Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

The present work was submitted to the Institute of Software and Tools for Computational Engineering

FMI Based Spacecraft Simulation Along Life Cycle Phases

Master Thesis

Presented by

Sally Bebawi
Matr.-Nr. 384684

Supervised by

Prof. Dr. Uwe Naumann
Prof. Dr. Matthias Müller
Philipp Martin Fischer
Markus Flatken

Aachen, May 2, 2019

Contents

I Acknowledgements

First and foremost, I want to thank God for the strength and encouragement especially during all the challenging moments in completing this thesis. I am truly grateful for the exceptional love and grace during this entire journey.

I extend my gratitude for the great deal of support and assistance I received from my supervisors at RWTH Aachen University throughout the thesis. I would first like to thank Prof. Uwe Naumann whose expertise was invaluable in the formulating of the research topic and methodology in particular. Moreover, I would also like to thank Prof. Matthias Müller who has kindly agreed to serve as my second examiner.

Furthermore, I am very grateful for the support and guidance I received from my supervisors at DLR, Braunschweig. Thus, I would like to express my sincere gratitude for my supervisors, P. Fischer and M. Flatken, for their valuable guidance. You provided me with the tools that I needed to choose the right direction and successfully complete my dissertation.

In addition, I would like to thank my family for their wise counsel and sympathetic ear. Finally, there are my friends, who were of great support in deliberating over our problems and findings, as well as providing happy distraction to rest my mind outside of my research.

Simulations were performed with computing resources granted by the German Aerospace Center (DLR) under project *Virtual Satellite*. I am grateful for providing me the opportunity to conduct my research and dissertation *FMI Based Spacecraft Simulation Along Life Cycle Phases*.

II Abstract

In notion of the recently encountered collisions and the crescent population of objects in the low Earth orbit, a significant motivation arose to develop effective and novel techniques to estimate the behavior and reactions of the spacecraft in a simulation environment. Early research efforts have used the Simulation Model Portability (SMP2) space-related standard in spacecraft simulation development. However, the simulators compatible with this standard only provide a C++ reference implementation. This thesis proposes utilization of the so-called Functional Mock-up Interface (FMI) standard in the spacecraft simulation development. It provides encapsulation of models and allows co-simulation at early design stages. Furthermore, this FMI simulator is leveraged to incorporate optimization algorithms within the FMU infrastructure. Benefits of such incorporation are exact determination of the required components before purchasing and effective emulation of a mission maneuver former to the spacecraft launching. This thesis provides detailed methodologies for developing a FMI spacecraft simulator and incorporation of the accumulation of Jacobian matrices optimization technique for sensitivity analysis of the simulation models.

1 Introduction

Simulation Engineering is the field of developing a model that imitates the main characteristics and behaviors of a specific abstract or physical process or system. Basically, a real-world system is represented by a model and its operation over time is represented by the simulation. In other words, software models recreate the hardware behavior when using the real system. It is impossible to use the hardware components directly due to costs, safety, or operational constraints. Spacecraft simulation is an integral part of mission planning, operations, training, and systems engineering [1].

Real-world systems or products function by assembling a variety of parts that communicate together in a complex way. Thus, creating a virtual product could be done in the same manner by assembling a variety of models representing the physical laws and control frameworks. The Functional Mock-up Interface (FMI), developed by Daimler AG, is a standard that has been used in simulations to represent the operation of a real-world process or system. It is a tool independent standard to support both model exchange and co-simulation of the implemented models [11]. The FMI standard in this manner provides a way for model-based development of systems. FMI has been utilized and adapted by several modeling and simulation software vendors for vehicle engineering such as the automotive and shipyard industries [13]. The usage and adaptation of FMI as a standard for automotive embedded world has been previously investigated [14]. However, investigating it in the aerospace industry has not been investigated yet.

Basically, Spacecraft missions are managed in so called phases from 0/A to F. Where 0/A and B focus on the design, Phase C and D focuses on the implementation, and E finally on the operation before phase F takes care of the disposal [3]. Mission Concept Simulator (MCS) and Functional Engineering Simulator (FES) are simulators used early in the design process around phases 0/A and B which require coarse and more precise models respectively [6, 7]. Furthermore, Software Validation Facilities (SVF) is used later in the process and requires highly detailed models. Still all simulators have some heritage, and their models are finally replaced, bit-by-bit, by real hardware until the real satellite is tested in the loop [7].

The Virtual Satellite tool, developed by DLR, has been used in driving simulator configurations. Virtual Satellite is addressing the whole lifecycle of the spacecraft and has successfully been used in Phase 0/A/B [5]. The first task of system engineering is the model-based system engineering which could be realized via the Virtual Satellite built upon the Eclipse Java framework; whereas the implemented FMI simulator in this thesis utilizes the C programming language. Furthermore, each of the simulation models is encapsulated in a FMU container and developed via XML-file and compiled C code [12]. The simulation models implemented throughout the thesis cover simple environmental and dynamic models, as well as a generic state machine to simulate equipment, and a generic On Board Computer (OBC) plus interface models to simulate proper command handling. Simple Orbit models could be used together with a ground contact model to evaluate optimal navigation scenarios [17].

A basic solution to several scientific and engineering problems is the precise calculation of derivatives for a mathematical model. Based on the chain rule, the computation of these derivatives or Jacobians can be done by multipying the locally computed derivatives of each mathematical model [21].

The forward vector mode of the Automatic Differentiation (AD) [22] can be used to accumulate the derivatives. The described optimization approach in computing the Jacobian based on the chain rule was suggested in [23]. Furthermore, this accumulation approach and a number of heuristic strategies have been adopted in many scientific researches such as [24].

An approach to compute the Jacobian matrices has been described in [25]. The introduced approach is to evaluate the Jacobian matrices by calculating the chained products of local extended Jacobians.

Based on the above-mentioned description, the following research questions will be tackled throughout the thesis.

- Possibility of the incorporation and adaption of the FMI standard in developing a spacecraft simulation framework.
- Investigating the incorporation of design optimization algorithms and techniques on basis of accumulation of Adjoint methods within the FMI standard framework.

The thesis is organized as follows. Chapter 1 presents the introduction. Chapter 2 includes an overview of the spacecraft system engineering and simulation. Furthermore, it introduces the Functional Mock-up Interface (FMI) based simulation standard framework. In addition to that, it also gives an overview of the numerical optimization as well as several optimization algorithms and techniques in the context of spacecraft simulations. The system design and implementation of the FMI simulator as well as the incorporation of design optimizations into the simulator are demonstrated in Chapter 3. Furthermore, Chapter 4 presents the evaluation of the FMI-based simulator and the results from incorporating the optimization techniques into it. Finally, the thesis is concluded and the future work are suggested in Chapter 5.

2 Literature Review and State of the Art

This chapter delves into the literature review required for the thesis work. It is organised as follows. The first section introduces system engineering in the context of spacecraft industry. The next section contains the standards used in spacecraft system simulation. In addition to that, an overview of the so-called Functional Mock-up Interface (FMI) with its two variants and benefits of using FMI in the system simulation domain are introduced in the third section. Finally, design optimization techniques such as the parameters sweeps technique and the Gradient Descent optimization algorithm are introduced which are essential concepts that are used in the thesis to investigate the possibility of incorporating design optimization techniques into the FMI-based spacecraft simulator.

2.1 Spacecraft Systems Engineering

Systems engineering is a field that focuses on how to structure and manage complex frameworks or systems over their life cycles. Model-Based Systems Engineering (MBSE) is a methodology that focuses on performing systems engineering by utilization and exchange of domain models. MBSE creates the simulator configuration. It aims to achieve a fully integrated system simulation and validation of the integrated system's operations and functionalities [2]. MBSE is developed in many places such as the Virtual Spacecraft Design (VSD) developed by ESA [4] and Virtual Satellite tool developed by the "Simulation and Software Technology, Software for Space Systems and Interactive Visualization" team at the German Aerospace Center (DLR) [5].

System engineering in the spacecraft industry is the process of designing and simulating spacecrafts or spacecraft's subsystem. Space agencies such as the National Aeronautics and Space Administration (NASA) and the European Space Agency (ESA) are typically responsible for the development of spacecraft systems. Standard terms and processes have been introduced and adapted by all partners to ensure unambiguous communication and efficient usage of all documents [2].

The life cycle of space systems are managed in so-called phases from 0/A to F [1]. Figure 1 presents the phases as well as the key event of each phase. The key event or outcome of Phase 0 is the approval of mission concept and feasablility based on advanced studies. The next phase results in reaching a preliminary agreement regarding the mission and system definition. After that, the approval of a preliminary design and the contract signing is done in phase B. Realizing this point denotes that the mission formulation is over and the next phase would advance with the implementation.

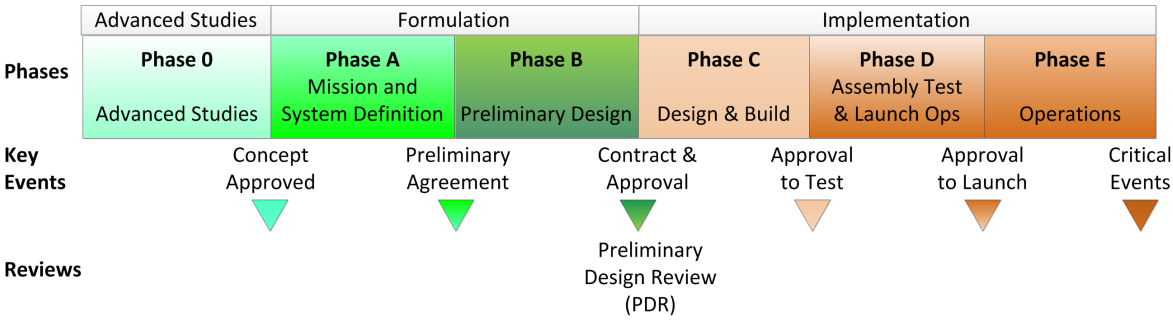

Figure 1: Spacecraft Lifecycle Phases

Phase C focuses on design and building the spacecraft. The system should be approved for testing by the end of this phase. The subsequent phase then holds assembly tests and launch operations before reaching the point where the spacecraft launching is approved. Finally, the actual operations and mission is performed at phase E until the mission is accomplished or critical events occur leading to the spacecraft disposal in space [1].

In particular between phases B and C, the Preliminary Design Review (PDR) is held. It aims at discussing whether the project is ready to move from phase B to C. After a successful review, contracts are made with suppliers, and actual hardware gets ordered and built. Design changes at this point in time become expensive due to contractual changes and cross-effects in the overall design [1].

Basically, MBSE stores the spacecraft design in one common system model that can be used to drive simulations e.g. by a Functional Engineering Simulators (FES), which is explained in the next Section of this Chapter. The configuration of such simulators is directly generated from the system model [6].

2.2 Spacecraft System Simulation

Simulation is a main activity that supports the specification, design, verification and operations of space systems. Several use cases across the spacecraft development lifecycle can be supported by system modeling and simulation [7], including activities such as system design validation, software verification and validation, spacecraft unit and subsystem test activities, etc.

Significant efforts are done in projects to improve the simulation and test facilities. It has been broadly perceived that simulation forms a pivotal part of the system engineering procedure. However, there is an inclination to consider simulation as a supporting activity. The utilization of simulation by the distinctive disciplines and throughout project phases is sometimes not coordinated with facility specific solutions that is being developed to address specific needs. It is recently recognized that it would bring considerable advantages if a more coordinated and consistent approach to the development of simulation products across project phases is followed. This would advance the best utilization of simulation in the system engineering process in order to minimize the overall space program risk and cost [10].

Simulation has become an increasingly vital activity in the support of a wide scope of engineering and operational activities throughout the lifecycle of a spacecraft development. Various simulation and test facilities are procured and used over all space programs. There is a lot of commonalities between the infrastructure and models developed for each of these facilities. These commonalities has been documented in the ECSS-E-TM-10-21 "System Modeling and Simulation" Technical Memorandum [7]. This technical memorandum provides a guide to system engineers on the best way to utilize simulation in supporting their system engineering tasks. Moreover, it presents the facilities and their related high level requirements.

A lot of rational and incremental test and simulation facilities from Phases A to E provide the software engineering functions. The 'Virtual System Model' is the core of these simulation facilities emulating the behavior and functions of the complete system. The virtual system model includes the simulation infrastructure and the models of the space system, including the ground segment and space environment. The developer should reflect the different configurations explicitly in the design, development and validation plan. Moreover, the virtual

system model should be considered as a part of the overall model. [7]

Each facility represents an instance or a subset of the space system. Consequently, these facilities should be integrated to the overall model in the system level, including both virtual and physical models. Furthermore, the System Database is gradually populated and validated over these steps, practically from Phase C onwards [7].

ESA and several stakeholders in the European Space Industry have developed a standard for simulation models named Simulation Model Portability (SMP). The current available and used version is SMP2. It defines certain mechanisms that allows connecting and exhchanging information between components. Only a C++ reference implementation is available for the realization of these SMP2 concepts [8].

The ECSS-E-TM-10-21 technical memorandum explains simulation facilities and their purposes, such as the Functional Engineering Simulators (FES), the Software Validation Facilities (SVF), the Training, Operations and Mainetenance (TOM) and others. These facilities are used to support the analysis, design and verification activities on system level [7].

The purpose behind these FES, SVF and TOMS facilities is to provide a basis upon which a consistent approach can be followed in the simulation products development across project phases. Depending on the system engineering tasks being executed, a certain simulation facilities is required during different phases of the project. Figure 2 summarizes the exact point in which each facility is required throughout the project lifecycle. Furthermore, the figure displays the model resuse across the phases as well as the expanding of the System Database along the lifecycle [7].

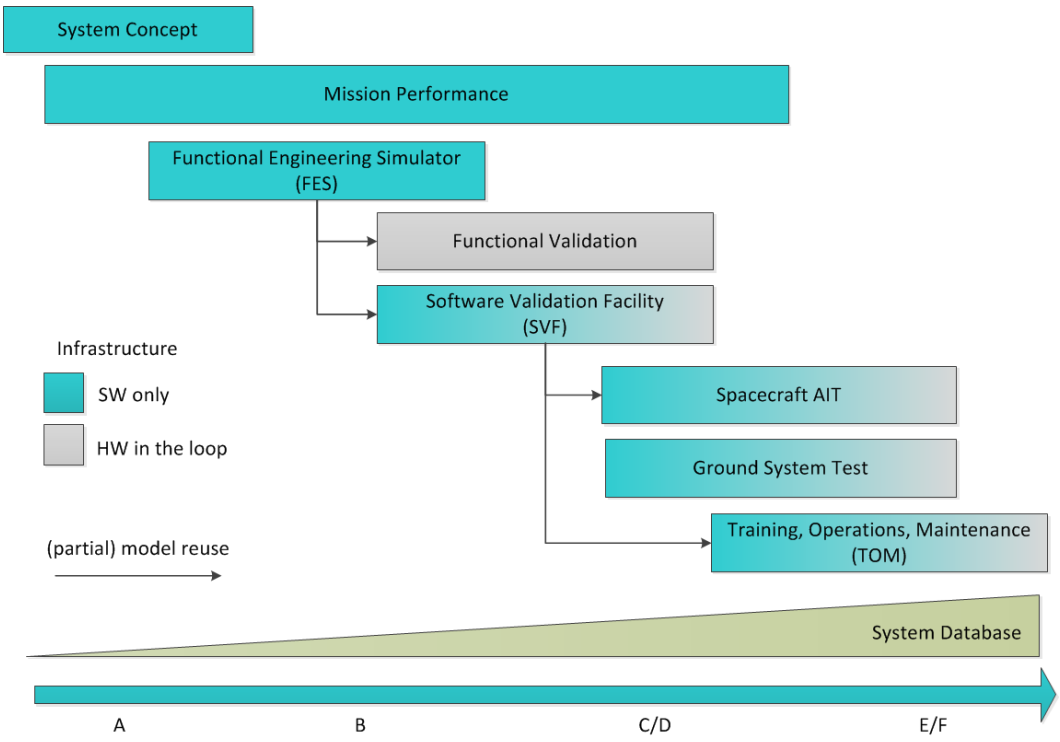

Figure 2: FES, SVF and TOM Facilities across the Life-cycle Phases

The following subsections present the most common simulation facilities supporting the spacecraft life cycles which are the Functional Engineering Simulators (FES) and the Software Validation Facilities (SVF) Simulators.

2.2.1 Functional Engineering Simulators (FES)

Functional Engineering Simulator (FES) is a facility set-up that supports the validation of ground based and critical algorithms of the On-Board Software (OBSW). This validation includes the Guidance, navigation and control systems (GNC) as well as the Attitude and Orbit Control Systems (AOCS) [7].

The FES configuration includes the functional representation and models of the real system required for the algorithms validation. However, it does not have to contain representation of the real interfaces, protocols or data handling subsystem. In other words, the functional models implemented are a representation of the behavior of real elements [7].

The architectural and functional interfaces of the system design should be included when developing an FES-based simulator. Furthermore, the simulation capability to evaluate engineering requirements and algorithms performance should be provided. In other words, the FES should be configured in a way that would make it easier for the system designer to instantiate and configure elements [7].

2.2.2 Software Validation Facilities (SVF) Simulator

The Software Validation Facility (SVF) is a complete representation of the functional and performance aspects of a simulation model of the spacecraft hardware [9]. Furthermore, it should allow a proper execution and validation of the OBSW. Moreover, it provides the essential spacecraft payloads and environmental simulation in order to properly execute the software in a closed loop [7].

Validation of the OBSW should be performed in a representative context to the spacecraft system in space and the ground interfaces [9]. The upper layers of the validation, which are related to AOCS, data handling, monitor and control of the payload equipment, are called the application software. On the other hand, lower layers, which are related to the interface between the OBSW and the OBC, are called basic software. The above mentioned validation is approved by the OBSW test and debugging capabilities of the SVF [7].

Fundamentally, the main goal of an SVF is to validate the OBSW. This target can be divided into several tasks which are OBSW integrated testing, OBSW parameter settings (e.g. AOCS), OBSW functional validation in open then closed loop, OBSW HW / SW interface verification (using Hardware in the Loop (HITL) SVF) in open loop and OBSW performance and robustness testing in closed loop [7].

Figure 3 shows how does the system engineering information in the data bases increase over time and how the FES and SVF simulators can be derived. As previously mentioned, the preliminary system design is realized by the end of phase B. The simulator would initially be in a low-fidelity FES form. Further along the project, data would continue until it reaches the SVF form. In other words, successive to performing the PDR between phases B and C, the project is then approved for implementation.

The actual more-detailed SVF facility would be carried out by implementing and building the spacecraft. Nevertheless, parts of the SVF share some data heritage with the FES. The SVF acts as the basis for hybrid benches which allows incorporating actual hardware into the simulation loop [9].

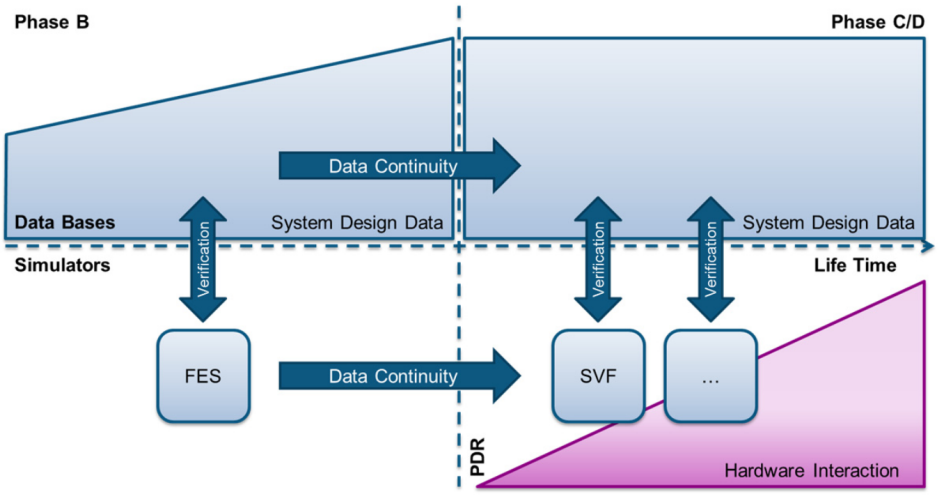

Figure 3: Data Continuity for Data Bases and Simulators
(Fischer, Eisenmann and Fuchs, 2014 [6])

2.2.3 Spacecraft Simulation Models

The development of a spacecraft undergoes the phases presented in Section 2.1. The space-related simulation standards and processes described in Section 2.2 are followed to develop the spacecraft from being just a concept until it becomes a fully simulated and developed spacecraft. The overall system architecture of a spacecraft simulation starts with the MBSE which uses modeling to analyze and realize the systems engineering lifecycle. The outcome of this model-centric MBSE approach is supporting the system requirements, analysis, design and applying V activities. Beginning with the conceptual design phase and advancing throughout development and later lifecycle phases. Based on this formalized application of modeling, the simulation and development of a spacecraft is guided [2].

Figure 4 shows the general main components of any spacecraft simulation. The OBC is the main controller that controls the spacecraft. It reads the outputs generated from the environmental simulation models such as the Solar Flux affecting the spacecraft through its sensors, as well as the Newton's gravitational force or any other forces exerted on the spacecraft. Accordingly, the OBC commands the spacecraft to change its dynamics, such as changing its orientation or its current position. For instance, this task is accomplished by commanding the Reaction Wheels (RWs) to exert a certain amount of required torque in a specific direction.

In order to execute a simulation for a spacecraft mission, several models needs to be implemented that would simulate the different factors affecting a mission such as simple environmental and dynamic models, a generic state machine to simulate equipment, and a generic On Board Computer (OBC) as well as an interface model to simulate proper command handling.

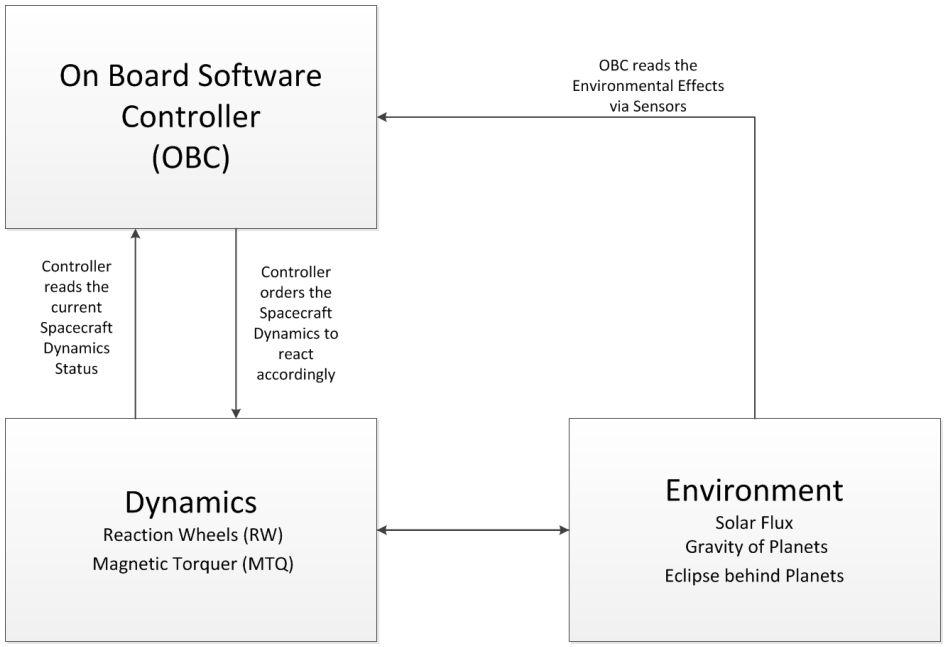

Figure 4: Diagram of the Simulators Components

The theory of the different models that have been implemented throughout the thesis in order to imitate the simulation environment are expained in more details in Appendix A.

2.3 Functional Mock-up Interface based Simulation Framework

A standard infrastructure by the name of Functional Mock-up Interface (FMI) has been developed [11]. The FMI development was initiated by Daimler AG in 2010 with the goal of improving the exchange of simulation models between suppliers and Original Equipment Manufacturers (OEMs). The latest available version of FMI is version 2.0 [12], which has been released in July 2014, which is the version that is being used in the thesis work.

The simulated models of a system are usually developed by different teams using various modeling and simulation environments. In order to simulate a complete system, its different models have to interact with each other. Thus, if different models in a system are developed by different teams using multiple different simulation tools, where n is the number of tools, then n squared interfaces would be needed to successfully interact and simulate these models together [12].

2.3.1 FMI Variants

FMI is a standard used for model exchange and co-simulation of dynamic models. The FMI standard provides a way to interface models from different simulation tools. The communication between the models is performed via the master/slave method, in which one device or process has unidirectional control over one or more other devices. The direction of control between an established master/slave relationship is always from the master to the slave. Communication between the master and a slave takes place at a discrete set of time instants, called communication points [12].

Model exchange and co-simulation are two FMI variants. The main goal of FMI for model exchange (Fig. 5, part a) is to generate a dynamic system model that could be reused by

other modeling and simulation environments. The models are C based and are described using differential, algebraic and discrete equations with different events such as time, state and step events [12].

On the other hand, the aim of FMI for Co-Simulation (Fig. 5, part b) is to provide an interface standard for simulation tools coupling in a co-simulation environment in which data exchange can only be done through discrete communication points. The subsystems are solved independently by a separate solver for each subsystem between two communication points. A simple or a sophisticated master algorithm, independent from the FMI standard, is used to control the data exchange between the slaves (i.e. all simulation solvers) [12].

More precisely, FMI is the open standard that exchanges simulation models between the different tools in a standardized format. These simulation models could be exported or developed as a model exchange or a co-simulation Functional Mock-up Unit (FMU) that adheres to the FMU standard. Solver integration is performed outside of the FMU for the case of model exchange, while the solver is inside the FMU for the case of co-simulation [12].

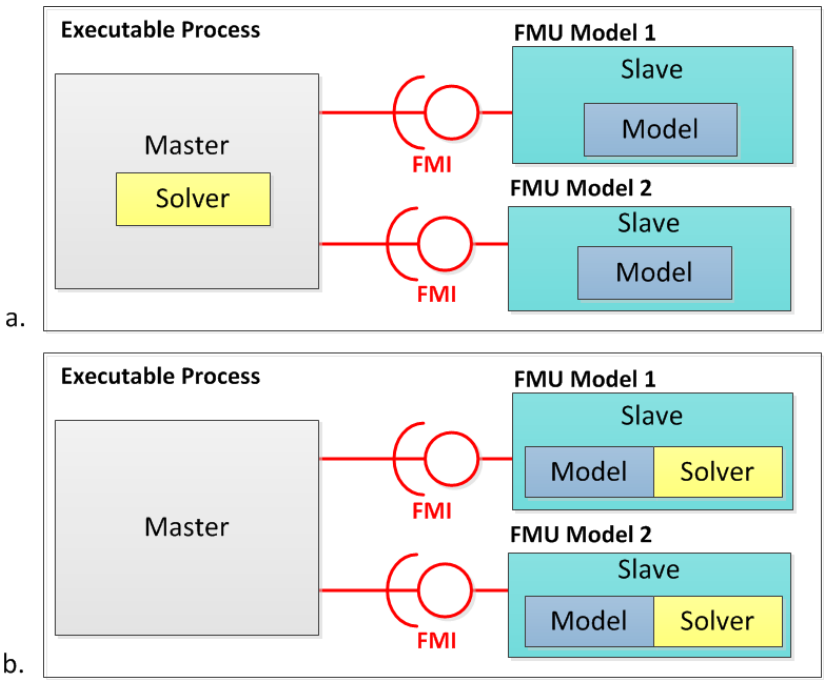

Figure 5: Functional Mockup Unit and Interface for a. Model Exchange b. Co-simulation

An FMU file is a container with the extension *.fmu. It contains several files organized in a pre-defined structure. It includes:

- An XML description file in which the model structure such as the variable names, inputs, outputs and parameters are defined. Moreover, the capabilities of the FMU are specified in the XML file. For instance, specifying that a co-simulation slave can support advanced master algorithms such as using variable communication step sizes, higher order signal extrapolation, etc.

- A C code and/or a binary file format which contains the implementation of the model functionality and behavior. Binary files such as a Windows dll can only be run on the platform they are compiled for. Therefore, most FMUs are restricted to run on a single platform such as Windows 64-bit or Linux 32-bit.

- Further information, such as documentation files, model icon (bitmap file) and/or dynamic link or object libraries, can be included in the FMU zip file utilizing them. The intention is to support platforms that are not known in advance (such as HIL-platforms or micro-controllers).

An FMU provides separation between the description of interface data (XML file) and the functionality implementation (C code or binary). Each FMU contains a XML description file, a C code and the libraries and/or dynamics links. The main idea in simulation engineering of a system is to connect and operate the pool of implemented FMUs together in such a way that emulates the system behavior.
The type of FMU simulation models implemented and used in this work is the co-simulation FMU. This means that the FMU simulation models includes the model and the simulation engine. FMI for Co-Simulation provides an interface standard for the solution of time dependent coupled systems. It consists of subsystems that are continuous in time (model components that are described by instationary differential equations) or time-discrete (model components that are described by difference equations like, for example discrete controllers) [12].

2.3.2 Industrial Domains

Spacecraft Systems Engineering requires simulation of components and models as an emulation of the system operations and functionalities. The FMI standard has been extensively used for simulation purposes in the automotive industry. It was found to be extensively beneficial in terms of model exchange and co-simulation of models.
However since it is not domain specific, it can be utilized in other industrial domains such as the aerospace industry.

Benefits of FMI utilization in the automotive industry

FMI-compliant models give engineers the freedom to use a wide variety of tools, and share amongst peers. It enables high level modeling with specialized tools instead of hand-coding. Prior to the FMI development, model in the loop (MIL) was bridged to hardware in the loop (HIL) by many hours of manual work to integrate the models to be simulated together. This process required tedious work and it was prone to errors. However, with the introduction of the FMI standard, the MIL to HIL development can be accomplished easily by exporting the models as an FMU. Several modeling and simulation software vendors have been adapting to the FMI standard. This rapid adaptation shows that there is a high demand for using FMI in model exchange [13].

In [13], Drenth et al. have utilized the FMI standard to use consistent models throughout the integration workflow from desktop to test bed in the engine controller development. This was accomplished by exporting the models as FMUs. Furthermore, the FMI approach offers a flexible and standard simulator that integrates the development with the production. It also permits exchange of the FMUs and co-simulation between these FMUs via a master simulator. As a result, different types of operating systems and hardware simulators can operate independently and still be able to control synchronization, monitoring and parameter tuning

via an FMU server or a simulation tool [14].

FMI standard has been widely exploited by various industrial partners. It is used as an efficient and fast option to exchange virtual prototypes between different teams. This provides a cross-domain co-simulation environment that integrates suppliers and development partners. It has been used as a commercial software that allows model, software and hardware in the loop. FMI has been developed independently of the industry domain, which makes it also beneficial for domains other than the automotive one, such as aerospace, industrial machinery and construction equipments [14].

2.4 Numerical Optimization

Engineers and computer scientists investigate the behavior of different real-world systems via numerical simulations [18]. Performing this investigation in reality is eminently challenging or even impossible. In general terms, the main aim of most developed computer programs is to simulate the dependence of a single or multiple objectives on a potentially large set of parameters.

It is difficult to directly find the best solution. However, it is comparatively much more achievable to establish a loss or cost function that computes how fit the solution is. This is performed by minimizing the function. In other words, computing its first-derivative and equating it to zero.

The corresponding target values can be obtained by a single run of the simulation program as $y = F(x)$ for a given set of input parameters. It can be extremely useful to simulate the studied real-world system. However, it leaves several questions to be investigated.

One question is related to how sensitive the objective according to changes in the input parameters. Sensitivity analysis is one of the fields that requires computation of the *Jacobian* matrix of F

$$\nabla F = \nabla F(x) \equiv \left(\frac{\delta y_j}{\delta x_i}\right)_{i=0,...,n-1}^{j=0,...,m-1} \tag{1}$$

where $y = F(x)$ represents the numerical simulation. Rows of the matrix includes the outputs sensitivity y_j, $j = 0, ..., m-1$ according to the input paameters x_i, $i = 0, ..., n-1$

By solving the linear differential adjoint equation, gradient values with respect to a particular quantity of interest can be efficiently computed. Analysis of sensitivity serves two main purposes. The sensitivities, on the one hand, are model diagnostics that are useful in understanding how it will change in accordance with parameter changes. But another use is simply that these derivatives are useful in many cases. Analysis of sensitivity provides a way to calculate the solution gradient that can be used in the estimating parameters and in other optimization tasks.

2.5 Optimization in the context of Spacecraft Simulations

There is a demand for high performance in aerospace engineering, modeling and design optimization has been used extensively. The aim of this section is to give an introduction on some used optimization techniques and algorithms.

The process of finding the best design parameters that meet project requirements is called design optimization. Experiment designs, statistics, and optimization techniques are usually used by engineers to evaluate trade-offs in order to determine the best design for their project. This Section illustrates the concepts of cost function, parameter sweeps technique and gradient descent algorithm which are used by engineers for design optimization.

The design of aircraft is a complex task that involves many disciplines. Aerospace architects make high-level decisions and provide concept design specifications, while engineers share domain-specific knowledge and prepare single-disciplinary models that are combined to analyze performance in the multidisciplinary, optimization-driven process.

2.5.1 Cost Function

There are multiple ways to learn the parameters that best suits the optimization goal of a project. The approach used illustrates statistical learning which is by minimizing the cost function [19].

There are various ways to compute the cost function. A cost function maps one or more variables event or values to a real number. In this case, the event we find the cost of is the difference between estimated values, or the difference between the hypothesis and the actual values. In other words, a cost function measures how well the current parameters estimate the relationship between X and Y. This is typically expressed as a difference or distance between the predicted value and the actual value. The Mean-Squared Error (MSE) cost function F(x) is the most commonly used representation and is expressed as follows:

$$F(x) = \frac{1}{2m} \sum_{i=1}^{m} (h(x^i) - y^i)^2 \tag{2}$$

where m is the number of samples. The actual calculation is simply the hypothesis value for h(x), minus the actual value of y. The summation of this calculation for all variables would eventually yield a scalar value [19].

2.5.2 Parameter Sweep Technique

Parameters Sweeps is the process of sweeping over a range of values for specific parameters. In product development, several variations of a model often need to be solved in order to find the optimal properties of its design. A parametric sweep can be performed instead of changing these parameters values manually and re-solving each time. A parametric sweep allows changing in a specified range of parameter values. It is useful to find the optimum value of a parameter and to study the sensitivity of a design performance to certain parameters or to run a series of simulations with a set of different parameters.

In the process of sweeping one or more parameters, the values of these parameters are updated after each simulation iteration. Then, comparison and analysis of the output data from each iteration is performed. Hence, sweeping parameters is used to adjust control parameters, estimate unknown model parameters, and test a control algorithm's robustness by taking into account real-world system uncertainty [26].

2.5.3 Gradient Descent Search Algorithm

Gradient Descent (GD) Search Algorithm is another approach to reach the goal of minimizing a cost function. Gradient descent is an optimization algorithm that tries to find the local or

global minimum of a function. Local minimum is the least point or value of a certain function and global minimum is the least value in the entire function plot [20].

In order to minimize a function and find the lowest error, the variables of the program models needs to be tweaked. This is done by calculating the slope of a function. In other words, computing the derivative of the cost function (Eq. 2) with respect to a variable and equating it to zero. Equation 3 represents the gradient computation.

$$\frac{\partial F(x)}{\partial x} = \frac{1}{m}\sum_{i=1}^{m} h(x^i) - y^i \tag{3}$$

More precisely, the gradient descent algorithm is basically a first-order iterative optimization problem that aims to find the minimum of a function. Steps proportional to the negative of the function's gradient are taken in order to reach the local minimum of a function. On the other hand, taking steps proportional to the positive of the gradient would lead to the local maximum [20].

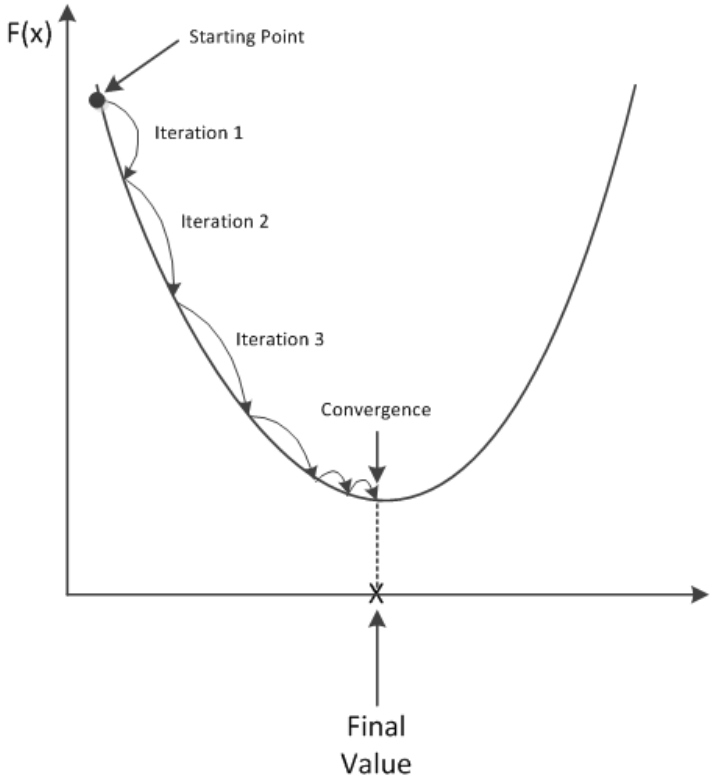

Figure 6: Gradient Descent Algorithm

The main goal of gradient descent algorithm is to learn the gradient or direction that would yield the minimum errors or costs (i.e. differences between actual y and predicted y should be minimized). As the program is iterating, it gradually converges to a minimum where further adjustments to the parameters generates little or zero loss, also called convergence as presented in Fig. 6.

Equation 4 is computed in a loop until the x_n variable converges. In other words, the iterations are performed by executing the entire simulation program for one iteration of GD, calculating the gradients, then updating the weights or variables. This is done for a number of iterations

of GD, as seen in Fig. 6, until the variables converges or the minimum value of the function is reached. At each iteration n, the design variables are updated using:

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n) \tag{4}$$

where x_{n+1} is the updated value of the previous x_n and the learning rate or the step size is representing via the γ_n. The term $\gamma_n \nabla F(x_n)$ is subtracted from x_n in order to move towards the minimum; opposite to the gradient direction. $\nabla F(x_n)$ is the derivative of the cost function with respect to the variables being optimized.

The main challenge in applying the gradient descent algorithm is finding the suitable learning rate γ_n to be used. If this step size, γ_n, is too big, the minimum is going to be overshot, that is, finding the minimum would not be possible since it is skipped by this large learning rate. On the contrary, if γ_n is too small, too many iterations would be required in order to reach the minimum [18].

3 System Design Methodologies and Implementation

The goals and expected outcomes of this master thesis are thoroughly introduced and tackled in this chapter. Space systems development is a complex task that would require the application of model-based system engineering in the simulation development process.

The general architecture and use of these system simulators is well discussed and understood but the portability and exchange of models is still a hurdle. The usage of FMI standard framework in simulators development solves the problem of integration and simulation of models by allowing model exchange and co-simulation capabilities, which is more beneficial and more easy to use compared to the SMP2 standard that is currently used as a space related standard for model portability.

Based on the fact that FMI framework has a lot of features and capabilities that could be used in model exchange, co-simulation and in adopting optimization approaches. It is well accepted in a lot of industries and tool vendors. Thus, it would be quite interesting to investigate the possibility of implementing space simulators and optimizers using the FMI technology. Thus, the main outcome of this thesis is to tackle the following research questions:

- Investigating the usage and extension of the FMI standard to incorporate optimization on basis of Adjoint methods.
- Applying the FMI standard to build a spacecraft system simulator rather than sticking with already existing space-related standards.

The chapter is organized as follows. The first section presents the overall system architecture. Consequently, the various parts of the described architecture will be presented. The second section contains the investigation of incorporating design optimization techniques and algorithms into the FMI-based simulator.

3.1 FMI-based Functional Engineering Simulators (FES)

The functional engineering simulators contain the system models in a low-fidelity representation. The goal is to build such a simulator using the FMI standard framework.

The development of the simulation flow has been implemented using C and C++ programming languages. This implementation has been carried out in Microsoft Visual Studio Professional 2017 Version 15.8.7 IDE

3.1.1 System Architecture

The overall system architecture consists of four major blocks which are the MBSE, the JSON Configuration file, the Optimizer and the Simulator which executes the interaction between the FMU models.

First of all, the MBSE block is basically the process of designing the spacecraft. Within this block, the number of components or models are defined and configured together. The MBSE activities can be performed using the Virtual Satellite tool.

The designed spacecraft architecture is then exported as a configuration file (in a .json representation for instance). This configuration file, which contains the design information, is

then used as a guide by the simulator to connect and simulate the pool of developed FMU models or components together. Furthermore, the optimizer reads this JSON Configuration file as well to obtain the optimization information included in the configuration file. This thesis work focuses on the:

- Design of the JSON Configuration File.
- Implementation of the Optimizer that calls the simulator to use it in applying different optimization techniques such as the parameter sweep and gradient descent search algorithms.
- Development of the FMU models in a low fidelity representation form that would be used in a use case proposed for the evaluation .

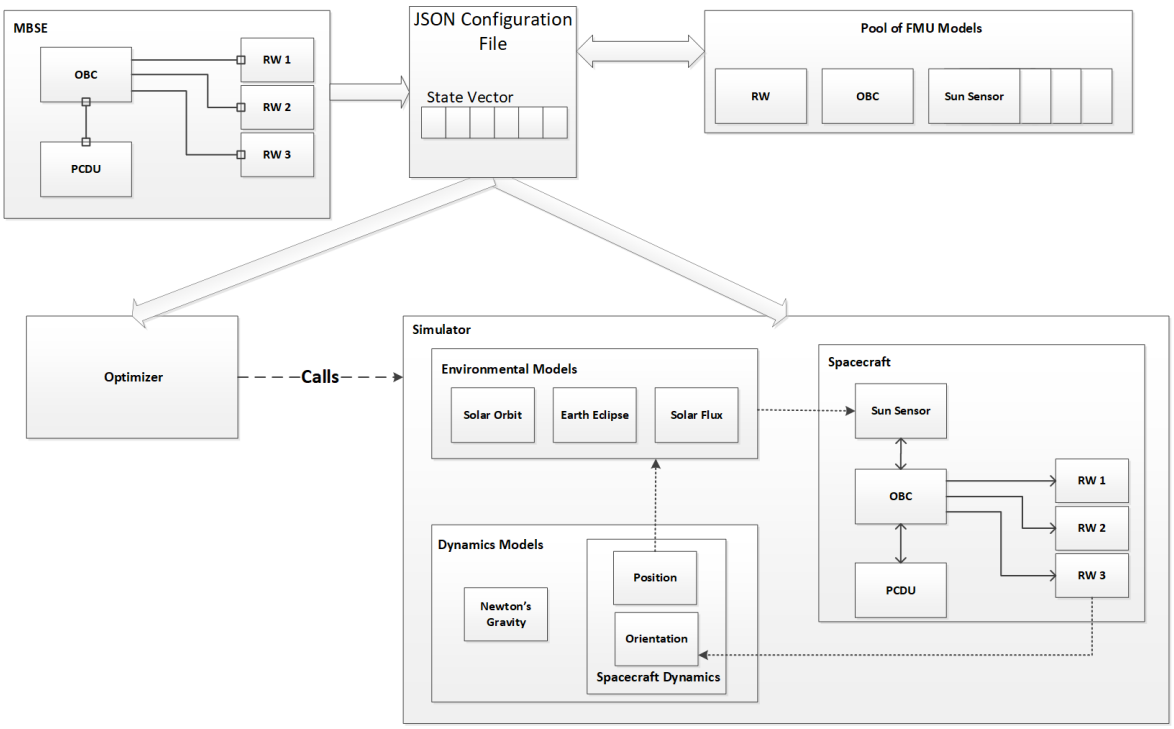

Figure 7: Overall System Architecture

To further illustrate how the four parts of the system architecture work together. The design of the FMI-based simulator is presented first. Basically, the simulator uses the JSON configuration file as a guide. It reads the components block, initializes and executes the FMU models according to the configuration information specified in the JSON file. Consequently, it reads the interfaces block from the JSON file. It then initializes and executes these FMU interfaces models. It saves the outputs from all the FMU models and interfaces into the state vector. Finally, it iterates again and saves the output values of the next simulation step. The number of iterations performed by the FMI simulator is based on the simulation duration specified in the configuration file. Hence, with each iteration, it builds up the state vector of the performed simulation (Refer to Fig. 8).

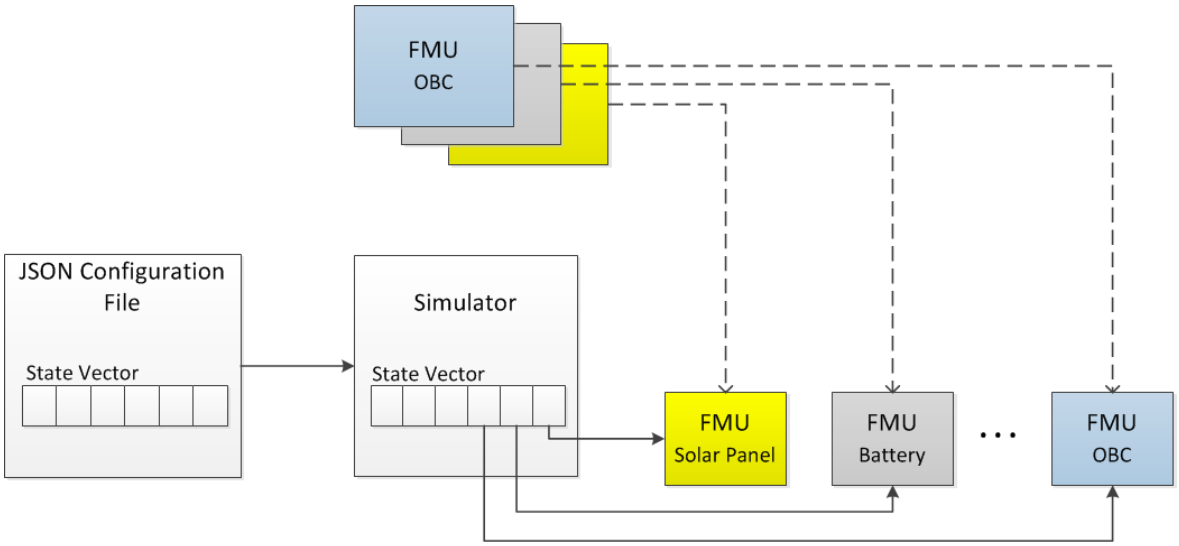

Figure 8: Design of the FMI-based Simulator

The above-presented figure and description illustrates the general steps and tasks performed by the implemented FMI simulator. The code used to perform these tasks can be found in Appendix B.

Fig. 9 represents the activity diagram that shows a step-by-step overview illustrating how the simulator works. The first column shows the task followed by its effect on the file system and on the simulator. The first step is to load the JSON configuration file from the file system into the simulator. The next step is to load the FMU components as specified in the configuration file. Furthermore, the interfaces are loaded and the state vector is set up. Finally, the components and interfaces are connected to each other through the State Vector.

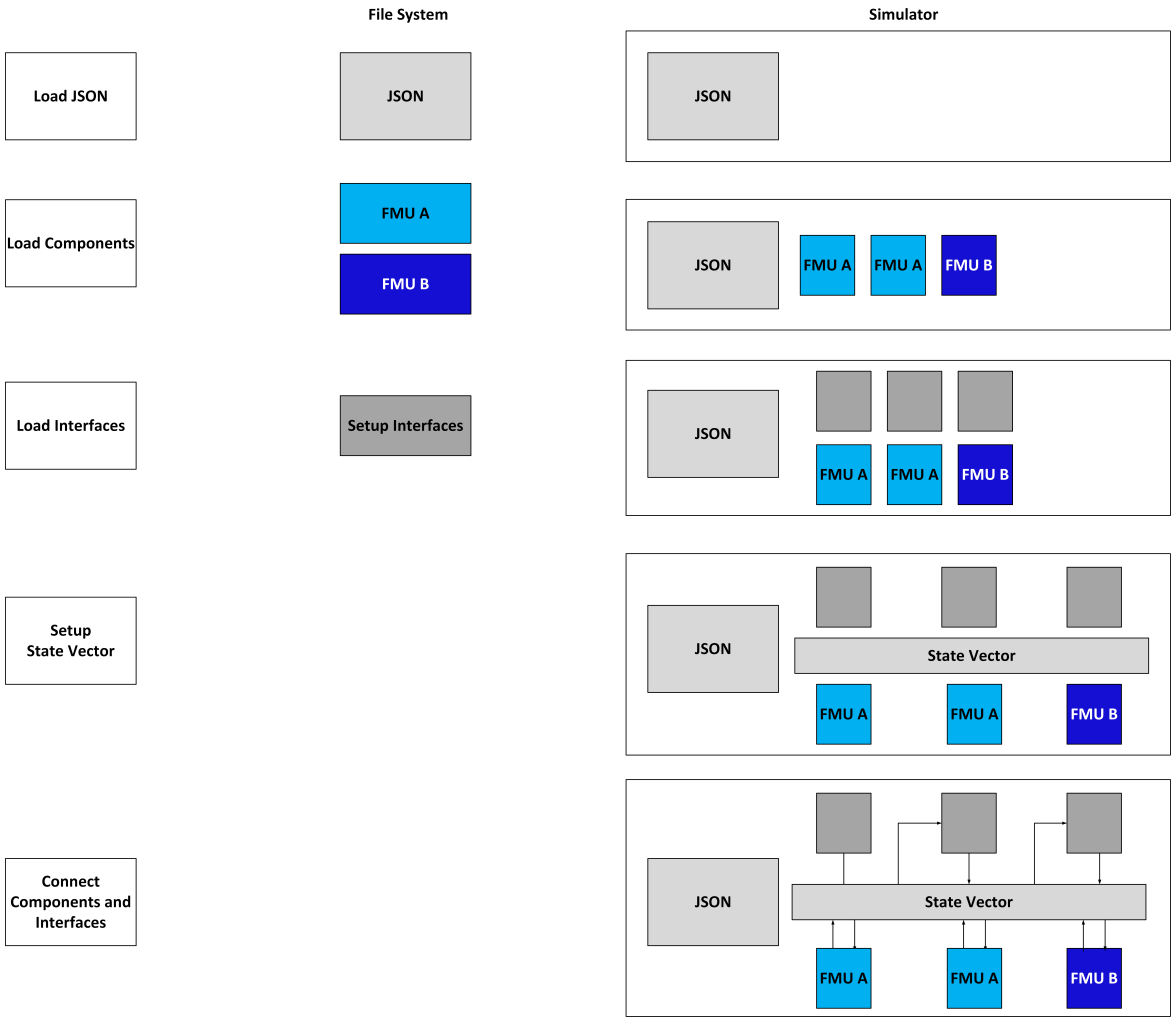

Figure 9: Activity Diagram of the Simulator

3.1.2 Simulator Configuration (JSON) file

A lot of projects use JSON for representing configuration files. Its main purpose is to give guidance to the simulator on how many components are used and how does each component interact with the remaining system models and components.

It includes necessary information needed for the simulator execution such as the simulation duration, initial state vector, as well as the design of each model and interface.

The simulation duration is specified using three variables which are the *simulation_start*, *simulation_end* and the *simulation_stepsize* defining the start time, end time and the step-size respectively. International System of Units (SI Units) are used throughout the program so these time variables are measured in seconds. The below snippet shows the simulation duration for a day.

```
{
        "simulation_start": 0,
        "simulation_end": 86400,
        "simulation_stepsize": 1,
}
```

In addition to that, the JSON configuration file includes the initial state vector. This vector initially conveys values of all model variables before the simulator is executed. For example, the Spacecraft Dynamics model, explained in Section A.7, would require variables that store the position and velocity for X, Y and Z axes respectively. This would look as following in the configuration file.

```
"simulationstatevector": {
        "Dynamics.position_x": 0,
        "Dynamics.position_y": 0,
        "Dynamics.position_z": 0,
        "Dynamics.velocity_x": 0,
        "Dynamics.velocity_y": 0,
        "Dynamics.velocity_z": 0,
}
```

Furthermore, the JSON file also includes the connection between all models and interfaces. The design of each model composes of an instance name, paths to the FMU model and the unzipped version of the model, as well as parameters, inputs and outputs variables. For instance, the design of the Spacecraft Position Dynamics model shown in Fig. A.6 is defined in the configuration file as shown below. Table 1 specifies the parameters used for the Spacecraft Dynamics and the Newton's Universal gravitational models respectively.

Table 1: Parameters of Dynamics and Newtons Gravitational models

Model	Parameters	Value	Unit
Spacecraft Dynamics	Initial Position	$[6.776 \times 10^6 \quad 0 \quad 0]$	m
	Initial Velocity	$[0 \quad 7660 \quad 0]$	m/s
Newton Universal Gravitation	Gravitational Constant	6.674×10^{-11}	$N(m/kg)^2$

```

"components": [
  {
        "parameters": {
                "position_x": 0,
                "position_y": 0,
                "position_z": 6371000,
                "velocity_x": 0,
                "velocity_y": 3000,
                "velocity_z": 7500,
        },

        "inputs": {
                "NG.force_x": "force_x",
                "NG.force_y": "force_y",
                "NG.force_z": "force_z",
                "NG.mass": "mass_spacecraft"
        },
        "outputs": {
                "position_x": "Dynamics.position_x",
                "position_y": "Dynamics.position_y",
                "position_z": "Dynamics.position_z",
                "velocity_x": "Dynamics.velocity_x",
                "velocity_y": "Dynamics.velocity_y",
                "velocity_z": "Dynamics.velocity_z"
```

```
        },
        "path": "..\\FMUs\\Dynamics.fmu",
        "unzipPath": "fmu1\\",
        "instanceName": "Dynamics"

  },
  {
          "parameters": {
                "gravitational_constant": 6.674E-11
          },
          "inputs": {
                  "Dynamics.position_x": "position1x",
                  "Dynamics.position_y": "position1y",
                  "Dynamics.position_z": "position1z",
                  "NG.earth_position_x": "position2x",
                  "NG.earth_position_y": "position2y",
                  "NG.earth_position_z": "position2z",
                  "NG.spacecraft_mass": "mass1",
                  "NG.earth_mass": "mass2"
          },
          "outputs": {
                  "force_x": "NG.force_x",
                  "force_y": "NG.force_y",
                  "force_z": "NG.force_z"
          },
          "path": "..\\FMUs\\Ng.fmu",
          "unzipPath": "fmu2\\",
          "instanceName": "NG"
  }
]
```

Finally, the configuration file also contains an interface block which connects the inputs and outputs of the models. Each interface includes the same attributes as the model except for the parameters. For example, the interface below represents the connection between OBC, Star Tracker, Sun Sensor and the Reaction wheels in which the OBC determines the required torque needed by the reaction wheel to orient the spacecraft based on the information gained from the Start Tracker and the Sun Sensor models.

```
"interfaces": [
  {
        "inputs": {
                "SS.activated_sun_sensor": "activated_sun_sensor",
                "STR.output_orientation_x":"SpacecraftOrientation_alpha",
                "STR.output_orientation_y": "SpacecraftOrientation_beta",
                "STR.output_orientation_z": "SpacecraftOrientation_gamma"
        },
        "outputs": {
                "RW1_required_Torque_x": "OBC1.RW1_required_Torque",
                "RW2_required_Torque_y": "OBC1.RW2_required_Torque",
                "RW3_required_Torque_z": "OBC1.RW3_required_Torque"
        },
        "path": "..\\FMUs\\Obc.fmu",
        "unzipPath": "fmu17\\",
        "instanceName": "OBC1"
  }
]
```

3.1.3 Generalized Simulation Flow

A spacecraft is affected by many forces and energies from other bodies in space. Thus, in the same manner, the simulation models introduced in Section 2.2.3 would have an influence on each other. For the purposes of generalization and evaluation of the FMI-based simulator, low fidelity FMU models have been developed. Fig. 10 shows the overall simulation flow. It clarifies how the models are connected to each other. The models are partitioned into

environmental, dynamics and spacecraft simulation models. The purpose and composition of each partition are explained subsequently.

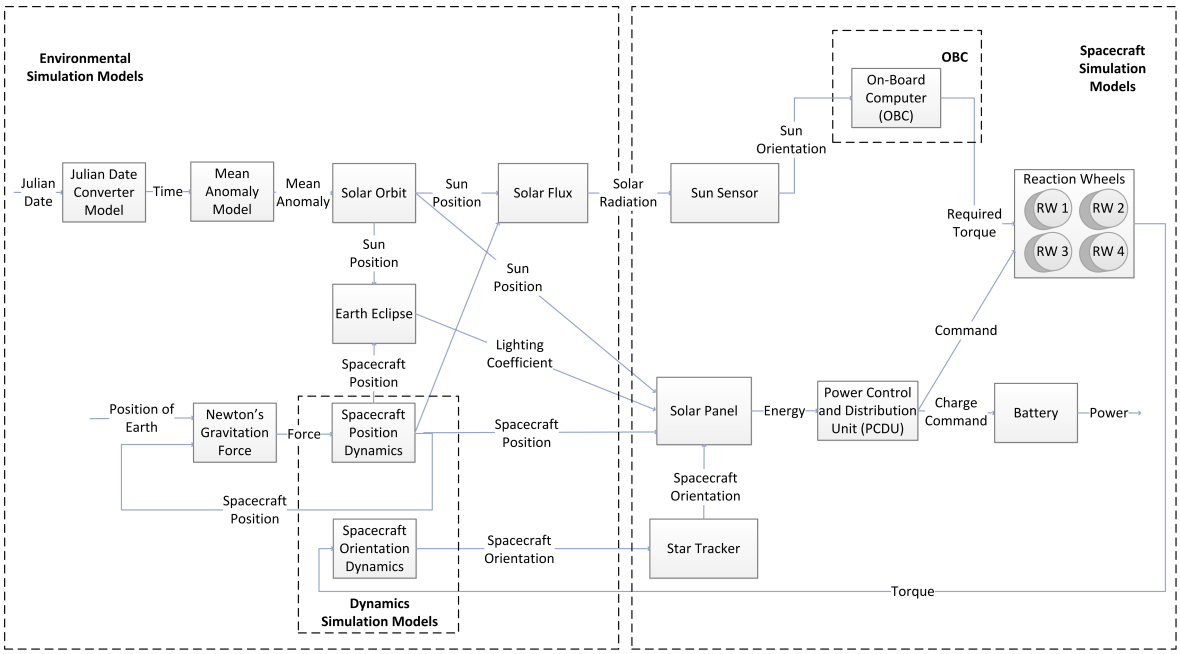

Figure 10: Simulation Flow

Environmental Simulation Models

The environmental simulation models represent an emulation of the environmental factors influencing the spacecraft throughout its mission. An overview of the overall connections of the environmental simulation models will be described, followed by more detailed schemas used in the implementation of each model.

First of all, the Julian Date which is a continuous count of days since the beginning of the Julian Period. It is used primarily by astronomers, and in software programs for easily calculating elapsed days between two events. The Julian Date is transformed to Time. In order to convert the Julian Date to Time, Equation 10 has been used.

Time is then used as an input for the Mean Anomaly model which is then fed to the Solar Orbit model to estimate the position of the Sun with respect to Earth. The output of the Sun Orbit model is further used as an input to the Solar Flux Model which computes the amount of Solar radiation reaching the spacecraft depending on the position of the Sun.

Moreover, the Earth Eclipse model takes as an input the position of the Sun and the position of the Satellite with respect to Earth in order to compute the lighting coefficient which represents the fraction amount of sunlight that reaches the spacecraft throughout its orbit around Earth.

The first environmental model in the pool of FMU models is the Mean Anomaly model. It has been implemented using the schema presented in Figure 11. The equations used to implement each block in the schema has been presented in Section A.2.

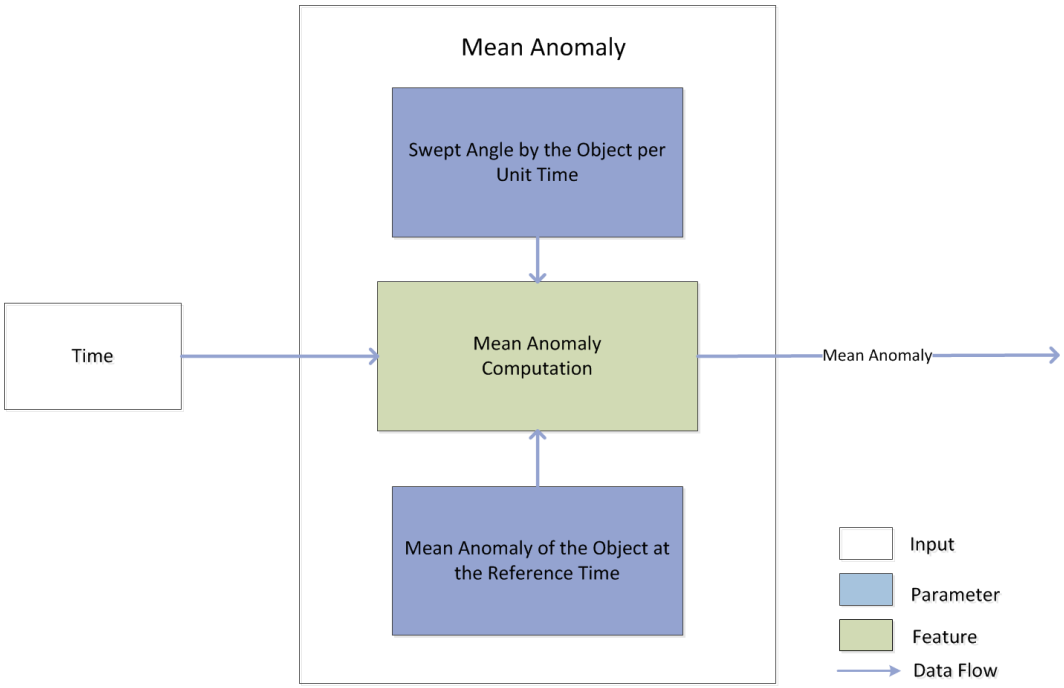

Figure 11: Schema of the Mean Anomaly Model

Based on the aforementioned description of the orbit, the Solar Orbit model schema (see Fig. 12) which calculates the solar vector in the EME2000 coordinate system has been implemented.

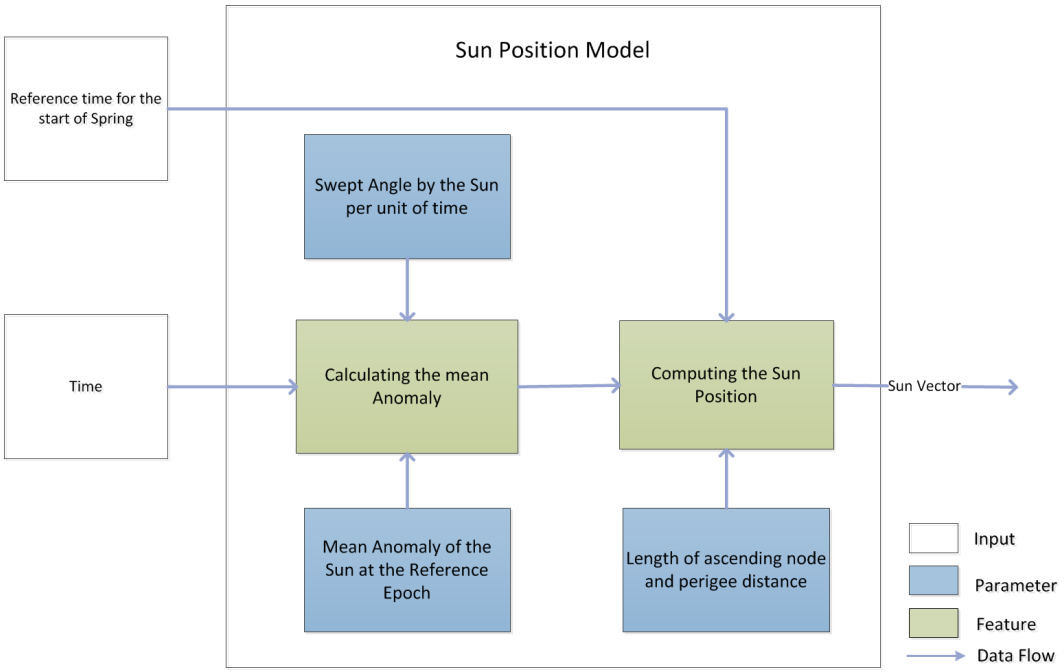

Figure 12: Schema of the Solar Orbit Model

The sun can be roughly described as a radiator, which can be described by the Stefan-Boltzmann law ($\sigma.T^4$). This law is now provided with a factor which is composed of the squared quotient of the radius of the sun R_{Sun} and the distance to the sun. This factor takes

into account the dependence of solar radiation on the distance to the sun. In order to take into account the information of the direction of the radiation, the product is still multiplied by the normalized sun vector. Whether the resulting vector points in the direction of the sun or in the direction of the origin of the coordinate system depends on the orientation of the incoming vector of the sun $\vec{r}_{sun}$. Figure 13 shows the schema used in implementing the Solar Flux model.

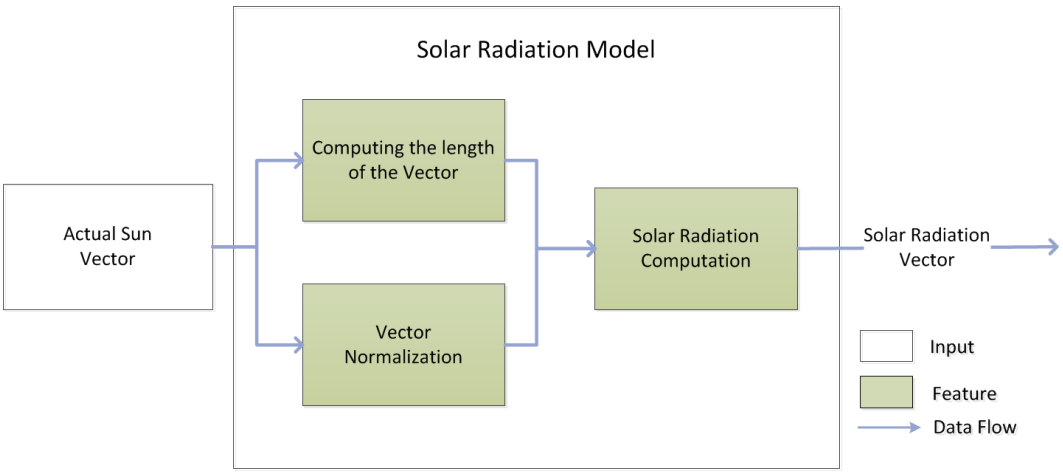

Figure 13: Schema of the Solar Flux Model

Finally, the Earth Eclipse Model is the last environemntal simulation model presented in the simulation flow. Figure 14 presents the schema of implemented Earth Eclipse Model.

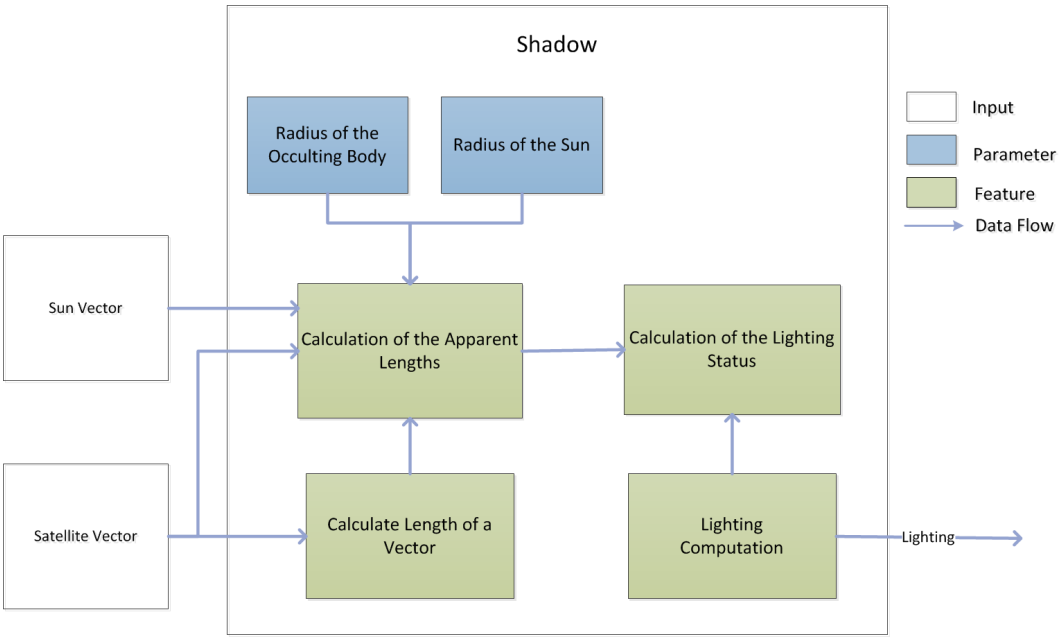

Figure 14: Schema of the Earth Eclipse Model

Table 2 shows the parameters used in each of the environmental simulation models.

Table 2: Parameters of Environmental Simulation Models

Model	Parameters	Value	Unit
Mean Anomaly	Sun Mean Anomaly	6.24	rad
	Sweeping Angle	628.302	rad
Sun Orbit	Length of ascending node + perigee distance	4.938	rad
	Ecliptic Inclination	0.409	rad
Sun Flux	Stefan-Boltzmann Constant	5.67×10^{-8}	$W/(m^2K^4)$
	Radius of Sun	6.96×10^8	m
	Temperature of Sun	5780	K
Earth Eclipse	Radius of Sun	6.96×10^8	m
	Radius of Earth	6.3781×10^6	m

Dynamics Simulation Models

Dynamics simulation models emulate the dynamics of a spacecraft. Basically, the spacecraft starts from an initial position with an initial velocity. Furthermore, the spacecraft is affected by Newton's universal gravitational force which would lead the spacecraft to rotate around Earth.

For example, if a mission is to launch a satellite that would rotate around Earth. The model would be given the mass and the velocity of the satellite as parameters. Moreover, it would be connected to the Newton's Gravity model described in Subsection A.6 as the spacecraft would be influenced by Earth's gravity. It would finally output the position of the Satellite. (Please refer to Figure 15)

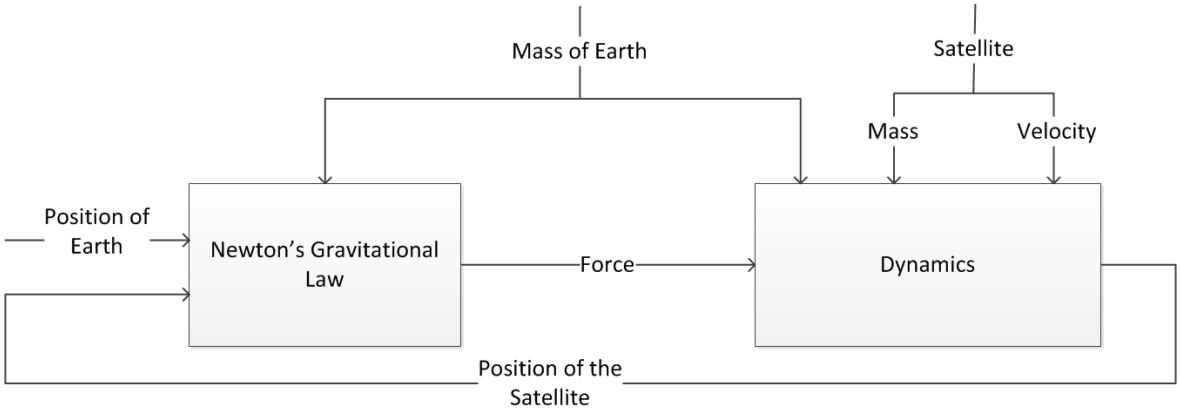

Figure 15: Relation between Newton's Gravity and Dynamics Models

The parameters used for the Dynamics and the Newton's Gravitational models have been presented in table 1.

In addition to that, the Dynamics of a spacecraft also includes its Orientation. The model for this aspect of spacecraft behavior is seen in the block diagram in Figure 16.

When an object has angular acceleration over time, it gains angular velocity, $\vec{\Omega}$. Hence, in order to determine the position of a spacecraft, represented by an angle θ, which describes the amount the spacecraft rotated from its previous position. By applying torque to a spacecraft with specific inertia, we create angular acceleration, leading to angular velocity. Hence, a change in its angular position.

Basically, the angular acceleration $\vec{\alpha}$ is computed via Eq. 23. The application of integration over time would yield the angular velocity $\vec{\Omega}$. Finally, applying a second integration over time would yield the angular position or attitude θ which represents the orientation of the spacecraft.

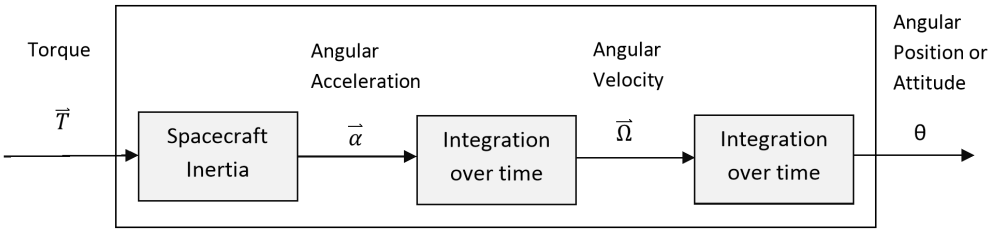

Figure 16: Block Diagram of the Orientation Dynamics Model

Spacecraft Models

Spacecraft models simulate the behaviors and functionalities of the components that would be built on the spacecraft. First of all, the sun sensor determines the sun orientation based on the solar radiation information from the solar flux model. The OBC commands the reaction wheels to exert a certain amout of torque based on the sun orientation or based on the spacecraft mission. The exerted torque from the reaction wheels affects the spacecraft dynamics orientation.

Moreover, the sun position from the sun orbit model, the lighting coefficient from the earth eclipse model, the satellite position from the spacecraft dynamics model and the satellite orientation from the star tracker model are all fed to the solar panel model which computes the amount of energy it gets from the sun to charge the battery and/or other satellite components.

The PCDU controls the battery charging depending on the remaining energy from the solar panel taking into consideration the payloads consumptions.

Table 3 represents the parameters used in the spacecraft simulation models. The values used for these parameters have been induced based on the TET-1 (Technologie Erprobungs Träger-1) microsatellite which has been demonstrated by DLR.

Table 3: Parameters of the Spacecraft Simulation Models

Model	Parameters	Value	Unit
Solar Panel	Orientation	[1 0 0]	
	Efficiency	0.5	
	Area	1	m^2
Reaction Wheel 1 (RW 90)	Orientation	[1 0 0]	
	Maximum Torque	0.15	$N.m$
	Maximum Power	5	W
Reaction Wheel 2 (RW 90)	Orientation	[0 1 0]	
	Maximum Torque	0.15	$N.m$
	Maximum Power	5	W
Reaction Wheel 3 (RW 90)	Orientation	[0 0 1]	
	Maximum Torque	0.15	$N.m$
	Maximum Power	5	W
Power Control and Distribution Unit (PCDU)	Payloads Power Consumption	20	W
Battery	Maximum Power	240	Ah

There are much more components on board of a spacecraft. However, these models were suffiecient in order to simulate the mission and evaluate the use case for a FMI-based simulator.

3.2 Design Optimization within the FMI Infrastructure

The new designs of today have increased in complexity and need to address more stringent requirements for society, the environment, finance and operations. A paradigm shift is therefore underway that challenges the design of complex systems. Advances in computing power, computational analysis, and numerical methods have also transformed the way design is conducted significantly and have affected it.

This section focuses on investigating the possibility of incorporating design algorithms and techniques into the simulator using the FMI standard infrastructure.

3.2.1 Optimization Goal

Before digging deeper in the application of different design optimizations within the FMI-based simulator, the optimization goal is presented first.

A specific scenario has been assumed to be used as the mission goal. It is assummed that the spacecraft would launch from the North pole and would fly in a certain trajectory to reach a target position. The aim is to find out the optimum values that should be used as initial velocity in the Y and Z directions. Figure 17 represents the assummed mission target.

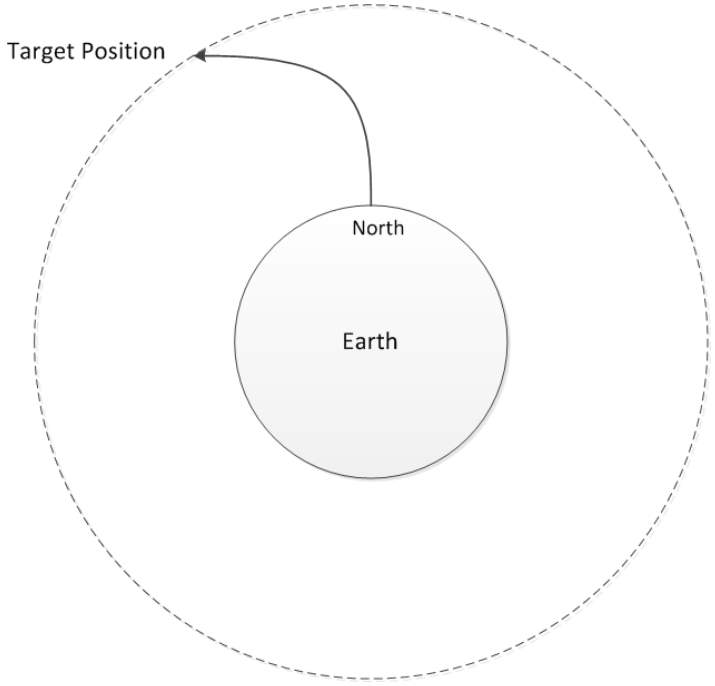

Figure 17: Target Position of the Mission

3.2.2 Parameter Sweep

The first technique is applying parameter sweeps on several variables in order to discover the optimum parameter values and investigate the sensitivity of such variables.

In order to incorporate that to the simulation, an additional block has been added to the JSON configuration file. It includes the name of the optimization technique or algorithm, the parameters to be swept over and the sweeping ranges. The below optimization block specifies that the Parameter Sweep technique will be performed on two variable of the dynamics model. First sweep will be on the initial velocity at the Y-axis direction starting from 2000 m/s and ending at 3000 m/s with a step of 100 m/s. The second sweep is for the initial velocity at the Z-axis direction starting from 6000 m/s and ending at 8000 m/s with a step of 200 m/s.

```
"optimization": [
  {
        "algorithm": "ParameterSweep",
        "parameters": {
                "Opt.velocity_y_start": 2000,
                "Opt.velocity_y_step" : 100,
                "Opt.velocity_y_end"  : 3000,
                "Opt.velocity_z_start": 6000,
                "Opt.velocity_z_step" : 200,
                "Opt.velocity_z_end"  : 8000
        },
        "instanceName": "Opt"
  }
]
```

Besides the adjustments made to the JSON configuration file, an optimizer has to be implemented which interprets this additional "optimization" block.

The optimizer would read the JSON configuration file and would call the simulator in a *for* loop in order to sweep over the range specified for a certain parameter. The presented optimization block includes sweeping ranges for two parameters. Hence, the optimizer would have two nested *for* loops. The following code shows the steps performed by the optimizer to sweep over the two parameters. The optimizer updates the configuration file at each iteration andruns the simulator with updated JSON configuration file.

```
for(double i = velocity_y_start; i <= velocity_y_end; i = i + velocity_y_step)
{
    int vely_currentSweepValue = (int) i;

    for (double j = velocity_z_start; j <= velocity_z_end; j = j + velocity_z_step)
    {
            int velz_currentSweepValue = (int) j;
            updateConfigurationFile(vely_currentSweepValue, velz_currentSweepValue);
            executeSimulator(updatedConfigFile.json);
            saveToFile(positionCost);
    }
}
```

Finally, after running the simulator with the updated values of the parameters. The final position cost computaion attained by simulating such a configuration is saved to a file. This iterations are executed until the final values of the sweeping ranges are reached.

3.2.3 Gradient Descent Search

The second design optimization algorithm adressed in this thesis work is the Gradient Descent Search. In order to achieve the optimization goal described in Section 3.2.1, two appraoches will be proposed. The first one is the classical approach which describes the traditional way to implement the algorithm. The second approach is the advanced one which presents the incorporation of this optimization algorithm into the FMI infrastructure.

Classical Approach

This approach implements the Graient Search algorithm in a traditional way. The following steps are taken consequently.

1. The final target position is specified in the configuration file.
2. Selecting random values for the initial velocities of the spacecraft.
3. The position cost is computed between the target position and the final position reached by the random velocity values selected in Step 2.
4. Steps 2 and 3 are repeated for a slight change in the initial velocity values. In other words, a certain delta is added to the initial velocities in the Y and Z directions.

$$\begin{aligned} Vel_{init} &= [0 \quad vel_y \quad vel_z] \\ Vel_{delta_y} &= [0 \quad vel_y + delta_y \quad vel_z] \\ Vel_{delta_z} &= [0 \quad vel_y \quad vel_z + delta_z] \end{aligned} \tag{5}$$

5. Based on the three computations of the position cost, the slopes are calculated.

$$slope(vel_{init}, vel_{delta(i)}) = \frac{PositionCost(vel_{init}) - PositionCost(vel_{delta(i)})}{vel_{init} - vel_{delta(i)}} \tag{6}$$

6. The final step in this iteration is to evaluate the next velocity values that would be used in the next iteration. This is computed using the Gradient Search equation states in Eq. 4. A relatively small learning rate is chosen to avoid overshooting the minimum position cost.

7. These steps are iterated over in a loop until it converges (i.e two successive values of the velocity are approximately equal).

In the above-mentioned sequence of steps, the simualator is called three times in order to compute the position cost with respect to three different vectors of initial velocity. The first vector is the initial velocity and the second is with an added delta to the velocity in the Y direction. The final third vector is the velocity with an added delta to the velocity in the Z direction. Based on the slopes calculated in Eq. 6, the initial velocity vetor of the next iteration in calculated.

Advanced Approach

The advanced approach is incorporating the implemetation of the Gradient Search algorithm into the FMI Framework. The main idea of this approach is that the derivatives or the sensitivity matrix will be calculated locally within each FMU model and passed on to the next FMU model. The next model would in turn compute its local derivates and multiply it with the previous sensitivity matrix.

The advantage of this approach is that the simulator will only be called one time instead of three times as in the case of the classical approach.

For simplicity, three FMI models will be considered to clarify the possinility of incorporating this approach to the FMI infrastructure. The three models are the Newton's Gravitational Force model, the Spacecraft Dynamics model and the Cost Function model. Basically, the simulator will instantiate and execute these models for the simulation duration and the final position at the end of the simulation run will be used in computing the Position Cost (See Fig. 18).

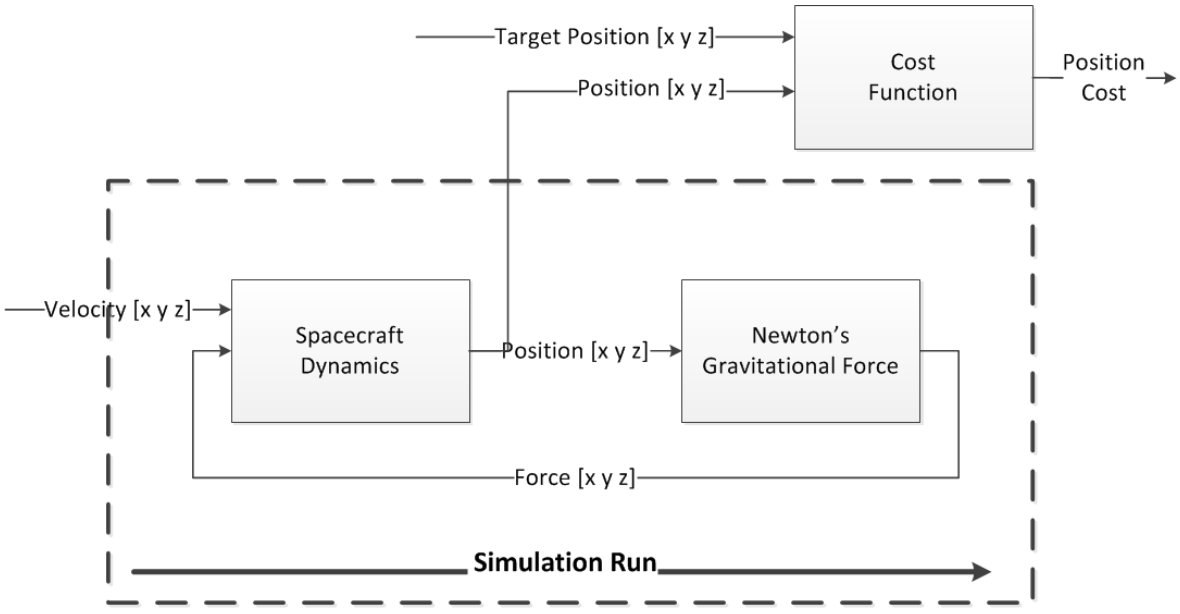

Figure 18: Advanced Approach

For instance, if the simulation duration is for 1 day (i.e 86400 seconds), then the simulator would run for these duration. It would unzip the FMU models, initialize and execute them, resulting in a final Position Cost. This would represent how far the spacecraft is located from the target position.

In order to incorporate the Gradient Search optimization algorithm within the FMI infrastructure, several adjutments needs to be made. First of all, the FMU models has to be extended with the additional variables and parameters required for the computations of the local derivatives and the sensitivity matrix.

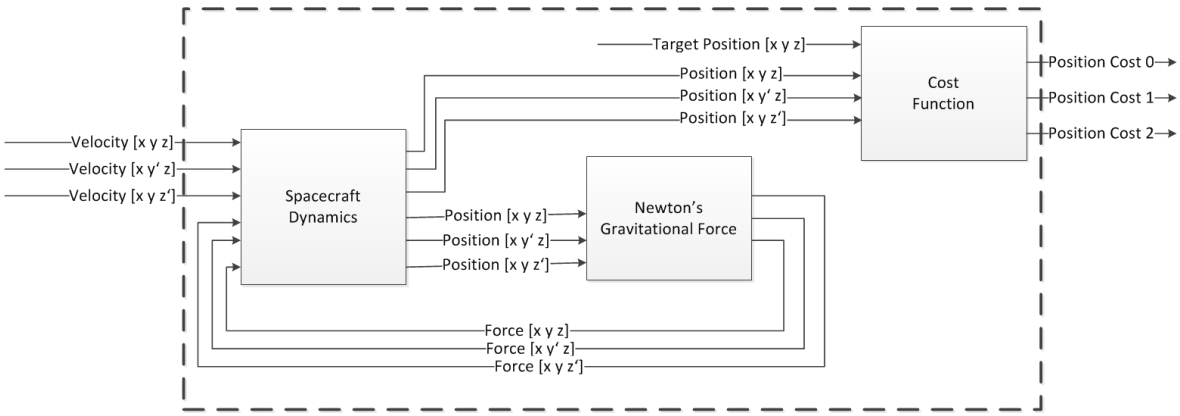

Figure 19: Advanced Approach: Extension of FMUs

Figure 19 illustrates the additional variables that are required to be added to the FMU models. These additional variables would be used for the computation of the three Position Costs in one simulation run. Instead of having one initial velocity as an input to the Dynamics model, there would be three velocity parameters, the initial one and intial with an additional delta in the Y direction and the initial with an additional delta in the Z direction. These three velocities vectors would yield three different position vectors as an output of the Dynamics model. On the other hand, the Newton's Gravitational Force model would take as an input the three position vectors as an input and would yield three Force vectors for each input vector respectively. Consequently, these three Force vectors would be used as an input to the Dynamics model leading to the new Position vectors in the next iteration. This scenario would be iterated over for the duration of the simulation. Finally, the last three Position vectors computed in the last simulation step would be used in the computation of the Position Cost. Hence, the final output would be the three Position Cost values resulting from the three initial velocities vectors.

The next step in the Gradient Search algorithm is to compute the slopes. However, these slopes or derivatives would be calculated locally within each FMU model. Equations 7 and 8 shows how the local derivatives are comuputed in the Dynamics model and the Newton's Gravitational Force model respectively.

$$\frac{\delta Position_i(vel_{init}, vel_{delta(i)})}{\delta Force_i} = \frac{Position_i(vel_{init}) - Position_i(vel_{delta(i)})}{Force_i(vel_{init}) - Force_i(vel_{delta(i)})} \tag{7}$$

$$\frac{\delta Force_i(vel_{init}, vel_{delta(i)})}{\delta Position_i} = \frac{Force_i(vel_{init}) - Force_i(vel_{delta(i)})}{Position_i(vel_{init}) - Position_i(vel_{delta(i)})} \tag{8}$$

where i refers to the X, Y or Z axes. $Position_i(vel_{init})$ and $Position_i(vel_{delta(i)})$ refer to the position values in the i axis resulting from introducing the initial velocity vel_{init} and the initial velocity with and additional delta in the i axis direction $vel_{delta(i)}$ respectively. In the same manner, $Force_i(vel_{init})$ and $Force_i(vel_{delta(i)})$ refer to the Newton's Gravitational force exerted in the i axis, resulting from the vel_{init} and $vel_{delta(i)}$ respectively.

These computations are performed in the $void\ calculateValues(ModelInstance\ *comp)$ function in the C code of the respective FMU model.

Figure 20 presents the locally computed derivatives of each FMU model. In the initial state, the simulations begin with the velocity parameter as an initial feed to the Spacecraft Dynamics model. The locally computed derivatives are change in position with respect to the initial velocity parameter in each of the Y and Z axes. This matrix of partial derivatives would be the input to the next FMU model in the chain. Hence, the matrix would be the input to each of the Newton's Gravitational Force model and the Cost Function model. In the same way, the locally computed derivatives matrix of the Newton's Gravitational force model is the partial derivatives of the force with respect to the position in the Y and Z axes respectively. Furthermore, the partial derivatives matrix returned from the Cost Function model is the partial derivatives of the Position Cost with respect to the position in the Y and Z axes.

Consequently, the upcoming simulation iterations throughout the simulation duration would have a different initial input to the Spacecraft Dynamics model. Since then, the feed to the Spacecraft Dynamics model is the Force vector which was the outcome of the Newton's Gravitational Force from the previous iteration. Hence, the output derivatives matrix from the Spacecraft Dynamics model is the partial derivative of the Position with respect to the Force in each of the Y and Z axes.

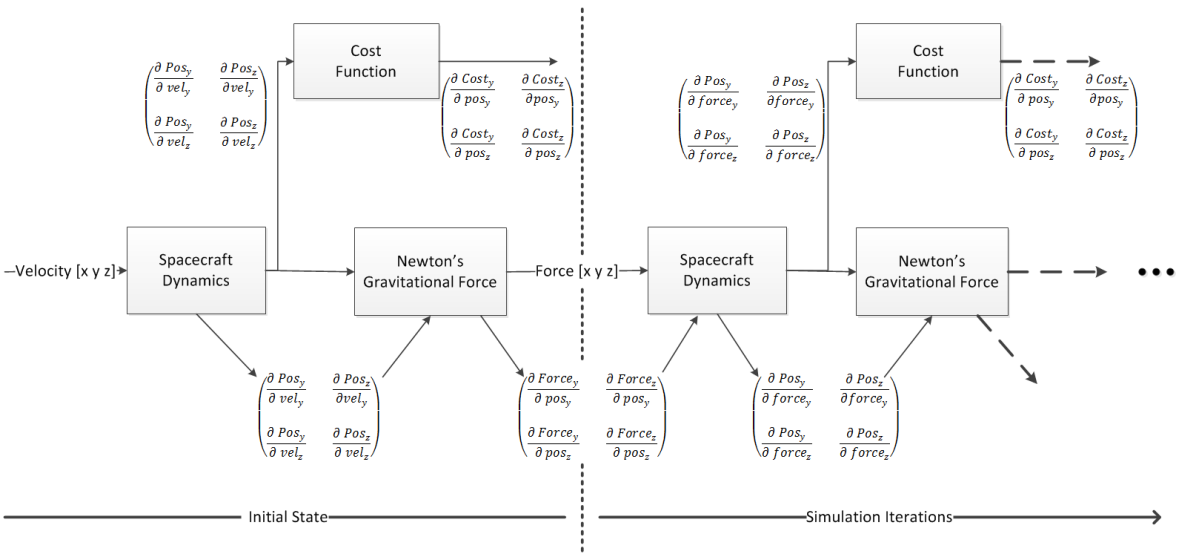

Figure 20: Local derivatives of each FMU model.

These locally computed derivatives indicates the sensitivity matrix of the respective FMU model. Hence, to compute the overall sensitivity matrix of all the models, each local derivative should be multiplied with the local derivative of the previous FMU model in the chain of models. More precisely, the multiplication of the partial derivatives would yield the partial derivative of the Position Cost with respect to the change in the initial velocities fed to the

system. This can be clarified by the following equation.

$$\frac{\delta Cost}{\delta Vel} = \frac{\delta Cost}{\delta Pos} \times \frac{\delta Pos}{\delta Force} \times \frac{\delta Force}{\delta Pos} \times ... \times \frac{\delta Pos}{\delta Vel} \tag{9}$$

where each element corresponds to the sensitivity matrix of an FMU model. The multiplication of the three model's matrices are re-multiplied throughout the simulation iterations and the final multiplicand is the sensitivity matrix of the initial state. Heanse, yielding partial derivative of the Position Cost with respect to the initially fed velocities.

The final step in the Gradient Search algorithm is to compute the new values that would be used as initial velocites in the next simulation cycle. This step is implemeted in the Optimizer C++ code that would decide the new velocity values based on the last computed sensitivity matrix.

Generalization

The advanced approach has been implemented and explained on three FMU models for simplicity and comprehensibility. However, it is more significant to further expand this approach to the entire simulator including all the FMU models drawn in the Simulation Flow Figure (Fig. 10).

Conventionally, several independent teams work simultaneously on the mission development project. Each team is specialized in a specific domain. Thus, each team is responsible of implementing a certain component. Naturally, various software development application might be used by the teams. Hence, for the purpose of integration of these models into one executable system, each team would export their implemented component as an FMU model. The System Engineer who is in charge of executing and simulating the models together, can choose between multiple approaches or alternatives.

Figure 21 presents different approaches that could be applied in order to incorporate the design optimization functionality into an FMI-based simulation system.

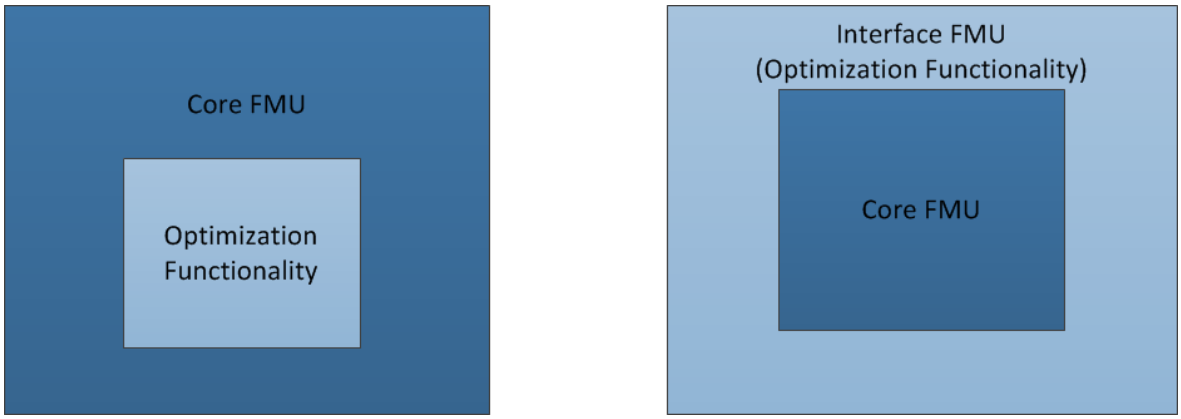

Figure 21: Approaches to incorportate the Design Optimization Functionality

One approach to resolve incorporating the design optimization functionality into the system, is to claim this requirement from the development teams or OEMs by stating it in the Requirements Specification Document. The extra requirement would state the necessity of appending

within the model an additional computation of the local partial derivatives or sensitivity matrix and multiplying it to the another sensitivity matrix fed to the model.

An alternative approach is to implement an interface FMU model which would act as a wrapper around the main FMU model. This interface is responsible of extending the model by incorporating the additional design optimization functionality.

4 Evaluation

This Chapter delves into the evaluation and validation of the implemented FMI-based simulation system as well as the assessment of incorporating the design optimization techniques into the FMI-based simulator. The Chapter is splitted into two sections. The first section presents the use case used to evaluate the FMI-based Simulator, followed by the results yielded from this use case. The second section presents the outcome from applying the Parameter Sweep technique and the Gradient Search algorithm to the simulator.

4.1 Evaluation of the FMI-based Spacecraft Simulator

In order to validate the implemented FMI-based Simulator, a use case will be proposed. The use case serves as a proof that the FMI-based Spacecraft Simulator operates as expected. It is important to note that since scalability is not in the scope of this thesis work, only low-fidelity FMU models have been implemented and used.

4.1.1 Use Case: Orientation and Charging Control

The use case used in evaluating and validating the FMI-based Simulator is based on applying orientation and charging control functionality to the simulator. The main idea of the use case is orienting the spacecraft in a way that would direct the solar panel on board of the spacecraft to point towards the sun direction. The purpose is to have the maximum amount of charging from the solar energy whenever the spacecraft is not in an eclipse region.

Figure 22 displays the set of FMU models used in simulating the use case. First of all, the environemnetal models are connected to mimic the environmental factors influencing the spacecraft. The Earth Eclipse model outputs the amount of lighting coefficient reaching the spacecraft. The Sun Sensor model uses this information as well as the solar radiation exerted from the Solar Flux model to indicate the direction of the Sun. For simplicity, the output of the Sun Sensor model is a scalar value indicating the activated sun sensor. A sensor gets activated whenever it is facing the Sun direction.

Based on the activated Sun Sensor, the OBC determines the amount and direction of torque that needs to be exerted by the Reaction Wheels (RWs). It then commands each RW to exert a certain amount of torque. Accordingly, the RWs would output the required torque. Thus, leading to the adjustments and changes in the orientation of the spacecraft.

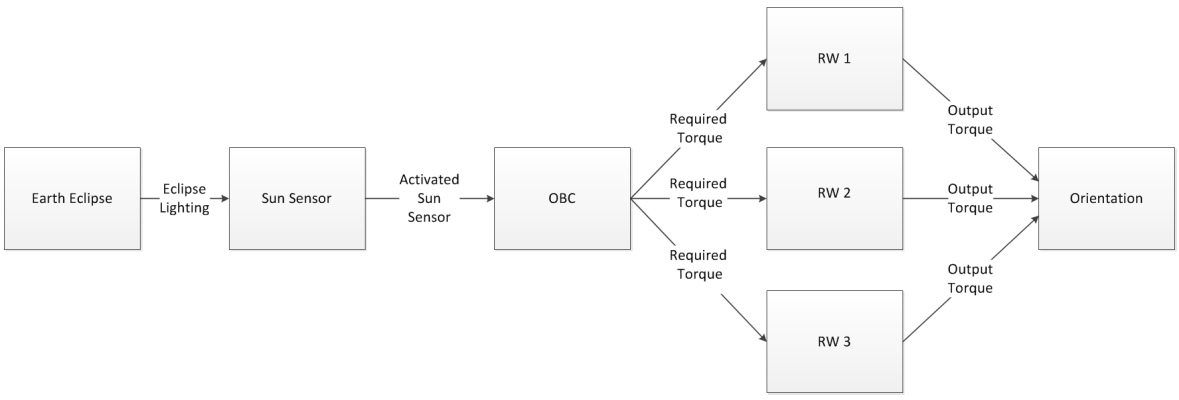

Figure 22: The Use Case FMU Models.

For simplicity, the spacecraft is assumed to be cube shaped on which the solar panel is attached onto one side and 5 sun sensors are attached to the remaining sides.

In order to further clarify the use case, Fig. 23 illustrates the scenario of the implemented use case. The orientation and charging control functionality starts operating once the spacecraft has merged out of the eclipse region into the sunlight. After running the simulation, it was found that the spacecraft emerges from the eclipse region in the orientation magnified in the bottom left corner of the Figure. The solar panel is represented by the thick dark blue border line on the spacecraft. It is oriented towards the +X axis direction. Furthermore, the Sun Sensors with the numerals 2, 4 and 5 are oriented towards the +Y, -X and -Y axes directions respectively.

The required goal is to orient the spacecraft in such a way that the solar panel would be directed towards the sun. In order to realize that, the spacecraft should be rotated around the -Z axis, based on the right-hand rule. Hence, the solar panel would be oriented on the -Y axis direction which is the Sun direction with respect to the spacecraft.

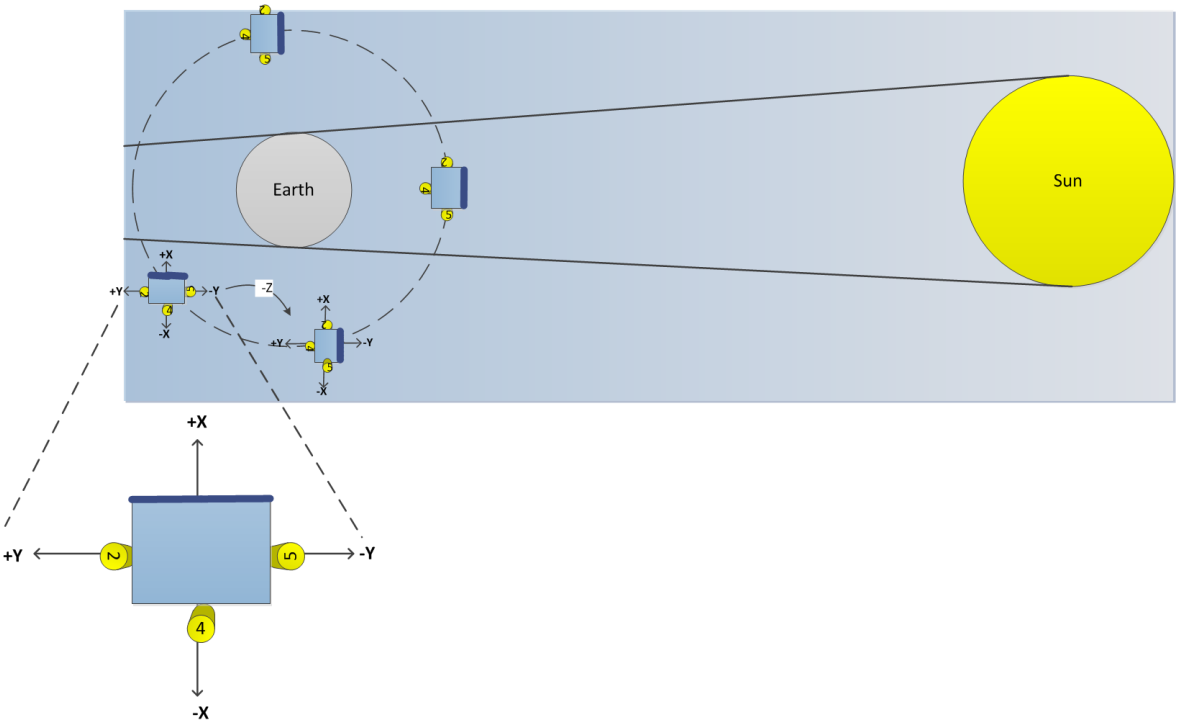

Figure 23: Orientation and Charging Control Use Case

4.1.2 Use Case Results

This upcoming segment is to demonstrate the output of each FMU model used in realizing the described use case. The stated scenario has been simulated for an overall duration of two hours (i.e. 7200 seconds).

The first plot in Figure 24 presents the output yielded from the Earth Eclipse model which exports the lighting coefficient reaching the spacecraft. Lighting coefficient of value 1 indicates full sunlight while value 0 indicates that the spacecraft is in an Eclipse region.

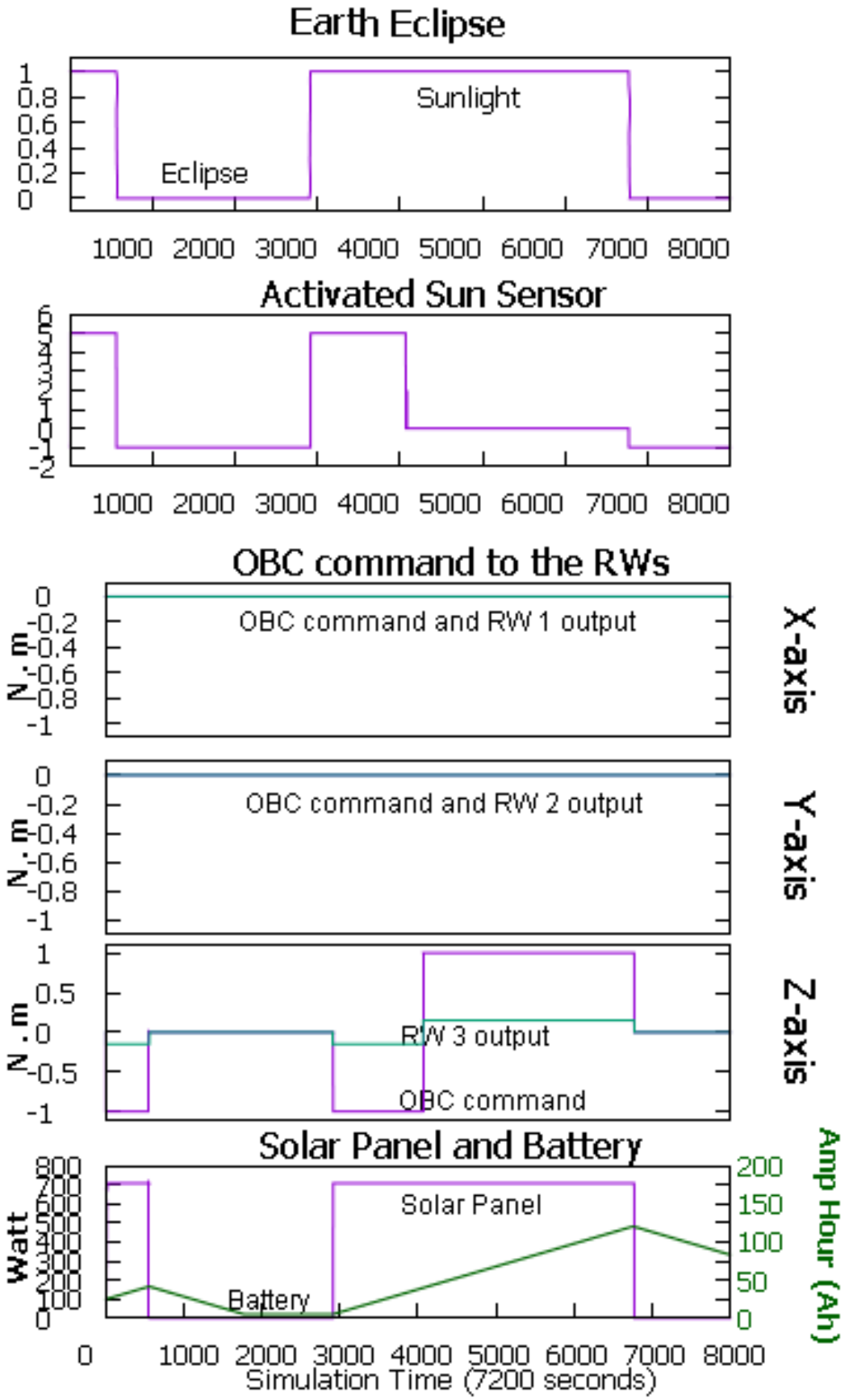

Figure 24: Orientation and Charging Control Use Case Result

The second plot demonstrates the activated Sun Sensor pointing towards the sun direction. Initially when the spaceraft emerges out of the Eclipse region, the Sun Sensor with the number 5 is activated since it is pointing towards the Sun direction (i.e. -Y axis). Hence, the Solar Panel is in the +X axis direction. Thus, the spacecraft should be oriented in such a way that it would rotate with an angle of 90°.

The OBC of the spacecraft would then control this change in orientation. It would command the reaction wheels to exert certain amounts of torques in order to accomplish the rotation around the -Z axis. It would command the first and second reaction wheels (RW1 and RW2) which are located on the X and Y axes of the spacecraft to exert 0 N.m of torque. Hence, RW1 and RW2 would perform the required task.

Furthermore, the OBC would command the third reaction wheel (RW3), oreinted on the Z-axis, to exert -1 N.m of torque. RW3 would then carry out the required amount of torque. However, it would output only 0.15 N.m which is the maximum amount o torque it could exert. The purpose of these OBC commands to the RWs is to change the spacecraft orientation to rotate about the -Z axis until the solar panel would face the Sun direction.

After changing the spacecraft orientation, this would reflect on the second plot presenting the activated sun sensor. The number refering to the activated sun sensor would change from 5 to 0 which means that the solar panel is facing towards the sun direction. Moreover, during Eclipse regions, there would be no activated sun sensors. Hence, the output of the Sun Sensor model would be -1.

Finally, the last plot in Fig. 24 shows the outputs of the Solar Panel and Battery models. It indicates the amount of power measured in Watt that the Solar Panel has charged throughout regions of Sunlight. On the other hand, as expected, it does not charge in regions of Eclipse due to the absence of the Solar energy. The output plot of the solar panel is expected to be bell-shaped. However, it is not because the implemented model is in a low fidelity representation, thus the orientation was not taken into consideration.

Furthermore, the dark green graph color displays the output of the Battery model. It gets charged during Sunlight regions from the energy gained by the solar panel. The amount of charging depends on the remaining energy after the payloads consumpsions has been taken into consoderation. On the other hand, it discharges during Eclipse region since it provides energy to the Spacecraft instead of the Solar Panel.

The change in orientation can be presented by plotting the angular speed from the Orientation model described in Fig. 16. Therefore, Figure 25 displays the dynamics angular speed experienced by the spacecraft from the described use case.

As anticipated, there would be no angular speed in the X and Y axis since there is changes only in the Z axis. As above mentioned, the OBC commands only the third reaction wheel to output certain amount of torque in the negative Z direction. Thus, the third subplot in Fig. 25 displays that the change of angular speed measured in radians per second. It shows that the angular speed in the Z-axis direction is negative as expected.

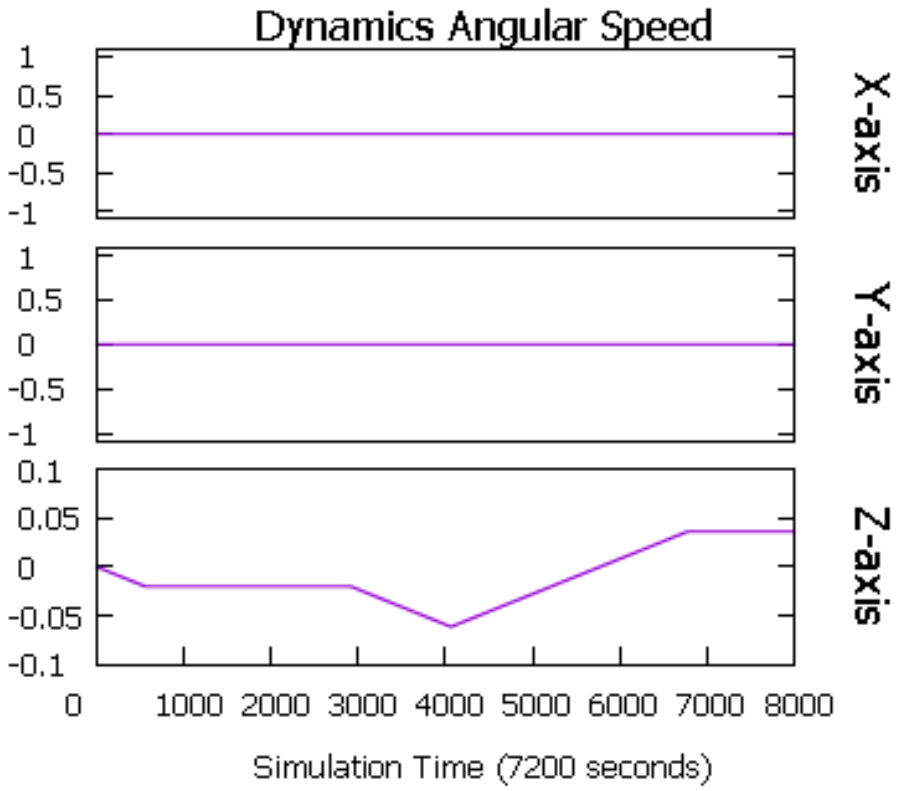

Figure 25: Angular Speed

4.2 Optimization Evaluation of the FMI Spacecraft Simulator

As mentioned earlier, the second section of this chapter presents the evaluation of applying optimization techniques on the FMI Spacecraft Simulator. The first section presents the results of applying the Parameter Sweep technique in optimizing the FMI simulator design.

4.2.1 Parameter Sweep

The parameter sweep technique was applied on two variables. Table 4 presents the parameter, its sweeping range, the step size or delta used and the unit of this variable respectively. The velocity in the Y axis direction was swept over by values ranging from 2000 to 3000 m/s with a delta of 100 and the velocity applied in the Z axis direction was swept over by values ranging from 6000 to 8000 m/s with a delta of 200.

Table 4: Parameters used in the Parameter Sweep Approach

Parameter	Sweep Range	Step Size (Delta)	Unit
Velocity Y	2000 - 3000	100	m/s
Velocity Z	6000 - 8000	200	m/s

Figure 26 displays the output from applying the parameter sweep approach to optimize the velocities applied on the Y and Z axes respectively. Velocity Z, Velocity Y and the Position Cost are displayed on the X, Y and Z axes respectively.

The Z axis of the plot displays the Position Cost yielded from applying each value of velocities Y and Z throughout the sweeping range. In other words, the first position cost is gained from using the initial velocities 2000 m/s in the Y axis and 6000 m/s in the Z axis. The second calculated position cost is from sweeping to the next value of velocity in the Z axis. Hence, using initial velocities 2000 m/s in the Y axis and 6200 m/s in the Z axis and so on till the end of the sweeong range of the Z axis. Then, the value of the initial velocity of the Y axis is incremeneted with the correcponding step size. In other words, the initial velocities would be 2100 m/s in the Y axis and 6000 m/s in the Z axis. Aftwerwards, the upcoming values proceed in the same manner. Sweeping over the range of Velocity Z values and then re-incrementing the Velocity Y to the next sweeping value.

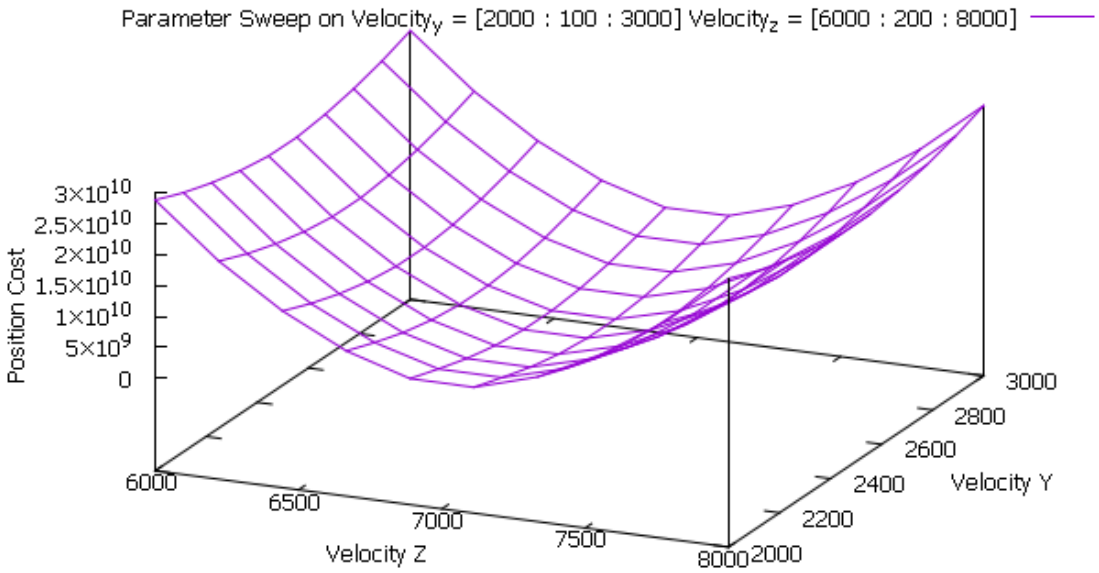

Figure 26: Parameter Sweep Optimization Approach

4.2.2 Gradient Descent Search

This Section presents the output gained by applying the Gradient Descent Search algorithm onto the implemented FMI-based simulator. Firstly, the result of implemeting the classical approach is presented, followed by the results yielded from applying the advanced approach.

Classical Approach

The parameters used to apply the classical approach is presented in Table 5. The parameter being optimized, its initial value, the step size used for it and its unit are presented respectively. The initial velocity used for the parameter "Velocity Y", which represents the velocity exerted in the Y axis, is 3000 m/s and its corresponding delta is 10 m/s. Moreover, the initial

velocity used for the parameter "Velocity Z" is equal to 7500 m/s with a step size of 10 m/s.

Table 5: Parameters used in the Gradient Search Classical Approach

Parameter	Initial Value	Step Size (Delta)	Unit
Velocity Y	3000	10	m/s
Velocity Z	7500	10	m/s

Figure 27 presents the result of applying the classical approach added to the parameter sweep graph. The idea of displaying both plots on the same graph is to show that the optimum velocity Vel_{opt} reached by the classical approach is indeed the minimum point in the parameter sweep plot. In other words, the Vel_{opt} is the velocity vector that would yield the minimal position cost.

The parameter sweep plot is represented via the purple linecolor while the output of the classical approach is represented via the red color. The initial velocity fed to the simulator is Vel_{init} with values 0, 3000 and 7500 m/s applied on the X, Y and Z axes respectively.

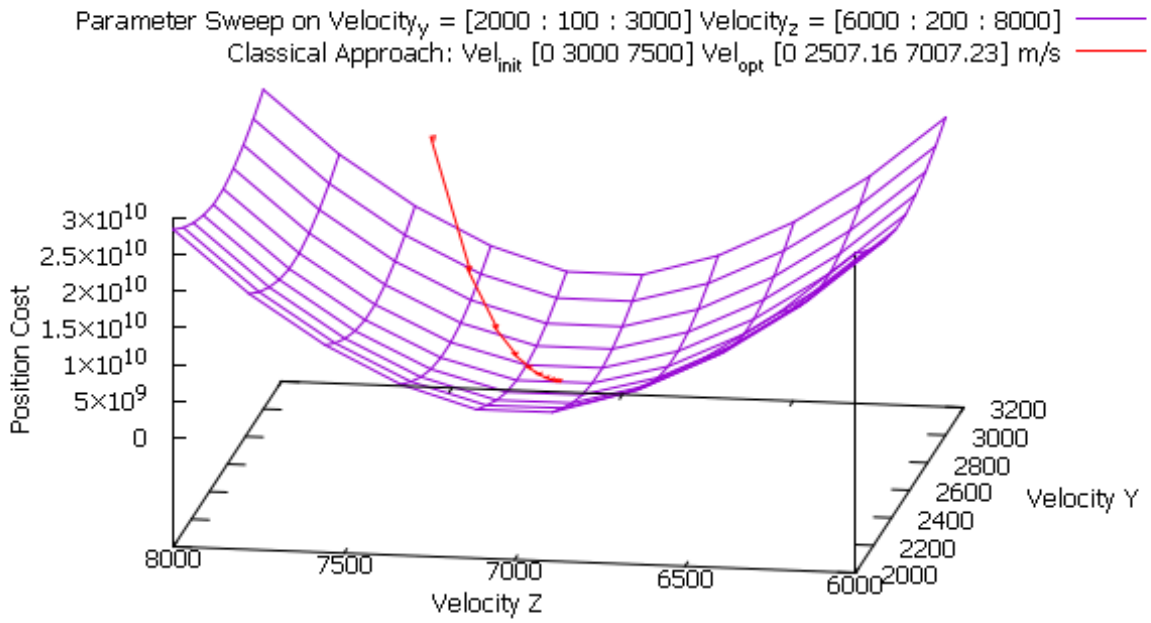

Figure 27: Parameter Sweep and Classical Gradient Search Approaches

The red plot shows that three different vectors of velocity are applied per iteration. Vel_{init} with an additional delta for each of the Y and Z axes respectively. Then, it would shift to the three position cost values gained from applying the next set of valocity vectors. The

shifts gets closer to each other when it gets closer to the minimum or the optimum due to the adjustments in the learning rate.

Advanced Approach

The same parameters and initial values have been used to test and validate applying the advanced optimization approach to the FMI simulator. Table 6 restates the parameters that has been used for the advanced approach.

Table 6: Parameters used in the Gradient Search Advanced Approach

Parameter	Initial Value	Step Size (Delta)	Unit
Velocity Y	3000	10	m/s
Velocity Z	7500	10	m/s

Figure 28 displays the result of applying the advanced optimization approach. Three different colors have been used to display the different optimization techniques. The purple color is used to present the plot resulting from applying the parameter sweep technique. Furthermore, the red color is used to present the classical approach and the green color displays the advanced approach.

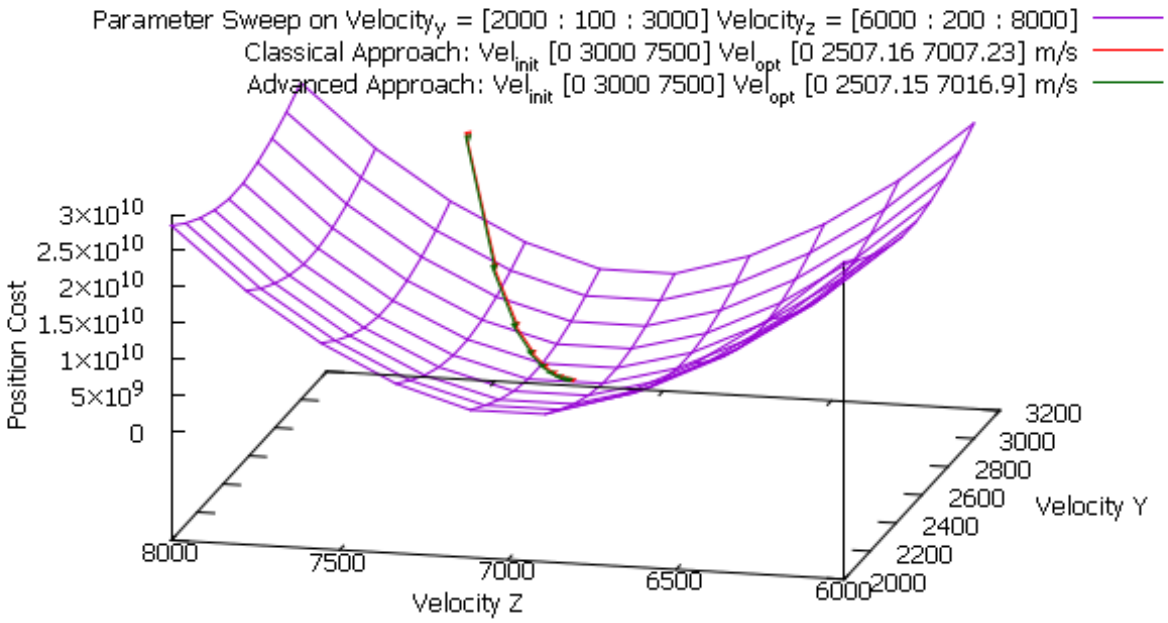

Figure 28: Parameter Sweep, Classical and Advanced Gradient Search Approaches

Both the classical and the advanced approaches were initiated with the same values. A value of 0 m/s has been used for the velocity in the X axis direction. Moreover, values of 3000 m/s and 7500 m/s have been used as velocities for the Y and Z axes respectively. The output of each approach is approximately equal.

5 Conclusion

Spacecraft dynamic simulations are becoming progressively complex. As the complexity increases, special consideration must be given to the software's sustainability, testability, and scalability.

The aim of this dissertation is to tackle several research questions. It offers a novel approach for the spacecraft simulation engineering task. That approach is to utilize the FMI standard in developing the spacecraft simulation system, which has been most effective in models so-simulation at early design phases. A second significant contibution is the incorporation of diffrent design optimization algorithms and techniques within the FMI infrastructure. It includes sensitivity analysis by applying parameters sweeps and incorporation of the Gradient Descent Search algorithm within the FMI standard infrastructure. This thesis employs the accumulation of Jacobian matrices on dynamics and envionmental simulation models, more specifically, on the "Newton's Gravitaional Force" and "Spacecraft Position Dynamics" models.

5.1 Summary

The development of spacecraft simulations follows a specific standard documented in the ECSS-E-TM-10-21 "System Modeling and Simulation" Technical Memorandum. A key feature of this standard is the type of simulator realized at each phase of the development.

Early research efforts have used the Simulation Model Portability (SMP2) standard in space-related simulation development. It has been used in model exchange and portability. However, the simulators compatible with this standard only provide a C++ reference implementation. It does not offer wrapping and encapsulation as FMI standard framework promises.

This thesis introduces the system design architecture of the FMI-based spacecraft simulator. The proposed architecture initiates from model-based system engineering task which delivers the configuration file. The FMI simulator then proceeds with the instantiation of the implemented FMU models and executing the simulation.

An additional contribution of this thesis is the incorporation of the design optimizations within the FMI simulator infrastructure. The sensitivity analysis was performed by performing parameter weeps and implementing the gradient search in two different approaches. The first approach is the classical one. The second approach is rendered by the accumulation of Jacobian matrices across the FMU models.

Investigation of the formulated research questions indicates that utilizing the FMI standard in spacecraft simulation development indeed yielded promising results. The capability of co-simulation in early design stages allow for ease in the maintenance, testing and scalability of the software code base. Furthermore, the possibilty of incorporating optimization techniques within the FMI infrastructure is highly promising, because applying the approach of accumulating Jacobians is not a trivial task in terms of computing gradients for fully coupled nonlinear equations.

The FMI simulator and incorporation of optimization techniques into it have been verified by exploiting a use case and an optimization scenario respectively. The charging and orientation control use case verifies the realization of a FMI simulator and the co-simulation of FMU simulation models representing the environmental and dynamical physics the spacecraft will

encounter. On the other hand, the optimizing of several parameters have been held in the context of an optimization goal or objective. The outcome realizes the optimum based on a cost function. It indicates that incorporating optimization within the FMI infrastructure is possible and advantageous in terms of early design flaws detection. In other words, this would benefit in determining the range of each parameter and a more precise description of the required components before purchasing them.

The advanced approach which applies the gradient search optimization technique on basis of accumulation of adjoint methods, becomes more significant and beneficial for systems with large number of parameters. It was applied for two parameters as it would be more comprehensible.

Later pursuits sought to extend the low-fidelity FMU models to become sophisticted highly detailed models. However, due to time restriction, scalability of the FMU models is not in the scope of the thesis. Nevertheless, different approaches will be described in the upcoming Future Work section.

5.2 Future Work

A straight forward next step is to attempt the FMU models extension into sophisticated ready-to-use FMU simulation models. Each model could be extended with different constraints, capabilities and qualifications. This section will propose a potential approach for the simulation models extension.

Extension of Low-fidelity Models to High-Fidelity Sophisticated Models

A potential exists based on the fact that FMI is tool independent. This denotes that these models could be attained from the components suppliers. Despite of the variant simulation tools that might be utilized by the different suppliers, the simulation models of these components could be exported as an FMU container and incorporated into the simulation.

Figure 29 displays the proposed potential approach for substituting the low fidelity model with a pre-implemented ready-to-use sophisticated model.

This above-described approach requires no changes or adjustments to be carried out in the FMI simulator. However, certainly there are some adaptation that are necessary in the models connection. In other words, work should be done in the interface FMUs (IF FMU) connecting the models together.

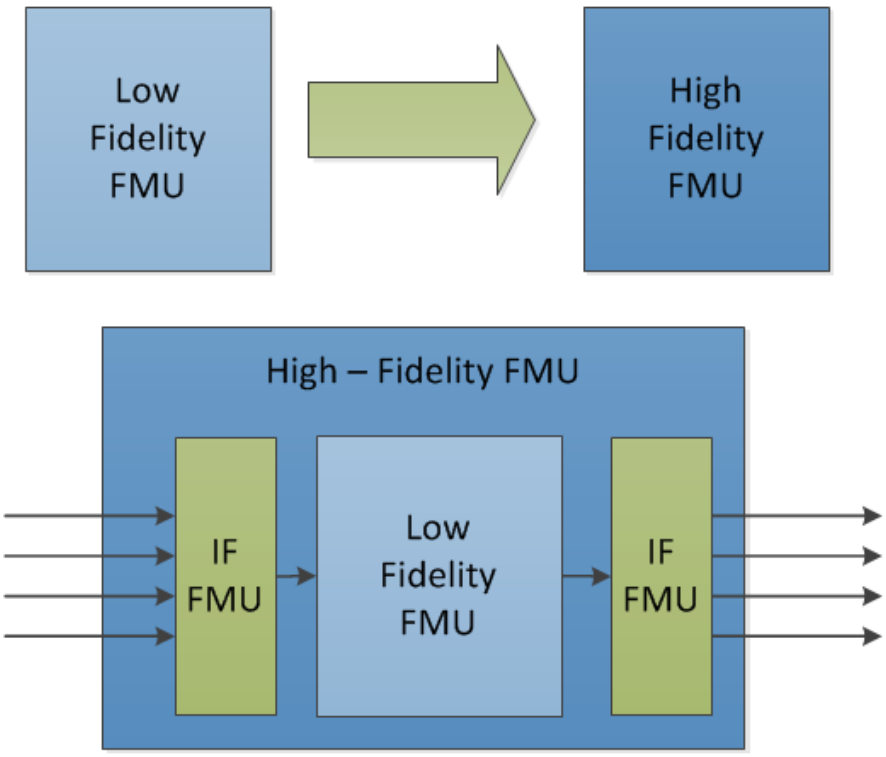

Figure 29: Extension of Low-Fidelity FMU model to High-Fidelity FMU Model

Inclusion of all FMU models in the optimization process

The included FMU models in the optimization scenario are the Newton's Gravitaional Force and Spacecraft Dynamics models. An alternative optimization scenario or objective could be assumed with the purpose of including all FMU models into the optimization. Hence, instead of optimizaing two variables in one of the models. Future work could be completed in order to leverage the number of variables to be optimized.

A generalized optimization scenario would require the adjustment of all FMU models. Two potential approaches exists to accomplish this generalization. The first approach could be adding an interface FMU that would chain the sensitivity matrix of each FMU model together with the next model. Another approach is to define this model extension as a requirement for the developers implementing the models.

III List of Figures

IV List of Tables

V References

[1] NASA *Systems Engineering Handbook (NASA/SP-2007-6105 Rev 1)* National Aeronautics and Space Administration, Washington, D.C. / USA, 2007.

[2] P. M. Fischer, D. Lüdtke, C. Lange, F. Roshani, F. Dannemann and A. Gerndt *Implementing Model Based System Engineering For The Lifecycle of a Spacecraft* Deutscher Luft- und Raumfahrtkongress 2016

[3] ECSS Secretariat *ECSS-M-ST-10C - Space project management* [*Project planning and implementation*]. ESA-ESTEC Requirements and Standards Division, Noordwijk, Netherlands, 2009.

[4] ESA *Virtual Spacecraft Design* [Online]. Available: http://www.vsd-project.org/ Access at 20 4 2016

[5] DLR *Virtual Satellite* https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-5135/8645_read-8374/ Started in 2007

[6] P. M. Fischer, H. Eisenmann und J. Fuchs *Functional Verification by Simulation based on Preliminary System Design Data* 6th International Conference on Systems and Concurrent Engineering for Space Applications (SECESA), Stuttgart, Germany, 2014.

[7] ECSS Secretariat *ECSS-E-TM-10-21A - Space Engineering* [*System Modeling and Simulation*]. ESA-ESTEC Requirements and Standards Division, Noordwijk, Netherlands, 2010.

[8] European Space Agency *SMP 2.0 Handbook* [*EGOS-SIM-GEN-TN-0099*]. Issue 1 Revision 2, 28 October 2005

[9] F. Pace, V. Barrena, N. Lindman, Q. Wijnands *Evolving Infrastructure for Avionics Verification and Validation Activities* Simulation and EGSE facilities for Space Programmes (SESP2010), 30 September 2010

[10] A. Walsh, Q. Wijnands, N. Lindman, P. Ellsiepen, D. Segneri, H. Eisenmann, T. Steinle *The Spacecraft Simulator Reference Architecture* Simulation and EGSE facilities for Space Programmes (SESP2010), 29 September 2010

[11] FMI-Standard.org *Functional Mock-up Interface* [Online]. Available: https://fmi-standard.org/ Access at 28 8 2018.

[12] FMI version 2.0 Guide *Functional Mock-up Interface for Model Exchange and Co-Simulation* July 25, 2014

[13] P. Chombart, Dassault Systems *Multidisciplinary modelling and simulation speeds development of automotive systems and software* ITEA2 innovation report, 2012

[14] E. Drenth, M. Törmänen, K. Johansson, B. A. Andersson, D. Andersson, I. Torstensson, J. Akesson *Consistent Simulation Environment with FMI based Tool Chain* Proceedings of the 10th International ModelicaConference, Lund, Sweden. March 10-12, 2014

[15] ECSS Secretariat *ECSS-E-TM-10-23A - Space engineering* [*Space system data repository*]. ESA-ESTEC Requirements and Standards Devision, Noordwijk, Netherlands, 2011.

[16] P. M. Fischer, M. Deshmukh, V. Maiwald, D. Quantius, A. Martelo Gomez und A. Gerndt *Conceptual Data Model - A Foundation for Successful Concurrent Engineering* Concurrent Engineering: Research and Applications, 2017.

[17] J. Eickhoff *Onboard Computers, Onboard Software and Satellite Operations* [*An introduction*]. Heidelberg, Berlin: Springer-Verlag, 2012.

[18] Uwe Naumann. *The Art of Differentiating Computer Programs* [*An Introduction to Algorithmic Differentiation*]. Society for Industrial and Applied Mathematics, Philadelphia, 2012.

[19] McDonald, C. *Machine learning fundamentals (I): Cost functions and gradient descent* https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220 Nov 27, 2017

[20] J. Hicken, J. Alonso, and Ch. Farhat *AA222 - Introduction to Multidisciplinary Design Optimization* Gradient-Based Optimization Department of Aeronautics & Astronautics, Stanford University , April, 2012

[21] Naumann, U. *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph* U. Math. Program., Ser. A (2004) 99: 399. https://doi.org/10.1007/s10107-003-0456-9

[22] Corliss, G., Faure, C., Griewank, A., Hascoet, L., Naumann, U. *Automatic Differentiation of Algorithms.* [*From Simulation to Optimization*]. Springer, New York, 2002

[23] Griewank, A., Reese, S. *On the calculation of Jacobian matrices by the Markovitz rule.* In [8], pp. 126 - 135, 1991.

[24] Naumann, U. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs.* PhD thesis, Technical University Dresden, Feb. 1999

[25] Griewank, A., Naumann, U. *Accumulating Jacobians as chained sparse matrix products.* Math. Program. 3(95), 555 - 571 2003 (Springer)

[26] BÃŒcker, M., Corliss, G.,Hovland P., Naumann, U., Norris, B. *Automatic Differentiation: Applications, Theory, and Implementations* Springer-Verlag Berlin Heidelberg 2006

[27] O. Montenbruck, E. Gill. *Satellite Orbits* [*Models, Methods, Applications*]. Springer-Verlag Berlin Heidelberg New York, 2000.

[28] M.J.F. Al-Bermani, A. S. Baron *Calculation of Solar Radiation Pressure Effect and Sun, Moon Attraction at High Earth Satellite* Journal of Kufa, PHYSICS VOL.2 NO.1, 2010

[29] P. Ortega, G. Lopez Rodriguez, J. Ricart, M. Dominguez, L. M. Castaner, J. M. Quero, C. L. Tarrida, J. Garcia, M. Reina, A. Gras, M. Angulo *A Miniaturized Two Axis Sun Sensor for Attitude Control of Nano-Satellites* IEEE Sensors Journal, Vol. 10, No. 10, October 2010.

[30] NASA. *Star Camera http : //nmp.nasa.gov/st6/TECHNOLOGY/starcamera.html* Access May, 2004.

[31] T. Urakubo, K. Tsuchiya, K. Tsujita *Attitude Control of a Spacecraft with Two Reaction Wheels* Jet Propulsion Laboratory, California Institute of Technology, December 2017

[32] A. Kron, A. St-Amour, J. de Lafontaine *Four Reaction Wheels Management: Algorithms Trade-Off and Tuning Drivers for the PROBA-3 Mission* The International Federation of Automatic Control, Cape Town, South Africa. August 24-29, 2

[33] NASA. *Reaction/Momentum Wheel https : //spinoff.nasa.gov/spinoff*1997/t3.*html* Retrieved 15 June, 2018

[34] C. C. Huang, P. Y. Ho *FORMOSAT-5 Satellite Power Control and Distribution Unit (PCDU) Development* National Space Organization, Hsinchu, Taiwan

[35] NASA. *Energy Storage Technologies for Future Planetary Science Missions* Graduate School of Engineering, Kyoto University, Japan

A Simulation Models

A.1 Julian Date to Time Converter

First of all, Julian Date (JD) is simply a continuous count of days and fractions of days since the start of the Julian Calender, which is noon Universal Time on January 1, 4713 BC. Julian dates has been widely used by astronomy and applied in many astronomical software to calculate elapsed days between two events or dates. Julian date is commonly used in computer science to calculate the difference between days, since all numbers in the system are consecutive integers. In our simulation, the initial Julain day number is 2451545 which refers to the day starting at 12:00 UT on January 1, 2000 (J2000) [27]

In order to compute the number of Julian centuries since 1.5 January 2000 (J2000), a model was created to perform this conversion by applying the following equation:

$$T = \frac{JD - 2451545}{36525} \tag{10}$$

A.2 Mean Anomaly

To describe the position of an astronomical object in an elliptical orbit, the mean anomaly M is used. It indicates the angle from the pericenter to a point y on a circle (perimeter around the ellipse orbit) relative to the center of the ellipse. This point would result if the object were on an ideal circular path and would move at a constant angular velocity. However, point (y) would have the same orbital period as the object on orbit. Please refer to Figure 30.

With the auxiliary magnitude of the mean anomaly M, orbital motion of an astronomical object can be described more simply than e.g. with the true anomaly θ would be possible.

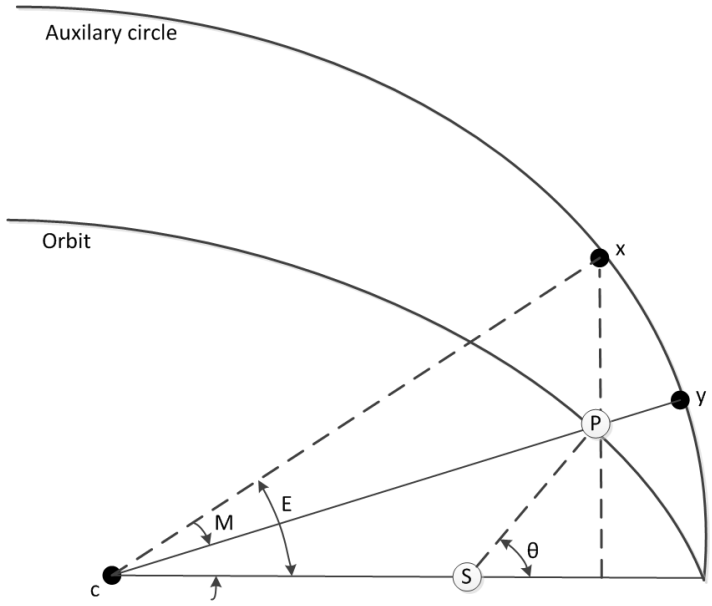

Figure 30: Representation of the mean anomaly M

The mean anomaly determines the position of a body on the circumference surrounding the orbital ellipse at average speed at a certain point in time (T). It is calculated as the sum of

the mean anomaly at reference time M_0 and the product of the angle swept by the body per unit time at medium speed on the auxiliary circle (perimeter around the ellipse) (ς), and the time difference T since the reference time has elapsed.

$$M = M_0 + \varsigma.T \tag{11}$$

where M_0 of the Sun is equal to 6.24 radians and the swept angle of the sun per Julian century ς is equal to 628.302 radians.

A.3 Solar Orbit

Earth travels around the sun. Seen from the earth, it seems as if the sun is moving. The model presented in this subsection will imitate the Solar Orbit.

The earth moves in a slightly eccentric orbit ($\epsilon = 0.017$) around the sun. The closest point to the sun (perihelion) is at a distance of about 147.1×10^6 km, while the point furthest away from the sun is 152.1×10^6 km. The plane of this orbit (ecliptic) is inclined at 23.5 °to the equator of the earth. The intersections of the ecliptic with the equatorial plane move about 50 arcseconds per year. [27]

The movement of the earth around the sun can also be described in an earth-related coordinate system as the apparent motion of the sun on the celestial sphere. The most common coordinate system for this is the so-called Earth-centered inertial (ECI) system. The ECI coordinate system is a geocentric equatorial Cartesian coordinate system whose x-axis points in the direction of the vernier point and whose z-axis is the axis of rotation of the earth.

Also included in the group of ECI coordinate systems is the Earth's Mean Equator and Equinox at 12:00 Terrestrial Time on 1 January 2000 (EME2000) coordinate system used here for the solar vector. To describe the apparent motion of the Sun around the Earth, several orbital elements are required as seen in Figure 31. These elements are the orbital plane which is inclined with respect to the equatorial plane of Earth. The angle through which the two planes are inclined is called inclination i.

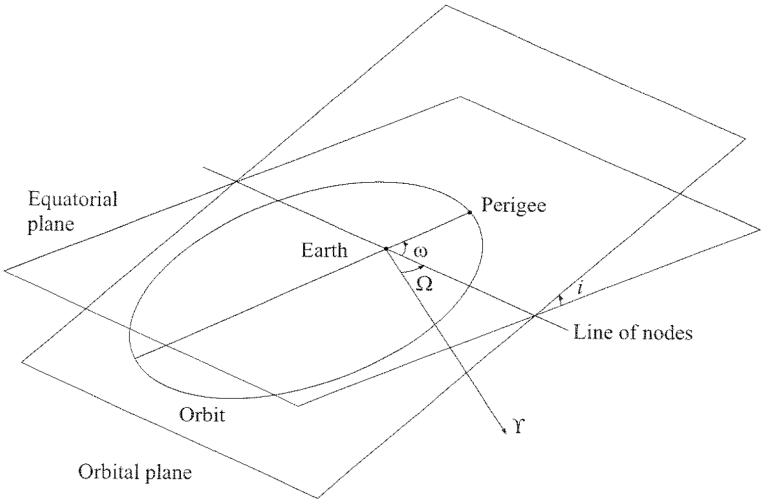

Figure 31: Orbit Elements i, ω and Ω
(Montenbruck and Gill, 2000 [27])

The Sun's apparent orbit pierces the equatorial plane at two points whose connecting line is called a line of nodes. The ascending node is the one where the sun moves from the southern hemisphere to the northern hemisphere, while the descending node is described by the migration of the sun from the northern hemisphere to the southern hemisphere. The angle from the vernal equinox to the ascending node measured in the equatorial plane is called the length of the ascending node Ω. The angle from the ascending node to the perigee (perigee) measured in the orbit plane is called the perigee distance ω.

the first step in the computations of the sun position is the geographic length of the apparent projection of the orbit to the earth in the ecliptic coordinate system. This formula derives from the equations for the general Kepler elements and was extended with two terms of a series expansion to account for confounding factors.

The length of the Sun λ_{sun} is calculated from the sum of the length of the ascending node Ω, the perigee distance ω, the mean anomaly of the sun M, the two terms for the confounding factors and the last summand to take into account the migration of the spring point over time. For this, the epoch of the reference spring point T_{eqx} is given in centuries since the reference time JD_0. [27]

$$\lambda_{sun} = \Omega + \omega + M + 6892^{"}.sin(M) + 72^{"}.sin(2M) + 1.3972^{'}.T_{eqx} \tag{12}$$

where the length of ascending node and the perigee distance $\Omega + \omega$ has the default value 4.938 rad. The next step is to calculate the distance from the Earth to the Sun in the ecliptic coordinate system. This is analogous to Formula 12 again from the original Kepler equation extended by terms of a series expansion.

$$|\vec{r}_{sun}| = (149.619 - 2.499.cos(M) - 0.021.sin(2M)).10^9 m \tag{13}$$

Since the latitude disappears due to the ecliptic coordinate system, the solar vector can be calculated using the inclination of the ecliptic ($\epsilon = 0.409$ rad) in the EME2000 coordinate system as follows.

$$\vec{r}_{sun} = \begin{pmatrix} |\vec{r}_{sun}|.cos(\lambda_{sun}) \\ |\vec{r}_{sun}|.sin(\lambda_{sun}).cos(\epsilon) \\ |\vec{r}_{sun}|.sin(\lambda_{sun}).sin(\epsilon) \end{pmatrix} \tag{14}$$

A.4 Solar Flux

The sun shines in all directions of space due to the chemical processes that take place in its interior. This radiation is used for example by spacecraft as an energy source. Solar cells convert the radiation into electrical energy through the process of photovoltaics.

The radiation that actually arrives on a surface depends primarily on the distance to the sun. In addition, there are variations in radiant power due to irregularities in the chemical processes of the sun.

The Solar Radiation Computation block is based on the position of the sun and is computed as follows:

$$\vec{\Phi}_{Sun} = \sigma.T_{Sun}^4.(\frac{R_{Sun}}{|\vec{r}_{sun}|})^2.||\vec{r}_{sun}|| \tag{15}$$

where the Stefan-Boltzmann Constant σ is 5.67×10^{-8} $W/(m^2K^4)$. The radius of the Sun R_{Sun} is 6.96×10^8 m and the temperature of the Sun is T_{Sun} is 5780 K.

A.5 Earth Eclipse

This model calculates the illumination of the satellite taking into account the shadow cast by a celestial body as it crosses the line from the sun to the satellite. Many satellites are on their orbits at some point on the night side of the celestial body around which they circle. Depending on the nature of the orbit, the shadow that the heavenly body throws traverses again and again. This shadow can be further subdivided into a core and a partial shade. In the following figure, for example, a satellite orbiting the earth is shown.

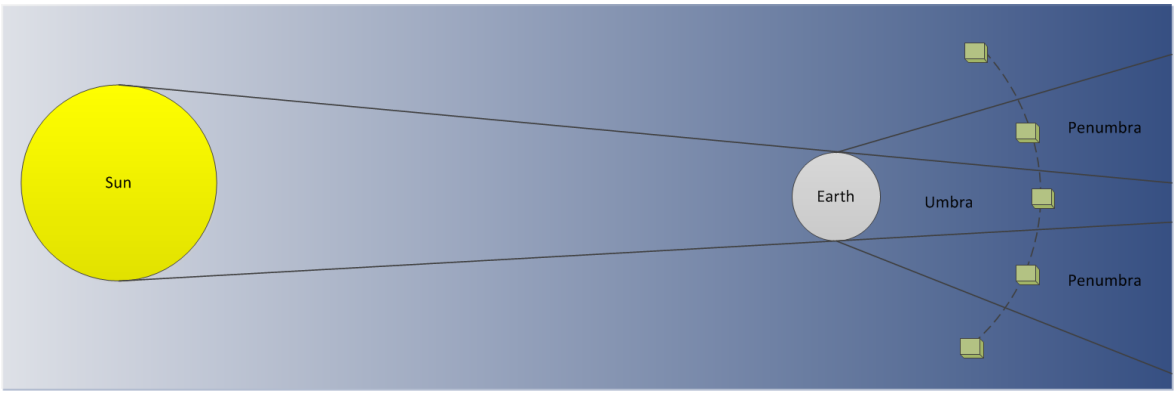

Figure 32: Earth Eclipse (Shadows and penumbra)

The penumbra has the characteristic that different levels of illumination prevail in this area. The closer the satellite gets to the shadow, the darker it gets. And the farther away it is from the shadow, the brighter it will be again until the satellite leaves the semi-shade and is fully illuminated.

For the satellites, the information about the shadow is very important, since within the partial shadow they can only obtain fractions of the energy and within the core shadow no energy from the solar cells and consequently have to switch their energy supply to battery operation.

The model described below for the calculation of the shadow ratios is based essentially on the information given in Chapter 3.4.1 "Eclipse Condition" and 3.4.2 "Shadow Function" [27]. In contrast to the statements in [27], the coordinate system, in the book "Fundamental Plane", was placed in the center of the obscuring celestial body. This has the advantage that no further coordinate system has to be introduced, but the coordinate system for the satellite vector and solar vector can be used. The downside is that both vectors have to refer to the occluding celestial body and therefore the model is not quite as flexible.

Due to the large distances of celestial bodies (e.g., the earth) and sun and the resulting apparent small diameter of the sun, in the model the earth's shadow is described by means of two overlapping circular disks. In this case, the small angle approach is also applied, whereby the sides a (distance $\bar{AC}$), b (distance $\bar{BC}$) and c (distance $\bar{BA}$) from figure 33 correspond to the corresponding angles. The associated angles are each enclosed by the lines from the satellite to the respective vertices of the sides.

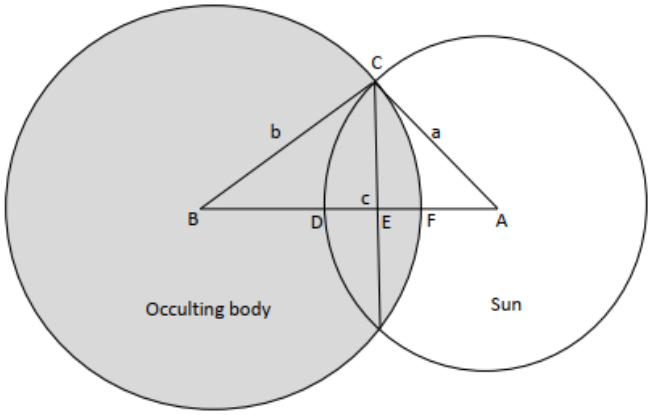

Figure 33: Darkening of the Sun by a Celestial Body

The first side of the triangle in Figure 33 is the apparent radius of the hidden celestial body a. In this case, the hidden celestial body is the sun. The apparent radius from the satellite is calculated as follows:

$$a = arcsin(\frac{R_{Sun}}{|\vec{r}_{Sun} - \vec{r}_{Sat}|}) \tag{16}$$

The following illustration shows the vectors $\vec{r}_{Sun}$, $\vec{r}_{Sat}$ and $\vec{r}_{Sun} - \vec{r}_{Sat}$.

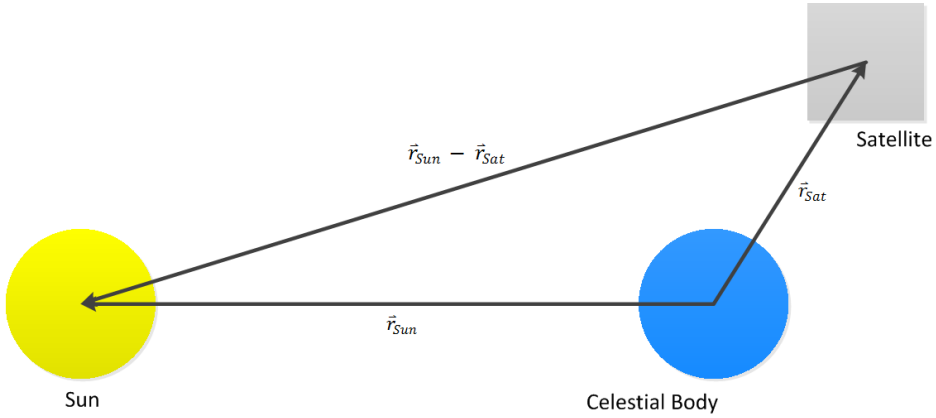

Figure 34: Overview of the Used Vectors

Next, the apparent radius of the obscuring body b can be calculated. For earthbound satellites, this could be Earth, for example.

$$b = arcsin(\frac{R_{Body}}{|\vec{r}_{Sat}|}) \tag{17}$$

The last required size for the triangle in Figure 33 is the apparent distance from the centers of the two celestial bodies c.

$$c = arccos(\frac{-\vec{r}_{Sat}^{T}.(\vec{r}_{Sun} - \vec{r}_{Sat})}{|r\vec{Sat}|.|\vec{r}_{Sun} - \vec{r}_{Sat}|}) \tag{18}$$

After calculating the apparent values a, b and c, the next step is to a feature which calculates if the satellite is in direct sunlight, partial shade or in the core shadow. For this purpose,

the apparent radii of the concealed and the concealing celestial body and the distance whose centers are compared.

$$f_{Sun} = \begin{cases} 1, & \text{if } a+b \leq c \\ 0, & \text{otherwise} \end{cases}$$
$$f_{Penumbra} = \begin{cases} 1, & \text{if } |a-b| < c < a+b \\ 0, & \text{otherwise} \end{cases} \tag{19}$$
$$f_{Umbra} = \begin{cases} 1, & \text{if } f_{Sun} \text{or } f_{Penumbra} \\ 0, & \text{otherwise} \end{cases}$$

The condition $a+b \leq c$ means that the circle of the sun is outside the circle of the obscuring body and therefore no shadow exists. On the contrary, the condition $|a-b| < c$ ensures that the perimeter of the sun is not completely within the circumference of the obscuring body and consequently has not yet adjusted to the core shadow.
The next step is to calculate the area (A) of the intersection of the Occulting Body and the Sun (refer to Fig. 33) using the below equation:

$$A = \begin{cases} 1, & \text{if } f_{Sun} \\ a^2.arccos(\frac{x}{a}) + b^2.arccos(\frac{c-x}{b}) - c.y, & \text{otherwise} \\ 0, & \text{if } f_{Umbra} \end{cases} \tag{20}$$

where $x = \frac{c^2+a^2-b^2}{2c}$ and $y = \sqrt{a^2-x^2}$

The remaining fraction of Sunlight can finally be computed by:

$$\nu = 1 - \frac{A}{\pi a^2} \tag{21}$$

A.6 Newton's Law of Universal Gravitation

Newton's universal gravitation law states that every particle attracts every other particle in the universe with a force which is directly proportional to the product of their masses and inversely proportional to the square of the distance between their centers.

The equation for universal gravitation thus takes the form:

$$F = G\frac{m_1 m_2}{r^2} \tag{22}$$

where F is the force of gravity that is acting between two objects, G is the gravitational constant which value is approximately $6.674 \times 10^{-11}\ N.kg^{-2}.m^2$. The two objects have the masses m_1 and m_2 and the distance between the center of their masses is r.
Implementing the Newton's gravity law as a simulation model would be useful for simulating spacecraft missions orbiting Earth or in the near proximity of Earth.

A.7 Spacecraft Dynamics

Dynamics of a Spacecraft is the modeling its position and orientation in response of external forces acting on it. Several forces should be taken in account for a launch from Earth such as engine thrust, aerodynamic forces, and gravity. In this section, the models which simulates the position of a spacecraft in outer space as well as its orientation will be presented.

• **Position**

The purpose of the Position Dynamics simulation model is to keep track of the position of a spacecraft in outer space. The position of the spacecraft is compute dwith respect to the center of Earth. For instance, if a low-earth orbit is taken into consideration, the position would refer to the exact location of the spacecraft in its orbit around Earth.

• **Orientation**

The orientation a spacecraft is simply the angular rotation of a body with respect to its center coordinate frame. As we know, any spinning body follows Newton Laws of Motion. The mass moment of inertia of an object, I, is described by the distribution of mass on that object. Based on Newton's Second Law, we would now be able to perceive how to relate torque and precise force. Similarly as force is equal to the time rate of change of linear momentum, torque is the time rate of change of angular momentum. In other words, in the event that we apply a torque to an object, its angular momentum will change. At the point when torque is zero, angular momentum remains constant.

If we apply a force to an object, it will accelerate. In the same manner, if we apply a torque to a free-floating object, it will begin to rotate with higher speed. That is, it will encounter angular acceleration α. Hence, the angular acceleration could computed given a precise torque T and the object's mass moment of inertia. This concept is summarized in Equation (23)

$$\vec{T} = I\vec{\alpha} \tag{23}$$

where I is the mass moment of inertia measured in $(kg \cdot m^2)$ and α is the angular acceleration measured in $(rads/s^2)$.

A.8 Solar Panel

Spacecrafts operating in Solar systems depends on Solar Panels in order to convert the Solar Energy into Electric Energy. Equation 24 describes the energy gained by the solar panel from the solar radiation pressure acting on the surface of the satellite [28]. The gained energy is directly proportional to the effective satellite surface area, to the solar flux and to the efficiency of the solar panel. Thus, the energy resulting from the solar radiation pressure $\vec{f}_{solar}$ is

$$\vec{E}_{solar} = A.\eta.\vec{\Phi}_{Sun}.\gamma.cos(\theta) \tag{24}$$

where

A is the cross sectional area of the solar panel
η is the efficiency of the solar panel
$\vec{\Phi}_{Sun}$ is the Solar Radiation pressure acting on the satellite
γ is the lighting coefficient ranging from 0 to 1 for umbra, penumbra to total sunlight phases
θ is the angle between the position of the sun and the position of the spacecraft

A.9 Sun Sensor

Sun Sensor is one of the attitude sensors that measures the sun vector for attitude determination. Sun Sensors are used to maximize energy conversion efficiency [29]. In satellite applications, measuring the sun vector in any position in a 360°range is a common necessity.

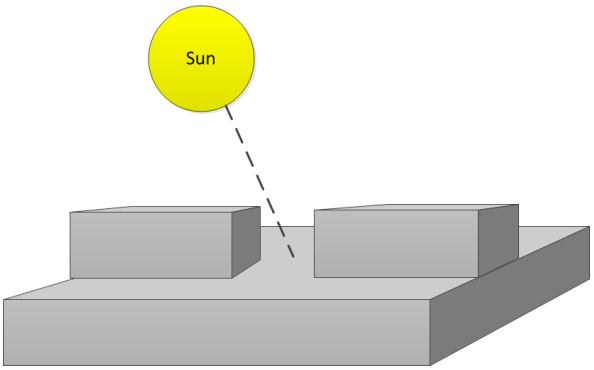

Figure 35: Simplified Sun Sensor that measures sun incidence angle

In normal sun sensors, a thin cut at the highest point of a rectangular chamber enables a line of light to fall on a variety of photodetector cells at the base of the chamber. By arranging two sensors perpendicular to one another, the direction of the sun can be calculated.

A.10 Star Tracker (STR)

A star tracker is an optical device used to estimate the stars positions utilizing photocells or a camera. Astronomers have measured with high accuracy the positions of many stars. The star tracker on the spacecraft or the satellite uses these measurements to determine the position and orientation of the spacecraft with reference to the stars [30]. More precisely, the star tracker has a star catalog containing the stars images which is used to compare the stars positions and the apparent position of the stars with respect to the spacecraft. Furthermore, a star tracker can distinguish stars by comparing the pattern of observed stars with the already known stars in the sky via an incorporated processor.

A.11 Reaction Wheels (RW)

Reaction wheels and gas thrusters are two commonly used actuators for controlling the attitude of a spacecraft. In general, they are located on the three principal axes of inertia of the spacecraft to produce three independent torques [31]. The majority of the Reaction Wheels Assembly (RWA) provides redundancy by including an additional reaction wheel. The standard methodology utilizes three of the four RWs to adjust the control torques but there are advantages in utilizing at least four skewed RWs [32].

Reaction wheels are utilized to control the orientation of a spacecraft without the utilization of thrusters. Hence, decreasing the fuel required for mass fraction. They work by having an electric engine connected to a flywheel that would change the rotation speed leading to the spacecraft to rotate in a counter-rotation proportional to the angular momentum. Reaction wheels can pivot a shuttle just around its focal point of mass; they are not fit for changing the position of the spacecraft. [33]

A.12 Power Control and Distribution Unit (PCDU)

There are three basic components in the electrical power subsystem of a spacecraft. The solar array, the battery and the power control and distribution unit (PCDU). The PCDU is similar to a human heart. It acts as the power control and distribution center in the spacecraft. The main purpose of the PCDU is to condition and distribute energy between the solar panels and the rest of the payloads in the spacecraft. During sunlight, the PCDU receives solar power

from the solar arrays, recharges the battery and controls the power distribution to various payloads in the spacecraft. While during eclipse, the battery powers the spacecraft through a bus in the PCDU [34].

A.13 Battery

Battery is one of the electrical power subsystem components in a spacecraft. The light energy from the sun is converted into electric energy by the solar array. The solar array then recharges the battery in sunlight, so that it would provide electrical power energy to the spacecraft during periods when the spacecraft is in earth eclipse.

Almost all spacecrafts used in planetary science missions nowadays use primary non-rechargeable batteries, secondary rechargeable batteries as well as capacitors in their energy storage devices.

Rechargeable batteries are electrochemical devices that convert the electric energy into chemical energy during charge and chemical energy into electrical energy during discharge several times. They are used to provide power for the spacecraft launch before deploying the solar panels, as well as providing power to the payloads during eclipse for operations, mobility and any other anomalies that might occur. Rechargeable batteries used in space missions include: silver-zinc (Ag-Zn), nickel-cadmium (NiCd), nickel-hydrogen (Ni-H2), and more recently, lithium-ion (Li-ion). Nowadays, the technology of choice for most of the aerospace applications is the Li-ion rechargeable batteries [35].

B Simulator Source Code

In order to get a better understanding on how the FMI simulator works, the steps performed by the simulator will be presented in more details.

```
1         Simulator oSimulator;
2         oSimulator.InitFromConfig(config);
3         oSimulator.Start();
```

Reading the JSON Configuration File

```
1         Simulator ReadSimulatorConfig(std::string const& filepath)
2         {
3         std::cout << "Reading simulator config from json file: " << filepath << std::
              endl;
4         Simulator oConfig;
5         std::ifstream i(filepath);
6         json          j_settings;
7         i >> j_settings;
8
9         return j_settings;
10        }
```

Initialization Step

After reading the JSON configuration file, the simulator starts initializing the different blocks. It initializes the state vector, components and interfaces respectively.

```
1         void Simulator::InitFromConfig(std::string strConfig)
2         {
3         // Initializing the simulator from JSON config
4         cfg = SimulatorConfig::ReadSimulatorConfig(strConfig);
5
6         // Initializing the State Vector
7         std::map<std::string, double>::iterator it;
8         for (it=cfg.mapStateVectorConfig.begin(); it !=cfg.mapStateVectorConfig.end();
               ++it)
9         {
10        m_oSimSateVector.addState(it->first, it->second);
11        }
12
13        // Initializing the Simulation Duration
14        m_dSimStart = cfg.dStart;
15        m_dSimEnd = cfg.dEnd;
16        m_dStepsize = cfg.dStepsize;
17
18        // Initializing the Components
19        for (int i = 0; i < cfg.mapComponents.size(); ++i)
20        {
21        FMUSimulation* pObj = new FMUSimulation;
22
23        pObj->InitFMU(cfg.mapComponents[i].path, cfg.mapComponents[i].unzipPath, cfg.
              dStart, cfg.dEnd, cfg.dStepsize);
24
25        // Setting the simulation parameters
26        std::map<std::string, double>::iterator it_param;
27
28        for (it_param = cfg.mapComponents[i].mapWriteParameters.begin(); it_param !=
              cfg.mapComponents[i].mapWriteParameters.end(); ++it_param)
29        {
30        pObj->SetSimulationVariableValue(it_param->first, it_param->second);
31        }
32
```

```
33          m_mapSimulatedComponents[pObj] = cfg.mapComponents[i];
34          }
35
36          // Initializing the interfaces
37          for (int i = 0; i < cfg.mapInterfaces.size(); ++i)
38          {
39          FMUSimulation* pObj = new FMUSimulation;
40          pObj->InitFMU(cfg.mapInterfaces[i].path, cfg.mapInterfaces[i].unzipPath, cfg.
                dStart, cfg.dEnd, cfg.dStepsize);
41          m_mapSimulatedInterfaces[pObj] = cfg.mapInterfaces[i];
42          }
43          }
```

Iteration over all Components and Interfaces

Consequently, the simulator starts executing the simulation task. It begins simulating each of the components and interfaces configured in the JSON file.

```
1           void Simulator::Start()
2           {
3           std::map<FMUSimulation*, SimulatorConfig::FMUComponent>::iterator itComp;
4           std::map<FMUSimulation*, SimulatorConfig::FMUInterface>::iterator itInterface;
5
6           double sim_time = m_dSimStart;
7           double end = m_dSimEnd;
8           double h = m_dStepsize;
9
10          while (sim_time <= end)
11          {
12          // Simulation of all Components
13          for (itComp = m_mapSimulatedComponents.begin(); itComp !=
                m_mapSimulatedComponents.end(); itComp++)
14          {
15          FMUSimulation* sim = itComp->first;
16          SimulatorConfig::FMUComponent config = itComp->second;
17
18          std::map<std::string, std::string>::iterator itInputs;
19          for (itInputs = config.mapReadSlots.begin(); itInputs != config.mapReadSlots.
                end(); itInputs++)
20          {
21          double varTmp = m_oSimSateVector.getState(itInputs->first);
22          sim->SetSimulationVariableValue(itInputs->second, varTmp);
23          }
24
25          sim->DoStep();
26
27          std::map<std::string, std::string>::iterator itOutputs;
28          for (itOutputs = config.mapWriteSlots.begin(); itOutputs != config.
                mapWriteSlots.end(); itOutputs++)
29          {
30          double varTmp = sim->GetSimulationVariableValue(itOutputs->first);
31          m_oSimSateVector.addState(itOutputs->second, varTmp);
32          }
33          }
34
35          // Simulation of all Interfaces
36          ...
37          }
38          }
```

Do Step

The simulator calls the *DoStep*() function on each iteration of the components and interfaces. The *DoStep*() is a function in the FMI 2.0 for Co-Simulation. It manages the computations at each time step.

```
fmi2Status fmi2DoStep(fmi2Component c,
fmi2Real currentCommunicationPoint,
fmi2Real communicationStepSize,
fmi2Boolean noSetFMUStatePriorToCurrentPoint);
```

where *currentCommunicationPoint* and *communicationStepSize* are the current communication point of the master and the communication step size respectively. The *communicationStepSize* must be greater than 0.0. Furthermore, the argument *noSetFMUStatePriorToCurrentPoint* is set to fmi2True if in this simulation run, the $fmi2SetFMUState$ will no longer be called for instants of time before *currentCommunicationPoint*.