Towards an Automated Hybrid Test and Simulation Framework to Functional Verification of Nanosatellites' Electrical Power Supply Subsystem

Italo Pinto Rodrigues[a,1,*], Ana Maria Ambrosio[a,2], Christopher Shneider Cerqueira[a,3]

[a]*National Institute of Space Research (INPE)*
Av. dos Astronautas 1758, São José dos Campos - SP, Brazil

Abstract

Tests in the space systems development life cycles are necessary to early verify requirements fulfillment, ensuring that the systems developed are correct. Nowadays, the efforts to develop miniaturized satellites and their test suite is increasing. Additionally, it is growing the initiatives is adopting MBSE (Model Based System Engineering) to automate the processes of: model design, simulation and model transformation. In MBSE development approach, models are the focus of the activities. The models describe requirements, functionalities and interfaces of a system, and their subsystems, considered here as "input models". In the context of an Electrical Power Subsystem (EPS), the design engineers have to (i) generate models representing solar array, battery, voltage regulators, loads, etc., for implementation solutions, and (ii) provide a verification plan, derived from requirements, to ensure the correctness of the developed functionality. In this scenario, the following question raises: "*how to interconnect the "input models" with verification plans, developed solutions and test executions?*" This paper aims to describe the structure of an automated verification framework to nanosatellite's EPS, using COTS (commercial-of-the-shelf) tools, such as MATLAB/Simulink®, MS. Excel®, and Arduino. We propose the models are as granular as in the verification plans (it is not possible to test internal behaviors from a black box artifact), so, each model represent an element in a unique file and a sequencer will integrate them, as a DSM (Design Structure Matrix) in Excel. In the context of the proposed framework, the subsystem verification enables three test configurations: fully simulated, fully simulated considering physical interface model, and hardware-in-the-loop (HIL). One advantage of the proposed framework is to reuse models from the start of the mission development, providing the reuse of these models throughout the life cycle, minimizing costs. The paper shows also results of development of the framework using an EPS behavioral model.

Keywords:
Automated Functional Testing, Simulation, Verification, Pico and nanosatellite, Electrical Power Subsystem, Hardware-in-the-loop.

1. Introduction

Developing Space Systems is a great challenge which involve the evolution of a need concept through multiple phases of product and process development until launch, and its proper use. Nanosatellites belong to the field of satellites designed to scientific research, alumni studies, technology validation and other

*Principal corresponding author.
Email addresses: italoprodrigues@gmail.com (Italo Pinto Rodrigues), ana.ambrosio@inpe.br (Ana Maria Ambrosio), christophercerqueira@gmail.com (Christopher Shneider Cerqueira)
[1]MSc. Student at INPE
[2]Senior Technologist at INPE (DSE/ETE).
[3]PhD. Student at INPE.

purposes. The development of NanoSats is of low cost; however, many of them do not succeed. In [1] it is shown that the Electrical Power Supply Subsystem (EPSS) failures represents 17% of nanosatellite failures that lead to loss of mission. The main faults in EPSSs are batteries and solar arrays, as seen in [2][3][4].

An alternative approach to develop NanoSats systems is the intensive use of simulation models into a MBSE (Model Based System Engineering) Environment. Into this environment, models help to: (i) define concepts, (ii) understand scenarios, (iii) derivate requirements, (iv) develop function models, (v) test prototypes, (vi) support acceptance and integration, (vii) train operation group, and (viii) test commands before send to the space segment [5].

Working with models allows performing test in the models in earlier phases, focusing the big picture of its use, instead of test only at an AIT (Assembly, Integration, and Test) phase [6].

In the context involving: (i) NanoSat Development, (ii) MBSE, (iii) EPSS, and (iv) tests, this work shows, in the following sections, a framework to speed up the test of models into a MBSE environment, to verify model consistency with its required functions, keeping the traceability of test cases in multiple simulation runs with the requirements and functions.

2. Concepts

The main concepts involved in this work are into three fields: (i) Modeling, simulation and MBSE, (ii) Verification of satellite functions: (a) simulated only, (b) physical models and (c) hybrid models, and (iii) the Nano and picosatellites Electrical Power Supply Subsystem (EPSS), as a case study. Fig. 1 summarize the fields of the work which are described in the following sections.

The proposed framework involves three main concepts: verification of space system, Model Based System Engineering (MBSE), Verification of Space Systems and, as case study the Electrical Power Supply Subsystem (EPSS), as summarized in .

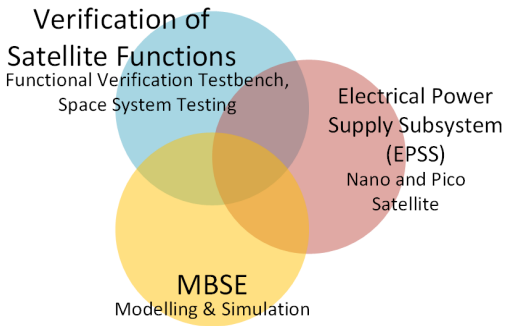

Fig. 1: The framework involves a multidisciplinary approach, including (i) MBSE, (ii) Verification, and (iii) Electrical Power Supply Subsystem - as test case.

2.1. Modeling, Simulation, and MBSE

Roughly speaking, projects are organized according to two approaches: (i) document centric - where all the documentation artifacts are not explicit connected and distributed, having no automatic revision and transformation process; and (ii) model centric - where all the documentation artifacts are connected and the information core are single to be used in any document it is necessary, their use is distributed but kept cohesion with a centered documentation Centre. International Council on Systems Engineering (INCOSE) states the MBSE (Model Based System Engineering) as a procedure guideline to a systemic approach into space (but not only) projects [7]. Software tools to model and simulate disciplines through the life cycle heavily support MBSE.

Based in the INCOSE definition [8], we understand MBSE as a formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.

Table 1 lists some important terms in the context of this work.

Table 1: Terms Definition [9].

Term	Definition
Model	Models are the representation of a system. Models examples: block diagram, physical, mathematical.
Metamodel	Metamodels are abstraction of the models, models of models. They define the model semantic and its relations.
Simulation	A method for evolve a model over time.

According to [5], the pervasive use of modeling and simulation (M&S) can provide several benefits, such as reduction of risks and costs. In this context, the ECSS describes different types of simulators throughout the project life cycle, dividing into three major groups: (i) simulators for analysis and design, (ii) facilities for spacecraft qualification and acceptance, and (iii) facilities for ground system qualification and testing and operations. The last groups involve not only simulation but also environments for test execution.

2.2. Verification of Satellite Functions

According to ECSS, the objective of verification is to demonstrate, that a product meets the specified requirements.

During Phase B, in life cycle development, simulation is required to support the verification of critical subsystem against the functional requirements [5].

Early models and physical prototypes are tested by a Functional Verification Testbench (FVT) (see Fig. 2) simulator type, described by European Cooperation for Space Standardization (ECSS) to support verification activities and focuses on the identified critical and prototyped elements. The main activities in the FVT are: (i) develop, test and integrate the functional models; (ii) integrate the product under test; and (iii) develop test scenarios / test plan. [5][10].

As shown in Fig. 2 the FVT can be composed by the following items:

- Virtual System Model: reflects the functions and behavior of the complete system to be built at the level required to support the respective analysis and verification tasks and comprises the simulation infrastructure and the models of the space system;
- Simulation Infrastructure: performs the interface between the system models and facility M&C. It is essential to allow model reuse to work smoothly;
- Models: are the models that represent the system, such as: environmental models, spacecraft dynamics models, spacecraft equipments models;
- M&C (Monitoring and Control): it is the interface between the user and simulator.
- Front-End Equipment (FEE): i is used to perform the interface between the simulator and physical equipment under test;
- Physical Equipment Under Test: it is a hardware the replaces one (or more) virtual model of the subsystem. It is stimulated by virtual simulation.

The FVT can adopt two configurations: (i) only software, where the simulation has only a virtual models; (ii) hardware-in-the-loop: it includes a hardware in the simulation. The last one have to allow gradual replacement of the virtual subsystem models by corresponding hardware.

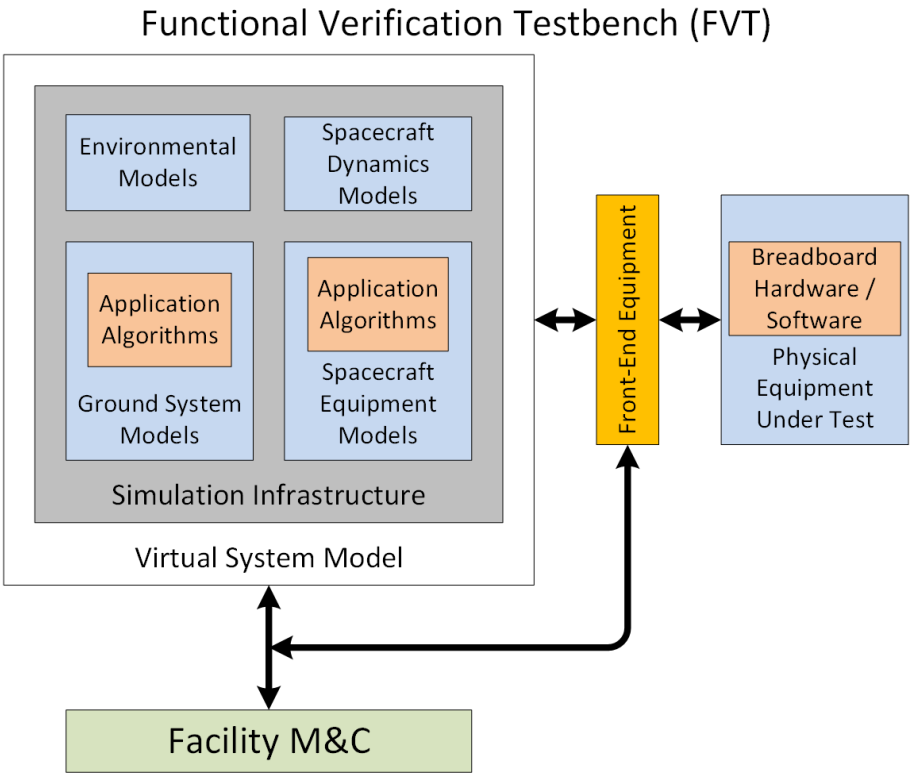

Fig. 2: Functional Verification Testbench [5].

2.3. Electrical Power Supply Subsystem

The Electrical Power Supply Subsystem (EPSS) is responsible to provide, store, distribute, and control (nano) satellite electrical power [11][12]. This subsystem comprises the followings items, as illustrated in Fig. 3:

- Solar Array: convert incident solar radiation to electrical energy;
- Energy Storage: stores energy for peak-power demands and eclipse periods;
- Power Distribution and Protect: ensures that the power is distributed to the satellite loads. It consists of cabling, fault protection, and switches to turn power on and off to the satellite loads.
- Regulation and Control: controls the use of power sources to ensure proper power distribution, it has three responsibilities: solar array control, bus voltage regulation, and battery charge

3. The Proposed Framework

In early development phases, as phases A, B and C, artifacts, as models (only software, hybrid, or only hardware prototypes) are available to early functional verification. Such artifacts might already be black boxes, allowing verification of functionalities only by the interfaces of the designed solutions. In a MBSE environment, an a important task is the traceability of requirements through models, products, and tests. To quicker achieve a proper verification, two teams should work in parallel, and independently: (i) Specialist Engineers - developing the systems models, and (ii) Test Engineers - developing the test tools and test cases

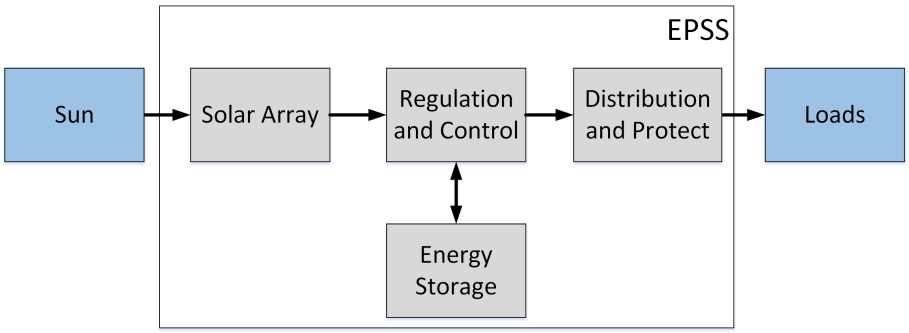

Fig. 3: Simplified EPSS Block Diagram.

to verify the requirements assessment into the models. Figure 4 illustrates this two-teams approach with the artifacts they produce: (i) subsystem related artifacts - subsystem models and execution matrix; and (ii) test related artifacts - test scripts, test case sequences, and test matrix. In order to speed-up testing procedures an automatic executor is proposed.

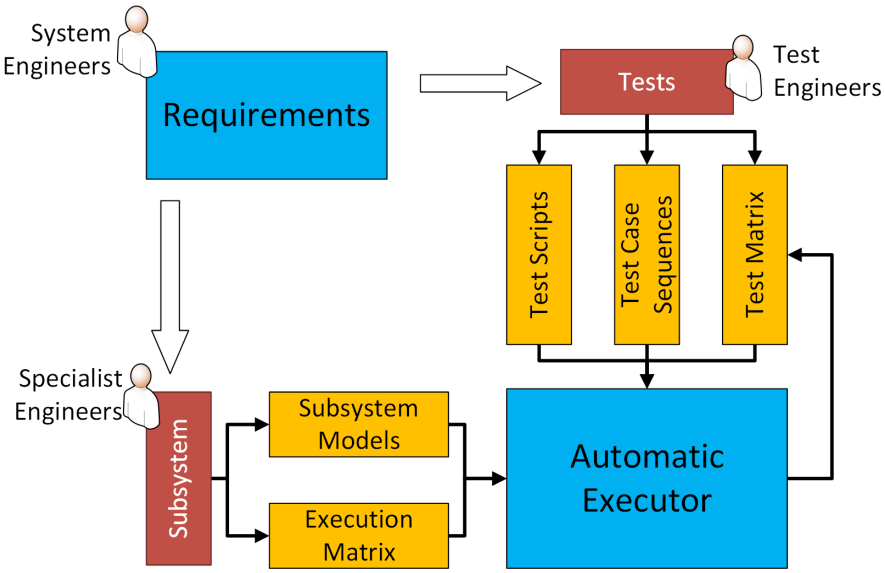

Fig. 4: Main view of the framework and inputs artifacts.

This section explains the artifacts created to execute an automated hybrid test batch with a connected orbital simulation to verify functional requirements as well as the process to integrate models and tests in simulation runs, as well as the resulting collection to be presented to the Test Engineers.

3.1. Materials

We chose to use MATLAB/Simulink as a common environment that allows both modeling and simulation [13][14][15][16]. Because it is a very powerful tool set, which allows co-simulation among several model types within MATLAB itself (state machines, dynamic models, block diagrams, by events, etc.) and with another tools, as LabVIEW, Solidworks, custom C++/Java models, etc. There are MATLAB/Simulink open source alternatives, as Octave, OpenModellica, and Scilab. However, we did not test those when we wrote this work.

3.2. Inputs

Specialist Engineers and Test Engineers must agree in a metamodel. The metamodel provides the characteristics that the model interface must have, so the models and tests communicate with each other's. The metamodel information consider the following artifacts:

a) **Subsystem Models** - contains the subsystem logic which mimics the desired system function behavior. The Specific Engineer has to provide those models with detailed interfaces - the input/output variables. To standardize the model control, on the software point of view, it is necessary the definition of an abstract software interface class. This interface class will convert data from/to the common execution area to/from the system model and provide two methods: **setup** - to start the initial configuration of the model, and **update** - to run an evaluation step of the model. The tests should also agree with this software interface class, only adding a third method to collect the results of the test. MATLAB has OOP (Object Orientation Programming), which allows creating such interface classes.
b) **Execution Matrix** - contains the models' executions and pooling sequence; it is described using a DSM (Design Structure Matrix). The DSM is a well know sequence representation in the space domain, and easy to create and debug. It describes the influence and dependence of each element in its rows and columns. To this framework, the rows are the sources and the columns are the destination. The DSM sequence of the Execution Matrix must have the same names defined in the System Models, as instance, *bus* - as a model (M) and *BusLoad* as an interface parameter (P). Note on Fig. 5: (i) the numbers, indicating the sequence, and (ii) three types of activity: M→P - write a parameter from interface class model into the common workspace, P→M - write a parameter from workspace into the interface class model, and M→M - execute the update method.
c) **Test Scripts** - contains the update routine of the tests cases. It is responsible to process an input and/or provide an output - as required by the given test. The script can log into a file, or just provide a simple answer to be collected; it all depends on how the Test Engineer design the test cases. To the framework does not matter the source of the test logic, however the models and test logic must have the same encapsulation rules to be parsed transparently by the framework adaptation routine.
d) **Test Case Sequences** - contains the description of each simulation run, the place and interfaces of test cases of each run, and the orbital and time parameters. The Test Case must inform, the inputs and outputs, and where the activity sequence will be adapted in the Execution Matrix.
e) **Test Matrix** - contains the traceability link between requirement and the test case. The Automatic Executor fills the matrix after each simulation run with the results produced by the test cases execution. The result can simply be fill with an "Ok" or "NOK" or even indication to look into a log file or do further testing - depending on how the Test Engineer coded the test case.

Figure 5 illustrates the four visible artifacts: (i) System Models, (ii) Execution Matrix, (iii) Test Case Sequences and (iv) Test Matrix. The Test script is an interface class "model" with an extra method that collects the results of the test.

3.3. Parsing the Inputs

The parsing process relates to processing these three artifacts: (i) Execution Matrix, (ii) Test Case Sequences, and (iii) Test Matrix, to create: (i) a configuration, (ii) update run, and (iii) a collecting sequence that will stimulate/read both models and tests.

- Parsing the Execution Matrix provides: (i) the execution sequence and (ii) the activities into MATLAB data structures.
- Parsing the Test Case Sequences provides: (i) the amount of simulation runs, (ii) the orbit propagator configuration and (iii) its test cases, with the input and output parameters to calculate orbit position. The parsing structure the test case information into groups of execution sentences. The adaptation routine place these groups after the desired position in the Execution Matrix.

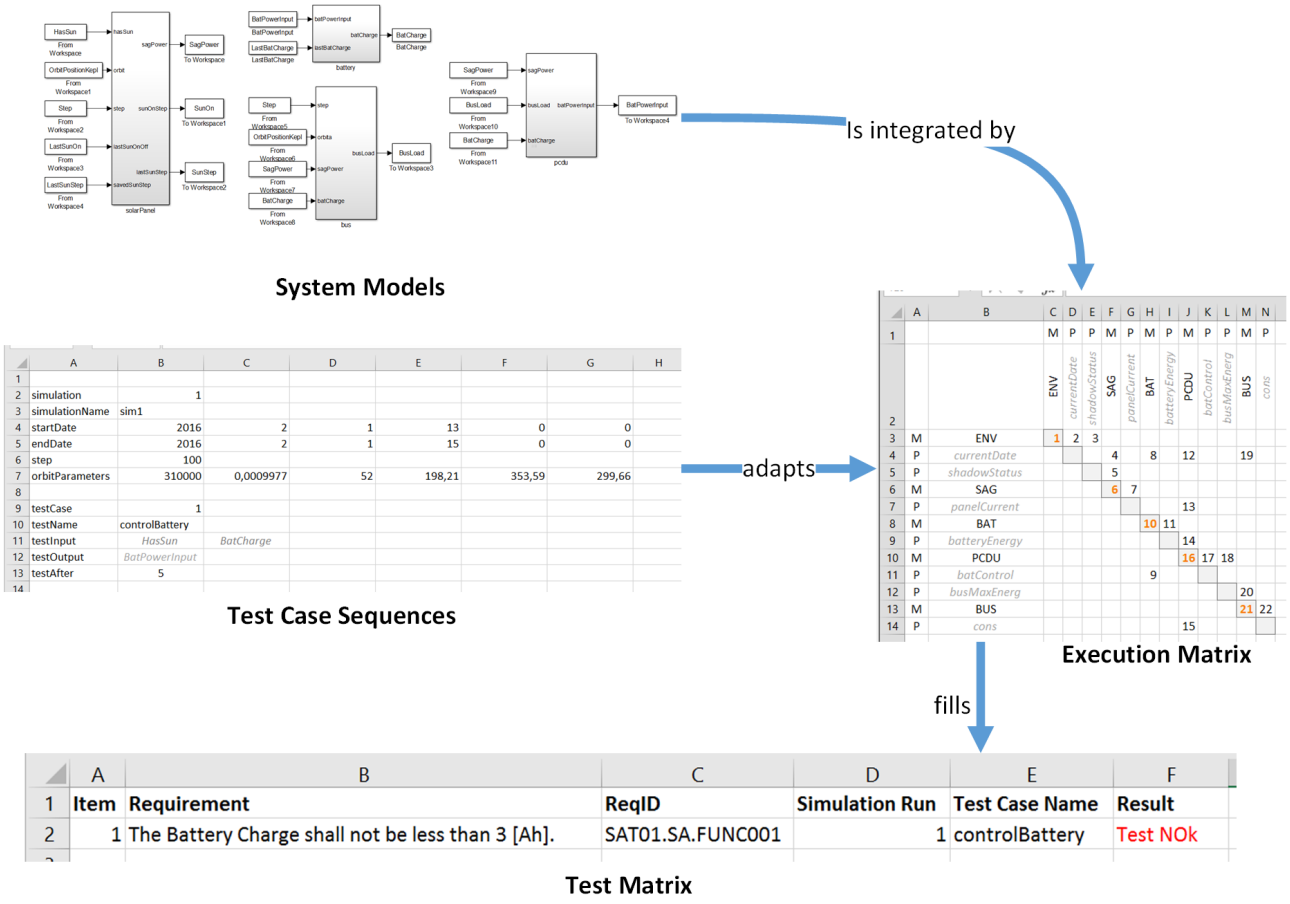

Fig. 5: Main view of the artifacts and its relations.

- Parsing the Test Matrix provides the correlation of where the collecting method will save the results string, so that the Test Engineer can visually verify if the test has passed or not.

Figure 5 gives an a overview of the artifacts and their relationships.

3.4. Combining Execution Matrix and System Models

The system models are combined with the execution matrix during the evaluation of the configuration and the execution sentences. It is done using MATLAB command called "`eval`", which evaluates a string. Each model or test case requires three configuration sentences:

- `import <name>.*`: import all classes inside the <name> folder;
- `<name> = <name>.controller`: instantiates the interface class into the workspace;
- `<name>.setup(<name>)`: calls the setup method to configure the model/test.

The execution sentences have three types:

- `<name>.set<parameterName>(<parameterName>)`: corresponding to a P→M;
- `<name>.update(<name>)` corresponding to a M→M;

- `<parameterName> = <name>.get(<parameterName>)`: corresponding to a M→P.

It is important to note that the evaluation of these strings only works correctly, if the Specific Engineer and Test Engineer respect the accorded naming dictate by a context metamodel. This script is created on execution time allows the reuse of this approach to other models.

3.5. Adapting the Standard Execution Sequence

The parse of the Execution Matrix produces a structure named Standard Execution Sequence with information provided by the Specific Engineer. The Test Engineer, however, need to adapt this standard sequence, coupling the test cases. The adaptation handles the test case as a group. The group contains: (i) input (`gets`), (ii) update, and (iii) output (`sets`) sentences. First, the algorithm must find the place where the test case will be placed, and then, shift the following activities by the number of activities in the test case activity block (see Figure 6(a)). Second, place the test cases in the end of the Execution Matrix (see Figure 6(b)), adding the activity index accordingly, and resorting again to organize the new items (see Figure 6(c)).

The limitation of this approach is when a test case needs to operate in different points of the Execution Sequence. In this case, we suggest to the Test Engineer to split a single test case into multiple test cases; placing it on the specific places that they are required and a test executor that will gather the data from the slipt cases.

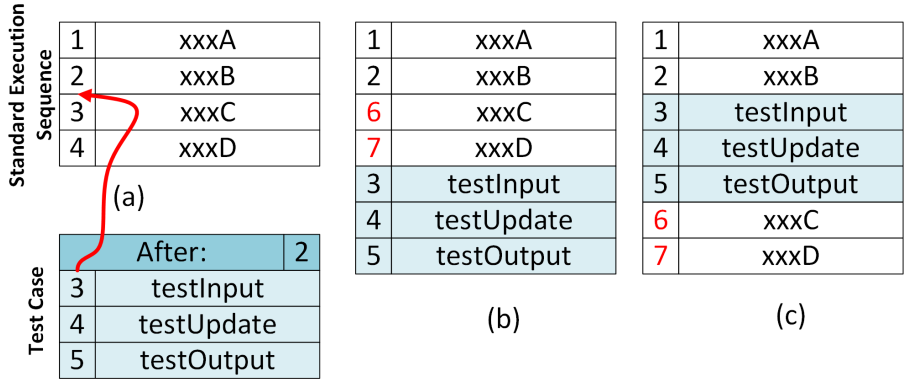

Fig. 6: Adaptation algorithm: (a) first - find the position to include the test case, (b) second - shift the activities and add the test case in the end of the sequence, and (c) third - re-organize the sequences.

3.6. Running a Simulation

Both tests and models share the same encapsulation interface class; this property allows them to run transparently together. The execution sequences on the main simulation loop step limit to handle interface flows (sets and gets) and classes updates. This is also true when using the hardware-in-the-loop (HIL) in a hybrid approach. The HIL encapsulation class must handle the communication from the simulated data from the models with the proper (Front-End Equipment) FEE that will acquire/stimulate the HIL data.

In the context of this work, a representative example of FEE was created using an Arduino Mega [17], as it is: (i) simple to handle, (ii) have several analog and digital inputs/outputs, (iii) cheap to buy, and (iv) easily controllable by the MATLAB/Simulink.

The last feature allows to directly control the Arduino pins by MATLAB commands, similarly as the Firmata library [18]. These control commands will exchange data with the physical FEE (Arduino made) inside the update method of the interface class. The setup method is responsible to: (i) instantiate an Arduino object type, (ii) indicate which USB COM, (iii) start the FEE, and (iv) place the initial configuration profile in the output pins.

A particular feature of MATLAB/Simulink allows embedding models created in the IDE into the Arduino. It is not the case of the FEE wich is controlled by the MATLAB with a Firmata like protocol. However, if someone plan to develop an Atmel based system, copying the Arduino USB hardware-programming interface this information is highly relevant, because the MATLAB/Simulink allows a much faster software development, and allows the engineers to use MATLAB/Simulink language instead of the Arduino IDE.

Figure 7 illustrates both a sketch of the HIL configuration using Arduino as the HIL and as FEE on (a) and a picture of this setup on real testing on (b).

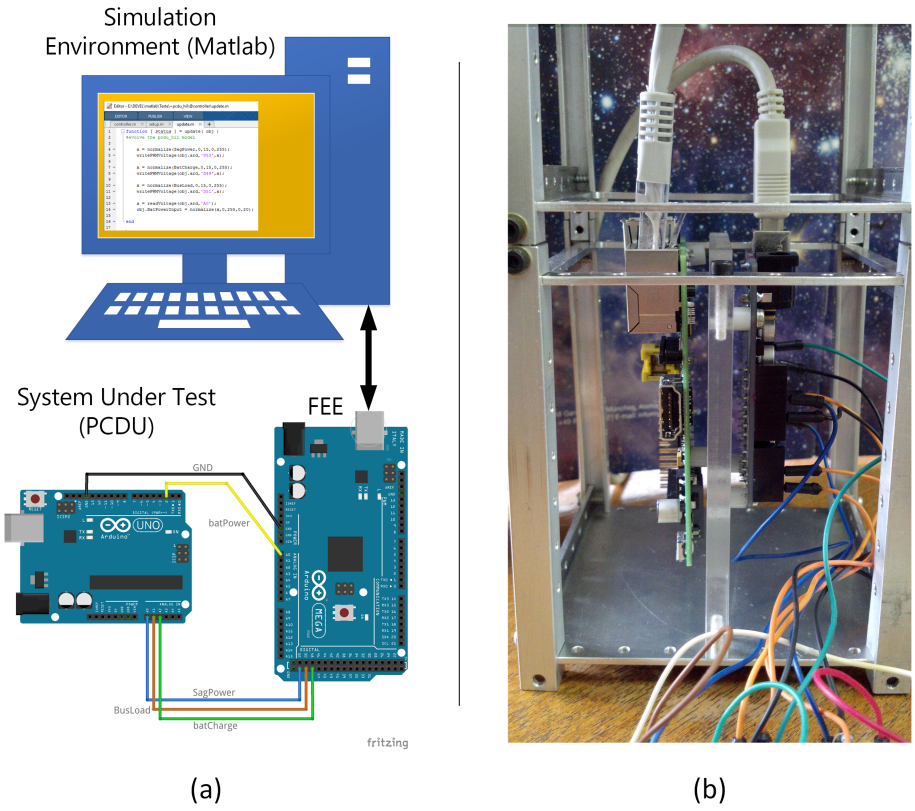

Fig. 7: Simple HIL architecture to stimulate a prototyped PCDU with an Arduino Mega as a FEE. (a) Design sketch and (b) implemented test bench.

3.7. Output - Collecting Test Data and Write Answers into Test Matrix

As tests shares with the models almost the same interface they can run together in the simulation loop. Instead of doing a simulation step on a model, the combination with the tests allows one to perform: (i) data analysis, (ii) data conversion, (iii) include data input into other models, (iv) log data, (v) evaluate parameter changes, so on.

In our framework, the Test Engineers will have the test case script in the simulation loop, related with answers back into the Test Matrix in a human understandable way. These answers are strings that can indicate, as example: test passed or not passed, log output to analysis, or happened event that requires extra conditions to evaluate, etc. Test Engineers may to program these answers, as well as, the choice of source of the test: (i) custom made or (ii) auto-generated. There are several tool that auto-generate test cases [19][20] , those tests need to be encapsulated following the metamodel abstract interface class.

To collect the test results, the tests has a third static method called “collect”. This method should return a phrase that indicates a suitable answer related to the evaluated conditions. The automatic executer writes

the answer string in the Test Matrix row, correspondingly to the Requirement, simulation run and test case name.

4. Discussions on the Uses of the Proposed Framework

The framework was firstly used to verify requirements of a Power Supply Subsystem, because was readily available at the time of writing this work.

However, this framework is generic and does not depend on the models described here. The models, and test cases, have only to respect the defined interface metamodel and the step-based nature. Inside a step time of the simulation loop, the model does not need to be discrete-based, or fixed-time, but the answer have to be accordingly to the time step.

Moreover, the framework focused the functional verification on Phases B / C, where the initial hardware prototypes appears. Nevertheless, changing the test natures allows to completely change the framework usage to, for example:

- Conceptual Modeling - Phase 0 / Pre-A - each Specialist Engineer develop their model solution in an environment called CDF (Concurrent Design Facility), and they commonly use a DSM to represent the influence/dependence of disciplines (models) and its interfaces. So, it is required to translate the influence/dependence DSM to describe the integration of the Execution Matrix, and encapsulate the models in the proper metamodel. " Tests" can be used to monitor the results of the models giving the Engineers feedback of the model coupling.

- Industrialization - Phase D - after defined the model boundaries and the respective tests to verify it, it is handled the test procedures to industry in order to verify their specific designs solutions, serving as a baseline, and earlier testing procedures.

- Assembly, Integration and Test (AIT) - Phase E - during AIT, there are environment tests, integration tests and functional tests. The environment tests are strongly related to specific hardware as (i) shakers and (ii) thermo-vacuum chambers; however, some integration tests and functional tests can be translated to a common framework. From the actual approach, in this framework, the AIT team must provide the different FEEs and the specific tests.

- Operation: Phase F - during operation, a simulation have to answer to operation protocols. Models still have to be interconnected, by Execution Matrix - if granularity is a requirement; tests can be used as interface to other custom systems, which will send/receive queued telecommands and telemetries.

5. Conclusion

Test are a great responsibility in Space Development. They allow to ensure the functionalities of the developed space product meets the requirements. In a MBSE Space Development, the process is highly based in simulating models. As so, these models can also be tested front of the required functions.

This paper presented a framework to automatically verify tests cases in a simulation environment, where the models are exemplified by the EPSS of a nanosatellite under development. The artifacts handled by the framework: subsystem models, execution matrix, test scripts, test case sequence and test matrix; and some algorithms that process and adapt the artifacts to execute the tests are described. Details of the solution using MATLAB and lessons learned in order to extend this framework to be applied in different verification phases of a nanosatellite is discussed.

The framework allows the reuse of meta-models for other projects. Moreover, the subsystem models under verification can also be reused with few adaptation for other projects.

Acknowledgments

We conduct this work during the masters program in Space Engineering and Technology at the National Institute of Space Research (INPE), with a CAPES - Brazilian Federal Agency for Support and Evaluation of Graduate Education scholarship.

References

[1] M. Swartwout, The First One Hundred CubeSats: A Statistical Look, Journal of Small Satellite 2 (2) (2013) 213–233.
URL http://web.csulb.edu/~hill/ee400d/Project%20Folder/CubeSat/The%20First%200ne%20Hundred%20Cubesats.pdf

[2] L. Alminde, M. Bisgaard, F. Gudmundsson, C. Kejser, T. Koustrup, C. Lodberg, T. Viscor, Power Supply for the AAU Cubesat, Tech. Rep. 1, University of Aalborg, Fredrik Bajers Vej 5, 9100 Aalborg, Dinamarca, report (fev. 2001).
URL http://www.space.aau.dk/cubesat/dokumenter/psu.pdf

[3] J. W. Cutler, J. C. Springmann, S. Spangelo, Initial Flight Assessment of the Radio Aurora Explorer, in: Proceedings..., 25th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, USA, 2011.
URL http://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1137&context=smallsat

[4] NANOSATC-BR, NanoSatC-BR (2014).
URL http://www.inpe.br/crs/nanosat/noticias_2014.php#noticia5

[5] EUROPEAN COOPERATION FOR SPACE STANDARDIZATION (ECSS), Space engineering: System modelling and simulation (ECSS-E-TM-10-21A), Tech. rep., ESA-ESTEC, http://www.ecss.nl/ (apr. 2010).

[6] J. Eickhoff, Simulating Spacecraft Systems, 1st Edition, Springer Aerospace Technology, Heidelberg, Germany, 2009.

[7] MBSE WIKI, start [MBSE Wiki], http://www.omgwiki.org/MBSE/doku.php (2016).

[8] INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING (INCOSE), Systems Engineering Vision 2020, Tech. rep., INCOSE (sep. 2007).
URL http://oldsite.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf

[9] DEPARTAMENT OF DEFENSE (DOD), Modeling and Simulation (M&S) Glossary, Tech. Rep. 1, DOD, http://goo.gl/dSZeeW (oct. 2011).

[10] EUROPEAN COOPERATION FOR SPACE STANDARDIZATION (ECSS), Space engineering: Verification guidelines (ecss-e-hb-10-02a), Tech. Rep. 1, ESA-ESTEC (dec. 2010).
URL http://www.ecss.nl/

[11] J. Wertz, W. Larson, Space Mission Analysis and Design, 3rd Edition, Space Technology Library, Springer Netherlands, 1999.

[12] M. R. Patel, Spacecraft Power Systems, CRC Press, 2004.

[13] G. Colombo, U. Grasselli, A. D. Luca, A. Spizzichino, S. Falzini, Satellite power system simulation, Acta Astronautica 40 (1) (1997) 41 – 50. doi:http://dx.doi.org/10.1016/S0094-5765(97)00022-2.
URL http://www.sciencedirect.com/science/article/pii/S0094576597000222

[14] M. Zahran, In Orbit Performance of LEO Satellite Electrical Power Subsystem - SW Package for Modelling and Simulation Based on MatLab.7 GUI, in: Proceedings..., 2006 IASME/WSEAS International Conference on Energy & Environmental Systems, Chalkida, Greece, 2006, pp. 379–384.
URL https://www.researchgate.net/publication/237755595_In_Orbit_Performance_of_LEO_Satellite_Electrical_Power_Subsystem_-_SW_Package_for_Modelling_and_Simulation_Based_on_MatLab.7_GUI

[15] H. Farid, M. El-Koosy, T. El-Shater, A. El-Koshairy, A. Mahmoud, Simulation of a Leo Satellite Electrical Power Supply Subsystem In-Orbit Operation, in: Proceedings..., 23rd European Photovoltaic Solar Energy Conference and Exhibition, Valencia, Spain, 2008.
URL http://www.eupvsec-proceedings.com/proceedings?paper=2472

[16] A. Berres, M. Berlin, A. Kotz, H. Schumann, T. Terzibaschian, A. Gerndt, A Generic Simulink Model Template for Simulation of Small Satellites, in: Proceedings..., 7th Symposium on Small Satellites for Earth Observation, Wissenschaft und Technik Verlag, Berlin, Germany, 2009, pp. 247–250.
URL http://elib.dlr.de/61201/1/IAA_2009.pdf

[17] ARDUINO, Arduino - ArduinoBoardMega2560, https://www.arduino.cc/en/Main/ArduinoBoardMega2560 (2016).

[18] ARDUINO, Arduino Firmata, https://www.arduino.cc/en/Reference/Firmata (2016).

[19] A. M. Ambrosio, E. Martins, S. V. d. Carvalho, N. L. Vijaykumar, An approach for concurrent fsm-based test case generation, in: Anais..., Workshop dos Cursos de Computação Aplicada do INPE, 3. (WORCAP)., Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2003, pp. 25 – 30.
URL http://urlib.net/lac.inpe.br/worcap/2003/10.31.14.20

[20] A. C. Pinheiro, A. Simão, A. M. Ambrosio, Fsm-based test case generation methods applied to test the communication software on board the itasat university satellite: A case study, Journal of Aerospace Technology and Management 6 (4) (2014) 447–461. doi:10.5028/jatm.v6i4.369.
URL http://urlib.net/sid.inpe.br/mtc-m21b/2015/01.13.18.16