Aurora

Evidences for the assessment report

D6.2

Document Code:	AUR-ESC-RP-0014
Document Version:	2.2
Document Date:	02/06/2023
Internal Reference:	DOC00304822

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101004291

Signature Control

Written	Checked	Approved Configuration Management	Approved Quality Assurance	Approved Project Management
V. Fišer A. Lyko A.I. Rodriguez M.A de Miguel A.G. Pérez J. Gómez del Pulgar	P. Česák A.I. Rodriguez	R.M León	A. López	A.I. Rodriguez
Date and Signature	Date and Signature	Date and Signature	Date and Signature	Date and Signature

Signature not needed if electronically approved by route

Changes Record

Rev	Date	Author	Affected section	Changes
0.1	24-06-2022	V. Fišer	All	Initial issue
0.2	05-08-2022	V. Fišer	1, 4, 5, Annex B	Modified document structure, transformed Annex B
0.3	12-08-2022	V. Fišer	4, 5, Annex C	Added part of UPM data, enhanced chapter 4 structures, expanded table in chapter 5
0.4	24-08-2022	A.I. Rodriguez	1 and 3; All	Minor changes in 1 and 3. All: general revision of the document
0.5	12-09-2022	V. Fišer, A. Lyko A.G. Pérez	4 and 5; All 1.1, 1.21.3, 2.3, and 5	Split ESC/SAE and UPM results into two chapters All: implemented feedback from comments Updated document's purpose in section 1.1. Added document's contents in section 1.3. Added UPMSat-2 overview in sections 5.1 and 5.2. Added UPMSat-2 KPI values.
0.6	26-09-2022	V. Fišer A.G. Pérez	4.2 and 4.3 9, 10, 11, and Annex C	Added results of TASTE integration and related chapters Add more SIL and PIL test cases for UPMSat-2
1.0	03-10-2022	V. Fišer	4	Included all feedback into initial version, added links into references in chapter 4
1.1	03-10-2022	A.G. Pérez	5	Added elements from 0.6 version lost during the first integration in v1.0.
2.0	26-01-2023	P. Česák A.G. Pérez V. Fišer J. Gómez	4.1 4.2 4.12 5.12	Prepared for MS8 (editorial changes) Added sections 4.1and 4.2 about Euclid KPIs Updated Euclid KPI data Summary Updated UPMSAT2 KPI data Summary
2.1	06-02-2023	P. Česák A.G. Pérez V. Fišer J. Gómez	General	Released after internal review
2.2	02-06-2023	J. Gómez	Annex A	Included the Excel file referenced in Annex A.

Table of contents

1. Introduction 6
1.1. Purpose 6
1.2. Scope 6
1.3. Contents 6
2. Related Documents 7
2.1. Applicable documents 7
2.2. Reference documents 7
2.3. Acronyms 8
2.4. Terms and definitions 9
3. Overview 10
4. Euclid KPI data 12
4.1. Introduction to the Euclid Attitude and Orbit Control System 12
4.2. Euclid Simulink and TASTE models 13
4.3. Code interfaces 14
4.4. Taste 14
4.5. Tools 16
4.6. AdaCore's support 16

4.7. Effort 17
4.8. Models 19
4.9. Code Metrics 21
4.10. Coverage 23
4.11. Tests 24
4.12. Euclid KPI data Summary 26

5. UPMSAT2 KPI data 31
5.1. Introduction to the UPMSat-2 Attitude Control System 31
5.2. UPMSat-2 ACS Simulink and TASTE models 32
5.3. Code interfaces 34
5.4. Taste 36
5.5. Tools 37
5.6. AdaCore's support 38
5.7. Effort 38
5.8. Models 38
5.9. Code Metrics 42
5.10. Coverage 47
5.11. Tests 50
5.12. UPMSAT2 KPI data Summary 52

Annex A – Euclid KPI results 57

Annex B – Models adaptation metrics..... 58

Annex C – UPMSat-2 PIL & SIL test and coverage results 60

1. Introduction

1.1. Purpose

This document is an output from Task 6.2 "Preparation of the key data TRA for QGen" included in WP6 "Demonstration Viability Assessment". This document covers the evidence for the applicability of the QGen tool set in the software design, modelling, simulation and verification of Attitude, Orbit and Control Systems (AOCS) applications in space missions.

This document shall serve as an entry point for performing the analysis of key data and evaluation of TRL.

1.2. Scope

The scope of this document is to provide information about measurement of KPIs defined in D6.1. This document aims to describe the methods used to measure data, the result of the measurement and pointers to the evidence.

1.3. Contents

The overall contents of this document have been organized as follows, including this introductory chapter:

- Chapter 2 contains the applicable and reference documents, the list of terms, definitions, and acronyms.
- Chapter 3 gives a general overview of the evaluators and the TRA plan phases.
- Chapter 4 presents the KPIs and analyses the KPI results from the EUCLID technology demonstrator.
- Chapter 5 presents the KPI results from the UPMSat-2 demonstrator.

2. Related Documents

The following documents in the latest issue/revision are part of this document.

2.1. Applicable documents

AD #	Title	Reference	Issue	Rev
[AD01]	AURORA Grant Agreement	GA number 101004291	-	-
[AD02]	Quality Assurance Management Plan	AUR-SAE-PL-0001	-	1
[AD03]	AURORA SW Development Plan	AUR-SAE-PL-0002	-	1
[AD04]	AOCS/GNC Code Generator Requirements specification	AUR-SAE-SP-0001	-	1.0

Table 1 Applicable documents

2.2. Reference documents

RD #	Title	Reference	Issue	Rev
[RD01]	D3.4 QGen tool-set and SW development environment	AUR-ESC-RP-0022		
[RD02]	D3.9 Test cases reporting (PIL & HIL)	AUR-ESC-RP-0008		
[RD03]	D3.10 SW Verification report (PIL & HIL)	AUR-ESC-RP-0009		
[RD04]	D3.6 QGen evaluation report	AUR-ESC-RP-0024		
[RD05]	D3.7.2. Test cases specification	AUR-SAE-SP-0002		
[RD06]	D6.1 TRA plan	AUR-UPM-PL-0005		
[RD07]	D3.5 Demonstration Simulink Models	AUR-SAE-RP-0023		
[RD08]	Demonstration for the QGen integration into TASTE	Link to the website		
[RD09]	UPMSat-2 ACS implemented in TASTE	Link to the GitHub repo		
[RD10]	UPMSat-2 OBDH implemented in TASTE	Link to the GitHub repo		
[RD11]	UPMSAt-2 ACS Simulink models	Link to the GitHub repo		
[RD12]	WP6 Study on effort estimation in Autocoding SW Development	AUR-SAE-TN-0002		

Table 2 Reference documents

2.3. Acronyms

Acronym	Description
AOCS	Attitude and Orbit Control (sub-)System
AD	Applicable Document
GNC	Guidance Navigation Control
HDD	Hard Disk Drive
HW	Hardware
HTML	Hypertext Markup Language
KPI	Key Performance Indicator
LOC	Lines of code
MIL	Model in the loop
N/A	Not Applicable or Available
OS	Operating System
PIL	Processor in the loop
PWM	Pulse Width Modulation
QA	Quality Assurance
RAM	Random Access Memory
RD	Reference Document
SIL	Software in the loop
SW	Software
TASTE	The ASSERT Set of Tools for Engineering
TBA	To be added
TBD	To be Determined
TBW	To be written
TRA	Technological Readiness Assessment
TRL	Technological Readiness Level
VM	Virtual Machine

Table 3 Acronyms

2.4. Terms and definitions

Acronym	Description
AdaCore	SW development tools supplier company
Matlab	Programming and numeric computing platform used to analyse data, develop algorithms, and create models.
Processor in the loop	In processor-in-the-loop (PIL) simulation, the generated code from model runs directly on the target hardware, which means you can test models on the hardware using the same test cases as on the host.
QGen	Qualifiable code generator and static analyser for Simulink(R) and Stateflow(R)
Software in the loop	The Production Software Code is incorporated into the mathematical simulation that contains the models of the Physical System.
Simulink	Block diagram environment for simulation and Model-Based Design integrated with MATLAB.
TASTE	A toolchain targeting heterogeneous embedded systems, using a model-based development approach.

Table 4 Terms and definitions

3. Overview

This document aims to explain methods and ways the KPI data was obtained for the second phase of the TRA plan. The KPI data were extracted from within the two demonstrator projects – the Euclid project and UPMSAT2 project – on which the QGen's code generation process was performed.

The TRA Plan includes three main phases:

1. Definition of the details of the TRA plan. This plan includes the identification of KPIs to provide TRL evidences and the criteria to do their evaluations. This plan will be centered on the evaluation of two TRL levels TRL6 and TRL7.
2. Quantification of key data. This phase will be elaborated on in task T6.2. This phase will include the preparation of the key data TRA for QGen and some of key data for previous version of EUCLID project. Those two evaluations will be used as comparison evaluators that will be used as evidence.
3. The final phase will reuse the key data produced in the previous phase to report the TRA for QGen. The key data will be used to perform a comparative analysis of development cost with previous technologies and QGen technologies, evaluate the applicability of QGen technologies and their impact on the software development process, and validate the results produced with QGen technologies.

The next figure includes the different phases and activities developed in every phase:

1. Definition of TRA Plan:
    a. Definition of KPI categories and KPI.
    b. Definition of TRL analyzed.
    c. Definition of KPI quantification for the evaluation of TRLs.
2. Evaluation of key data
    a. Compilation of evidences for the assessment, gathering data on indicators and additional information where necessary. The results of this gathering are used for the evaluation of KPIs.
    b. Monitoring the specific targets of the KPIs to evaluate for the different models and projects.
    c. Quantification of KPI and Generation of evidences.
3. Analysis of key data and evaluation of TRL.
    a. Integration of key data produced for the different projects and models.
    b. Evaluation of TRL 6 and 7.
    c. Elaboration of conclusions.

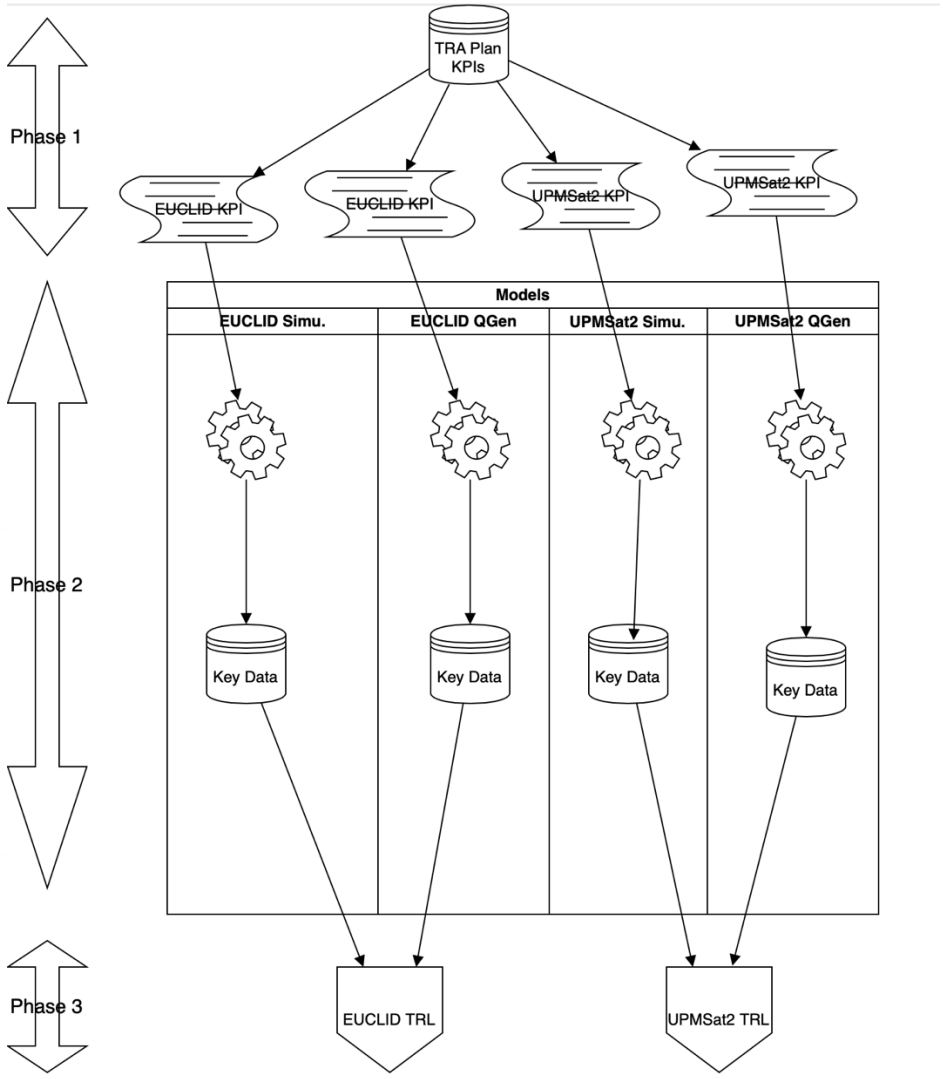

Figure 1 TRA plan phases

4. Euclid KPI data

The following chapter describes methods used to extract defined KPI data in [RD06] section 6.4, the obtained KPI data results and the location of the evidences supporting the results. If more KPIs had similar results acquisition or are stored at the same place, check subchapter text for more information.

For a generic definition of the environment used and its evaluation, please refer to RD01 and RD04.

The test cases specification and reporting can be found in RD05and RD02respectively.

4.1. Introduction to the Euclid Attitude and Orbit Control System

Euclid is a medicum-classs mission of ESA's Science Program whose objective is the elucidate the geometry and the nature of the dark energy and dark matter components with unprecedented accuracy, of the order of micro arcseconds when in science mode. For that a complex multi-mode AOCS has been developed by Sener Aeroespacial.

The AOCS is composed of sensors (Sun sensors, IMU, Coarse Rate sensors, STR and Fine Guidance Sensor) and actuators (Reaction wheels, Reaction Control System and Micro Propulsion System) for the different activities needed. The different sensors and actuators are needed in function of the AOCS mode the satellite is, being science mode the more demanding mode due to the hard limitations in terms of accuracy and stability.

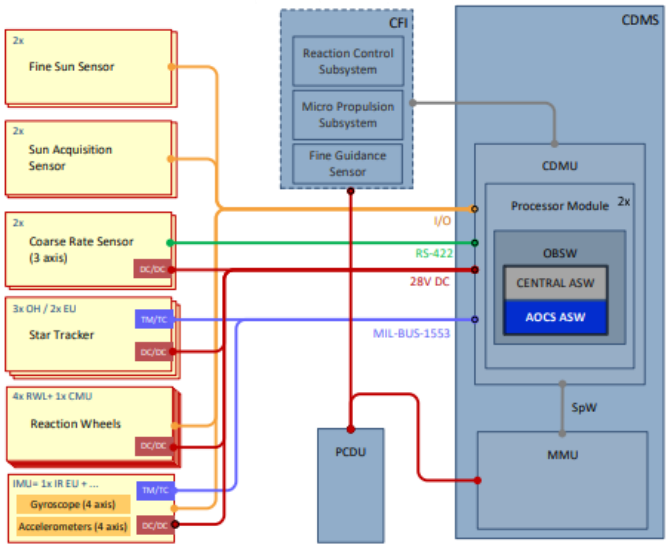

Figure 2: Euclid AOCS Architecture

Euclid AOCS has the special feature that it is one of the first missions to be launched where the GNC algorithms were autocoded based on Simulink models shifting the traditional manual code validation to an autocoded philosophy, which focus on reduction on developing times. This code is then integrated inside a manual coded application software containing the FDIR functionalities, communication with the rest of the systems and the mode manager.

4.2. Euclid Simulink and TASTE models

The model tested in TASTE was sam_ctrl model. The top part of the model can be seen on Figure 4 sam_ctrl model in Matlab. The model was modified for each of the 4 submodels (at, dz, rd, sa) and saved separately for TASTE usage. The reason for splitting the model is further described in section 4.4.2.2.

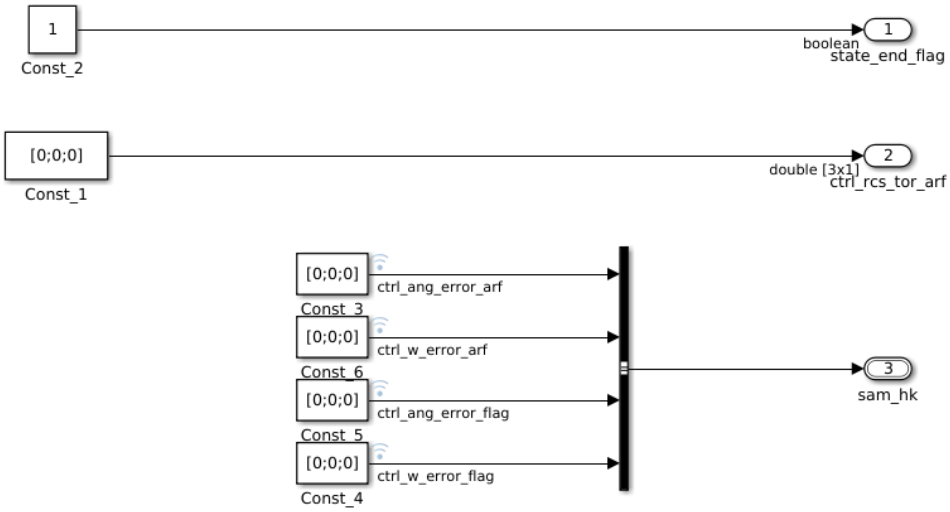

Figure 4 sam_ctrl model in Matlab

Once split in MATLAB, the model was setup in the following way. The application, written in C, contained the data. It had the tick interface periodically sending the data through the data transfer interface to the samCtrlAt section. The samCtrlAt section was marked implemented in QGenC and it generated support files as the model.slx file, where the target model was pasted and in which directory support libraries were put. Once setup, the model was built, code generated and the application ran, printing the output data to console.

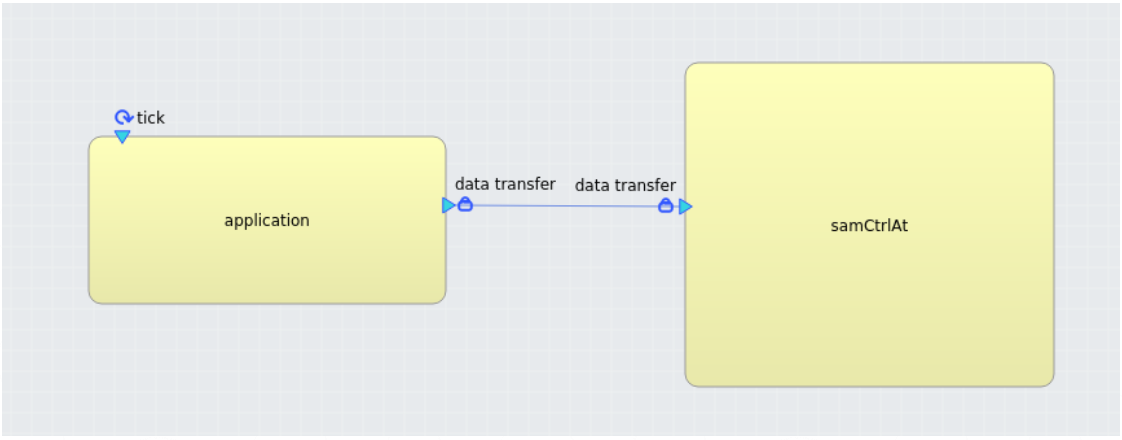

Figure 3 sam_ctrl_at model in TASTE

4.3. Code interfaces

4.3.1. OIQ01 – Integration: Other_Sw(QGen_Code)

4.3.1.1. Measurement

This KPI was not measured, its status was determined based on the QGen generated code observable characteristics. It has an entry point in the form of callable function. Therefore, it can be integrated into any project's source code at pre-build phase.

4.3.1.2. Results

QGen generated code was called from the SIL framework during SIL testing and PIL simulation framework during the PIL phase testing. Each test scenario was run with a framework. Results of the simulated runs are in Annex A, list *DYN stat.*, in the *PIL test status* column.

4.3.1.3. Evidence

The tests, where Qgen generated code was run through other code, can be found at *Teams/WP3 – AOCS-GNC-generators/Files/PIL/test*

4.3.2. OIQ02 – Inclusion: QGen_Code(Other_Sw)

4.3.2.1. Measurement

Simulink models can use C or MATLAB code via s-functions. QGen supports s-functions in code generations. This statement was tested by using a human-written code within the *fpmrcs_mm* model, specifically *fpmrcs_matlab_function.c*. The code was compiled without issues and worked as expected. A similar approach was taken by UPM as well.

4.3.2.2. Results

Other code can be used in QGen generated code. Results of the simulated run with said file are in Annex A, list *DYN stat.*, in the *SIL test status* column.

4.3.2.3. Evidence

The test scenario with the file is available at *Teams/WP3 – AOCS-GNC-generators/Files/PIL/test/19_fpmrcs_mm* with results in the log file within test scenario in former file path. QGen support for s-functions is described here: https://docs.adacore.com/live/wave/qgen/html/qgen_ug/legacy.html#calling-c-or-ada-code-using-s-function-block

4.4. Taste

4.4.1. OIQ03 – QGen integration into TASTE

4.4.1.1. Measurement

QGen was deemed integrable if it was able to be launched from TASTE. The ability to launch was verified with building correctly the provided model.

4.4.1.2. Results

QGen is integrable into TASTE with few issues. We have encountered small issues with the pipeline/process during our testing. Firstly, known bug with very simple models (QGen didn't generate init function for output=input model) and one documented in 4.4.2.2.

4.4.1.3. Evidence

Location of the built models can be found at *Teams/WP3 – AOCS-GNC-generators/Files/TASTE_outputs*

4.4.2. OIQ04 – number of Simulink models integrated into TASTE

4.4.2.1. Measurement

Final number of Simulink models (or their submodels) integrated into (build with) TASTE.

4.4.2.2. Results

After initial attempts with very simple model to familiarize ourselves with the environment, we tried to build the entire sam model. During the process, we encountered the error shown in Figure 5 and we decided to break the model into its parts and test each of them separately.

We managed to make 2 submodels work, specifically sam_ctrl_at and sam_ctrl_dz. We encountered some issues with the integration of the other submodels, however, colleagues from N7S managed to make them work. For unknown reasons, the sam_ctrl_rd model and the sam_ctrl_sa failed to compile on both default deployment and on specified x86 linux cpp deployment. In both cases, the error was the following:

```
qgenc ./xmi/data_transfer.xmi --gen-entrypoint --wrap-io --pre-process-xmi --incremental \
    --no-misra --language c --output src
[INFO] Importing XMI data_transfer.xmi
[INFO] Analyzing previous generation data
[INFO] Preprocessing model data_transfer
[INFO] Finished preprocessing model data_transfer
[INFO] Sequencing model data_transfer
[INFO] Finished sequencing model data_transfer
[INFO] Generating code model data_transfer
[INFO] Finished code model generation data_transfer
[INFO] Optimising code model data_transfer
[INFO] Finished optimising code model data_transfer
[INFO] Printing to C data_transfer
[ERROR] Unknown error occurred
raised CONSTRAINT_ERROR : Ada.Tags.Displace: invalid interface conversion
Call stack traceback locations:
0x7fcfb6fa806c 0x14fabfb 0x14e46d5 0x14ec14c 0x14d9204 0x14f2506 0x14dab2c 0x14db52a 0x138ee57 0x1f8d602 0x1f91137 0xc4f844 0xc4f2d4 0x7fcfb6b98099 0xc4f328 0xfffffffffffffffe

make[3]: Leaving directory '/home/taste/sc_sam_control_rd/work/samctrlrd/implem/default/QGenC'
make[3]: *** [Makefile:16: xmi/data_transfer_built] Error 1
make[2]: *** [Makefile.taste:69: samctrlrd] Error 2
make[1]: *** [Makefile:34: build-default] Error 2
make: *** [Makefile:38: debug] Error 2
make[2]: Leaving directory '/home/taste/sc_sam_control_rd/work/build'
make[1]: Leaving directory '/home/taste/sc_sam_control_rd/work'
16:29:33: The process "/usr/bin/make" exited with code 2.
Error while building/deploying project sc_sam_control_rd (kit: Taste_Kit)
When executing step "Custom Process Step"
16:29:33: Elapsed time: 01:13.
```

Figure 5 error report from building sam_ctrl_rd model

This error was encountered on the latest version of TASTE VM and even when reverting to a known commit, where N7S reported that the build worked. Therefore, we assume that the issue might be with our installation of the VM, however we are unable to confirm that.

The two models that worked as expected were tested on tiny extract of the dataset provided for SIL testing and their output was compared to reference SIL output. Both tests provided exact same numbers in two of three monitored numbers. The third number was different due to different environment setup, specifically due to different constants values. Only this number was modified by them, hence the difference.

4.4.2.3. Evidence

Location of the built models can be found at *Teams/WP3 – AOCS-GNC-generators/Files/TASTE_outputs*

4.5. Tools

4.5.1. OMQ01 – number of modelling tools for QGen code generation

4.5.1.1. Measurement

The result was calculated as the total number of tools to be used with the TASTE pipeline.

4.5.1.2. Results

The number of tools is **3**, specifically TASTE and MATLAB/Simulink and QGen

4.5.1.3. Evidence

Location of the built models can be found at *Teams/WP3 – AOCS-GNC-generators/Files/TASTE_outputs*

4.6. AdaCore's support

This chapter focuses on all KPIs related to communications with AdaCore, creators of the QGen tool. The communication with AdaCore started on 22.12.2020 and measurements were taken as of 11.5.2022, measuring the interval of roughly a year and a half. Results and data related to this chapter are stored in Annex A – Euclid KPI results, list *AdaCore*. All data for measurement were gathered via their bug/issue tracking website https://gt3-prod-1.adacore.com/.

4.6.1. OAS01 – number of support requests to AdaCore

4.6.1.1. Measurement

AdaCore themselves separate tickets on their website to closed and opened with number of tickets in each section and their classification was used here.

4.6.1.2. Results

There were **25** tickets sent via the bug reporting interface to date. 22 of them were closed to date, 3 were still open.

4.6.1.3. Evidence

Refer to chapter 4.6 introduction.

4.6.2. OAS02 – AdaCore's response time for support requests

4.6.2.1. Measurement

For each individual issue/bug time delta was calculated between the initial message and human response. Automatic responses sent immediately after the initial message were not counted.

4.6.2.2. Results

Out of the 21 analysed tickets, average was counted at 454.81 minutes, so it takes roughly **7.58 hours** for reply from the AdaCore's team.

4.6.2.3. Evidence

Refer to chapter 4.4 introduction.

4.6.3. OAS03 – number of issues and bugs sent to AdaCore

4.6.3.1. Measurement

There were 3 tickets classified as minor bugs in the ticket system titled: failure in QGen Verifier reporter, Problem with pointers as input parameters in S-functions and Problem with structure as Model argument input.

4.6.3.2. Results

There was also one ticket classified as major bug titled Problems with Qgenc generated code. So, the result is **3** minor bugs, **1** major bug and **17** issues, meaning tickets not classified as bugs.

4.6.3.3. Evidence

Refer to chapter 4.4 introduction.

4.6.4. OAS04 – number of days to solve a bug by AdaCore

4.6.4.1. Measurement

Time to close ticket was defined as time from ticket opening until fix was released in the wavefront version. The difference in time was then converted to days. Time measured for the major bug ticket was from initial post to post with confirmation of success from the customer team, as there was no clear announcement, that new wavefront was released.

4.6.4.2. Results

The average time to solve a bug by AdaCore is **17,177** days.

4.6.4.3. Evidence

Refer to the introduction chapter (section 4.6).

4.7. Effort

Comparison of spent effort on generated code versus manual coding was provided by SAE. Work time was measured on several SAE internal projects during each phase (Modelling, SW development and SW testing). Total

effort spent on a project was estimated and projects of similar complexity compared. The following are some relevant data taken from Table 8 of [RD12]:

Project	LoC	AOCS/GNC SW Development (Hours)	Testing Phase (Hours)	Complexity	Remarks
Euclid	56000	11.710	6.490	High	Autocoding
Hershel	41000	25.370	10.148	High	Manual coding
VNE (NAVIGA)	3600	5.003	2.001	High	SW Category Level A / SENER: Library SW development
AFTU/ERIS	10.000	2.040	816	Medium	Autocoding based on Euclid coding rules; reuse from VNE (models and simulator)

Table 5 Efforts and complexity

For the detailed process description, internal projects summary and comparison per phase see the full study at [RD12].

Effort Allocation	Manual Code	Autocoding
Management (PM+QM+RAMS)	10%	7%
Spec + Architecture	20%	0%
Detailed Design + Coding + Unit testing	40%	12%
Integration + Validation	30%	21%
Total	**100%**	**40%**

Table 6 Efforts allocation per Phase

4.7.1. DPI01 – reduction of development effort

4.7.1.1. Measurement

The total reduction in development effort was measured as percentile difference between the estimated auto coding effort and the manual coding effort, normalized at 100%. A comparison of an estimation in allocation of efforts to the SW Development & validation processes is displayed in the previous table.

4.7.1.2. Result

Therefore, the reduction of development effort is

reduction of development effort = (1.0 – 0.4) / 1.0

which is **60%.**

4.7.1.3. Evidence

For more information refer to the study in [RD12].

4.7.2. DPI02 – reduction in testing time

4.7.2.1. Measurement

The reduction in testing time was calculated as normalized percentile difference between the effort allocated on Integration + Validation section from Table 6 from the previous chapter.

4.7.2.2. Result

The reduction in testing time is therefore

$$\text{reduction in testing time} = (0.3 - 0.21) / 0.3$$

which is **30%.**

4.7.2.3. Evidence

For more information refer to the study in [RD12].

4.8. Models

Let's clarify some terms used in this chapter. A basic block is defined as a normal Simulink block that is part of the internal Simulink library or another external one, i.e., the gain block. A Simulink element is defined as a basic block with connections between them (signals).

4.8.1. ORS01 – percentage of modified blocks for QQen compatibility

4.8.1.1. Measurement

The percentage of modifications of model was measured as number of modified blocks divided by total number of blocks in model.

4.8.1.2. Result

Euclid models had 184 out of total 5460 modified. Total number of modified blocks in all models was **3%**. The number of modified blocks and the percentage of modified model can be found in Annex B in columns *Modified blocks* and *% Model modified* respectively for each individual model.

4.8.1.3. Evidence

Evidence can be found at [RD07].

4.8.2. DQA01 – number of models used in project

4.8.2.1. Measurement

For Euclid models, number was determined as number of sub models/sub tests of the main three models (SAM, FPMRCS, OCM).

4.8.2.2. Result

Total number of these sub models is **33**. Complete list of models and sub models for the main triplet can be found in Annex B.

4.8.2.3. Evidence

Evidence can be found at [RD07].

4.8.3. DQA02 – number of Simulink elements

4.8.3.1. Measurement

Number of Simulink elements was gained as a sum of number of basic blocks in each model added to the number of signals in said model. For definition of basic block see 4.8.

4.8.3.2. Result

The total number of Simulink elements in the models is **6005**. The number for each individual model can be found in Annex B – Models adaptation metrics, column *Number of elements*.

4.8.3.3. Evidence

For details about the used Euclid models, see [RD07]. Those models are proprietary. For more details contact SENER Aeroespacial.

4.8.4. DQA03 – effort to develop Simulink models

4.8.4.1. Measurement

Number of hours required to develop models in 4.8.2.

4.8.4.2. Result

The effort to create Euclid models was 5220 hours. The following adaptation of the models to QGen took 410 hours.

Data can be found in the Annex A.

4.8.4.3. Evidence

Data available at internal project control facility for SENER AE.

4.8.5. DQA004 – maximum subsystem depth

4.8.5.1. Measurement

For a subsystem depth measurement, standard MATLAB metrics for measuring subsystem depth were utilized.

4.8.5.2. Result

Maximum subsystem depth is **5** for model *fpmrcs_nm_gui_ls_gui*. Subsystem depth for each individual model can be found at Annex B – Models adaptation metrics, column *Subsystem depth*.

4.8.5.3. Evidence

For details about the used Euclid models, see [RD07]. Those models are proprietary. For more details contact SENER Aeroespacial.

4.8.6. DQA005 – maximum number of basic Simulink blocks

4.8.6.1. Measurement

For definition of basic Simulink block see 4.8.

4.8.6.2. Result

The maximum number of basic Simulink blocks is **1386** blocks for *fpmrcs_nm_gui_ls_gui* model. Only two models contained over 1000 blocks, third largest had 312 blocks, rest had less than 300 blocks.

Number of basic Simulink blocks for each individual model can be found in Annex B – Models adaptation metrics, column *Number of basic Simulink blocks*.

4.8.6.3. Evidence

For details about the used Euclid models, see [RD07]. Those models are proprietary. For more details contact SENER Aeroespacial.

4.8.7. DQA006 – maximum number of nested bus structures

4.8.7.1. Measurement

For number of nested bus structures, definition of the global bus definition is looked up. In addition, the nested bus structure directly relates with the maximum nested structure variable definition of the parameters.

4.8.7.2. Result

The maximum number of nested bus structures is **4**. This value was reached by two models, *fpmrcs* and *ocm* model. Number of nested bus structures for each individual model can be found in Annex B – Models adaptation metrics, column *Number of nested bus structures*.

4.8.7.3. Evidence

For details about the used Euclid models, see [RD07]. Those models are proprietary. For more details contact SENER Aeroespacial.

4.9. Code Metrics

SourceMonitor was used for gathering the code metrics with following parameters. A project was opened with following settings:

- Code language: C; File extension: *.c, *.h
- Use standard complexity metric
- Do Not count blank lines
- Ignore continuous Header and Footer comments

4.9.1. DQA007 – code cyclomatic complexity

4.9.1.1. Measurement

The maximum number of nested statements in a function is calculated via the default option in SourceMonitor. They are calculated as defined in the book Code Complete, Microsoft Press, 1993, p.395 by Steve McConnell. By

default, complexity is one. Each branch statement such as if, else, for, while add one complexity, ternary operators add one complexity. Each condition in if statement adds one complexity. Switch statements add one complexity for each exit from a case and one for default case even when none is present. Each catch or except statement in a try block adds one complexity as well.

4.9.1.2. Results

Average complexity in the models is **7.68**. Result is stored in [RD03] in Annex B list *QGEN_AUTOGENSW_1.0.0-baseline*.

4.9.1.3. Evidence

Generated data from Euclid models are stored in [RD03] in Annex B.

4.9.2. DPI005 – maximum number of nested statements in a function

4.9.2.1. Measurement

Maximum number of nested statements in a function is the maximum nested block depth level found. At the start of each file the block level is zero. Depths up to 9 are recorded and all statements at deeper levels are counted as depth 9. This is indicated by the "9+" label for the deepest level.

4.9.2.2. Results

The maximum number of nested statements in the generated models is **6**. Result is stored in [RD03] in Annex B list *QGEN_AUTOGENSW_1.0.0-baseline*.

4.9.2.1. Evidence

Generated data from Euclid models are stored in [RD03] in Annex B.

4.9.3. DPI006 – number of statements

4.9.3.1. Measurement

In C, computational statements are terminated with a semicolon character. Branches such as if, for, while, and goto are also counted as statements. Preprocessor directives #include, #define, and #undef are counted as statements. All other preprocessor directives are ignored. In addition, all statements between an #else or #elif statement and its closing #endif statement are ignored, to eliminate fractured block structures. This metric counts all the statements in a file, or in all files in a checkpoint.

4.9.3.2. Results

There are **29541** statements in the generated models. Result is stored in [RD03] in Annex B list *QGEN_AUTOGENSW_1.0.0-baseline*.

4.9.3.3. Evidence

Generated data from Euclid models are stored in [RD03] in Annex B.

4.9.4. DSQ001 – comment frequency within the generated functions

4.9.4.1. Measurement

Comment frequency takes all lines with C style (/* … */) or C++ style (// …) comments and divides their number by number of all lines. If option ignore headers and footers is checked, they are not counted, blank lines are not counted as well if ignore blank lines is checked.

4.9.4.2. Results

The rate of comments in the generated code is **27.4%**. Result is stored in [RD03] in Annex B list *QGEN_AUTOGENSW_1.0.0-baseline*.

4.9.4.3. Evidence

Generated data from Euclid models are stored in [RD03] in Annex B.

4.9.5. DPI007 – number of lines of generated code per function

4.9.5.1. Measurement

Calculated in similar way as 4.7.3.1. but only for a function.

4.9.5.2. Result

Average number of statements of generated code per function is **28.2**. Result is stored in [RD03] in Annex B list *QGEN_AUTOGENSW_1.0.0-baseline*.

4.9.5.3. Evidence

Generated data from Euclid models are stored in [RD03] in Annex B.

4.10.Coverage

All the coverage data in this chapter were obtained by running the tests on a Debian virtual machine. The QGen generated code was run on the machine for SIL testing, and on via TSIM2 emulator on the machine for PIL testing. The top models (sam, fpmrcs, ocm) were run with QGen version 23, other with version 22. Data were gathered via gcov and lcov tools, transferred to html reports using genhtml tool and processed.

For a detailed description of the testing process, please refer to [RD03], section 5.2.1, for details on PIL testing process, check the document [RD03], section 5.3.

4.10.1. DSQ002 – coverage % of branches during SIL

4.10.1.1. Measurement

Branch coverage is a percentage of branch taken from all branches in the code. For each model, only specific previously defined files were monitored, and their branch coverage was averaged. For detailed procedure of tools involved refer to 4.8.

4.10.1.2. Result

Average branch coverage for all test cases is **96.14%**. The averaged coverage for each test case can be found in Annex A, list *DYN stat.*, columns *% statement coverage (measured)*. Relevant files for which coverage was measured for each SAE test scenario and their values can be found in [RD03], Annex C.

4.10.1.3. Evidence

HTML reports for all tests are available on *Teams/WP3 – AOCS-GNC Code Generator/Files/SIL/COV-RP.zip*. Relevant data for this section from these reports are also stored in Annex A, list *DYN Stat.*

4.10.2. DSQ003 – coverage % of function statements during SIL

4.10.2.1. Measurement

Function statement coverage is a percentage of statements hit from all the statements in the code. For each model, only specific previously defined files were monitored, and their branch coverage was averaged. For detailed procedure of tools involved refer to 4.8.

4.10.2.2. Result

Average statement coverage for all test cases is **96.11%**. The averaged coverage for each test case can be also found in Annex A, list *DYN stat.*, columns *% branch coverage (measured).* Relevant files for which coverage was measured for each SAE test scenario and their values can be found in [RD03], Annex C.

4.10.2.3. Evidence

HTML reports for all tests are available on *Teams/WP3 – AOCS-GNC Code Generator/Files/SIL/COV-RP.zip*. Relevant data for this section from these reports are also stored in Annex A, list *DYN Stat.*

4.10.3. DSQ004 – coverage % of branches during SIL

Duplicate of DSQ002.

4.10.4. DSQ005 – coverage % of function statements during PIL

Given that the monitored files were not affected in any way during the SIL->PIL transformation, we assumed the same results as for SIL, since the test scenarios are the same.

4.11. Tests

All the testing data in this chapter were obtained by running the tests on a Debian virtual machine. The QGen generated code was run on the machine for SIL testing, and on via TSIM2 emulator on the machine for PIL testing. The top models (sam, fpmrcs, ocm) were run with QGen version 23, other with version 22.

For a detailed description of the testing process, please refer to [RD03], section 5.3, for details on PIL testing process, check the document [RD03], section 5.3.

4.11.1. DPI003 – error tolerance in the MIL

4.11.1.1. Measurement

Values obtained during runs in MATLAB/Simulink environment. For more information, please refer to [RD05].

4.11.1.2. Results

All MIL error tolerances are **0.00E+00**. MIL error tolerance results for each test are in the table from Annex A, list *DYN stat.* column *DPI003 – MIL tolerance (with respect to Euclid; see D3.7)*

4.11.1.3. Evidence

See [RD05], section "3.1.2 Reference Test", "Figure 3: Comparison graphs modified vs unmodified models". For details about the used Euclid models, see [RD07]. Those models are proprietary. For more details contact SENER Aeroespacial.

4.11.2. DPI004 – error tolerance in the MIL-SIL

4.11.2.1. Measurement

The software evaluation metric with respect to the numerical threshold is set to 1e-15 between MIL and SIL. Any larger numerical difference will result in the test to be considered FAIL, while if differences are below this value, the test is considered PASS. The numerical threshold that was set internally in the SIL execution test was set to 0. This is due to ensure that all numerical discrepancies are collected and evaluate the tolerance of the tool.

4.11.2.2. Results

All tests **passed** with required tolerance. For details on which tests passed with which tolerance, see Annex A, list *DYN stat.* column *SIL tolerance (see D3.10)* under SAE section. For comments and more details on results, check [RD03], section 4.1.

4.11.2.3. Evidence

A zip file named MIL-SIL comparison is attached to D3.10 Software Verification Report [RD03], It contains all the results from the SIL campaign, i.e., MIL-SIL comparison and SIL results.

4.11.3. DSQ006 – SIL test without error

4.11.3.1. Measurement

Each test that ran correctly within specified threshold was deemed successful. Even some tests with early errors (due to uninitialized values) were deemed as successful if justification was right.

4.11.3.2. Results

Due to several issues with various models during our SIL testing, data from [RD03], section 4.1 SIL Tests Report were used. Comments where specific model failed and the result of running the model can be found in Annex A, list *DYN stat.* column *SIL test status* under ESC section.

4.11.3.3. Evidence

A zip file named MIL-SIL comparison is attached to D3.10 Software Verification Report [RD03], It contains all the results from the SIL campaign, i.e., MIL-SIL comparison and SIL results.

4.11.4. DSQ007 – PIL test without error

4.11.4.1. Measurement

All the testing data in this chapter were obtained by running the tests on a Debian virtual machine utilizing the TSIM2 emulator for PIL testing. The top models (sam, fpmrcs, ocm) were generated with QGen version 23, other with version 22. For more details on the test machine setup see [RD03], section 5.3.

4.11.4.2. Results

All models except *fpmrcs* and *fpmrcs_mm* passed with zero tolerance. The two that didn't, passed partially with tolerance of 1.00E-15. The reason they passed only partially is due to having some errors during the first iterations of the model. Relevant data for this section from these reports are also stored in Annex A, list *DYN Stat* columns *PIL tests status* and *PIL tolerance*.

4.11.4.3. Evidence

Results of the runs for each individual test and its scenarios are available on *Teams/WP3 – AOCS-GNC Code Generator/Files/PIL/test/test_name/scenario_name/log/*.results*.

4.12. Euclid KPI data Summary

The following table provides quick summary of data above. Specifically, procedures used to gather result for the KPI, where data from measurement is located and where evidence/proof is located.

ID	Name	Result	Procedure description	Evaluation result	Evidence location
OIQ01	Integration: Other_SW (Qgen code)	True	Observable characteristics of code	Annex A – Euclid KPI results, DYN stat. list	Teams/WP3/PIL/test
OIQ02	Inclusion: Qgen code (Other_SW)	True	See section 4.3.2.1	Annex A – Euclid KPI results, DYN stat. list	Teams/WP3/PIL/ /test/fpmrcs_mm
OIQ03	Qgen integration into TASTE	True	Check if any Simulink model was buildable with QGen from TASTE	4.4.1.2	Teams/WP3/TASTE_output
OIQ04	number of Simulink models integrated into TASTE	3	Number of models that were successfully built on our VM	4.4.2.2	Teams/WP3/TASTE_output
OMQ01	number of modelling tools for Qgen code generation	3	Number of tools necessary to build Qgen code with the TASTE pipeline	4.5.1.2	Teams/WP3/TASTE_output
OAS01	number of support requests to AdaCore	25	Number of tickets on the ticket website	Annex A – Euclid KPI results, AdaCore list	https://gt3-prod-1.adacore.com/
OAS02	AdaCore's response time for support requests	7.58 hours	Time delta between opening and closing the ticket	Annex A – Euclid KPI results, AdaCore list	https://gt3-prod-1.adacore.com/
OAS03	number of issues and bugs sent to AdaCore	3 minor, 1 major bug, 17 issues	Number of issues and bugs sent	Annex A – Euclid KPI results, AdaCore list	https://gt3-prod-1.adacore.com/
OAS04	number of days to solve a bug by AdaCore	avg. 17,177 days	Time between opening ticket and message confirming solving bug	Annex A – Euclid KPI results, AdaCore list	https://gt3-prod-1.adacore.com/

ID	Name	Result	Procedure description	Evaluation result	Evidence location
DPI01	reduction of development effort	60%	Difference between estimated efforts	Table 6	[RD12]
DPI02	reduction in testing time	30%	Normalized difference between estimated efforts	Table 6	[RD12]
ORS01	percentage of modified blocks for Qgen compatibility	3%	Percentage of modified blocks for Qgen compatibility	Table 8	Annex B – Models adaptation metrics
DQA01	number of models used in project	33	Number of Qgen models used in project development	Table 9	Annex B – Models adaptation metrics
DQA02	number of Simulink elements	6005	Sum of numbers and signals in used models	Table 8	Annex B – Models adaptation metrics
DQA03	effort to develop Simulink models	5630 / 410 hours	Time development of Simulink models	Overview list	Annex A – Euclid KPI results,
DQA004	maximum subsystem depth	5	Maximum subsystem depth measured by MATLAB metrics	Table 8	Annex B – Models adaptation metrics
DQA005	maximum number of basic Simulink blocks	1386	Maximum number of basic Simulink blocks.	Table 8	Annex B – Models adaptation metrics
DQA006	maximum number of nested bus structures	4, 0	Maximum number of nested bus structures.	Table 8	Annex B – Models adaptation metrics
DQA007	code cyclomatic complexity	7.68	Code cyclomatic complexity measured through SourceMonitor	D3.10 Annex B	D3.10-SVR-AnnexB.

ID	Name	Result	Procedure description	Evaluation result	Evidence location
DPI005	maximum number of nested statements in a function	6	Maximum number of nested statements in a function measured with SourceMonitor	D3.10 Annex B	D3.10-SVR-AnnexB.
DPI006	number of statements	29541	Number of statements.	D3.10 Annex B	D3.10-SVR-AnnexB.
DSQ001	comment frequency within the generated functions	27.4%	Proportion of comments within the generated functions.	D3.10 Annex B	D3.10-SVR-AnnexB.
DPI007	number of lines of generated code per function	28.2	Number of lines of generated code per function (including comments but not including blank spaces)	D3.10 Annex B	D3.10-SVR-AnnexB.
DSQ002	coverage % of branches during SIL	96.14%	Coverage % of branches during SIL unitary test using gcov	D3.10 Annex C	Teams/WP3/SIL/COV-RP.zip
DSQ003	coverage % of function statements during SIL	96.11%	Coverage % of function statements during SIL unitary test using gcov	D3.10 Annex C	Teams/WP3/SIL/COV-RP.zip
DSQ004	coverage % of branches during SIL	Duplicate of DSQ002	-	-	-
DSQ005	coverage % of function	Similar to DSQ003	Coverage % of function statements during PIL unitary test using gcov	N/A	Teams/WP3/SIL/COV-RP.zip

ID	Name	Result	Procedure description	Evaluation result	Evidence location
	statements during PIL				
DPI003	error tolerance in the MIL	0.0E+00	Error tolerance in the MIL validation environment with respect to the reference Euclid models.	D3.7.2	D3.5
DPI004	error tolerance in the MIL-SIL	1.0E-15	Error tolerance in the MIL-SIL validation environment with respect to the MIL reference values	D3.10 section 4.1	D3.10 MIL-SIL comparison zip
DSQ006	SIL test without error	100%	Percentage of exercised SIL test without error execution.	D3.10 section 4.1	Teams/WP3/SIL/COV-RP.zip
DSQ007	PIL test without error	100%	Percentage of exercised PIL test without error execution.	D3.10 section 5.2.1	Teams/WP3/PIL/test

Table 7 KPI results summary

5. UPMSAT2 KPI data

Another significant model in assessing the TRL evaluation of the QGen tool-set is the UPMSat-2 mission. UPMSat-2 is a micro-satellite designed and developed by the IDR and STRAST research groups at the Universidad Politécnica de Madrid (UPM) and it is in orbit since September 2020.

UPMSat-2 was developed for educational and technology demonstration purposes. Consequently, it includes several experiments and subsystems such as the Attitude Control System (ACS) designed and validated by aerospace and software engineers using the MATLAB and Simulink modelling tools. Specifically, The Simulink Embedded Coder tool was used to transform the Simulink models into source code for later integration into the OBSW. The validation and verification from these models and its autogenerated code was performed with additional MATLAB and Simulink tool-boxes.

This evaluation is based on the Simulink models used for the UPMSat-2 ACS subsystems. These models were taken as reference inputs for the QGen tools. In particular, the QGen code generator was used for the model to code transformation and QGen Debugger for the SIL validations.

5.1. Introduction to the UPMSat-2 Attitude Control System

The UPMSat-2 ACS is in charge of the satellite's attitude determination based on the magnetic interaction with the Earth's magnetic field and the one produced though magnetic torquers. The ACS sets the satellite rotation rate controlled with a constant angular speed, and also maintains the vehicle's attitude perpendicular to its orbit plane, thereby the communication antenna is properly oriented to the Earth.

The satellite is equipped with three magnetometers (MGM) to measure the Earth's magnetic field, each one measuring in the three axes. Three magnetorquers (MGT), one per axis, are used to generate the required torque for attitude control.

Figure 6 UPMSat-2 ACS software architecture depicts the high-level architecture from the ACS software component, which includes three tasks and three shared resources accessed in mutual exclusion to allow the communication between the real time tasks. The implementation corresponds to the classical control cycle: sense, compute, and actuate. These three steps are decoupled in three different tasks: Measurer, Control, and Actuator, respectively.

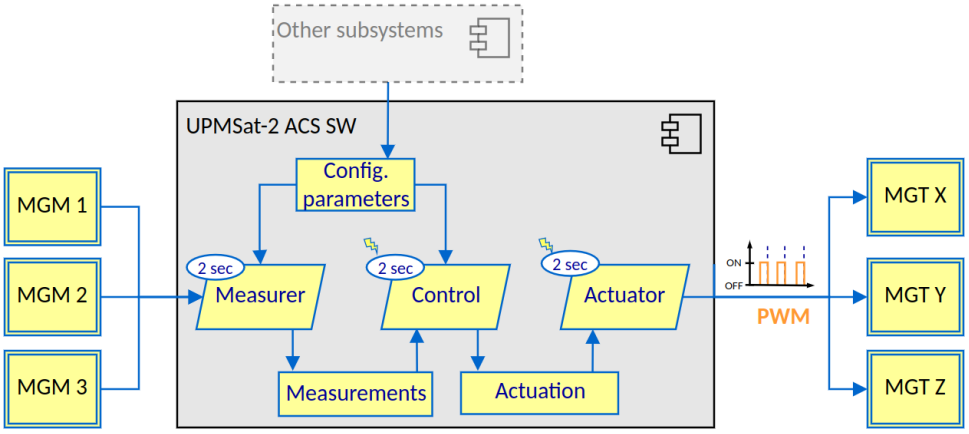

Figure 6 UPMSat-2 ACS software architecture

Measurer is a periodic task that reads five magnetic field measurements from the three magnetometers and calculates their average. These values are then passed to the Control sporadic task though the Measurements protected object.

The Control task imports the autogenerated code from the "Control" Simulink model, explained in the next subsection. It first reads the measurements and configuration parameters to the auto-coded control algorithm and then passes the results to the Actuator sporadic task by means of the Actuation protected object.

Finally, the Actuator takes the actuation values for each axis and behaves accordingly. Such values represent the Pulse Width Modulation (PWM) duty cycle for the three magnetorquers.

The behaviour of the Measurer and Control tasks is based on the configuration parameters contained in the protected object under the same name. These parameters include the setpoint and calibration values for the control algorithm and data acquisition. This allows external subsystems, such as TMTC (Telemetry and Telecommand) or the OBSW manager, to configure the ACS behaviour dynamically.

5.2. UPMSat-2 ACS Simulink and TASTE models

This section is concerned with the Simulink models used for the control algorithm, and the TASTE project that integrates the ACS architecture making use of its modelling elements. This integration serves as an evidence for KPIs OIQ03 and OIQ02 discussed at section 5.3.

The software architecture for the ACS, presented in section 5.1, is based on the workflow from the MIL Simulink models illustrated in Figure 7. The Sensor task performs the activities modelled in the Sensor block, i.e.: it reads the magnetometer values periodically based on configurable parameters (green boxes). This relationship is applicable also for the Control task and Algorithm block in the middle, and the Actuator task and PWM block on the right.

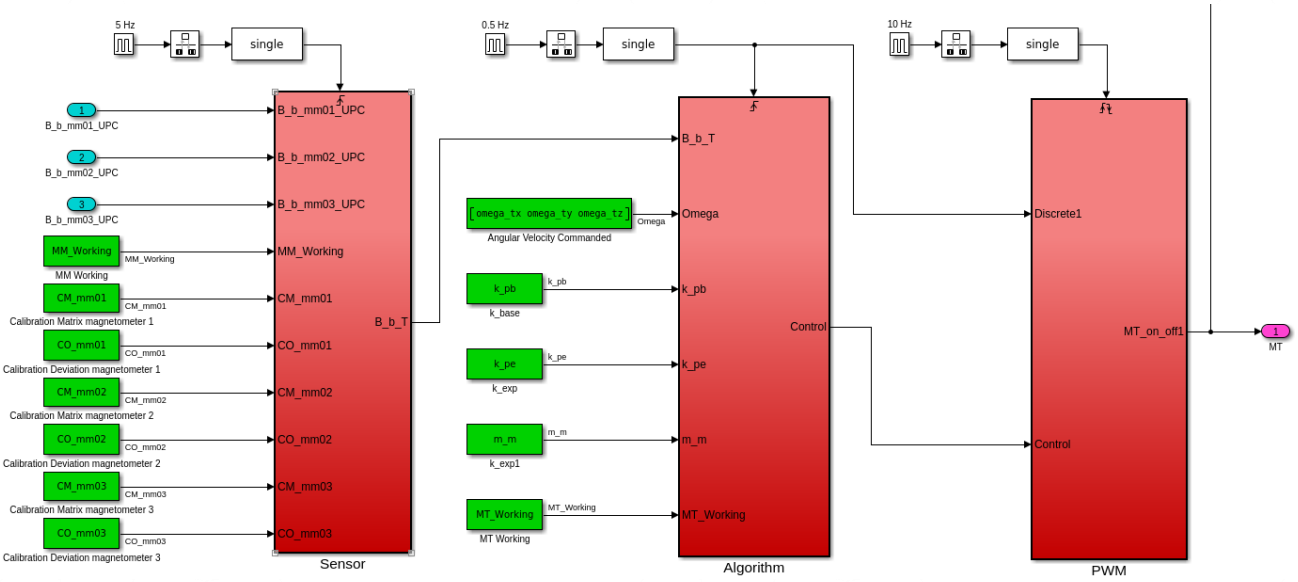

Figure 7 MIL model for the UPMSat-2 ACS

Algorithm is the most interesting block for this assessment report because it has been integrated into the UPMSat-2 OBSW with auto-coding tools, Embedded Coder. In this use case, we use QGen instead to autogenerate the C source code from the Algorithm block. The autogenerated code is also verified with QGen Debugger, which allows us to compare the difference between the MIL and SIL executions.

Figure 8 illustrates the internal elements from the Algorithm block, which conform a functional chain to process the raw measurements and compute the required actuation. The Control law follows the mathematical equation to compute the magnetic torquers. Then, these values are passed as the M output signal to If Fail Recalculate, an FDIR module that behaves accordingly to the current operative magnetometers (MT_Working input parameter).

Finally, these values are passed through for the **limitation** and **discretization** steps to transform the computed torquers to digital values (PWM duty cycles in milliseconds).

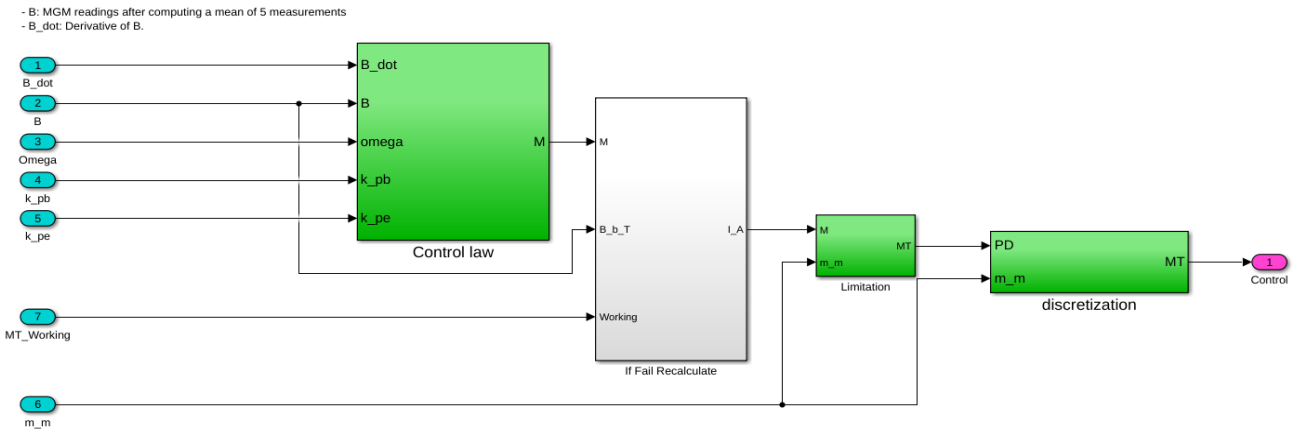

Figure 8 UPMSat-2 Attitude control algorithm implemented in Simulink

As discussed above, our main concern for this use case is the ACS algorithm block. Consequently, we integrated this model into a TASTE project taking advantage of the improvements and fixes applied to the TASTE tools (namely: KAZOO, Data Modelling Tools, and Space Creator) during the development of AURORA WP5. This integration mimics the ACS behaviour, although this has been achieved with a sequential approach rather than concurrently with interconnected tasks.

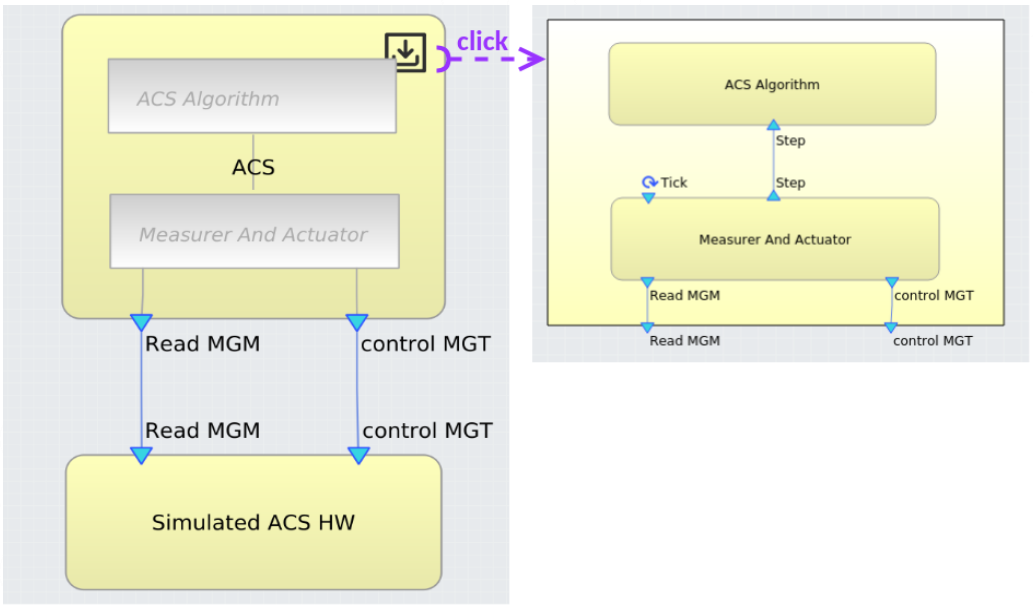

Figure 9 TASTE Interface View for the UPMSat-2 ACS

Figure 9 shows the interface view from the ACS project in TASTE. The project is divided into two TASTE functions: **ACS** at the top and Simulated **ACS HW** at the bottom. The latter is a C function that communicates with a Simulink model similar to the one shown in Figure 8, but with the following modifications:

- The control block was removed, the goal is to import it into the TASTE model (explained later).
- The outputs from the Sensor block were connected to an asynchronous **send** TCP/IP Simulink block.
- The inputs from the PWM block were connected to an asynchronous **receive** TCP/IP Simulink block.

Thanks to this configuration, we could communicate our modified Simulink model (client) to the TASTE project (server) by means of TCP/IP sockets. This modified model is referred to as SIL hereafter since it allows us to execute the ACS algorithm independently from the Simulink environment. In addition, the Simulated ACS HW component offers two unprotected interfaces: Read MGM to receive the simulated readings sent by the SIL environment, and control MGT to send the calculated magnetorquer commands.

On the other hand, ACS acts as a composite TASTE function. This means that it serves as a façade or container for other TASTE functions. Specifically, ACS is composed of ACS Algorithm and Measurer And Actuator as depicted in the upper left corner of Figure 9. ACS is also connected to the two interfaces provided by Simulated ACS HW, such connections are also visible to its two subcomponents as "required interfaces".

Regarding the Measurer and Actuator, it gathers the three steps discussed earlier in section 5.1 into a single element. In consequence, it must orchestrate and set the peace for these three activities, which is implemented with the Tick cyclic interface. This interface is executed periodically by a dedicated thread. However, it does not include the Control block because this activity is deferred to the ACS algorithm function.

ACS algorithm is a *QGenC* TASTE function. This means that it receives a Simulink model as the implementation language and automatically generates C source code taking this model as input. In addition, the TASTE toolchain facilitates the integration with QGen creating more elements such as: the code to interconnect the autogenerated C code with the TASTE middleware, scripts to map the ASN.1 data types (defined in the data-view) into Simulink data types, and Makefiles to automate the code generation process.

The connection between ACS algorithm and Measurer and Actuator is possible thanks to the Step connection. Further information about this TASTE project has been documented in the guide for the QGen integration into Space Creator, described in detail at [RD08].

5.3. Code interfaces

5.3.1. OIQ01 – Integration: Other_Sw(QGen_Code)

5.3.1.1. Measurement

The objective of this indicator is to evaluate the integrability of QGen autogenerated code into existing software architectures. This enables the interoperability of QGen with other modelling tools and frameworks like TASTE, F Prime, or CFs; and also enhances the applicability of QGen in the industry.

This is measured testing Simulink projects that test this functionality and analysing the autogenerated code. The integrability of QGen mainly depends on its code generation strategy, but also on the tools and options that it offers to automate this process.

5.3.1.2. Results

Code generators like Embedded Coder or QGen offer a well-defined interface to allow communication with other systems or tools. For instance, the TASTE tool-chain supports software elements implemented in Simulink. To do so, it generates code wrappers that enable the integration and interoperability of the code generated by Embedded coder. This is possible because Embedded Coder follows patterns dependent on a "function signature" that is also known by the TASTE code generators.

In general, the "TASTE/Simulink" approach can be followed. In the same way, existing software architectures would only need an "adapter" module that, in the lowest level, invokes the generated C/Ada code by QGen. This way, the autogenerated code is deferred from the core modules that contain the main logic.

QGen autogenerated code offers one subprogram that implements the Simulink model behaviour, and contains all the necessary data structures. The subprogram signature (name, formal parameters, and return type) is the only interface that the caller ("adapter" module) needs to know in order to work with it.

Therefore, it is **TRUE** that QGen is integrable into existing software architectures.

5.3.1.3. Evidence

The evidence for this KPI are the implementations of both the UPMSat-2 ACS [RD09] and UPMSat-2 OBDH [RD10] in TASTE/Space Creator.

5.3.2. OIQ02 – Inclusion: QGen_Code(Other_Sw)

5.3.2.1. Measurement

Like KPI OIQ01 (section 5.3.1), this KPI aims to evaluate the applicability and integrability of QGen in existing software applications, but from a bidirectional perspective, i.e.: evaluate the ability of QGen to reuse other software elements. This is measured testing Simulink projects that test this functionality.

5.3.2.2. Results

Currently, MATLAB offers *S-functions* blocks which allow Simulink models to import Matlab scripts, C, and Ada source code. QGen supports the usage of the S-Functions blocks and many demonstration models/programs have been developed to probe this functionality.

For instance, the figure depicted below illustrates a basic S-Function that performs the addition of two 32-bit integers. The autogenerated code from that model was obtained with QGen and successfully tested and integrated into the TASTE toolchain. Some issues were reported (and fixed by Ada Core) during this integration process, especially those related to pointer types in the formal parameters.

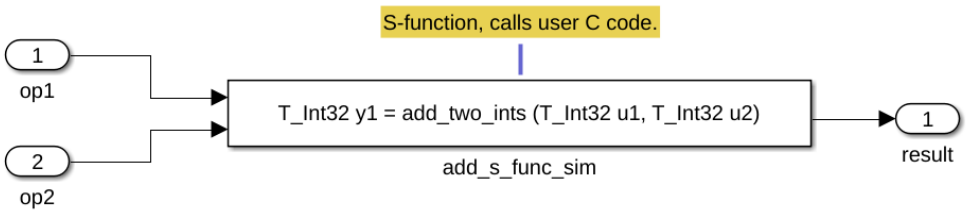

Figure 10 Demo model for an S-Function with QGen

Therefore, it is **TRUE** that QGen can include external software elements.

5.3.2.3. Evidence

The evidence for this KPI are the TASTE projects and conventional (non-TASTE) projects that use S-Functions and QGen for the code generation, these projects are not publicly available, contact the STRAST-UPM group for further information.

5.4. Taste

5.4.1. OIQ03 – Qgen integration into TASTE

5.4.1.1. Measurement

As the name implies, this KPI aims to evaluate the integration of QGen into the TASTE toolchain. This requires the TASTE wrappers to understand the functions signature generated by QGen, and also to specify and know input and output parameters passing strategy. These two aspects conform an interface implemented by the QGen autogenerated code, and required by the TASTE autogenerated code.

5.4.1.2. Results

Currently, TASTE supports QGenC components but with a restricted component model, they can contain only synchronous provided interfaces and no required interfaces. These restrictions apply also to the current Simulink integration. In fact, the QGen/TASTE integration developed during the AURORA project is based on the Embedded Coder/TASTE integration.

UPMSat-2 ACS model was successfully integrated into a TASTE project using QGenC as the code generator, this is explained in detail in section 5.2.2. The core features implemented while doing this integration are:

- *Automatic source code generation from QGen TASTE functions*. Previously, the user had to manually invoke QGenC; now TASTE invokes it automatically in such a way that the configuration for the generated code could match the TASTE wrappers interface.
- *Support for editing Simulink components from Space Creator*. This includes QGenC and QGenAda functions and allows the user to edit the underlying model in a similar way to SDL (Specification and Description Language) models.
- *Improvements in the code generation process*. Specifically, we restricted the code generation only when necessary. This is when the underlying Simulink model has been changed.

Therefore, it is TRUE that QGen is integrable into TASTE, but with some limitations.

5.4.1.3. Evidence

The features mentioned above are merged in the TASTE repository. The implementations of both the UPMSat-2 ACS [RD09] and UPMSat-2 OBDH [RD10] in TASTE/Space Creator demonstrate the usage of these features. The ACS integration is explained in section 5.2

5.4.2. OIQ04 – number of Simulink models integrated into TASTE

5.4.2.1. Measurement

This KPI not only serves as an indicator, but also as an evidence for the KPI OIQ03 (c.f. section 5.4.1). If the result of KPI OIQ03 is TRUE, this indicator helps to evaluate the complexity of the process of integrating Simulink/Qgen applications into TASTE.

5.4.2.2. Results

Until now, we have successfully created 3 TASTE projects that make use of the Qgen generator:

1. UPMSat-2 ACS. This project includes the ACS Simulink model.
2. UPMSat-2 OBDH. This project incorporates the ACS model and the whole UPMSat-2 OBDH system.
3. Simple calculator. This project uses completely different Simulink model that includes an adder and multiplier block. Although it is a straightforward model, it helped us to identify issues.

Although, the first and second models use the same ACS Simulink model, we have created two different TASTE models, one more complex than the other. In addition, we have partitioned the whole ACS model into two subsystems for the measuring and actuation. Therefore, it can be said that:

3 models were successfully integrated into TASTE.

5.4.2.3. Evidence

The evidence for this KPI are the implementations of both the UPMSat-2 ACS [RD09] and UPMSat-2 OBDH [RD10] in TASTE/Space Creator. The simpler adder model is not publicly available yet. **The ACS integration was previously explained in section 5.2**

5.5. Tools

5.5.1. OMQ01 – number of modelling tools for QGen code generation

5.5.1.1. Measurement

This indicator aims to estimate the support provided by Qgen for its use in software applications. The support is measured in terms of the number of tools offered for the code generation process.

5.5.1.2. Results

Qgen offers three different way for the code generation process:

1. Via a combination of the system and the MATLAB interface (this method is used in TASTE).
2. Via the MatLab command Line interface.
3. Via the Simulink graphical user interface.

Therefore, there are **3 methods** to use the Qgen code generator.

5.5.1.3. Evidence

These methods are explained in detail in the QGen user manual. Figure 11 was taken from this manual and shows the three generation methods explained before.

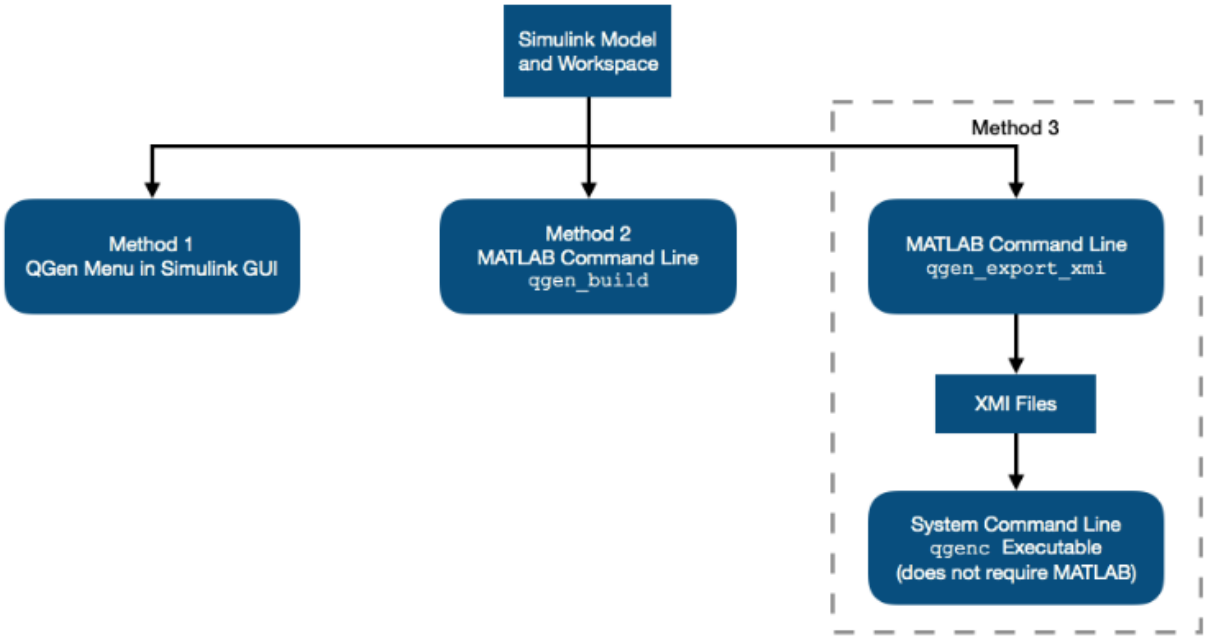

Figure 11 Code generation methods forQGen, taken from the QGen user manual

5.6. AdaCore's support

The indicators from this category are related to the support quality provided by AdaCore and they are common for both EUCLID and UPMSat-2 projects. Therefore, the KPI results presented in section 4.6 are applicable to this use case, too.

5.7. Effort

These indicators (DPI01 and DPI02) are not applicable to this use case. Much of the worked developed in this WP include the models developed during the UPMSat-2 project, which took years to conceive.

5.8. Models

5.8.1. ORS01 – percentage of modified blocks for QGen compatibility

5.8.1.1. Measurement

The percentage of modified blocks of our ACS model was measured following the next formula:

$$\%\, modified\ blocks = \frac{\#\, modified\ blocks}{\#\, blocks} \cdot 100$$

5.8.1.2. Result

As discussed earlier, only one model was used for the UPMSat-2 use case. This model contains 305 basic blocks, from which 7 were modified. All these changes were related to the usage of range expressions in selector blocks and did not the affect the MIL behaviour, but the generated code was incorrect, affecting the behaviour during the SIL testing. Therefore, a 2.29 % of the blocks were modified for QGen compatibility.

5.8.1.3. Evidence

The errors detected with the original model were reported in GNAT Tracker as a "major bug". Further information of this bug is gathered at tracking website https://qt3-prod-1.adacore.com/. However, it is available only to the members of the SENER Aeroespacial account.

Figure 12 shows a screen-capture of the reported bug title and creation date.

Figure 12 Reported bug related to the modified blocks in in the ACS model

5.8.2. DQA01 – number of models used in project

5.8.2.1. Measurement

This indicator is determined by the total number of Simulink models used for the UPMSat-2 ACS.

5.8.2.2. Results

the UPMSat-2 ACS was evaluated using the whole model against one simulation of the environment and equipment during a significant amount of time where the vehicle goes through different stages: detumbling, orientation, and stabilization. Consequently, UPMSat-2 consists only of one model for the ACS.

5.8.2.3. Evidence

The UPMSat-2 ACS model is publicly available at the STRAST research group GitHub repositories [RD09, RD10].

5.8.3. DQA02 – number of Simulink elements

5.8.3.1. Measurement

This indicator calculates the structural complexity of a Simulink model which is determined by the number of elements and the communication of these elements. To quantify this indicator, we have defined the following terms:

- Basic block: Simulink block that is either part of the Simulink library or from an external one. E.g.: the *gain* block.
- Connection: A signal that communicates two basic blocks through the input/output parameter pair.

The following trio of metrics help us to calculate the Complexity of a Simulink model:

Nb Number of basic blocks

Nc Number of connections between basic blocks

Ne Number of Simulink elements. $Ne = Nb + Nc$

5.8.3.2. Results

The original ACS model used currently in orbit contains **250** Simulink blocks. The modified model for the support of QGen uses the same number of blocks since it required few changes. These models contain **369** signals that allow communication between the blocks. Therefore, we have:

$$Ne = 250 + 369 = \mathbf{619}\ elements$$

5.8.3.3. Evidence

The UPMSat-2 ACS model is publicly available at the GitHub repository from the STRAST research group [RD11].

5.8.4. DQA03 – effort to develop Simulink models

5.8.4.1. Measurement

This KPI aims to estimate the time required to obtain deployable and production ready software from existing Simulink models. In this case we are considering the ACS model from UPMSat-2.

Note that, the spent time for the creation of the evaluated Simulink model is not considered for these reasons:

- The development and implementation of Simulink models are always performed with the *MathWorks* tools: Simulink and MatLab. The QGen toolset is of interest only during the model to code transformation, and "In-The-Loop" validation processes.
- The complexity of a Simulink model is already determined by other KPIs based on the internal structure and organization of such models. E.g., DQA02, DQA04, DQA05, and DQA06.
- The development of a Simulink model for the TRA is not within the objectives of AURORA, we proceed on the basis of previous projects (UPMSat-2 and EUCLID) which took a significant period of time for the creation and validation of its models.

5.8.4.2. Result

During the development of the UPMSat2 satellite, aerospace engineers designed the ACS models including those required for their validation. During the AURORA project, software engineers have taken these models to perform the following activities:

- Study and analyse the model internals.
- Study the QGen technologies including its code generator (QGenC) and verifier (QGen Debugger).
- Generate C code from these models and validate it.
- Throughout the generated code validation, some issues were found, which were reported to AdaCore and required few modifications in the actual model (see section 4.6).
- Develop TASTE projects to mimic the behaviour of the UPMSat-2 OBDH and integrate its ACS.

All these activities took approximately **5 person-months** to complete.

5.8.4.3. Evidence

The evidence for this indicator are the TASTE models, tutorials, and information videos published so far [[RD08], RD09, RD10, RD11].

5.8.5. DQA004 – maximum subsystem depth

5.8.5.1. Measurement

Simulink models are functionally divided into different systems, which in turn are composed of nested subsystems to ease readability and maintainability. The nesting levels are directly proportional to the complexity of the developed system. Therefore, this indicator is used to estimate the complexity of the evaluated Simulink models.

5.8.5.2. Result

The deepest Simulink blocks have **5 nesting levels** and correspond to an FDIR subsystem from the ACS when one magnetorquer is failing and two are working. The subsystem is located in the following absolute path:

Control/If Fail Recalculate/1 Fail/B Tot/Subsystem

5.8.5.3. Evidence

The UPMSat-2 ACS model is publicly available at the GitHub repository from the STRAST research group [RD11]. The maximum subsystem depth can be visually verified opening the Simulink model.

5.8.6. DQA005 – maximum number of basic Simulink blocks

5.8.6.1. Measurement

This KPI aims to estimate the complexity and maintainability of Simulink models based on the number of basic blocks. If there are more than one Simulink projects in a use case, the maximum of the set of maximum values is used.

5.8.6.2. Result

Two versions of the ACS models were used; the one that is currently used in orbit contains **250** Simulink blocks, and the modified version contains the same number of blocks.

5.8.6.3. Evidence

The UPMSat-2 ACS model is publicly available at the GitHub repository from the STRAST research group [RD11]. The maximum number of basic blocks can be verified with specific MATLAB commands.

5.8.7. DQA006 – maximum number of nested bus structures

5.8.7.1. Measurement

This KPI aims to estimate the complexity and maintainability of Simulink models based on the maximum number of nested bus structures.

5.8.7.2. Result

The UPMSat-2 ACS model does not contain any bus structure like bus creators or selectors. It only contains signals with basic types such as floating point, or sequence of floating-point elements. Therefore, the result for this KPI is: **0** nested buses.

5.8.7.3. Evidence

The UPMSat-2 ACS model is publicly available at the GitHub repository from the STRAST research group [RD11]. The maximum number of basic Simulink blocks can be visually verified opening the Simulink model.

5.9. Code Metrics

All these metrics help to evaluate the code quality generated by Qgen compared to traditional methods (hand-written code). All of them were analysed with the Source Monitor free software. The project was configured the same way as for the EUCLID analysis (c.f. section 4.9). Figure 13 shows the Kiviat diagram for most of the metrics discussed below, such as the average and maximum complexity.

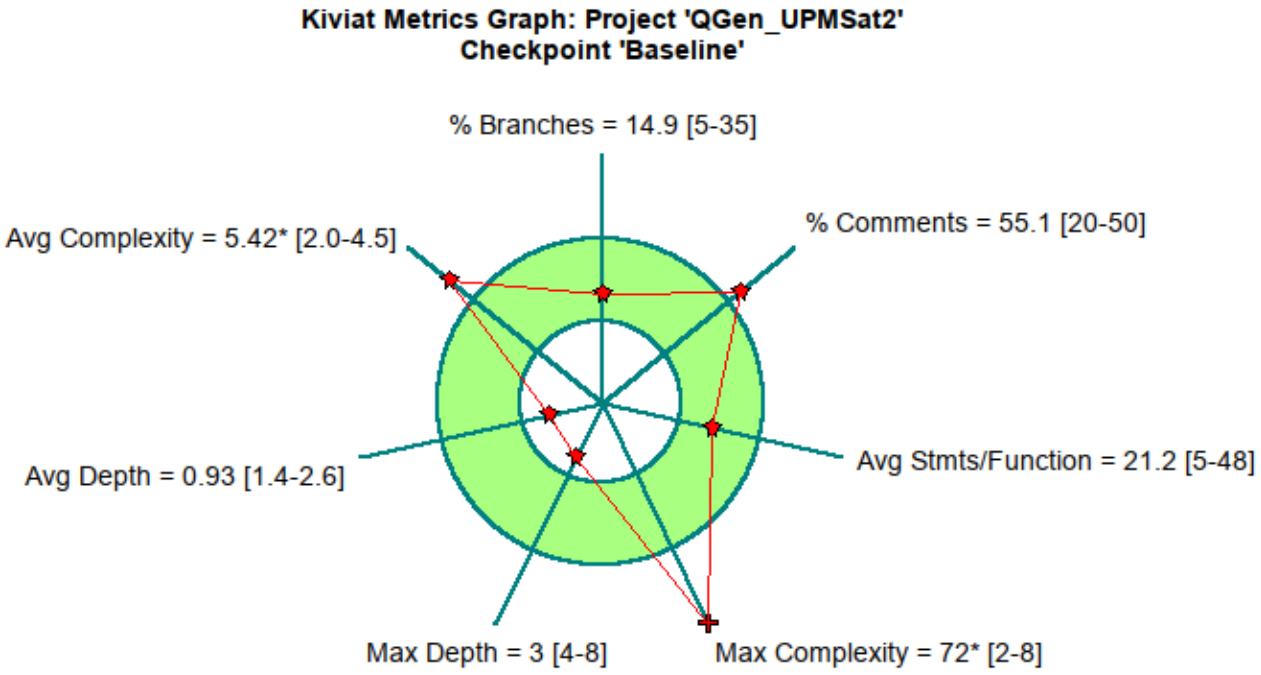

Figure 13 Kiviat graph for the UPMSat-2 project

On the other hand, the table presented in Figure shows individual metrics per function, like the number of statements, or the maximum depth. As can be seen, the most complex and largest function is control_Reference_Model_comp, which implements the step function for the control algorithm, i.e. it receives the inputs and return the output from the input and output Simulink ports, respectively.

Name of Function	Complexity	Statements	Maximum Depth	Calls
control_Reference_Model_comp	72*	316	2	9
control_Reference_Model_initOutputs	1*	3	1	3
control_Reference_Model_initStates	2*	7	2	1
control_Reference_Model_up	2*	3	2	0
Fails_X_comp	9*	50	3	5
Fails_X_initOutputs	2*	3	2	0
Fails_Y_comp	8*	53	3	5
Fails_Y_initOutputs	2*	3	2	0
Fails_Z_comp	9*	50	3	5
Fails_Z_initOutputs	2*	3	2	0
Noone_Fails1_comp	3*	6	2	0
Noone_Fails1_initOutputs	2*	3	2	0
q_1_Fail_comp	20*	83	2	7
q_1_Fail_initOutputs	2*	6	2	3
q_1_Fail_initStates	1*	3	1	0
q_2_Fails_comp	4*	31	2	0
q_2_Fails_initOutputs	2*	3	2	0
qgen_abs_gasingle	1*	1	1	1
qgen_entry_control_Reference_Model_comp	1*	2	1	2
qgen_entry_control_Reference_Model_init	1*	2	1	2
qgen_floor_gasingle	1*	1	1	1
qgen_getinf_gareal	1*	1	1	0
qgen_getnan_gareal	1*	1	1	0
qgen_isfinite_gareal	1*	1	1	0
qgen_isinf_gareal	2*	1	1	2
qgen_isnan_gareal	1*	1	1	0
qgen_mod_gauint8	2*	2	1	0
qgen_pow_gasingle	1*	1	1	1
qgen_round_gasingle	4*	5	2	1
qgen_safe_div_gareal	7*	11	3	3
qgen_sqrt_gasingle	1*	1	1	1

Figure 14 Source Monitor metrics for all the autogenerated functions

5.9.1. DQA007 – code cyclomatic complexity

5.9.1.1. Measurement

This KPI aims to estimate the complexity and maintainability of Simulink/QGen models based on the average cyclomatic complexity (CYC) of all the QGen autogenerated source code. The CYC represents the number of linear-independent paths from the graph that models the structure of a given function.

A low CYC value represents a simple and readable function, which increases the effectiveness during the testing phases. In the AURORA project, the upper limit value for the CYC is set to 10, based on the recommend value from the MISRA-C coding standard. These metrics are obtained with the Source Monitor tool.

5.9.1.2. Results

The following figure shows the CYC metrics and chart obtained from the QGen autogenerated code. The chart on the left shows a range of complexity values on the X axis; and on the Y axis the number of functions whose complexity is within that range. The figure on the right shows some of the functions and their CYC values.

By way of example, the most-right bar means that there is only one function with a CYC between 72 and 79, this corresponds to the control_Reference_Model_comp function with a CYC of 72.

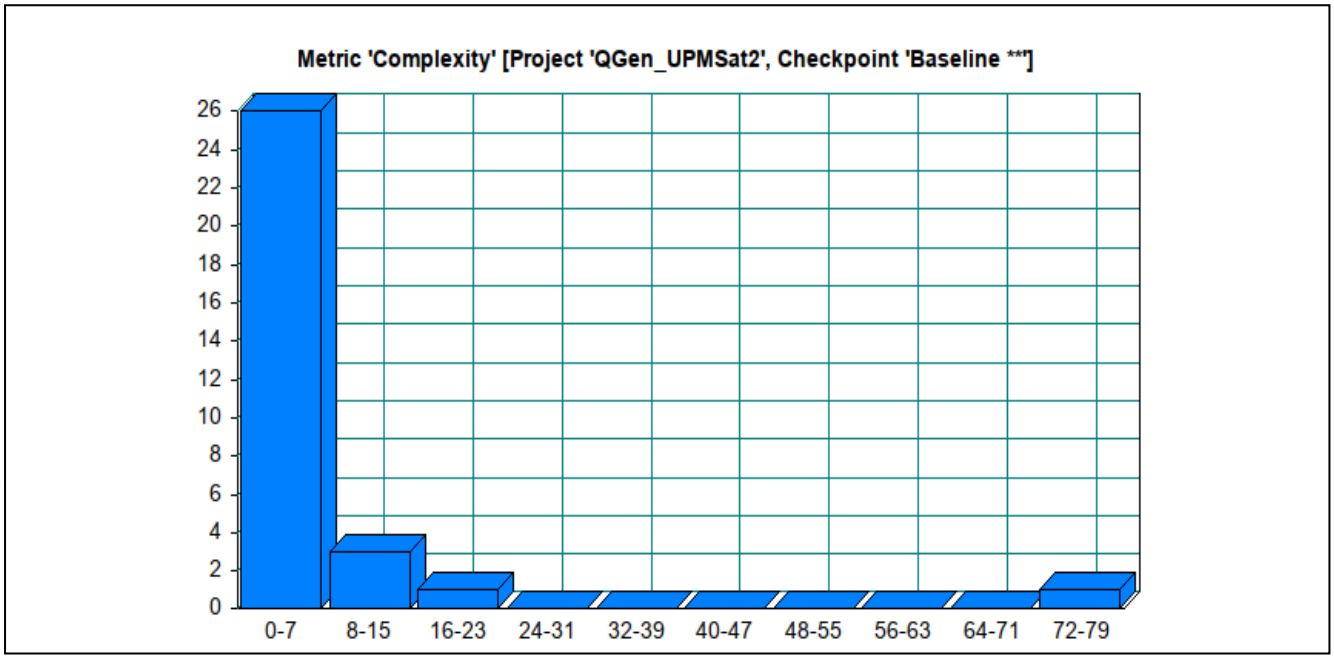

Figure 15 CYC results for UPMSat-2 ACS generated code

Based on the values presented in the above figure, we can conclude that the average CYC from all the autogenerated functions is 5.42.

5.9.1.3. Evidence

The average CYC can be easily verified from the figure and table depicted in Figure 13 and Figure 14, respectively.

5.9.2. DPI005 – maximum number of nested statements in a function

5.9.2.1. Measurement

This indicator seeks to perform a comparison of the complexity of traditional development methods (hand-written code) and QGen generated code. Usually, autogenerated software tends to be messy, as it does not have to be read or corrected by human. This is applicable only to conventional software applications like websites, but this is not allowed for high integrity applications (like space/flight software) which establish certain limits for code quality metrics.

In this case, the maximum number of nested statements in a function provides a measurable value proportional to the code complexity, thus, lower values are desirable.

5.9.2.2. Results

The maximum number of statements is 3 and corresponds to functions: Fails_X_comp, Fails_Y_comp, and Fails_Z_comp. These three functions map to blocks located inside the "IF Fail Recalculate" Simulink subsystem (c.f. Figure 8). The chart bar depicted in Figure 16 shows the maximum depths on the X axis, and the number of functions with that depth in the Y axis. As you can see, 3 is the "maximum depth" value with more than 0 occurrences.

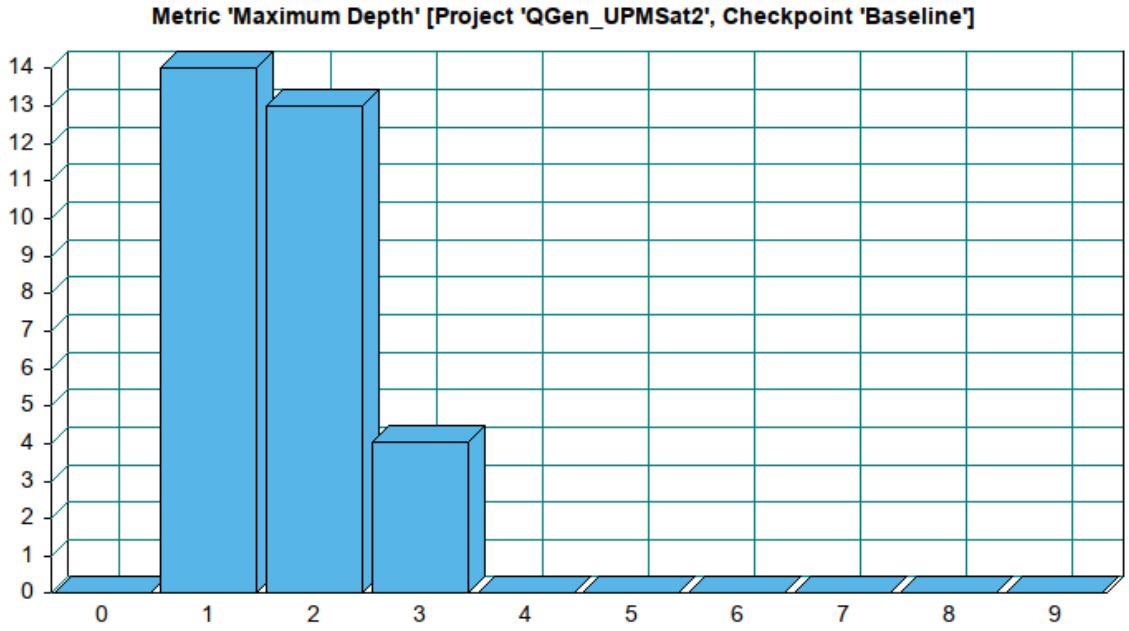

Figure 16 Nested statements (max depth) diagram

5.9.2.3. Evidence

This KPI can be easily verified from the table depicted in Figure 16, and graphically inspected in Figure 17.

5.9.3. DPI006 – number of statements

5.9.3.1. Measurement

This indicator and DPI05 share the same objective, comparing traditional hand-written code and QGen autogenerated code, considering the total number of statements from each project. This value can be automatically obtained with the Source Monitor tool.

5.9.3.2. Results

In total, there are 888 statements. The following chart bar presents a range of "number of statements" values on the X axis; and on the Y axis the number of functions whose "number of statements" is within that range.

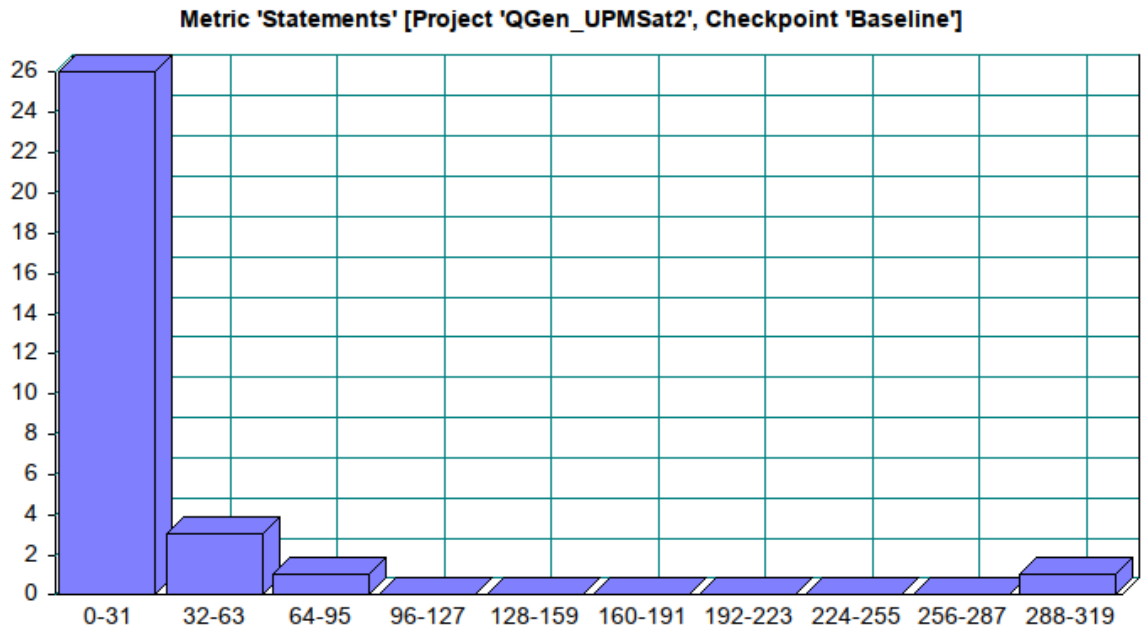

Figure 17 Statements bar chart produced by Source Monitor.

5.9.3.3. Evidence

The "number of statements" can be easily verified from the table depicted in Figure 16, and graphically inspected in Figure 17.

5.9.4. DSQ001 – comment frequency within the generated functions

5.9.4.1. Measurement

This indicator and DPI05 (c.f. section 6.2.12) share the same objective, comparing traditional hand-written code with QGen autogenerated code. But it also helps to estimate the dependability and reliability of QGen applications. To do so, the following formula shall be used:

$$Coment\ Frequency = \frac{\#\ comment\ lines\ excluding\ headers}{\#\ LOCs\ excluding\ blanks}$$

Source code comments are a controversial issue in software engineering. The truth is that in the realm of auto-generated software, comments significantly enhance readability mainly for these reasons:

- Autogenerated software does not suffer the outdated comments problems since they are always updated when the associated source text is changed.
- Autogenerated code tends to be complex, in those cases, comments can help to emphasize the code structure. Although, there is nothing better than small functions or easy to read code.
- Comments help to provide additional information that cannot be expressed in the C or Ada language. Especially, the mapping between the Simulink modelling elements with the actual code.

5.9.4.2. Results

In total there is a comment frequency of **55.10 %**.

5.9.4.3. Evidence

This data was automatically calculated by the Source Monitor tool and is included in the Kiviat graph (c.f. Figure 13).

5.9.5. DPI007 – number of lines of generated code per function

5.9.5.1. Measurement

This indicator and DPI05 (c.f. section 6.2.12) share the same objective, to compare traditional hand-made code and QGen autogenerated code. But it also helps to estimate the increase of software productivity based on auto-code applications.

This indicator is obtained from the number of LOCs (Lines Of Code) generated per function, including comments but not including blank spaces. Since there are many functions per module and per project, the average shall be considered. This value can be automatically obtained with the Source Monitor tool

5.9.5.2. Result

In average there are **21.2** LOC/function.

5.9.5.3. Evidence

This data, as all the other KPI values from this section, was obtained from the Source Monitor tool. The statements per function are depicted in Figure 14 and Figure 17. The actual result (21.2 LOC/function) is shown in the Kiviat graph (Figure 13).

5.10. Coverage

All the coverage data in this chapter were obtained by running the tests on an Ubuntu 2020.04.1 machine. The QGen generated code was compiled and run on this machine for the SIL testing. Although, the code coverage was not performed on a TSIM3 emulator, its results are assumed to be the same as the coverage during SIL. Consequently, only DSQ002 and DSQ003 KPIs are discussed.

Data were gathered via **gcov** and **lcov** tools, and transformed to html reports using **genhtml** tool. Multiple runs were executed for 10 test cases, their individual coverage reports were merged into a single report with LCOV and it is depicted in Figure .

Current view: top level - qgen_generated		Hit	Total	Coverage
Test: merged.info	Lines:	472	482	97.9 %
Date: 2022-09-27 11:09:46	Functions:	29	31	93.5 %
	Branches:	276	290	95.2 %

Filename	Line Coverage		Functions		Branches	
control_reference_model.c	100.0 %	219 / 219	100.0 %	4 / 4	100.0 %	142 / 142
fails_x.c	100.0 %	37 / 37	100.0 %	2 / 2	100.0 %	26 / 26
fails_y.c	100.0 %	39 / 39	100.0 %	2 / 2	100.0 %	24 / 24
fails_z.c	100.0 %	37 / 37	100.0 %	2 / 2	100.0 %	26 / 26
noone_fails1.c	100.0 %	10 / 10	100.0 %	2 / 2	100.0 %	6 / 6
q_1_fail.c	100.0 %	70 / 70	100.0 %	3 / 3	100.0 %	34 / 34
q_2_fails.c	100.0 %	24 / 24	100.0 %	2 / 2	100.0 %	8 / 8
qgen_abs_gasingle.c	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0
qgen_entry_control_reference_model.c	100.0 %	8 / 8	100.0 %	2 / 2	-	0 / 0
qgen_floor_gasingle.c	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0
qgen_getinf_gareal.c	0.0 %	0 / 2	0.0 %	0 / 1	-	0 / 0
qgen_getnan_gareal.c	0.0 %	0 / 2	0.0 %	0 / 1	-	0 / 0
qgen_isfinite_gareal.c	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0
qgen_isinf_gareal.c	100.0 %	2 / 2	100.0 %	1 / 1	25.0 %	1 / 4
qgen_isnan_gareal.c	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0
qgen_mod_gauint8.c	100.0 %	2 / 2	100.0 %	1 / 1	50.0 %	1 / 2
qgen_pow_gasingle.c	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0
qgen_round_gasingle.c	100.0 %	6 / 6	100.0 %	1 / 1	50.0 %	3 / 6
qgen_safe_div_gareal.c	50.0 %	6 / 12	100.0 %	1 / 1	41.7 %	5 / 12
qgen_sqrt_gasingle.c	100.0 %	2 / 2	100.0 %	1 / 1	-	0 / 0

Generated by: LCOV version 1.14

Figure 18 Code coverage report after the SIL execution (in a PC platform)

As can be seen in Figure , there is a good coverage for the generated code after running all the test cases. Individually, these test cases cover only some parts of the autogenerated code, but together, they cover almost all the generated code. This is because the UPMSat-2 ACS has an FDIR module named "IF Fail Recalculate" which considers three cases:

1. None magnetorquer fails: This case is modeled in the "None Fail" Simulink subsystem and it is entered when all MGTs can be used. Then, this block is just a by-pass which does not modify the signals. Only *a single test* is performed for this test case.
2. One magnetorquer fails: This case is modeled in the "1 Fail" Simulink subsystem and it is entered when two MGMs can be used. Then, the MGT that fails is processed. This is implemented by the fails_x, fails_y, and fails_z C modules, which represent a failure in the MGT oriented to the specified axis (X, Y or Z). *Three different tests* were performed for this test case, one per failing magnetorquer).
3. Two magnetorquers fail: This case is entered two MGTs are failing and is modeled in the 2 Fails models. This is considered a "degraded" mode for the ACS since there are actuations only on one axis. *Three different tests* were performed for this test case, one for a failure in the X & Y; X & Z; and Y & Z MGTs.

Together, the 10 tests consider all these cases, achieving a high coverage percentage (greater than 93.5%). The low coverage in the individual test cases is due the fact that we have only one Simulink model, then, the autogenerated code maps to every subsystem, even though some of them are not used in a test case. It should be noted that these metrics not only evaluate the quality of the generated code, but also the quality of our test cases. The individuals and merged coverage are summarized in Table 1.

Test Case	Description	Code Coverage

		Statements	Functions	Branches
None MGT Fails	All magnetorquers (MGT) are working correctly	58,90%	71,00%	56,60%
X MGT Fails	All except the X-axis MGT are working correctly	76,30%	80,60%	73,80%
Y MGT Fails	All except the Y-axis MGT are working correctly	76,80%	80,60%	73,10%
Z MGT Fails	All except the Z-axis MGT are working correctly	81,20%	88,50%	69,00%
X, Y MGT Fail	Only the Z-axis MGT is working correctly	61,80%	71,00%	57,20%
X, Z MGT Fail	Only the Y-axis MGT is working correctly	61,80%	71,00%	57,20%
Y, Z MGT Fail	Only the X-axis MGT is working correctly	61,80%	71,00%	57,20%
	Merged coverage:	97,90%	93,50%	95,20%

Table 1 Code coverage for the individual and merged tests.

5.10.1. DSQ002 – coverage % of branches during SIL

5.10.1.1. Measurement

Branch coverage is the percentage of branches taken from all branches in the code. For our model, all files generated by QGenC were monitored, and their branch coverage was averaged.

5.10.1.2. Result

Considering the merged results from our 10 test cases, the average branch coverage is **95.2%**.

5.10.1.3. Evidence

The code coverage report is presented in Figure . The repository which gathers this coverage analysis is hosted in a private server, please contact with the STRAST research group for further information.

5.10.2. DSQ003 – coverage % of function statements during SIL

5.10.2.1. Measurement

Function coverage is a percentage of statements hit from all the statements in the code. For our model, all files generated by QGenC were monitored, and their branch coverage was averaged.

5.10.2.2. Result

Considering the merged results from our 10 test cases, the function coverage for this test case is **97.9%**.

5.10.2.3. Evidence

The code coverage report is presented in Figure . The repository which gathers this coverage analysis is hosted in a private server, please contact with the STRAST research group for further information.

5.11. Tests

These indicators evaluate the correctness of the code generated by QGen performing MIL and SIL testing. In the case of the SIL and PIL testing, we follow this strategy:

1. QGenC is invoked passing the actual model as an input.
2. The code is executed in a processor simulator, LEON-2 for EUCLID and LEON-3 for UPMSat-2.
3. The inputs generated during the MIL testing are recorded and passed to the generated code.
4. The outputs obtained during the previous step are compared against the outputs obtained during the MIL testing.

The QGen Debugger tool facilitates the whole process since it automatically generated "testing code" that performs all these steps. In the case of the SIL testing, the program run on the TASTE Virtual Machine (Debian Bullseye). This "testing code" makes use of the file system services to read and write the input and expected values vectors.

However, these filesystem services are not available for PIL testing which run on the LEON-3 simulators. That is why, the "testing code" generated by QGen was significantly modified to redirect the output writings to the standard output. The expected outputs and input values vectors were embedded as a C module (.c/.h) holding one matrix; this strategy is used also in the Simulink SIL and PIL simulation tools from MathWorks.

All the tests performed with QGen Debugger have been configured with a tolerance of 0.00E+00.

5.11.1. DPI003 – error tolerance in the MIL

5.11.1.1. Measurement

This KPI is related to the correctness of the C code generated by QGen. This value is inferred from the deviation between the MIL (Model-In-the-Loop) validation environment and the reference EUCLID models. In essence, the MIL technique tests a model developed in Matlab/Simulink taking inputs and giving outputs from/to a test harness developed in another modelling language (Simulink in this case) to simulate the execution and physical environment.

5.11.1.2. Results

There was no difference between the MIL models used by QGen and Embedded Coder for the following reasons:

- The simulated environment from the UMSat-2 ACS was implemented in Simulink and it was not changed for the validation of the ACS model modified for the QGen code generation.
- Although the changes applied to the ACS model produced key changes in the QGen code behavior, the model's behaviour remained unchanged because the updated blocks were modified, but not replaced. Additionally, these changes were related to the usage of range expressions in selector blocks and did not affect the MIL behaviour.

So, it can be said that there is no deviation in the MIL validation.

5.11.1.3. Evidence

The UPMSat-2 ACS model is publicly available at the GitHub repository from the STRAST research group [RD11].

5.11.2. DPI004 – error tolerance in the MIL-SIL

5.11.2.1. Measurement

This indicator and the one described before (KPI DPI03) share the same objective, determine the correctness of the C code generated by QGen.

5.11.2.2. Results

As stated before in the description, this testing was performed in a LEON-3 simulator since the UPMSat-2 OBC is a LEON-3 processor with SPARCv8 architecture. We have performed **ten** tests with **5000** inputs each and **0.00E+00** tolerance, of which **16** have failed, so it is considered partially passed. The effectiveness of this test is calculated as follows:

$$Effectiveness_{UPMSat2} = \frac{5000 \times 10 - (9 + 6)}{5000 \times 10} \cdot 100 = \mathbf{99.97}\ \%$$

In addition, visual inspection has been performed during the SIL testing, this allowed use to visually determine the angular velocity evolution on the 3-axes, the results were identical to the MIL testing since the angular velocity converged to the established set-points.

5.11.2.3. Evidence

The SIL and PIL projects are hosted on a private repository. For more details contact the STRAST research group from UPM.

5.11.3. DSQ006 – SIL test without error

5.11.3.1. Measurement

This indicator is related to DPI004 since it shares the same objective and uses the same QGen Debugger project. This KPI depends on the effectiveness of the performed SIL test obtained in DPI004 (section 5.11.2). Possible values for this KPI range from 0 to 10.

5.11.3.2. Results

As previously presented in the Code Coverage section, there are ten single tests that cover the three equivalence classes 1, 2, and 3. Each test contains 5000 inputs, from which 9 points failed in the "**none mgt fail**" test case, and 6 points in the "**x mgt fails**" test case. This is considered to be successful since it performed as expected in 99.96% of the test points. Therefore, it can be said that we have performed **10 tests without errors**.

5.11.3.3. Evidence

The SIL and PIL projects are hosted on a private repository. For more details contact the STRAST research group from UPM.

5.11.4. DSQ007 – PIL test without error

The value for this indicator is the result from the execution of the test created for the SIL project, but deployed on a LEON3 simulator. The results were exactly the same as in the SIL testing, **99.97 %** of effectiveness for the performed test.

5.12. UPMSAT2 KPI data Summary

ID	Name	Result	Procedure description	Evaluation result	Evidence location
OIQ01	Integration: Other_SW (Qgen code)	True	Observable characteristics of code	Section 5.3.1	Teams/WP3/PIL/test
OIQ02	Inclusion: Qgen code (Other_SW)	True	See section 4.3.2.1	Section 5.3.2	Teams/WP3/PIL/ /test/fpmrcs_mm
OIQ03	Qgen integration into TASTE	True	Creation of TASTE projects that integrate QGen components.	Section 5.4.1	Implementations of UPMSat-2 ACS [RD09] and UPMSat-2 OBDH [RD10] in TASTE/Space Creator.
OIQ04	number of Simulink models integrated into TASTE	3	Creation of TASTE projects that integrate QGen components.	Section 5.4.2	Implementations of UPMSat-2 ACS [RD09] and UPMSat-2 OBDH [RD10] in TASTE/Space Creator.
OMQ01	number of modelling tools for Qgen code generation	3	Methods available for the QGen code generation.	Section 5.5.1	QGen user manual, c.f. section 5.5.1.3
OAS01	number of support requests to AdaCore	25	Number of tickets on the ticket website	Annex A – Euclid KPI results, AdaCore list	https://qt3-prod-1.adacore.com/
OAS02	AdaCore's response time for support requests	7.58 hours	Time delta between opening and closing the ticket	Annex A – Euclid KPI results, AdaCore list	https://qt3-prod-1.adacore.com/
OAS03	number of issues and bugs sent to AdaCore	3 minor, 1 major bug, 17 issues	Number of issues and bugs sent	Annex A – Euclid KPI results, AdaCore list	https://qt3-prod-1.adacore.com/

ID	Name	Result	Procedure description	Evaluation result	Evidence location
OAS04	number of days to solve a bug by AdaCore	avg. 17,177 days	Time between opening ticket and message confirming solving bug	Annex A – Euclid KPI results, AdaCore list	https://gt3-prod-1.adacore.com/
DPI01	reduction of development effort	N/A	N/A	N/A	N/A
DPI02	reduction in testing time	N/A	N/A	N/A	N/A
ORS01	percentage of modified blocks for Qgen compatibility	2.29%	Percentage of modified blocks for Qgen compatibility	Section 5.8.1	*KPI_Results_UPM_table.ods*
DQA01	number of models used in project	1	Number of Qgen models used in project development	Section 5.8.2	*KPI_Results_UPM_table.ods* and public repository [RD11]
DQA02	number of Simulink elements	619	Sum of # of basic blocks and their connections.	Section 5.8.6	*KPI_Results_UPM_table.ods* and public repository [RD11]
DQA03	effort to develop Simulink models	5 * 160 = 800 hours	Time development of Simulink models. Described in section 5.8.4.1.	Section 5.8.4	*KPI_Results_UPM_table.ods*
DQA004	maximum subsystem depth	5	Maximum subsystem depth measured manually and visually verified.	Section 5.8.5	*KPI_Results_UPM_table.ods* and public repository [RD11]
DQA005	maximum number of basic Simulink blocks	250	Maximum number of basic Simulink blocks.	Section 5.8.6	*KPI_Results_UPM_table.ods* and public repository [RD11]

ID	Name	Result	Procedure description	Evaluation result	Evidence location
DQA006	maximum number of nested bus structures	0	Maximum number of nested bus structures.	Section 5.8.7	*KPI_Results_UPM_table.ods* and public repository [RD11]
DQA007	code cyclomatic complexity	5.42	Code cyclomatic complexity measured through SourceMonitor	Section 5.9.1	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DPI005	maximum number of nested statements in a function	3	Maximum number of nested statements in a function measured with SourceMonitor	Section 5.9.2	*KPI_Results_UPM_table.ods* public [RD11] *and private repositories.*
DPI006	number of statements	888	Number of statements.	Section 5.9.3	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DSQ001	comment frequency within the generated functions	55.1%	Proportion of comments within the generated functions.	Section 5.9.4	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DPI007	number of lines of generated code per function	21.2	Number of lines of generated code per function (including comments but not including blank spaces)	Section 5.9.5	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DSQ002	coverage % of branches during SIL	95.20%	Coverage % of branches during SIL unitary test using gcov	Section 5.10.1	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DSQ003	coverage % of function statements during SIL	97.90%	Coverage % of function statements during SIL unitary test using gcov	Section 5.10.2	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*

ID	Name	Result	Procedure description	Evaluation result	Evidence location
DSQ004	coverage % of branches during SIL	Duplicate of DSQ002	-	-	-
DSQ005	coverage % of function statements during PIL	Similar to DSQ003	Coverage % of function statements during PIL unitary test using gcov	N/A	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DPI003	error tolerance in the MIL	0.0E+00, N/A	Error tolerance in the MIL validation environment with respect to the reference Euclid models.	D3.7.2	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DPI004	error tolerance in the MIL-SIL	99.97 % of effectiveness with 0.0 tolerance for error.	Error tolerance in the MIL-SIL validation environment with respect to the MIL reference values	Section 5.11.1	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DSQ006	SIL test without error	100%	Percentage of exercised SIL test without error execution.	Section 5.11.2	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*
DSQ007	PIL test without error	100%	Percentage of exercised PIL test without error execution.	Section 5.11.3	*KPI_Results_UPM_table.ods,* public [RD11] *and private repositories.*

Annex A – Euclid KPI results

The attached file includes the measured KPI results. How the results were obtained is described in the Euclid KPI data chapter.

Annex B – Models adaptation metrics

metric	Modified blocks	Maximum subsystem depth	Maximum number of basic Simulink blocks	Maximum number of nested bus structures	Deviation from reference models
Result	3%	5	1386	4	0

Table 8 Summary of QGen model adaptation metrics

Simulink model	Modified blocks	% Model modified	Subsystem depth	Number of basic Simulink blocks	Number of signals	Number of elements (blocks + signals)	Number of nested bus structures	Deviation from reference models
sam	2	1,980%	2	101	138	239	3	0
sam_acm_rcs_seq	0	0,000%	0	10	9	19	0	0
sam_acm_rcs_seq_aux	0	0,000%	0	12	17	29	0	0
sam_acm_rcs_seq_pt1	22	12,717%	2	173	164	337	0	0
sam_acm_rcs_seq_pt2	6	2,419%	2	248	228	476	0	0
sam_ctrl	5	4,673%	1	107	92	199	1	0
sam_ctrl_at	4	4,255%	2	94	77	171	0	0
sam_ctrl_dz	3	6,667%	2	45	35	80	0	0
sam_ctrl_rd	0	0,000%	1	49	38	87	0	0
sam_ctrl_sa	10	5,495%	3	182	167	349	0	0
sam_mm	6	5,085%	1	118	94	212	0	0
sam_nav_sun	8	0,625%	1	1281	1086	2367	0	0
fpmrcs	2	1,235%	2	162	188	350	4	0
fpmrcs_mm	1	1,250%	0	80	83	163	0	0
fpmrcs_nm	0	0,000%	0	47	46	93	3	0
fpmrcs_nm_ctrl	15	11,811%	1	127	138	265	0	0
fpmrcs_nm_gui	3	2,256%	1	133	121	254	2	0
fpmrcs_nm_gui_ls_gui	34	2,453%	5	1386	1367	2753	0	0
ocm	12	4,255%	2	282	361	643	4	0
ocm_acm_rcs	0	0,000%	0	18	16	34	1	0
ocm_acm_rcs_orb	6	3,191%	2	188	171	359	0	0
ocm_ctrl	0	0,000%	0	15	15	30	1	0
ocm_ctrl_cl	26	12,150%	2	214	242	456	0	0

Simulink model	Modified blocks	% Model modified	Subsystem depth	Number of basic Simulink blocks	Number of signals	Number of elements (blocks + signals)	Number of nested bus structures	Deviation from reference models
ocm_ctrl_ff	0	0,000%	2	150	136	286	0	0
ocm_dvto	0	0,000%	1	64	62	126	0	0
ocm_gui	10	10,526%	2	95	84	179	0	0
ocm_hmng	0	0,000%	0	43	53	96	2	0
ocm_hmng_bias	0	0,000%	1	36	27	63	1	0
ocm_hmng_bias_ctrl	2	2,174%	1	92	91	183	0	0
ocm_hmng_bias_flg	0	0,000%	0	31	25	56	0	0
ocm_hmng_null	0	0,000%	1	70	60	130	0	0
ocm_mm	1	2,500%	0	40	34	74	0	0
ocm_nav_dv	6	1,923%	3	312	271	583	0	0

Table 9 QGen model adaptation metrics

Annex C – UPMSat-2 PIL & SIL test and coverage results

Test Case	Description	SIL Tests		PIL Test		Code Coverage		
		Effectiveness	Result	Effectiveness	Result	Statements	Functions	Branches
None MGT Fails	All magnetorquers (MGT) are working correctly	99,82%	PASSED (9 failed)	99,82%	PASSED (9 failed)	58,90%	71,00%	56,60%
X MGT Fails	All except the X-axis MGT are working correctly	99,88%	PASSED (6 failed)	99,88%	PASSED (6 failed)	76,30%	80,60%	73,80%
Y MGT Fails	All except the Y-axis MGT are working correctly	100,00%	PASSED	100,00%	PASSED	76,80%	80,60%	73,10%
Z MGT Fails	All except the Z-axis MGT are working correctly	100,00%	PASSED	100,00%	PASSED	81,20%	88,50%	69,00%
X, Y MGT Fail	Only the Z-axis MGT is working correctly	100,00%	PASSED	100,00%	PASSED	61,80%	71,00%	57,20%
X, Z MGT Fail	Only the Y-axis MGT is working correctly	100,00%	PASSED	100,00%	PASSED	61,80%	71,00%	57,20%
Y, Z MGT Fail	Only the X-axis MGT is working correctly	100,00%	PASSED	100,00%	PASSED	61,80%	71,00%	57,20%
Total:		99,96%	PASSED	99,96%	PASSED	97,90%	93,50%	95,20%
		In average				**Merged coverage**		

Table 2 Test results after PIL and SI