.exports = {add};
```

4.4.3 Call Level

During test execution or evaluation, e.g., for displaying the test tree browser in the IDE, the call to these files starts with the innermost file. From there, the code evaluation follows the require statements going to increasingly outer layers. At each layer, the calls are nested such that in the end, the required nested structure is created.

4.5 Information Extraction and Annotations for Document Generation

One aim of the testing framework is to reduce duplication of information and to unify the sources of information required for testing and verification. Therefore, the testing framework approach aims on extracting as much information as possible for document generation from source code artefacts. Mechanisms to do this are:

- Test artefact names are, where possible, derived from their file system level identifiers as single source of truth.
- The pass/fail assertion library was selected to offer “human-readable” mnemonic notations.
- The document generation approach extract pass/fail assertion statements from the test code and includes these excerpts in the generated documents.
- As good practice to leverage the utility of the assertion library mnemonics, we suggest to check against human readable variable names, e.g., instead of
  expect(x).to.equal(123);
  we suggest to use
  expect(parameterCount).to.equal(123);
  even in otherwise short test snippets as this information will be included in the auto-generated documents and can support their understanding.

- The assertion libraries expect statements can take optional strings to explain their intent, which can also be used for enriching the auto-generated documentation.
- References to requirements are established at the source code level of test cases by including the imported requirements as JavaScript module and adding them with a utility function that supports the document generation.
- For including screenshots of HMI tests as information, an integrated approach is provided that eases the definition of screenshot executions and automatically includes them in generated documents.

Wherever it is not possible to avoid duplication, the testing framework aims on working this around. One example for this is the automation of the import of requirements from requirement database, which is the single source of truth for requirements.

During the use of the testing framework, it showed that for improving the information exchange, additional information, that cannot be derived from source code artefacts, is needed. This can be seen as a kind of duplication as the test automation implementation and the additional information need to be kept inline.

This is a problem that we consider as similar to source code documentation. Thus, we chose to use a similar approach to source code annotation approaches. Additional documentation that shall be included in auto-generated documents can be added at the different test artefact levels with annotations such as "/*DOC … DOC/*" for generic test artefact documentation or "/*EXPECT … EXPECT*/" for documentation related to pass/fail criteria expect statements.

These annotations support text formatting using markdown syntax. This way, formatting can be expressed in a way that makes it easily readable in both the source code and the auto-generated documents.

These annotations imply a certain degree of duplication. Annotations provide supplemental information to the test code and need to be maintained together with the test code. However, we consider that by using annotations that are included in the source code, the maintenance of code and documentation is eased as both are located very close to each other.

5 Development

We aimed on re-using as many OSS solutions as possible. However, for some parts, we had to develop custom solutions. In addition, in some cases, established solutions had to be extended or customized. In this section, we describe which parts we considered necessary to add and to customize and how we approached this customization.

5.1 Custom Tools

5.1.1 Document Generation

We developed custom tools to convert the test reports from the raw mochawesome JSON format into the test results HTML and Word documents. This involves several post-processing stages, including

- Raw results are merged into one JSON file. There may be individual result files from several runs of subsets of the test plan. The results are merged into one file, if a test procedure has been run more than once then the most recent result is taken.
- Text annotations are extracted from the test code, descriptions of the test procedures, test steps and pass/fail criteria.
- Any screenshots created during the tests are collected and linked to their associated test steps.
- The version of the tests is set. This is done using git to compare the test code to previous tagged versions in the repository.
- A new JSON results file is created containing this extra information.
- We use handlebars to populate a HTML template from the JSON file. The document contains test procedure overviews, detailed results, screenshots.
- We use Pandoc to convert the HTML document into a Microsoft Word document

For the document generation, as shown in Figure 1, we consider the generation of the following documents:

- Test Procedures
  The test procedures document provides a human readable document view on a test plan implementation. It is intended for review before an official test campaign, e.g., FAT, is run.
- Test Results
  The test results document provides a human readable document view on the results of executing a test campaign.

The above process can largely be applied to both documents. The difference between the documents is mainly that in comparison to the Test Results document the Test Procedures document does not contain any results and screenshots.

However, the Test Procedures document generation still requires a test execution result JSON as input. For this, we use what we call a "pseudo" test execution, which runs tests but with a intentionally short timeout to make all tests fail quickly. This way, the JSON output can be generated faster than when running all tests. However, in principle, also a full test result JSON can be used for generating the Test Procedures. Aside from this, the document generation is largely similar except that different Handlebars templates are used for the different documents.

5.1.2 DOORS Requirements Import

In our context, the main source of truth for requirements is the DOORS application for managing requirements in a database. To link requirements to test artefacts and to provide convenience functionality such as code completion for requirement identifiers and help display for requirements, we import the DOORS requirements into an internal format.

The internal format stores requirements as JavaScript objects in a dedicated module and exports the objects. In addition, help strings are added for each object. This enables the above features.

For importing requirements from DOORS to the testing framework, first requirements need to be exported from DOORS in Tab Separated Value (TSV) format. Using TSV showed to be easier handleable than Comma Separate Value (CSV).

The exported TSV file is intended to be stored in the test plan git repository. It is imported automatically when installing dependencies of the test plan via a custom JavaScript script that is called through in the pre-installation phase of npm install.

5.1.3 Microsoft Excel Spreadsheet Generation

We produce a Microsoft Excel spreadsheet containing all of the test steps with space to add comments. This is intended to be used to manually record notes during a test run. We created a JavaScript script which uses the sheetjs node module to produce a Microsoft Excel document. The information is populated from the same test results JSON output used to generate the Microsoft Word document.

5.2 Extension to Open Source Software

In some cases, the of the shelf software we use does not quite fulfil our needs, so we customized it ourselves. Where possible we created a wrapper for the existing package, avoiding forking the original project, because this creates extra effort in maintaining the software and staying up to date with new versions. However, in some cases this was not possible so we created a custom patch.

5.2.1 Mocha Runner

In order to provide custom extensions to the test plan without requiring changes to the test plan, we created a wrapper script which makes calls to the Mocha API instead of invoking Mocha directly. This allows us to override Mocha functions on the fly without having to patch Mocha.

We override the Mocha describe function in order to inject a custom "before" functions, which will be called at the start of every test suite as part of the execution mode implementation. Setting this function in the wrapper script instead of adding it manually to each test suite in the test plan means that no changes are required to an existing test plan to get it working with the test framework.

5.2.2 VScode Test Explorer

The VSCode Test Explorer was not able to handle the structure of our test plans because of the multiple nesting levels of test artefacts. The stock implementation of the Test Explorer provides a "merge suites" option but it is only able to merge one level of nesting. So, we created a patch which enables Test Explorer to handle an arbitrary number of nesting levels.

5.2.3 VScode Mocha Test Adapter

The VSCode Mocha Test Adapter is the entry point to Mocha when running the tests from the VSCode Test Explorer. The same changes needed to be applied here as in the Mocha runner, but it is not possible in this case to write a wrapper because the Mocha API is hidden inside the VSCode extension with no way to access from the outside. Therefore, in this case we forked the code and patched it directly.

5.2.4 Playwright

We use playwright for testing electron-based HMI applications. Playwright for electron is in an experimental stage of development and is lacking some of the features already built into playwright for chromium. We want the possibility to connect to applications remotely, and adjust the execution speed of HMI actions, both of which are possible in playwright for chromium but not yet available for electron. We provide a patched version of playwright, which opens up these features for electron. For both types of application playwright uses Chrome DevTools Protocol (CDP) as the interface, so we use the same mechanism as chromium to implement these features for electron.

5.3 Lessons Learned

5.3.1 Extension of Open Source Software

We aimed on re-using as much Open Source Software (OSS) as possible. However, in some cases, the available OSS had to be modified or extended for providing the intended functionality. For this extension, we considered the options below. As criteria to selecting which approach to use, we considered maintainability and ease of implementation.

- Wrapper
  With a wrapper, we implement code "around" the original OSS project. This way, the OSS dependency can be used unchanged. We consider that this makes it easier to follow upstream version updates of the OSS project as we mainly rely on public API that should be somewhat stable.
- Fork/Patch
  In some cases, we considered it easier to fork/patch an OSS project. Reasons for choosing to fork/patch was, e.g., that we needed to use private API to reach the intended functionality. We consider this as a trade-off between ease of implementation of the missing functionality and other benefits like tighter coupling to the OSS dependency and possibly making it more difficult to port the fork/patch to updated versions of the OSS.

We chose to extend OSS using a wrapper whenever possible rather than creating a fork. Only when using a wrapper was not easily possible, we chose to fork/patch the existing OSS project.

When making changes to OSS code, we have so far created forks of the original project and have not pushed the changes upstream. We have found this is simpler in getting started because pushing upstream would require more polishing of the code and would require justification for the changes in order for them to be accepted. In future it would be beneficial if we are able to push these changes upstream, because they would then be maintained as part of the baseline of the OSS, taking some of that maintenance work off us.

5.3.2 Usage Experience

The testing framework was developed in parallel to the development and verification of the System Under Test (SUT). Because of this, we could gather experience regarding the usage of the testing framework during its development and quickly include feedback into the implementations of the testing framework and the test plans using it. We consider the following usage experiences relevant:

- Integrated Development Environment (IDE) Support
  The IDE is the main tool for developing tests based on the testing framework. To support this development, the following capabilities showed to be very helpful. These features were enabled by importing the requirement definitions into a code-readable format and adding metadata, e.g., auto-generated documentation strings, to them.
  - Source Code Completion and Check
    Code completion makes it easier to enter requirement identifiers. Furthermore, the included source code check highlights misspelled requirement identifiers, which reduces errors during the implementation.
  - Code Hints
    Based on the requirements imported as code with associated metadata, the IDE can display code hints. These hints include the full text of the requirements and additional supplemental information. This allows reading requirement details in the IDE as single source of information without having to switch between different documents.
- Test Structure and Browsing

The SUT required complex test implementations and campaigns. The hierarchical structure of the test implementation and the integrated support for browsing it in the IDE via the tree view in the Test Explorer GUI made the navigation of the tests easier.

- Selective Test Execution
  Selective test execution enables running a subset of tests from the overall test implementation. The following aspects were considered helpful:
  - Test Explorer GUI
    Through the tree view in the Test Explorer GUI, tests can be executed selectively. This was considered helpful during test development and debugging but also during official test campaigns as this made it easier to explain, discuss, and reason about test implementations, e.g., regarding if the test implementation fully verifies the set of requirements associated with it.
  - Pattern Matching on the Command Line
    Pattern matching equally provided a efficient tool for selective test execution, which was similarly used during development and test campaigns.
- Display of Test Errors
  - Display of Results in the Tree View
    The tree view displays results. This makes it easy to navigate to elements that failed. Through this view, the problematic parts can be opened and further analyzed.
  - Inline Display of Errors
    The IDE test integration displays errors of pass/fail criteria assertions inline in the source code. This makes it easier to identify problems and to fix them.
- Document Generation Feedback
  - Understandability of Test Logic
    We started with primarily relying on the mnemonics offered by the assertion library for documenting the intent of the test logic. It showed that this often was not enough to efficiently communicate the intent of test implementations. To improve this, we added support for the “EXPECT” annotations that enable adding more human readable content to pass/fail criteria implementations.
  - Understandability of Comments and Annotations
    Initially, comments and annotations were aimed on being reduced in favor of relying more on code. It showed that this was often not enough to communicate the logic of tests. To improve this, the annotations supported by the test framework were reworked and the test implementations were updated as well.
- Automated vs. Manual Tests
  We initially aimed on automating as much as possible. However, it showed that some test cases required significant effort for implementing them in an automated way. Furthermore, in these cases, it showed that the test code was often convoluted. This made it more difficult to establish a common understanding across teams about the test logic and how it verifies the associated requirements. In these cases, we found that it was often easier to use manual tests instead. Overall, our lesson learned here is that the decision whether using automated or manual tests should be done judiciously on a case-by-case basis.

6 Summary and Conclusion

In this paper, we present a framework for testing and verification of complex software systems in a formal process. We discuss the formal process and how the framework integrates with it. We explained the demands of this process and how the design and implementation of the testing framework and test plans implemented with the framework address these demands.

Key factors for our design are the aim on re-using as much Open Source Software (OSS) as possible. For this, we chose the testing approach and test plan structure etc. to enable good re-use of existing OSS. However, some additions had to be custom developed to link the different OSS solutions and some OSS solutions had to be customized.

The resulting testing framework was and is still actively used in production as the sole testing and verification tool for the corresponding System Under Test (SUT). As the testing framework implementation happened in parallel to the SUT development, lessons learned etc. could immediately be taken into account and flown into further enhancements of the testing framework.

The testing framework is actively used by contractor and customer teams. Our experiences indicate that the testing framework helps in handling the big testing scope of the complex SUT with several hundreds of requirements.

References

[1] F. Croce, C. Miranda, R. Gad, A. Simonic, Concepts and Implementation of Generic Services and HMI Elements, 17th International Conference on Space Operations (SpaceOps 2023), Dubai, United Arab Emirates, 2023, 6 - 10 March.
[2] ESA Requirements and Standards Division, ECSS-E-ST-40C – Space Engineering Software, European Cooperation for Space Standardization (ECSS), 2009-03-06, https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/, (accessed 2023-01-27).
[3] F. Croce, C. Miranda, R. Gad, A. Simonic, Layered Service Oriented Design for M&C Applications, 16th International Conference on Space Operations (SpaceOps 2021), Cape Town, South Africa, 2021, 3 - 5 May.
[4] IBM, IBM Engineering Requirements Management DOORS Family, https://www.ibm.com/products/requirements-management, (accessed 2023-01-27).
[5] Microsoft, Visual Studio Code, https://code.visualstudio.com/, (accessed 2023-01-27).
[6] H. Benl, Test Explorer for Visual Studio Code, https://github.com/hbenl/vscode-test-explorer, (accessed 2023-01-27).
[7] H. Benl, Mocha Test Explorer for Visual Studio Code, https://github.com/hbenl/vscode-mocha-test-adapter, (accessed 2023-01-27).
[8] The OpenJS Foundation, Mocha, https://mochajs.org/, (accessed 2023-01-27).
[9] A. Gruber, mochawesome, https://github.com/adamgruber/mochawesome, (accessed 2023-01-27).
[10] Y. Katz, handlebars - Minimal templating on steroids, https://handlebarsjs.com/, (accessed 2023-01-27).
[11] Pandoc Contributors, Pandoc - a universal document converter, https://pandoc.org/index.html, (accessed 2023-01-27).
[12] The OpenJS Foundation, Node.js, https://nodejs.org/en/, (accessed 2023-01-27).
[13] npm Inc., npm, https://www.npmjs.com/, (accessed 2023-01-27).
[14] Chai Contributors, Chai Assertion Library, https://www.chaijs.com/, (accessed 2023-01-27).
[15] Microsoft, Playwright, https://playwright.dev/, (accessed 2023-01-27).