Software and Systems Modeling (2022) 21:2367–2394
https://doi.org/10.1007/s10270-021-00940-8

THEME SECTION PAPER

Model-driven system-level validation and verification on the space software domain

Aarón Montalvo[1] · Pablo Parra[1] · Óscar Rodríguez Polo[1] · Alberto Carrasco[1] · Antonio Da Silva[1] · Agustín Martínez[1] · Sebastián Sánchez[1]

Received: 1 November 2020 / Revised: 4 October 2021 / Accepted: 7 October 2021 / Published online: 9 November 2021

Abstract
The development process of on-board software applications can benefit from model-driven engineering techniques. Model validation and model transformations can be applied to drive the activities of specification, requirements definition, and system-level validation and verification according to the space software engineering standards ECSS-E-ST-40 and ECSS-Q-ST-80. This paper presents a model-driven approach to completing these activities by avoiding inconsistencies between the documents that support them and providing the ability to automatically generate the system-level validation tests that are run on the Ground Support Equipment and the matrices required to complete the software verification. A demonstrator of the approach has been built using as a proof of concept a subset of the functionality of the software of the control unit of the Energetic Particle Detector instrument on-board Solar Orbiter.

Keywords MDE · Validation · Verification · Space · Software · ECSS

1 Introduction

The European Space Agency (ESA) has adopted a set of standards that apply to every engineering process involved in space missions. These standards have been defined by the European Cooperation for Space Standardization (ECSS), which is supported by ESA and several space national agencies such as CNES (France), UK Space (UK), DLR (Germany), and CSA (Canada). The ECSS standards that apply to the software development process are ECSS-E-ST-40 (Space Engineering. Software) and ECSS-Q-ST-80 (Space product assurance. Software product assurance). Both standards have undergone several revisions since their inception. Current versions are called ECSS-E-ST-40C (March 2009) [1] and ECSS-Q-ST-80C Rev.1 (February 2017) [2].

ECSS-E-ST-40 applies to the software development process for space missions. It defines the procedures to be followed and the documentation to be provided during software construction, covering all the project development phases, from the software specification to the subsequent validation, verification, and maintenance. ECSS-Q-ST-80, in turn, defines the product assurance mechanisms that guarantee the quality of the developed software. Both standards are combined during all phases of the on-board software development, from the initial specification to the final validation and verification stages.

One of the main objectives of verification is to ensure that the necessary evidence is provided that the developed software conforms to the specifications. A fundamental part of this information is the tracing of the user specifications (called *Software System Specification* in the ECSS standard) against all the elements derived from them, such as the software requirements, the architecture, or the detailed design. It is also necessary to trace the specifications and the mentioned derived elements against the different scheduled analyses and tests and their corresponding reports.

The management of requirements, validation tests, and verification evidence under the ECSS standard is a costly task that is generally supported by one or more tools. There are currently several tools on the market that facilitate the management of verification evidence, including the traceability of the requirements specification against the corresponding validation tests. Rational DOORS [3] deals with managing

Communicated by J. Araujo, A. Moreira, G. Mussbacher, and P. Sánchez.

✉ Aarón Montalvo
aaron.montalvo@uah.es

[1] Space Research Group, University of Alcalá, Alcalá de Henares, Madrid, Spain

possible changes in the requirements and verifying that the produced validation evidence covers all the requirements. Another example is Sparx Systems Enterprise Architect [4], which is a general-purpose tool based on UML [5] that can be used for requirements management and traceability within the lifetime of any development project.

In the space domain, system-level validation tests involve, in addition to the on-board software itself, other mission-specific elements such as telemetry and telecommand (TM/TC) interface information or the Ground Support Environment (GSE). The TM/TC information is provided as a database that follows the format defined by ESA's Satellite Control and Operation System 2000 (SCOS-2000) mission control system [6]. This database includes the definition of all possible TM/TC packets that the on-board software can send or receive. The GSE consists of the test harness that emulates the flying environment where the on-board software operates, providing the required hardware interfaces to the on-board processor and enabling the execution of the system-level validation test using a hardware-in-the-loop approach. The management of the GSE is centralized through the so-called Ground Support Software (GSS), a software tool or set of tools used to perform the validation tests.

Within this context, model-driven engineering (MDE) techniques can be beneficial when dealing with the validation and verification process defined in the ECSS standard. On the one hand, MDE avoids the errors inherent to the direct use of documents and ensures consistency between the elements incorporated into the process in the different phases. On the other hand, this type of approach makes it easier to obtain, through transformations, the different products, and evidence, whether intermediate or final.

The approach presented in this paper aims to use MDE techniques to support the system-level validation and verification process for space software applications under the standards ECSS-E-ST-40 and ECSS-Q-ST-80. The work covers common aspects of requirements engineering, e.g., checking for inconsistencies during the definition of any element contained in the development process and its traceability against the requirements. These checks include that requirements marked to be validated by testing should be covered by test procedures, not by inspection or analysis reports, or that requirements removed in a previous version are no longer in the updated traceability matrix.

Besides, as a novel aspect, the proposed approach uses intermediate platform-independent models for defining the validation tests, the validation environments, and the different scenarios. The complete set of meta-models that are defined conform to the system-level on-board software domain defined by the ECSS-E-ST-40 and ECSS-Q-ST-80 standards. The model elements associated with the TM/TC database are subsequently used to determine the system-level tests and verification matrices. In this way, the elements available for defining tests go beyond the on-board software domain and are customized for each project. The system-level test procedures are defined as a list of steps that establish the telecommands to be sent and the expected telemetries to be received, thus using the elements resulting from incorporating the information from the TM/TC database. In addition, since the software to develop is part of an embedded system, the models of the validation environments can describe both the capabilities of the test harness and its interfaces to automate the tests and import the information from the reports obtained after the execution. This approach makes it possible to generate, from the procedure models, and through transformations, the input files of the GSS toolset, thus achieving complete automation of the system-level validation tests.

A proof of concept of these MDE techniques is presented in this paper. It has been built from a subset of the requirements and validation tests of the on-board software of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) [7], that is part of the Solar Orbiter mission. The software was developed by our group, namely the Space Research Group of the University of Alcalá (SRG-UAH). It is currently running in nominal mode on-board the Solar Orbiter, which was launched in February 2020.

The solution adopted also integrates the complete TM/TC database of EPD and shows how to use the platform-independent models for defining the validation environments and scenarios needed to automatically generate the input files corresponding to the system-level validation tests for a specific GSS tool. This tool, called SRG-GSS (Space Research Group—Ground Support Software), was used for the validation of the on-board software of the ICU of EPD, thus demonstrating that the solution provided is feasible to be used in a real environment. The whole process is automatic: only the selection of the report logs at the end of the verification process is performed manually. Finally, the proof of concept demonstrates that the validation tests generated automatically, as a product of the model-based approach proposed in this work, are the same as those used to qualify the on-board software of the ICU of EPD.

The rest of the paper is organized as follows. Section 2 covers the related works. Section 3 describes the general validation and verification process. Later, Sect. 4 details all the models present in the approach. The proposed proof of concept is then summarized in Sect. 5. Finally, the last section contains the conclusions and future works.

2 Related works

The use of model-based engineering (MBE) and model-driven engineering (MDE) techniques to automate the activities involved in the validation and verification of software

systems has been approached from different perspectives and with different scopes.

On the one hand, in the last decades, many research works have been carried out trying to apply MBE and MDE techniques to managing software project requirements. The model-based requirement engineering approach [8] uses models to facilitate the management and organization of requirements. The consistency between the requirements and their derived design elements provides agile management of software change requests and software problems during the development process.

The integration of MDE and requirements engineering (RE) [9,10] goes one step further and uses models as a central element in requirements management. A direct application of MDE techniques to RE allows the automatic generation of reports such as the traceability matrix between requirements and design elements of different abstraction levels.

On the other hand, and complementary to requirements management, efforts have also been made to integrate MBE techniques in different software development methodologies focused on validation and verification, such as test-driven development (TDD) and behavior-driven development (BDD).

Test-driven development (TDD) [11] is based on a simplified approach to software development in which the effort is mainly focused on testing, whose design and implementation are addressed before the actual implementation of the software. In this way, priority is given to activities aimed at reducing errors detected by software testing [12], as opposed to the classical approach to the software development process, where the system requirements and their transposition into a design are completed before addressing the design of the tests. The pragmatism of this approach was welcomed with great interest by the community that adopted the agile software development principles [13]. In this way, an agile software development process based on TDD proposes to approach detailed design and implementation tasks as an iterative process of refactoring and continuous integration, reducing, in each iteration, the number of errors obtained in the tests. The variant known as acceptance test-driven development (ATDD) [14] is a type of TDD where the development process is driven by acceptance tests that meet the user requirements, unlike the classic TDD approach, which is more centered on unit and integration tests. MBE techniques have been broadly applied to TDD [15–18] and ATDD [19]. In these solutions, models and tools are integrated into TDD and ATDD methodologies to facilitate the test definition and automation.

On the other hand, behavioral-driven development (BDD) was introduced by Dan North in 2006 to address some of the problems of ATDD and TDD [20]. While ATDD and TDD conceive software development as a test-centric process, BDD focuses more on the specification of requirements that express the desired behavior of the system. This approach is consistent with the fact that the primary source of errors in software projects is the misunderstanding of the specification [21].

Although BDD proposes that the artifacts that drive development should not actually be the tests but rather a specification of the program's behavior, they can also be used for test generation. In this sense, Tavares et al. [22] have a perspective of BDD as a methodology that allows validation to be integrated into the design phase so that acceptance tests can be obtained before the system is fully designed. This approach proposes that the system to be designed is the one that must pass the automatically generated tests, conforming to a formal specification of the expected behavior of the system and using a tool that transforms this specification into tests. This BDD perspective converges in objectives with the model-based testing (MBT) approach. MBT proposes using formal models to define the behavior and interface of the software and, optionally, the environment with which it interacts. These models allow the automatic generation of the elements involved in their validation, such as test cases or the data that feed them [23]. They can also automate the generation of evidence needed during verification, such as matrices that provide information on the coverage achieved by test cases, represented, for example, in terms of coverage of states or transitions mapped to the behavioral model [24]. The synergies between BDD and MBT have been explored in several papers [25,26].

In an area such as ESA's space missions, where a large part of the critical functionality of the system relies on the onboard software, the adoption of MDE techniques from agile methodologies requires an adaptation that fits the exhaustive process of validation and verification required by the ECSS standard. Therefore, in our opinion, the direct application of the approaches proposed by TDD and BDD is not entirely adequate. In them, the modeling of elements such as the specification or the design is left in the background, giving priority to the behavioral model, from which it is feasible to obtain, in an assisted way, or even automatically, tests at different levels, and it is these tests that drive the development process. This criticism about the benefits of reducing the design effort has been remarked in [27]. Following the same direction, the ECSS standard does not alleviate the effort of modeling the specification. On the contrary, it requires the completion, in detail, of both a user specification (called the *software system specification*), which constitutes the requirement baseline of the software, and a technical specification, which arises from translating the user specification into software requirements. In this sense, a model-driven approach such as the one presented in this article, which is focused both on requirements management and the definition of validation tests through a specific meta-model, and which simultaneously assists in the verification of the process itself, is better

adapted to the ECSS standard than the solutions based on TDD or BDD.

It also represents an evolution compared to the MBT approach since, as will be presented in the following sections, it is the models that drive the validation and verification process entirely at the system level. Moreover, its objective is not to replace the specifications with a logical model that allows the automation of the tests but to ensure that the logical model of the system-level response maintains traceability with the specifications to enable not only the automatic generation of the tests but also the generation of the verification matrices with respect to the specifications. Since this is also a domain in which the product to be developed is embedded software, it is necessary to address the need to model the test harness capabilities that physically implement the validation environment and its interfaces. The modeling of these elements will allow automating the tests and importing the reports obtained after their execution to build the mentioned verification matrices.

The MDE techniques, such as those proposed in the article, have also been widely used to manage the validation and verification activities required by the software development process. On the one hand, the model-driven testing (MDT) approach uses models to drive the entire testing process and integrates model transformation technologies to automatically generate oracles and test cases [28]. Model validation and model transformations can also be used to automate more complex tasks such as the verification of non-functional properties of software systems. In this direction, some frameworks, such as MICOBS [29], or TASTE [30], enable the software system timing analysis [31,32], or the analysis of software quality metrics [33]. The MDT approach has also been used in combination with the BDD methodology in [34].

There have also been some interesting proposals for using MDE to automate the activities of software product lines [35–37]. The use of MDE techniques in software product lines is of particular interest. This is an area characterized by reuse, where standards that specify the processes, artifacts, and documents to be completed during software development are often applicable. These characteristics facilitate the construction of models that fit both the products and the activities of their development process. Furthermore, they enable the integration of tools that are used on a recurrent basis. Examples in this direction can be found in the application of MDE techniques to the development of software under the avionics standard RTCA DO-178C [38–40]. The first one characterizes different model-based approaches and analyzes their philosophy, achievement of DO-178C coverage, information handling, and usage. The second example instead focuses on software development life cycles, namely Waterfall, V, Spiral, and Incremental life cycles, and the different tailorings needed to make them compliant with the DO-178C standard. Finally, the third example shows various UML diagrams for modeling a module of DO-178C. These three examples are centered on modeling the avionics domain, showing how different MDE techniques can facilitate the software development processes.

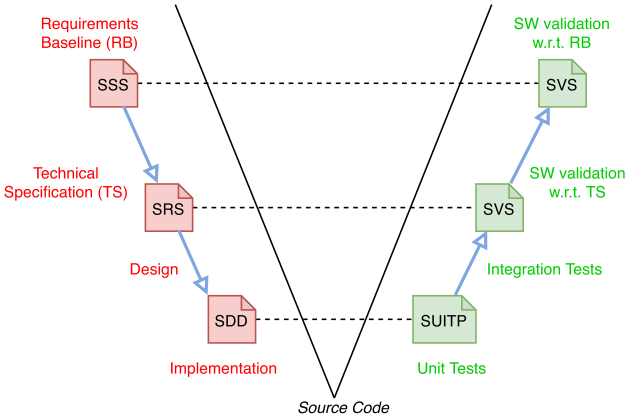


Fig. 1 Model-driven validation and verification process

Similarly, the ECSS-E-ST-40 standard for space software development adopted by ESA is a good candidate for using MDE techniques, and there are examples such as [41] or [42] which also show the suitability of these techniques applied to the standard. The first example uses MDE techniques for modeling the ECSS-E-ST-70-01C standard, creating a Domain-Specific Language (DSL) for deploying a full model-based IDE. Similarly, the second example builds a toolchain for a space software development process from requirements using models and transformations. Both of them show how MDE techniques for software development in the space domain using ESA standards contribute to improving system reliability and usefulness.

In these works, using MDE techniques allows accelerating the software development process under the corresponding standard, contributing to improving its reliability. However, none of them explicitly covers the automation of one of the most time and effort-consuming tasks, i.e., the development of the functional system-level tests that validate the requirements.

3 Validation and verification process

The software development process described in the ECSS-E-ST-40 standard, and complemented with ECSS-Q-ST-80, defines several phases. Each phase concludes with a specific review and the generation of a particular deliverable document. Figure 1 shows a general outline of the whole software development process, focusing on the documents produced on each stage.

In this software development process, *validation* refers to the activities that answer the question "have you devel-

oped the right system?", while *verification* addresses the question "have you developed the system correctly?". Thus, validation focuses on providing the set of tests and analyses, with their corresponding reports, that demonstrate that the behavior of the system is correct, both in its functional (the outputs obtained are those expected in each test case), and extra-functional aspects (response time, memory utilization, or energy consumption comply with the limits set as system metrics). On the other hand, verification analyzes whether the entire development process has been carried out following the product assurance plan. To this end, verification analyzes the matrices that guarantee that all the required steps have been completed, verifying that there is traceability between the elements incorporated in the different phases. In system-level verification, the ECSS standard requires that the specifications are covered by the software requirements, and these in turn by the validation tests, and that once these tests have been executed, their reports are available and provide the expected results. This information, generated in the form of a *Software Verification Report* (SVR), also incorporates the status of "To Be Defined" (TBD) or "To Be Confirmed" (TBC) items in the different pre-qualification reviews. Finally, system-level verification also includes configuration control of all documents related to specifications, software requirements, validation tests, and reports. These data are necessary so that the specific version of each document delivered in the software data package can be determined at each project review.

The process begins with the software specification. This activity involves the compilation and classification of the requirements settled by the customer that apply to the software. As a result of this activity, the so-called *software system specification* (SSS) document is defined. This document can be considered as an input to the software development process, and it is generated in collaboration with the customer, and it is focused on "what" the software shall do to meet scientific and operational goals. It is, therefore, a customer-defined specification, and it must be included as part of the *requirement baseline* (RB). In some projects, the elaboration of a formal SSS is substituted by the identification, inside several system-level documents, of which customer requirements apply to the software. Interface requirements at the customer level can also be compiled in a specific document, called the *interface requirements document* (IRD). Under this approach, it is possible to consider more than one document contributing to the software RB.

The next step is to derive the software requirements from the requirements defined in the RB. The document generated as a result of this activity is the *software requirement specification* (SRS). This document must be issued by the software developer/supplier team and is part of the *technical specification* (TS).

Once the SRS has been released, two activities are developed in parallel. On the one hand, the software validation team defines the *software validation specification* (SVS) document. SVS includes the test design, cases, and procedures needed to fulfill every requirement specified in the SRS. The SVS is mainly focused on testing, as this is the preferred method for requirement validation. Review, analysis, or inspection methods can also be used when validation by testing cannot be performed. The validation process also requires validating the SSS requirements. However, the traceability between SSS and SRS requirements can be used to achieve this, so the addition of new test cases may not be necessary. The SVS, regardless of whether it is defined with respect to the SRS or the SSS, is part of the Design Justification File (DJF).

On the other hand, the engineers in charge of the design and implementation shall transform the requirements specified in the SRS into an architecture. This architecture is provided as a preliminary version of the *Software Design Document* (SDD), which is part of the Design Definition File (DDF). This document is subsequently completed with the detailed design. Both the architecture and the detailed design are traced against the SRS to ensure that all the requirements have been covered.

As noted in "Introduction," the use of MDE techniques to cover activities of the on-board software development has been addressed in other research works. The work presented in this paper focuses specifically on an unaddressed part of the development process: the automation of the Validation and Verification (V&V) of software requirements. Thus, the elements related to the design and implementation activities have not been taken into account. For these activities, our research group has developed its component-based model-driven framework called MICOBS. One of the future objectives is the integration of the V&V models into it, thus performing all the activities of the development process inside an integrated MDE environment.

As mentioned above, to complete the validation and verification process, a document called *Software Verification Report* (SVR) is also required. This document contains the definition of the verification process that has been followed during the development of the software, as well as the reports produced by it. These reports include verification control matrices that trace the software validation items against the requirements defined in both the TS and the RB. Specifically, the TS traceability is presented as a matrix of the SRS requirements against the SVS test cases and reports. At the same time, traceability of the RB is obtained by transitively tracing the SSS requirements against the ones of the SRS.

Besides, as far as feasibility is concerned, the document must contain the reports of the software validation tests. Therefore, it must include another matrix that maps the SVS tests to the evidence resulting from a successful running of the test campaign in the form of appropriate log information. Based on the traceability of the SRS from the SSS,

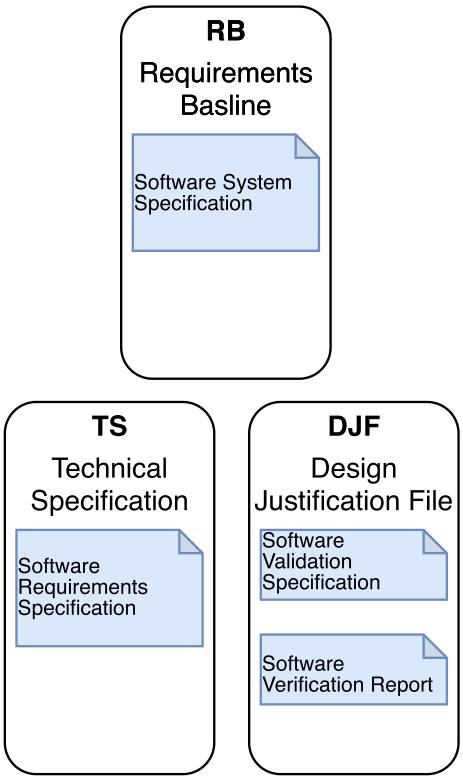


Fig. 2 ECSS documents for the RB, TS, and DJF folders

the requirements of the SSS can also be validated using the information obtained from the tests. Like this, this process goes from software to customer requirements. Figure 2 shows the aforementioned documents and the deliverable folders to which they belong.

4 Model-driven approach

The proposed model-driven approach defines one main model for each one of the documents established in the ECSS-E-ST-40 standard, namely the software system specification (SSS), the software requirement specification (SRS), the software validation specification (SVS), and the software validation report (SVR). These four models represent the contents of the documents established in the ECSS-E-ST-40 standard described in the previous section. Apart from these four models, the solution includes several other support models and model-to-model and model-to-text transformations. The main outputs of the process are the final deliverable documents and the files that contain the definition of the validation tests. The following subsections describe in detail the complete model-driven validation and verification process and the different models defined by approach. The meta-models are available online.[1]

4.1 Model-driven validation and verification process

The complete validation and verification process using the proposed model-driven solution comprises the following stages:

1. Create the SSS models with all the requirements at the requirement baseline (RB) level provided by the final customers. More than one model can be considered if customer requirements are selected from different system-level documents.
2. Using the SSS models, create a single SRS model instance containing the complete set of software requirements defined at the technical specification (TS) level.
3. Create the instance of the SVS model. This model contains the definition of the validation tests. Each test establishes references to the requirements that itself validates and that are defined in the SRS model. The SVS, in turn, refers to other models that might be either new or reused from other projects. These models are as follows:
   - Test Setup. This model describes the test scenarios. They contain the description of the configuration that has to be carried out to reproduce the environment necessary for the execution of the tests.
   - TC and TM Templates. These models contain the basic formatting data and templates needed to define the telemetry and telecommand (TM/TC) packets that can be received and sent during the execution of the tests. This information is extracted directly from the TM/TC database.
4. Create the Ground Support Software (GSS) test configuration files. These files are automatically generated from the description of the test cases established in the SVS. The transformation that allows generating the configuration files will depend on the GSS that is being used.
5. Execute the validation tests. This execution will result in the production of a set of log reports by the GSS tool. These log reports should be incorporated into the SVR. For this purpose, an intermediate model has been defined, called test campaign report (TCR). This model is used to describe the results of the validation tests. For each GSS tool, a transformation will be implemented to automati-

[1] The standard models are located in https://github.com/uah-srg-tech/verification, the document models in https://github.com/uah-srg-tech/documents, and the models for the database in https://github.com/uah-srg-tech/tmtcif. Finally, the transformation for creating the final OOXML documents is in https://github.com/uah-srg-tech/gss_log_to_tcr.

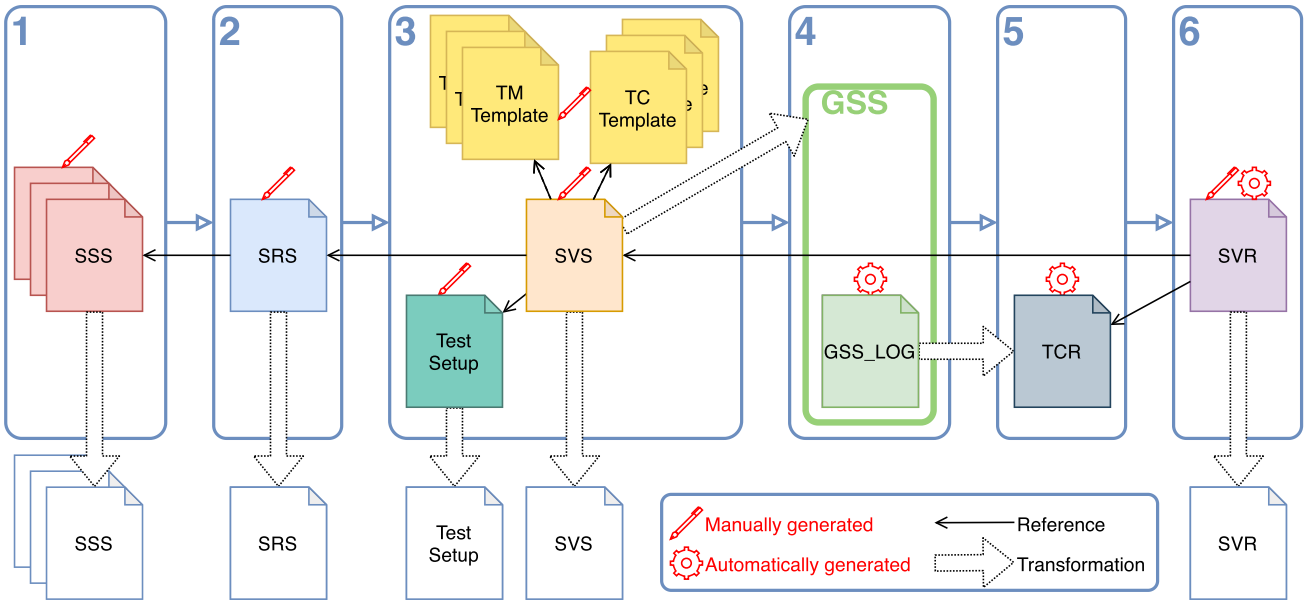


Fig. 3 Validation and verification process flowchart

cally translate the execution log reports into instances of TCR models.

6. Create the SVR using the TCR models obtained from the outputs of the test execution. This step is performed by a transformation that is independent of the GSS in use. The selection of the desired report logs is performed manually, but then the transformations themselves are done automatically. The SVS will also be used for generating the required matrices automatically.

In addition, for each of the primary documents, namely the SSS, the SRS, the SVS, and the SVR, transformations have been defined to obtain the final deliverable files in the form of OOXML documents. If the Test Setup model used has been specifically defined for this validation and verification process, it will also be necessary to generate its corresponding OOXML document using the appropriate transformation. Figure 3 shows a diagram with all the steps mentioned and the different transformations that take place, and the products generated in each case.

4.2 Common generic models

The proposed solution includes the definition of three generic models for facilitating the establishment of model relationships and the implementation of model transformations. The following paragraphs describe them in detail.

4.2.1 Document Template

The Document Template models the contents of a text-based formatted document with a minimal structure that supports the hierarchical organization of its elements. Figure 4 presents the simplified meta-model. The root class of the meta-model is DDocumentTemplate. All the document models of the approach inherit from it. It contains references to the elements commonly present in a document, such as sections, tables, or figures, and the lists of applicable and reference documents required in every document of the software development project. The transformations use these references to obtain the complete sets of elements instantiated in the final documents. The implementation of these references is done for each particular document model. It is in each of these concrete document models where the actual containment relationships between elements are established.

An Applicable Document (AD) is a binding document that defines several requirements or constraints. A Reference Document (RD) is a document taken as a source of information for the current one. The Applicable and Reference Documents of the current document are modeled using the classes DApplicableDocument and DReferenceDocument, respectively; both of them inherit from the abstract class DAbstractRelatedDocument, which has several attributes: Document title and identifier are compulsory; issue, revision, and date are required only if available.

The abstract class DAbstractSection models a section of the document. There are two types of sections: *fixed* and *instantiable*. The class DFixedSection represents the fixed type, and it models the mandatory sections whose title and structure are fixed by the very definition of the document. The instantiable sections, modeled through the class DInstantiableSection, are optional sections whose name and location within the document are established in the definition of the

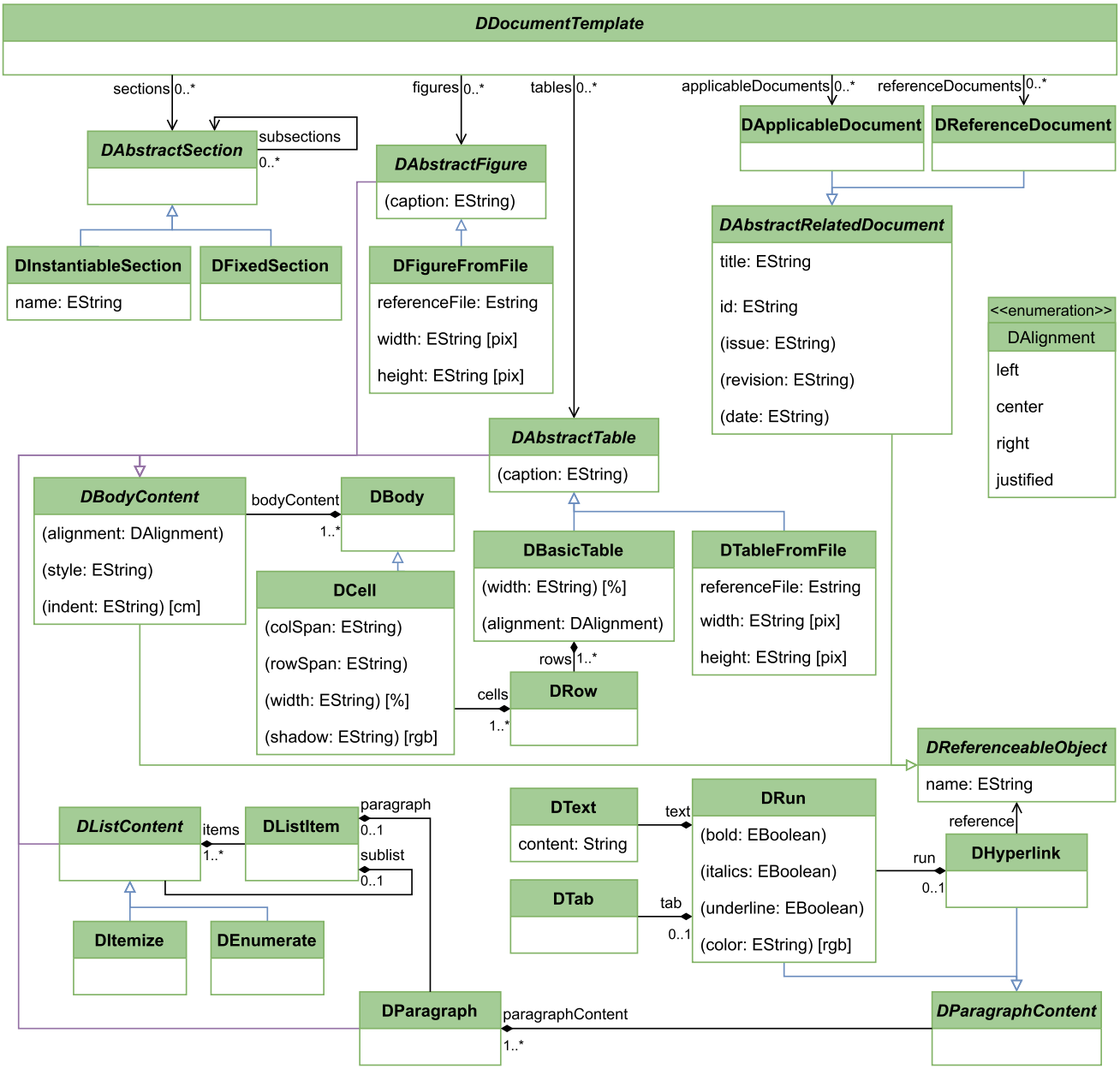


Fig. 4 Meta-model of the document template model

document model instance. Aggregation relationships can be established between sections in order to define the hierarchical organization required by each document type.

Figure 5 shows, as an example of the use of this template, a simple specifications document model that uses these features. The root class of the document is Specification, which inherits from DDocumentTemplate. The document has four fixed sections in the first level of the hierarchy, each modeled using a class that inherits from DFixedSection:

– "Applicable Documents," modeled through the class ApplicableDocsSection. This mandatory section contains the list of ADs to the current document. To model this list, the class contains a set of references to objects of the DApplicableDocument class. These objects are used to store the information related to the documents as described above.
– "Reference Documents" The class ReferenceDocsSection models the section that lists the RDs of the current document. Like in the previous case, this list is modeled by a set of references to objects of the class DReferenceDocument. These objects store the data related to the RDs.

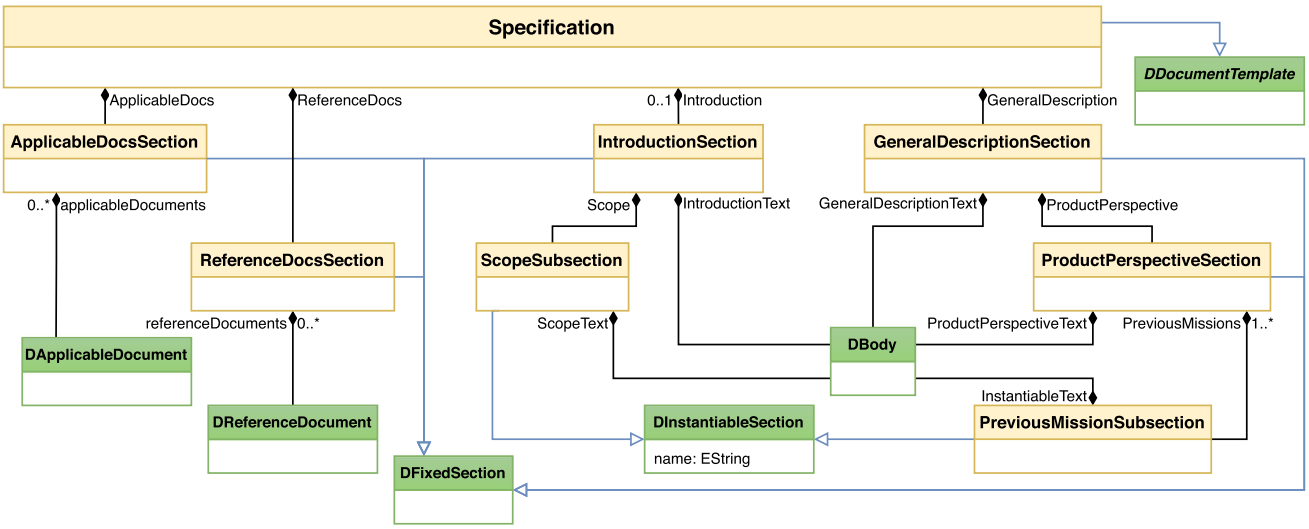


Fig. 5 Meta-model of a basic specification using sections and text

– "Introduction" The IntroductionSection class represents the introduction section of the specifications document. This fixed section contains a separate body, modeled through the class DBody, and one instantiable subsection that is modeled through the class ScopeSubsection. The body of the subsection is also modeled through the class DBody, whose description can be found below.
– "General Description" This section of the document contains a general description of the product to be developed. It is modeled using the GeneralDescriptionSection class. This class contains a body and a fixed section called "Product Perspective," which is modeled using the ProductPerspectiveSection class. This last section, in turn, contains a body and a set of instantiable sections that correspond to information about previous missions related to the one that is the subject of the current document. This section is modeled using the PreviousMissionSubsection class and only contains a body.

As mentioned above, the class DBody represents a body. A body is composed of one or more objects which can either be figures, tables, lists, or text paragraphs. All the objects contained in a body inherit from the abstract class DBodyContent. This class contains a set of attributes that are used in the generation of the deliverable OOXML document files. The first attribute sets the alignment type of the body content as one of the fields of the enum DAlignment, which includes the traditional values left, center, right, and justified. The other two attributes are style and indentation. These attributes are directly translated when generating the OOXML document. The first one is the name of the style that will be assigned to the content, and the second one is the indentation.

Figures are modeled through the abstract class DAbstractFigure. Each of the possible sources of the figure is modeled by a concrete class that inherits from it. This class contains a single attribute, which is the optional caption of the figure. Currently, the meta-model only defines one concrete class: DFigureFromFile. It models a figure obtained from an image file. It contains several attributes, such as the path to the file in the file system and the width and height of the corresponding image in pixels.

Following the same approach, tables are defined using the abstract class (DAbstractTable). This class only contains an optional attribute that models the caption of the table. Two different concrete classes have been defined. The first one is used when the table is stored in an external image file. This class, called DTableFromFile, contains the same attributes as the class DFigureFromFile. The other class is DBasicTable. It models a user-defined table and contains two optional attributes: width and alignment. The first one is used to define the relative width of the table expressed as a percentage of the total text width. The second attribute defines the alignment of the text within the table using the enumerate DAlignment, described above. User-defined tables contain rows, modeled using the DRow class, which in turn are composed of cells, represented by the class Cell. Cells can span multiple rows and columns using the rowSpan and colSpan attributes, respectively, which provides flexibility in the definition of tables. Besides, cells can also define their specific width as a percentage of the total table width and the color of the background shading.

The third type of body content is a list of items. The meta-model defines an abstract class, called DListContent, and two concrete classes, namely DItemize and DEnumerate. The first one corresponds to a bulleted list, while the second one models a numbered list. Lists are composed of items,

modeled using the DListItem class, which in turn can contain one paragraph of text and one sublist.

The last type of body content is a paragraph. They are modeled through the DParagraph class and, like bodies, must contain at least one paragraph content. Paragraph contents are modeled using the abstract class DParagraphContent. The meta-model currently defines two different types of content: text runs and hyperlinks.

A text run, modeled using the class DRun, represents a sequence of characters that are subject to a given format. This format can be set by using the boolean attributes bold, italics, and underline. A text run contains an object of the class DText which stores the actual characters. Also, a text run can contain a DTab object to model an optional horizontal tab located prior to the text.

Hyperlinks are modeled through the class DHyperlink. It includes a text run, which stores the text to be displayed on the link inside the document, and a reference to the linked object, represented by the abstract class DReferenceableObject. Hyperlinks can point to six different types of items, namely ADs, RDs, tables, figures, paragraphs, and lists.

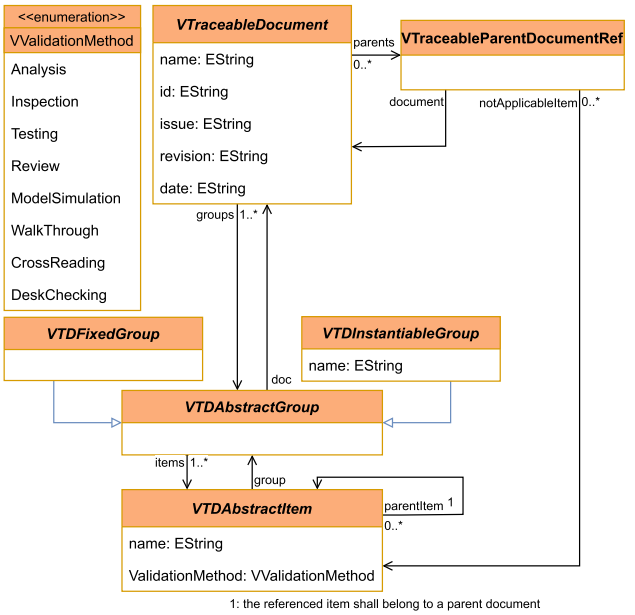


Fig. 6 Traceable document meta-model

4.2.2 Traceable document model

The second common generic model is the traceable document model (TDM), depicted in Fig. 6. It contains the classes required for defining traceable items within a document. A traceable item is an element or part of a document that can be referenced from other elements. The root class of the meta-model is VTraceableDocument. All of the document models of the approach containing requirements inherit from it. It contains several attributes for establishing the name, identifier, issue and revision numbers and date of the document, and requirement groups for sorting requirements.

Traceable documents can have parent documents. The references to these parent documents are modeled using the VTraceableParentDocumentRef class. These parent documents contain the items that can be referenced from the ones in the current document since an item cannot reference items that belong to its container document. For example, an SSS document shall be the parent of an SRS document.

It may be that a customer-defined specification document that is part of the RB contains requirements that do not apply to the software. To filter out those requirements, the class VTraceableParentDocument includes a list of non-applicable items. These items shall not be taken into account when the final relationship matrices are generated.

Referenceable items themselves are modeled using the abstract class VTDAbstractItem. Each document will define its specific classes that shall include the attributes necessary to model the concrete type of item. The generic class includes two attributes common to all item types, which are the name and the validation method. The type of the latter is an enumeration whose fields are the values defined by the ECSS-E-ST-40 standard. As mentioned above, an item can also reference other items, which shall belong to a parent document. For example, an SVS requirement item can reference an SSS specification item.

Finally, all items defined within a TDM must belong to a group. There is an abstract class VTDAbstractGroup for representing a group of items. Groups can be fixed or instantiable. Fixed groups are modeled using the class VTDFixedGroup, and they are groups with a predefined name that must always be defined in a given instance of a TDM-based model. As for instantiable groups, represented by the class VTDInstantiableGroup, they are optional groups of items whose names are established in the definition of the document model instance. For example, several fixed groups are defined in ECSS-E-ST-40 standard, such as "Capabilities requirements" or "System interface requirements," and they must be modeled using fixed groups. However, sometimes software requirements are tailored, and some requirements cannot be grouped into any group of the standard. Those requirements must be modeled using an instantiable group, as the name and meaning are project-scoped.

4.2.3 Validation document model

The third and last common generic model is the validation document model (VDM), whose meta-model is depicted in Fig. 7. It contains the classes needed for modeling documents that validate the traceable items included in other documents.

The root class of the meta-model is VValidationDocument. All the document models of the approach that contain

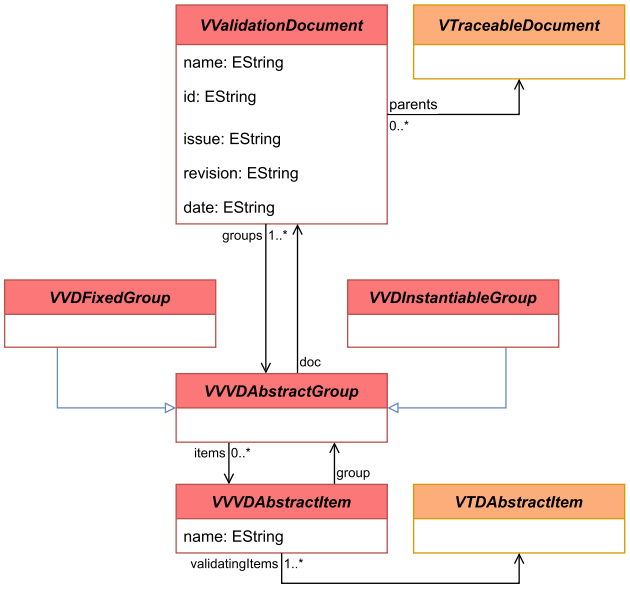


Fig. 7 Validation document meta-model

validation items, i.e., test cases, inherit from it. As with the TDM, the root class has attributes for establishing the name, identifier, issue and revision numbers, and date of the document, as well as validation groups for sorting the validation items.

A validation document can have one or more parent documents, which are necessarily traceable documents. The traceable items to be validated will be extracted from these parent documents. In the case of the ECSS-E-ST-40 standard, an SVS document shall have at least one SSS or SRS document as a parent. In our approach, an SVS document will always have an SRS document as a parent. SSS documents are always validated transitively by using a matrix that maps the requirements defined in the SRS to their corresponding SSS requirements.

The rest of the structure of the VDM is similar to that of the TDM. Validation items are modeled using the abstract class VVDAbstractItem. Final document models shall specify this class depending on the information associated with the validation procedure and the items to be validated. All validation items belong to a group that can be fixed if it is defined in the standard associated with the document and is present in all models, or instantiable if, on the contrary, it is a specific group of a particular document. The meta-model defines two abstract classes to model the groups: VVDFixedGroup for fixed groups and textsfVVDInstantiableGroup for instantiable groups.

The main validation document defined in the ECSS-E-ST-40 standard is the SVS. This document contains the test cases that comprise the software validation campaign. As shown in Sect. 4.4, these test cases will be modeled as validation items, and all of them will be within the same fixed group called "Test Cases."

4.3 Requirement models

In the ECSS-E-ST-40 standard, there are two documents that contain requirements: the software system specification (SSS) and the software requirement specification (SRS). They contain requirements at requirement baseline and at technical specification levels, respectively. Thus, SSS and SRS models have been created with all the classes required for expressing all the information contained in the corresponding documents according to the standard.

Both SSS and SRS models have the same structure. The root of the meta-models inherits from the root classes of two of the common generic models, namely DDocumentTemplate and VTraceableDocument. The remaining classes of the meta-models can be divided into two groups: those that model the descriptive text of the document and those that model the requirements themselves. The following paragraphs describe both document models in detail.

4.3.1 Software system specification model

Figure 8 contains a simplified diagram with the main classes of the SSS meta-model. The root of the model is the class called VSSSDocument. This class includes a set of objects that represent the different sections of the document. The first four sections belong to the first group described above, that is, those containing the descriptive text part of the document. They are fixed sections that present in all the document models. These sections are the following:

- "Introduction" It is used to contextualize the document and its objectives and is modeled by the VSSSIntroduction class. This section contains one or more instantiable sections, represented by the class VSSInstantiableSection, which inherits from the class DInstantiableSection defined in the Document Template model. Each of them contains a body in which you can display both text and figures and tables.
- "Applicable documents" This section contains the list of applicable documents. It is modeled using the VSSSApplicableDocumentsSection class. In turn, this class contains the references to the documents, modeled by the class DApplicableDocument defined in the Document Template model.
- "Reference documents" Modeled by the VSSSReferenceDocumentsSection class, this section contains the list of reference documents. The references to the documents are modeled using the DReferenceDocument class, defined in the Document Template model.

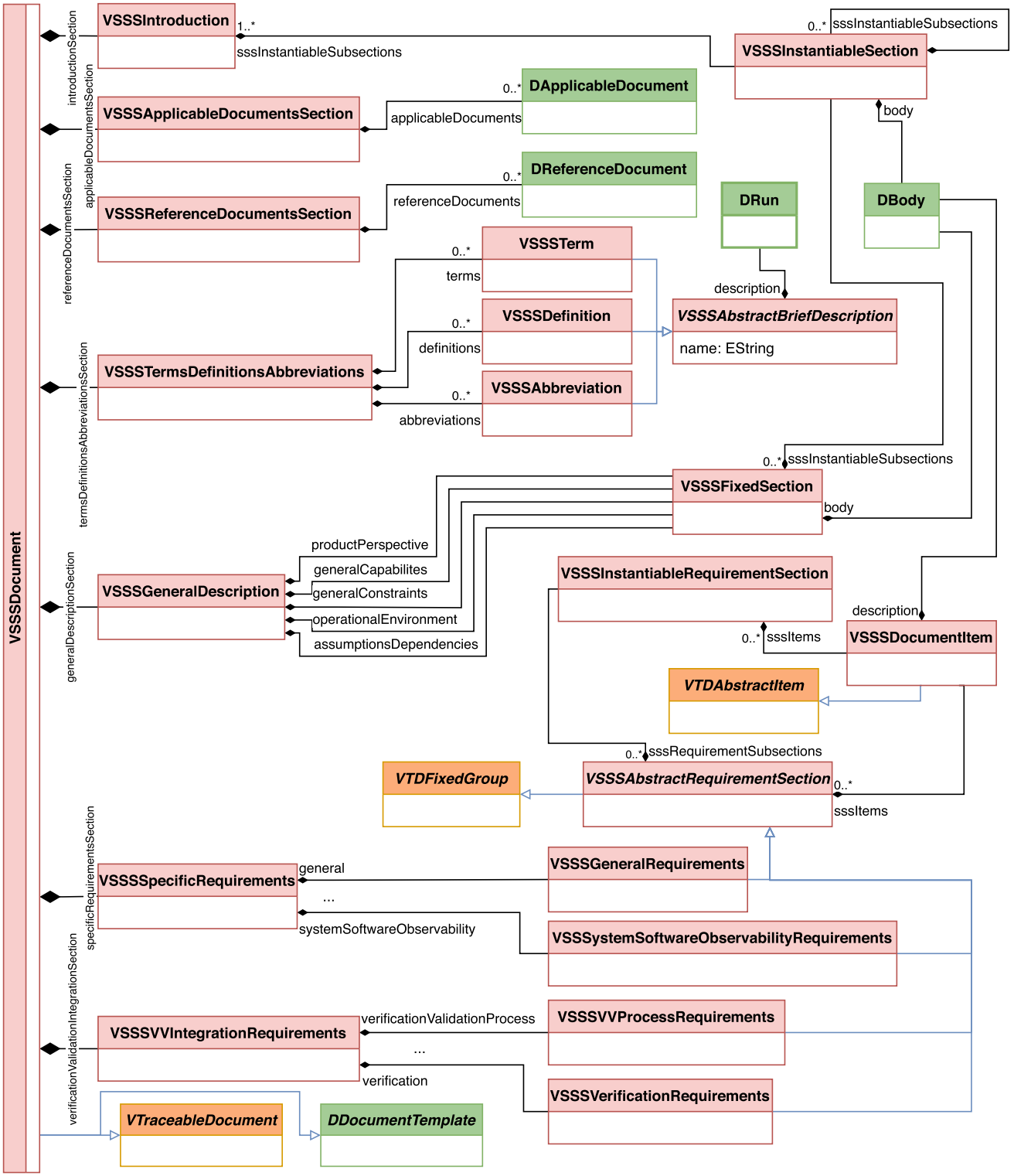


Fig. 8 Meta-model of the software system specification model

– "Terms definitions and abbreviations" This section contains a list of the terms, definitions, and abbreviations used during the definition of the document. It is modeled using the class VSSSTermsDefinitionsAbbreviations. The items themselves are finally modeled as name/description pairs through concrete classes that inherit from the abstract class VSSSAbstractBriefDescription.

The class VSSSGeneralDescription models the "General description" fixed section. This is the main text section of the document. It describes, from the customer point of view, the main capabilities and constraints of the software, its operational environment, and any dependencies that may affect it. According to this, it contains several fixed subsections defined in the standard, namely "Product perspective," "General capabilities," "General constraints," "Operational environment," and "Assumptions and dependencies."

The standard defines two types of requirements in the SSS: the *specific requirements* and the *verification, validation, and integration requirements*. To keep this distinction in the model, there are two different fixed sections, one for the specific requirements, modeled using the class VSSSSpecificRequirements, and another one for the verification, validation, and integration requirements, represented by the class VSSVVIntegrationRequirements.

Each of these sections, in turn, is divided into fixed subsections. These subsections, most of which have been removed from the diagram for simplicity, group and organize the various requirements according to the ECSS-E-ST-40 standard. Also, they act as fixed groups according to the Traceable Document model. The requirements are modeled using the class VSSSDocumentItem. This class in turn inherits from the class VTDAbstractItem, making the requirements traceable items. Each of these items contains a body that can include text paragraphs, figures, or tables as appropriate for the definition of the different requirements.

4.3.2 Software requirement specification model

A simplified version of the SRS document meta-model is depicted in Fig. 9. The root class is called VSRSDocument, and its structure is similar to that of the SSS. It contains the same four initial sections described above, namely "Introduction," "Applicable Documents," "Reference Documents," and finally "Terms definitions and abbreviations."

The main text section of the SRS as defined in the standard is "Software overview." This section describes the main functionality and purpose of the software, its interfaces, and operational environment, and also any constraints that may affect it. This section is modeled through the class VSRSSoftwareOverview. This fixed section, in turn, includes four other fixed subsections whose names and scope are also stated in the standard.

These sections are: "Function and purpose," "Environmental considerations," "Relation to other systems," and "Constraints." Each of these subsections is modeled as fixed sections using classes that inherit from the ones defined in the Document Template model. All of them define their own bodies which may include paragraphs of text, figures, or tables.

The SRS does not distinguish, at the top level, between specific requirements and verification, validation, and integration requirements, as does the SSS, so only one type of requirement is defined at this level. Thus, the model includes a single fixed section to include them, which is modeled using the class VSRSRequirements. This fixed section is divided into several fixed subsections, most of which have been removed from the diagram for the sake of simplicity.

Each subsection groups a specific set of requirements. As in the case of the SSS, SRS requirements are modeled also as traceable items inheriting from the class VTDAbstractItem defined in the TDM. These items also include their own bodies that can include text paragraphs, figures, or tables.

According to the ECSS-E-ST-40 standard, the SRS document must include a requirements traceability matrix. This matrix must link every requirement present in the SSS documents to a requirement in the SRS. However, one of the targets of our approach is the automatic generation of this matrix with information from the SSS and the SRS models. After being generated, this matrix is incorporated into the final deliverable SVR document.

4.4 Validation models

The ECSS-E-ST-40 standard defines the software validation specification (SVS) as the main document of the validation process. This model behaves like a domain-specific language for defining test procedures according to the ECSS standard. It also integrates the telemetry and telecommand information of the project, provided via a SCOS-2000 database. This way, the definition of the test procedures is done by establishing the sequence of the required TCs with the related expected TMs, according to the database, and adjusting the field values where appropriate.

A simplified diagram of the SVS meta-model is depicted in Fig. 10. This diagram only represents the classes corresponding to the sections that contain text and the ones that model the test designs and cases. Another figure below shows the diagram corresponding to the specific classes that model the test procedures.

SVS document models incorporate elements from both the Document Template and Validation Document models. The root class of the SVS document meta-model is called VSVSDocument. This class inherits from classes DDocumentTemplate and VValidationDocument.

The document contains an initial set of four fixed sections which have the same structure and semantics as the ones described in Sect. 4.3. These sections are "Introduction," "Applicable Documents," "Reference Documents," and "Terms definitions and abbreviations." There are also five other fixed sections that provide information about the software to validate, the available validation methods, the test platforms that are going to be used, the way to identify the tests, and any other additional information that must be taken in account during the testing campaign. These sections con-

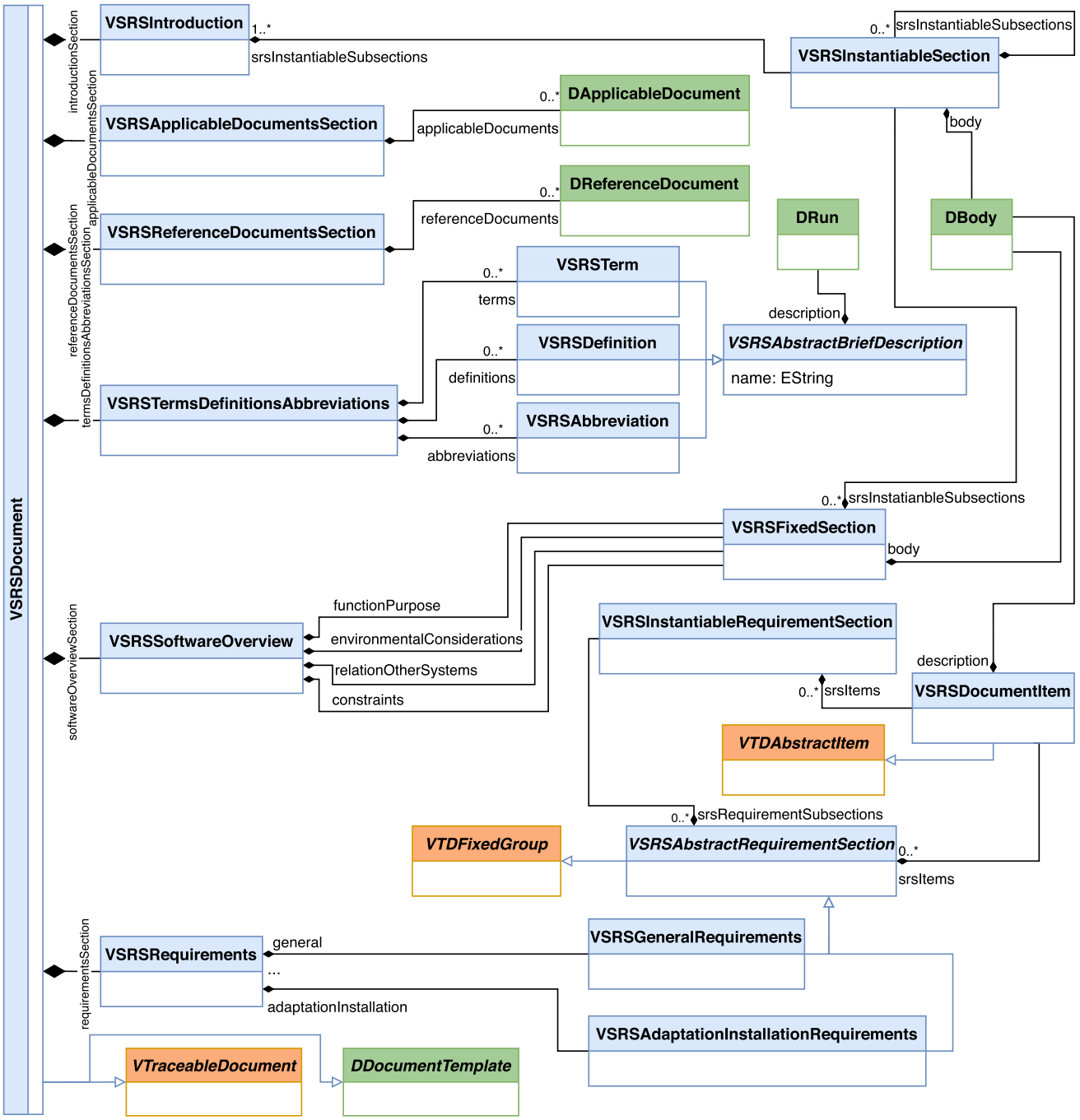


Fig. 9 Meta-model of the software requirements specification model

tain only descriptive text and are unique to this document. All these fixed sections and their corresponding subsections are modeled using the class VSVSFixedSection, which uses the classes provided by the Document Template model to define the body of the sections as objects of the DBody class.

The last three fixed sections of the document are:

– "Software validation testing specification design" It specifies the test grouping criteria and a general description of the design of the validation tests.

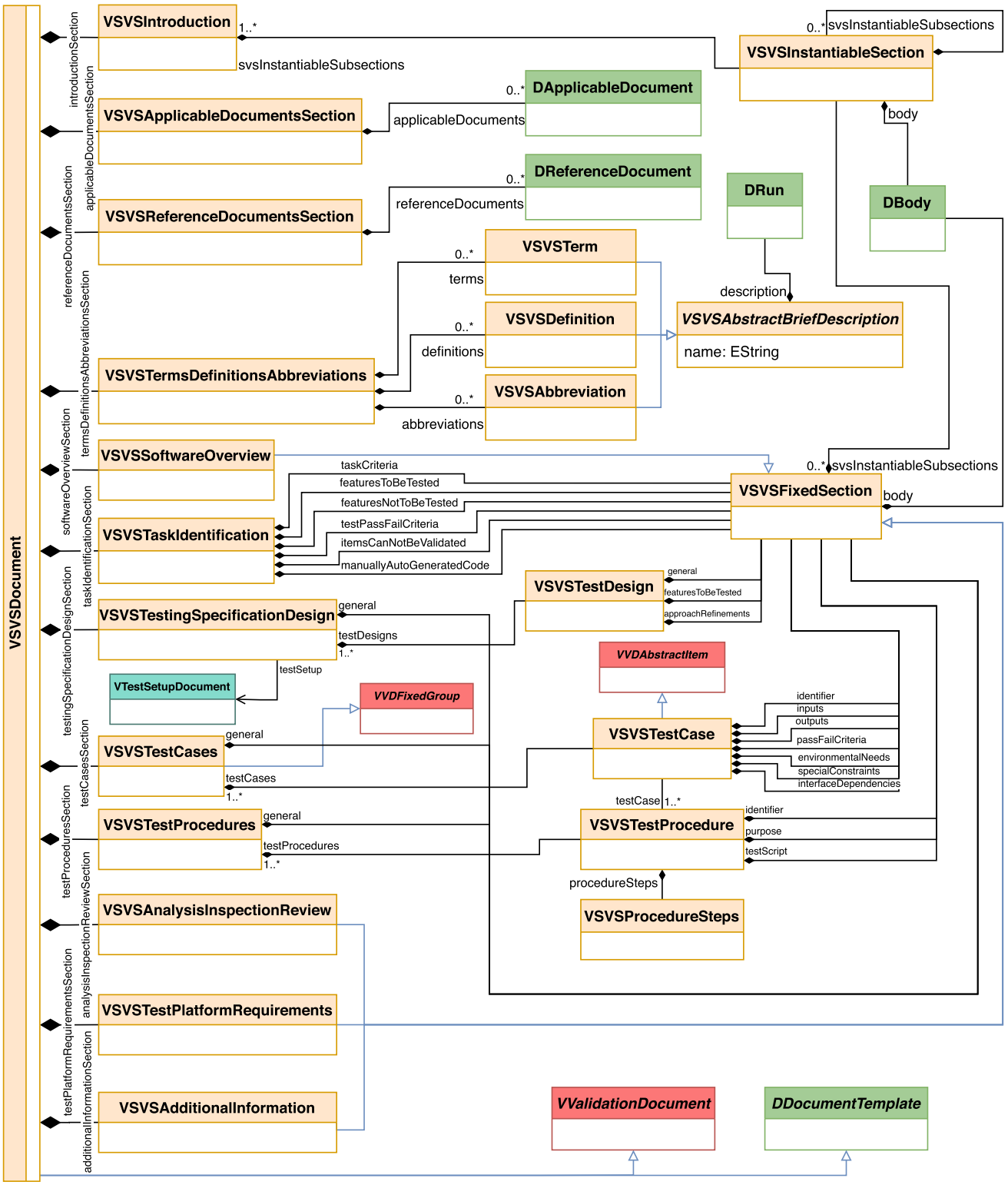


Fig. 10 Meta-model of the software validation specification document model

– "Test cases" It defines the inputs and outputs of the test, the pass-fail criteria, and any environmental needs.

– "Test procedures" It defines the steps to follow for the execution of the tests.

In the SVS document meta-model, the first section is modeled through the class VSVSTestingSpecificationDesign. It references an object of the class VTestSetupDocument, which is defined as part of the Test Setup model and described in Sect. 4.5. Test designs themselves are represented by the class VSVSTestDesign, which contains the general description of the design, the features to be tested, and the refined description of the testing approach.

The "Test cases" fixed section is modeled through the class VSVSTestCase. This section is implemented as defined in the related ECSS standard ECSS-E-ST-40C, annex L [1]. It contains a fixed subsection, called "General description" that includes a description of the global aspects of the different test cases. Test cases themselves are represented by the class VSVSTestCase. Each test case refers to one or more traceable items corresponding to the requirements validated by every particular case. Inside the test case section, several pieces of information must be present: the *identifier*, which is composed of a unique name and a short description; *inputs and outputs*, which are the elements needed to execute the tests and the elements expected afterward; the *pass or fail criteria*; the *environmental needs*, containing configurations of both setup and support software versions; the *special constraints* if any; and the *interface dependencies*, i.e., all the test cases to be executed before the current one.

Test procedures are grouped inside the section called "Software validation test procedures," modeled using the class VSVSTestProcedures. This section includes a general description and at least one test procedure. Each procedure, represented by the class VSVSTestProcedure, contains fixed text sections: a unique *identifier*, different from the test case identifiers; the *purpose* of the procedure, including a reference to the related test case; the *test script*, which can be collected in Appendix, and at least one procedure step. Each procedure can optionally reference an object of the class VTestSetupSupportedActionOverVariable which belongs to the Test Setup model and is detailed in Sect. 4.5.

Steps are modeled by the class VSVSProcedureStep. The class uses several auxiliary models for creating the steps. For its definition, it may use telecommands and telemetries from the database or actions taken from the ones defined in the Test Setup model associated with a given scenario. Figure 11 shows a diagram of the classes relevant to the definition of the procedure steps.

The standard defines the validation procedures as sequences of inputs (TCs) and outputs (TMs) that fit with the Arrange–Act–Assert paradigm; thus, this is the most important part of the procedure steps model. Inputs and outputs are modeled using abstract classes VSVSStepInputs and VSVSStepOutputs, respectively.

The input for each step can be of one of the following types:

- *A telecommand sequence* It is an ordered sequence of telecommands that are to be sent in the corresponding step. This input type is modeled through the concrete class VSVSStepTelecommandSequence.
- *An action*. It models the interaction between the test and the test conductor. For example, the test conductor has to check whether a voltage value is not bigger than a given limit for 30 seconds. Actions are represented by the class VSVSStepAction.

Telecommand sequences are composed of one or more telecommands. Each telecommand is modeled using the class VSVSStepTelecommand, which contains attributes that allow you to set a name for the telecommand to be sent, and apply an optional delay before sending. In addition, the class defines a reference to an object of the VSVSTestSetupSupportedInterface class. This class, defined as part of the Test Setup meta-model, identifies the interface through which the telecommand should be sent. The interfaces must have been defined in the Test Setup, and they are the ports used to receive and send telecommands and telemetries. Finally, the telecommand meta-model allows you to set both its header and the payload to be sent.

The telecommand header has a fixed structure that is determined by the Consultative Committee for Space Data Systems (CCSDS) standard [43]. The abstract class TMTCIFTCHeader represents the header. It contains a series of fields, modeled by the class TMTCIFTCHeaderField, whose values are fixed by the standard. These classes must be specified depending on the GSS tool used, adding in each case the necessary attributes to generate the test configuration.

Within the telecommand step model, the content of the fields can be modified explicitly to produce erroneous headers. In this way, the software routines in charge of Fault Detection Isolation and Recovery (FDIR) can be validated. These assignments are defined through the class VSVSStepTelecommandHeader.

The organization of the payload follows the Packet Utilization Standard (PUS) [44]. This standard sets both the fields of the payload and its format. Each mission establishes a database with the telemetry and telecommand packet to be used. From this database, and following the standard, the structure of each packet is defined. The abstract class TMTCIFTC represents this structure. Each object in this class represents a specific telecommand packet. For each possible GSS tool to be used, this class must be specified with the necessary elements and attributes to automate the generation of TC packets. Telecommand and telemetry models are stored externally to the SVS document definition itself and are generated from the information contained in the TM/TC database.

The SVS document model uses templates, stored in separate models as well to facilitate reuse, for assigning values to

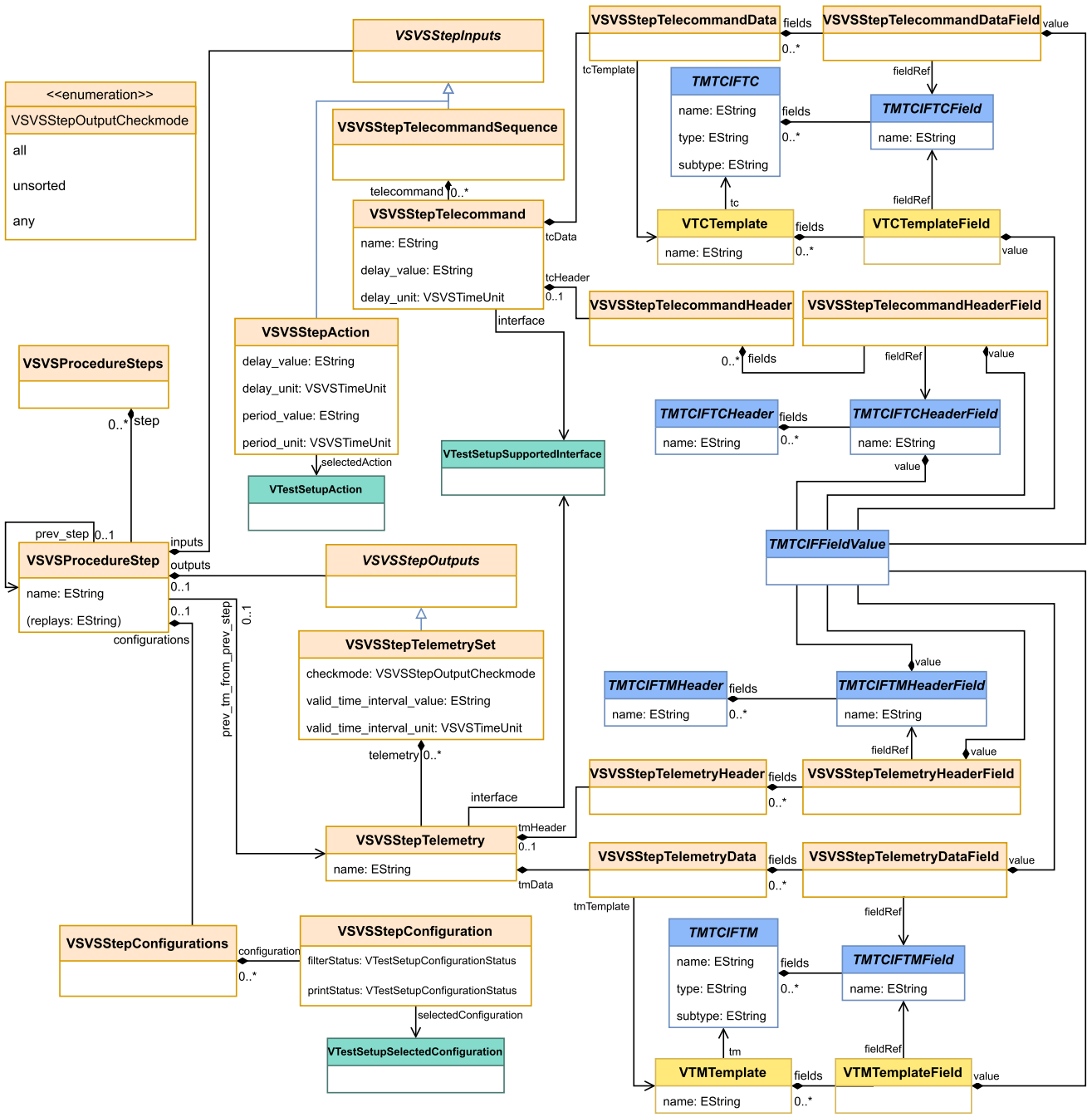


Fig. 11 Meta-model of the procedure steps of the software validation specification model

the fields of the telecommand packets. Templates are modeled by the VTCTemplate class. This class belongs to the TC Template meta-model.

When you define a telecommand step, you establish which template is going to be used. Templates can assign values to all the fields of the payload or leave some of them open to be defined in the step itself. The class VSVSStepTelecommandData is used to make the assignment of values to these open fields. In this way, the same template can be used in multiple steps.

As mentioned above, the other type of input that can be part of a procedure step is an action. Actions are represented by the class VSVSStepAction. For defining an action inside a procedure, an object of the class VTestSetupAction must be referenced. This class and its contents belong to the Test Setup model and are explained below. Actions allow the def-

```
<ProcedureSteps>
    <Step name="TestConnection">
        <TelecommandSequence>
            <Telecommand name="TestConnectionTC" interface="SpW" delay_value="120" delay_unit="
    milliseconds">
                <TelecommandData template="tc_17_1" />
            </Telecommand>
        </TelecommandSequence>
        <TelemetrySet checkmode="all" valid_time_interval_value="500" valid_time_interval_unit="
            milliseconds">
            <Telemetry name="AcceptACK" interface="SpW">
                <TelemetryData template="tm_1_1" />
            </Telemetry>
            <Telemetry name="TestConnectionReport" interface="SpW">
                <TelemetryData template="tm_17_2" />
            </Telemetry>
            <Telemetry name="ExecACK" interface="SpW">
                <TelemetryData template="tm_1_7" />
            </Telemetry>
        </TelemetrySet>
    </Step>
</ProcedureSteps>
```

Listing 1 XML script example of VSVSProcedureStep

inition of two timing attributes: delays, defined as time to wait before the action should begin, and spans, which is the time span during when the action should take place.

A procedure step may produce outputs, which are modeled through the abstract class VSVSStepOutputs. Currently, there is only one concrete type of output available, a set of expected telemetries. Expected telemetry sets are modeled using the class VSVSStepTelemetrySet. This class is similar to the input telecommand sequence. The telemetry set contains an attribute for expressing the expected telemetry order inside the set, which is an enumeration modeled using the class VSVSStepOutputCheckmode. The enumeration fields are: "all," which means all telemetries in the set must be received and the order has to be the one in the procedure; "unsorted," for expecting all the telemetries but in any order; and "any," for receiving at least one of the telemetry packets.

Telemetries are modeled by the class VSVSStepTelemetry. The telemetry model is identical to that of the telecommand. It also allows you to define both the header of the packets and their payload. Telemetry headers are set by the CCSDS standard and modeled using the same classes described above. The payload follows the PUS standard, and the assignment of values to the fields is also done using templates. The class TMTCIFTM represents the structure of a telemetry packet. As mentioned above, these objects are stored in separate files and are generated from the TM/TC database. Telemetry templates are modeled by the VTMTemplate class. This class belongs to the TM Template meta-model. These templates are also stored in separate files to facilitate reuse.

A simple test procedure is defined in the XML script listing in 1 . As shown in the model, the XML root element for test procedure steps is always VSVSProcedureSteps. For this simple test, a single step named "TestConnection" is defined. It contains a telecommand sequence with only one telecommand. This single telecommand, named "TestConnectionTC," is configured to be sent via the SpaceWire "interface," identified by the key "SpW," which is a space communications protocol coordinated by ESA. The telecommand is configured to be sent 120 milliseconds after starting the procedure with the "delay_value" and "delay_unit" attributes. For defining this TC, the TC "template" used is a test connection telecommand, which is identified in the PUS standard as type 17, subtype 1, and so is called "tc_17_1." An example of the TC template and its corresponding format is shown in Listing 2.

```
<TCTemplate name="tc_17_1" tc="tc_epd_17_1_ack"
    />

<Export from="epd_pus_tc_format.xml"
to="epd_pus_tc_17_1_format.xml">
    <settings>
        <settingFromConst value="9" toFieldRef=
    "ACK"/>
        <settingFromConst value="17" toFieldRef
            ="ServiceType"/>
        <settingFromConst value="1" toFieldRef=
            "ServiceSubtype"/>
        <settingFromConst value="120"
            toFieldRef="SourceID"/>
    </settings>
</Export>
```

Listing 2 XML script example of TC Template type 17 subtype 1 and its corresponding format

For the expected telemetries, this simple test script configures a single step with one telemetry set containing three different telemetries, which are the expected responses according to the PUS standard: two acknowledge packets and the test connection response. Following the check mode keyword, "all" the three packets must be received mandatorily. Moreover, as stated by the attributes "valid_interval_value" and "valid_interval_unit," they must all be received within 500 milliseconds.

The first packet is the acknowledgment of acceptance and corresponds to packet type 1, subtype 1 of the PUS

standard. It is called "AcceptACK," and it is received via the same "SpW" interface and created using the TM template "tm_1_1." The other two telemetries correspond to the test connection report (PUS type 17, subtype 2) and the acknowledgment of execution (PUS type 1, subtype 1). They are called "TestConnectionReport" and "ExecACK," are received through the same "SpW" interface, follow the same structure, and use the TM templates "tm_17_2" and "tm_1_7," respectively.

4.5 Test Setup model

The Test Setup model contains all the classes needed to define a Test Setup document. It is worth recalling that this document is not in the ECSS-E-ST-40 standard. The reason for modeling it, even though it is not included in the standard, is to allow the reuse of configuration files for the testing process of different projects.

Figure 12 contains a simplified diagram with the main classes of the model. The root class is VTestSetupDocument. The meta-model incorporates the elements from the Document Template model, and its root class inherits from DDocumentTemplate. The document contains the same initial set of four fixed sections that are present in all the requirement and validation documents, i.e., "Introduction," "Applicable Documents," "Reference Documents," and "Terms definitions and abbreviations." Besides, it contains four other fixed sections that describe the features and options that define a test setup.

The first fixed section is called "Interfaces" and is modeled using the class VTestSetupInterfacesSection. It contains the description of the interfaces available within the test environment. Each interface is modeled through the class VTestSetupInterface. This class contains the name of the interface and a description.

The second fixed section of the document is "Packet configuration." It is an optional section, modeled through the class VTestSetupPacketConfigurationsSection, and it contains one or more packet configurations. A packet configuration is a feature for filtering certain telemetry packets out of the test flow. One of the uses of this mechanism is to make the housekeeping telemetry packets, that can arrive at every moment, not interfere with the test. Any telemetry can be selected, thus making easier the automation of the test procedures.

The third fixed section is "Actions," which is another optional section and is modeled using the class VTestSetupActionsSection. It contains one or more actions. An action, represented by the class VTestSetupAction, is an operation to be performed by the test conductor and not automated by the test. These actions have a name, a type, and a description associated.

The type is modeled using the enumeration VTestSetupActionType, whose values have been defined according to our experience in the development of validation tests. The following list shows the currently available values, although the model can be extended by adding other values if necessary:

- *instruction*, a single instruction given to the test conductor like "turning on any external device."
- *checking*, a requirement for the test conductor to check values in an instrument such as an oscilloscope.
- *tmtc_checking*, a checking related to telecommands or telemetries, can be automated depending on GSS.

The fourth and final section is called "Scenarios," and it is modeled using the class VTestSetupScenariosSection. It contains a list with at least one scenario. Each scenario relies on all the previous four optional features for creating a full-defined environment for testing. Thus, it contains the list of interfaces supported by the scenario, the list of packet configurations to be applied, and the list of actions available for the different test procedures.

4.6 Report models

The approach provided in this paper includes the definition of a report document model called Test campaign report (TCR) and the modeling of the software validation report (SVR) document. Both document models are described in the following paragraphs.

4.6.1 Test campaign report model

The TCR is not a document explicitly defined in the ECSS-E-ST-40 standard. It contains the results of the execution of the validation test campaign. In this way, the document is part of the Design Justification File (DJF). Specifically, it contains the results of the validation tests. It is intended to be generated automatically from the execution logs of the Ground Support Software tool used to launch and manage the tests. Figure 13 contains a simplified diagram with the main classes of the model.

The root class of the TCR meta-model is VTCRTestCampaignReport. It contains a list of test reports, modeled by the class VTCRTestReport. Each report has two attributes: the identifier of the test procedure associated with the report and the resulting status of the test. This status will have as value one of the following: *pass* if the test was run successfully, *fail* if the test failed to obtain the expected result, and *not tested* if the test was not run in this particular campaign. Reports also contain a text string with the evidence for the status, e.g., the date and time when the test was successfully run.

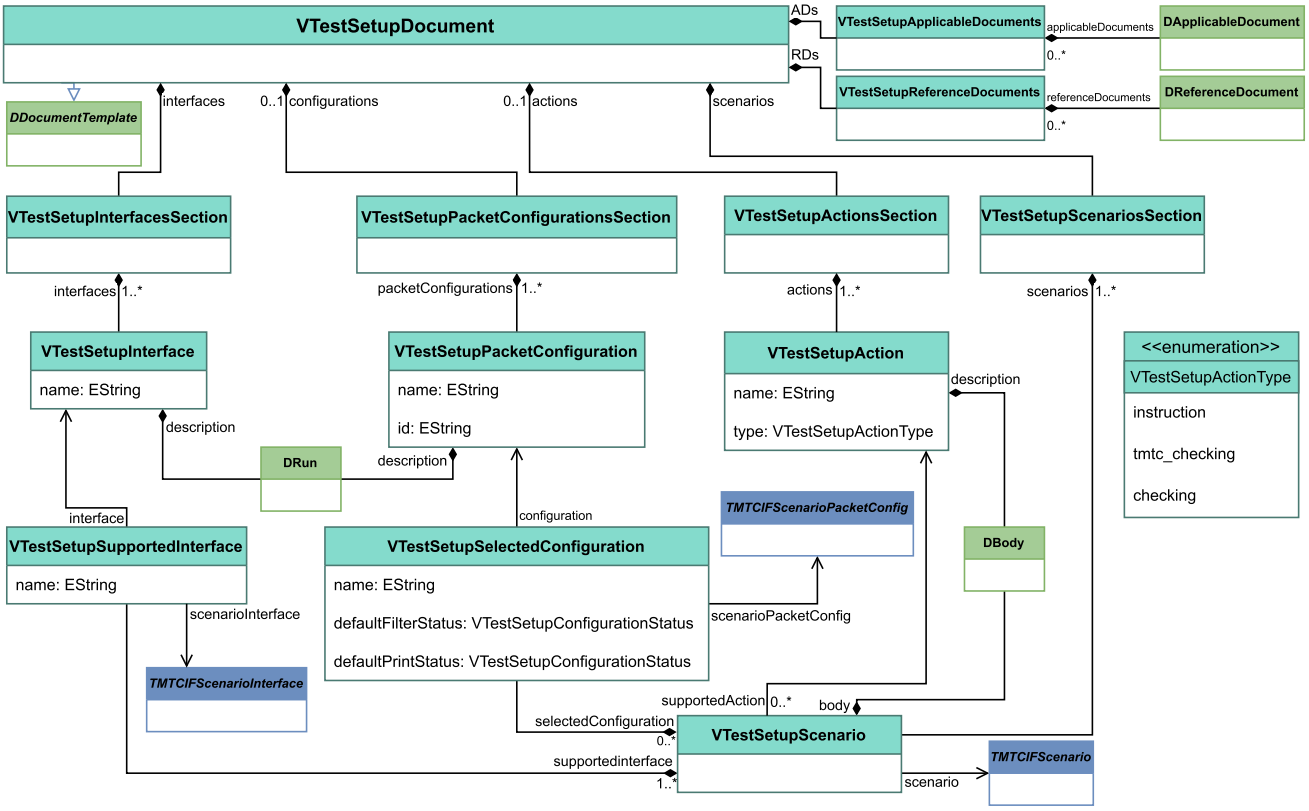


Fig. 12 Meta-model of the test setup model

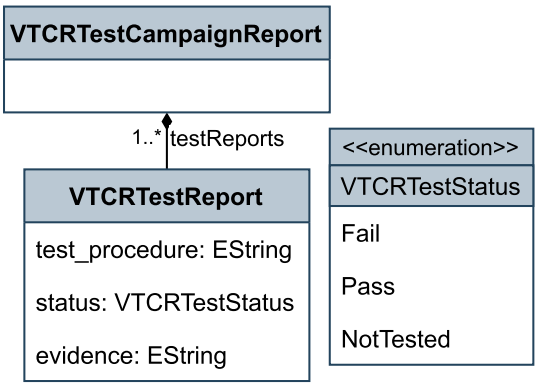


Fig. 13 Meta-model of the test campaign report model

4.6.2 Software validation report model

The SVR model contains all the classes needed to perform the definition of an SVR document. Figure 14 contains a simplified diagram with the main classes of the meta-model. The root of the model is the class VSVRDocument. Like all document models, this class inherits from DDocumentTemplate. The document shares the initial set of fixed sections, namely "Introduction," "Applicable Documents," "Reference Documents," and "Terms definitions and abbreviations," with the rest of the documents. The root class references the parent SVS document model containing the definition of the validation tests and the TCR models that incorporate the results of those tests.

The main content of the document is located in the section called "Software validation process verification." This section contains the verification matrices for the traceability of software requirements to system requirements and from the software requirements to the test cases, and for the feasibility, including the test reports.

The SVR document model does not explicitly model this section, since its contents are automatically generated from the information stored in both the SVS and the TCR models. These contents shall be directly incorporated into the deliverable document produced as a product of the corresponding transformation.

This information, generated automatically, allows you to close the verification cycle. The required test configuration for the Ground Support Software is automatically generated from the requirement and test definitions in the SSS, SRS, and SVS documents.

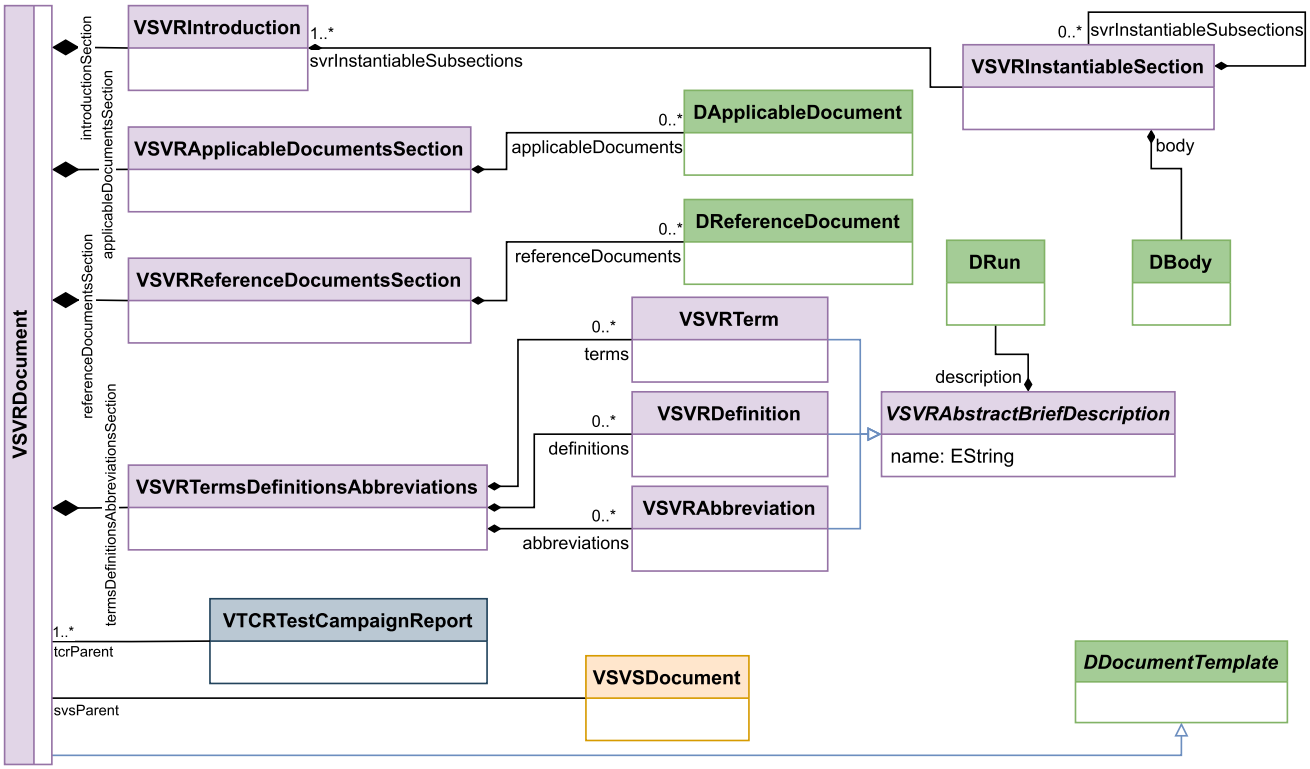


Fig. 14 Meta-model of the software validation report model

Furthermore, the results of the test execution are incorporated into the SVR verification matrices by means of TCR models. In this way, the verification process is accelerated, as the required final evidence is automatically obtained and traced from the beginning.

4.7 Implementation of the proposed approach

Figure 15 shows a general outline of the models and their relationships, together with the different transformations and the products that are obtained.

For the implementation of all the models, we have used the Ecore meta-model defined within the eclipse modeling framework (EMF) [45]. We have also used Xtext [46] to generate textual representations and editors that allow the definition of the different model instances. We have generated editors for all the document models of the approach.

The complete set of transformations can be divided into two groups. The first group corresponds to the transformations needed to produce deliverable documents in Office Open XML (OOXML) format [47]. These documents are to be delivered as part of the folders established by the standard ECSS-E-ST-40. To maximize the reuse of the transformations, we have defined an intermediate model called Document (DOC). This model inherits from the Document Template model and contains the classes necessary to produce an OOXML file directly. Thus, these models include only the sections and bodies that will make up the final documents without any other associated semantics. These intermediate documents are automatically generated from the SSS, SRS, SVS, SVR, and Test Setup document models through specific model-to-model transformations implemented in the QVT operational language (QVTo). Finally, an additional model-to-model transformation generates the final OOXML documents. This last transformation, independent of the original document models, is implemented in C/C++. In total, 3,165 lines of QVTo and 4,228 lines of C/C++ have been used to define the complete set of transformations.

The second group of transformations is related to the generation of the input models of the Ground Support Software (GSS) tool and the incorporation of the log files resulting from the execution of the tests. In this case, we have integrated a GSS tool called SRG-GSS (Space Research Group—Ground Support Software). This tool was designed and implemented from scratch by our group during the development of the Instrument Control Unit (ICU) software of the Energetic Particle Detector (EPD) on-board the Solar Orbiter mission to enable automation of the execution of the validation tests. It uses as inputs a set of models serialized in XML format. These models define the test procedures to

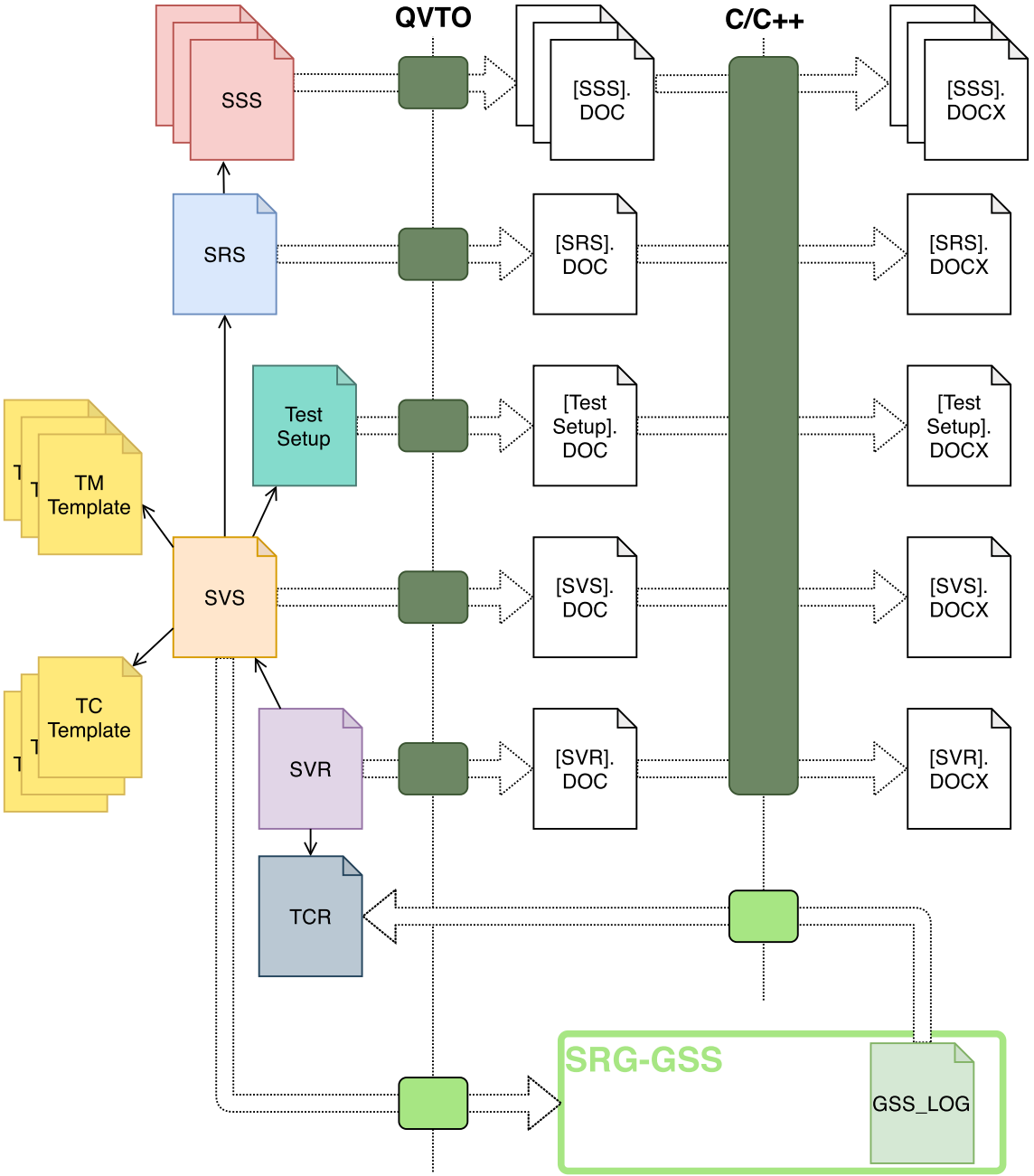


Fig. 15 Models, transformations, and products

be executed as sequences of steps. These steps contain the telecommands to be sent and filters that define the expected telemetries. The execution of the tests is done automatically, and the tool itself generates logs in plain text containing the test results. Figure 16 shows the main interface of this tool.

We have defined a model-to-model transformation implemented in QVTo that allows obtaining, from the SVS definition and the Test Setup, the input models of the SRG-GSS tool. Moreover, a second transformation, implemented in C/C++ allows creating the instances of the TCR models needed for obtaining the final SVR document. This transformation uses as inputs the log report files obtained by the SRG-GSS tool. The selection of the required log reports to be used as test evidences is performed manually, but the transformation itself and the generation of the final SVR document are done automatically. In total, 883 lines of QVTo and 309 lines of C/C++ have been used to define the complete set of transformations.

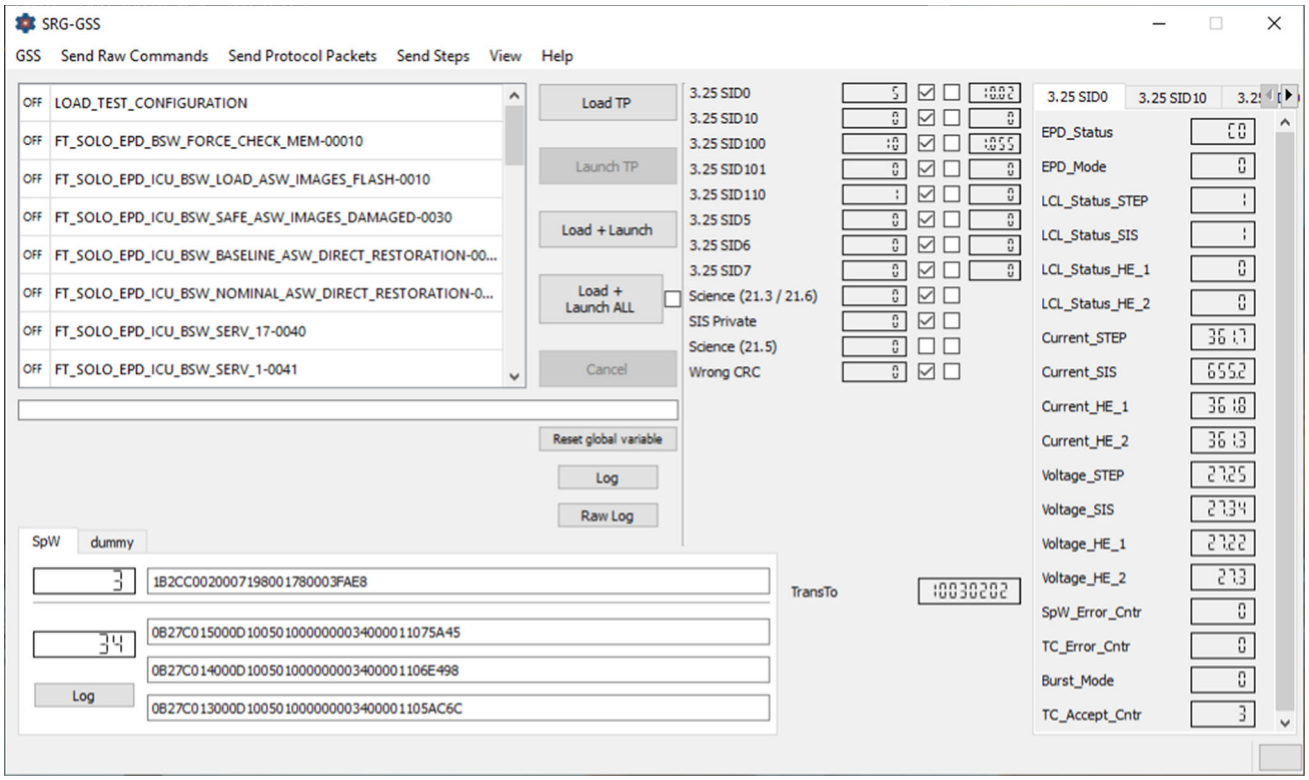


Fig. 16 SRG-GSS

5 Proof of concept

The Space Research Group of the University of Alcalá (SRG-UAH) developed the on-board software of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) on-board the Solar Orbiter mission. This development was a complex process that involved a significant number of requirements. During this process, SRG-UAH used a model-driven component-based approach for the design and deployment of the application software supported by the MICOBS framework [29,32,48].

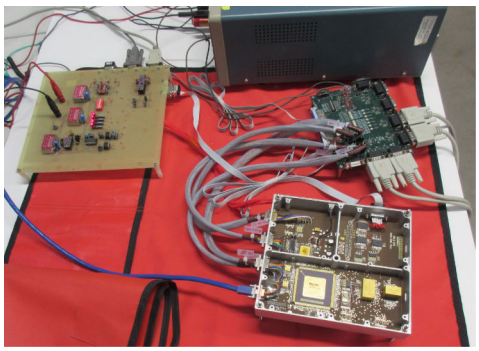
A large number of validation tests were needed to cover both the requirements and the whole set of supported telemetry and telecommand packets. The execution of these validation tests was automated with the aforementioned SRG-GSS (Space Research Group—Ground Support Software) tool.

Several scenarios were used for the original validation of the software of the ICU of EPD. Figure 17 shows three of these scenarios that were actually used in the validation campaigns. The scenarios shown in Fig. 17a and b were designed for the early validation of the software and also for the tests that involve the injection of failures in the behavior of the sensors. The two of them use an engineering model of the ICU connected via SpaceWire to the SRG-GSS. In both cases, several of the instrument's sensors are emulated using the SRG-GSS through UART ports. The scenario shown in Fig. 17a has all four sensors emulated by the SRG-GSS. Then, the one shown in Fig. 17b uses two hardware sensor emulators, while the SRG-GSS emulates the other two.

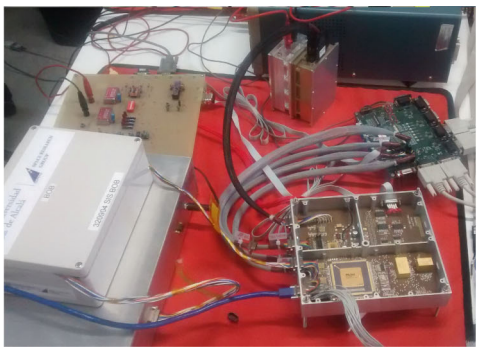
Finally, the scenario displayed in Fig. 17c is based on the Engineering Qualification Model (EQM) of the ICU. It also has a SpaceWire port connected to the SRG-GSS, and it was used to run the black-box validation tests of the software using the EQM sensors.

During the software development process, we used a document-based approach for managing requirements, tests specifications, and generating the different traceability matrices. This approach, however, was error-prone and not specifically designed for these purposes. There are tools that significantly improve the management of requirements, such as the previously mentioned Rational DOORS [3] or Sparx Systems Enterprise Architect [4]. However, none of them supports the automatic configuration of the SRG-GSS to generate the validation tests nor incorporate automatic feedback from the reports to complete the verification matrices.

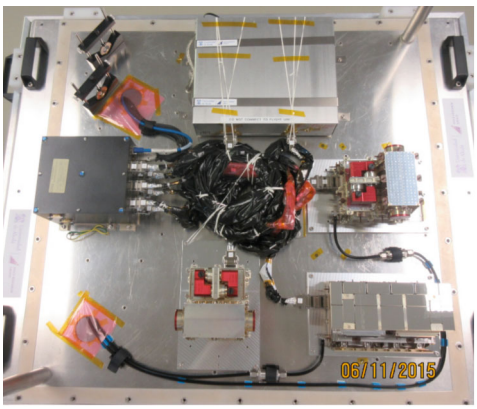
The proof of concept of the model-driven approach presented in this paper has been built from a subset of the requirements and validation tests of the on-board software of the ICU of EPD. We have been able to generate as products the same validation tests that were used for its qualification.



(a) EPD ICU Test Scenario with 4 emulated sensors



(b) EPD ICU Test Scenario with 2 real and 2 emulated sensors



(c) EPD EQM ICU Test Scenario

Fig. 17 EPD EQM ICU test scenario

Table 1 Validation and verification metrics of the EPD's ICU software and the proof of concept

Total number of SSS requirements	351
Total number of SRS requirements	532
Total number of validation tests	212
Total number of supported telemetry packets	318
Total number of supported telecommand packets	118
Selected SSS requirements for the proof of concept	51
Selected SRS requirements for the proof of concept	24
Selected validation tests for the proof of concept	21
Selected telemetry packets for the proof of concept	10
Selected telecommand packets for the proof of concept	17

The solution adopted integrates all telemetry and telecommand (TM/TC) from the EPD database. Finally, we have defined the concrete classes and transformations needed to incorporate the SRG-GSS tool into the model-driven development process. These transformations allow us to generate the input models for the SRG-GSS tool automatically and include the output logs with the results of the validation tests.

For the proof of concept, we have selected a subset of the original SSS requirements of the software of the ICU. Specifically, those corresponding to services 1, 3, 5, and 17 of the Packet Utilization Standard (PUS) [44]. We instantiated the SSS and SRS models with these requirements and an SVS model with the same functional tests designed for covering the software requirements related to them, along with the needed procedures for maintaining the test campaign flow. Table 1 shows the total number of requirements defined in the original SSS and SRS documents and the total number of validation tests and the telemetry and telecommand packets supported by the ICU, together with the numbers corresponding to the services selected for this proof of concept. We have also modeled a Test Setup for the proof of concept, based on the original scenario shown in Fig. 17.

We generated the documents in OOXML from the four main models SSS, SRS, SVS, and Test Setup, along with the input files that configure our SRG-GSS to implement the validation tests. We compared the generated OOXML documents with the originals of the project, confirming that the information contained in the new documents was proper and complete, and structured according to the ECSS-E-ST-40 standard.

The validation tests created with this approach were the same as the original ones, which were created manually. As we used the same SRG-GSS tool, the reports generated in the execution of the validation tests were similar to the original ones that had been produced in the V&V process of the ICU software. Then, we produced the TCRs using these reports and finally obtained the SVR document automatically using those TCRs. We compared the SVR document and the summary matrices contained inside with the original one, verifying the results were the same.

The proof of concept is available online, along with several examples. [2] The specific SRG-GSS models and transformations are also available online.[3]

6 Conclusions and future work

This paper has presented a model-driven engineering approach to the validation and verification process for space software applications. The solution follows the standards ECSS-E-ST-40 and ECSS-Q-ST-80 that are applicable in space software development. The final objective of this approach is to provide model-driven engineering techniques that maximize the automation of the different products required during validation and verification.

To this end, we have defined the necessary models not only for managing the requirements and validation tests, but also the test environments and scenarios required for test execution. Besides, the TM/TC database information has been integrated into the models, so that validation tests are defined according to the telemetry and telecommand format that is actually used in the space mission. Based on these models, a set of transformations have been implemented that enable the automatic configuration of the Ground Support Software in charge of the execution of the validation tests. Finally, to make it possible to close the verification cycle, the information of the logs obtained from test execution has also been integrated so that the verification matrices can be automatically generated.

The model-driven process provides a straightforward roadmap to actors to provide the inputs required in each phase and obtain as outputs the products necessary for the software validation and verification. Specifically, the requirement and validation test documents are generated, and also the verification reports that provide the traceability matrices between requirements and test, and also incorporate the test results extracted from the logs generated by the ground support software.

A completed proof of concept has been explained in the paper. This case corresponds to the development of the on-board software of an instrument that is part of an ongoing space mission, as it is the Energetic Particle Detector instrument of Solar Orbiter. We have selected a subset of requirements and tests to demonstrate the features of the solution. From this subset, we have completed the generation of the corresponding deliverable documents, as well as the test environment configuration files. These configuration files have enabled the ground support software to execute the validation tests automatically. We have verified that the results obtained from these tests are identical to those provided by the manually constructed tests, confirming that the effort required to complete the process was significantly lower due to the automation of a large part of it. Finally, the information contained in the test logs has been imported as models, and the verification report that traces requirements, tests, and test results has been generated. The report has been compared successfully with the original as an end-to-end result of the whole process, so it can, therefore, be concluded that the model-based approach presented provides an effective solution to assist in the software validation and verification process under ESA's ECSS-E-ST-40 and ECSS-Q-ST-80 standards.

As future works, and to have a single MDE environment that supports all the on-board software development activities, we will integrate the models and transformations that make up this work into the MICOBS model-driven framework. This framework currently provides support for component-based on-board software design, implementation, and deployment. Besides, it also facilitates the analysis of non-functional properties by applying the principles of compositionality and composability. All these MICOBS capabilities have been used to develop, according to the ECSS-E-ST-40 and ECSS-Q-ST-80 standards, the software of the control unit of the Energetic Particle Detector, so the extension of MICOBS to also integrate the activities of the V&V process will provide a model-driven environment ready to develop software compliant with the ECSS standards.

Acknowledgements This work was supported by Spanish Ministerio de Economía y Competitividad under the grants ESP2013-48346-C2-2-R, ESP2015-68266-R, and ESP2017-88436-R.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

[2] https://github.com/uah-srg-tech/mde-req-docs.

[3] The models are located in https://github.com/uah-srg-tech/gss-eclipse. The SRG-GSS specific transformations for creating the configuration files from the SVS and for parsing the reports for creating the TCRs are located in https://github.com/uah-srg-tech/test_campaign_parser and in https://github.com/uah-srg-tech/mde_to_ooxml, respectively.

References

1. Secretariat, E.C.S.S.: Space Engineering. Software, ECSS-E-ST-40C (2009)
2. Secretariat, E.C.S.S.: Space product assurance. software product assurance. ECSS-E-ST-80C Rev. **1** (2017)
3. IBM.: IBM Engineering Requirements Management DOORS Family. https://www.ibm.com/es-es/products/requirements-management
4. Sparx Systems.: Enterprise Architect User Guide, https://sparxsystems.com/products/ea/index.html
5. OMG.: About the Unified Modeling Language Specification Version 2.5.1. https://www.omg.org/spec/UML/About-UML/
6. Peccia, N.: SCOS-2000 ESA's Spacecraft Control for the 21st Century, 2003 Ground System Architectures Workshop (2003)
7. Rodriguez-Pacheco, J., Wimmer-Schweingruber, R.F., Mason, G.M., Ho, G.C., Sanchez-Prieto, S., et al.: The energetic particle detector–energetic particle instrument suite for the solar orbiter mission. Astron. Astrophys. **642**,(2020)
8. Schätz, B., Fleischmann, A., Geisberger, E., Pister, M.: Model-based requirements engineering with autoraid, Informatik 2005—Informatik Live! Band 2 (2005)
9. da Silva, A.R., Saraiva, J., Ferreira, D., Silva, R., Videira, C.: Integration of RE and MDE paradigms: the projectit approach and tools. IET Softw. **1**(6), 294–314 (2007)
10. Baudry, B., Nebut, C., Le Traon, Y.: Model-driven engineering for requirements analysis. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), p. 459. IEEE (2007)
11. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer **38**(9), 43–50 (2005)
12. Williams, L., Maximilien, E., Vouk, M.: Test-driven development as a defect-reduction practice. In: 14th International Symposium on Software Reliability Engineering, 2003, pp. 34–45. ISSRE (2003)
13. Fowler, M., Highsmith, J., et al.: The agile manifesto. Softw. Dev. **9**(8), 28–35 (2001)
14. Koskela, L.: Test Driven. Manning Publications, Greenwich, Connecticut, USA (2008)
15. Steel, J., Lawley, M., Steel, J., Lawley, M.: Model-based test driven development of the tefkat model-transformation engine. In: 15th International Symposium on Software Reliability Engineering, pp. 151–160 (2004)
16. Wieczorek, S., Stefanescu, A., Fritzsche, M., Schnitter, J.: Enhancing test driven development with model based testing and performance analysis. In: Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008), pp. 82–86. IEEE (2008)
17. Mou, D., Ratiu, D.: Binding requirements and component architecture by using model-based test-driven development. In: First IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks), pp. 27–30. IEEE (2012)
18. Sadeghi, A., Mirian-Hosseinabadi, S.-H.: Mbtdd: model based test driven development. Int. J. Softw. Eng. Knowl. Eng. **22**(08), 1085–1102 (2012)
19. Ramler, R., Klammer, C.: Enhancing acceptance test-driven development with model-based test generation. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 503–504. IEEE (2019)
20. North, D.: Introducing BDD. [Online] (2006). http://dannorth.net/introducing-bdd/
21. Horner, J.K., Symons, J.: Understanding error rates in software engineering: conceptual, empirical, and experimental approaches. Philos. Technol. **32**(2), 363–378 (2019)
22. Tavares, H.L., Rezende, G.G., Santos, V.M., Manhaes, R.S., de Carvalho, R.A.: A tool stack for implementing behaviour-driven development in python language, arXiv preprint arXiv:1007.1722 (2010)
23. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-based testing in practice. In: Proceedings of the 21st International Conference on Software Engineering, pp. 285–294 (1999)
24. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Testing Verif. Reliab. **22**(5), 297–312 (2012)
25. Snook, C., Hoang, T.S., Dghyam, D., Butler, M., Fischer, T., Schlick, R., Wang, K.: Behaviour-driven formal model development. In: International Conference on Formal Engineering Methods, pp. 21–36 . Springer (2018)
26. Sivanandan, S., et al.: Agile development cycle: approach to design an effective model based testing with behaviour driven automation framework. In: 20th Annual International Conference on Advanced Computing and Communications (ADCOM), pp. 22–25. IEEE (2014)
27. Kollanus, S.: Critical issues on test-driven development. In: Caivano, D., Oivo, M., Baldassarre, M.T., Visaggio, G. (eds.) Product-Focused Software Process Improvement, pp. 322–336. Springer, Berlin Heidelberg, Berlin, Heidelberg (2011)
28. Javed, A.Z., Strooper, P.A., Watson, G.N.: Automated generation of test cases using model-driven architecture. In: Second International Workshop on Automation of Software Test (AST'07), p. 3. IEEE (2007)
29. Parra, P., Polo, O.R., Knoblauch, M., Garcia, I., Sanchez, S.: MICOBS: multi-platform multi-model component based software development framework. In: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, Series. CBSE'11, pp. 1–10. ACM, New York, NY, USA (2011)
30. Hugues, J., Perrotin, M., Tsiodras, T.: Using MDE for the rapid prototyping of space critical systems. In: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, pp. 10–16. IEEE Computer Society, Washington, DC, USA (2008). http://portal.acm.org/citation.cfm?id=1447559.1447631
31. Garrido, J., de la Puente, J.A., Zamorano, J., de Miguel, M.A., Alonso, A.: Timing analysis tools in a model-driven development environment. IFAC-PapersOnLine (2017)
32. Parra, P., Polo, O.R., Fernandez, J., Da Silva, A., Sanchez, S., Martinez, A.: A platform-aware model-driven embedded software engineering process based on annotated analysis models. IEEE Trans. Emerg. Topics Comput. **9**(1), 78–89 (2021)
33. Johann Hoflinger, K., Sommer, J., Nepal, A., Maibaum, O., Lüdtke, D.: PaTaS—Quality assurance in model-driven software engineering for spacecraft. In: Proceedings of the ESA SW Product Assurance and Engineering Workshop 09 (2017)
34. Bünder, H., Kuchen, H.: A model-driven approach for behavior-driven gui testing. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1742–1751 (2019)
35. Perovich, D., Rossel, P.O., Bastarrica, M.C.: Feature model to product architectures: applying MDE to software product lines. In: 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, pp. 201–210. IEEE (2009)

36. Lamancha, B.P., Usaola, M.P., de Guzman, I.G.R.: Model-driven testing in software product lines. In: 2009 IEEE International Conference on Software Maintenance, pp. 511–514. IEEE (2009)
37. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-based coverage-driven test suite generation for software product lines. In: International Conference on Model Driven Engineering Languages and Systems, pp. 425–439. Springer (2011)
38. Paz, A., El Boussaidi, G.: On the exploration of model-based support for do-178c-compliant avionics software development and certification. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 229–236. IEEE (2016)
39. Marques, J., da Cunha, A.M.: Tailoring traditional software life cycles to ensure compliance of rtca do-178c and do-331 with model-driven design. In: IEEE/AIAA 37th Digital Avionics Systems Conference (DASC), pp. 1–8. IEEE (2018)
40. Grant, E.S., Datta, T.: Modeling rtca do-178c specification to facilitate avionic software system design, verification, and validation. Int. J. Future Comput. Commun. **5**(2), 120 (2016)
41. Pomante, L., Candia, S., Incerto, E.: A model-driven approach for the development of an IDE for spacecraft on-board software. In: IEEE Aerospace Conference, pp. 1–17. IEEE (2015)
42. Hovsepyan, A., Van Landuyt, D., Michiels, S., Joosen, W., Rangel, G., Fernandez Briones, J., Depauw, J. et al.: Model-driven software development of safety-critical avionics systems: an experience report. In: 1st International Workshop on Model-Driven Development Processes and Practices co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), vol. 1249, pp. 28–37 (2014)
43. CCSDS Secretariat.: Space Packet Protocol, CCSDS 133.0-B-1 Cor. 2 (2012)
44. Secretariat, E.C.S.S.: Telemetry and Telecommand Packet Utilization, ECSS-E-70-41C (2003)
45. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
46. Eysholdt, M., Behrens, H.: 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, ser. SPLASH '10, pp. 307–309. ACM, New York, NY, USA (2010)
47. ECMA International.: Standard ECMA-376. Open Office XML File Formats, 5th edn. ECMA-376 (2016)
48. Parra, P., Polo, O.R., Carrasco, A., da Silva, A., Martinez, A., Sanchez, S.: Model-driven environment for configuration control and deployment of on-board satellite software. Acta Astronautica **178**, 314–328 (2021). http://www.sciencedirect.com/science/article/pii/S0094576520305555

Aarón Montalvo is a Telecommunications Engineer and Master in ICT from the University of Alcal. He is currently is doing his PhD in Engineering at the University of Alcal. He has participated in several research projects, including the Solar Orbiter EPD project by the Space Research Group (SRG) of the University of Alcal. His research interests include the Ground Support Teams (GSE), including GSE software, the Validation and Verification processes, and the quality control in systems integration.

Pablo Parra received his Ph.D. in Information and Communication Technologies from the University of Alcal in 2012. Since 2006, he has been working with the Computer Engineering Department and the Space Research Group (SRG) of the University of Alcal. His research interests include component-based software engineering and model-driven engineering applied to the field of real-time embedded systems. He has taken part in numerous research projects in the field of on-board satellite software development, such as the NANOSAT programme, Solar Orbiter and Euclid.

Óscar Rodríguez Polo degree in Physical Sciences from the University of the Basque Country and a PhD in Physical Sciences from the Complutense University of Madrid in 2003. He is currently Associate Professor in the Department of Computer Engineering at the University of Alcal. His research work focus on the areas of Embedded and Real Time Systems and Model-Driven Engineering. He has actively participated in several national and international research projects as flying software project manager, in missions such as NANOSAT-01, NANOSAT-1B, MICROSAT or Solar Orbiter.

Alberto Carrasco Gallardo is a Telecommunications Engineer from the University of Alcal. He is currently employed as research staff associated to the Solar Orbiter EPD project by the Space Research Group of the University of Alcal. His main interests are real time operating systems, embedded systems, software for critical systems. He has actively participated in several national and international projects, missions such as NANOSAT-1B, MICROSAT and the Solar Orbiter EPD instrument.

Antonio da Silva is an Electronic Engineer and holds a PhD in Engineering from the University of Alcal. He is currently Associate Professor in the Department of Computer Engineering at the University of Alcal. He has participated, within the Space Research Group (SRG) of the University of Alcal, in the project Control unit for the Solar Orbiter EPD instrument. His priority areas of interest those related to fault tolerance and Time and Space Partitioning (TSP) of critical embedded systems through virtualization.

Agustín Martínez received the M.S. degree from the Universidad Polit'cnica de Madrid (UPM) in 1986 and the Ph.D. degree from the Universidad de Alcal in 2001. Professor of Computer Engineering Department at the University of Alcal in Spain, Head of Department from 2010 to 2016. His research and teaching activities are in the areas of DSP, Computer Architecture, Embedded Systems and Space Systems. He has been in charge of technical and management issues at Telettra, Alcatel and Alcatel-Lucent over more than 20 years, accounting a deep experience over technical and management leadership activities in international projects.

Sebastián Sánchez holds a PhD in Telecommunications Engineering. He is currently Professor at the University of Alcal (Computer Engineering Department). His teaching and research work focuses on the areas of Operating Systems, Computer Architecture, Embedded Systems and Real Time Systems. He has written several books on the subject of Operating Systems in general and UNIX and Linux in particular. He has actively participated in several research projects, both national and international, in the areas of on-board hardware and software, in missions such as SOHO, PHOTON, FUEGO2, NANOSAT, Exomars, MICROSAT, Solar Orbiter and Euclid.