AMORE – AN ACCREDITED MODEL REPOSITORY TOWARDS THE REUSE ON AOCS PROJECT

Leandro T. Hoffmann[1], Carlos J. A. Moreira[2], Igor M. Lopes[3], Marco Hidalgo[4], and Roberto V. F. Lopes[5]

[1][5]*Brazilian Institute for Space Research (INPE), Av. dos Astronautas, 1758, 122270-010, São José dos Campos-SP, Brazil, +55 (12) 3208-6000, {leandro.hoffmann, roberto.lopes}@inpe.br*

[2]*cjamoreira@gmail.com*

[3]*rogimainenti@gmail.com*

[4]*marco.hidalgo@rocketmail.com*

Abstract: *In this work, we present a methodology for creating an accredited model repository to increase the level of software reuse in the development of Spacecraft Dynamics Simulators (SDS). After a brief introduction of the state-of-the-art on spacecraft simulators and their role in space engineering activities, the paper describes our process for selecting, converting and building a catalogue of models for future reference and reuse. This is demonstrated using a set of models developed for the Attitude and Orbit Control Subsystem (AOCS) of Amaznia-1, the first mission that will be based on the Brazilian Multi-Mission Platform. The developed models are implemented in MATLAB® and C++ and the latter are executed by a flexible simulation infrastructure that defines standard interfaces for allowing the rapid reconfiguration of scenario and model interchange. Next, the paper describes case studies of simulations built from models available in the repository, in which the benefits of the repository creation can be valued. The assessed level of software reuse encourages the adoption of the methodology for future projects of SDS.*

Keywords: *Spacecraft dynamics simulator, AOCS verification and validation, model repository, electrical ground support equipment.*

1. Introduction

Simulation plays an important role in the AOCS design, construction, testing, verification and validation and hence, a reliable infrastructure is crucial to achieve a successful result. Typically, the AOCS projects involve the interaction of several teams and, often, each team employs specialised tools in their activities, with little interoperability and lacking on reuse. As the demands for embedded software increases, additional verification and validations tools are required, bringing the opportunity to rationalize their use and the overall engineering process.

Nowadays, a common approach for building space systems is to adopt a model-based system engineering process, where the activities are conducted by the elaboration of domain models, which can be interchanged among teams. Such a process is often supported by simulators that execute the models in certain scenario conditions and anticipate the verification of the product. In the case of AOCS, for instance, the control engineering team designs the control law (typically using tools like MATLAB/Simulink®) and in the next phase the computing team verifies and validates the on-board software, after the control algorithm has been automatically generated to a programming language for embedded systems (.e.g. Assembly, C, ADA). In both cases, the test harness is composed by infrastructures that simulate all environmental conditions to evaluate interfaces, functionalities and performance of the controller software.

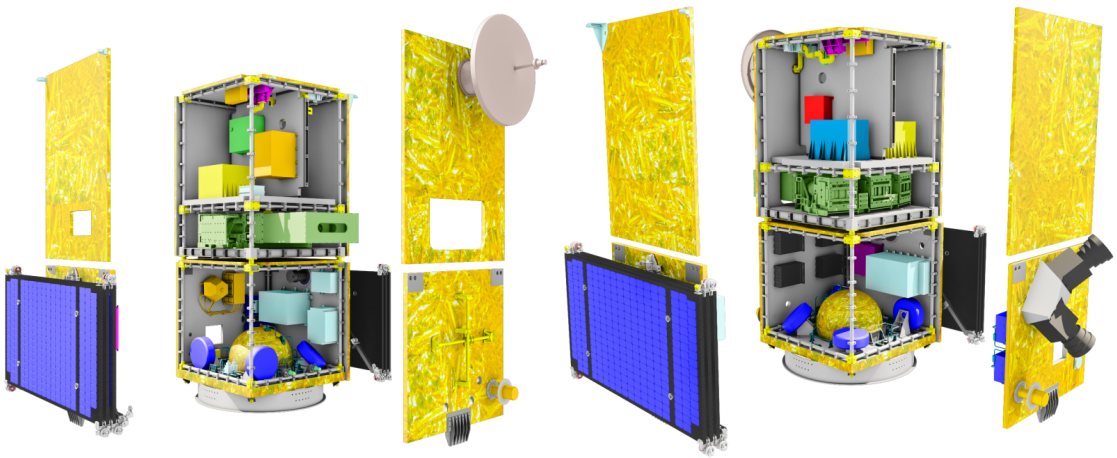


Figure 1: Amazonia-1 and Multi-Mission Platform concepts. *Source: MMP Project's Image Repository.*

In this context, a Spacecraft Dynamics Simulator (SDS) is being developed at Brazilian Institute for Space Research (INPE), envisaging the future Brazilian missions based on the Multi-Mission Platform (MMP) as well as the formation of specialized human resources to the local space engineering industry, which will be in charge of the final project. The MMP is designed to attend the requirements of the next INPE space missions in the scientific, meteorological and earth observation categories and the general concept of first MMP-based mission, the Amazonia-1 satellite, is illustrated in Figure 1.

The SDS project is composed of three products: functional engineering simulator; software validation facility; and test control module, which includes the test setup, visualization and recording tools. In general, the work packets are oriented by simulation models and a high software modularity in foreseen to allow their reuse to other MMP missions.

Initially, each model is designed in MATLAB and then a real time version is implemented in C++ with Qt Creator environment. Both codes are made independently by at least two members of staff, based on the same source of documentation. This gives different opportunities of skills improvement and knowledge sharing, as well as cross verification of the final product.

It is also strongly encouraged that the simulation engineers make use of existing simulation models and software infrastructure as much as possible. Nevertheless, after a short survey on available simulation product from other projects at INPE, it is observed that some sort work of rework was unavoidable. The main challenges for reusing heritage artifacts are the interface heterogeneity of models and a lack of software flexibility in existing infrastructures. Therefore, the establishment of guidelines for building a generic environment is required for the SDS design and the implemented models should comply with the following design goals: (i) all interfaces must be well-defined for easing its assembly and reconfiguration to different simulation scenarios; (ii) all C++ models must be executable in a common simulation infrastructure; and (iii) models must be accessed from a configured repository, in which a clear description of its functionalities, interfaces, constraints and qualification process is available.

To cope with these objectives, in this work a methodology for creating an Accredited Model Repository (AMoRe) is presented to address the configuration management of models and increase the level of software reusability in the SDS context.

Our approach consists in guiding the development of models in a component-based way, which starts with the configuration of MATLAB models built in the earlier phases of algorithms design and validation activities. The candidate scripts for reuse are identified and a set of metadata is collected, such as name, description, constraints and limitations, input and output parameters, external dependences, etc. Before the model can be made available in the repository, its source code must comply with a standardised naming and coding style and be accompanied by a set of testing artefacts (harness, data, scripts and reports). This methodology has been applied for the last three years in a training project in the field of AOCS testing, taking the AOCS of Amazonia-1 satellite as a test bench, which is currently under development at INPE with an Attitude Control Subsystem provided by Argentine company INVAP. The repository provides artefacts for modelling typical pieces of equipment (Coarse Solar Sensors, Star Trackers, Magnetometers, Gyroscopes, Reaction wheels, magnetic torque rods and Thrusters) and space environment (Orbit dynamics, attitude dynamics, geomagnetic fields, celestial sphere, Earths albedo, Sun position and dynamic perturbations). The remaining of this document is organised as follows: after a brief introduction of the state-of-the-art on spacecraft simulators and their role in space engineering activities, in section 2., the adopted model development process and the basis for defining a model catalogue are presented in section 3.. The overall features of the repository and the currently available models are listed in section 4.. The section 5. demonstrates the usage of existing models for conducting simulation analysis in the scope of MMP missions. Later, the section 6. gives the final conclusions of this work.

2. Spacecraft Simulation

Traditionally, the strategy of building simulators to support space engineering process has been adopted since the early days of space race, wherein, for instance, models executed in analogue computers have been used to assist spacecraft designers from the *National Aeronautics and Space Administration* (NASA), in 60's [1]. All the determination for developing new simulation technology since then is now reflected in the vast number of engineering fields that make use these tools in their processes, such as mission analysis, on-board software development, verification and validation in system and subsystem levels, assembly & integration campaigns, ground teams and astronauts training, support on mission operations, among others [2, 3, 4, 5, 6, 7, 8, 9, 10].

A spacecraft simulation infrastructure comprises of a facility composed of hardware and software elements that executing a simulation model extracts behavioural information or emulates environmental conditions to test a space system device and validate operational procedures. The increasing number of simulation applications in engineering fields stimulates the pursuit for better and cheaper methods for facilities construction, which frequently raises the number of requirements for flexibility, software reuse, interoperability, high fidelity models, and reliable infrastructures.

Besides the complexity of the spacecraft, the construction of simulators itself demands substantial amounts of resources. Due to the particular needs of each developing phase, the features of simula-

tion infrastructure have to be tailored (e.g. different levels of fidelity, real time constraints). Still, many software components, modelling efforts and test procedures can be available from one phase of the project to another. In this sense, it is expected that reusability and automation requirements increases on quantity and quality on ground software systems, envisaging the optimization the project's timeline and resources. This concern is being reflected in the way the spacecraft simulation architectures are designed and in the process of building simulations as the project evolves.

To achieve such degree of reusability, it is recommended that the facility design takes into account the adoption of standard interfaces and best practices for generic software development, like generic programming techniques, meta-modelling methods or component and model-driven architecture approaches. In this direction, led by the European Space Agency (ESA), significant effort has also been conducted in European community for harmonising the development and integration of simulation tools into the engineering activities. As a result, several technical memorandums have been published in the last years to guide the construction of infrastructures, development tools and simulation models.

One of these documents is the ECSS-E-TM-10-21A [11], which describes a set of eight types of simulation facilities that can be used to support space engineering activities performed along the mission life time:

- **System Concept Simulator (SCS)**: supports the execution of trade-off studies and the definition of mission concept;
- **Mission Performance Simulator (MPS)**: assesses the system interfaces and end-to-end design trade-off;
- **Functional Engineering Simulator (FES)**: assists the system requirements consolidation, supporting the functional design and algorithms validation;
- **Functional Validation Testbench (FVT)**: performs performance analysis and validation of critical elements for the Preliminary Design Review;
- **Software Validation Facility (SVF)**: provides a test bench for on-board software development, test, verification and validation, which can be performed in closed-loop or open-loop modes and includes a hardware-in-the-loop;
- **Spacecraft AIV Simulator (AIV)**: supports system qualification and acceptance during assembly, integration and test campaigns with incremental spacecraft verification & validation;
- **Ground System Test Simulator (GST)**: provides a testing environment for the mission control and ground systems; and
- **Training, Operations and Maintenance Simulator (TOM)**: performs simulation for mission control team training, validation of operation procedures, such as on-board software patch upload and anomaly investigations.

These environments have many elements in common, which brings the chance for hardware and software reusing and designing of a modular architecture. In addition to that, special attention is given for enhance model reuse along facilities, which can be particularly challenging when different simulation infrastructures are used and multiple scenario reconfiguration may be required. To address this point, the *Simulation Modelling Platform* (SMP2) standard formalizes all the software interfaces between the component models and the simulation kernel [12]. Moreover, it provides a platform independent definition language to describe the simulation models, their assemblies and

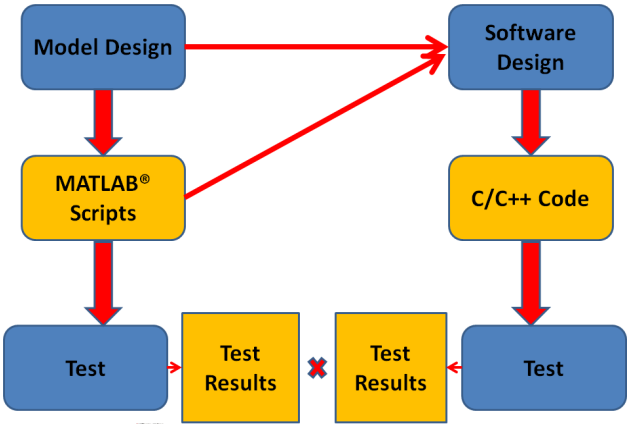


Figure 2: Model development process adopted for comparing output from MATLAB and C++ implementations.

scheduling, and model packages.

By using such standard, independent groups can work in parallel during the model development, since all simulation components will comply with the same interface. This is especially interesting for suppliers who can deliver both flight equipment and respective simulation model which will be later integrated with other models. This approach enhances the reliability of the models and reduces integration issues, since all models must implement the same communication interface. Aiming the production of highly flexible infrastructures and envisaging software reuse, development of this work have been based on concepts presented in this section.

3. Model Development Process

At present, the main applications of the models that are being developed in our projects consist in building a FES and a SVF. For this reason, MATLAB and C++ are the target languages in which the models should be available in the repository, since the former provides a rapid prototyping and the latter is suitable for real-time applications.

The general process is depicted in Figure 2. It starts with the mathematical modelling of devices or physical phenomena in MATLAB and then both documentation and m-files serves as a reference for model translation into C++. In this conversion step an automated code generation tool could be adopted, but usually the generated code lacks on readability or the tool is too expensive for the scope of our project. Thus, a semi-automated process was adopted, where the MATLAB code is codified in C++ using a common algebraic library and then wrapped by a standardised interface.

In the first step of C++ implementation, the algebraic library plays an important role for converting m-files, since it provides a set of matrix, vector and quaternion operations and makes the task less cumbersome (see subsection 3.1.). Usually the codification results in a single C++ class with few methods for initialising the model and computing its outputs. For improving the quality in the implementation, the programmer must also follow a coding standards document with programming style definitions and patterns for the class structures.

Before the C++ model can be made available in the repository, its interface must comply with standards adopted by the simulation infrastructure. In this step, the simulation engineer designs the model faade and creates a formal definition in catalogue, thereby a C++ wrapper can be automatically generated for the model. This process will be further detailed in the subsection 3.3..

In the end of the process, a formal verification and validation procedure is executed to check if the models are properly implemented or not. The same test cases are applied to the MATLAB and C++ components and then the results are automatically compared to ensure that they are compatible. Due to particularities of the computational platforms, it is not possible to obtain the very same results (i.e. bit-to-bit) from both implementations. Still, it is found that to minimize these differences a worthy strategy is to base all the codification on the same reference of constants. This approach has contributed to keep the variances in double precision variables smaller than 10^{-6}.

Our approach consists in guiding the development of models in a component-based way, wherein each simulation model has well-defined services to interface with other models and with simulation kernel. Once available in a proper catalogue, an ensemble of simulation component can be connected to run in a common infrastructure and to answer the needs of a given simulation scenario.

In the following subsections, details of the algebraic library and the simulation environment are presented.

3.1. Algebraic library

The algebraic library is a fundamental set of C++ components to perform operations with matrices, vectors and quaternions. It provides a common and reliable basis for modelling AOCS elements, providing more than hundred services for converting MATLAB script, such as multiplication, addition, subtraction, rotations, length, inversion, dot and cross product of vectors, and representation conversion (i.e. matrix to quaternion and vice-versa).

By adopting a Standard Template Library design, any size of bi-dimensional matrices and vectors may be declared. Nevertheless, a specialised *inline* implementation is provided for matrices and vectors of size 3, thus optimizing the computational performance in these usual cases.

Thanks to the operator overloading feature available in C++, the conversion of m-files is straightforward and the readability of the code is kept extremely high. The Table 1 gives a list of implemented operators, which makes the syntaxes of both codes very similar. An example of equivalent code written in MATLAB and then converted to C++ can be compared in the Table 2.

3.2. Simulation Environment Context

The construction of Software Validation Facility is based on generic programming and metadata techniques to ensure highly software flexibility and promote model reuse. It is built on a simulation framework, named *SimuBox*, which provides a *Simulation Development Kit* (SDK) with a set of C++ libraries, editing tools and simulator kernel to support the design and codification of models and simulation applications (Figure 3).

Table 1: C++ operator overloading converting implemented in the algebraic library.

Op.	Class			Operand				Description
	M	V	Q	M	V	Q	S	
()	✓							Read/write access to an element of matrix.
[]		✓	✓					Read/write access to an element of the vector or quaternion.
=	✓	✓	✓	✓	✓			Assignment.
==,!=	✓	✓	✓	✓	✓			Comparison.
+,+=	✓	✓	✓	✓	✓	✓	✓	Addition.
−,−=	✓	✓	✓	✓	✓	✓	✓	Subtraction.
−	✓	✓						Unary subtraction.
,=	✓	✓	✓	✓	✓	✓	✓	Scalar product, matrix multiplication or dot product.
/,/=	✓	✓	✓				✓	Division by scalar.
∧, ∧=		✓			✓			Cross product.

M=Matrix; V=Vector; Q=Quaternion; S=Scalar.

Table 2: Illustration of code readability when converting MATLAB script to C++ using the operator overloading functions.

Variables	$Matrix<3,3>I$; $Vector<3>w,hm,tw,torq$;
C++	$w = I.inv() * (((I * w + hm) \wedge w) + tw + torq)$;
MATLAB	$w = inv(I) * ((cross((I * w + hm), w) + tw + torq))$;

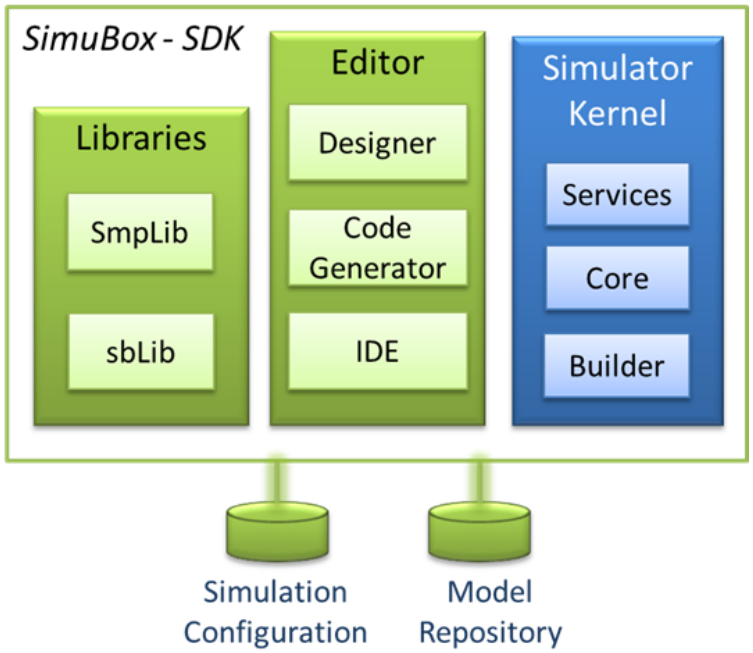


Figure 3: *SimuBox*'s Simulation Development Kit elements.

After the ECSS-E-TM-40-07 publication, the project has been motivated to support SMP2 models, in order to assess the potentials of the standard and to define a flexible interface between models and simulation kernel. These interfaces are defined and implemented in the provided *Libraries* of the Figure 3.

The *Simulator Kernel* is delivered as a runtime component compliant with the metadata and model packages produced with the editor and SDK libraries. During execution time, configuration files are used to load the binary packages and instantiate the models into the environment.

The editing tools automate the generation of wrapper code for the models and facilitate their configuration in various simulation scenarios. By designing with a diagrammatical tool, the simulation engineer describes the types of models, their instantiations, interconnections, and scheduling for a given application. Later on, the diagrams, which adopt a formal semantic, are exported to a set of metadata artefacts, in accordance to the SMP2 meta-model scheme (i.e. following the XML format for the catalogues, assemblies, packages, and schedules). Moreover, for each model package, corresponding C++ code is produced to enable classes factoring and deployment into dynamic libraries. After these artefacts and binary packages are made available via the *Configuration* database and *Model Repository*.

3.3. Simulation Editor

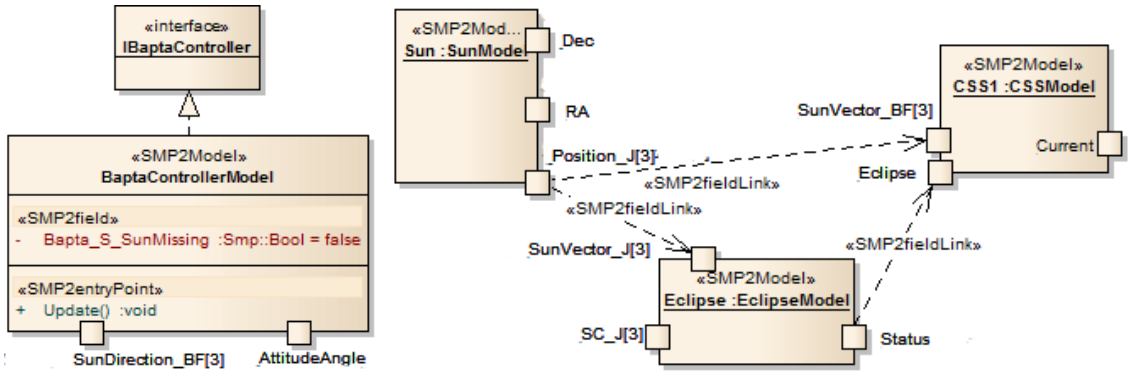
The generation of the wrapper code for the C++ models starts with the definition of their SMP2 interfaces in a model catalogue. Since SMP2 adopts concepts from object oriented programming, it is easy to adapt a Class Diagram from Unified Modeling Language (UML) for defining a catalogue. This method is used in the construction of the SDK Editor, which defines a set of stereotypes in the Enterprise Architect modelling tool to allow model definition with a class dialect.

An example of such approach is given in Figure 4-a, in which a Bapta Controller Model is described with a class diagram. Using customized stereotypes, parameters can be modelled as class attributes, entry points and operations as methods, and the input and output parameters as ports. Specialization, aggregation and composition connections can also be used to express model relations. In a similar strategy, Object Diagram can be adopted for defining the run-time instances of simulation models and their connections. In this direction, the Figure 4-b shows and example of assembly three models using field links. Based on these representations, a plugin can be used in the editor to automatically generate the wrapper code and assembly description files for configuration simulation scenarios.

4. The Accredited Model Repository

Currently, the implementation of this model repository is based on Apache Subversion, a free system for software versioning and revision control. In order to use this application as an accredited model repository, special precautions have to be taken when adding new models. In our approach, a process for quality assurance is defined taking the model bundle as input and generating a certificate and a unique identifier as output.

The certification is done based on the completion of the bundle and ensures that the documentation



(a) Definition of model

(b) Assembly of models

Figure 4: Representation of model catalogues and assembly using UML class and object diagrams.

and source code have no discrepancies found and all test cases have been executed. The source code must comply with the coding standard and both MATLAB and C++ implementation must adopt the same constant values.

A rigorous control is established to keep track of model versions and when an error is notified a report describing the bug and possible solutions is attached to the model package.

4.1. Bundle

The model bundle contains all the source code of the model package and additional information for describing the model:

- *Source code*: header, implementation or script files and makefiles needed to build the binary package;
- *Configuration files*: any data file used to configure the model in execution time;
- *Catalogue*: a XMI file containing a UML class diagram that describes the SMP2 model;
- *Manifest*: a formal description of bundle content (see section 4.2.);
- *Certificate*: a document assuring that the model has been formally verified;
- *Design Documentation*: a set of documents that describes the mathematical modelling of the simulation component, requirements, and design.
- *Test harness*: set of data used to test the model, additional test bench elements for executing the test suite and related documentation.

In addition, each model bundle is accompanied by a MD5 file, i.e. a hash value used to verify data integrity of the package.

4.2. Manifest

The manifest is a metadata file describing the contents of the bundle, which should clearly present the model, its interfaces, dependences and usage context. In the Table 3, a complete list of its fields is provided.

Table 3: Content of bundle manifest.

Field	**Description**
Identification section	
Identifier	Unique identifier
Name	Name of the model. The name have not to be unique, but all models assembled in a given simulation must be unique.
Purpose	A brief description of the model.
Version	Implementation version of the model.
Author	The author and reference of mathematical model.
Programmer	The responsible for the codification.
Date	Date of implementation.
License	End user agreement, if applicable.
Platform information section	
Language	Programming language and version.
IDE	Name and version of Development Environment used to generate the model.
Wrapper	Type and version of wrapper, e.g. SMP2.
Interface and configuration section	
Inputs	A list of models inputs, which can be input operation arguments or input parameters. The limits, units and adopted reference frames must be specified for each field.
Outputs	A list of models outputs, which can be output operation arguments or output parameters. The limits, units and adopted reference frames must be specified for each field.
Paramerters	A list of models internal parameters. The limits, units and adopted reference frames must be specified for each field.
External Dep.	All configuration data available from external sources, e.g. files or environmental configurations.
Dependencies section	
Libraries	External library usage in compilation time (e.g. matrix operations).
Models	A list of mandatory models and their identifiers that must be present in the simulation.
Providers	A list of optional models and their identifiers that could be attached to the input fields of the current model. Differently from the Models dependencies, an input field can be disconnected on run-time.
Constraints & Limitations section	
Constraints	Operation limits and validity range of model application.
Bugs	Known bugs of the model.

Table 4: Sensor Models.

Model	Inputs	Outputs	Parameters
Solar sensor	sun vector, albedo vector	current	minimum elevation mask
Solar sensor (Amz)	sun vector, albedo vector, eclipse flag	current	shadow masks.
Gyros	angular velocity vector, acceleration vector	angular velocity	noise and sensor position matrix
Gyros (Amz)	angular velocity vector, increment angle and step	increment angle	
Accelerometer	satellite position vector, angular velocity vector, date, step	acceleration vector	
Magnetometer	satellite position vector, date	magnetic field vector	
Magnetometer (Amz)	satellite position vector, date	digital sensor measure	
Star Tracker Sensor	magnitude of visible stars, FOV, noise	attitude matrix	
Star Tracker Sensor (Amz)	satellite state vector, satellite position vector, sun vector	satellite state vector	

4.3. Currently Available Models

Currently, the repository contains models of sensors, actuators, AOCS, environments, orbit, and attitude dynamics. Generic models and specific ones for the Amaznia-1 satellite are implemented in MATLAB and C++, as described in Tables 4 and 5.

In addition to models of sensors and actuators described in the above tables, the following models also have implemented:

1. Atmospheric drag;
2. Gravitational Field;
3. Moon Position;
4. Geomagnetic Field;
5. Sun Position;
6. Eclipse;
7. Solar Radiation Pressure;
8. Simple AOCS model;
9. Albedo model;
10. Date and time conversion models; and
11. Orbit and Attitude dynamics propagation.

Table 5: Actuator Models.

Model	Inputs	Outputs	Parameters
Reaction wheel	motor paramters and current	torque vector and angular moment vector	noise and position matrix
Reaction wheel (Amz)	desired torque, angular velocity of motor, time and step	torque, angular moment vector, angular velocity of motor	noise and control parameters
Coil Torque	tension applied in coil, satellite position vector and date	torque	noise and position matrix
Coil Torque (Amz)	current, magnetic field vector		
Thruster	force		
Thruster (Amz)	pressure in tank, opening valves times	torque, impulse	noise and position of thrusters

4.4. Repository Usage

The usage of models is quite simple and requires a few steps to integrate them in an existing simulation environment.

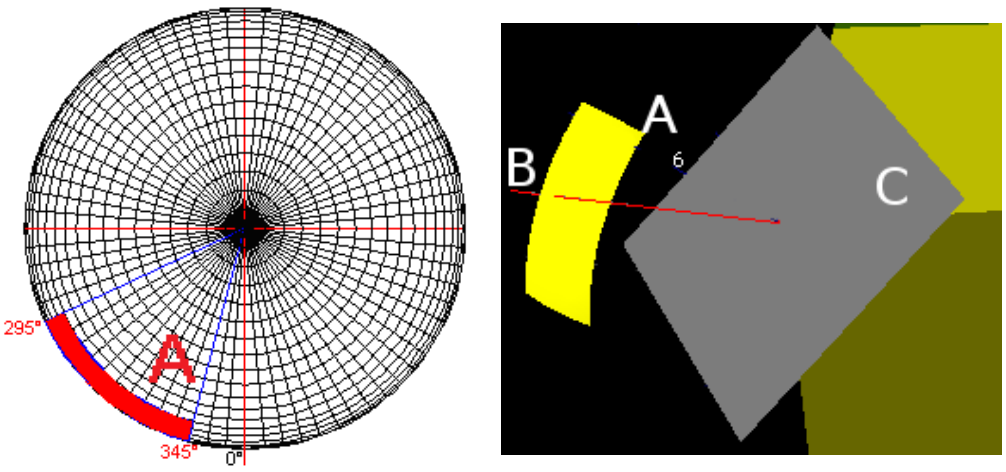
Before using a model from the repository, the user must register to get access to the Subversion and make sure that the proper development environment is installed in the target computer. It is also important to ensure that he/she agrees with the user license.

The integration of MATLAB script does not require further configuration, unless external dependences exists.

The C++ models, even when accompanied by a SMP2 wrapper can be used as a regular class, but their integration in the infrastructure with a formal assembly is encouraged. By importing the XMI file containing the catalogue into the UML editor, it will be possible to create new instances of the model and connecting it to other models in an assembly definition. After the entry points of the model have been scheduled and the binary package built, the model can be dynamically loaded in a running simulation.

5. Case Studies

In this section, the flexibility on reconfiguring repository models is demonstrated in various simulation scenarios. The first set of simulations, in subsection 5.1., is applied to the Amazonia-1 mission to analyse the placement of coarse solar sensor and validate the on-board algorithm for sun determination. The second subsection 5.2. a study for estimating the propellant consumption in the SabiaMar spacecraft lifetime, a future mission to be based on MMP. Finally, the subsection 5.3. describes a study for orbit decay and nadir pointing, based on models from the repository.



(a) Definition of shadow mask

(b) 3D scene with CSS6

Figure 5: Definition of Coarse Solar Sensor's shadow mask caused by satellite structure interference.

5.1. Coarse Solar Sensors of Amazonia-1

In Amazonia-1 mission, one method for determining the Sun direction consist on processing the data from eight Coarse Solar Sensors (CSS), which are photovoltaic cells that generate an electrical current accordingly to the light incident upon them. The sensors are carefully placed so at least two cells receive Sun light, regardless the direction it comes from. Their positions can be roughly described as the eight vertices of a rectangular parallelepiped formed by two cubes, one representing the platform structure and other the payload. The boresight axis of each sensor is aligned with the cube's diagonal (i.e. the diagonal formed with their opposite vertex).

The CSS model developed in work is very simple and produces an output current based on the angle of light sources (i.e. Sun or Earth's Albedo) incident over sensors plane. Moreover, shadows on the sensors produced by mechanical interference from other structural elements (e.g. antennas or solar array generators) have been also modelled as masks on the sensors field of view. For the sake of simplification, these shadow masks are considered to be static and have been pre-computed in a CAD tool. Thus, the shadows caused by the solar array generators are modelled as the worst case of their orientation, i.e. as if the interference were done by a cylindrical object. In the Figure 5, for instance, the shadow mask for the CSS 6 and a corresponding 3D representation are shown

Following the procedure described in section 3., two components of the CSS Model are configured in the repository. The MATLAB implementation is provided in a single m-file and contains 116 logical Lines of Code (LOC) (i.e. ignoring commented lines and non-executable code). The C++ implementation consists in a composition of three classes of models, as illustrated in the UML class diagram from the Figure 6. The component interface is handled by the *CSSAsb* class that is composed of multiple CSS models, which in their turn can also be composed of several models for computing shadows. This generic design, in addition to the adoption of algebraic library, led to a very lean implementation with only 60 logical LOC. In total, the component source code has 389 logical LOC, including the SMP2 wrapper auto generated from the model catalogue.

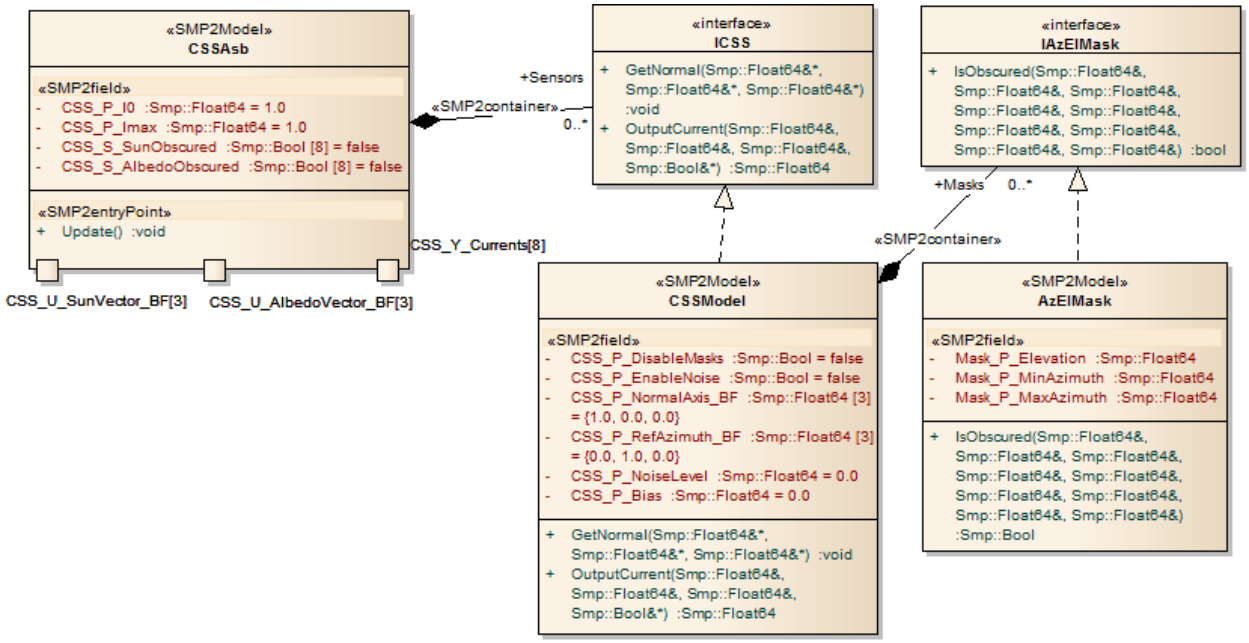


Figure 6: Simulation package modelling for the Coarse Solar Sensor model.

5.1.1. Coverage Analysis of Coarse Solar Sensors

The main goal of the first simulation using the CSS model is provide a coverage mapping of the sensors with their combined field of view. Due to their geometrical arrangement, four sensors should always see the Sun at the same time, if there were not shadows produced over them. Hence, a better study is necessary to understand the impact of the shadow masks on the sky coverage of the sensors. A secondary objective of this simulation is to provide insights of the sensors output as a function of the Sun vector input and support the mathematical models validation.

For conducting these analyses, the CSS implementation in MATLAB is reused from the repository and a new script is written for describing the simulation scenario and executing the simulation. In addition, the auxiliary script automatically generates a set of Sun vector inputs for stimulating the CSS models, varying the light direction for the $4\pi rad$. Including plot commands, the total number of logical LOC in the auxiliary script is 31, which represents about 21% of the simulation code.

The result of the study is summarized in the Figure 7 and shows the number of solar sensors that receive direct light from the Sun at the same time, considering different direction all around the spacecraft. It can be observed that in most cases, four sensors receive direct light and most shadows are caused by the solar array generators (Figure 7-A). The structure of launcher adapter also produces an important shadow (Figure 7-B) and a smaller region of the sky is affected by the presence of payloads antenna (Figure 7-C). In the worst case, two sensors are obscured by a combination of shadows.

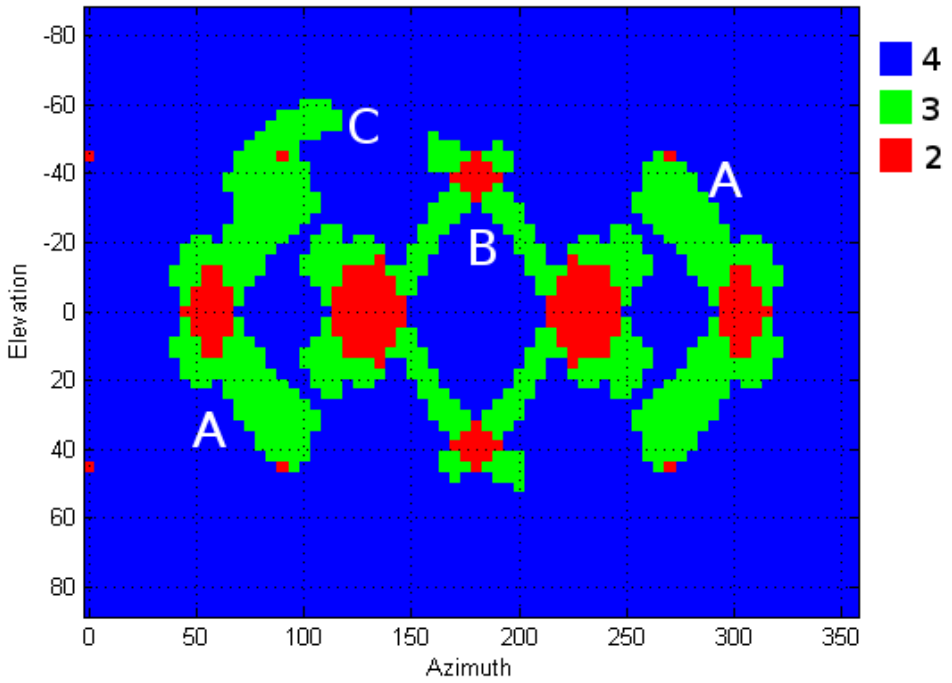


Figure 7: Coverage mapping of eight Coarse Solar Sensors considering their shadow masks.

5.1.2. Verification of Sun Determination Algorithm

The second use case of CSS Model aims to validate a Sun Determination Algorithm that is embedded in the on-board computer. In this scenario, the C++ component is reused from the repository and connected with the flight software. The model assembly is depicted in Figure 8, where in the configuration parameters for the scenario can be defined for each model instantiation.

A new model named *SunDeterminationModel* implements the embedded algorithm, which has a total of 138 logical LOC (from those 84 is the wrapper source code). Using a customized graphical user interface (Figure 9), the simulation conductor is able then to inject arbitrary Sun vectors to the CCS model and compare it with the vector computed on-board.

5.1.3. Verification of Sun Determination Algorithm with orbit and attitude dynamics

In this third presented simulation, the behaviour of CSS Model and the Sun Determination Algorithm is assessed in a more complete scenario, including additional models for the repository for simulating the spacecraft orbit and attitude dynamics, the Sun position and irradiance, and eclipse (i.e. caused by Earth blocking the Sun as viewed from spacecraft). In this scenario, the simulation engineer can guide the spacecraft attitude and orbital position and analyse the effect on sun determination algorithm. The model instantiation and connection for this scenario is described by the assembly in the Figure 10, in which 569 new LOC are added from the repository, achieving 87% of reused code.

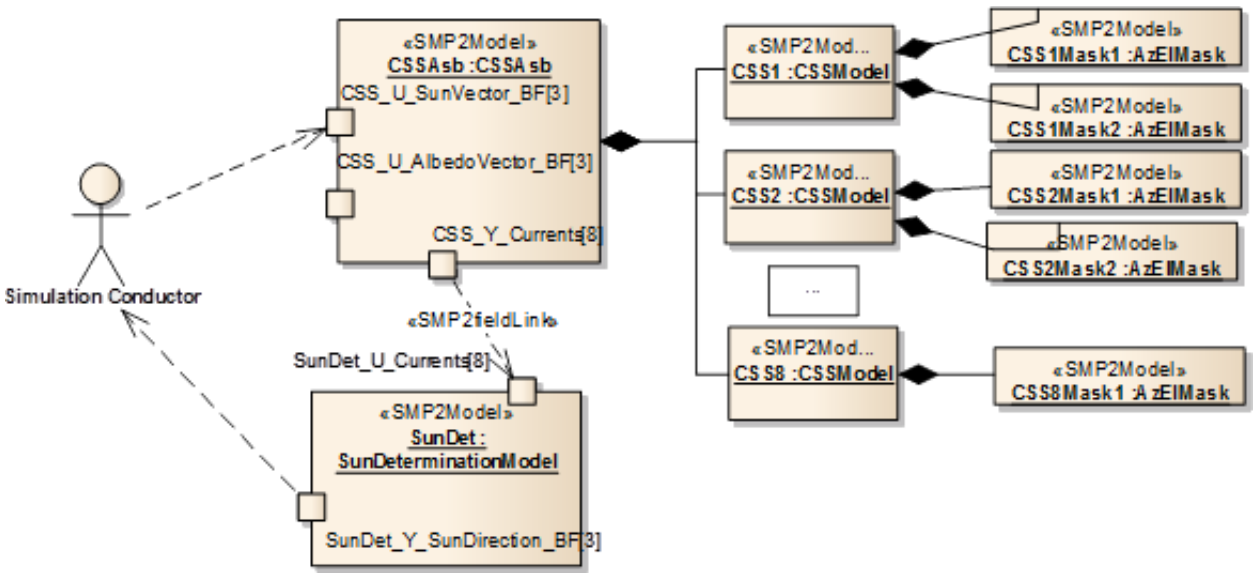


Figure 8: Model assembly specified to connect the Sun Determination Algorithm model with the eight Coarse Solar Sensors.

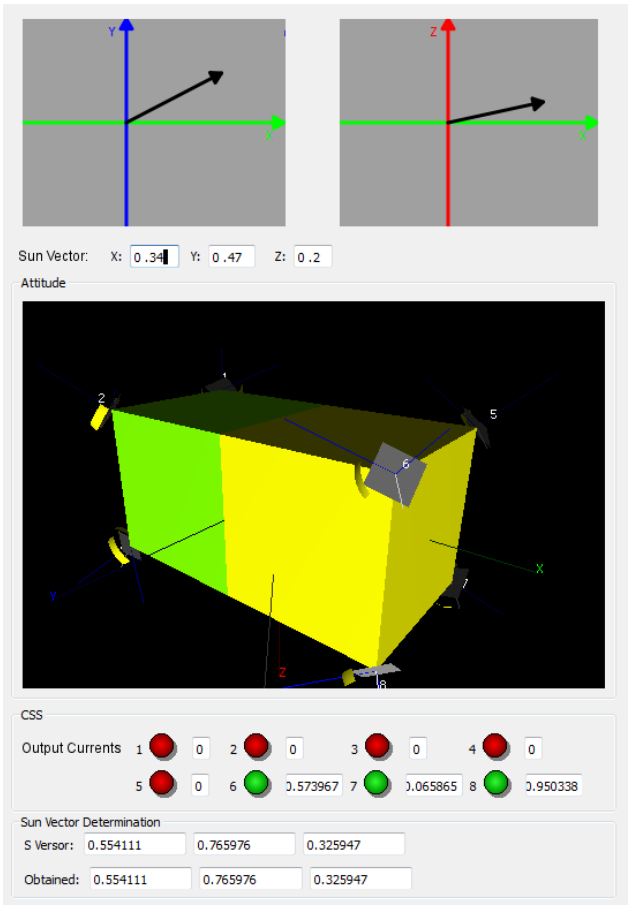


Figure 9: Graphical Interface used by the simulation engineer to validate the Sun Determination algorithm.

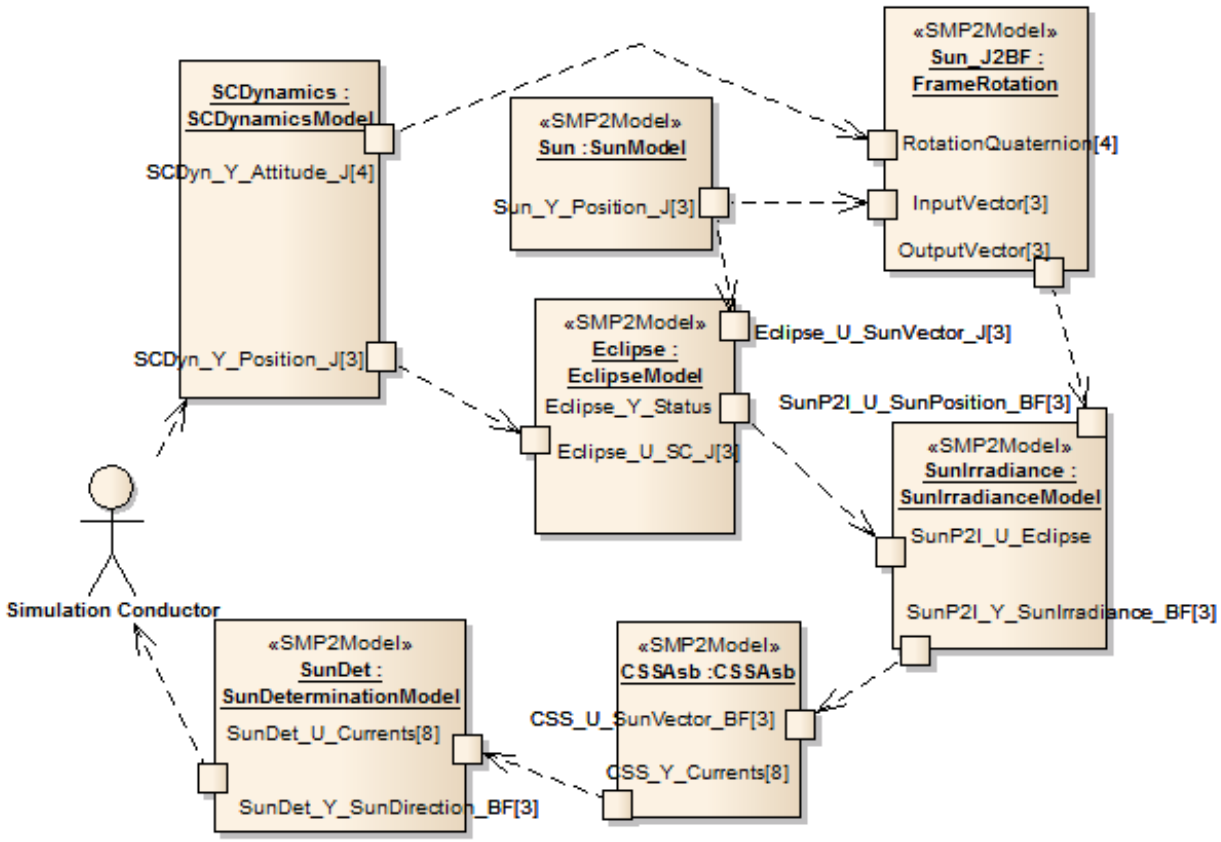


Figure 10: Model assembly for configuring the verification scenario of Sun Determination algorithm with orbital and attitude dynamics models.

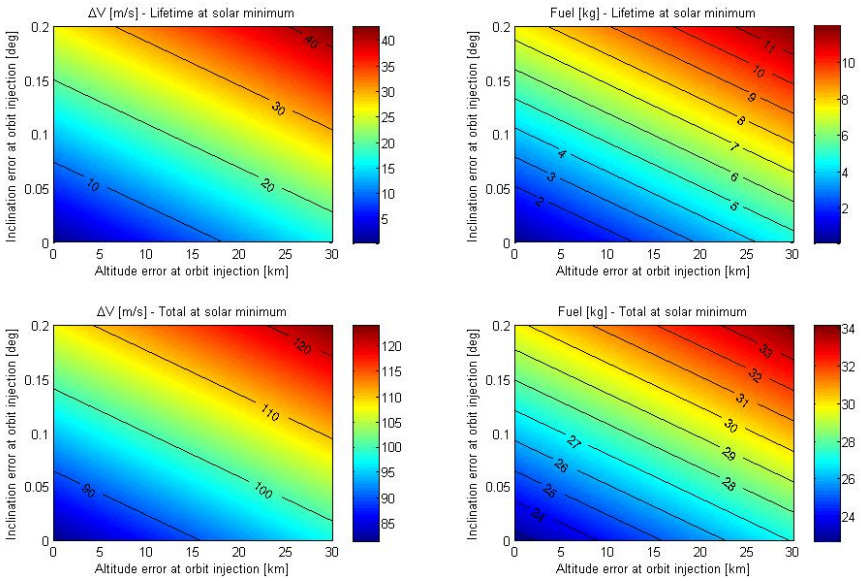


Figure 11: Fuel Budget simulation

5.2. Estimation fo Propellant Consumption for the *SabiaMar* mission

The simulation of the fuel consumption of the *SabiaMar* satellite was performed using the parameters from satellite and environment models of atmospheric drag and solar activity. In total, the simulation is performed with 119 LOC of MATLAB scripts, from which 41 are reused from the repository.

In this analysis it is possible to verify whether the amount of propellant is sufficient for the lifetime of the mission. The Figure 11 shows a result example from this simulation.

5.3. Orbital Decay Estimation and Nadir Pointing Analysis

Another activity performed that did reuse models of the repository, was the orbital decay of a satellite study and the control system with the AOCS pointing to nadir using reaction wheels.

The Figures 12 and 13 show a result example for this simulation, in which all models of environment, attitude and orbit propagator, reaction wheel actuator and the star tracker sensor was reused from the repository.

6. Conclusions and Future Works

In this work, an initiative for enhancing the level of software reusability in Spacecraft Dynamics Simulators has been presented in the context of Amaznia-1 mission. Along with the establishment of guidelines for building flexible simulation facilities, an Accredited Model Repository (AMoRe) has been created to configure simulation models and manage their reuse in different use cases.

The assessed level of software reuse encourages the adoption of the methodology for future projects of SDS at INPE. The case studies has demonstrated the benefits for configuring incremental

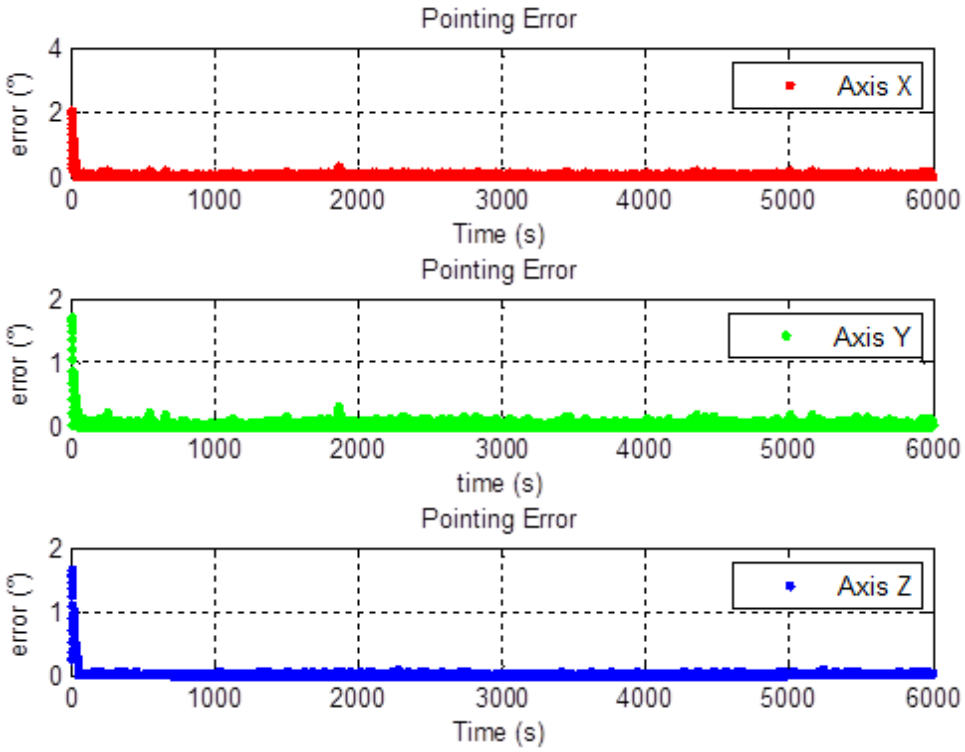


Figure 12: Attitude error

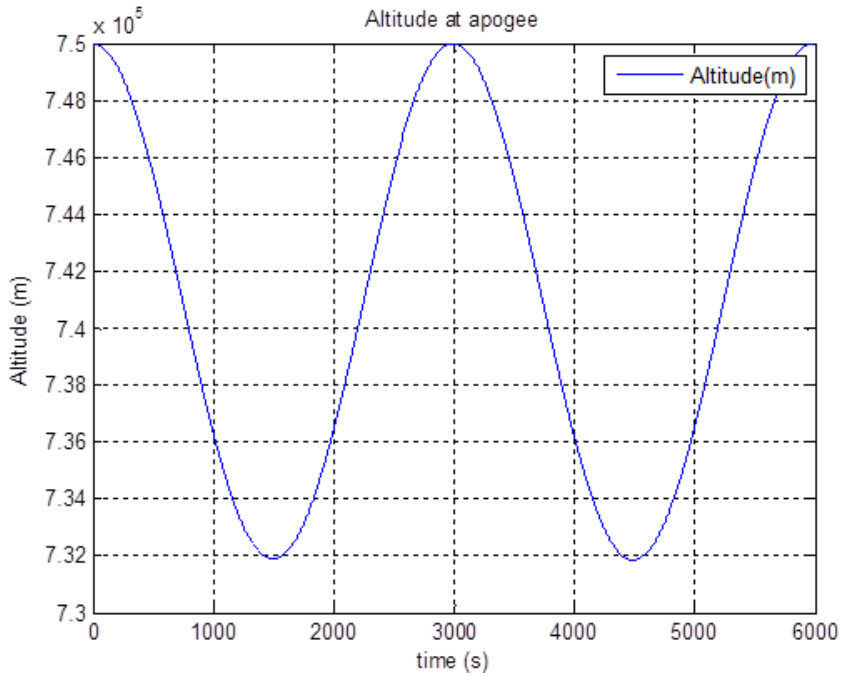


Figure 13: Apogee Altitude variation

simulation scenarios from existing models, wherein the proportion of additional code required for test harness or customization is minimal in terms of LOC. Further, the model distribution in standardised packets has facilitated the access to simulation artefacts in a controlled way which may encourages new users to download models and to contribute with the repositorys growth.

In addition to that, the adoption of SMP2 concept for building the simulation infrastructure has brought significant flexibility for reusing models and rapidly reconfiguring then for various scenarios. Regarding the process for building simulation models, the algebraic library has proven to be a fundamental component for increasing code readability and avoid programming errors. In this step, the adoption of a unique reference of constants has contributed for reducing output discrepancies between MATLAB and C++ implementations.

Furthermore, the auto-generation of wrapper code for C++ models has played a crucial role on alleviating the programming tasks and encouraging the insertion of new models into the repository. In this direction, the development of additional automating tools will be investigated in future works to further reduce the implementation effort. A critical point in this process would be the automatic conversion of MATLAB script to C++ code, taking as example the *Model-Oriented Software Automatic Interface Converter* (MOSAIC) utility [13].

Another topic for improvement in future works is the construction of a web interface for browsing the repository, where the users may search for models in a database. In this system, the usage feedback could also be registered, based on lessons learned from real missions.

Finally, a key point for the repository realisation is to promote users perception on the reliability of offed models, increasing the availability of information on model accreditation processes. This is believed to be one of the most important barriers on model reuse and changing this organizational culture would definitely be a good opportunity to configure legacy models and improve the knowledge management in simulation projects.

7. Acknowledgment

The authors gratefully acknowledge the support of this work by Brazilian "National Council for Scientific and Technological Development (CNPq), under Grant No. 560116/2010-1.

8. References

[1] John, R. H. S., Moorman, G. J., and Brown, B. W. "Real-Time Simulation for Space Station." Proceedings of the IEEE, Vol. 75, No. 3, pp. 383–398, 1987.

[2] Agre, J. R., Clarke, J. A., Atkinson, M. W., and Shahnawaz, I. H. "Computer simulation of communications on the space station data management system." pp. 809–818. Conference on Winter simulation, Atlanta, 1987.

[3] NASA. "Space Station Simulation Computer System (SCS) study for NASA/MSFC: overview and summary." Tech. Rep. TRW-SCS-89-T7, Huntsville, 1989. Technical Report.

[4] Liceaga, C. A. "SPASIM: A Spacecraft Simulator." Tech. Rep. NASA–97–isuac–cal, Hampton, 1997. Technical Report.

[5] Betts, B. J., Mundo, R. D., Elcott, S., McIntosh, D., Niehaus, B., Papasin, R., and Mah, R. W. "A Data Management System for International Space Station Simulation Tools." pp. 500–504. IASTED International Conference Applied Simulation and Modeling, Crete, 2002.

[6] Papasin, R., Betts, B. J., Mundo, R. D., Guerrero, M., Mah, R. W., McIntosh, D. M., and Wilson, E. "Intelligent Virtual Station." International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2003.

[7] Freund, E., Rossmann, J., and Tumer, C. "Application of Robotic Mechanisms to Simulation of the International Space Station." International Conference on Intelligent Robots and Systems, 2003.

[8] Pisanich, G., Plice, L., Neukom, C., Flückiger, L., and Wagner, M. "Mission Simulation Facility: Simulation Support for Autonomy Development." AIAA Aerospace Sciences Conference, Reno, 2004.

[9] Nesnas, I. "CLARAty: A Collaborative Software for Advancing Robotic Technologies." p. 7. NASA Science and Technology Conference, University of Maryland University College, Adelphi, 2007.

[10] Hammers, S. R. "Virtual Satellite." NASA Tech Briefs, p. 1, 2008.

[11] ECSS. "Space engineering: System modelling and simulation." Tech. Rep. ECSS-E-TM-10-21a, Noordwijk, The Netherlands, April 2010. Technical Memorandum.

[12] ECSS. "Space engineering: Simulation modelling platform - Volume 1." Tech. Rep. ECSS-E-TM-40-07, Noordwijk, The Netherlands, January 2011. Technical Memorandum.

[13] Lammen, W., Nelisse, A., and ten Dam, A. "MOSAIC: Automated Model Transfer in Simulator Development." Workshop on Simulation for European Space Programmes (SESP), Noordwijk, 2002.