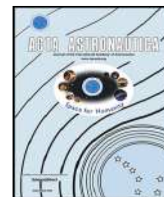
Acta Astronautica 211 (2023) 574–591

Contents lists available at ScienceDirect

Acta Astronautica

journal homepage: www.elsevier.com/locate/actaastro



Research paper

Model-driven engineering for low-code ground support equipment configuration and automatic test procedures definition ☆

Aaron Montalvo *, Óscar R. Polo, Pablo Parra, Alberto Carrasco, Antonio da Silva, Agustín Martínez, Sebastián Sánchez

Space Research Group, Universidad de Alcalá, Alcalá de Henares, Madrid, Spain

A R T I C L E I N F O

Keywords:
Ground support equipment
Low-code
System-level validation
Hierarchical checking process

A B S T R A C T

Space domain systems must go through different types of ground testing. For system-level black-box functional testing, ground support equipment is built ad hoc, customized for each different mission. Building any of this specific equipment involves considerable engineering effort that can be diluted using techniques that allow reusing configurations of the test setup environment without recodification. These techniques can be merged into a hierarchical checking process based on independent filters to compare the received outputs with the expected ones. The hierarchical checking allows the separate definition of several layers or levels of validation, from the lowest protocol packet levels to the logical abstractions at the highest application level. This paper introduces a solution based on Model-Driven Engineering for Low-Code Ground Support Equipment. It uses the abovementioned mechanisms to reduce the effort to develop and customize ground support systems for different missions. This solution is integrated within an environment called MASSIVA, which allows the automatic configuration and definition of system-level test procedures. This environment has been used in the software verification and validation process of the Instrument Control Unit of the Energetic Particle Detector on board Solar Orbiter.

1. Introduction

Verification and validation processes of space systems require exhaustive ground testing using customized equipment called Ground Support Equipment (GSE). Typically, the GSE consists of specific hardware with a harness compatible with the device under test and specialized software called Ground Support Software (GSS). This software controls the hardware, monitors the system activities, and may execute validation procedures, generating reports that are used for system verification. In order to carry out these functions, the GSS usually runs in a dedicated machine, part of the GSE's hardware. Since the GSS is used in the development, debugging, verification, and validation processes, it is advantageous that it can support different options and configurations depending on the test scenario used in each case. Thus, GSS can be used for both verification and validation testing, i.e., to verify, on the one hand, that the tested system is valid and, on the other hand, that the tests have been successfully performed and all requirements have been verified.

The model philosophy of space systems [1], as well as the different simulation and fault injection environments used to complete the validation of flight software, has led to an increase in the number of deployment alternatives to be managed by the GSS. In addition, the current trend for space engineering teams to develop multi-mission platforms [2] requires the recurring use of hardware and software between missions. This approach can benefit from test environments that systematically reuse and reconfigure tests associated with the product line.

Model-Driven Engineering (MDE) [3] is a discipline that employs models as the basic unit of systems development. Models store and represent relevant information about abstractions or concepts inherent to the problem domain to be solved. They are defined using a meta-model, i.e., a model specifically defined to describe other models.

The other fundamental element of the MDE methodology is transformations. Starting from at least one model, they allow obtaining different products such as source code, input files for analysis tools, or other models of different levels of abstraction. If a given transformation has as a product another model, it is called a Model-to-Model (M2M) transformation. In addition, if the transformation generates a text file, such as a code file, it is a Model-to-Text (M2T) transformation.

Nowadays, there are many tools and frameworks that allow us to define models and transformations. One of the most widely used technologies is the Eclipse Modeling Framework (EMF), integrated within

☆ This work has been supported by Spanish Ministerio de Economía y Competitividad under the grant ESP2017-88436-R.

* Corresponding author.
E-mail address: aaron.montalvo@uah.es (A. Montalvo).

https://doi.org/10.1016/j.actaastro.2023.06.027
Received 15 March 2023; Received in revised form 12 May 2023; Accepted 15 June 2023
Available online 26 June 2023
0094-5765/© 2023 The Author(s). Published by Elsevier Ltd on behalf of IAA. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

the Eclipse Development Environment (Eclipse IDE). EMF defines its meta-model, Ecore, and integrates different tools to perform M2M and M2T transformations. Furthermore, it also includes Xtext, a toolset that can be used to define domain-specific languages from the different models and generate textual editors that can be added seamlessly to the Eclipse IDE.

All software development processes are living processes that can vary over time. The implementation of test procedures that require software validation is usually based on descriptions and definitions taken from requirement documents and other applicable and reference documents that detail the interfaces and formats involved. If these documents and references change, the test procedures shall change, and those changes are usually done manually by the software V&V team with the support of the developers. At worst, if any part of the test is hard-coded in the GSS, any minor modification would require updating and modifying the GSS itself. In summary, project changes involve activities that drastically affect the V&V process. These changes, moreover, are prone to occur as space developments are complex. Sometimes, due to unforeseen circumstances, these changes are even made at later stages of the project.

The Space Research Group of the University of Alcalá (SRG-UAH) has developed the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) instrument of the Solar Orbiter mission of the European Space Agency (ESA), and NASA [4]. Within this project's scope, a fully configurable Ground Support Software called MASSIVA was conceived. MASSIVA has been designed for the verification and validation process of the operational procedures and is capable of performing all test procedures in any PC with the proper testing equipment, i.e., hardware devices and drivers. It uses plain XML files for configuration, which are human-readable and easily editable.

This approach allows working with multiple test scenarios, which made it possible to address the validation of the ICU software against different EPD models. Four different scenarios were defined for validating the EPD's ICU, three for an EBB (Elegant Breadboard) model, and the last for an Electrical and Qualification Model (EQM), making all test procedures fully testable on dedicated computers at the UAH facilities in Alcalá. As MASSIVA only relies on available operating system drivers, it can also be used on any computer that has compatibility with the test equipment, making any computer a potential GSE. In addition to all this, MASSIVA also has several useful tools for test development and manual testing.

General settings are contained in an XML configuration file, and every different test procedure is configured within its own XML test procedure file. Every scenario used within the test campaign must also be configured using its own XML scenario configuration file. Making changes and fixing bugs is easier using this approach when compared to hard-coded tests in closed systems. Besides, the same test can be performed in different scenarios by keeping the test procedure file and changing only the XML scenario configuration file.

Testing operations are generally defined as a sequence of steps containing inputs and outputs, which are telecommands to be sent and expected telemetries. As expected outputs are defined in their own XML test procedure file, the verification process can be performed autonomously by GSS if required. However, sometimes when dealing with concurrent data sources such as sensors, some operations are not performed sequentially but concurrently. GSS must also be aware of concurrent operations for verifying them autonomously, and GSS must be able to alternate concurrent with sequential steps.

All those configuration files can be created manually and were done so at the beginning of the V&V process. Subsequently, a Model-Driven approach compliant with ECSS-E-ST-E40 standard was adopted, defining models corresponding to the software specifications and requirements at the system level, along with the models that support the detailed description of the validation procedures that fulfill the requirements, including the TM/TC (Telemetry/Telecommand) format definitions. Then, through a set of model-to-text transformations, the GSS was integrated so that all the information provided by the models was used to automatically configure the GSS and perform the system-level test campaigns according to the procedures defined within the models. Finally, the test logs reported by the GSS are also processed to integrate their data into the corresponding verification models. A detailed description of this model-driven approach can be found in [5].

The case study of this work comprises the black-box functional tests of the verification and validation process of the on-board software of the EPD's ICU. Several projects were developed from that one, and all of them were used to test the versatility of MASSIVA for low-code reconfiguration and reuse, oriented to product line development, demonstrating that it is feasible to generate the correct reports and checks automatically by using this tool.

The rest of the paper is organized as follows. Section 2 covers the related works, followed by Section 3 describing the verification and validation process as defined in the ECSS-E-ST-40 standard. Section 4 describes the MASSIVA tool, focusing on XML files, mainly configuration and test procedures. Later, Section 5 describes the models created for supporting MASSIVA features. Next, Section 6 describes the proposed model-based test procedures and their relationship to MASSIVA in the context of the Verification and Validation processes. Then Section 7 summarizes the proposed case study. Finally, Section 8 contains the conclusions.

2. Related works

Several tools developed for space missions' verification and validation processes provide support at the test and integration phases. They usually use scripting languages for creating the command sequences, and they should have a database connection for the definition of the telecommands and telemetries.

An example of these tools is GSEOS [6], a software package designed to be used in any space mission's integration and tests phase. It was originally devised for NASA's MESSENGER mission development in 2003. The application suite built from the GSEOS software package runs on Microsoft Windows. It has a modular architecture, so new components can be easily added, such as new hardware drivers or connections to databases. It uses the Python language to create scripts that can be used for improving processes.

Another software tool is CMDVS (Control, Monitoring, Data processing, and Visualization Software) [7], a software package produced by Celestia-STS (previously known as SSBV Aerospace & Technology) for the construction of GSEs. It also runs on Microsoft Windows and is modular, with different GUIs for monitoring the embedded system under testing. For scripting, it uses Tcl/Tk along with TOPE, and it is compatible with SCOS-2000 database format MIB, containing all telecommands and telemetries in CCSDS/ECSS defined formats, along with the implementation of PUS services. SCOS-2000 [8] is the generic Mission Control System (MCS) of the ESA, which must be tailored for each mission, and MIB (Mission Information Base) is the format used by this MCS, so this software tool is full-compatible with ESA's database.

TERMA is a company in The Netherlands that has developed different solutions for the space domain. One of their products is called TSC, an SCOE (Specific Check-Out Equipment) created for the development of instruments or other subsystems. For more extensive activities, at the spacecraft or system level, they offer CCS5, a multi-user Central Check-out System (CCS) [9]. Both share several features, such as the sequence language called uTOPE, which is a reimplementation of the original TOPE, based in turn on Tcl, which is used in SCOS-2000, and the use of SCOS-2000 database format MIB. Another system called VOSSCA [10] is also based in CCS5 and uses TOPE. Still, it adds some automation modules and a GUI based on a web application for facilitating the operation.

Similarly, the German Space Operations Center (GSOC), the mission control center of DLR (Deutsches Zentrum für Luft- und Raumfahrt/German Aerospace Center), has developed its own MCS, called GECCOS

(GSOC Enhanced Command- and Control System for Operating Spacecrafts) [11], based in SCOS-2000 Release 3.1, and compatible with MIB database. GECCOS supports several different spacecraft and satellite platforms. It is not an MCS limited only to the operation phase but also a CCS that can be used in the AIT (Assembly, Integration, and Test) phase.

Another tool for software validation is MaTeLo [12], developed by Alena Spazio starting in 2002. It has a different approach, as the testing process is based on Statistical Usage Testing (SUT) using Markov chains. It was designed with a Use Case model instead of test sequences like the other products, but it can also be used for defining a test campaign, focusing on automation as the final goal.

Apart from specific tools, embedded software is a prevalent topic nowadays. The survey carried out in [13] performed a systematic literature review. Their results showed, for example, that since 2001 the number of papers related to testing methods and techniques has increased almost exponentially, and the ones regarding test tools and platforms also have experienced a significant increase. Also, the number of papers proposing a solution has increased, and most articles cover real (not simulated) systems under test. Most of the papers belong to the automotive and industrial auto business, with aviation and space in third place. Finally, among the most used words in the titles, they found expected terms such as *automotive* and *real-time*. Still, there are ones like *generation*, *automated* and *automatic*, and most prominently *model-based* that show the future trends on this topic.

Another study [14] is not focused on reviewing papers but on directly interviewing embedded software professionals. Specifically, they pay attention to the information flow in software testing, asking for challenges and improvement approaches. Among the challenges, they talk about feedback, root cause, and traceability, and when talking about approaches, they cite test report automation and visualization of results, among others.

Other review studies focus on model-based testing and requirements -based tools [15,16]. This first one classifies the tools depending on the kinds of modeling languages used, such as state-based, transition-based, or stochastic models. The other approach can be seen as a guide for creating system models in a model-based space system context. The focus this time is on three different elements: the modeling language, including semantics; the methodology for obtaining data specification; and the Conceptual Data Model (CDM) for formalizing the system itself, finally proposing the SCDML (Semantic Conceptual Data Modeling Language).

In line with these model-based solutions, a test process model solution was proposed for testing embedded software [17]. This model is based on a V-Model diagram, and it goes from user requirements to final acceptance testing, describing all the steps needed in the proposed process.

Among the solutions that use modeling instead of coding test definitions, even a framework based on XML markup language was proposed, called TSVF (Testing Specification Validation Framework) [18]. It was used to validate the software of the Meteosat Third Generation (MTG) satellite on-Board computer. The architecture of TSVF consists of the core and the drivers. The core performs the test cases while the drivers communicate the different elements in the test scenario.

Back in 2010, [19] studied the future of space programs, proposing virtualization for the future GSEs. This approach uses blade servers with a virtualized testing environment for easier maintainability and higher availability, and thin desktop clients connected remotely.

Apart from solutions and studies used in the space domain nowadays, embedded systems are present in other engineering domains and share several features related to testing and verification. Thus [20] is focused on the automotive industry and the ECU (Electronic Control Units) and proposes CANoe+ for generating and executing test procedures. This tool is based on CANoe, but it includes randomness by using GraphWalker for generating random test sequences.

MCAPI (Multicore Communications API) [21] standard can be used not as a process or a simple model but as a software solution, which also includes a GUI and uses serial COM ports for communication.

Finally, another solution is Ball Aerospace COSMOS [22]. It is an open-source suite of several applications for developing and testing embedded systems. It can be used to show real-time telemetry display and graphing and to create and send commands. However, all the testing functionality relies not on scripted procedures or application configuration files. Dealing with telemetries and telecommands as a whole, it lacks the support of a multi-level approach that could be easily integrated into a MDE environment. Besides, using scripts instead of textual languages makes it difficult for COSMOS to generate automatic file generation from validation sources.

Our solution is an all-in-one tool called MASSIVA (Monitoring and Analysis System for Software Inspection, Verification/Validation, and Assessment) that can perform any test campaign on different setups with different levels of log reporting information automatically. It uses models for all the components needed for the campaign: procedures, steps, telecommands, and telemetries. All of them can be generated automatically from the Functional Test document and manually edited, as all are human-readable XML files. Recodification can be avoided using reusable XML files, as it is only needed to select the desired configuration and procedure to be launched once it has been created.

Apart from the test campaign performing and log reporting, MASSIVA offers several options for manually debugging or performing different operations on any defined setup, such as loading the software, showing the values from the housekeeping data in real-time (including the calculation of values with mathematical formulas when available like currents or voltages). All the options can use all the available interfaces and be edited on the fly, resulting in a robust but extensible software tool.

3. ECCS-E-ST-40 software verification and validation process

Fig. 1(a) shows the normal development of the software verification and validation (V&V) process following the ECSS-E-ST-40C [23] and ECSS-Q-ST-80C Rev.1 [24] standards. Every different document has to be defined by various experts with different roles.

The software development process starts at the leftmost top when the Requirements Engineer writes or compiles the SSS (Software System Specification). Based on this document, the Software Architect writes the SRS (Software Requirement Specification) and defines the software architecture as a part of the SDD (Software Design Document) or as a separate document named ADD (Architecture Design Document). After that, the SDD is completed by the Software Developer, which can also be done through a separate document called the Detailed Design Document (DDD), and the software coding starts. In parallel to this specification, design, and coding process, the software V&V process is carried out at different levels.

Working at the lowest levels, the Unit & Integration Tests Designer writes the Software Unit Integration Test Plan (SUITP) to be compliant with the SDD. At the upper level, and to validate the requirements defined in the SRS, the Functional Tests Designer writes the SVS (Software Validation Specification), where the system-level software validation procedures are defined. Based on this document, the Functional Tests Developer carries out the Test Campaign, implementing these procedures on a defined setup by the Functional Tests Conductor. This same Functional Tests Conductor annotates the Log reports, basically a checklist containing the results of all the tests after the Test Campaign.

Finally, using the traceability between the SSS specifications and SRS requirements, together with evidence both from the log reports and from the analysis, inspection, and demonstration reports, the Product Assurance Engineer writes the SVR (Software Verification Report) containing all the matrices that prove the software has been verified and validated.

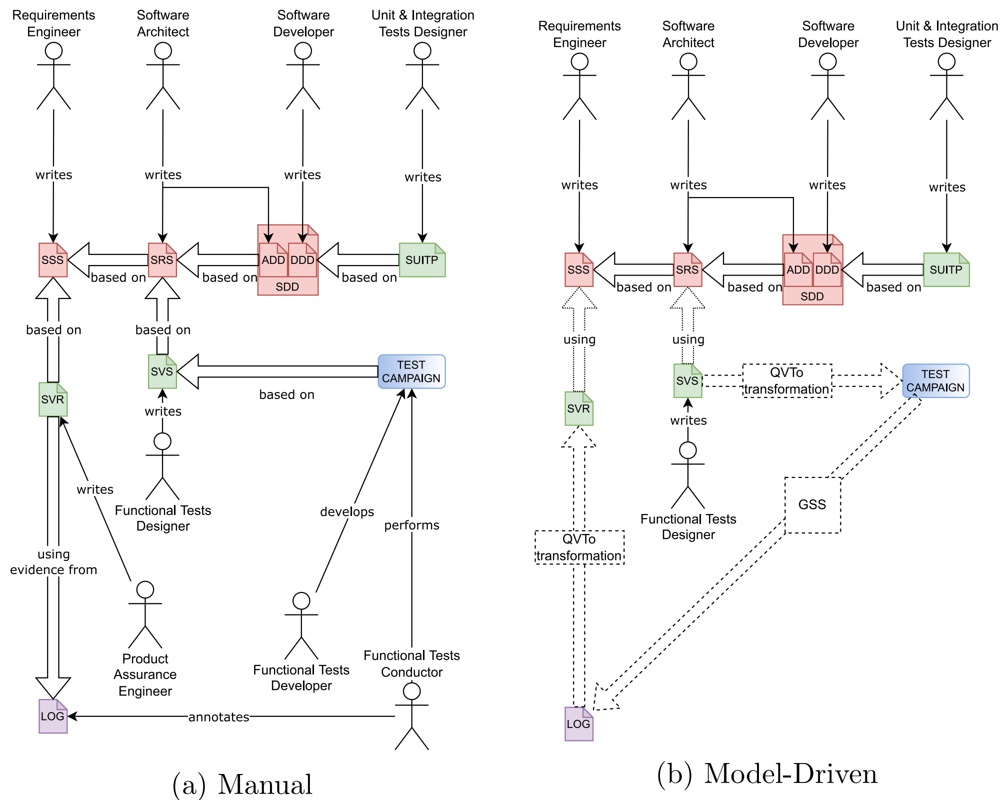


Fig. 1. Verification and validation process in ECSS standards.

Our approach reduces the intervention of human actors, as can be seen in Fig. 1(b). First of all, the SSS, SRS, SVS, and SVR are not documents but model instances that will be transformed into the final deliverable documents in Office Open XML (OOXML) format [25], and for this reason, are shown with a dotted instead of a solid line. The SDD and SUITP are not system-level documents and, thus, are outside the scope of this paper and remain represented as documents and not models in the current version of the proposed workflow. In comparison to the traditional process, the Model Driven approach reduces the effort to maintain consistency between the documents that are part of the process, avoiding direct editing by the different human actors.

All the models have been implemented using the Ecore meta-model defined in the Eclipse Modeling Framework (EMF) [26]. Then they are instanced using textual editors generated by Xtext [27].

Then the Test Campaign is generated through a QVTo transformation and launched using a GSS (Ground Support Software), which can perform the validation tests in the given format and automatically return the Log document. This Log document is transformed using QVTo again, together with information from the SSS instance, to create the final SVR.

This scheme also motivates reusing the test procedures and set-ups which have already been defined. The different test procedures for the test campaign are written for the current software developments. Still, as their structure is based on different levels, every piece of a designed test campaign can be used at another one.

4. MASSIVA technical features and capabilities

MASSIVA (Monitoring and Analysis System for Software Inspection, Verification/Validation, and Assessment) is a software tool designed to perform test campaigns defined on different scenarios using a single piece of software that can be deployed on multiple operating systems. MASSIVA is based on a low-code approach that relies on human-editable XML files. These input files allow modeling and configuring the test scenarios and defining the procedures that make up the black-box validation tests in a hierarchical form that fits with the multilevel data used during the operation of the on-board software [28]. The XML files are also suitable to be automatically generated, so the whole approach enables the tool's integration within a Model-Driven Engineering (MDE) workflow that facilitates the automatization of the procedures and the reuse at multiple levels of the different input models.

MASSIVA was initially designed to validate and verify the on-board software of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) in the Solar Orbiter mission, which was launched in February 2020.

The original idea for the creation of this environment arose from previous experiences in our research group when developing the GSE (Ground Support Equipment) for Nanosat −01 [29] and Nanosat −1B [30]. The main objective was to replace scripting programming languages with a markup language that would allow the definition of test procedures with a syntax similar to that of the ECSS test procedure specification. If test procedure definition models are created from the standard using MDE techniques, then the test procedure definition files can be generated automatically by instantiating those models.

In addition, the main objective was to avoid the system rebuilding every time new test procedures or new data monitors were added to the test campaign. On the other hand, it was proposed that the data validations were performed hierarchically, thus treating independently the different lower communication levels and their physical protocols and the higher application levels which define the data logic structure. This way, various communication protocols could be used while reusing the same application logic, and the different data structures could be used with the same physical ports.

For each project, each of the available options can be fixed or left open for future dynamic configuration at runtime. In the latter case, using human-readable languages, such as XML, to perform the final assignment of values is common.

By combining these ideas, the final approach could be used without recompiling for different testing phases, campaigns, or projects. Any

Table 1
MASSIVA components reusability.

Component	Reusability	How to reuse
Debug information verbosity	Yes	low-code edition (XML)
List of interfaces required	Yes	low-code edition (XML)
Interface configuration	Yes	File reference
Interface control	No	–
Plot chart design	Yes	File reference
Test campaign monitors and global variables	Yes	low-code edition (XML)
Special packet configuration	Yes	low-code edition (XML)
Test campaign procedures list	Yes	low-code edition (XML)
Test procedures definition	Yes	File reference
TM/TC formats and filters	Yes	File reference

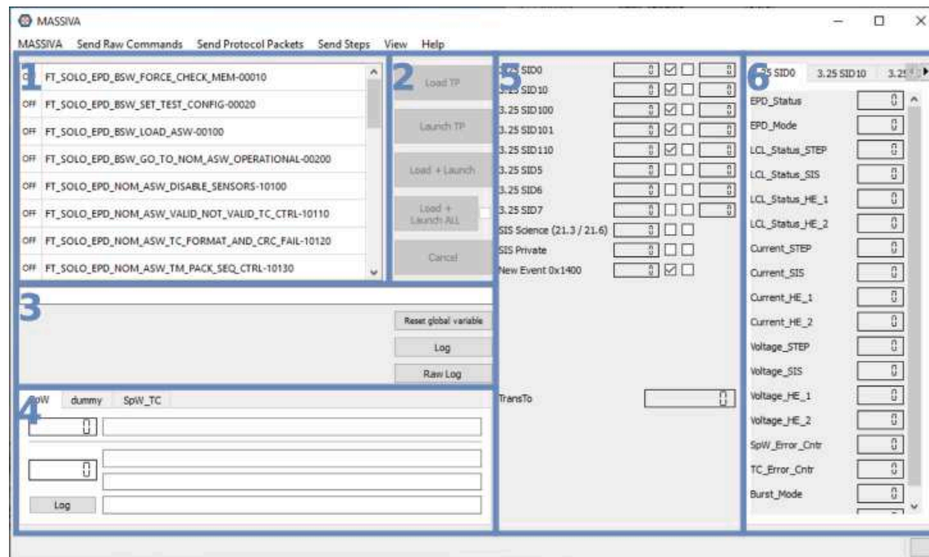


Fig. 2. Main MASSIVA GUI.

configuration or procedure has to be configured only the first time it is being used, and then any other setup can reuse it.

Besides, existent ECSS documented standards lead to defining a textual language where procedures can be defined similarly to the standard structure, using steps, inputs, and outputs. This way, a domain expert can define and edit procedures without the knowledge of any scripting language used in other similar tools.

MASSIVA has been developed to have a flexible structure, as it is intended to be used in different scenarios, models, and projects. There are also several useful features to use along with tests, which are helpful for test developing and debugging, but they are not compulsory.

The main goal is to promote reuse in the testing process. Once a test campaign has been defined, any of its component parts can be reused in future developments without recoding. For example, test procedures and interface configurations are defined in different files than those of the primary test campaign. Thus, they can be individually reused directly in any test campaign. In addition, any file already developed can be easily edited, or automatically generated, since all configuration files are written in XML. This low-code approach allows anybody to be the test conductor, as nothing needs to be known about the software under test. Even another machine can launch any test campaign, as shown below. Table 1 shows the different components and how they can be reused.

4.1. Graphic user interface

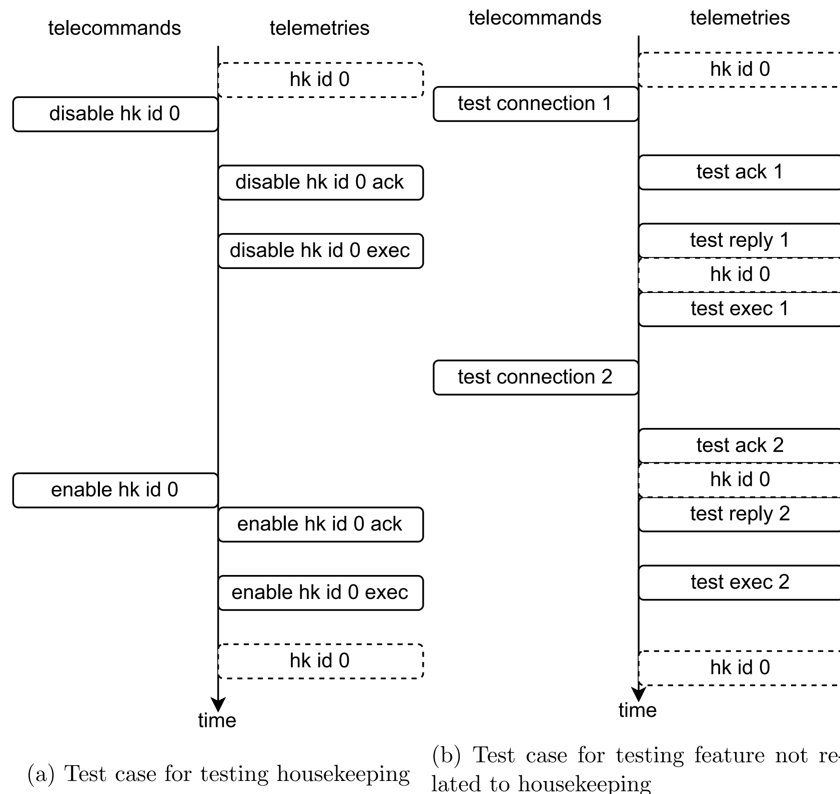
As most visualization features in the MASSIVA GUI (Graphic User Interface) are configurable, the displayed information can vary from project to project by customizing the configuration files.

The most important areas of MASSIVA main GUI have been numbered in Fig. 2:

1. This area shows all the test procedures defined in the main MASSIVA configuration file. When there are too many procedures, scrolling is automatically enabled.
2. This area contains all the buttons helpful in loading and launching tests, along with menu options.
3. This area contains the name of the test procedure currently loaded and, when loading or launching a procedure, the current number of steps and inputs/outputs.
4. This area shows the last packet sent from GSS and the last three packets received. The numbers on the left are counters for packets sent and received.
5. This is the Special Packets area information. Special Packets are periodically received packets filtered for special treatment, typically housekeeping and scientific data packets with no relevant testing information. They will be explained later in Section 4.1.1.
6. Finally, this other area is also related to Special Packets. There is an option in the configuration file for displaying the content of any packet payload if the packet structure is known.

4.1.1. Special packets

Special Packets are an important feature of MASSIVA, as they are essential in the test automatization process. In every software made for the space domain, several telemetries usually are sent periodically with information about the space system. Most of these telemetries are housekeeping packets with data from all the subsystems. In scientific missions, there are also scientific packets with meaningful data for the



(a) Test case for testing housekeeping

(b) Test case for testing feature not related to housekeeping

Fig. 3. Special packets in test flow.

mission's scientists. But in both cases, this information has to be taken into account by the test designer.

All housekeeping and scientific telemetries must be identified, and test cases and procedures for their validation must be defined. However, in other test scopes or when testing other features, the test procedure flow may be interrupted by these housekeeping or scientific packets since, in many cases, they are received asynchronously (i.e., not in response to a telecommand). Since MASSIVA processes all received packets, the test will fail if any of the housekeeping or scientific data is received when another response packet is expected in the test flow. Even so, the target software may have worked correctly. Fig. 3(a) shows a simple test procedure flow that disables and re-enables an item of housekeeping telemetry, the so-called "hk id 0". In this test procedure, it should be verified that housekeeping packets are not received when disabled, so they will not be configured as special packets. Next, Fig. 3(b) shows another test procedure that only performs 2 test connections, expecting a "test ack", "test reply" and "test exec" telemetry for each "test connection" telecommand. In this procedure, any "hk id 0" telemetry received will cause the test procedure flow to fail since it is not an expected telemetry response for "test connection", so it must be configured as a Special Packet.

To circumvent this undesired behavior, the packets that do not belong to the test flow and thus are not expected to be received must be taken out of the test flow. And this is what Special Packets are helpful for.

When configuring the Scenario (see Section 5.1), one of the optional features which can be configured is "Special Packets". When a Special Packet is configured, it will be shown in the main MASSIVA as aforementioned. There is also a counter for each Special Packet, and the time since the last one was received is also shown. All the Special Packets are not written in the main test campaign log report to keep the log reports as simple as possible.

Sometimes a given telemetry must be configured as a Special Packet in most tests in the test campaign, but there may be some tests for checking this particular telemetry. For this reason, all the pre-configured Special Packets can be disabled or can have their information written back in the test campaign log report. Both options can be enabled/disabled using the two checkboxes in the interface or in the test procedure definition, as shown in Section 5.3.

Finally, there is another area in the MASSIVA interface for Special Packets. As already told, these packets contain information about the system that may not be useful for testing but interesting to be shown. Because of that, when configuring a Special Packet, MASSIVA can be configured to display the content of its different fields if the packet has a well-known format.

4.1.2. *Dynamic interface*

Most of the functions are fully configurable via XML configuration files. At least one Main Interface must be configured; including several basic protocol options, but apart from that, any other options are not mandatory. Fig. 4(a) shows a minimally configured MASSIVA, without several of the areas described above, without several menu options, and without tests in the list of test procedures. This minimal configuration is a valuable scenario for testing communications at the most superficial level. All packets received from the corrected device will be shown (and written in the log report file). Commands can always be sent via the "Send Raw Commands" menu using plain text files containing the commands.

The configuration can be as flexible as necessary. The most straightforward setup for a test campaign includes only a list of test procedures. Adding some "Protocol Packets" for debugging is very useful but is not mandatory for the campaign itself. For more information about Protocol Packets, see Section 4.2.2. Fig. 4(b) shows a MASSIVA instance configured with only tests and a Protocol Packets configuration file, with no Special Packets and no monitoring.

As mentioned in the previous subsection, the Special Packets feature is a configuration attribute to display information about packets that do not belong to the test flow but may contain useful information.

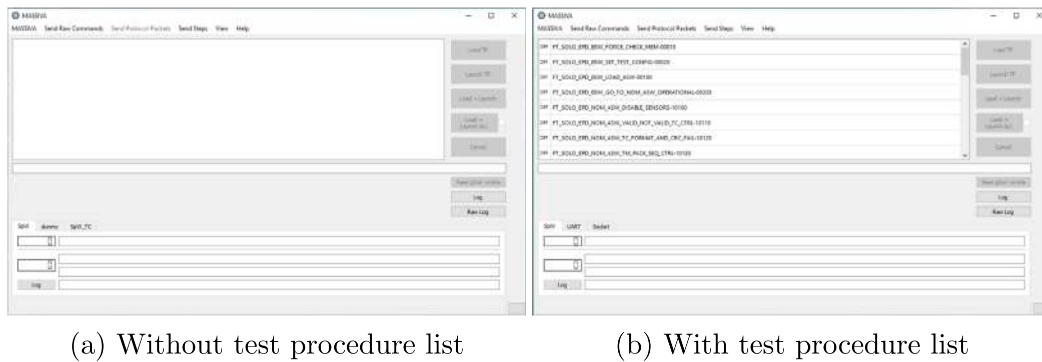

(a) Without test procedure list (b) With test procedure list

Fig. 4. Main MASSIVA GUI basic configuration.

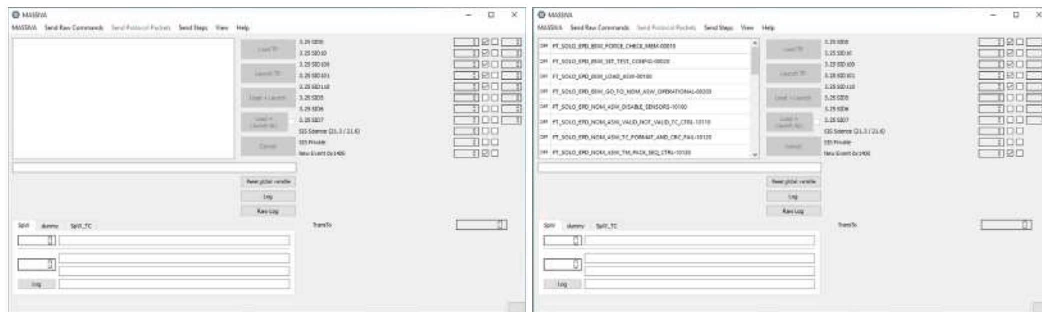

(a) Without test procedure list (b) With test procedure list

Fig. 5. Main MASSIVA GUI with special packets.

Housekeeping telemetries are an excellent candidate to be configured as Special Packets.

The test procedures list can be configured independently from the Special Packets feature, as they are separate components in the MASSIVA configuration architecture. Both MASSIVA instances shown in Fig. 5 have Special Packets configured (on the right side). Fig. 5(a) shows MASSIVA without any test procedure, and Fig. 5(b) shows both features configured.

Finally, the last area mentioned above is also related to Special Packets. For any Special Packet defined in the configuration file, you can select its packet structure to be displayed. Like the main Special Packets feature, this option can be configured without any list of test procedures. If several packet structures are configured to be displayed, each will be in a different tab.

4.2. Features for development and debugging

Several other features are included in MASSIVA for test development and debugging, namely:

- Sending packets from an external file that contains data in hexadecimal format.
- Sending any predefined telecommand manually directly from the interface.
- Sending predefined steps autonomously. A test procedure consists of steps with inputs and outputs (see Section 5.3).
- Periodic sending of any telecommand automatically.
- Live monitoring and telemetry filtering, including plots.
- Command-line options.

All these features will be summarized in this section.

4.2.1. Raw commands

MASSIVA has several functions to send/command the test target manually. This is a handy feature to develop and debug the tests themselves and to more easily control the flow of the tests in case any manual action needs to be performed.

The first sending option is to send a command in raw format, i.e., a list of hexadecimal values. As shown in Fig. 6(a), the menu will display all available interfaces, and clicking on any one will display the “Send raw command” dialog, shown in Fig. 6(b).

Input files can contain not only a single command but any number of commands, one per line. The interval between commands can be selected when there is more than one. In addition, there is a “Send in loop” option to send the telecommands repeatedly in a loop.

4.2.2. Protocol packets

The second sending option uses the available predefined TM/TC format files to create a list of formatted packets that can be sent manually. Each configured packet will be available in the “Send Protocol Packets” menu, as shown in Fig. 7(a).

When the selected Protocol Packet has any fields, the dialog shown in Fig. 7(b) will be displayed. This dialog allows the user to set the values of the fields and supports variable packet formats.

4.2.3. Predefined steps

The last sending option of MASSIVA allows sending not a single command but a complete test procedure step, which includes inputs or telecommands and may include outputs or telemetries. For a full explanation of MASSIVA test procedures, see Section 5.3. There are two different options for sending packets from the “Send Steps” menu, as shown in Fig. 8.

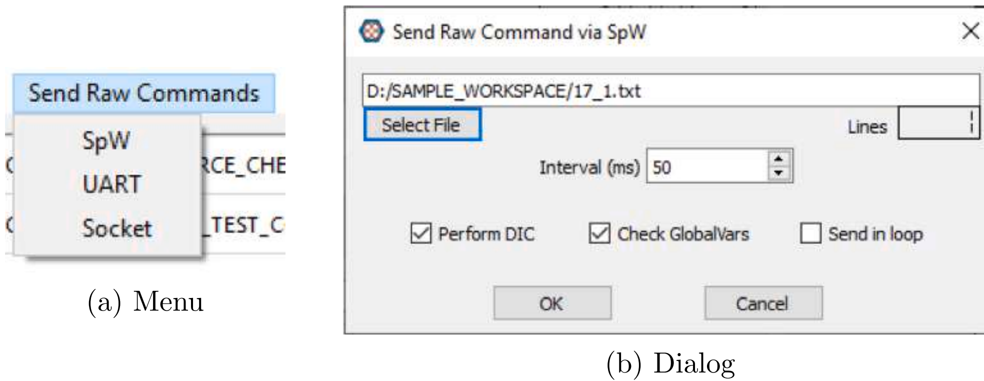


Fig. 6. MASSIVA send raw commands.

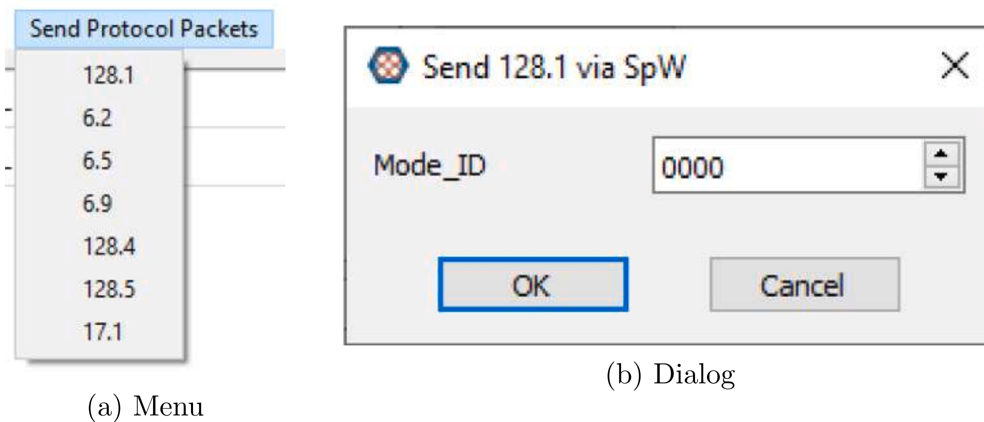


Fig. 7. MASSIVA Send protocol packets.

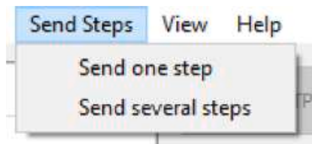


Fig. 8. MASSIVA send steps menu.

The first option, "Send one step", prompts the user to select an XML file containing a step in the MASSIVA step format. The second option, "Send multiple steps", takes as a parameter a list with the number of desired steps in the MASSIVA step format, which must be located in the appropriate folder of the MASSIVA workspace. Both options are handy for debugging test procedures, as they allow the test designer to perform all procedures step by step or per block.

4.2.4. Periodic telecommands

The last menu of the MASSIVA main interface is the "View" menu. It contains different options depending on whether the running instance of MASSIVA is SpaceWire compatible or not (see Section 5.2). When it is, this menu will show two SpaceWire time-code options, or SpW time-codes, as shown in 9.

The other two options are always displayed, as seen in Figs. 9 and 9(b), as they are presented regardless of the MASSIVA version. But they can be disabled if they have not been configured correctly. In this case, they will be displayed in the menu with a gray color and will not be clickable, as in 9(c).

The first option in the "View" menu is "Periodic Telecommands", which launches the dialog with the same name. This dialog shows a list of pre-configured telecommands, as shown in 10. Each Periodic Telecommand has the same information. The checkbox next to the name activates or deactivates the associated Periodic Telecommand. The editable number to the right of the checkbox shows the current target period in milliseconds. The other two non-editable number displays will show the number of Telecommands sent and the actual period of the last cycle.

4.2.5. Live monitoring and plots

The other important feature in the "View" menu is called "Plots". When the menu item is clicked, the dialog shown in Fig. 11 is displayed.

This dialog will display the configured packet fields in a two-dimensional scatter plot chat with time as the X-axis and the named value as the Y-axis. The dialog allows multiple values to be displayed on the same chart or separate chart organized in several tabs.

4.2.6. Command-line options

Another essential feature in MASSIVA is neither a menu nor a visible feature. MASSIVA supports command-line arguments, which help place it in a complete development and debugging toolchain. Configuration files can be selected using command line options, which allow multiple configurations to be maintained simultaneously on the same computer without requiring any manual modification or adjustment to the tool.

Another helpful command-line option in MASSIVA is the possibility to run a single test or a whole test campaign from the command line. This way, MASSIVA can be easily integrated with other tools and verification and validation workflows, thus helping automatize the different processes.

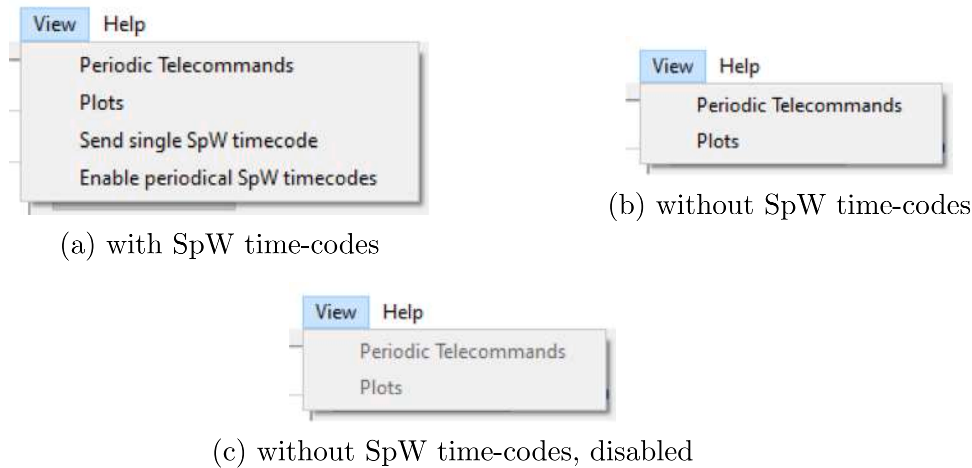


Fig. 9. MASSIVA view menu. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

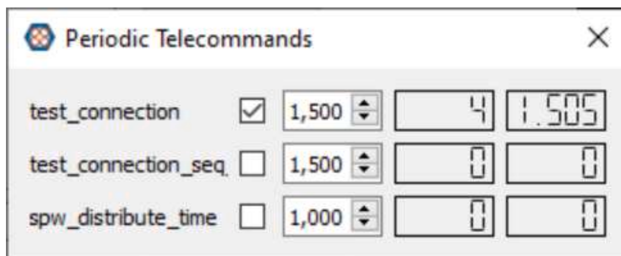


Fig. 10. MASSIVA periodic telecommands dialog.

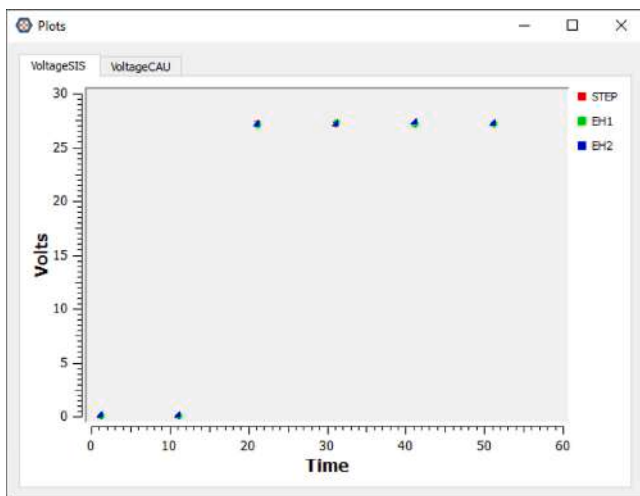


Fig. 11. MASSIVA plots dialog.

4.2.7. Port interfaces

This subsection will only deal with the port interfaces available at MASSIVA. For the models of the interfaces, see Section 5.2. The currently available interfaces in MASSIVA are:

- Serial port, using a Universal Asynchronous Receiver-Transmitter (UART) and compatible with USB-Serial converters.
- IPv4 Sockets:
  - Server socket, which will wait for clients to be connected
  - Client socket, which can connect to an existing server
- SpaceWire port, using a SpaceWire transceptor. Currently supported devices are several from Star-Dundee: SpaceWire PCI, SpaceWire Brick USB, and SpaceWire Brick Mk2.
  - SpaceWire time-codes port, which can send time-codes independently from the main packet transmission and reception.

Most ports have several configuration options that can be changed quickly, making different logic configurations with the same physical devices easier.

5. MASSIVA models and algorithms

5.1. Configuration models

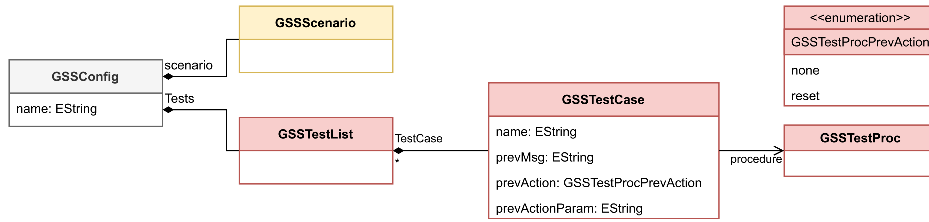
Fig. 12(a) shows the primary MASSIVA configuration meta-model. The main root class of the configuration meta-model is GSSConfig. It contains a scenario, modeled using the class GSSScenario, and a list of tests for the current campaign, modeled using the class GSSTestList. The list contains the different test cases, modeled using the class GSSTestListTestCase. This class includes several attributes: a previous message to display before the start of the test case; a previous action to perform before the beginning of the test case, which is modeled using the enumerated GSSTestProcPrevAction; and an optional parameter for that previous action. It also references the corresponding Test Procedure, which will be modeled using the GSSTestProc class discussed below.

The previous approach is helpful for projects with a single scenario, but for further reusability, another MASSIVA configuration approach uses environments and campaigns. In this other case, shown in Fig. 12(b), the main configuration file is the GSS Campaign, modeled using the class GSSCampaignCampaign, and the scenario is not contained but referenced. This time, an object of class GSSEnvironmentEnvironment contains all the scenarios, thus making the scenario configuration part reusable and keeping the Test List separate.

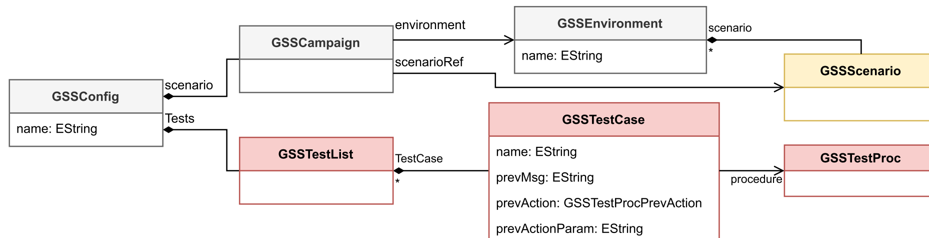
In both approaches, an object of class GSSTestList is used to define the test list, each modeled using class GSSTestListTestCase and containing a name and a reference to its own XML test procedure file. This allows reusing the same procedure files with different names in different scenarios or projects.

The GSSScenario class is the one that defines the configuration for both hardware and port configuration as well as other options. Its meta-model can be seen in Fig. 13.

The only mandatory features for GSS to function correctly are Options, Protocols, and Interfaces, which are modeled respectively



(a) MASSIVA Configuration Model



(b) MASSIVA Campaign Model

Fig. 12. MASSIVA scenario models.

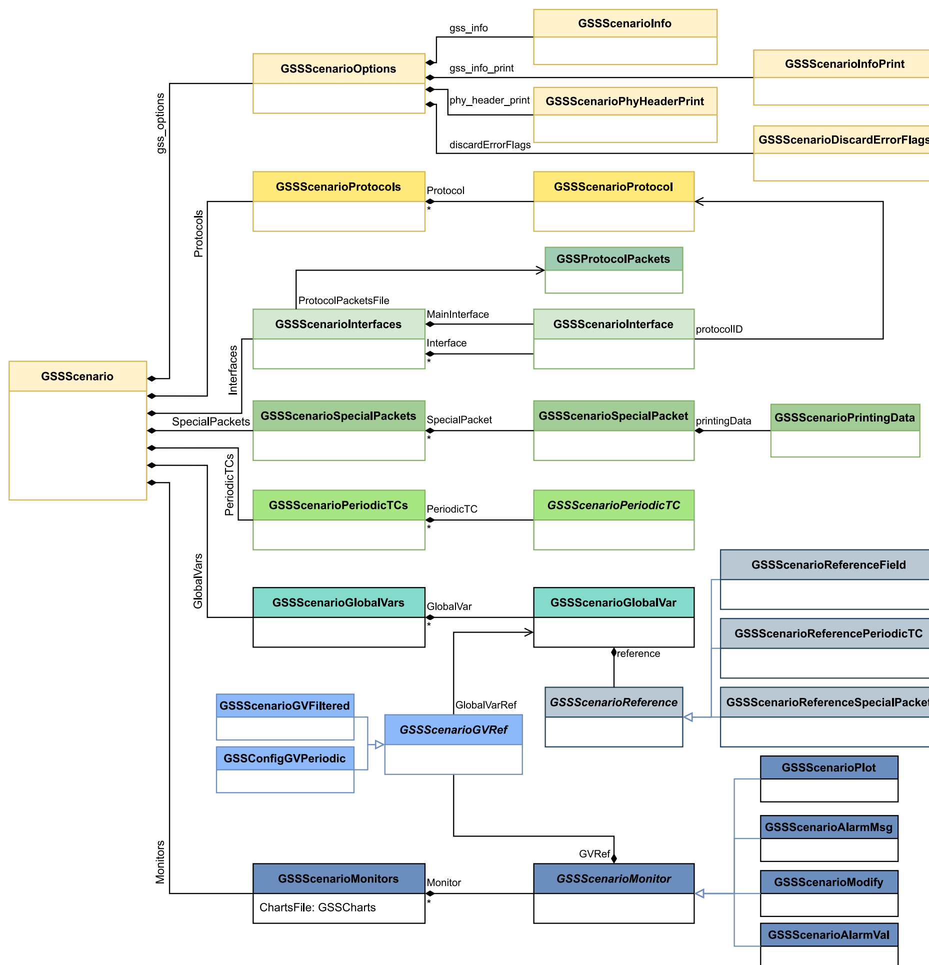


Fig. 13. GSS scenario model.

with the classes GSSScenarioOptions, GSSScenarioProtocols, and GSSScenarioInterfaces.

Both classes GSSScenarioOptions and GSSScenarioProtocols contain objects that model the general features to be configured. The first feature, GSSScenarioOptions, includes options about general information and scenario version, information printing options and debugging; physical port printing options; and scenario error flags configuration; modeled through the classes GSSScenarioInfo, GSSScenarioInfoPrint, GSSScenarioPhyHeaderPrint and GSSScenarioDiscardErrorFlags, respectively. The second general feature, GSSScenarioProtocols, shall define the type and subtype offset of any protocol used in any interface, thus facilitating the identification of TMs and TCs. Each protocol is modeled using GSSScenarioProtocol classes.

The class GSSScenarioInterfaces contains the objects that model the interfaces that MASSIVA must configure (e.g., serial or SpaceWire ports) using the drivers provided by the device's vendor and/or by the operating system and configuring the appropriate parameters for each of the interfaces. There are two types of interfaces, one for the main interface to be used within the configuration and one for the rest, but both are modeled using the class GSSScenarioInterface. The only difference between the main interface and the others is that the main interface is the interface related to Protocol Packets, seen in Section 4.2.2. The class GSSScenarioInterfaces also references an object of class GSSProtocolPackets, which must contain the desired Protocol Packets to be configured. These Protocol Packets must use the protocol defined in the previous feature, but they are not further related.

There are other options related to feature configuration, but they are not mandatory, although they are helpful for different needs of test campaigns. They are modeled using the classes GSSScenarioSpecialPackets, GSSScenarioPeriodicTCs, GSSScenarioGlobalVars and GSSScenarioMonitors.

The first feature allows the test designer to select which received telemetries should be treated differently and thus processed separately from the test telemetry stream. As mentioned above, candidates for these Special Packets (see Section 4.1.1) are those that are received periodically and are not relevant to testing, such as housekeeping information and science data telemetries. Each Special Packet is modeled using the GSSScenarioSpecialPacket class, and the interface display options are configured with the Printing Data option, modeled using the GSSScenarioPrintingData class.

The next non-mandatory feature is Periodic Telecommands. This feature can help distribute time information or schedule a periodic "you are alive" operation. In MASSIVA, a Periodic Telecommand is a telecommand that is sent periodically. These telecommands are modeled using the GSSScenarioPeriodicTC class.

The non-mandatory Global Variables feature is used to define a variable within the MASSIVA scope that can later be used for various purposes. Each variable is modeled using the class GSSScenarioGlobalVar, and must contain a reference modeled using the abstract class GSSScenarioReference. This reference may be a field within a previously configured telecommand or telemetry, modeled using GSSScenarioReferenceField; a reference to a Periodic Telecommand, modeled using GSSScenarioReferencePeriodicTC, or a reference to a Special Packet, modeled using the GSSScenarioReferenceSpecialPacket class.

The last non-compulsory feature is the MASSIVA monitors, which are related to the Global Variables. A monitor can be used for checking or modifying a Global Variable. Each monitor is modeled using the class GSSScenarioMonitor, which must reference a Global Variable using the abstract class GSSScenarioGVRef, which can be concreted in the class GSSScenarioGVFiltered for Global Variables defined on a Field or a Special Packet, or in the class GSSScenarioGVPeriodic for Global Variables defined on a Periodic Telecommand.

Since different types of monitors can be defined, GSSScenarioMonitor is also an abstract class. In short, MASSIVA supports the definition of four different types of monitors: plots, modeled through the GSSScenarioPlot class and described in Section 4.2.5; pop-up alarms, which carry an associated text message and are modeled through the GSSScenarioAlarmMg class; monitors that allow us to modify the monitored value automatically and are modeled through the GSSScenarioModify class; and, finally, monitors that enable us to display the monitored value in the alarm area of the MASSIVA interface and that are modeled through the GSSScenarioAlarmVal class. For the plotting function, a charts file is modeled using the GSSCharts attribute within GSSScenarioMonitors.

5.2. Port interface models

Each GSSScenarioInterface must contain a reference to a port configuration modeled using the GSSIfacePortConfig class. Fig. 14 shows the section of the meta-model involving both classes.

The root class GSSIfacePortConfig must contain an object of class GSSIfacePort, which models the physical properties of the interface. The class is abstract, with several concrete classes inheriting from it that model the different types of ports that can be configured. There are six port types, including two auxiliary types. This subsection will only deal with the meta-model elements since the interfaces are described in Section 4.2.7.

The first type of port is the SpaceWire (SpW) port. This port type is modeled using the class GSSIfaceSpWPort. The different SpW port device types that can be configured are modeled using the enumerate GSSIfaceSpwPortType. A link number, a write port, and at least one read port (modeled using the GSSIfaceReadingPort class) must be configured.

The second type of port is the UART port. As the name implies, it configures a universal asynchronous serial receiver-transmitter port using the class GSSIfaceUartPort. The different hardware options for the UART ports are modeled using various enumerations: GSSIfaceUartPortBaudRate for the baud rate, GSSIfaceUartPortDataBits for the data bit length, GSSIfaceUartPortStopBits for the stop bit length, GSSIfaceUartPortParity for the parity and GSSIfaceUartPortFlowControl for selecting RTS/CTS or XON/XOFF flow control.

The following two port types are the client socket and server socket ports. They are modeled using the classes GSSIfaceSocketCliPort and GSSIfaceSocketSrvPort, respectively. Both inherit from the abstract class GSSIfaceSocketPort, which contains the TCP port number of the socket. In addition, the client class GSSIfaceSocketCliPort stores the IP address of the target server.

Several port types, such as UART and TCP sockets, do not manage packet data at the application level. Instead, they enable the seamless transmission and reception of byte streams across the interface. In such cases, a minimum Port Protocol must be configured, which is modeled using the class GSSIfacePortProtocol. This class defines several fields to create a logical packet from the sequential bytes sent or received. This protocol information is composed of a synchronization pattern and several size attributes, modeled using the classes GSSIfaceSyncPattern and GSSIfaceSize, respectively.

Next, the GSSIfaceSpWTCPort class models an auxiliary port related to the main SpaceWire port. The SpaceWire protocol allows sending time-codes routed from the SpaceWire port but out of the primary packet flow, so this feature has been implemented separately from the main SpaceWire type option in MASSIVA using this class.

Finally, another type of auxiliary port has been included for procedural steps that may not require sending data but where a response is expected. The most common case is system power-on: no packets are sent, but the power supply is turned on, and several packets are expected to acknowledge the system boot process and its status. For these cases, a dummy port has been created, which is modeled using the GSSIfaceDummyPort with no attributes.

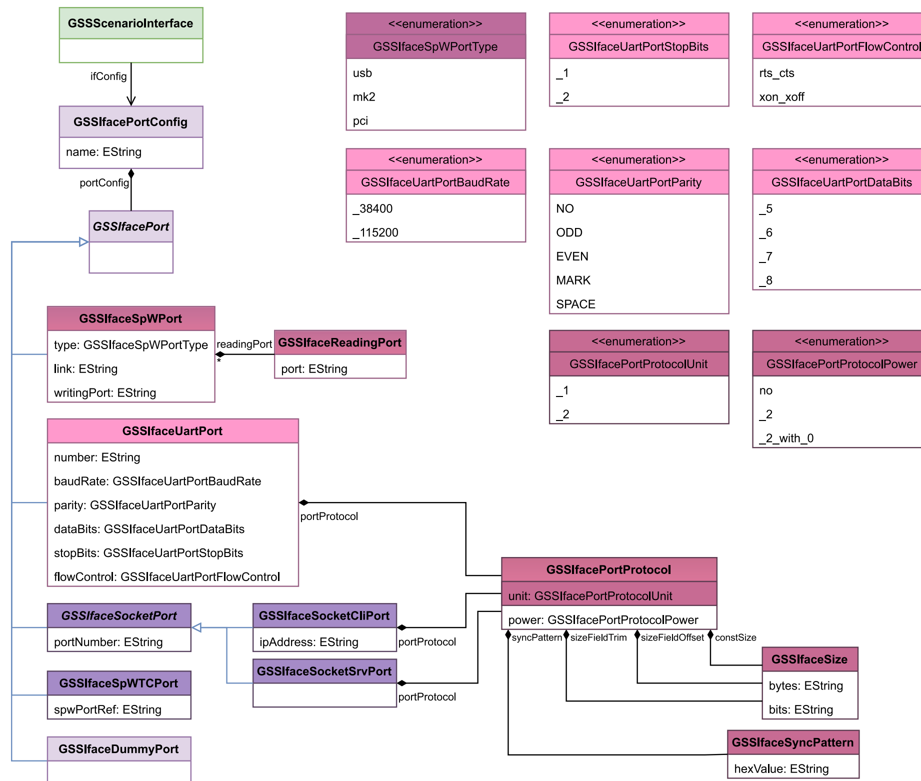


Fig. 14. MASSIVA interfaces model.

5.3. Tests procedure models

As aforementioned, every test procedure is configured in its own XML file. This structure facilitates modifying test procedures and the correction of errors and typos compared to hard-coded tests in closed systems. Besides, performing the same test in different scenarios is more accessible by changing only the interface configuration files. Fig. 15 shows the metamodel of a test procedure, including all its elements.

The root class of the test procedure is modeled by the GSSTestProc class, located in the upper, rightmost area of the figure. This class contains several attributes: a name, a unique identifier, a number of replays, and a mode, modeled with the enumerate GSSIfaceTestProcMode. A test procedure is simply a list of steps, each modeled using the GSSTestProcStep class. All the steps must contain *inputs* and can contain *outputs*, modeled respectively using the GSSTestProcAbstractInputs and GSSTestProcOutputs classes. The first one is an abstract class, as an input can be:

- a simple list of actual inputs, using the class GSSTestProcInputs, which contains the different inputs, modeled using the abstract class GSSTestProcInput. This class includes several attributes: a name, the numeric identifier of the desired interface where the input shall be launched, and a delay value composed of the attributes"delay_value" for the cipher and "delay_unit", which uses the GSSTestProcTimeUnit enumerate that allows us to select between seconds and milliseconds.
- an action, modeled using the GSSTestProcAction class. This class contains a message to be shown, and a delay and a timespan composed with the "delay_value", "delay_unit", and "span_value", "span_unit" attributes. Besides, it defines the action to be performed, which must be one in the enumerate GSSTestProcActionType:
  - instruction: a single message to be given to the test conductor.
  - tmtc_checking: an automated checking related to telecommands or telemetries.
  - checking: a checking that the test conductor in an external instrument must perform.

While the inputs list has a fixed order, the outputs list is a set of expected single outputs, and the class GSSTestProcOutputs contains two attributes named "valid_time_interval_value" and "valid_time_interval_unit" for defining the time validity of the output set, i.e., whether a telemetry was received before its expected time slot expires. Besides, the attribute "checkmode" defines how the outputs are expected to be received, as defined using the type GSSTestProcCheckmode:

- all: all defined outputs must be received for the step to be completed. Besides, they must be received in the same order as listed in the test procedure.
- allunsorted: all defined outputs must be received for the step to be completed, but they can be received in a different order than the order in the test procedure.
- any: only one defined output must be received for the step to be considered complete.

Every output is defined using the abstract class GSSTestProcOutput, which contains several attributes: a unique identifier, a name, the numeric identifier of the expected interface, and an optional flag for marking any particular output to be optional.

By defining both the inputs and outputs, MASSIVA can check in real-time whether the test result was as expected and, therefore, whether there is no need to check the register reports afterward. Although MASSIVA can be used in any domain, it has been designed for space software validation. In this domain, radiation tolerance requires that communication frequencies used are below the ones used on ground equipment. In the specific case of EPD, it can process a maximum of 10 telecommands per second, each telecommand being lower or equal

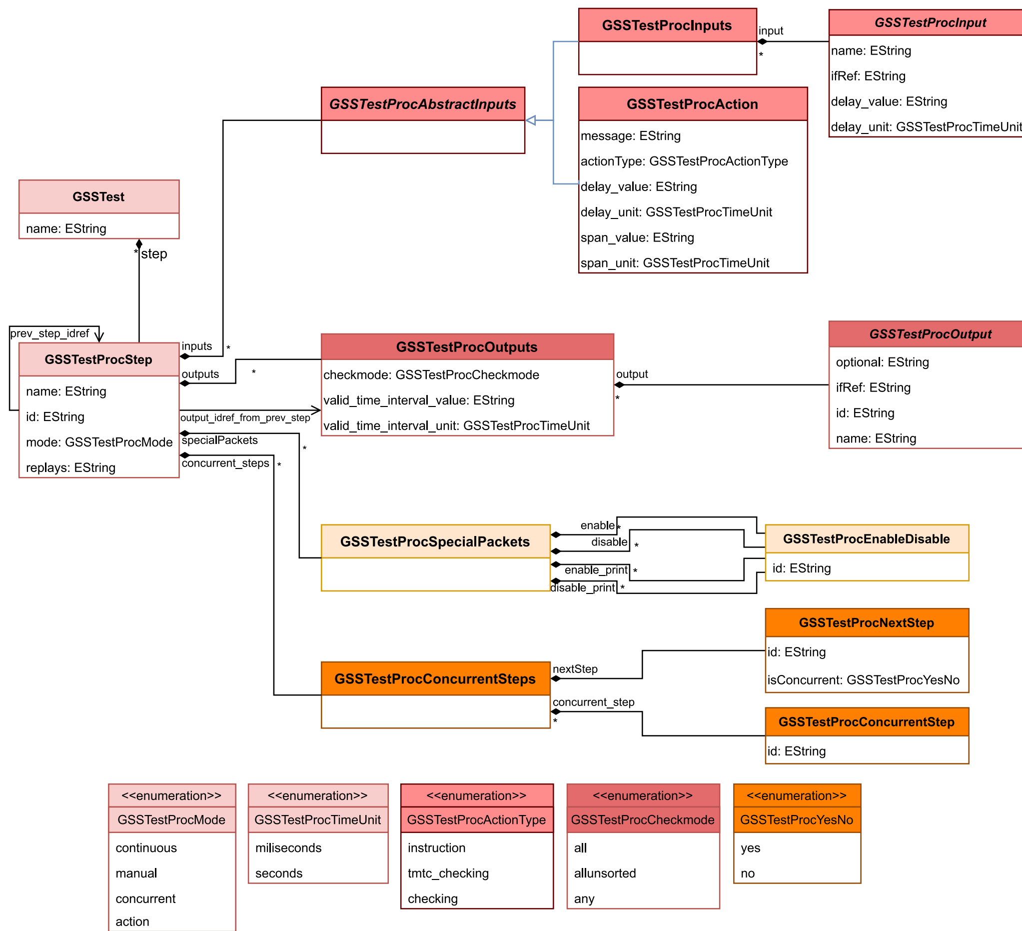


Fig. 15. MASSIVA test procedure model.

than 256 bytes, and for telemetries the maximum science rate in burst mode is 13630 bps. For this reason, MASSIVA is only limited by the PC interface and its processing capacity.

```
<test_proc name="TP_FT_SOLO_EPD_ICU_BSW_SERV_17">
       <step name="test_connection" id="0"
           prev_step_idref="NULL"
           output_idref_from_prev_step="NULL"
      mode="continuous">
       <inputs>
              <input_level_1 name="test_connection"
    ifRef="0" delay_value="120" delay_unit="miliseconds">
              <level1 format="DEFAULT"/>
              <app_to_level1 export="Tx/
                  epd_pus_tc_17_1_to_epd_pus_tc.xml"/>
              <level0 format="DEFAULT"/>
              <level1_to_level0 export="DEFAULT"/>
              </input_level_1>
       </inputs>
       <outputs checkmode="all"
           valid_time_interval_value="500"
           valid_time_interval_unit="miliseconds">
              <output_level_1 name="
                  test_connection_report" id="0" ifRef=
                  "0">
              <level1 format="DEFAULT"/>
              <level1_filter apply_def_filter="yes"
                  extra_filter="Rx/
                  epd_pus_tm_17_2_filter.xml"/>
              <level0 format="DEFAULT"/>
              <level1_from_level0 import="DEFAULT"/>
              <level0_filter apply_def_filter="yes"
                  extra_filter="NULL"/>
              </output_level_1>
       </outputs>
       </step>
</test_proc>
```

Listing 1: XML script example of simple Test Procedure

As aforementioned, both GSSTestProcInput and GSSTestProcOutput are abstract classes. The Test Designer can define the inputs and outputs with different levels for a more flexible test configuration. The lower ones represent the communication levels with their physical protocols, including the physical headers. The higher levels contain the telecommand or telemetry payloads. Listing 1 shows a simple MASSIVA test procedure defined in XML.

Packet formatting uses a hierarchically stacked structure, both for telecommands and telemetries. Lower levels are closer to the physical interface, while upper levels comprise the payload or application data level.

All telecommands are from top to bottom, starting at the payload of the highest level, usually the application level. Intermediate headers are added until the lowest level is reached, in which the physical headers are added just before sending the telecommand.

The validation hierarchy goes the other way around for telemetries. All the headers are removed from bottom to top as the content is extracted from the lowest level to the highest one. Validations and checkings can be performed at any level, for example, checking the communications protocol version at the lowest level, the telemetry type received at an intermediate level, and the final content at the highest level.

Fig. 16 shows different stacks that can be defined using MASSIVA. Both test inputs and outputs can be defined as having 4 levels named

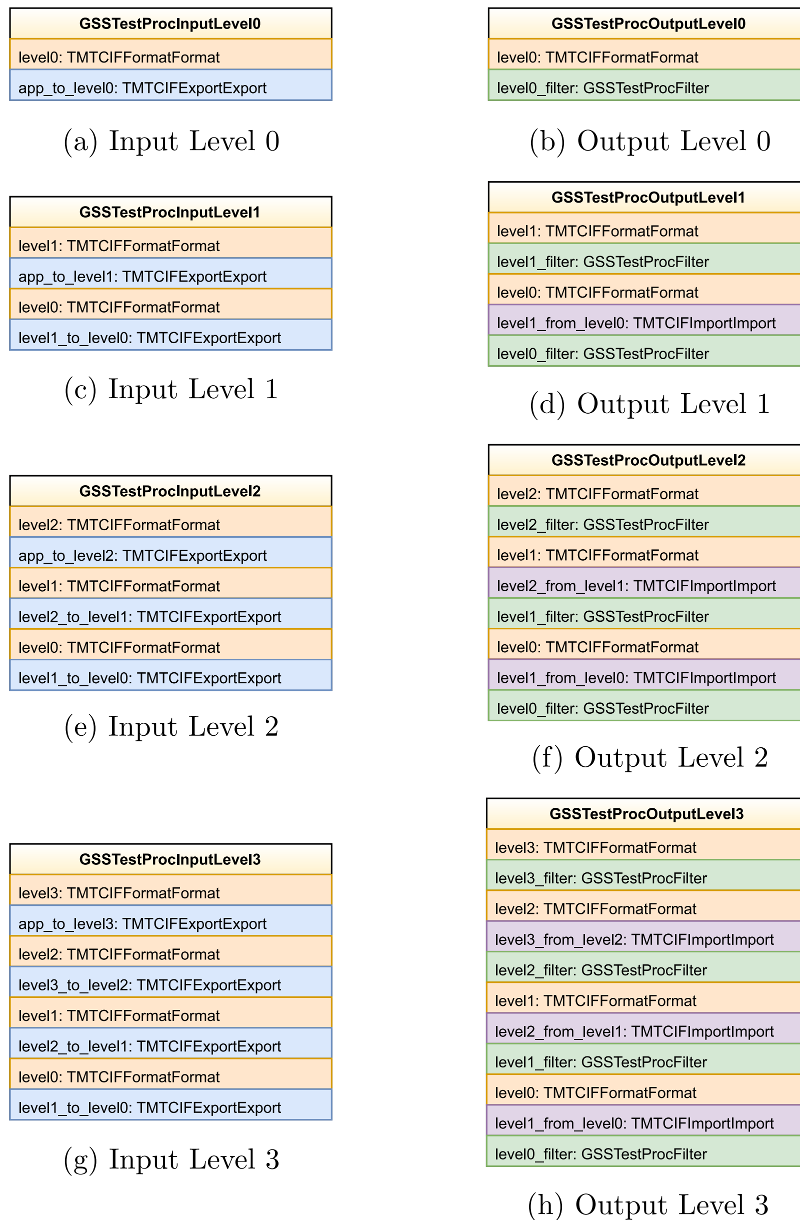


Fig. 16. MASSIVA hierarchical organization.

from 0 to 3. Fig. 16(a) shows the GSSTestProcInputLevel0 class, and Fig. 16(b) shows the GSSTestProcOutputLevel0 class, both of them used for creating inputs and outputs with a single level. Fig. 16(c) shows the GSSTestProcInputLevel1 class, Fig. 16(d) the GSSTestProcOutputLevel1 class. Then, Fig. 16(e) shows the GSSTestProcInputLevel2 class and Fig. 16(f) the GSSTestProcOutputLevel2 class. Finally, Fig. 16(g) shows the GSSTestProcInputLevel3 class and Fig. 16(h) the GSSTestProcOutputLevel3 class.

There are four other different classes involved in the process, which will not be studied in detail in this paper as they belong to the TMTCIF metamodel and are out of the scope of this paper. However, they will be briefly described next.

Independently of the number of stacked levels chosen, a format must be described for the inputs and outputs at each level (0 to 3) using the TMTCIFFormat class. For inputs, the desired information to be exported (i.e., the actual values to be sent in the telecommand) are modeled, from the top application level to the final headers, with the TMTCIFExport class. For the outputs, the expected data for comparison is stored in the so-called filters, defined using the TMTCIFFilterFilter class. Besides, another class, called TMTCIFImport, is used to slice the headers and pass the payload to the next level.

Any chosen format can be edited with low-code techniques since the entire TMTCIF metamodel uses XML files for input and output configuration. Any number of fields and parameters can be defined for a given level format. In addition, any field size can be defined as constant size, variable size, and in array form. These features facilitate the representation of any packet to be received or sent and allow defining telemetry and telecommand packets even late in the development and testing process.

It is also important to note that the number of levels is not fixed: a number of levels can be defined for telemetries and another one for telecommands. Besides, the default packet-level stacked structure can be overridden at any input or output within any test procedure. This last feature is useful for testing wrong-formatted telecommands or fields whose value is a limit or an edge.

The tests defined for the verification and validation of the aforementioned on-board software of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) in the Solar Orbiter mission

used three different levels. Lower levels 0 and 1 were used for CCSDS and PUS headers, respectively, whereas level 2 was used for the main payload.

All defined inputs and outputs rely on a predefined set of XML files containing all TM/TC data formats and expected values for the filters. These XML files can be created and edited both manually and automatically. Manual editing is suitable for creating files for testing fault-detection mechanisms, for example, with a wrong telecommand or format or an incorrect size. A tool was developed in Java for automatically parsing files in SCOS-2000 database format.

The same set of data formats and default filter files can be reused in the projects that share the same TM/TC database. Besides, previously created inputs and outputs can be used in different projects, both directly or as a basis for creating new complex inputs and outputs. In order to facilitate the reuse of steps, another tool was created for generating new procedures using previously defined steps.

For defining the filters, minterm/maxterm canonical forms are used. Boolean algebra variables result from the comparison operations, which can be defined using the classic operators *equal*, *bigger than*, etc. Then, all of these variables can be grouped using a sum of products or a product of sums.

The sum of products is a disjunction of minterms: first, a logic AND is performed with all the variables to create a single minterm, which will test *true* when all the variables are *true*. Then all the minterms are grouped in a logic OR operation, which means that the final result will test *true* when any of the minterms are *true*.

The dual operation of the previous one, the product of sums, is a conjunction of maxterms. First, a logic OR is performed with all the variables to create a single maxterm, which will test *true* when any of the variables is *true*. Then all the maxterms are grouped in a logic AND operation, which means that the final result will test *true* when all of the maxterms are *true*. Listing 2 shows a simple MASSIVA Minterm Filter defined in XML.

```xml
<MintermFilter formatFile="epd_pus_tm_5_1_1002_format.xml
    ">
    <BoolVar id="0" name="RID1002" constantType="hex">
        <FieldRef name="RID" />
        <Op type="equal" />
        <Constant value="1002" />
    </BoolVar>
    <BoolVar id="1" name="ASW_ver0" constantType="hex">
        <FieldRef name="ASW_version" />
        <Op type="equal" />
        <Constant value="0" />
    </BoolVar>
    <BoolVar id="2" name="ASW_subver7" constantType="hex"
        >
        <FieldRef name="ASW_subversion" />
        <Op type="equal" />
        <Constant value="7" />
    </BoolVar>
    <Minterm id="0">
        <BoolVarRef idRef="0" />
        <BoolVarRef idRef="1" />
        <BoolVarRef idRef="2" />
    </Minterm>
</MintermFilter>
```

Listing 2: XML script example of Minterm Filter

Outputs within steps are not mandatory since only one input is required to define a step. To define a test procedure, at least one step is required. This structure leads to a very flexible definition of the test procedure and makes it possible to use MASSIVA with test protocols and configurations that do not have to send responses after all requests. Apart from the mandatory inputs and optional outputs, there are two other features available in each step of the MASSIVA test procedure.

5.3.1. Test procedures optional features

The first optional feature is related to the MASSIVA Special Packets introduced in Section 4.1.1. In short, Special Packets are taken out of the test flow in the main configuration, but there may be some situations where their content or validity must be tested. For this purpose, the class GSSTestProcSpecialPackets allows performing changes in the Special Packets flow within the test procedure. Four different actions can be performed on each Special Packets, namely:

- disable: temporarily removes the "Special" status and adds it to the test flow.
- enable: rolls back the "disabling" and adds back the "Special", keeping again the packet out of the test flow.
- enable_print: keeps the Special Packet outside the test flow but prints all its information in the report for later checking.
- disable_print: stops the Special Packet information printing.

Step identifiers must be sequential and, by default, are launched one after another. However, another optional feature in a MASSIVA test procedure allows a group of steps to be defined as concurrent. This is an advantageous option when dealing with different interfaces which produce data at the same time.

When a group of concurrent steps is configured, MASSIVA will perform these steps in parallel, using a different thread for each interface. The class GSSTestProcConcurrentSteps contains a list of concurrent steps, modeled using the class GSSTestProcConcurrentStep, each of them including the identifier of a step that is concurrent with others, which also must be defined as concurrent. This class also contains a single next step attribute, modeled using the class GSSTestProc-NextStep. This class must include the identifier of the next step that must be performed and an attribute modeled using the enumerate GSSTestProcYesNo. The information in these two attributes allows MASSIVA to decide whether the next step should be performed on the same interface (i.e., on the same thread) or whether the concurrency block should be terminated and the corresponding threads merged. All the concurrent threads must lead to the same non-concurrent next step to ensure all test procedure sequences are adequately finished.

This concurrency mechanism allows the definition of several concurrent steps for any test procedure, which is usually necessary for testing devices with several interfaces running simultaneously, such as a set of sensors. The "next step" technique allows effortless convergence of the concurrency threads and easy detection of concurrent block failures within the test procedures.

6. Model-driven test procedures

When developing the test procedures for EPD's ICU validation, the Space Research Group (SRG-UAH) crew manually generated most of the XML files, only using automation tools for a small set of input files. The problem with that approach was that a modification in any document defining a test procedure or a telemetry or telecommand value implied test procedures creators had to apply the changes manually in all needed procedures, which is a very prone-error process.

SRG-UAH has started several spin-off projects that use the knowledge gained from experience in the EPD project. Among these is a project that uses model-driven processes to create the documents required for validation within the European Space Agency standards ECSS-E-ST-40 and ECSS-Q-ST-80C Rev.1 [5].

In these software verification and validation standards, the most important document is called Software Validation Specification (SVS), which defines the tests needed to fulfill all the requirements to be validated by testing. The verification and validation process requires another document called Software Verification Report (SVR). This document contains all the verification processes and reports related to the developed software, including the verification reports that must connect the tests defined in the SVS to the evidence taken from the test campaign reports.

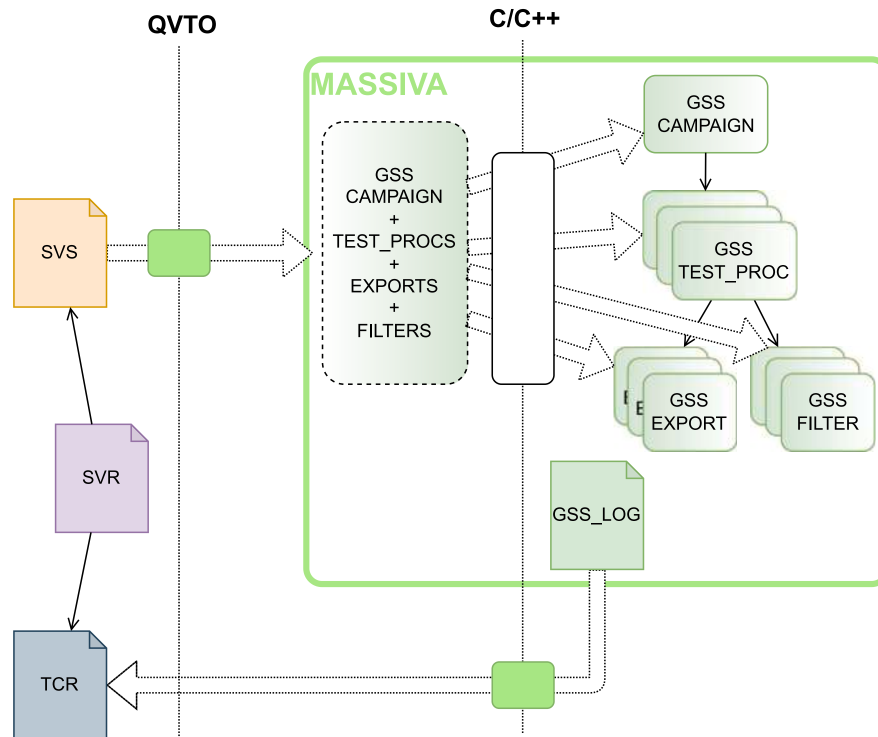


Fig. 17. GSS transformations.

MASSIVA fills the gap between SVS and SVR. The tests described in the SVS can be used to define a MASSIVA test campaign. MASSIVA then generates log reports that can be used to create the SVR.

We created model-to-model transformations between SVS and MASSIVA models using QVT operational (QVTo) and C/C++. Fig. 17 shows the models and transformations diagram of a V&V process focusing on the generated models related to MASSIVA.

The SVS and SVR models of the standard are shown along with the Test Campaign Report (TCR) model. The latter is not present in the standard but represents the report of a test campaign with the corresponding evidence and the status *Pass* or *Fail*.

All models located inside the large rectangle of the figure are the previously seen MASSIVA models, which are independent of the ECSS standards. MASSIVA produces a log report called "GSS_LOG" after the execution of the tests, which is a plain text file containing the status of the executed test procedures. This log report is used with the TCR model to create the final verification report.

There is a first model-to-model transformation using QVTo for generating a complete single file from SVS, which contains all needed information for a test campaign, namely the campaign itself, test procedures, exports (for TCs), and filters (for TMs), but this is not the file structure that MASSIVA requires. Then another transformation, created using C/C++, generates the files needed for configuring the test campaign in MASSIVA.

Most test campaigns have different set-up configurations, depending on the items under test. Each set-up configuration has different procedures, so different MASSIVA log files are usually produced, and several TCR instances must be generated. The last transformation, also in C/C++, generates a TCR from each MASSIVA log report. The SVR can then be created using all the TCR instances and the SVR, coming full circle to the standards.

Besides the ECSS standards workflow presented here as a use case, MASSIVA can be easily integrated into any development, verification, and validation workflow. MASSIVA test procedures and test entries are XML files, and MASSIVA log report outputs are plain text files. This means that any model-driven software development process based on ECSS-E-ST-40 or any other standard can be integrated into MASSIVA.

In order to integrate MASSIVA into any MDE workflow using modeled standards, only two new transformations are required: the first from the standard validation procedure models to MASSIVA test campaigns and the second from the MASSIVA report files to the corresponding standard verification report models. To illustrate this feature, the models and transformation for the XML files are available online.[1]

7. Case study

This tool traces its roots back to the development of the on-board software of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) on-board the Solar Orbiter mission carried out by the Space Research Group of the University of Alcalá (SRG-UAH). The original tool, called SRG-GSS (Space Research Group - Ground Support Software), was designed to verify and validate all the original requirements and the whole set of telecommands and telemetries.

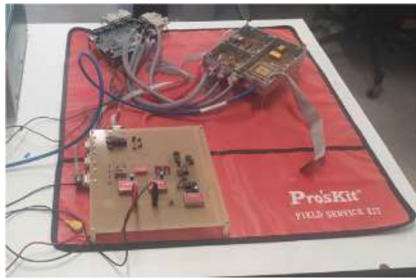
Within the EPD software development, SRG-UAH used a model-driven component-based approach to design and deploy the application software supported by the MICOBS framework [31–33].

Fig. 18 shows the scenarios where MASSIVA was used for the verification and validation campaigns. The scenarios shown in Figs. 18(a) and 18(c) were designed for the validation of the ICU software in EPD using an Engineering Model (EM) and an Engineering Qualification Model (EQM). They both share the SpaceWire connectivity port and four serial UART ports.

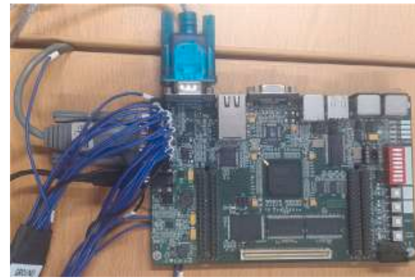
In the first scenario, the four instrument sensors were emulated at the opposite end using MASSIVA, while in the second scenario, EQM versions of the actual instruments were used. In these two models, both EPD's ICU Boot Software (ICU BSW) and EPD's ICU Application Software (ICU ASW) were tested. The main difference with our case study is connectivity since only the SpaceWire port is used for testing EPD's ICU BSW.

Finally, the scenario shown in Fig. 18(b) is a more straightforward scenario used for testing the so-called SRG-BSW, another boot software

[1] The models are located in https://github.com/uah-srg-tech/gss-eclipse.



(a) EPD ICU EM Test Scenario with 4 emulated sensors



(b) SRG BSW Test Scenario

(c) EPD EQM ICU Test Scenario with 4 EQM sensors

Fig. 18. MASSIVA test scenarios.

Table 2
Verification and validation port configuration for different software projects using MASSIVA.

Project	EPD's ICU BSW	EPD's ICU ASW	SRG-BSW
Main interface	SpaceWire	SpaceWire	UART/IP Sockets
Main interface protocol	CCSDS/PUS	CCSDS/PUS	CCSDS/PUS
Other interfaces	–	4 UART	–
Other protocols	–	Custom	–

based on the EPD's ICU BSW. This time a single serial UART port was used instead of the SpaceWire port of the EPD's ICU BSW. Another version of this SRG-BSW was tested on a board (not shown) using IP sockets instead of the serial UART port. Table 2 shows the port configuration for these three scenarios.

With both EPD's ICU software project interfaces, a SpaceWire device, including SpaceWire time-codes, was connected via USB to the MASSIVA host PC. The other projects did not use SpaceWire, only UART ports and sockets, so no other device was needed.

Several of the above projects share the same payload telecommands and telemetries for basic tests such as test connection or reset, as they are defined in the PUS services list. When using MASSIVA, only the low physical interface configuration (SpaceWire, UART, or IP Sockets) had to be changed, reusing the application data. Conversely, when several projects used the same physical interface, for example, a given UART port at a given baud rate, only the logical packet definitions had to be changed, keeping the physical port configuration.

Table 3 shows the total number of requirements, test procedures, telecommands, and telemetries involved in the verification and validation of different projects: EPD's ICU BSW, EPD's ICU ASW, BSW UART.

MASSIVA was executed on different computers with different operating systems and port configurations during these test campaigns. It worked correctly in all Microsoft Windows-based systems, from Windows XP and above. A Linux version was also successfully used in Ubuntu and other Debian-based Linux distributions for 32-bit and 64-bit architectures. The vendor of the SpaceWire device used did not provide drivers for operating systems other than Microsoft Windows. Still, any SpaceWire device whose vendor provides drivers for Linux could be easily integrated into MASSIVA. Finally, MASSIVA has also been tested on virtual machines running all these operating systems, functioning as on the real machines.

MASSIVA's source code is available online[2] for anybody who wants to compile the tool for their system. The only compiling dependencies are the SpaceWire drivers, which can easily be deactivated using preprocessor definitions.

8. Conclusions

This paper has presented a model-driven engineering approach and a tool for assisting in the verification and validation process for space software applications called MASSIVA. The solution provides a model scheme for the configuration files, test campaigns, and related files. The final objective of this approach is to provide a testing environment

[2] The source code in C/C++ is located in https://github.com/uah-srg-tech/massiva.

Table 3
Verification and validation metrics for different software projects using MASSIVA.

Project	EPD's ICU BSW	EPD's ICU ASW	SRG-BSW
Total number of V&V Test Procedures	46	212	29
Total number of Files	14314	88377	11119
Total number of Tested Telecommands	15	118	13
Total number of Tested Telemetry Packets	30	318	29

with a hierarchical organization for promoting the reuse of tests with no recodification.

To this end, we have implemented several features for automatically performing the test campaigns and developing and debugging them. Besides, MASSIVA features have been explained, and how to fit the tool into a space software development project using ECSS standards has been shown. Based on that model structure, several examples of the hierarchy levels have been shown, allowing any project to follow the same structure. Finally, several test campaigns using different configurations have been performed using different tool compilations, validating the test campaigns with the related documents and requirements.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] P. Fortescue, G. Swinerd, J. Stark, Spacecraft Systems Engineering, John Wiley & Sons, 2011.

[2] D. Rossetti, B. Keer, J. Panek, B. Ritter, B.B. Reed, F. Cepollina, Spacecraft modularity for serviceable satellites, in: AIAA SPACE 2015 Conference and Exposition, 2015, p. 4579.

[3] A. Rodrigues da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Comput. Lang. Syst. Struct. 43 (2015) 139–155, http://dx.doi.org/10.1016/j.cl.2015.06.001, URL https://www.sciencedirect.com/science/article/pii/S1477842415000408.

[4] J. Rodriguez-Pacheco, R.F. Wimmer-Schweingruber, G.M. Mason, G.C. Ho, S. Sanchez-Prieto, et al., The energetic particle detector - energetic particle instrument suite for the solar orbiter mission, Astron. Astrophys. 642 (2020) http://dx.doi.org/10.1051/0004-6361/201935287, URL https://www.aanda.org/articles/aa/full_html/2020/10/aa35287-19/aa35287-19.html.

[5] A. Montalvo, P. Parra, O. Polo, A. Carrasco, A. Silva, A. Hellín, S. Sánchez Prieto, Model-driven system-level validation and verification on the space software domain, Softw. Syst. Model. (2021) 1–28, http://dx.doi.org/10.1007/s10270-021-00940-8.

[6] T. Hauck, J. Finnigan, Use of the ground support equipment operating system (GSEOS) software on the messenger mission: A case study, 2003.

[7] A. Armitage, Right-sizing test tools. a low cost alternative to full-scale system checkout equipment for instrument & payload, in: 11th Intr. WS on Simulation and EGSE Facilities for Space Programmes, (SESP) 2010, 2010.

[8] N. Peccia, SCOS-2000 ESA's spacecraft control for the 21st century, in: 2003 Ground System Architectures Workshop, 2003.

[9] G. Patrick, R. Patrick, Empowering the check-out user: Integrated simulation, in: Simulation and EGSE Facilities for Space Programmes, (SESP) 2015, 2015.

[10] S. Chusri, U. Khowsuwan, J. Plaidoung, P. Pipitsunthonsan, S. Chaimatanan, Blackbox to open innovation: Experience in self learning in developing its own satellite control system - the VOSSCA, in: 68th International Astronautical Congress, IAC, 2017.

[11] C. Stangl, A. Braun, M. Geyer, GECCOS - the new monitoring and control system at DLR-GSOC for space operations, based on SCOS-2000, 2014, http://dx.doi.org/10.2514/6.2014-1602.

[12] A. Guiotto, B. Acquaroli, A. Martelli, MaTeLo: Automated testing suite for software validation, 532, 2003, p. 30.

[13] V. Garousi, M. Felderer, Ç.M. Karapıçak, U. Yılmaz, Testing embedded software: A survey of the literature, Inf. Softw. Technol. 104 (2018) 14–45, http://dx.doi.org/10.1016/j.infsof.2018.06.016, URL https://www.sciencedirect.com/science/article/pii/S0950584918301265.

[14] P. Strandberg, E. Enoiu, W. Afzal, D. Sundmark, R. Feldt, Information flow in software testing – an interview study with embedded software engineering practitioners, IEEE Access PP (2019) 1, http://dx.doi.org/10.1109/ACCESS.2019.2909093.

[15] R. Marinescu, C. Seceleanu, H. Guen, P. Pettersson, A research overview of tool-supported model-based testing of requirements-based designs, Adv. Comput. 98 (2015) 89–140, http://dx.doi.org/10.1016/bs.adcom.2015.03.003.

[16] C. Hennig, From Data Modeling To Knowledge Engineering in Space System Design, KIT Scientific Publishing, 2018, p. 306, http://dx.doi.org/10.5445/KSP/1000073688.

[17] H.-m. Qian, C. Zheng, A embedded software testing process model, 2009, http://dx.doi.org/10.1109/CISE.2009.5366362.

[18] I. Fernandez, A. Cerbo, E. Dehnhardt, M. Tipaldi, B. Bruenjes, A testing framework for critical space SW, 2015.

[19] A. Armitage, K. Leadbeater, A. Polverini, R. Patrick, Future proofing EGSE- is virtualization the answer? in: 11th Intr. WS on Simulation and EGSE Facilities for Space Programmes, (SESP) 2010, 2010.

[20] R. Darwish, L. Gwosuta, R. Torkar, A controlled experiment on coverage maximization of automated model-based software test cases in the automotive industry, 2017, pp. 546–547, http://dx.doi.org/10.1109/ICST.2017.65.

[21] S.P. Karmore, A.R. Mahajan, G.C. Jarbias, Development of software interface for testing of embedded system, in: 2013 15th International Conference on Advanced Computing Technologies, ICACT, 2013, pp. 1–6, http://dx.doi.org/10.1109/ICACT.2013.6710547.

[22] R. Melton, Ball aerospace COSMOS open source command and control system, 2016.

[23] ECSS Secretariat, Space Engineering. Software, ECSS-E-ST-40C, 2009.

[24] ECSS Secretariat, Space Product Assurance. Software Product Assurance, ECSS-E-ST-80C Rev.1, 2017.

[25] ECMA International, Standard ECMA-376. Open Office XML File Formats, ECMA-376, fifth ed., 2016.

[26] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, second ed., Addison-Wesley Professional, 2009.

[27] M. Eysholdt, H. Behrens, Xtext: implement your language faster than the quick and dirty way, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH '10, ACM, New York, NY, USA, 2010, pp. 307–309.

[28] ECSS Secretariat, Telemetry and Telecommand Packet Utilization, ECSS-E-70-41C, 2016.

[29] O. R. Polo, L. de Salvador, M. Angulo, J.M. de la Cruz, Development plan of the on board satellite software based on room modelling and evolution of component based prototypes, in: Real-Time Programming 2003 (WRTP 2003): A Proceedings Volume from the 26th IFAC/IFIP/IEEE Workshop, Łagów, Poland, 14–17 May 2003, Elsevier Science Limited, 2003, p. 93.

[30] Ó.R. Polo, P. Parra, M. Knoblauch, I. García, S. Sánchez, Component-based engineering and multi-platform deployment for nanosatellite on-board software, in: Proceedings of the DASIA 2012 Conference, 2012.

[31] P. Parra, O.R. Polo, A. Carrasco, A. da Silva, A. Martinez, S. Sanchez, Model-driven environment for configuration control and deployment of on-board satellite software, Acta Astronaut. 178 (2021) 314–328, http://dx.doi.org/10.1016/j.actaastro.2020.09.017, URL http://www.sciencedirect.com/science/article/pii/S0094576520305555.

[32] P. Parra, O.R. Polo, M. Knoblauch, I. Garcia, S. Sanchez, MICOBS: multi-platform multi-model component based software development framework, in: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, CBSE '11, ACM, New York, NY, USA, 2011, pp. 1–10.

[33] P. Parra, O.R. Polo, J. Fernández, A. Da Silva, S. Sánchez, A. Martínez, A platform-aware model-driven embedded software engineering process based on annotated analysis models, IEEE Trans. Emerg. Top. Comput. 9 (1) (2021) 78–89, http://dx.doi.org/10.1109/TETC.2018.2866024.