MASTER THESIS

OPS-SAT Software Simulator

Author:
Silviu Cezar SUTEU

Examiner:
Prof. Dr. Andreas Nüchter
Informatics VII : Robotics and Telematics

Examiner:
Anita Enmark
Computer Science, Electrical and
Space Engineering Department

Supervisor:
Dr. Mehran SARKARATI
European Space Operations Centre
Human Spaceflight and Operations
Application and Special Projects Data Systems

A thesis submitted in fulfillment of the requirements
for the degree of Master of Space Science and Technology

September 21, 2016

JULIUS MAXIMILIAN UNIVERSITY WÜRZBURG
LULEÅ UNIVERSITY OF TECHNOLOGY

Declaration of Authorship

I, Silviu Cezar SUTEU, declare that this thesis titled, "OPS-SAT Software Simulator" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at these Universities.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at these Universities or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"There is a single light of science, and to brighten it anywhere is to brighten it everywhere."

Isaac Asimov

JULIUS MAXIMILIAN UNIVERSITY WÜRZBURG
LULEÅ UNIVERSITY OF TECHNOLOGY
EUROPEAN SPACE OPERATIONS CENTER

Abstract

Master of Space Science and Technology

OPS-SAT Software Simulator

by Silviu Cezar SUTEU

OPS-SAT is an in-orbit laboratory mission designed to allow experimenters to deploy new on-board software and perform in-orbit demonstrations of new technology and concepts related to mission operations [1]. The NanoSat MO Framework facilitates the process of developing experimental on-board software for OPS-SAT by abstracting the complexities related to communication across the space to ground link as well as the details of low-level device access. The objective of this project is to implement functional simulation models of OPS-SAT peripherals and orbit/attitude behavior, which integrated together with the NanoSat MO Framework provide a sufficiently realistic runtime environment for OPS-SAT on-board software experiment development. Essentially, the simulator exposes communication interfaces for executing commands which affect the payload instruments and/or retrieve science data and telemetry. The commands can be run either from the MO Framework or manually, from an intuitive GUI which performs syntax check. In this case, the output will be displayed for advanced debugging. The end result of the thesis work is a virtual machine which has all the tools installed to develop cutting edge technology space applications.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

OREKIT	**O**rbital **E**xtrapolation **Ki**t
CCSDS	**C**onsultative **C**ommittee for **S**pace **D**ata **S**ystems
MO	**M**ission **O**perations
SVF	**S**oftware **V**alidation **F**acility
ICD	**I**nterface **C**ontrol **D**ocument
AGSA	**A**dvanced **G**round **S**oftware **A**pplication
TLE	**T**wo **L**ine **E**lement
IMU	**I**nertial **M**easurement **U**nit
GUI	**G**raphical **U**ser **I**nterface

Chapter 1

Introduction

1.1 OPS-SAT Mission

The OPS-SAT spacecraft is a Cubesat mission run by ESA and TU Graz in order to demonstrate advanced technologies for operating satellites. To quote from [1], among the goals there is breaking the cycle *has never flown, will never fly*. As can be seen from figure 1.1, the spacecraft has a 3U form factor design with solar panels on all sides, to ensure it will produce power regardless of attitude.

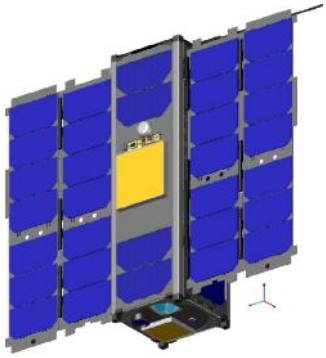


FIGURE 1.1: OPS-SAT

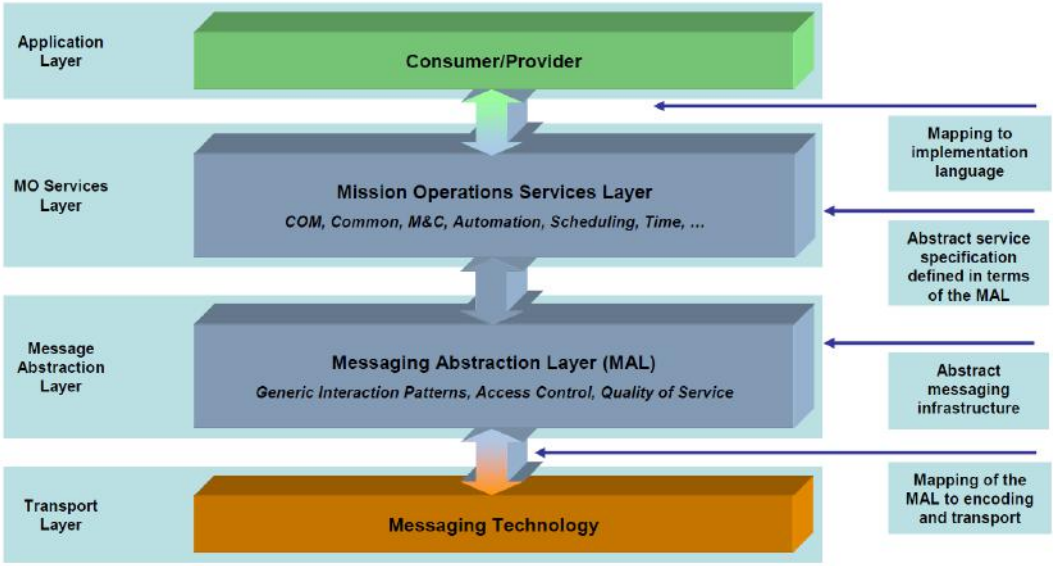


FIGURE 1.2: Overview of MO Framework concept

1.2 NanoSat MO Framework

The CCSDS committee proposes in [2] a service oriented architecture, called the MO Framework. The main idea is to achieve re-usability and compatibility between components by separating layers that adhere to standardized interfaces. In Figure 1.2 there is depicted a generic, platform independent overview of the framework.

As described in [3] and [4], the NanoSat MO Framework is a JAVA implementation of the MO framework. Figure 1.3 represents the conceptual organization: Apps are the high-level products on board which interact through the NanoSat API with the spacecraft peripherals and ground segment. OPS-SAT will be the reference mission for NanoSat MO Framework.

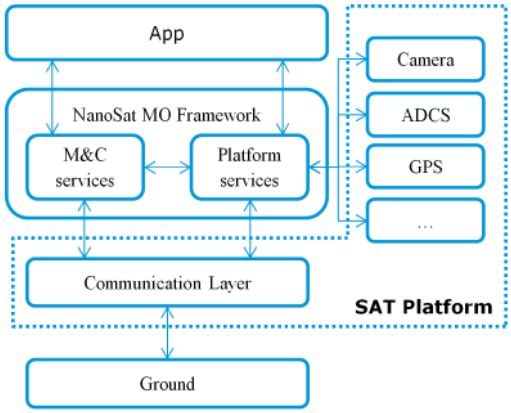


FIGURE 1.3: Overview of NanoSat MO Framework

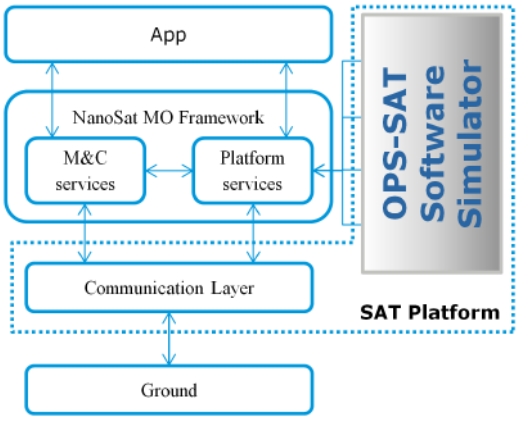


FIGURE 1.4: Integration of software simulator into NanoSat MO Framework

1.3 Software simulator

As shown in figure 1.4 the simulator integrates in the NanoSat MO framework and provides data to the platform services. To achieve the integration goal easily, it is a standalone software component developed in Java which is designed to be abstract and permit reuse of most modules. It consists of processing, communication and

graphical display classes which are organized in an efficient way, according to the object-oriented programming paradigm. In chapter 2 the design choice of classes and threads is detailed and explained. Ensuring that data exchanged with the devices strictly respects protocols defined in interface design documents is also the role of the simulator as software validation facility, as presented in section 2.1. The graphical interface application is meant to assist both NanoSat MO framework developers and experimenters by providing an easy and intuitive way of interacting with operation of the simulator. The functions of the GUI are detailed in chapter 3. Finally, chapter 4 contains information about various peripherals modeled and simulated:

- GPS: Receiver unit which processes signals from the GPS constellation to determine the spacecraft position, section 4.2.
- FineADCS: An attitude control system containing reaction wheels and magnetorquers, section 4.1.
- Camera: HD camera for Earth observation experiments, section 4.3.
- Software Defined Radio: Broadband receiver unit, section 4.4.
- Optical receiver: A device which converts received pulsed light stream into data pattern, section 4.5.

In the context of OPS-SAT, the simulator covers the functionality of the OPS-SAT peripheral devices by either being instantiated directly within the MO framework or by running on a separate machine and performing communication protocol functions. These two use cases are referred to as "Platform Software Simulator" and "Platform Raspberry PI Target". This two-fold use case will ultimately support the development and implementation of the NanoSat MO Framework. In chapter 5 the actual outcome of the thesis work is described, namely:

- a development environment that encompasses all the tools needed to create applications for space
- a hardware target platform deployment of the NanoSat MO Framework running an instance of the simulator with some additional sensor input
- a re-sum of the conclusions drawn.

Chapter 2

Simulator application organization

2.1 Software validation facility

The simulator interfaces are based exclusively on Java primitive types. This ensures maximum cross-compatibility over different architectures and environments. Table A.1 shows the supported data types with added limit information. The term Software Validation Facility refers to a system that is intended to support the test and validation of on-board software. Usually it contains a copy of the target processor and a simulated environment for the peripherals involved. This is also known as hardware in the loop testing, when a part of the flight computing stack is exposed to emulated stimuli in order to validate software response. The test scenarios extend over many areas, as follows:

- Protocol level data communication, correct data telegram sizes, error checking
- Middle-ware level operation, assessment of the software modules capacities to cope with data rates and bandwidths of communication
- Application level functions, troubleshooting and validating mission goals

2.2 Device interface control documents

The suppliers of each component of the OPS-SAT provide an ICD file, which details the communication protocol and data commands information. Figure 2.1 shows an extract from such file, outlining several data commands to the fine attitude control and determination device. Each row in the ICD document is converted into a java method which constitutes an interface that the simulator exposes and the NanoSat MO Framework utilizes. For example, the input parameters with fixed data size, the return parameters with fixed data size are declared into a java interface file, as per listing 2.1.

```
1 byte Identify();
2 void SoftwareReset();
3 void I2CReset();
4 void SetDateTime(long seconds,int subseconds);
```

LISTING 2.1: Java interface translation

2.3 Task node

The building block of the simulator is the task node. This is a class that is designed to run independently and to exchange data asynchronously to another task node, as

5.5 Command and Telemetry Description

5.5.1 iADCS Main Processor: iAD = 0xAA

cID	N_Param	Cmd	N_Data	T_exec [ms]	Description
0x01	0	Identify	8	< 10	0..7: 'i'A'C'D'S'1'0'0'
0x02	1	Software Reset	0	< 10000	-
0x03	0	I2C Reset	0	-	TBD
0x05	6	Set Date+Time	0	-	Epoch: 01.01.1970 0:00:00 UTC UI32: Seconds UI16: Sub-seconds [msec]
0x06	0	Get Date+Time	6	TBD	Epoch: 01.01.1970 0:00:00 UTC UI32: Seconds UI16: Sub-seconds [msec]

FIGURE 2.1: Example ICD entry

shown in figure 2.2. The design choice supports multiple running configurations and efficient utilization of system resources. Each task node runs the following execution sequence in a cyclical loop: data input, processing, data output. For the interface to another task node, concurrent linked queues [5] are used. Each task node has an input queue and an output queue which is shared with an other task node's output and input queues, respectively. A task node is also capable of

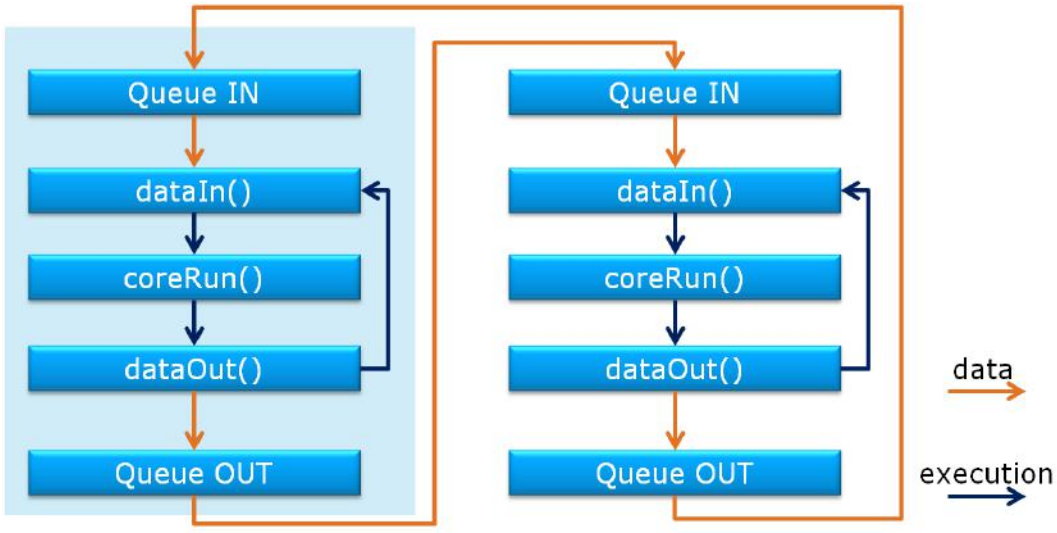


FIGURE 2.2: Task nodes

generating logging messages with a settable level. The logging messages can either be output to the console or to files on local storage. Figure 2.3 shows the thread structure of the simulator running with all the settings on (see section 2.4.5).

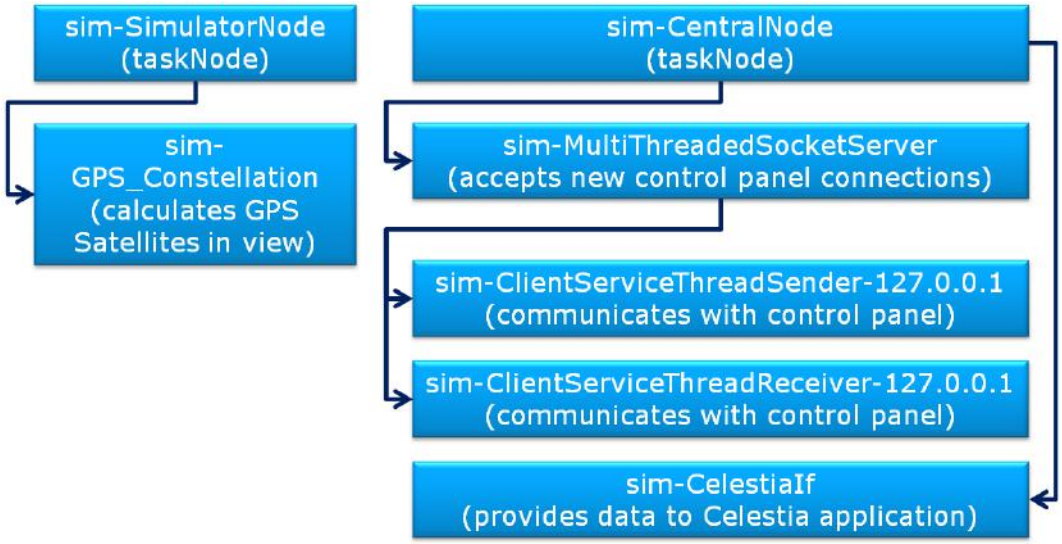


FIGURE 2.3: Independent running threads on a simulator instance

2.4 Simulator node

The simulator node extends the base class task node, handles the main application logic and is able to fulfill the following requirements independently:

1. Hold instances of simulated peripheral devices classes which expose the direct commands
2. Load all settings from files
3. Keep track of the simulated time with possibility to speed up time
4. Initialize and run models for the simulated peripheral devices

The UML class diagram with the related classes and interactions can be found in figure A.2.

2.4.1 Time system

The simulator node evaluates on every processing loop the system time and obtains how many milliseconds have elapsed since last execution. This amount is then multiplied by the time acceleration factor and then fed into the simulated models. It is possible to stop the time. In this case, the time elapsed will be default to zero each processing loop.

2.4.2 Logging system

Each task node creates a log file which records run-time information depending on the logging level selected. The logging level is initialized from an entry in the simulator header file. A sample of a log file content is shown in listing A.4.

FineADCS	1001
GPS	2001
Camera	3001
Nanomind	4001
FDIR	5001
SDR	6001
Optical Receiver	7001
CCSDS Engine	8001
MityARM	9001

TABLE 2.1: Internal IDs

2.4.3 Configuration files

On starting the application, configuration files are looked up in the current directory. This means a different location depending on which application instantiates the simulator. If these files are not present,some error messages will be logged, default settings will be used. There are three different settings files which correspond to modules of the application:

1. Header - contains general information like start/stop time, automatically start models and time, use optional modules (like Orekit, Celestia).
2. Templates - contains definitions for calling simulator commands (as defined in 2.4.4)
3. Scheduler - contains a list of time-tagged command IDs with respective templates, to be executed within the simulator time.

2.4.4 Simulator commands

As introduced in 2.2, each interface defined from the ICDs is a list of commands or the simulator operates with commands from the interface. All the commands have an internal ID, which is numbered according to the table 2.1. To ensure that commands are defined consistently with regard to the defined execution chains an Excel file is used, which takes all the information about a command and generates Java code stubs based on it. There are three code stubs automatically generated by the Excel file:

- interface file method stub
- device class forward call
- simulator class case processing switch handler

The content of the Excel file is arranged in three parts: input, intermediate processing and formatted java code output. For each command the following fields are necessary:

1. interface name
2. device name
3. command name
4. number of input arguments

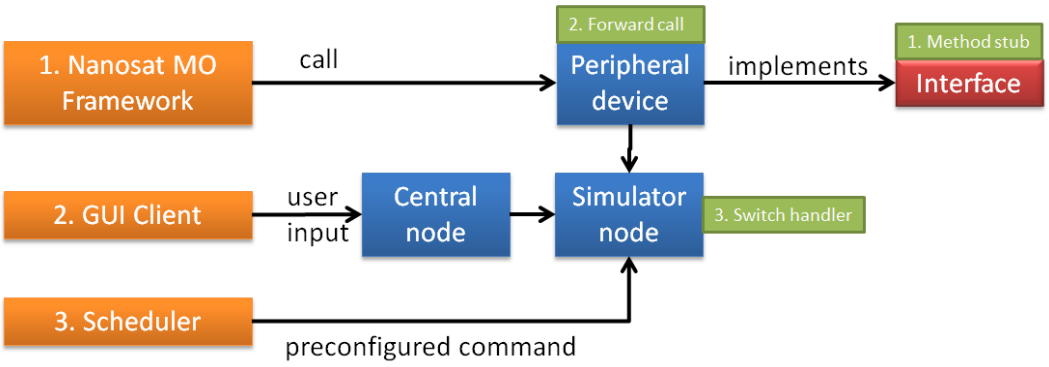


FIGURE 2.4: Data flow of simulator commands. Orange: execution chains. Blue: Java classes. Red: Java interfaces. Green: generated code.

Method	Caller	Comments
Direct calls to peripheral devices	MO Framework services	The results (if defined) are returned to the caller function. If the command failed, the result will be null.
Manual/user call	GUI client	The results are returned to the GUI client, which displays them. If the command failed, the exception and/or error information will also be displayed.
Call from the simulator scheduler	Scheduler	The results are returned to the simulator scheduler itself, which discards them.

TABLE 2.2: Command execution chains

5. type and name for each input argument
6. return variable type (if any)
7. return variable size (if an array)
8. description about the command

The appendix table A.9 and listings A.1 A.2 A.3 contain the formulas to generate the three types of code stubs depicted in figure 2.4 and table 2.2. To illustrate the working of the code generation a method belonging to the GPS device will be explained, namely the one used to query the GPS receiver with NMEA identifiers.

Interface file method stub

At the interface file level, the description of the method is added within <pre> brackets in order to embed pre-formatted javadoc comments. Since the peripheral device classes implement the interface, all methods defined using the stub listed in 2.2 will have to be overridden in order to pass the Java compilation.

```
<pre>
Obtain a NMEA response for a given NMEA sentence
Input parameters:String inputSentence
Return parameters:String
Size of returned parameters: 0

</pre>
*/
String getNMEASentence(String inputSentence);//2001
```

LISTING 2.2: Java interface stub definition

Device class forward call

This stub fulfills two roles:

1. redirect an incoming method call to the main simulator method
2. embed additional information used by the GUI client

To ensure all command executions go through the same function, the method runGenericCommand is used, which takes input parameters internal ID and a collection of argument objects (referred to as ArgumentDescriptor in the simulator terminology). This collection can hold any kind of objects and during the execution it will be cast to the expected data type. The simulator uses reflection [6] to create a list of all the commands supported by the devices. This list will be sent to the GUI client program (see chapter 3) whenever the new client is connected. For example, the java file listed in 2.3 which implements the GPS device contains a method called getNMEASentence. At compilation, the Java byte code file contains embedded annotation @InternalData and the header of the method itself. During run time, at start up, the simulator will use reflection to infer that a method called getNMEASentence exists, which accepts a string type parameter that is named inputSentence. By loading the commands list from the compiled code itself, program consistency at function, parameters and return type level is ensured.

```
@Override
@InternalData (internalID=2001,commandIDs={'',''},argNames={''
    inputSentence''})
public String getNMEASentence(String inputSentence) {
  ArrayList<Object> argObject = new ArrayList<Object>();
  argObject.add(inputSentence);
  return (String) super.getSimulatorNode().runGenericMethod(2001,
    argObject);
};
```

LISTING 2.3: Java device class forward function with annotations

Simulator class switch handler

The switch handler is a part of the main simulator function which processes commands. Its role is to unpack the list of argument objects into the corresponding type. The internal ID is used in the function input parameters to decide which command will be called.

```
case 2001: {//Origin [IGPS] Method [String getNMEASentence(String
    inputSentence);//201//Obtain a NMEA response for a given NMEA sentence
    ]
```

```
  String inputSentence = (String) argObject.get(0);
  String result = '';
  globalResult=result;
  break;
```

LISTING 2.4: Java simulator node switch case stub

2.4.5 Simulator header

This file is named "OPS_SAT_SIMULATOR-header.txt" and contains options for the run time, such as enabling/disabling modules. It is loaded at start up and when there is a change made on one of the clients (see chapter 3). The options available are shown in table 2.3 and a sample entry in listing 2.5.

```
#Run the processing of internal models
startModels=true
#Increment the simulated time (depends on startModels)
startTime=true
#Speed up of time factor
timeFactor=1
#Enable the Orekit library
orekit=true
#Configuration of the Celestia server
celestia=true
celestiaPort=5909
#Start and end dates of simulation
startDate=2016:09:14 14:07:16 CEST
endDate=2016:09:14 14:07:16 CEST
#Logging level to files found in USER_HOME/temp/_OPSSAT_SIMULATOR/
#Possible values SEVERE,INFO,FINE,FINER,FINEST,ALL
centralLogLevel=SEVERE
simulatorLogLevel=INFO
consoleLogLevel=INFO
```

LISTING 2.5: Simulator header example content

2.4.6 Simulator input argument templates

Each command has a number of different input argument templates. These offer a convenient way to define executions of actions that rely on constant parameters. For example listing 2.6 shows several overloaded expressions of the same command getNMEASentence with ID 2001, however with different argument templates. The templates can be used in conjunction with execution methods from control panel and scheduler.

```
2001|GLMLA|inputArgs=[String inputSentence={GLMLA}]
2001|GPALM|inputArgs=[String inputSentence={GPALM}]
2001|GPGGA|inputArgs=[String inputSentence={GPGGA}]
```

LISTING 2.6: Argument templates definition.

2.4.7 Simulator commands results

A command result is the output of an executed command. It can be successful or failed, if there was an exception during the execution. The simulator GUI client shows the complete information for each executed command in manual mode. The structure of a command result is designed to contain all the relevant details. For

Field	Comments
startModels=[true/false]	If true, the simulator will enable model loops for all devices (including time system) at start up.
startTime=[true/false]	If true, the simulator will enable time system keeping at start up. This setting is dependent on start models.
orekit=[true/false]	If true, orekit class will be instantiated and used as a provider for simulation data (orbit, attitude, etc). See chapter 4.1.
celestia=[true/false]	If true, celestia server will be instantiated and used as a provider for satellite visualization. See section 3.7.
timeFactor=[1..1000]	The value of this parameter will be used as a multiplier for the passing of time.
startDate	Contains the start date/time for the simulation, in the format 2017:07:13 15:47:01 CEST.
endData	Contains the end date/time for the simulation.

TABLE 2.3: Header contents explanation

Field	Description
Interface name	The interface inherited by the generic device
Method body	The actual method body of the command, i.e. **byte[runRawCommand(int cmdID,byte[data,int iAD)]]**
Execution time	The real time on the simulator machine
Simulator time	The time on the simulator when the method was executed
Input argument	The input data for the method
Output	The output data (if defined) for the method

TABLE 2.4: Command result structure

example, calling the GPS device getNMEASentence with position identifier GPGGALONG would return the following command result:

```
CommandResult{
  intfName=GPS,
  methodBody=String getNMEASentence(String inputSentence),
  internalID=2001,
  executionTime=Mon Jul 18 16:01:50 CEST 2016,
  simulatorTime=Thu Jul 13 16:23:41 CEST 2017,
  inputArgs=[String inputSentence={GPGGALONG}],
  output=[String ={\$GPGGALONG,042341.216,4414.0420950,N,09019.0301373,W
    ,1,0,0,650000.000,M,0,M,,,*XX}]}
```

LISTING 2.7: Command result object

2.4.8 Simulator commands scheduler

The simulator scheduler functions with the information parsed from the configuration file "OPS_SAT_SIMULATOR-scheduler.txt'. The typical structure of this file is shown in listing 2.8.

```
#days:hours:minutes:seconds:milliseconds|milliseconds|internalID|
    argument_template_name
00000:00:00:20:000|0000000000000020000|1001|CUSTOM
```

LISTING 2.8: Simulator scheduler entry

The information stored is time interval from start of simulator to running this command, kept in two formats, internal ID of the command and argument template. By using the internal ID and argument template effective run data is stored efficiently because the simulator will make a look up to retrieve the actual input argument content for the specific template.

Validation rules

The simulator will parse the entry for correct representation of time as well as existence of internal ID and argument template. This is the first check. The second check is implemented because there are two possible ways to represent the simulator data, namely the

1. days:hours:minutes:seconds:milliseconds
2. the milliseconds format

At the beginning a check will be made on both fields. If the time intervals are not equal, the first format representation which is different than zero (in this case priority is for above 1) will be accepted. The other format will then be update by the software to the accepted value. If both values are zero, they shall be kept as they are and the respective command will be launched immediately at start. Lastly, the third validation rule is that if the entries are not in chronological order, they will be reordered. Whenever any of the above rules is not respected for at least one entry, the whole scheduler configuration file will be rewritten in a consistent way and a backup copy will be created in the same folder.

2.5 Central node

The central node is another extension of the base class task node. Its role is handling the communication between the graphical user interfaces used for control and visualizations. If the central node is instantiated, it will create TCP listener sockets. There are two incoming ports, one for the GUI app (see subsection 2.5.1), the other for the Celestia app. By default, the port numbers for the GUI app start at 11111 and the Celestia app is restricted to port 5909. If there are multiple instances of the simulator, the GUI app server will use additional ports, up to 10 different instances (using up to port 11121).

The advantage of using a TCP architecture is that of separating the graphical user interface from the main application. The use cases of the simulator involve running on machines which don't necessarily have a display port or may be residing on a different location. By using the IP communication stack, the simulator instance can be reached across networks. Moreover, this design supports multiple clients connecting and sending commands at the same time.

2.5.1 Multi-threaded socket server

Once the server listener socket is created in its own thread, it will enter an accept loop. After that, every client connected will be managed separately, with a pair of threads, one for sending and the other for receiving. By running in parallel the send and receive routines into separate threads, the system resources can be used efficiently and there is almost no lag in responsiveness. The exchange of data is done through Object streams [7], which permits anything. For example, the simulator header is a class and is directly exchanged on the communication channel. This intuitive, symbolic approach simplifies the program architecture and ensures consistency. A more complex example can be a collection of classes (such as a list of all the possible commands which the simulator can execute).

2.5.2 Celestia data server

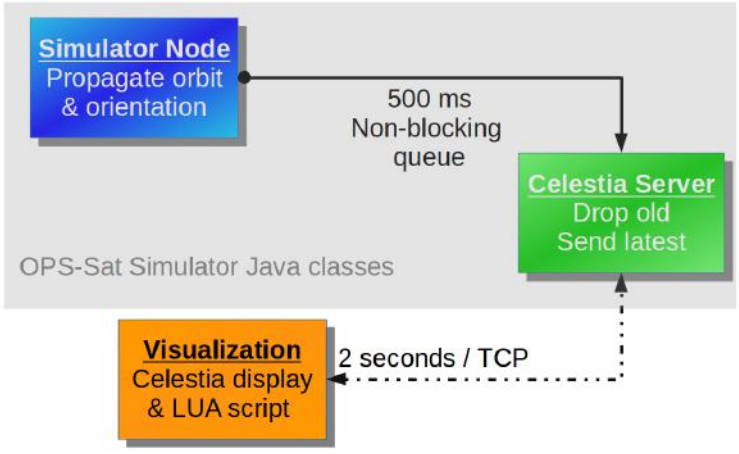


FIGURE 2.5: Communication of Celestia data from simulator node to visualization

The Celestia server expects connections from a single client. Once a client is connected there will be a handshake process. After this the server will send messages in string format and waiting acknowledge confirmation each time. On the Celestia side, a LUA script parses the received data and calls Celestia commands to display the spacecraft position and orientation. The LUA script is executed every two seconds. To use the system resources efficiently, the simulator should generate the visualization data as often as requested by the LUA script however due to the complexity of the orbit propagation it cannot calculate it on demand. Therefore the solution has been found to generate in advance visualization data and to store it in a queue. Whenever the LUA script requests a new message, the newest message in the already prepared buffer will be selected and all the others cleared. This flow is shown in figure 2.5. Also care was taken to ensure that the communication loop between the Celestia server and the LUA script can recover automatically in the event of one of the partners restarting or communication drops. This recovery is typically done by enclosing the code which instantiates the TCP sockets within a while loop and by ensuring that all exceptions are handled.

Chapter 3

Graphical user interface

The GUI client offers the possibility to remote connect to a simulator instance and retrieve running information. It is a Java GUI application, based on the Swing library. At start, it displays the target IP and port to which it is trying to connect. The default target connection is the local computer: 127.0.0.1:11111.It is possible to change with conditional arguments the target IP and port by simply appending them to the Java VM arguments. During runtime it is also possible to change the

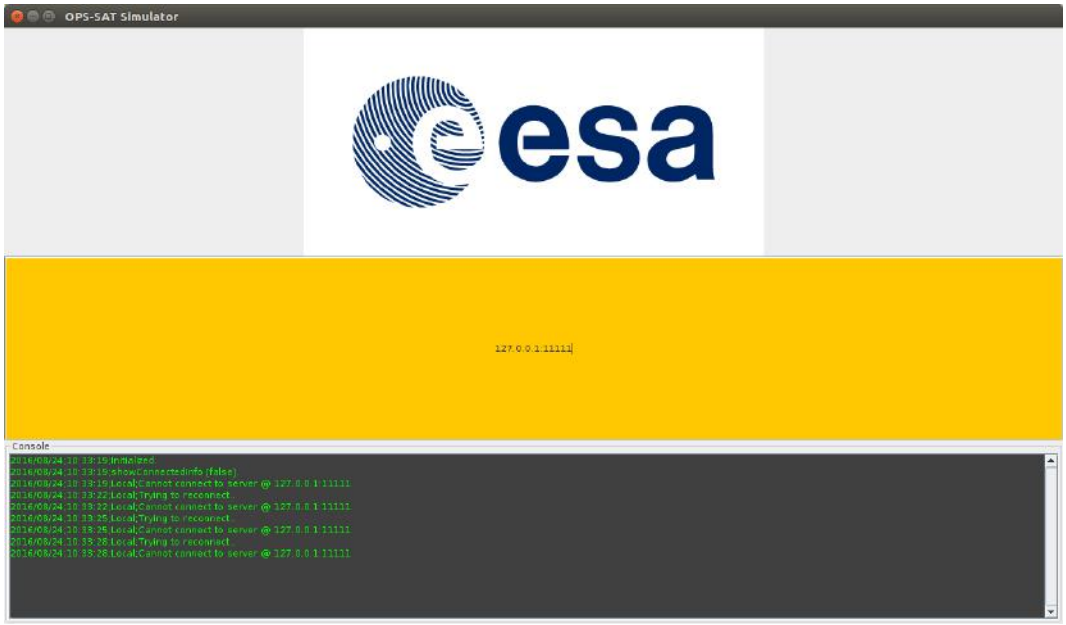


FIGURE 3.1: Simulator app establishing connection

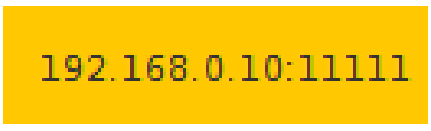


FIGURE 3.2: Changing the target connection

target by editing the yellow text field with the desired address. Once the connection is established, the window will display data received information from the server, as in figure 3.3. The UML class diagram for the GUI application is shown in figure A.3.

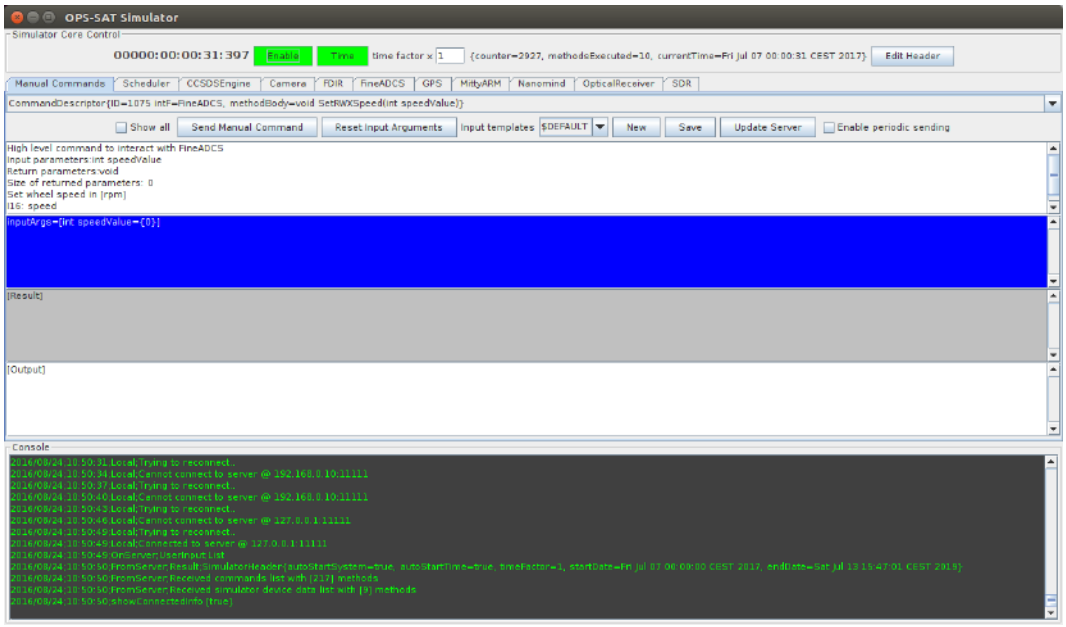


FIGURE 3.3: Simulator app connected

3.1 Simulation data

The simulator data is displayed at the top of the simulator window, in the frame called simulator core control. The information contained is related to the inner workings of the simulator. The user can see how much time has passed since the beginning, what is the active time factor, the number of computations done on the models (counter), the total number of methods executed (see 2.4.4) as well as the current time. By clicking on the buttons enable and time it is possible to stop the execution of models and/or time.

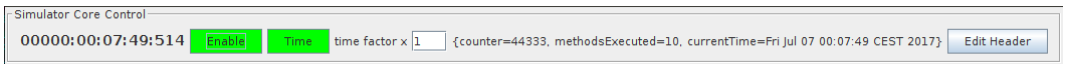


FIGURE 3.4: Simulator data

FIGURE 3.5: Simulator data with stopped time

3.2 Simulation header editor

By clicking the Edit Header button, a separate window will appear. This window allows the user to change the default behavior of the simulator. See 2.3 for the meaning of the parameters. If the input data is invalid, i.e. start or end dates are out of allowed range, the fields will be highlighted in red. Once correct data is inserted (both start and end fields have white color) the user can press "Submit to server button". This will modify the local copy of the simulator header and

then forward it to the server. At this point, the fields' color will be yellow. Once the server receives the new header, it will process it for validity, write it to the file system and then send it back to the client. The editor window fields will be green and the window will close itself after two seconds. With a successful submit to server action, the complete simulator internal state is reinitialized and reset.

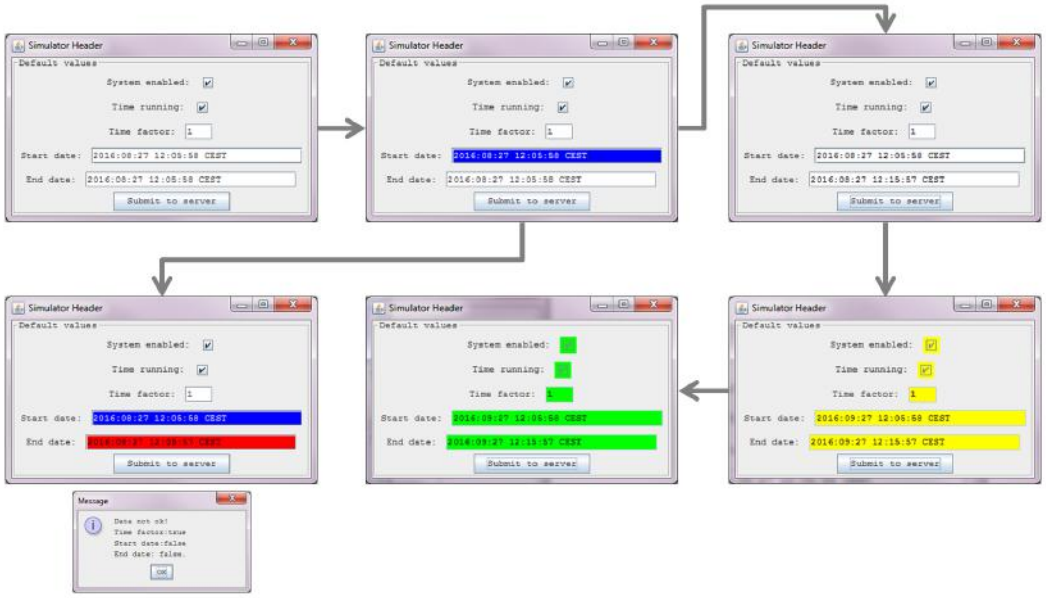


FIGURE 3.6: Simulator header editor

3.3 Manual commands

The manual commands tab offers the possibility to run specific commands from the entire list supported by the simulator. To select a command, the combo box is used. Once a command is selected (this is done automatically at start up for the first in the list command), the information displayed in the text areas bellow will be updated as shown in figure 3.7.

1. Description of method: includes information about input and output parameters. The background color is always white.

2. Input arguments: displays the input arguments in an easy readable form which allows changing values. The background color is blue when input is expected, red when the input data is not valid.

3. Result: will show the actual output of the executed command. The background color is gray when a new command is selected, yellow when a command has been sent to the server, green when a command has been executed successfully, red when a command has failed execution on the server.

4. Output: will show a detailed information about the run of the command, as described in 2.4. The background color is always white.

To select between different commands, the combo box depicted in figure 3.8 is used. To allow easy identification of the command and its role, the internal ID, peripheral device to which it belongs to and method body are shown. The number of

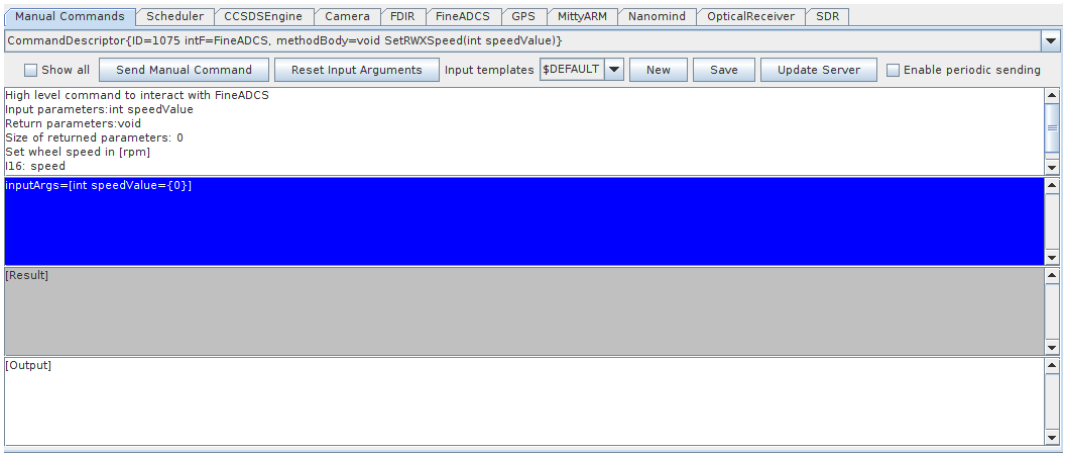


FIGURE 3.7: Simulator manual commands

simulator commands is around 250 which makes using a filter a necessity. The filter is implemented as a text file (see example listing 3.1) which contains the "visible" commands in the drop down list. If the check box "Show all" is ticked, the commands list will be entirely populated.

```
#Filter for FineADCS
1168
1170
1174
1178
```

LISTING 3.1: Filter file example

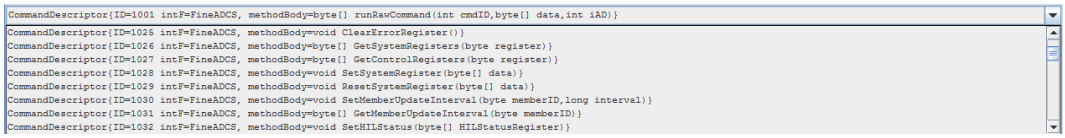


FIGURE 3.8: Simulator command selection combo box

By pressing "Send Manual Command" button the simulator will parse the content of the input arguments text area. The syntax it is expecting is based on the method body, namely the number of input arguments and their type. If the content cannot be parsed, the color will change to red and the output text area will contain the reason, as per figure 3.9. If the syntax of the input argument is correct, the command will be sent to the server. The color of the input area will become yellow, indicating the request is in progress. When the result is received the input text area will become blue again, while the result area will be either green or red, depending on whether the command execution succeeded. Successful command means that the input arguments passed additional logic validations on simulator side and no exceptions were encountered during the execution. This flowchart is shown in figure 3.10 .

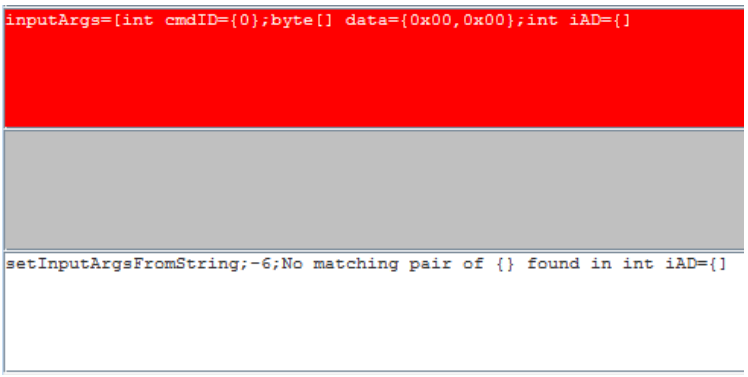


FIGURE 3.9: Parse error, wrong syntax

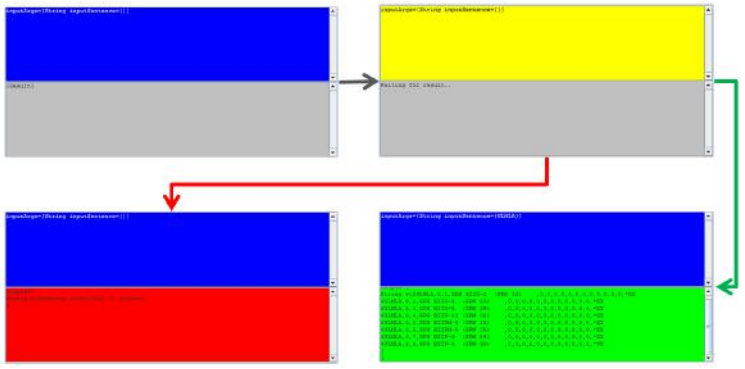


FIGURE 3.10: Flowchart showing possible interactions for a manual command

3.4 Simulator scheduler view

The tab scheduler shows a progress list of all the configured scheduled commands. The internal ID, execution time, argument template name and input arguments are displayed.

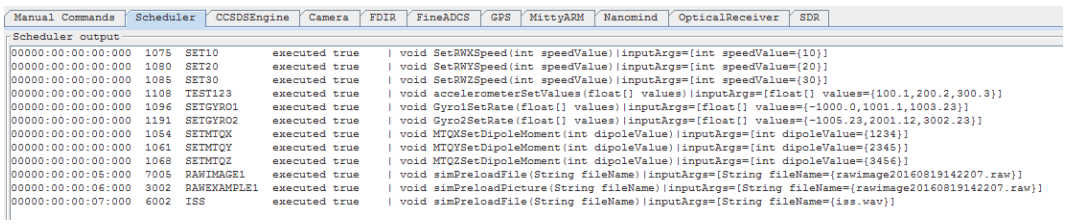


FIGURE 3.11: Simulator scheduler progress view

3.5 Inspection of simulator parameters

The remaining tabs in the graphical user interface contain variable representations of the devices running models(figure 3.12). This can be used for monitoring at a glance key simulation parameters. An example of the typical data listing is shown in 3.2

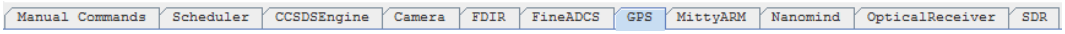


FIGURE 3.12: Simulator tabs containing device views

```
double latitude={81.64848}

double longitude={102.18936}

String altitude={663793.1049071697}

double groundStationESOC_Elevation={-18.23559}

double groundStationESOC_Azimuth={12.69174}

String satsInView={[11] satellites
GPSSatInView{name=GPS BIIR-2  (PRN 13)    , distance=20747.77[km],
    azimuth=108.65, elevation=49.11}
GPSSatInView{name=GPS BIIR-4  (PRN 20)    , distance=21079.06[km],
    azimuth=150.96, elevation=44.08}
GPSSatInView{name=GPS BIIR-5  (PRN 28)    , distance=24322.79[km],
    azimuth=61.84, elevation=14.03}
GPSSatInView{name=GPS BIIR-7  (PRN 18)    , distance=21951.30[km],
    azimuth=233.17, elevation=40.14}
GPSSatInView{name=GPS BIIR-9  (PRN 21)    , distance=22120.02[km],
    azimuth=200.89, elevation=37.72}
GPSSatInView{name=GPS BIIRM-4 (PRN 15)    , distance=20931.30[km],
    azimuth=159.39, elevation=45.20}
GPSSatInView{name=GPS BIIRM-6 (PRN 07)    , distance=23640.39[km],
    azimuth=8.10, elevation=16.62}
GPSSatInView{name=GPS BIIF-4  (PRN 27)    , distance=21548.50[km],
    azimuth=295.40, elevation=39.22}
GPSSatInView{name=GPS BIIF-5  (PRN 30)    , distance=22067.43[km],
    azimuth=34.07, elevation=32.93}
GPSSatInView{name=GPS BIIF-10  (PRN 08)   , distance=22470.19[km],
    azimuth=342.19, elevation=29.01}
GPSSatInView{name=GPS BIIF-11  (PRN 10)   , distance=24085.19[km],
    azimuth=259.38, elevation=13.46}}
```

LISTING 3.2: GPS device models data view

3.6 Console

The simulator console provides a list of messages which aid the user into understanding the program function and interaction between server and client. It automatically scrolls down upon receiving new data and will keep a record of oldest 500 messages. The types of messages displayed by the console can be found in table 3.1.

3.7 Celestia

The Celestia program is a free space simulation that allows a user to explore the universe in 3D [8]. An existing framework developed by ESA was used to display the OPS-SAT simulator state. This interface is a scripted communication via TCP channel between a server which provides the information and a LUA script which

Source	Type	Comment
Server	OnServer	A specific event has occurred on the server. Frequently this is an echo to a command that has been initiated by the client or the Nanosat MO framework parent class.
Local	Local	The client has successfully established connection to the simulator server.
Server	FromServer	Data has been received on the server side. This can be a simulation header, a commands list or a command result.

TABLE 3.1: Console output possible contents

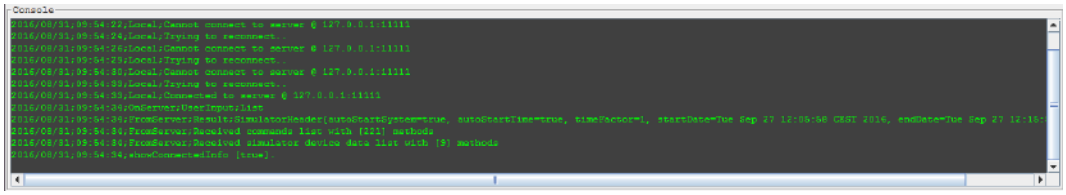


FIGURE 3.13: Simulator console report

is running on the background of the Celestia application. In addition to the display of OPS-SAT position in the orbit, some other useful information for planning and control is provided:

1. next ascending crossing
2. descending node crossing
3. signal acquisition to the ground station
4. signal loss to the ground station

The figure 3.15 shows multiple windows opened that are possible thanks to Celestia's scripting mechanism. At the instant displayed in the picture, the OPS-SAT is in view with the ground station and it will keep the view for another few minutes. The planning and control information related to node crossings and signal windows is calculated in the simulator node core on demand, as soon as the current propagated state exceeds the respective monitored event.

3.8 WebEUD

The Ground MO Web-EUD is a prototype project that provides a web-based client for CCSDS MO services. This is achieved by integrating Web-EUD together with the Ground MO Adapter [3]. This will allow OPS-SAT experimenters to monitor and control their "app" using a normal web client. Figures 3.16 and 3.17 show applications of the Web-EUD project in terms of telemetry view. The provider for the parameters is a GPS demonstration application which runs an instance of the simulator. It is also possible to execute actions from the Web-EUD client, actions which go through the MO framework.

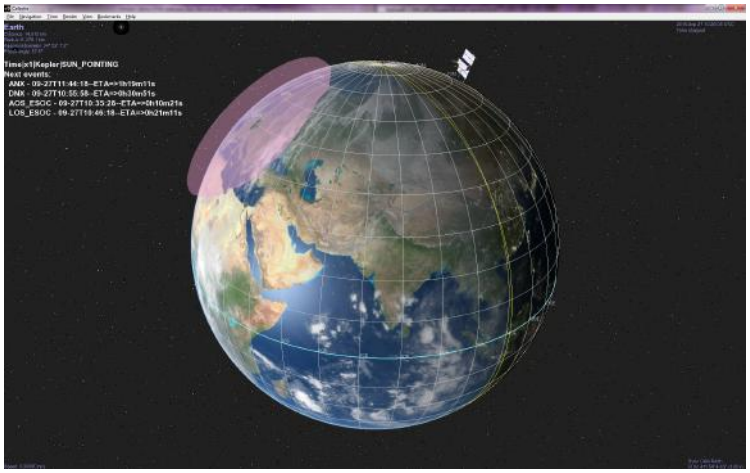


FIGURE 3.14: Celestia visualization

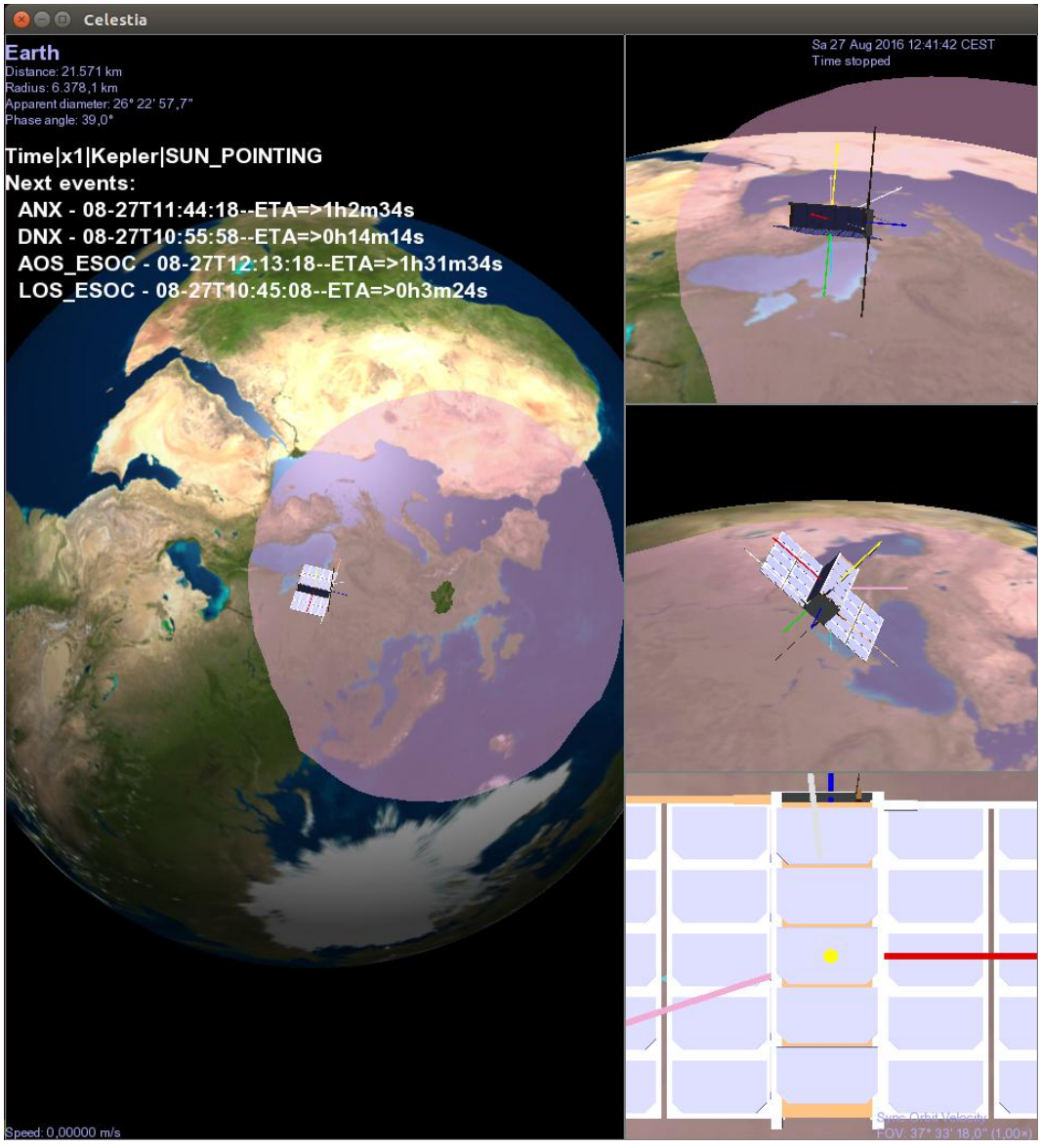


FIGURE 3.15: Extended view

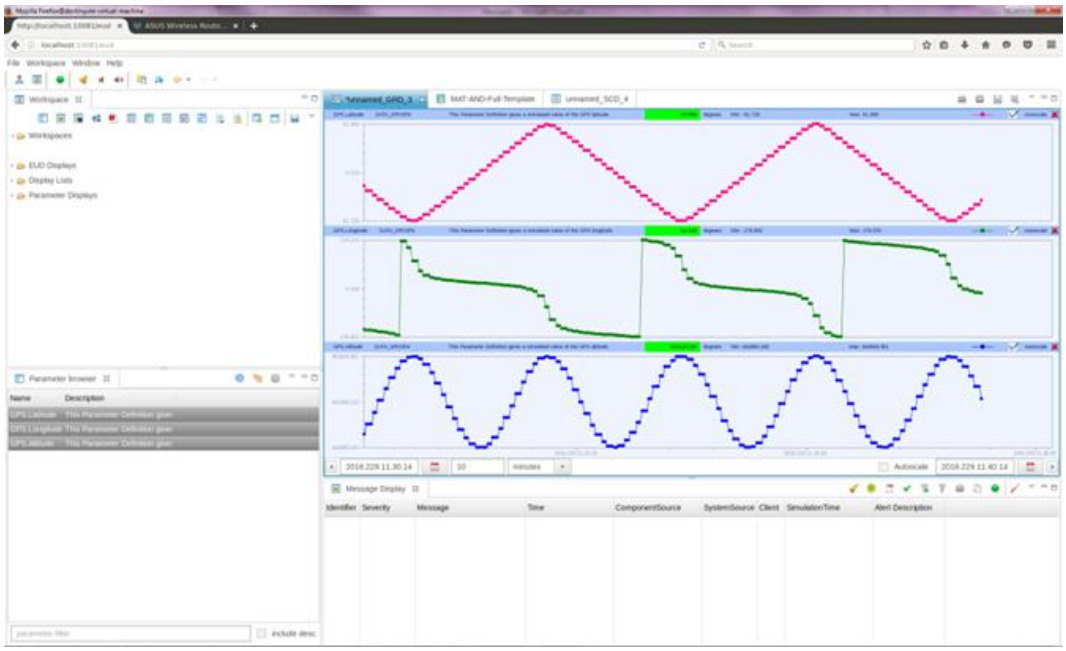


FIGURE 3.16: Browser image of Web-EUD

unnamed_GRD_3 | MAT-AND-Full-Template | unnamed_SCD_4

Parameter	Description	Value	Unit	Validity	OOL	Sample Time
GPS.Altitude	This Parameter Definition	643148.935	degrees	VALID	NOT_APPLICA	2016.229.11.33.48.811
GPS.Longitude	This Parameter Definition	-170.256	degrees	VALID	NOT_APPLICA	2016.229.11.33.48.809
GPS.Latitude	This Parameter Definition	6.666	degrees	VALID	NOT_APPLICA	2016.229.11.33.48.787

FIGURE 3.17: MAT display of parameters using Web-EUD

Chapter 4

Peripheral devices

4.1 FineADCS

The FineADCS peripheral device ensures multiple functions related to the attitude of the spacecraft. It is the first ever designed to be COTS for nano satellites. It communicates to the experimenter's platform over I2C. The simulator reproduces command data buffers down to byte level. Work was done to write based on the ICD document all the commands in the simulator. The complete list can be found in appendix A, table A.3. Some of the functions and/or data telemetry provided of the ADCS is reproduced by physical simulations whereas other parts are represented as static variables. The main modules of the fine attitude determination and control system concern with the following:

1. Magnetometer containing a model of the Earth's magnetic field
2. Attitude providing quaternions of the spacecraft position in an inertial frame
3. Sun sensor which returns the relative position of the spacecraft with regard to the sun
4. Actuators namely the raw values of magnetorquers and reaction wheels

4.1.1 Magnetometer

The magnetometer uses the from the orekit library the GeoMagneticField classes to instantiate an implementation if the IGRF magnetic field, based on [9]. The model requires coefficients files which are published periodically and are valid for a period of five years. In order to validate the simulator results, an online test program provided by the developers of the coefficients file was used(see [10]). The results are shown in table 4.1. Since the magnetic field model class provides the magnetic field vector in an inertial frame (north, east and vertical components), it is required to apply a rotation to obtain the real sensor values, like shown in figure 4.1.

4.1.2 Attitude

The attitude of the satellite is obtained in forms of quaternions. There are different attitude control modes:

1. B-DOT or de-spin
2. Sun pointing
3. Target tracking where the satellite points to a specific point on Earth given by latitude and longitude

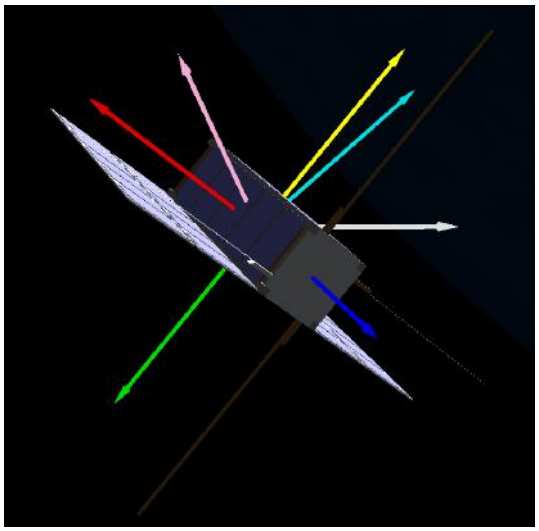


FIGURE 4.1: Red, green, blue body frame. Cyan, pink, white inertial frame. Magnetometer rotation

Test	Parameter	Units	Web calculator	OPS-SAT Sim	Diff
Random test @ latitude 10.62875 N, longitude	North component	[nT]	22238.8	22245.82	-7.02
149.57669 W, elevation 643585.056657 m	East component	[nT]	3720	3719.1	0.9
GPS , date 2016.08.27	Vertical component	[nT]	9461.1	9471.36	-10.26

TABLE 4.1: Magnetometer comparison results

4. Local orbital frame orientation with spinning

To compute the respective quaternions of each of the above modes, a collection of classes AttitudeProvider is used. This is an AttitudeSequence which is reset each time a new attitude is commanded.

4.1.3 Sun sensor

The sun sensors are important measurements which are used in the processing loop for determining the s/c attitude. Depending on the orientation and whether the spacecraft is in eclipse, it will have a different value. The simulator obtains the sun vector by intersecting front body axis of the spacecraft with the sun. Afterwards this vector is normalized to give the direction.

4.1.4 Actuators

The actuators are represented by the magnetorquers and the reaction wheel. They are not included in the simulator models and only have static values which can be

modified by hardware in the loop test commands. For example these are reaction wheels, accelerometer rates, gyros and magnetorquers. The typical use scenario is to set values via the scheduler system.

4.1.5 Orbit information

The FineADCS devices supports update of the position and epoch of the s/c from TLEs [11]. This is a very convenient method because it is asynchronous and Celestrak will always provided updates. The simulator uses the TLE classes from the orekit library to validate new TLEs and propagate the spacecraft position based on them. This will be part of the mission-maintenance aspect. Listing 4.1 shows a successful parse and convert command of a two line element set into a byte array, in the format expected by the on board device.

```
CommandResult{
  intfName=FineADCS,
  methodBody=byte[] simGetOrbitTLEBytesFromString(String tleLine1,String
    tleLine2),
  internalID=1182,
  executionTime=Tue Sep 20 09:24:42 CEST 2016,
  simulatorTime=Sat Aug 27 12:06:35 CEST 2016,
  inputArgs=[
    String tleLine1={1 40055U 14034C   16222.22560006  .00000137  00000-0
      28532-4 0  9995};
    String tleLine2={2 40055  98.1939 309.9879 0013817 343.0399  17.0346
    14.74468021113576}],
  output=[
    byte[] ={(0)0x31,(1)0x20,(2)0x34,(3)0x30,(4)0x30,(5)0x35,(6)0x35,(7)0
    x55,(8)0x20,(9)0x31,(10)0x34,(11)0x30,(12)0x33,(13)0x34,(14)0x43,(15)0
    x20,(16)0x20,(17)0x20,(18)0x31,(19)0x36,(20)0x32,(21)0x32,(22)0x32
    ,(23)0x2E,(24)0x32,(25)0x32,(26)0x35,(27)0x36,(28)0x30,(29)0x30,(30)0
    x30,(31)0x36,(32)0x20,(33)0x20,(34)0x2E,(35)0x30,(36)0x30,(37)0x30
    ,(38)0x30,(39)0x30,(40)0x31,(41)0x33,(42)0x37,(43)0x20,(44)0x20,(45)0
    x30,(46)0x30,(47)0x30,(48)0x30,(49)0x30,(50)0x2D,(51)0x30,(52)0x20
    ,(53)0x20,(54)0x32,(55)0x38,(56)0x35,(57)0x33,(58)0x32,(59)0x2D,(60)0
    x34,(61)0x20,(62)0x30,(63)0x20,(64)0x20,(65)0x39,(66)0x39,(67)0x39
    ,(68)0x35,(69)0x0A,(70)0x32,(71)0x20,(72)0x34,(73)0x30,(74)0x30,(75)0
    x35,(76)0x35,(77)0x20,(78)0x20,(79)0x39,(80)0x38,(81)0x2E,(82)0x31
    ,(83)0x39,(84)0x33,(85)0x39,(86)0x20,(87)0x33,(88)0x30,(89)0x39,(90)0
    x2E,(91)0x39,(92)0x38,(93)0x37,(94)0x39,(95)0x20,(96)0x30,(97)0x30
    ,(98)0x31,(99)0x33,(100)0x38,(101)0x31,(102)0x37,(103)0x20,(104)0x33
    ,(105)0x34,(106)0x33,(107)0x2E,(108)0x30,(109)0x33,(110)0x39,(111)0x39
    ,(112)0x20,(113)0x20,(114)0x31,(115)0x37,(116)0x2E,(117)0x30,(118)0x33
    ,(119)0x34,(120)0x36,(121)0x20,(122)0x31,(123)0x34,(124)0x2E,(125)0x37
    ,(126)0x34,(127)0x34,(128)0x36,(129)0x38,(130)0x30,(131)0x32,(132)0x31
    ,(133)0x31,(134)0x31,(135)0x33,(136)0x35,(137)0x37,(138)0x36,(139)0x0A
    }
]}
```

LISTING 4.1: TLE validation and conversion into byte array

The following listing 4.2 shows a malformed TLE (the second line vehicle number does not match the first line). Here the command result contains a string representation of the Orekit exception (failed check-sum).

```
CommandResult{
  intfName=FineADCS,
  methodBody=byte[] simGetOrbitTLEBytesFromString(String tleLine1,String
    tleLine2),
  internalID=1182,
  executionTime=Tue Sep 20 09:50:29 CEST 2016,
```

```
  simulatorTime=Sat Aug 27 12:06:18 CEST 2016,
  inputArgs=[
    String tleLine1={1 40055U 14034C   16222.22560006  .00000137  00000-0
      28532-4 0  9995};
    String tleLine2={2 40056  98.1939 309.9879 0013817 343.0399  17.0346
    14.74468021113576}],
  output=[
    String ={java.lang.Exception: org.orekit.errors.OrekitException:
    wrong checksum of TLE line 2, expected 6 but got 7 (2 40056  98.1939
    309.9879 0013817 343.0399  17.0346 14.74468021113576)}
]}
```

LISTING 4.2: TLE invalid arguments with rejection

4.1.6 Quaternion server

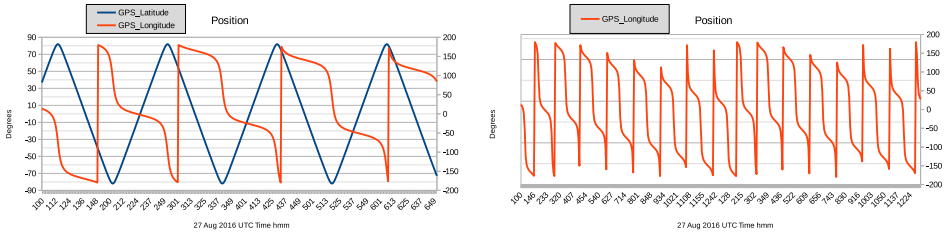
To allow direct data input from a physical sensor, a simple TCP server that receives lines of data has been implemented. The initialization of this requires running a special simulator command shown in listing 4.3. The command takes the keyword "quaternionServer' and port number '10500'. From the other end, of the real sensor, it is enough to connect at the IP address of the simulator and send each of the quaternion values on a line.

```
CommandResult{
  intfName=FineADCS,
  methodBody=void simRunDeviceCommand(String data),
  internalID=1204,
  executionTime=Tue Sep 20 12:08:50 CEST 2016,
  simulatorTime=Sat Aug 27 12:06:17 CEST 2016,
  inputArgs=[String data={quaternionServer:10500}],
  output=[null]
}
```

LISTING 4.3: Quaternion server command initialization

Test	Parameter	Units	JSatTrak	OPS-SAT Sim	Difference
Initial conditions	Latitude	[deg]	0.092067842	0.09206146	6.38259E-06
	Longitude	[deg]	85.15249256	85.1524885	4.06236E-06
	Altitude	[m]	642863.7548	642863.0548	0.700007595

TABLE 4.2: GPS position comparison results



(A) 3 orbits (B) 24 hours

FIGURE 4.2: GPS position simulation

4.2 GPS

The GPS unit flying on the OPS-SAT mission belongs to the Novatel OEM6 family, with firmware reference manual [12]. It provides support for a number of NMEA sentences which are presented in table A.5. For the device, the list of executable commands can be found in table A.4.

4.2.1 Validation of results

To ensure that data produced by the simulator is accurate, it has been cross-checked with a free satellite tracking program called JSatTrak. The table 4.2 shows the differences in results of the different propagators used for given duration of time.In figure 4.2a the orbit data over a period of almost 7 hours or 3 orbits is reproduced, which is indicative of the circular, sun-synchronous, dawn-dusk orbit of OPS-SAT (refer to table A.2). As can be seen in figure 4.2b, the polar-crossings occur at different longitudes, with an overall period around the globe equal to the sidereal day.

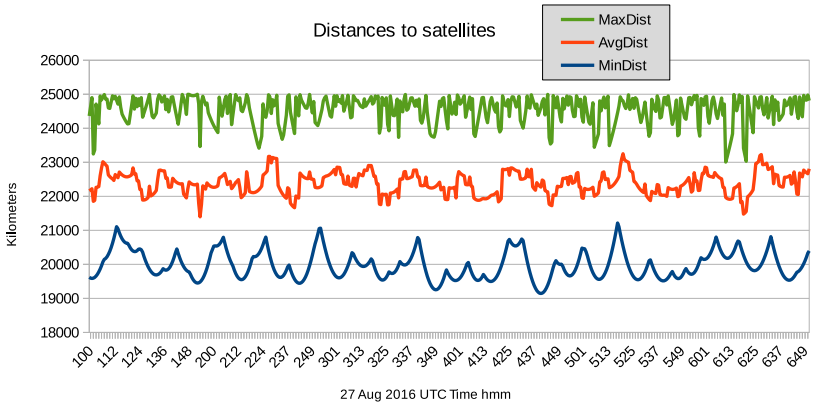


FIGURE 4.3: GPS constellation distance to satellites

4.2.2 Ground position simulation

The simulator uses Orekit to calculate the ground position of the satellite. From object oriented point of view, this means intersecting the satellite position vector with the Earth body shape and obtaining the coordinates of that intersection. Next the frame of reference is converted to a geodetic one, which contains latitude, longitude and altitude. Given the Earth geoid shape, the altitude value will be smaller at Equator latitudes and bigger at the poles.

4.2.3 Contact windows

The simulator calculates several orbital parameters such as:

1. ascending node crossing - passing the Equator from south to north
2. descending node crossing - passing the Equator from north to south
3. ground station acquisition of signal
4. ground station loss of signal

To calculate the next contact windows, the simulator propagates in the future the current position of the spacecraft. There is a specified time step for each iteration and a maximum number of steps to be executed before the algorithm will stop.

4.2.4 GPS satellites in view

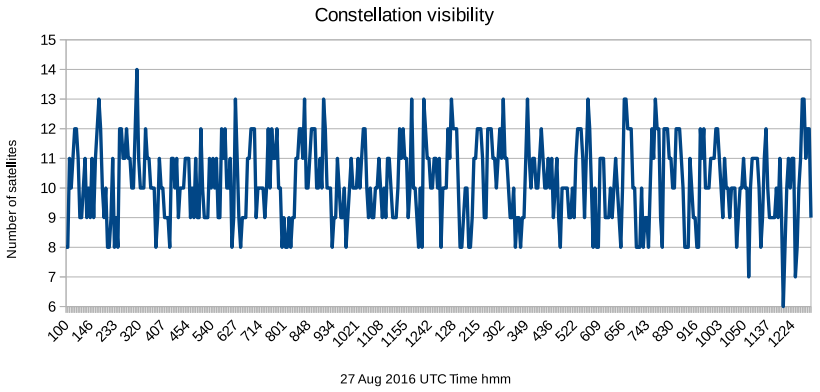


FIGURE 4.4: GPS constellation visibility

The GPS receiver provides information about all the satellites in view. This may be used for checking that all receiver channels are working properly. In the scope of the simulator, only the Navstar constellation is simulated. In figure 4.4 the ouput number of satellites in view is shown, this number ranging between 7 to 14, with an average value of 10-11. As expected, these results indicate the simulated GPS constellation is in good health and provides worldwide coverage.
To simulate the GPS constellation TLEs are downloaded from Celestrak at the initialization of the simulator. If no Internet connection is available the last saved data will be used. Next distances and elevation angles are calculated for each of the GPS satellites. If the distance is below a threshold and the elevation is positive, the satellite is considered into view. The distance-based approach is a little naive because it

does not take into account the Doppler shift due to relative velocities of the spacecrafts (however for the purpose of the simulation it is realistic enough). This limitation can be observed in figure 4.3 where the distance to the farthest satellite seems to plateau at the threshold value 25000 km. The minimum distance however has a more distinctive trend of convex curves which correspond to closest approaches of different satellites. With regard to the elevation statistics, represented in figure 4.5, the following conclusion can be drawn: there are fewer GPS satellites passing near zenith-elevations so the trend is dominated by these solitary contacts. The low horizon elevations have more satellites and the curve tends to be flatter.

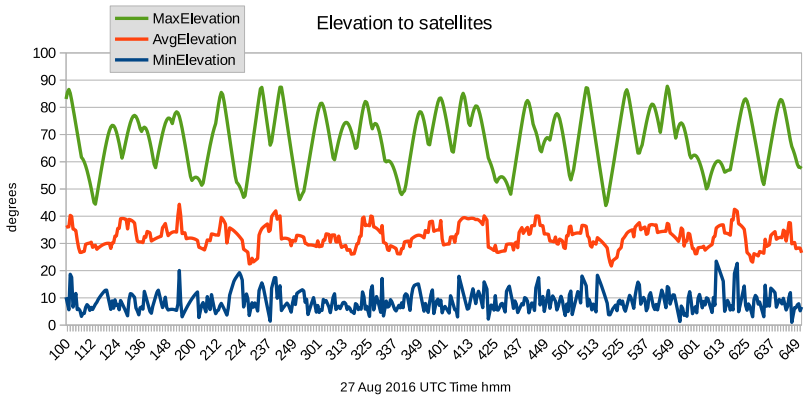


FIGURE 4.5: GPS constellation elevation to satellites

Balancing the computation effort

Because there are 33 operational satellites at the moment calculating the in view condition which involves distance and elevation angle is executed in a thread separated from the main propagation of OPS-SAT found in the run sequence (see task node organization in 2.2). The program checks on each run cycle whether the constellation thread is finished and if so, it starts another one. The result is that satellites in view information is calculated less frequently than the main propagation, however for simulation purposes it holds the advantage of not slowing down the main application.

4.3 Camera

The camera used on the OPS-SAT mission is provided by Hyperion Technologies, model ST200 Star Tracker [13]. It is connected via TTL UART interface. To the on-board processing platform (MityArm), the camera is a serial communication device for control commands and a file system device for retrieving images. There are two available methods supported by the camera device, in table A.6. The camera sensor is masked with a Bayer pattern for color information. For each obtainable image, the maximum resolution is 2048 by 1944 pixels of 16 bits. This amounts to a total of 7962624 bytes. The ST200 camera was available in the AGSA lab so there is raw material available. The original and converted images are shown in figures 4.6 and 4.7. For returning image data, the method takePicture requires a width and a height in pixels. By using the scheduler, a camera raw file can be preloaded and will be available for all subsequent takePicture commands.

```
byte[] data = instrumentsSimulator.getpCamera().takePicture((int)
    resolution.getWidth().getValue(), (int) resolution.getHeight().
    getValue());
```

LISTING 4.4: Obtaining picture information from camera device

If the maximum resolution image is taken with input arguments the operating buffer will jump the whole image size and return to zero. However if the resolution parameters are one quarter of the maximum image (1024 by 972), the operating index will jump to 1990656 (one fourth) , 3981312 (half) , 5971968 (three fourths) respectively. An example of the first retrieve operation is shown in listing 4.5.

```
CommandResult{intfName=Camera, methodBody=byte[] takePicture(int width,
    int height),internalID=3001, executionTime=Fri Sep 16 14:32:20 CEST
    2016, simulatorTime=Sat Aug 27 12:10:56 CEST 2016,inputArgs=[int width
    ={1024};int height={972}], output=[
byte[] ={(0)0xD0,(1)0x0B,(2)0x90,(3)0x0D,(4)0x50,(5)0x0C,(6)0xE0
    ,..,(1021)0x0D,(1022)0x70,(1023)0x10,(1024)0x10,+ [1989631] more ,
    total [1990656] bytes.}
```

LISTING 4.5: Take picture command example

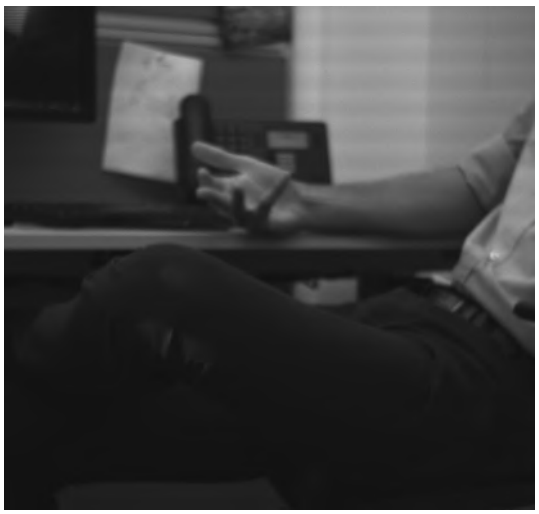


FIGURE 4.6: Raw image with Bayer filter

FIGURE 4.7: Debayer image with color information retrieved

4.3.1 Camera Script

It is possible to enable the simulator run a dedicated script when the command takePicture is invoked. This is done by using a special simulator command that requires the keyword "cameraScript" and absolute path of the required script, as shown in listing 4.6.

```
CommandResult{
  intfName=FineADCS,
  methodBody=void simRunDeviceCommand(String data),
  internalID=1204, executionTime=Tue Sep 20 12:17:31 CEST 2016,
  simulatorTime=Sat Aug 27 12:14:58 CEST 2016,
  inputArgs=[String data={cameraScript:/home/pi/repos/RPI_GPS_Tracker/
    helper/takePicture2.sh}],
  output=[null]
}
```

LISTING 4.6: TLE invalid arguments with rejection

4.4 SDR

The transceiver on board will be capable of broadband spectrum sweeps between 300MHz to 3.8GHz. It contains two antennas:

1. deployable UHF antenna delivered by ISIS - Innovative Solutions in Space company [14]
2. micro strip dipole antenna in the L-Band

The experimenter's platform is connected via parallel IO to the SDR board. This will allow complete spectrum data collection at different data rates when the on board FPGA is used. Different parameters of the SDR can be set, such as gain, frequency, sample rate. The output of the SDR is a stream of I/Q samples (in-phase and quadrature). These can be written into a .wav audio file for standardized storage. The OPS-SAT simulator contains a .wav file reader based on the implementation found on [15]. The simulator offers commands for preloading .wav files and then reading a number of samples as shown in table A.8. Once a preload command is issued, the simulator will output the .wav file information and a sample buffer will be available.

```
File: /home/dev-tinyvm/ops-sat-simulator-resources/iss..wav
Channels: 2, Frames: 3330048
IO State: READING
Sample Rate: 55555, Block Align: 4
Valid Bits: 16, Bytes per sample: 2
```

LISTING 4.7: Output of wav file reader

```
String operatingBuffer={double[] {-8.544921875E
    -4,0.00103759765625,3.0517578125E-5,7.9345703125E-4,-1.220703125E
    -4,-9.1552734375E-4,-0.00177001953125,-9.1552734375E-5,-8.85009765625E
    -4,8.544921875E-4,-6.103515625E-4, and [6660085] more , total
    [6660096] doubles.}}
```

LISTING 4.8: Samples operating buffer

The listing 4.9 shows the result of polling data from the simulator for 10 samples. The information is stored as pairs of I and Q samples, so ,as a general rule, for N samples in the request, there will be 2*N double values returned.

```
CommandResult{intfName=SDR, methodBody=double[] readFromBuffer(int
    numberSamples),internalID=6003, executionTime=Fri Sep 16 14:29:04 CEST
     2016, simulatorTime=Sat Aug 27 12:07:39 CEST 2016,inputArgs=[int
    numberSamples={10}], output=[
double[] ={-8.544921875E-4,0.00103759765625,3.0517578125E-5,7.9345703125E
    -4,-1.220703125E-4,-9.1552734375E-4,-0.00177001953125,-9.1552734375E
    -5,-8.85009765625E-4,8.544921875E-4,-6.103515625E
    -4,0.001129150390625,-8.85009765625E-4,7.62939453125E-4,-3.96728515625
    E-4,-2.74658203125E-4,-2.13623046875E-4,7.9345703125E-4,-3.0517578125E
    -5,0.001129150390625}
]}
```

LISTING 4.9: Retrieve samples command example

4.5 Optical Receiver

The optical receiver is a photon detector that will be used for data up link to the satellite. It will be used with the laser ground station in TU Graz. It offers the possibility of secure information exchange because it relies on a single converged laser beam. Similar to the software defined radio, the simulation model relies on a data buffer. This can be set directly with a command or from a file. In addition there is a success rate parameter which represents the probability that the message will be read from the buffer without errors. It is the likelihood of a bit flip for each of the bits in the optical data stream.

```
String operatingBuffer={byte[] {0xAA,0xAA,0xAA,0xAA,0xAA,0x3E}}
int operatingBufferIndex={0}
int successRate={10000}
```

LISTING 4.10: Optical receiver model parameters

The successRate parameter is confined between 5000 and 10000. This corresponds to 50.00% and 100.00%. The maximum value represents no errors during transmission while the minimum represents the worst case where one in two bits is flipped. The listing 4.11 shows the same command with different success rates (second is 80%).

```
100%
CommandResult{intfName=OpticalReceiver, methodBody=byte[]
    readFromMessageBuffer(int bytesNo),internalID=7004, executionTime=Fri
    Sep 16 14:58:20 CEST 2016, simulatorTime=Sat Aug 27 12:36:55 CEST
    2016,inputArgs=[int bytesNo={6}], output=[
byte[] ={(0)0xAA,(1)0xAA,(2)0xAA,(3)0xAA,(4)0xAA,(5)0x3E}
]}
80%
CommandResult{intfName=OpticalReceiver, methodBody=byte[]
    readFromMessageBuffer(int bytesNo),internalID=7004, executionTime=Fri
    Sep 16 15:07:10 CEST 2016, simulatorTime=Sat Aug 27 12:07:06 CEST
    2016,inputArgs=[int bytesNo={6}], output=[
byte[] ={(0)0xEB,(1)0xAE,(2)0x98,(3)0x6A,(4)0xAE,(5)0xBE}
]}
```

LISTING 4.11: Optical receiver command results

Chapter 5

Results

5.1 Platform virtual machine test bed

One of the goals during the simulator development and testing was to package it into a easily deployable virtual machine container. A virtual machine in the AGSA lab server was used for this. The machine specifications are found in table 5.1. The VM will start automatically the development stack consisting of:

1. Demo Application using MO Framework and simulator
2. Simulator control panel connected locally to simulator
3. Celestia application connected locally to simulator
4. WebEUD server (see section 3.8) connected locally to MO Framework
5. Logging monitor window displaying the simulator node (see section 2.4)

The above applications are started in a terminal multiplexer (with the tmux program), each in one window. This allows attaching to the same session from multiple computers via different connections (SSH), while keeping the same content displayed.

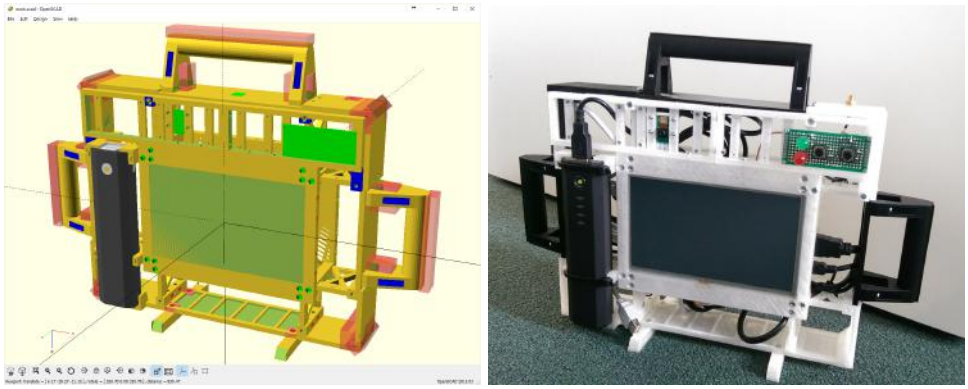
5.1.1 Demo Application

The demo application is a NanoSat MO Framework provider which creates an instance of the simulator. By default, all the development options are enabled on the simulator (simulator control panel TCP server, orekit library, Celestia server). The MO provider is configured to use one of the following transport back-ends:

1. rmi:// Remote method invocation
2. tcp:// Transport control protocol
3. malhttp:// A protocol based on string stream libraries

Characteristic	Value
Operating system	Ubuntu 16
Architecture	x64
Memory	4GB

TABLE 5.1: Test bed virtual machine configuration



(A) 3D Model front view (B) Printed object from front

FIGURE 5.1: Model and concrete object view

5.1.2 Simulator control panel

This application is described in chapter 3. The connection setting is chosen as the localhost because it is residing on the same machine.

5.1.3 Celestia application

The Celestia application as described in chapter 3, section 3.7, connects through the LUA script to the simulator instance and receives string messages with the orbital position of the spacecraft as well as some additional information about currently running propagator and next orbital events.

5.1.4 WebEUD server

This application connects to the MO framework provider and acts as a consumer. It is able to retrieve the existing parameter definition list, invoke actions, display action progress messages, all within a modern Eclipse-based user interface.

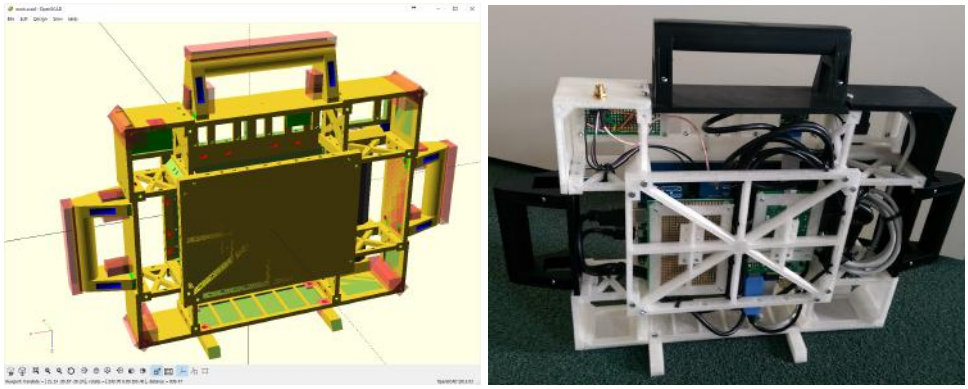
5.1.5 Logging monitor

This window is a tail command which monitors the simulator node output. This is a dedicated file sink for the simulator which contains, depending on the verbosity of the logging, internally logged messages in the simulator node. Such a file is partially listed in A.4 (the start up of the application).

5.2 Platform Raspberry PI Target

The portability of Java and the NanoSat framework can be easily demonstrated by deployment to various target machines. The Raspberry PI 3 is an ARM architecture system on a board with comparable characteristics to the MityARM and makes a ideal candidate as a test platform. A computer/tablet device was constructed around the Raspberry PI to provide, among others, a realistic satellite platform:

- 3D printed structure using OpenSCAD design software, shown in figures 5.1 and 5.2



(A) 3D Model rear view (B) Printed object from rear

FIGURE 5.2: Model and concrete object view from rear

- Raspberry PI 3 system on a board system
- GPS receiver using the MTK3339 chipset [16]
- 9 degree of freedom orientation sensor BNO055 [17]
- temperature and pressure sensor BMP180 [18]
- HD camera [19]
- 10000mAh battery bank

At start up a demonstration application is launched which instantiates the normal simulator instance, however using the sensors as providers. One of these is the IMU: the quaternions are read and fed into the telemetry, allowing the consumer applications or the Celestia program to replicate the movement of the physical device. In addition to this, when the takePicture command is invoked, the device will take a picture with the camera and save it to the file system. To enable all these additional simulation modules, the scheduler system is used to preload files, configure target script for picture taking, start listener server for quaternions, etc. The output of the simulator scheduler, once all the commands have executed, is shown in listing 5.2. The standard Raspberry PI camera takes color pictures with

FIGURE 5.3: Raspberry PI camera

resolution 2592 by 1944 pixels which are saved to the file system as jpg images. The program used is the standard raspistill.

```
filename=~/Pictures/$(date +"%Y%m%d_%H%M%S").jpg
raspistill -f -t 2000 -o $filename
```

LISTING 5.1: Raspberry PI take picture command

```
00000:00:00:00:000  1075  SET10        executed true    | void
    SetRWXSpeed(int speedValue)|inputArgs=[int speedValue={10}]
00000:00:00:00:000  1080  SET20        executed true    | void
    SetRWYSpeed(int speedValue)|inputArgs=[int speedValue={20}]
00000:00:00:00:000  1085  SET30        executed true    | void
    SetRWZSpeed(int speedValue)|inputArgs=[int speedValue={30}]
00000:00:00:00:000  1108  TEST123      executed true    | void
    accelerometerSetValues(float[] values)|inputArgs=[float[] values
    ={100.1,200.2,300.3}]
00000:00:00:00:000  1096  SETGYRO1     executed true    | void
    Gyro1SetRate(float[] values)|inputArgs=[float[] values
    ={-1000.0,1001.1,1003.23}]
00000:00:00:00:000  1191  SETGYRO2     executed true    | void
    Gyro2SetRate(float[] values)|inputArgs=[float[] values
    ={-1005.23,2001.12,3002.23}]
00000:00:00:00:000  1054  SETMTQX      executed true    | void
    MTQXSetDipoleMoment(int dipoleValue)|inputArgs=[int dipoleValue
    ={1234}]
00000:00:00:00:000  1061  SETMTQY      executed true    | void
    MTQYSetDipoleMoment(int dipoleValue)|inputArgs=[int dipoleValue
    ={2345}]
00000:00:00:00:000  1068  SETMTQZ      executed true    | void
    MTQZSetDipoleMoment(int dipoleValue)|inputArgs=[int dipoleValue
    ={3456}]
00000:00:00:05:000  7005  RAWIMAGE1    executed true    | void
    simPreloadFile(String fileName)|inputArgs=[String fileName={
    rawimage20160819142207.raw}]
00000:00:00:06:000  3002  RAWEXAMPLE1  executed true    | void
    simPreloadPicture(String fileName)|inputArgs=[String fileName={
    rawimage20160819142207.raw}]
00000:00:00:07:000  6002  ISS          executed true    | void
    simPreloadFile(String fileName)|inputArgs=[String fileName={iss.wav}]
00000:00:00:07:000  1204  QSERVER      executed true    | void
    simRunDeviceCommand(String data)|inputArgs=[String data={
    quaternionServer:10500}]
00000:00:00:08:000  1204  CAMERASCRIPT executed true    | void
    simRunDeviceCommand(String data)|inputArgs=[String data={cameraScript
    :/home/pi/repos/RPI_GPS_Tracker/helper/takePicture2.sh}]
```

LISTING 5.2: Simulator scheduler listing

The application to interact over serial communication with the is available on the Adafruit repository [20]. It has been complemented with a small TCP stack which connects to the quaternion server (see 4.1.3). The output of the sensor program is listed in 5.3.

```
connecting to localhost port 10500
Connected!
sending "0 0 0 0"
System status: 5
Self test result (0x0F is normal): 0x0F
Software version:   785
Bootloader version: 21
Accelerometer ID:   0xFB
Magnetometer ID:    0x32
```

Machine	1500 runs [ms]	# gps constellations in 1500 runs	reach 3000 runs [ms]
ESA Laptop	15132	1484	42966
Agsa VM Testbed	16035	1494	42160
Raspberry PI	19433	493	139511

TABLE 5.2: Overall performance benchmark results

```
Gyroscope ID:       0x0F
Reading BNO055 data, press Ctrl-C to quit...
sending "0.0 0.0 0.0 0.0"
sending "-0.628540039062 0.357116699219 -0.34814453125 0.596801757812"
```

LISTING 5.3: TCP client sending quaternion values

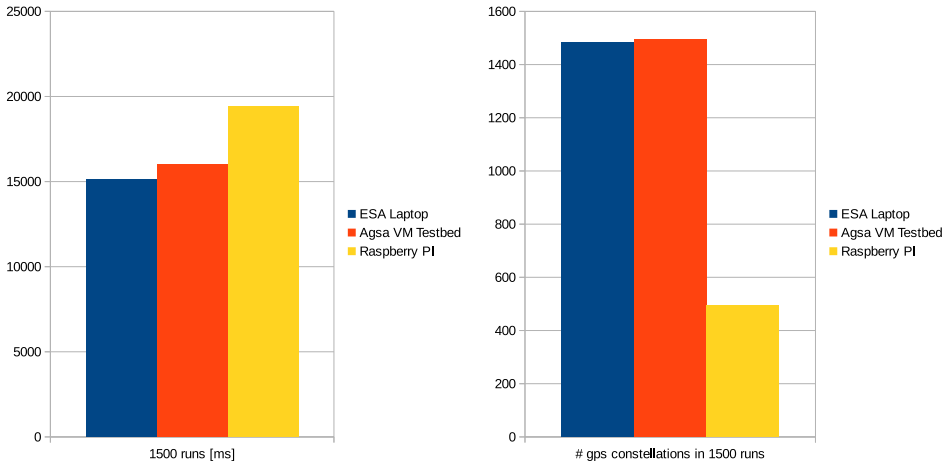
5.3 Performance analysis

The simulator instance has been tested extensively for stability. Typical long-run scenarios spanned over the duration of several days, with continuous parameter polling and commands executions. To evaluate and compare the behavior of the simulator, several benchmark tests have been defined and executed. During the benchmark tests, the TCP server had one simulator control panel client connected, the Celestia server had one client connected and the simulator instance was instantiated within a NanoSat MO Framework app.

1. time required to perform a number of core executions
2. within a number of core executions, how many GPS constellation propagations are performed
3. time required to reach steady operation (start up)

The tests have been run on three different machines: the virtual machine shown in table 5.1, a personal computer and a Raspberry PI (revision 3). Due to the lightweight design of the simulator, it was expected that computations tests (first two) are similar, whereas the start up time would be slower for the ARM device.

From figure 5.4a it can be seen that all machines have similar performance which is good, because the baseline operation of the simulator should be same regardless of the system it is running on. However this changes for the GPS propagation, figure 5.4b, which is more numerically intensive since it requires a step propagation and an elevation/azimuth calculation for each of the GPS satellites. Therefore, the separate thread which is described in 4.2.4 is definitely required to keep the response time for all machines and the only consequence of running the simulator on the Raspberry PI is that data will not be updated to the same frequency like on the other machines (it will be three times slower). The application start time also reflects the poorer performance of the ARM board, as seen in figure 5.5. Since the orekit library requires initialization of ephemeris data providers which are loaded from a resources file, the disk read time adds considerable delay to the Raspberry PI.



(A) Core runs (B) GPS constellations

FIGURE 5.4: Computation benchmarks

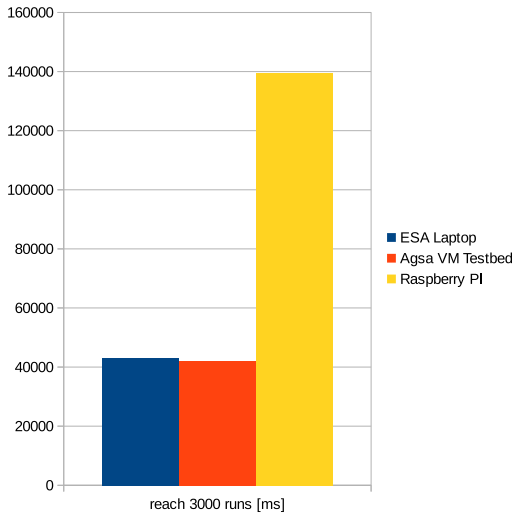


FIGURE 5.5: Start time comparison

5.4 Conclusions

The major objectives of this thesis work have been pursued and fulfilled. First of all, the simulator has been developed in JAVA and is a lightweight module which can be instantiated within the NanoSat MO Framework. As seen from the performance analysis results, it is capable to run reliably on a Raspberry PI, albeit taking more time to start. Second of all, the simulator reproduces with primitive immutable data types the interfaces of the peripheral devices used on OPS-SAT. To achieve this a simple process was devised by using an Excel file which automatically generates code stubs for the Java classes, therefore eliminating duplicate definitions in the code. The mission peripherals ICDs have been inspected and input in the document, totaling close to 300 different commands. Next, the simulator allows control of the time system, as well as possibility to run manual commands remotely from a simple, intuitive GUI. This gives a feeling of the raw satellite data and permits

complex debugging scenarios. Moreover, to generate realistic science data, existing proven libraries for orbital propagation have been used (reaching a complete GPS satellite constellation in view simulation). Finally, the Celestia application allows visualization of crucial satellite information such as position, attitude and some orbital planning information.

Appendix A

Appendix

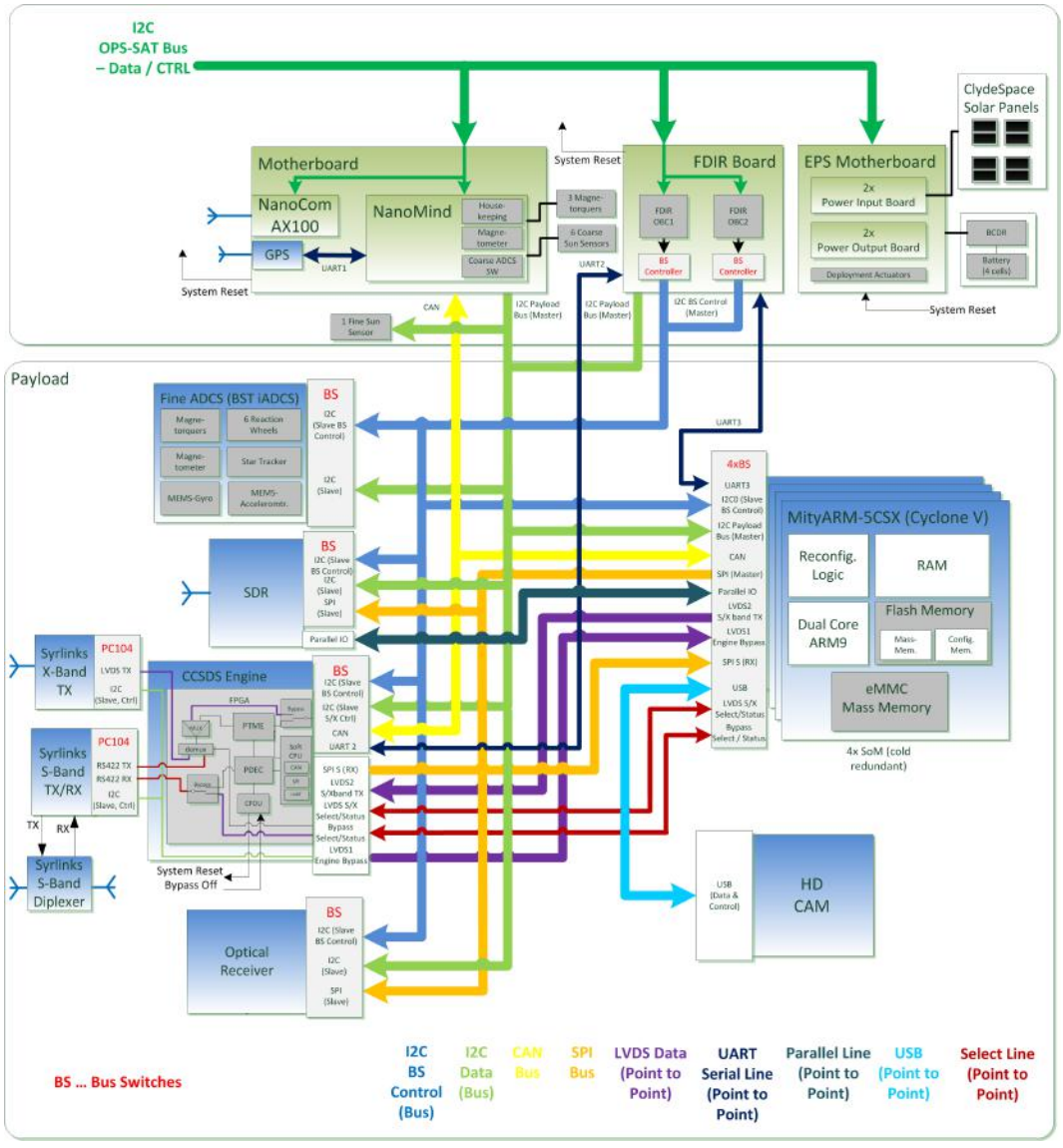


FIGURE A.1: OPS-SAT Architecture Overview

Data type	Default value / format	Value range
Byte	{0x00}	{0x00} to {0xFF}
Byte[]	{0x00,0x00}	{0x00} to {0xFF}
Int	{0}	{-2147483648} to {2147483647}
Int[]	{0,0}	{-2147483648} to {2147483647}
Long	{0}	{-9223372036854775808} to {9223372036854775807}
Long[]	{0,0}	{-9223372036854775808} to {9223372036854775807}
Float	{0.0}	{1.4E-45} to {3.4028235E38}
Float[]	{0.0,0.0}	{1.4E-45} to {3.4028235E38}
Double	{0.0}	{4.9E-324} to {1.7976931348623157E308}
Double[]	{0.0,0.0}	{4.9E-324} to {1.7976931348623157E308}
String	{}	{}

TABLE A.1: Primitive data types handled by simulator

Semi major-axis [km]	7021
Eccentricity	0
Inclination [deg]	98.05

TABLE A.2: OPS-SAT orbital elements

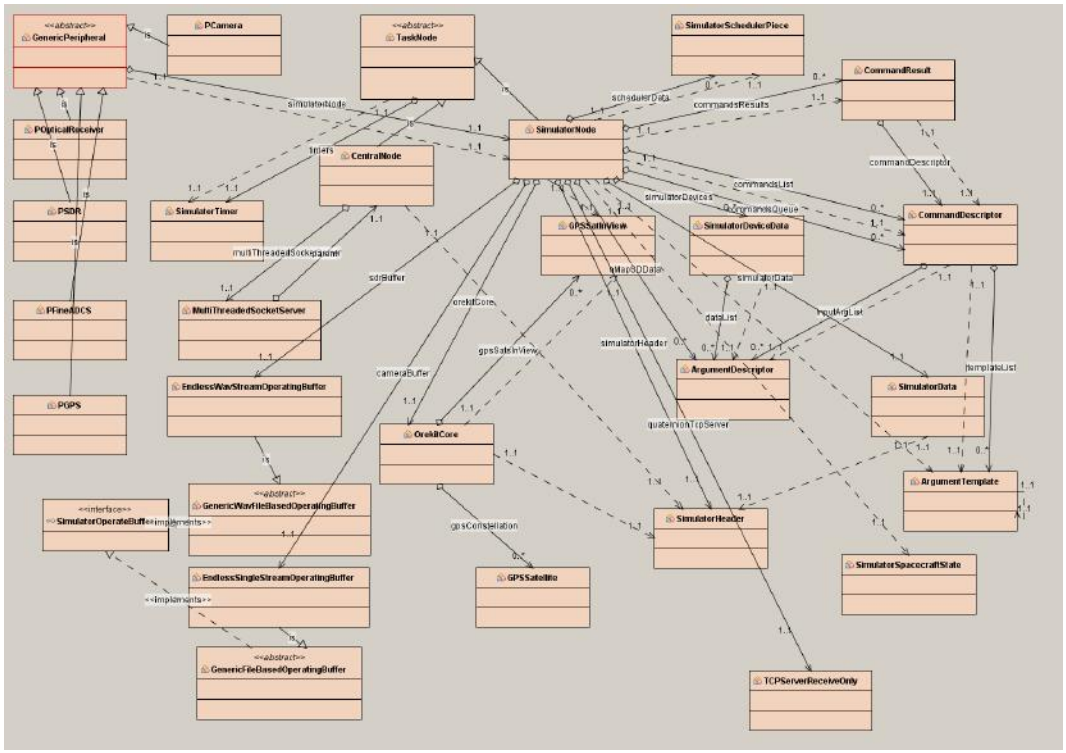


FIGURE A.2: UML class diagram main application

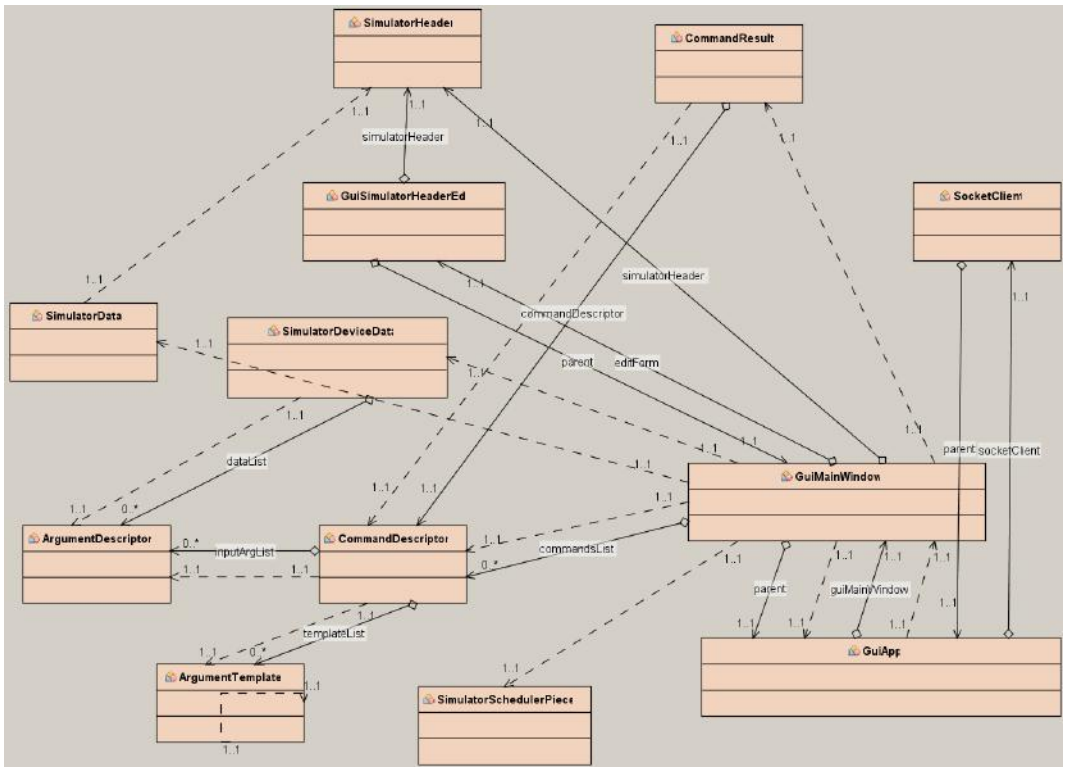


FIGURE A.3: UML class diagram GUI application

1015	FineADCS	byte[] GetSensorTelemetry(
1016	FineADCS	byte[] GetActuatorTelemetry(
1021	FineADCS	byte[] GetMagneticTelemetry(
1022	FineADCS	byte[] GetSunTelemetry(
1075	FineADCS	void SetRWXSpeed(int speedValue)
1080	FineADCS	void SetRWYSpeed(int speedValue)
1085	FineADCS	void SetRWZSpeed(int speedValue)
1096	FineADCS	void Gyro1SetRate(float[] values)
1108	FineADCS	void accelerometerSetValues(float[] values)
1162	FineADCS	void orbitSetTLE(byte[] tleData)
1167	FineADCS	void opModeDetumble(byte start,long[] times)
1168	FineADCS	void opModeSunPointing(byte[] mode,long[] times,float[] targetSunVector)
1169	FineADCS	byte[] opModeGetSunPointingStatus(
1170	FineADCS	void opModeSetModeSpin(byte mode,long[] times,float[] targetVector)
1171	FineADCS	byte[] opModeGetSpinModeStatus(
1174	FineADCS	void opModeSetNadirTargetTracking(byte mode,long[] times)
1175	FineADCS	byte[] opModeGetNadirTargetTrackingStatus(
1178	FineADCS	void opModeSetFixWGS84TargetTracking(byte mode,long[] times,float[] latitudeLongitude)
1179	FineADCS	byte[] opModeGetFixWGS84TargetTracking(
1182	FineADCS	byte[] simGetOrbitTLEBytesFromString(String tleLine1,String tleLine2)
1183	FineADCS	float simGetFloatFromByteArray(byte[] data,int byteOffset)
1184	FineADCS	byte[] simGetByteArrayFromFloat(float data)
1185	FineADCS	double simGetDoubleFromByteArray(byte[] data,int byteOffset)
1186	FineADCS	byte[] simGetByteArrayFromDouble(double data)
1187	FineADCS	int simGetIntFromByteArray(byte[] data,int byteOffset)
1188	FineADCS	byte[] simGetByteArrayFromInt(int data)
1189	FineADCS	long simGetLongFromByteArray(byte[] data,int byteOffset)
1190	FineADCS	byte[] simGetByteArrayFromLong(long data)
1191	FineADCS	void Gyro2SetRate(float[] values)
1202	FineADCS	byte[] Gyro2GetQuaternionFromSunSensor(
1203	FineADCS	int simGetInt16FromByteArray(byte[] data,int byteOffset)

TABLE A.3: FineADCS commands

2001	GPS	String getNMEASentence(String inputSentence)
2002	GPS	String getLastKnownPosition()

TABLE A.4: GPS commands

Identifier	Description
GLMLA	Almanac data for GLONASS satellites.
GPALM	Raw almanac data for each GPS satellite PRN contained in the broadcast message.
GPGGA	Time and position for the GNSS receiver without a valid almanac.
GPGGALONG	Same as GPGGA but with higher precision.
GPGGARTK	Same as GPPGA but with higher precision and different algorithm.
GPGLL	Latitude and longitude of present vessel position.
GPGRS	Range residuals after psotion calculation.
GPGSA	GNSS receiver operating mode, satellites used for navigation and DOP values.
GPGST	Pseudorange measurement noise statistics.
GPGSV	GPS satellites in view.
GPHDT	NMEA heading log.
GPRMB	Navigation information.
GPRMC	GPS specific information.
GPVTG	Track made good and ground speed.
GPZDA	UTC time and date.

TABLE A.5: GPS receiver supported sentences

3001	Camera	byte[] takePicture(int width,int height)
3002	Camera	void simPreloadPicture(String fileName)

TABLE A.6: Camera commands

7002	OR	void simSetMessageBuffer(byte[] buffer)
7003	OR	void simSetSuccessRate(int successRate)
7004	OR	byte[] readFromMessageBuffer(int bytesNo)
7005	OR	void simPreloadFile(String fileName)

TABLE A.7: Optical receiver commands

6001	SDR	byte[] runRawCommand(int cmdID,byte[] data)
6002	SDR	void simPreloadFile(String fileName)
6003	SDR	double[] readFromBuffer(int numberSamples)

TABLE A.8: SDR commands

```
1 =CONCATENATE("/**",CHAR(10),"<pre>",CHAR(10),$M213,CHAR(10),"Input
      parameters:",$V213,CHAR(10),"Return parameters:",$C213,CHAR(10),"Size
      of returned parameters: ",IF($D213="","0",$D213),CHAR(10),$N213,CHAR
      (10),"</pre>",CHAR(10),"*/")
2 =CONCATENATE(Formatting!B$1,C213," ",E213,"(",V213,");//",Q213,"//",M213)
3 =CONCATENATE($X213,CHAR(10),$C213," ",$E213,"(",$V213,,");//",$Q213)
```

LISTING A.1: Excel code to generate interface method

```
1 =CONCATENATE(Formatting!B$1,"@Override",CHAR(10),Formatting!B$1,"
      @InternalData (internalID=",Q213,",commandIDs={",CHAR(34),O213,CHAR
      (34),",",CHAR(34),P213,CHAR(34),"},argNames={",IF(F213="",CONCATENATE(
      CHAR(34),CHAR(34)),IF(F213=1,CONCATENATE(CHAR(34),H213,CHAR(34)),IF(
      F213=2,CONCATENATE(CHAR(34),H213,CHAR(34),",",CHAR(34),J213,CHAR(34)),
      IF(F213=3,CONCATENATE(CHAR(34),H213,CHAR(34),",",CHAR(34),J213,CHAR
      (34),",",CHAR(34),L213,CHAR(34)),"other"))))),"})",CHAR(10),Formatting!
      B$1,"public ",C213," ",E213,"(",V213,") {",CHAR(10),IF(F213="",
      CONCATENATE(Formatting!B$1,Formatting!B$1,"ArrayList<Object> argObject
      =null;",CHAR(10)),IF(F213=1,CONCATENATE(Formatting!B$1,Formatting!B$1,
      "ArrayList<Object> argObject = new ArrayList<Object>();",CHAR(10),
      Formatting!B$1,Formatting!B$1,"argObject.add(",H213,");",CHAR(10)),IF(
      F213=2,CONCATENATE(Formatting!B$1,Formatting!B$1,"ArrayList<Object>
      argObject = new ArrayList<Object>();",CHAR(10),Formatting!B$1,
      Formatting!B$1,"argObject.add(",H213,");",CHAR(10),Formatting!B$1,
      Formatting!B$1,"argObject.add(",J213,");",CHAR(10)),CONCATENATE(
      Formatting!B$1,Formatting!B$1,"ArrayList<Object> argObject = new
      ArrayList<Object>();",CHAR(10),Formatting!B$1,Formatting!B$1,"
      argObject.add(",H213,");",CHAR(10),Formatting!B$1,Formatting!B$1,"
      argObject.add(",J213,");",CHAR(10),Formatting!B$1,Formatting!B$1,"
      argObject.add(",L213,");",CHAR(10))))),IF(C213="void",CONCATENATE(
      Formatting!B$1,Formatting!B$1),CONCATENATE(Formatting!B$1,Formatting!
      B$1,"return (",W213,") ")),"super.getSimulatorNode().runGenericMethod(
      ",Q213,",argObject);",CHAR(10),Formatting!B$1,"};")
```

LISTING A.2: Excel code to generate device forward method

```
1 =CONCATENATE(Formatting!B$1,"case ",Q213,": {//Origin [",A213,"] Method [
      ",TRIM(Y213),"]",CHAR(10),IF(F213="","",IF(F213=1,CONCATENATE(
      Formatting!B$1,Formatting!B$1,G213," ",H213,"=(",R213,") argObject.get
      (0);",CHAR(10)),CONCATENATE(Formatting!B$1,Formatting!B$1,G213," ",
      H213,"=(",R213,") argObject.get(0);",CHAR(10),Formatting!B$1,
      Formatting!B$1,I213," ",J213,"=(",S213,") argObject.get(1);",CHAR(10))
      )),IF(C213="void",CONCATENATE(Formatting!B$1,Formatting!B$1,"break; }"
      ),CONCATENATE(Formatting!B$1,Formatting!B$1,C213," result=",IF(C213="
      void","null",IF(OR(C213="byte",C213="int",C213="long"),"0",IF(C213="
      String",CONCATENATE(CHAR(34),"Placeholder",CHAR(34)),IF(C213="byte[]",
      CONCATENATE("new byte[",D213,"]"),"ERROR")))),";",CHAR(10),Formatting!
      B$1,Formatting!B$1,"globalResult=result;",CHAR(10),Formatting!B$1,
      Formatting!B$1,"break;}")))
```

LISTING A.3: Excel code to generate switch handler

Interface	IGPS
Class	PGPS
Return	String
N_Data	
Method Name	getNMEASentence
ArgC	1
Arg1 Type	String
Arg1 Act	inputSentence
Arg2 Type	
Arg2 Act	
Arg3 Type	
Arg3 Act	
Comment	Obtain a NMEA response for a given NMEA sentence
Extended comment	
iAD	
Command ID	
Interface	IFineADCS
Class	PFineADCS
Return	0
N_Data	0
Method Name	opModeSetTargetCapture1
ArgC	3
Arg1 Type	byte
Arg1 Act	mode
Arg2 Type	long
Arg2 Act	startTime
Arg3 Type	float[]
Arg3 Act	data
Comment	High level command to interact with FineADCS
Extended comment	Sets the 'HL Target Capture 1 Mode'. Enables 'Control Loop' based on RW, HP-gyro, Kalman Filter and ST or S/B-Sensors. Uses simple euler rotations for tracking. Turns off all other unnecessary activities. Parameter: UI8 Mode 1x F32 Latitude [rad] 1x F32 Longitude [rad] 1x UI64 Start Time 4x F32 Inertial Target Quaternion
iAD	0x36
Command ID	0x11

TABLE A.9: Inputs to Excel generator

```
tail: /home/cezar/temp/_OPSSAT_SIMULATOR/Sim.log: file truncated
2016:09:20;09:06:27.238;1474355187238;opssat.simulator.threading.TaskNode
    ;initLogging;SEVERE;File logging level is [INFO]
2016:09:20;09:06:27.239;1474355187239;opssat.simulator.threading.TaskNode
    ;initLogging;SEVERE;Console logging level is [INFO]
2016:09:20;09:06:27.240;1474355187240;opssat.simulator.threading.
    SimulatorNode;loadMethodsFromReflection;INFO;loadMethodsFromReflection
2016:09:20;09:06:27.340;1474355187340;opssat.simulator.threading.
    SimulatorNode;loadTemplatesFromFile;INFO;Loaded [60] custom templates
2016:09:20;09:06:27.343;1474355187343;opssat.simulator.threading.
    SimulatorNode;loadSchedulerFromFile;INFO;Loaded [12] scheduler
    commands
2016:09:20;09:06:27.343;1474355187343;opssat.simulator.threading.
    SimulatorNode;loadSchedulerFromFile;INFO;Scheduler file parsing ok!
2016:09:20;09:06:27.345;1474355187345;opssat.simulator.threading.
    SimulatorNode;loadSimulatorHeader;INFO;Header [/home/cezar/repos/
    NANOSAT_MO_FRAMEWORK_SOFTWARE_BUNDLE/NMF_PLT/
    PLATFORM_SOFTWARE_SIMULATOR/OPS-SAT_SIMULATOR/_OPS-SAT-SIMULATOR-
    header.txt] found!
2016:09:20;09:06:27.355;1474355187355;opssat.simulator.threading.
    SimulatorNode;loadSimulatorCommandsFilter;INFO;Filter [/home/cezar/
    repos/NANOSAT_MO_FRAMEWORK_SOFTWARE_BUNDLE/NMF_PLT/
    PLATFORM_SOFTWARE_SIMULATOR/OPS-SAT_SIMULATOR/_OPS-SAT-SIMULATOR-
    filter.txt] found!
2016:09:20;09:06:27.358;1474355187358;opssat.simulator.threading.
    SimulatorNode;initModels;WARNING;Errors found during parsing of
    simulator header. Loading default OPS-SAT keplerian elements.
2016:09:20;09:06:27.365;1474355187365;opssat.simulator.threading.
    SimulatorNode;initModels;INFO;Calling orekit constructor
2016:09:20;09:06:30.650;1474355190650;opssat.simulator.orekit.OrekitCore;
    changeAttitude;INFO;Changing attitude to SUN_POINTING
2016:09:20;09:06:33.149;1474355193149;opssat.simulator.orekit.OrekitCore
    ;<init>;INFO;Decimal year is [2016.6575342465753]
2016:09:20;09:06:33.399;1474355193399;opssat.simulator.orekit.OrekitCore
    ;<init>;INFO;Magnetic model loaded: [IGRF2015]
2016:09:20;09:06:33.400;1474355193400;opssat.simulator.orekit.OrekitCore
    ;<init>;SEVERE;Orekit module created with start date [2016-08-27T10
    :05:58.000]
2016:09:20;09:06:34.417;1474355194417;opssat.simulator.orekit.OrekitCore
    ;<init>;INFO;GPS Constellation has [31] satellites!
2016:09:20;09:06:34.417;1474355194417;opssat.simulator.threading.
    SimulatorNode;initModels;INFO;orekit initialized successfully
2016:09:20;09:06:34.483;1474355194483;opssat.simulator.threading.
    SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
    identifier;1075;inputArgs=[int speedValue={10}]
2016:09:20;09:06:34.486;1474355194486;opssat.simulator.main.ESASimulator;
    initDevices;INFO;Initializing devices..
2016:09:20;09:06:34.486;1474355194486;opssat.simulator.threading.
    SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
    identifier;1080;inputArgs=[int speedValue={20}]
2016:09:20;09:06:34.493;1474355194493;opssat.simulator.threading.
    SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
    identifier;1085;inputArgs=[int speedValue={30}]
2016:09:20;09:06:34.494;1474355194494;opssat.simulator.threading.
    SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
    identifier;1108;inputArgs=[float[] values={100.1,200.2,300.3}]
2016:09:20;09:06:34.497;1474355194497;opssat.simulator.threading.
    SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
    identifier;1096;inputArgs=[float[] values={-1000.0,1001.1,1003.23}]
2016:09:20;09:06:34.498;1474355194498;opssat.simulator.threading.
    SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
    identifier;1191;inputArgs=[float[] values={-1005.23,2001.12,3002.23}]
```

```
25 2016:09:20;09:06:34.498;1474355194498;opssat.simulator.threading.
       SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
       identifier;1054;inputArgs=[int dipoleValue={1234}]
26 2016:09:20;09:06:34.499;1474355194499;opssat.simulator.threading.
       SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
       identifier;1061;inputArgs=[int dipoleValue={2345}]
27 2016:09:20;09:06:34.507;1474355194507;opssat.simulator.threading.
       SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
       identifier;1068;inputArgs=[int dipoleValue={3456}]
28 2016:09:20;09:06:39.490;1474355199490;opssat.simulator.threading.
       SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
       identifier;7005;inputArgs=[String fileName={rawimage20160819142207.raw
       }]
29 2016:09:20;09:06:40.488;1474355200488;opssat.simulator.threading.
       SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
       identifier;3002;inputArgs=[String fileName={rawimage20160819142207.raw
       }]
30 2016:09:20;09:06:41.487;1474355201487;opssat.simulator.threading.
       SimulatorNode;runGenericMethodForCommand;INFO;runGenericMethod;
       identifier;6002;inputArgs=[String fileName={iss.wav}]
31 2016:09:20;09:06:41.490;1474355201490;opssat.simulator.util.
       GenericWavFileBasedOperatingBuffer;loadFromPath;INFO;
32 File: /home/cezar/ops-sat-simulator-resources/iss.wav
33 Channels: 2, Frames: 3330048
34 IO State: READING
35 Sample Rate: 55555, Block Align: 4
36 Valid Bits: 16, Bytes per sample: 2
37 2016:09:20;09:07:07.021;1474355227021;opssat.simulator.threading.
       SimulatorNode;coreRun;INFO;BenchmarkStart;Counter [3000]
38 2016:09:20;09:07:07.023;1474355227023;opssat.simulator.threading.
       SimulatorNode;coreRun;INFO;BenchmarkStartup;Counter [3000];TimeElapsed
        [39783] ms
39 2016:09:20;09:07:23.088;1474355243088;opssat.simulator.threading.
       SimulatorNode;coreRun;INFO;BenchmarkFinished;TimeElapsed [16078] ms;
       Counter [4500];Steps [1500]
40 2016:09:20;09:07:23.089;1474355243089;opssat.simulator.threading.
       SimulatorNode;coreRun;INFO;BenchmarkFinished;Orekit GPS constellation
       propagations [1500]
```

LISTING A.4: Simulator log file

Bibliography

[1] D. Evans Dr. Mario Merri. "OPS-SAT: A ESA nanosatellite for accelerating innovation in satellite control". In: *SpaceOps Conference*. May 2014.

[2] CCSDS Secretariat. *Mission Operations Services Concept - Green Book*. Tech. rep. The Consultative Committee for Space Data System, Dec. 2010.

[3] Dr. Mehran Sarkarati C. Coelho Dr. Mario Merri and Prof. O. Koudelka. "NanoSat MO Framework: Achieving On-board Software Portability". In: *SpaceOps Conference*. May 2016.

[4] O. Koudelka C. Coelho D. Evans. "CCSDS Mission Operations Services on OPS-SAT". In: *10th IAA Symposium on Small Satellites for Earth Observation*. Apr. 2015.

[5] *JAVA Platform Standard Edition 7. ConcurrentLinkedQueue API.* URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html.

[6] *JAVA Platform Standard Edition 7. Reflection API.* URL: https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html.

[7] *JAVA Object Streams*. URL: https://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html.

[8] *Celestia project homepage*. URL: http://celestiaproject.net/index.html.

[9] *International Geomagnetic Reference Field*. URL: http://www.ngdc.noaa.gov/IAGA/vmod/igrf.html.

[10] *Magnetic Field Calculator*. URL: http://www.ngdc.noaa.gov/geomag-web/?model=igrf#igrfwmm.

[11] *Two line element*. URL: https://en.wikipedia.org/wiki/Two-line_element_set.

[12] *OEM6®Family Firmware Reference Manual*. URL: http://www.novatel.com/assets/Documents/Manuals/om-20000129.pdf.

[13] *Hyperion Technologies ST200 Star Tracker*. URL: http://hyperiontechnologies.nl/wp-content/uploads/2016/08/HTBST-ST200-V1.0_Flyer.pdf.

[14] *UHF VHF deployable antenna*. URL: http://www.isispace.nl/brochures/ISIS_AntS_Brochure_v.7.11.pdf.

[15] *Java wav file reader*. URL: http://www.labbookpages.co.uk/audio/javaWavFiles.html#reading.

[16] *Adafruit Ultimate GPS Breakout - 66 channel w/10 Hz updates - Version 3*. URL: https://www.adafruit.com/products/746#tutorials.

[17] *Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055*. URL: https://www.adafruit.com/products/2472#tutorials.

[18] *BMP180 Barometric Pressure/Temperature/Altitude Sensor- 5V ready*. URL: https://www.adafruit.com/products/1603#tutorials.

[19] *Raspberry Pi Camera Board*. URL: https://www.adafruit.com/products/1367#tutorials.

[20] *Adafruit Unified BNO055 Driver (AHRS/Orientation)*. URL: https://github.com/adafruit/Adafruit_BNO055.

[21] Prabath Siriwardena. *Maven Essentials*. 1st ed. Packt Publishing, Dec. 2015. ISBN: 178398676X.