QGen Evaluation Report

D3.6

Document Code:	AUR-ESC-RP-0007
Document Version:	4.1
Document Date:	31/01/2023
Internal Reference:	DOC00205357

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
A. Lyko J. Gómez del Pulgar Javier Gomez del Pulgar Vazquez Firmado digitalmente por: Javier Gomez del Pulgar Vazquez ND: CN = Javier Gomez del Pulgar Vazquez email = javier.gomezpulgar@aeroespacial.sener C = AD O = Sistemas de Vuelo OU = Sener Aeroespacial Fecha: 2023.01.31 12:28:41 +01'00'	P. Česák A. Rodríguez Ana Isabel Rodriguez Firmado digitalmente por Ana Isabel Rodriguez Fecha: 2023.01.31 12:36:33 +01'00' [illegible] 2023.01.31 13:31:28 +01'00'	R. Talavera [illegible] S-1-5-21-3346289680-2170560887-3771389281-2139/fe7945e6-b381-47df-853f-2edf09466092/login.windows.net/764631aa-9abe-4d86-95e2-7847ae3d6e4a/raquel.talavera@aeroespacial.sener 2023.02.01 10:03:40 +01'00'	Alfonso López Alfonso López Firmado digitalmente por: Alfonso López ND: CN = Alfonso López email = alfonso.lopez@aeroespacial.sener C = ES O = SENER Aeroespacial S.A. OU = Quality Division Fecha: 2023.01.31 13:39:45 +01'00'	A. Rodríguez Ana Isabel Rodriguez Firmado digitalmente por Ana Isabel Rodriguez Fecha: 2023.01.31 17:37:13 +01'00'
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
1.0	2021-07-31	A. Lyko	All	Initial version - QGen Debugger - SIL - chapter 6.
1.1	2021-11-04	A. Lyko	6.6	Added workaround description for the "Parameters Connection" issue.
2.0	2021-11-11	A. Lyko J. Gomez del Pulgar	3, 4 and 5	Overview Section written and Model Verifier and Compatibility Checker sections included
2.1	2021-12-13	A. Lyko	10	Create chapter Annex A - QGen Findings
3.0	2022-01-25	A. Lyko	7, 10	Added PIL chapter, add PIL finding #4.
3.1	2022-02-15	J. Gómez	8	Added two additional QGen findings
3.2	2022-02-16	A. Lyko	10	Added finding #8
4.0	2023-01-16	J. Gómez del Pulgar P. Česák	1, 2, 3 8, 9	Minor changes for final version Added HIL section, Summary section.
4.1	2023-01-31	A. Rodriguez	General	Final version issue after internal review

Index

1. Introduction

1.1.Purpose

The purpose of this document is to evaluate QGen toolchain by means of practical exercises performed on proprietary internal AOCS model/s or its subsystems from former Euclid mission.

The whole process is described with all observed constraints and benefits.

1.2. Scope

The scope of this document is limited to QGen toolset that is used in the *WP3 AOCS/GNC Code Generator (QGen) Technology Demonstrator* of AURORA , as described in Annex 1 Part A of [AD01]

AURORA is funded under the European Union's Horizon 2020 – the Framework Programme for Research and Innovation (2014-2020) under the H2020 Space work programme 2014. The aim of AURORA is to explore and identify solutions for the application of auto-coded technologies in the software development of space applications, and the demonstration of applicability of auto-coding technologies.

The appropriate tools under scope of evaluation are the main tools from QGen toolchain: Model Verifier, Compatibility Checker, Code Generator and Debugger. The workflow for each tool or exercise is described in this report.

1.3. Prerequisites

The user should be familiar with Matlab and Simulink.

The user should have set-up the QGen toolchain and development environment as described in [RD02].

2. Related Documentation

2.1. Applicable Documents

AD #	Title	Reference	Issue	Rev
[AD01]	AURORA Grant Agreement	GA number 101004291	-	-
[AD02]	D1.4 Quality Assurance Management Plan	AUR-SAE-PL-0001	1	-
[AD03]	D2.3 AURORA SW Development Plan	AUR-SAE-PL-0002	2	-

Table 1 Applicable documents

2.2. Reference Documents

RD #	Title	Reference	Issue	Rev
[RD01]	QGen User Guide	January 12, 2021	-	21.1
[RD02]	QGen toolset and SW Development Environment	AUR-ESC-RP-0022	-	1.1
[RD03]	MATLAB Overview	Web site	-	-
[RD04]	Processor-in-the-loop simulation on embedded Linux boards	Web site	-	-
[RD05]	Software-in-the-Loop Testing glossary	Web site	-	-
[RD06]	Get Started with Simulink	Web site	-	-
[RD07]	Signal Builder user guide	Web site	-	-
[RD08]	D3.1 AOCS_GNC Code Generator Requirements Specification	AUR-SAE-SP-0001		1.2
[RD09]	D3.4 QGen tool-set and SW Development Environment	AUR-ESC-RP-22		1.2

Table 2 Reference documents

2.3. Acronyms

Acronym	Description
AD	Applicable Document
CLI	Command Line Interface
GPL	GNU General Public License
GUI	Graphical User Interface
HIL	Hardware in the loop
IDE	Integrated Development Environment
PIL	Processor in the loop

Acronym	Description
RD	Reference document
SIL	Software in the loop

Table 3 Acronyms

2.4. Terms and Definitions

Term	Description
AdaCore	SW development tools supplier company
Matlab	Programming and numeric computing platform used to analyse data, develop algorithms, and create models [RD03].
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 [RD04].
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 [RD05].
Simulink	Block diagram environment for simulation and Model-Based Design integrated with MATLAB [RD06], [RD05].

Table 4 Terms and Definitions

3. Overview

This document introduces QGen as a tool to generate code from a Simulink model in the scope of the AURORA project. QGen is a tunable code generator and model verification tool developed by AdaCore for a safe subset of Simulink and Stateflow models. General features, implementation of QGen, limitations and issues are all documented here to provide a general evaluation report.

Information about QGen implementation and the software environment can be found in [RD09].

This document starts by analyzing the two relevant tools that QGen offer to evaluate model verification and compatibility with QGen. Those tools are, respectively, QGen Model Verifier and QGen Compatibility Checker. These two functions should be run prior to a first auto generation attempt to ensure QGen compliance with the model. Later stages of the verification process involving QGen is the generation of code and its implementation in a proper environment to perform Model-in-the-loop and Software-in-the-loop comparisons. A major strength of QGen is the implementation of the QGen Debugger option, which generates a GNAT studio project capable of generating the code and run some predefined simulation given a set of inputs.

Processor-in-the-loop simulation is pending to be performed in the current status of the project.

This document is divided into different sections:

- Section 1 is the introduction of the document, with the scope and some prerequisites
- Section 2 is the related documentation necessary
- Section 3 is this overview of the document
- Section 4 explains the QGen Model Verifier and its results
- Section 5 deals about the QGen Compatibility Checker and its results
- Section 6 describes the QGen Debugger implementation and how the MIL-SIL comparison is performed

  This section is critical to the evaluation of the tool as one major issue to address is to ensure that the autogenerated code is equivalent to the original model. Information about the SIL theory, workflow, prerequisites and a few evaluation examples can be found here.
- Section 7 talks about the PIL implementation of the project.
- Section 8 describes the relationship between QGen and the hardware in the loop phase.
- Section 9 summarizes QGen usage.
- Annex A reports QGen findings. Any issue found during the process is stated here, with the proper description and justification of the workaround to be implemented, if any.

4. QGen Model Verifier

The QGen Model Verifier performs checks for violations of MISRA Simulink rules and verifies that the code generated from de model does not contain:

- Possible run-time errors
  - Datatype overflow check
  - Range check
  - Division by zero
  - Array index check
- Dead execution paths in the control flow
  - Conditions that might always result in the same execution flow
  - Expressions that always return the same value
- Other functional or safety property violations

Two different types of analysis are provided by QGen Model Verifier. A quick check, which performs a fast analysis and reports the most relevant errors to the user and a deep analysis, which carries out a thorough check of the model.

Whilst the quick check is suitable for routine regular checks and can be performed to any model size, the deep analysis will exhaustively detect all possible errors, resulting in typical longer execution times. More information about the Model Verifier and how to set up the models can be found in [RD01].

Requirement AUR-DES-0080 of [RD08] states that the Model Verifier shall be run with a deep level analysis on the models used for code generation as a first step to ensure correct generation. The line of code used is as follows:

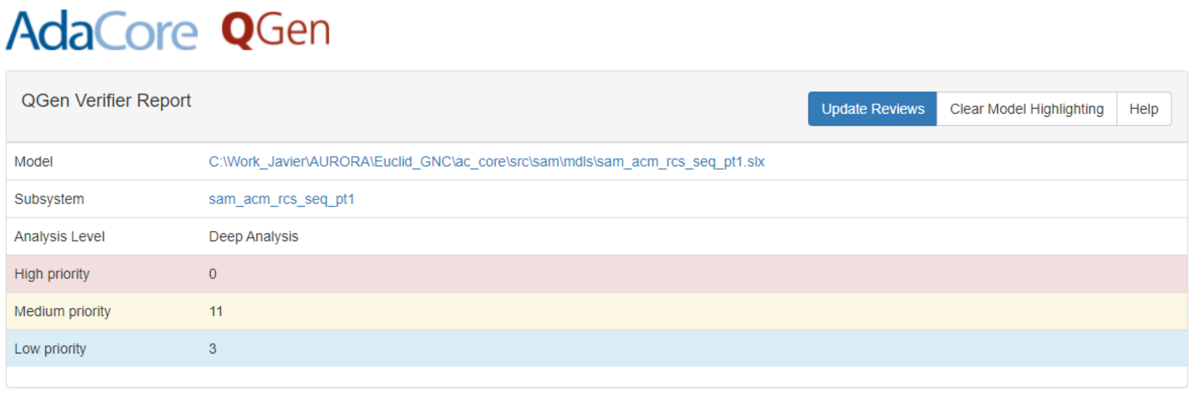
```
>> qgen_verify('mymodel.slx', 'deep', '—output,<XMI files path>','--
output','<Model Verifier results path>','-v','--clean');
```

Where:

- mymodel.slx is the model to be verified
- deep is the level of analysis
- XMI files path is the path where the XMI files will be located
- Model Verifier results path is the path where the results of the analysis will be located
- -v is the level of verbose used. This switch adds additional information to the command window
- --clean removes previous files to perform a new analysis

In order to perform the analysis, QGen first transforms the content of the model, serializing it into multiple XMI files to extract the relevant information as a preliminary phase, one file per model and one per base workspace. This is done with the command *qgen_export_xmi,* which is called inside *qgen_verify* function. Once this XMI files are obtained, Matlab, and the model, are not necessary for the completion of the process, as these XMI files can be called from the System Command Line to generate the results. Nonetheless, the process followed here is completely inside the Matlab environment in order to generate all the required outputs inside a unique environment and for easier implementation.

The result of the Model Verifier analysis is an HTML file containing the number of high, medium and low priority warnings that are reported. Each warning is stated with a description, an advance solution, and a message. Comments can be posted, and a historic review is possible for each warning. This report is generated by the function *qgen_verifier_report*



QGen Verifier Report

Model	C:\Work_Javier\AURORA\Euclid_GNC\ac_core\src\sam\mdls\sam_acm_rcs_seq_pt1.slx
Subsystem	sam_acm_rcs_seq_pt1
Analysis Level	Deep Analysis
High priority	0
Medium priority	11
Low priority	3

Figure 1: Model Verifier Report Example

4.1. Results of Model Verifier Analysis

All Euclid models relevant for the AURORA project have been analyzed, with reports which can be found annexed to this document. None of them shows any high priority warning, satisfying requirement AUR-DES-0080.

Issues found:

- Running the model verifier on some models with the Simulink block atan2 in version 21.1 results in a failure generation of the Model Verifier. This was due to the fact that, in the process of generating the report, the model is firstly exported to XMI files, which are used to generate Ada files for the analysis report. If the model contains the atan2 function block, the Ada file generated is called qgem_arctan2_generic.adb which include a call to the file a-ngelfu.adb, which stands for Ada.Numerics.Generic_Elementary_Function, an Ada package used to implement de function.

  This file was not available for this QGen version, and the report was not successfully generated. AdaCore responded by issuing a new wavefront of their software (QGen 22.0w-21210701), solving this issue.

5. QGen Compatibility Checker

The QGen Compatibility Checker is a tool that allows the user to check if there are blocks inside a given model to be incompatible with QGen, reporting violations and warnings to investigate. Similarly, to the QGen Model Verifier, different types of levels can be found:

- Quick check: checks that the solver type is FixedStepDiscrete and that the Simulink blocks in the model and the reference models and libraries are supported by QGen
- Data Types and Callbacks Checks: includes the Quick Check with the addition of two more features
  - All port does not have matrices with more than 2 dimensions
  - No function callbacks when transferring the mask parameters values from mask to parameters with the same name
- Full check: includes the aforementioned levels of check, invoking the initial steps of code generation to perform a more thorough check of the Simulink block constraints QGen has.

The result of the analysis is an HTML file containing the potential violations of the model to be analysed.

QGen Compatibility Checker is initiated via command window script as follows:

```
>>qgen_compatibility_check('model.slx''subsys','level','report_format')
```

Where:

- Model.slx is the model to be checked
- Subsys is the path to a subsystem in the model (Default. ' ')
- Level selects the level of checks to perform. Set to full
- Report_format is the format of the output (Default 'html')

Requirements AUR-DES-0060 and AUR-DES-0070 from [RD08] states that the Compatibility Checker must be launched with FULL level check and must not contain any error or warning message respectively.

Similar process as in the Model Verifier analysis, the Compatibility checker is run for all Euclid models relevant for the AURORA project with HTML reports attached to this document.

5.1. Results from Compatibility Checker analysis

- AUR-DES-0070 is not fully satisfied since reports shows several warning messages. This requirement was added to include warning messages in this analysis, although this requirement is not critical to the development of the project
- AUR-DES-0060 is satisfied in all models except in *ocm_ctrl_cl_* where a violation of the type check supported types shows a Failed message. This is because Simulink block Dead Zone Dynamic is used inside this model. According to the QGen Users Guide [RD01], there shall not be any error with this block as in Section 9, QGen constraints on input models, this block does not appear with any constraint. When generating the code for this model, QGen is capable to generate it with no apparent issue, so modification to this model in terms of substituting this block with a supported block is not foreseen, at least while MIL-SIL comparison does not introduce any relevant difference.

6. QGen Model Debugger - software-in-the-loop

At the point of an already generated code from the user's chosen model, the next logical step would be to verify that the generated code behaves as its model reference. This can be performed on the host computer, so called software-in-the-loop (SIL), or on the target platform, so called processor-in-the-loop (PIL).

From the evaluation perspective, the QGen Debugger functionalities may be distinguished into two categories:

- Debugger
- Software-in-the-loop (SIL)

The debugger functionalities provide features like performing execution steps over the generated code lines side to side the reference model, setting breakpoints within the code or the model, reading variable current values etc., where additional details are described in document [RD01].

The software-in-the-loop test functionalities provide features like calling out the generated code with the prepared set of input values, gathering the calculated output values and comparing them with the expected results, measuring the min, max, avg. execution time of all simulation steps and providing the report with the test results.

From the generated code verification perspective, the SIL test would be the first step and the final verification goal on the host platform. The Debugger functionalities would come handy in place mostly in case of some irregularities and for issue solving activities after which the SIL test with pass result would still be the next step. Therefore, the following sections focus on describing the SIL exercises with the QGen Debugger tool.

The full SIL workflow has been exercised and evaluated (6.1, 6.2, 6.3 and 6.4). Some issues and workarounds are discussed (6.5 and 6.6).

6.1. SIL Theory

In theory, the reason behind the SIL is to verify that the transition from the model perspective to the generated (manually or automatically) code perspective was done without introducing any new unintentional or unexpected behavior/issues for defined use-cases – test-cases.

This is done by passing the same set of input values into the system model as into the system executable (code). Ideally theirs output values shall be identical.

In practice we can measure some small differences between those two sets of output values. Therefore, some small acceptable tolerance of output values shall be defined, e.g.: 1E-10.

By a test-case, the set of input-output pair values (obtained by running model simulation) is meant. This Test-case is then used as a reference for the validation of the code behavior.

Figure 2 displays software-in-the-loop test ideal workflow.

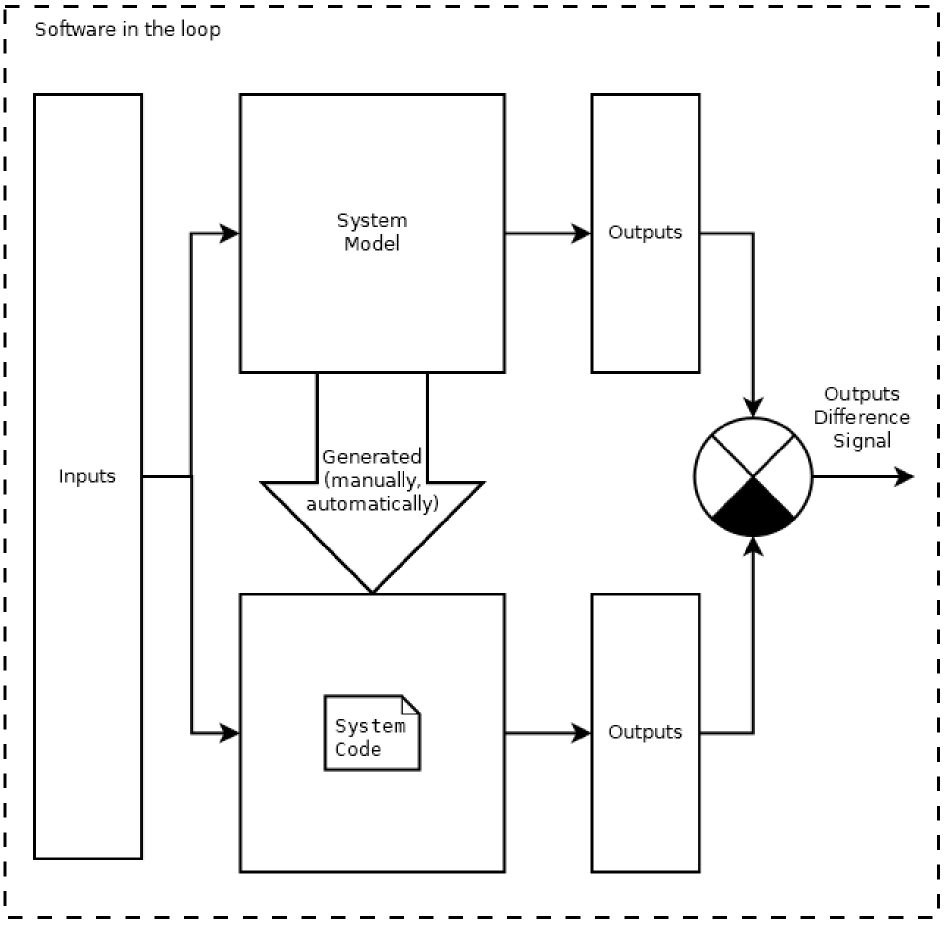


Figure 2 SIL Theory

6.2. QGen Debugger Workflow

Document [RD01] in chapter 5.2.1 describes three types of QGen Debugger workflows based on the starting point within the whole process:

- The first time to debug the model
- When the model has not changed
- When the model has changed

Figure 3 tries to visualize and summarize the main essence of those workflow types.

The first workflow, "first time to debug the model", guides the user, beginning in the model perspective, throughout intermediate steps, towards to running SIL and obtaining test summary report in form of explicitly defined simulation steps when the output values were outside of the chosen tolerance.

Based on Figure 3, the workflow "first time to debug model" would follow the sequence:
[A/B I., A II., A III., B II., B III.].

Second workflow, "when model has not changed", advice to start debugging IDE – GNAT Studio immediately without performing preceding steps originated in the user model, assuming this was already done at least once.

Based on Figure 3, the workflow "when model has not changed" would start directly at step:
[B III].

Third workflow, “when model has changed”, advice to use similar procedure to the first workflow, “first time to debug model”, except of using GUI to trigger the intermediate XMI model export, it is advised to use CLI function instead, skip running the simulation and thus not-updating simulation results and only re-generate system code within Debugger IDE – GNAT Studio (at this point, the information about the model for the code generation shall be taken from intermediate XMI model export – so there is no direct connection to/need for the Simulink model itself) - and start debugging. This workflow also assumes that the first workflow, “first time to debug the model” was performed at least once.

Based on Figure 3, the workflow “when model has changed” would follow only the B branch: [A/B I., B II., B III.].

For Aurora QGen Debugger evaluation, the first workflow, “first time to debug the model”, was chosen to be used, as:

- It is needed to be performed at least once when using other workflows
- It contains all steps included in other two workflows plus additional steps

The whole evaluation process therefore starts at step A/B I. from Figure 3.

It is important to mention, that all transitions between steps from Figure 3 are automated by QGen and user does not have to worry about implementation details of intermediate steps. User can just trigger transition from one step to another via GUI (or in some cases via CLI).

In particular, the simulation model (A II.), simulation log csv (A III.), model export XMI (B II.), system code (B III.), simulation framework code (B III.), the SIL executable (B III.) and summary test report (B III.) are generated by QGen automatically.

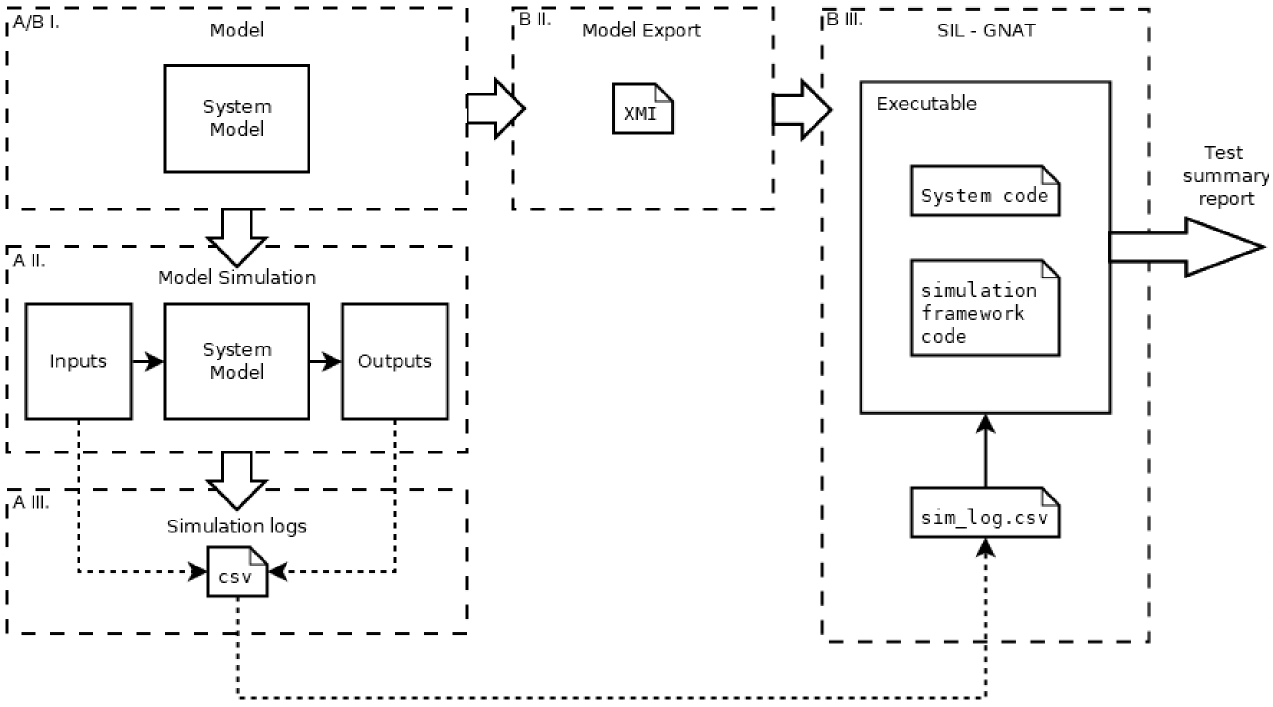


Figure 3 QGen Debugger – SIL- workflow

6.2.1. Simulation Framework Code

As mentioned in section 6.2, the 'simulation framework code' is generated by QGen automatically and consists of two main parts: the model/simulation specific part and the model independent or generic part.

The main functionalities included within the 'simulation framework code' are:

- Parsing the SIL executable arguments, which are the:
    - 'Simulation log csv file' path (A III., B. III.),
    - Chosen 'tolerance' for this SIL execution,
    - Chosen name of the 'test summary report' output file.
- Parsing the 'simulation log csv' file with built-in knowledge about the port and signals order within.
- Calling the system code in iterations and providing the appropriate input values from 'simulation log csv' file for each simulation step and obtaining the calculated system output values.
- Measuring the execution time of the 'system code' for each simulation step and determining the min., max., and avg. execution time.
- Comparing the obtained output values from the 'system code' with the expected output values from the 'simulation log csv' file (taken from model simulation) for each simulation step
- Logging all simulation steps with the 'system code' output values outside of the acceptable tolerance interval
- Generating the output 'test summary report' file

Some of those functionalities are generic and model independent like capability to parsing CLI arguments, working with files (reading, writing), measuring the execution time, or generating the output report file.

Some functionalities are model/simulation specific, e.g.: build-in knowledge about the 'simulation log csv' file structure (number and dimension of input-output ports may vary for each model, each simulation may have different number steps to be simulated) or calling the system code (based on the arguments used with `qgenc`, the generated system code may use different type of interfaces – separated function parameters, inputs and outputs bundled into structures or use global variables instead of function parameters).

Therefore the 'simulation framework code' is generated for each specific model and its simulation.

The legal section within the 'simulation framework code' states, that there is no warranty and user may change those files.

6.2.2. Simulation Log CSV

Based on the 'simulation framework code' (described in section 6.2.1, B III. in Figure 3) and 'simulation log csv' file (A III., B III. in Figure 3) analysis, the magic keyword, called 'Reset', was revealed.

When putting the 'Reset' string into the 'simulation log csv' file on a separated line, the 'simulation framework code' will treat this as test-scenario separator and will reset all its internal variables, re-initialize the 'system code' and start a new SIL test.

6.3. SIL Evaluation workflow

6.3.1. Prerequisites

As mentioned in section 6.2, for the QGen Debugger – SIL – evaluation was chosen type of workflow, which begins in the model perspective and ends with SIL test summary report. Therefore, all necessary intermediate steps are exercised.

This type of workflow is called "the first time to debug model" in [RD01] nomenclature and in the Figure 3 perspective is represented by sequence of steps: [A/B I., A II., A III., B II., B III.].

Base on everything so far said, it may seem that the only thing needed for beginning the SIL exercise is the 'system model' itself for the step A/B I. from Figure 3 as a starting point. This statement is not completely true. The second essential ingredient for valuable SIL exercise is a defined test-case or test-cases of interest via the set of chosen input values, step A II. from Figure 3.

To summarize the SIL starting point discussion, the essential prerequisites for the QGen Debugger – SIL – evaluation are:

- **System model** (step A/B I. in Figure 3)
- Test scenario **input values** (step A II. in Figure 3)

Note: the reference output values used within the SIL workflow are generated during the model simulation (step A II. in Figure 3).

Both of those two used prerequisite-items were provided by the Euclid mission project as its heritage.

6.3.2. Input Signals

As defined in document [RD01], the tool used in QGen Debugger workflow for input values definition is a so called 'Signal Builder'.

Based on a simple and quick user familiarization with the Signal Builder tool, the conclusion was that signals can be defined only manually. Therefore, the first idea how to import the user defined input signals, was via the 'simulation log csv' file from step A III. and B III. in Figure 3.

This approach worked fine for the simplest model examples with one-dimensional ports, where inputs and outputs signals were ordered accordingly in columns (in1, in2, …, out1, out2).

After using model with multi-dimensional ports [e.g.: 3x1, etc.], the order of the signals within the 'simulation log csv' file got mixed up (e.g.: out2, in2, out1, in1, …). Therefore, the automated transition from input-output values to simulation log csv file would not be possible without the explicit knowledge of the signals order rule.

Note: Document [RD01] does not contain direct instructions how to import the test-scenario signal values, only a reference to the Signal Builder documentation, which was not at first analyzed deeply enough.

Therefore, upon a request how to properly import the user defined input values into the QGen Debugger workflow, and propagate them till the end – SIL executable, the AdaCore support team recommended to use the CLI interface function for generating the model simulation (A II. in Figure 3) together with input signals already imported within the Signal Builder:

```
>> qgen_make_sim_model('mymodel.slx', '--csv', 'input_values.csv');
```

Where:

- mymodel.slx is the name of the model for which the simulation is generated
- input_values.csv is file which define the set of input values for a particular test scenario

An example of a data format for a '`input_values.csv`' file is the following:

```
, 0.0, 0.1, 0.2, 0.3, 0.4
in1_signal_name, 0.0, 1.0, 2.0, 3.0, 0.4
in2_2x1_signal1_name, 0.0, 0.0, 0.0, 0.0, 0.0
in2_2x1_signal2_name, 1.0, 1.0, 1.0, 1.0, 1.0
```

The first line must start with comma character and contain the timeseries for all following signals. Other lines contain the signal values, where first item of each line represent the signal name. For signal with higher dimension than 1 (e.g.: [2x1], etc.), each dimension is placed on separated line.

Based on the `qgen_make_sim_model` help output, several csv files may be provided to generate simulation model for several test-scenarios in a row (not limited to one test-case).

6.3.3. Used Evaluation Workflow

The Figure 4 tries to summarize previous statements into a one picture describing the used evaluation workflow.

When QGen simulation model is generated (A II. in Figure 4), this is a good opportunity to validate transition from Euclid Simulation model (D II. in Figure 4) to QGen Simulation model (A II. in Figure 4). Both, the inputs, and outputs signals shall be equal (E I. and E II. in Figure 4) if no error occurs throughout the process, including the self-csv generation (C I.).

For the comparison purpose, input and output values within the simulation models (A II., D II.) were captured, using the 'bus creator' and 'to file' block and *.mat file format.

To summarize the final evaluation workflow, the used sequence would be the following:

1. Run Euclid Simulation model and prepare the inputs and output values (D II.)
2. Generate the csv file containing the input signals (C I.)
3. Generate the QGen simulation model by `qgen_make_sim_model` and by referencing the csv file containing the input signals and model itself (A II., C I., A/B I.)
4. Run the QGen model simulation (A II.)
5. Compare the Euclid and QGen simulation input-output values (E I. and E II.)
6. Generate the simulation log csv file (A III.)
7. Start GNAT – generate system code, model-specific simulation framework code and build executable (B II., B III.)
8. Run SIL, check summary report (B III.)

Notes:

The used evaluation workflow was based on a wrong assumption, that input signals can be defined only manually within the Signal Builder and AdaCore support team specific workflow recommendation. Although it is important to mention, that within the AdaCore support request, the importance of an automatable solution was stated. And indeed, current evaluation workflow can be automated at least within the Matlab perspective.

The activities focusing on automated solution within the GNAT studio (simulation framework code generation and specific project file *.gpr generation were not successful because of limited time allocation and other task priorities).

In fact, during writing sessions of this document, the information about possible imports of input signals into the Signal Builder [RD07] was revealed thanks to the originally overlooked link within document [RD01], section 6.2.

Based on the quick analysis it looks like imports are possible from several sources, csv, xls and mat format, where mat files could be the most suitable and 'handy' for the evaluation purposes.

Because the imports are triggered via GUI, it is not clear if automated solution would be possible in this way, whereas the QGen CLI supports it.

Following observation were therefore based on the workflow described in this section 6.3.3.

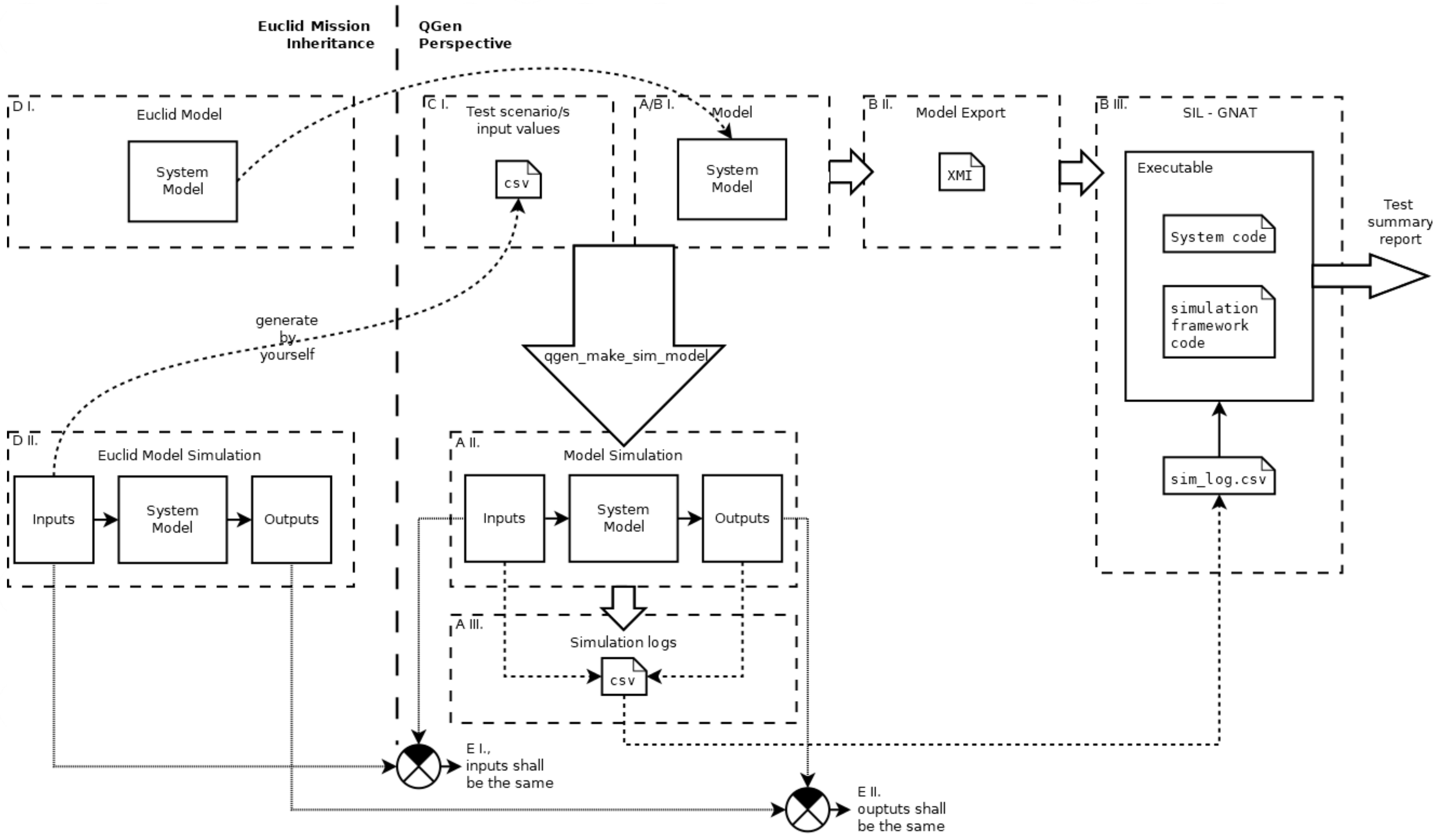


Figure 4 QGen Debugger – SIL – evaluation workflow

6.4. Exercises

Based on the used evaluation workflow, described in section 6.3.3, several models from Euclid mission were exercised this way.

Because those models are proprietary, only the generally used techniques and relations between models will be stated for sake of explanation clarity but without any implementation details.

The following Table 5 tries to explain the hierarchy between the exercised top model and its sub-systems: ([Y] – yes exercised directly, [N] – not exercised directly, only indirectly via top-model)

Top model	Referenced sub-systems
sam_ctrl [Y]	sam_ctrl_at [Y]
	sam_ctrl_dz [Y]
	sam_ctrl_rd [N]
	sam_ctrl_sa [N]

Table 5 Exercised models hierarchy

6.4.1. Exercise 1: sam_ctrl_dz

The first QGen Debugger SIL exercise performed on one of the Euclid's heritage models was done with the sam_crtl_dz sub-system.

Table 6 summarize the general sam_crtl_dz model properties:

Property	Value
Referenced Library	yes
Referenced external sub-system	no
Multi-dimensional ports	yes
Referenced WS parameters	yes
Ref. WS param. Types	Structure
	Matrix
Used Enums within the model	no

Table 6 sam_ctrl_dz general properties

The following table summarizes the status of each workflow step with observed constraints/issues and used fixes:

Workflow step	Performed	Status	Constrains / Issues	Fix
1. (D II.) - Run Euclid Simulation - prepare inputs-output values	yes	PASS	-	-

Workflow step	Performed	Status	Constrains / Issues	Fix
2. (C I.) **- Generate test scenario csv file (input values only)**	yes	PASS	-	-
3. (C I., A/B I., A II.) **- Generate QGen simulation model** **(run `qgen_make_sim_model`)**	yes	PASS	-	-
4. (A II.) **- Run QGen simulation model**	yes	FAIL → PASS	E1_I1 WS param. not referenced automatically	Add param. reference manually: ModelRef → Block Parameters →Arguments
4. (A II.) **- Run QGen simulation model**	yes	FAIL → PASS	E1_C1 "Model will not inherit sample time because the block 'SignalBuilder/ FromWs' disallows it"	Set: Simulation → Model Configuration Parameters → Solver → Periodic sample time constraint: Unconstrained
4. (A II.) **- Run QGen simulation model**	yes	UPDATE → PASS	E1_C2 The simulation model did not inherit the simulation start, stop and step time from the Signal Builder signals (csv time-series)	Set manually or via CLI: simulation start, stop and step time.
5. (E I., E II.) **- Compare Euclid vs QGen simulation input-output values**	no	-	-	-
6. (A III.) **Generate simulation log csv file**	yes	PASS	-	-

Workflow step	Performed	Status	Constrains / Issues	Fix
7. (B III.) **- generate system code**	yes	FAIL → PASS	E1_C3 Cannot open input file: model Input file not foundd	Model path shall not contain any spaces. The model XMI path is passed to the CLI `qgenc`. Spaces probably breaks the internal arguments parser.
7. (B III.) **- generate simulation code** **- build executable**	yes	FAIL → PASS	E1_I2 Compilation fails. In generated simulation code, system initialization function, the WS parameters were connected incorrectly.	Connect WS parameters manually, located at `qgen_base_workspace.h` file
8. (B III.) **- run the SIL executable** **- check the output report**	yes	PASS	-	-

Table 7 sam_ctrl_dz exercise result

The root-cause of the issue E1_I1 was not investigated, nor AdaCore support team was asked about it. To reveal if the whole workflow is performable to the SIL report, stated fix was applied and exercise continued to the next steps.

The consequence of the constraint E1_C1,, its root-cause or other possible solutions were not investigated. Related solution was applied, and exercise continued to the next step. Although it may be important to mention this constraint. Expert Simulink users may be aware of other consequences or different solutions.

The constraint E1_C2 caused, that start time of the generated simulation model (A II.) was set to 0.0s, stop time to 10.0s and step size to 'auto'. This happened even though the time-series information was already included in the 'test scenario csv' file (C I.) and properly included into the Signal Builder. As a fix of this was used a manual or via CLI update of those values.

Constraint E1_C3 message is self-explaining, although its root-cause may not. Information about this constraint was not found in document [RD01], although not using spaces in directories names and paths is considered generally as a good practice.

After investigating the root-cause of the issue E1_I2, several problems were revealed in the generated 'simulation framework code' within the initialization function and passing the system configuration parameters in (in model perspective they were in workspace).

The implementation method, of passing the model WS parameters into the initialization function as an argument and internally copying them into the component internal memory seem like a good practice. It is the one of necessary steps for utilizing the same model across the project only with different configuration parameters – like a class-objects relations. Although, in this exercise, model was used only once, with constant set of configuration parameters.

Another benefit of this type of component configuration parameters implementation is, that it allows to integrator, to change the component configuration during the run-time.

It was not investigated if this behavior (type of system-code configuration parameters implementation) can be changed or configured within QGen code generator `qgenc`.

The first problem with passing the system configuration parameters into the initialization function was with wrong arguments data type casting. Because of this, the compilator reports an error and executable was not built.

The second problem with passing the system configuration parameters into the initialization function was, that the local structure which shall hold the configuration parameters values was just declared, but not initialized. Its data may be random or empty.

After connecting the proper parameters structure from the `qgen_base_workspace.h` file (which is generated by QGen automatically) into the initialization function, an executable was successfully built.

After running the SIL executable and passing the 'simulation log csv' file in, the minimum acceptable tolerance of 5E-12 was found by experiment over several iterations. After minimizing acceptable tolerance even more, the output report file was containing each failed simulation step with explicitly defined the expected and obtained output value.

Because of the successful SIL output results, there were no suspicions to question the transition from the Euclid simulation model (D II.) to QGen simulation model (A II.) and therefore the evaluation workflow step no. 5. (E I. and E II.) was not performed. The SIL validates only the transition from the system/simulation model (A/B I., A II.) to the system code behavior (A III.), therefore the SIL result has no relation to the transition from the Euclid model simulation (D II.) to the QGen simulation model (A II.). Therefore, this assumption was not right.

After experimenting with multiple test scenarios placed in one 'simulation log csv' file, separated by 'Reset' keyword as explained in section 6.2.2 and intentionally letting the test failed by setting zero tolerance to observe the format of the output test report file, the following issue was found:

- The first test case correctly contained the failed simulation steps
- The second test case also contained the failed simulation steps from previous test case

The failed simulation steps were being reported over several test-cases in cumulative way. Because of that, for further test cases was difficult to determine, which simulation steps indeed failed in those test cases (and which reported failures was from previous test cases).

As mentioned in section 6.2.1, for the 'simulation framework code', which is responsible for generating the output report file, there are no warranties and user may change those files.

Because of this, the fix in form of proper re-initialization of the 'simulation framework code' internal variables, on the event when 'Reset' keyword occurred, was applied. This solution led to cleared-up test report.

6.4.2. Exercise 2: sam_ctrl

The second exercised model from the Euclid mission heritage was the top-model sam_ctrl, which reference all the other sub-systems from the Table 5, including already exercised sam_ctrl_dz sub-system from the section 6.4.1.

Table 8 summarize the sam_ctrl top-model general properties:

Property	Value
Referenced Library	yes
Referenced external sub-system	yes
Multi-dimensional ports	yes
Referenced WS parameters (indirectly via ref. sub-sys.)	yes
Ref. WS param. Types	Structure
	Matrix
Use Enums within the model	yes

Table 8 sam_crtl top-model properties

The following table summarizes the exercised state of each step from evaluation workflow 6.3.3 for sam_ctrl top model:

Workflow step	Performed	Status	Constrains / Issues	Fix
1. (D II.) - Run Euclid Simulation - prepare inputs-output values	yes	PASS	-	-
2. (C I.) - Generate test scenario csv file (input values only)	yes	PASS	-	-
3. (C I., A/B I., A II.) - Generate QGen simulation model (run `qgen_make_sim_model`)	yes	FAIL → PASS	E2_I1 Usage of Enums type prevent from generating the QGen simulation model (triggered from GUI and CLI) "Cannot call the constructor of 'SAM_STATE' outside of its enumeration block".	Significant update of the `qgen_make_sim_model.m` file by AdaCore (a wavefront release to v21.1).

Workflow step	Performed	Status	Constrains / Issues	Fix
4. (A II.) **- Run QGen simulation model**	yes	FAIL → PASS	E2_C1 "Model will not inherit sample time because the block 'SignalBuilder/ FromWs' disallows it"	Set: Simulation → Model Configuration Parameters → Solver → Periodic sample time constraint: Unconstrained
5. (E I., E II.) **- Compare Euclid vs QGen simulation input-output values**	no	-	-	-
6. (A III.) **Generate simulation log csv file**	yes	PASS	-	-
7. (B III.) **- generate system code** **- generate simulation code** **- build executable**	yes	PASS	-	-
8. (B III.) **- run the SIL executable** **- check the output report**	yes	PASS	-	-

Table 9 Top model sam_ctrl evaluation workflow states

The step 3. of the evaluation workflow 6.3.3 – the generation of the simulation model via `qgen_make_sim_model` CLI (or triggered from GUI without importing the input values via csv file) failed (**E2_I1**) for the sam_ctrl top model because of the usage of the enumeration types within the model.

The extensive AdaCore update of the `qgen_make_sim_model.m` file, as the wavefront release to the version 21.1, fixed the issues and simulation model could be successfully generated then.

Within the Signal Builder in QGen simulation model (A II.) were used for the Enum representation integer values. Therefore, the input part of the QGen simulation model (A II.) uses the 'convert' block, to cast the integer values back to the Enum representation before passing them in the model.

Note: the location of the `qgen_make_sim_model.m` file is `$QGEN_INSTALL/share/matlab`.

The previously mentioned file update also partially fixed the constraint **E1_C2** from Exercise 1: sam_ctrl_dz. The simulation start and stop time were now propagated from the Signal Builder to the simulation model settings (C I., A II.). The simulation step was set to 'auto'.

The constrain **E2_C1** and its status stays the same as constrain **E1_C1** from exercise Exercise 1: sam_ctrl_dz.

All other steps from the evaluation workflow 6.3.3 passed without an issue. The problems with the model configuration parameters connection **E1_I2** from the Exercise 1: sam_ctrl_dz did not reveal themselves – the parameters were connected correctly into the structures from `qgen_base_workspace.h` file.

It is important to mention, that even if the model from Exercise 1: sam_ctrl_dz, where the issue originally occurred, was referenced from the top-model in Exercise 2: sam_ctrl, the issue **E1_I2** not happened this time.

The evaluation workflow step no. 5. was not performed with the same reasoning as mentioned in the exercise Exercise 1: sam_ctrl_dz.

6.4.3. Exercise 3: sam_ctrl_at

The third and last exercised model from the Euclid mission heritage was sam_ctrl_at.

Table 10 describes some of the general sam_ctrl_at model properties:

Property	Value
Referenced Library	yes
Referenced external sub-system	no
Multi-dimensional ports	yes
Referenced WS parameters	yes
Ref. WS param. Types	Structure
	Matrix
Use Enums within the model	no

Table 10 sam_ctrl_at model properties

Note: for this exercise an already receive wavefront release to the version 21.1 of the file `qgen_make_sim_model.m` was used (described in Exercise 2: sam_ctrl).

The following table summarizes the SIL results for the sam_crtl_at subsystem:

Workflow step	Performed	Status	Constrains / Issues	Fix
1. (D II.) **- Run Euclid Simulation** **- prepare inputs-output values**	yes	PASS	-	-
2. (C I.) **- Generate test scenario csv file (input values only)**	yes	PASS	-	-
3. (C I., A/B I., A II.) **- Generate QGen simulation model** **(run `qgen_make_sim_model`)**	yes	PASS	-	-
4. (A II.) **- Run QGen simulation model**	yes	FAIL → PASS	E3_I1 WS param. not referenced automatically	Add param. reference manually: ModelRef → Block Parameters →Arguments

Workflow step	Performed	Status	Constrains / Issues	Fix
4. (A II.) **- Run QGen simulation model**	yes	FAIL → PASS	E3_C1 "Model will not inherit sample time because the block 'SignalBuilder/ FromWs' disallows it"	Set: Simulation → Model Configuration Parameters → Solver → Periodic sample time constraint: Unconstrained
5. (E I., E II.) **- Compare Euclid vs QGen simulation input-output values**	yes	FAIL → PASS	E3_I2 outputs and the inputs were different in Euclid vs QGen simulation model	Update the csv file (C I.) self-generation procedure. Increase precision to 15 digits after decimal points, fix the order of the multidimensional signals.
6. (A III.) **Generate simulation log csv file**	yes	PASS	-	-
7. (B III.) **- generate system code** **- generate simulation code** **- build executable**	yes	PASS	-	-
8. (B III.) **- run the SIL executable** **- check the output report**	yes	FAIL → PASS	E3_I3 system model vs code outputs are outside of the acceptable tolerance	connect the same configuration parameters to the code simulation as were connected in model simulation

Table 11 sam_ctrl_at SIL result

The issue E3_I1 is the same as the issue E1_I1 in case of Exercise 1: sam_ctrl_dz. The model WS configuration parameters were not referenced in simulation model (A II.) and had to be linked manually.

The constrain E3_C1 is the same as the constrains E1_C1 and E2_C1 from the Exercise 1: sam_ctrl_dz respectively Exercise 2: sam_ctrl.

Let's skip the issue E3_I2 for now, since the evaluation workflow step no. 5 was introduced based on further issue.

From the Table 10 can be seen, that all remaining steps up to building a SIL executable passed. This was also the first impression of the overall status before truly running the SIL executable. The results were way off, far away from the expected tolerance (e.g.: expected: 4.269761517581880e+00, Found: 1.834243230426479e-312) as reported in Table 11 as issue E3_I3.

This phenomenon was the impulse to start investigating each evaluation workflow step transition. Based on this, the step no. 5 (E I., E II.) was introduced. This led to uncovering several issues with the self-written script for generating the csv file (C I.) with input values in it.

At first, the output precision had to be increased at least to 15 digits after decimal point.

Secondary, the rows and columns for the multidimensional signals were mixed-up and did not correspond to the description from section 6.3.2.

After those were fixed, the step no. 5 (E I., E II.) could pass.

Unfortunately, the csv file generation script update did not fix the issue E3_I3 with the SIL execution results. And it makes perfect sense. The SIL examine only if the system model vs. system code produces the same output. If the csv (C I.) was corrupted, this would lead only to exercising unintentional test scenario, but should not have any effect on the model vs. code relation if the generation and all other steps went well.

After further investigation, the similar issue to the E1_I2 from Exercise 1: sam_ctrl_dz was revealed. This time, the connection for the system configuration parameters were grammatically right (the wrong type casting did not happen this time – and therefore the simulation framework code with system code could be compiled without warning). But the linked parameters were again pointing to the empty uninitialized structure.

Therefore, the model simulation run with the manually setup system configuration parameters from the workspace whereas the code simulation run with some random configuration parameters. Based on that, in fact, two same systems were compared but with different configuration parameters and therefore, it is reasonable to expect, that theirs outputs could differs, as SIL properly reported.

After manual connection to the proper system configuration parameters from the `qgen_base_workspace.h` file, the SIL execution passed.

It is important to mention, that the sam_ctrl_at model was already indirectly exercised within the top model Exercise 2: sam_ctrl, where this issue was not observed.

6.5. Summary

Table 12 summarizes the issues and constraints relevant for the QGen Debugger – SIL – (the self-created issues were skipped here – although they are described within the chapter 6) which were observed during the exercises on the models:

- Exercise 1: sam_ctrl_dz
- Exercise 2: sam_ctrl
- Exercise 3: sam_ctrl_at

Where:

-	means, not relevant or N/A
must be	means, the constraint must be set to the specific value
ERR	means, that the issue was observed within the exercise
OK	means, that the issue was not observed within the exercised

During the Exercise 2: sam_ctrl, the file `qgen_make_sim_model.m` was changed from v21.1 to the wavefront release version. Therefore, all issues and constrains, except the first one, are not relevant for the v21.1 within the Exercise 2: sam_ctrl. This exercise was not after then performed with v21.1 of `qgen_make_sim_model.m` file but with the wavefront release version.

The Enums were used only within the Exercise 2: sam_ctrl, and therefore this issue (E2_I1) is not relevant for other two exercises.

The 'Periodic sample time constraint' was set to value 'unconstrained' for all exercises.

The wavefront release of `qgen_make_sim_model.m` file seems to fix the constraint (E1_C2) with the inheritance of the time-series from the Signal Builder into the simulation model settings.

The issue with the wrong system parameters connection (E1_I2, E3_I2) seems to occur only for some models (even with the wave front version of `qgen_make_sim_model.m` file). When it happened, it occurs within the 'QGen simulation model' and within the 'simulation framework code'.

The interesting thing is, that when the same model (Exercise 3: sam_ctrl_at, Exercise 1: sam_ctrl_dz) where the issue occurs is referenced from the top model (Exercise 2: sam_ctrl), the parameters seems to be connected correctly.

The root-cause of this was not investigated, but it is important to realize, that all the exercised models must be compatible with the QGen otherwise the code generation would not be possible. Therefore, the usage of an unsupported techniques could be rule out. At the same time, it cannot be clearly stated if this is a QGen issue (event though it looks likely to be) or if the issue was caused by some bad user (model) practice. On the other hand, the applied fixes are straight forward and easy to implement.

		`qgen_make_sim_model.m`			
		v21.1		wavefront release	
		1. sam_crtl_dz	2. sam_ctrl		3. sam_ctrl_at
QGen simulation model	Sim. Model gen. failed - Enums prevent it	-	ERR	OK	-
	WS param. Not referenced automatically	ERR	-	OK	ERR
	Periodic sample time constraint: unconstrained	must be	-	must be	must be
	Start and Stop time not inherited from Signal Builder	ERR	-	OK	OK
Simulation framework code	Wrong type casting of system config. Param. (Init) - build failed	ERR	-	OK	OK
	Wrong connection of system config. Param. (Init) - empty struct.	ERR	-	OK	ERR

Table 12 QGen Debugger – SIL – evaluation – Issues/Constrains summary

The QGen toolchain is an active project under development, and therefore this exercises results can apply only for the specified version 21.1 of the QGen and the wavefront release of the `qgen_make_sim_model.m` file.

For the future exercises, e.g., the processor-in-the-loop (PIL), the version under evaluation is updated to the latest one available , which is v21.2.

The AdaCore QGen team is informed about the observed issue with the system parameters connection on the simulation model and simulation code level.

6.6. Workaround

The observed issues with the wrong system configuration parameters connection on the simulation model and code level were reported to the AdaCore for the Exercise 1: sam_ctrl_dz and Exercise 3: sam_ctrl_at.

The support Team has indeed confirmed its occurrence and suggested following workaround:

1. Create a new model (e.g.: "ref_model.mdl") and add a Model Reference block to the 'sam_ctrl_at' model. Create Inport and Outport blocks to match the signals.
2. In the Block Parameters (ModelReference) -> Arguments of the model reference block, the arguments should be set to the concrete values:
   * at: gnc_acdb_struct.sam.ctrl.ats
   * sc_inertia: gnc_acdb_struct.dat.sc.inertia
3. Remove from the Base Workspace the variables named "at" and "sc_inertia" (because there is already one in the Model Workspace with the same name, and QGen does not understand).
4. Now, "Create Debug session" from the new model "ref_model". "ref_model_sim" should get created. You can then "Start the QGen Debugger in GNAT Studio".

The workaround was tested with success and the configuration parameters were then connected properly. This requires either update all the tested models manually or create a script which automates the above procedure.

7. QGen Processor-in-the-loop (PIL)

7.1. Objective

The processor-in-the-loop (PIL) tests (runs on target processor) is the next logical extension of its predecessor, software-in-the-loop (SIL) tests (runs on the host development machine).

The objective is to validate the numerical equivalency between results obtained from MIL/SIL vs. PIL simulation tests and provide average, minimum, and maximum execution time measured for the tested code on the target.

In the scope of Aurora project, the objective is to compare the obtained PIL's numerical results accuracy from the code generated by QGen with its precursor – the Euclid project, which was originally based on the same models but used different code generator.

7.2. qgenpil

As of the date of the Aurora project start – the latest version of QGen v21.1 was available. Its documentation [RD01] prompt the usage of `qgenpil` tool for the PIL purposes.

The main architectural blocks of this QGen PIL solution are Matlab Simulink S-Function, GDB debugger and target executable compiled with debug options and additional `qgenpil` wrapper files. The S-Function block provides communication interface between Simulink and target executable via the GDB debugger. Then the simulation can be controlled from the Simulink, while the tested code is executed on target. The `qgenpil` tool is used in process of generating this model specific communication S-Function interfaces.

The [RD01] describes three ways how to invoke the qgenpil tool: from the Simulink GUI, from the Matlab command line or from the system command line. Unfortunately, any of those option revealed them self's viable during the `qgenpil` evaluation. Each of them ends up with different type of error.

Respectively, the Simulink GUI does not provide any related option to the PIL functionalities, Matlab command line interface `qgen_pil_build` (line 167) reports undefined function `qgen_export_types` and system command line interface `qgenpil` reports "QGEN BUG DETECTED" message.

The AdaCore support team confirmed this, see Table 16 Finding #4: Promotion of unfunctional qgenpil tool.

Based on these information and observations, it can be stated that the QGen nor QGen Debugger packages version 21.1, 21.2 or 22.0w provides tools for direct utilization of PIL testing.

8. QGen Hardware-in-the-loop (HIL)

The hardware-in-the-loop (HIL) phase is the natural continuation of the PIL phase and the final phase until adquiring a final qualified software product. Its main objective is to demostrate that the results obtained in the early validation phases can be also reproduced with real hardware on the loop.

In the scope of the Aurora's project the autogenerated code is integrated in an available Application Software of Euclid's project and tested at Sener premises. Qgen does not actively participate in this phase other than to provide the code so no issues and reports are present in this document.

9. QGen Evaluation Summary

The Demonstration of Autocoding technology has been carried out in this project using QGen product by means of practical exercises performed on proprietary internal AOCS model/s or its subsystems from former Euclid mission.

Consecutive testing phases have allowed the identification of some issues in the QGen SW Generator that have been resolved in new releases by AdaCore, as detailed in the "Annex A – QGen Findings.

QGen contains several tools that were used in different phases of the project. The evaluation was executed in the following order:

1. QGen Model Verifier that performs checks for violations of Simulink rules and verifies the code generated from the model. All Euclid models relevant for the AURORA project have been analyzed, with reports which can be found annexed to this document. None of them shows any high priority warning, satisfying requirement AUR-DES-0080.
2. QGen Compatibility Checker is a tool that allows the user to check if there all blocks inside a given model are compatible with QGen Generator. Reports show several warning messages.
3. SIL: QGen Debugger was used during the software-in-the-loop phase.
4. PIL: QGenpil tool was expected for the processor-in-the-loop purposes. No QGen packages of version 21.1, 21.2 or 22.0w provide tools for direct utilization of PIL testing.
5. HIL: The autogenerated code is integrated in an available Application Software of Euclid's project and tested at Sener premises. QGen does not actively participate in this phase other than to provide the source code.

The QGen release that is subject to the assessment of a Technology Readiness Level in the *WP6 Demonstration Viability Assessment* of AURORA corresponds to version that was exercised during the PIL phase, i.e., QGen Release 23.0w.

10. Annex A – QGen Findings

This section provides information about QGen findings observed during the progress on WP3.

Table 13 Finding #1: Simulation Model & Code parameters connection

Simulation Model & Code parameters connection	
ID:	#1
Component:	QGen Debugger
Version:	21.1. 22.0w
Type:	Bug
Reported:	Yes
Fixed:	Yes
Description:	Calling QGen Debugger directly from a model with parameters create a wrong connection between those arguments and their call inside the generated code
Workaround:	Workaround explained in Section 6.6

Table 14 Finding #2: _enable function/s dead code

_enable function/s dead code	
ID:	#2
Component:	qgenc
Version:	21.1. 22.0w
Type:	Feature
Reported:	Clarifies with AdaCore Support
Fixed:	-
Description:	qgec may generates so called *_enable function/s which essentially are not called anywhere from other code parts. This creates “dead code”. AdaCore Support explanation: *“The purpose of the enable function is to reset the memories inside Enabled Subsystems or Action Subsystems when the systems become active. This is also propagated through all the contained subsystems and contained model references.* *Since in incremental code generation mode we generate the code for the model regardless of where it was referenced from, we had to make the assumption that it could be called from any of the contexts above, so we generate the enable and disable function by default.”*
Workaround:	Use the --remove-unused-enable flag for the code generation with qgenc. AdaCore Support explanation:

_enable function/s dead code	
	"However, we do understand that in most cases this is not the case and those functions essentially become dead code. For that purpose, we have added the "Remove unused enable" flag ('--remove-unused-enable') in the code generation options to assess the entire model hierarchy to determine whether the enable function is actually needed for these subsystems/models or not. "

Table 15 Finding #3: C-syntax violation due to exact parameter definition

C-syntax violation due to exact parameter definition	
ID:	#3
Component:	qgenc
Version:	21.1. 22.0w
Type:	Bug
Reported:	Yes
Fixed:	Yes. Fixed in version 23.0w
Description:	When a top model contains two or more reference models with the same struct parameter definition, the generated code can contain C-syntax errors. This mostly appears as unfinished line statements with structure, array, or pointer references.
Workaround:	No workaround other than manually fixing it. From version 23.0w, this bug is fixed.

Table 16 Finding #4: Promotion of unfunctional qgenpil tool

Promotion of unfunctional qgenpil tool	
ID:	#4
Component:	qgenpil
Version:	21.1
Type:	Misleading information, Bug
Reported:	Yes
Fixed:	Yes – newer 22.0w does not contain any reference to qgenpil
Description:	The [RD01] contain reference to `qgenpil` tool, chapter seven. This tool revealed themself as unfunctional during this evaluation. AdaCore support team confirmed this information, quote: *"PIL support was experimental and is dropped for the time being."* The specific observed issues were: - The QGen Simulink GUI does not provide any option related to PIL - Using Matlab command line interface `qgen_pil_build` (line 167) reports undefined function `qgen_export_types`

Promotion of unfunctional qgenpil tool	
	- System command line interface `qgenpil` reports "QGEN BUG DETECTED" message
Workaround:	Use different tool for communication with the target HW, quote AdaCore support: *"However, to get it to run on the target, you need to do some manual work. The options are either:* *1) Use the tools provided by Simulink to communicate with the target.* *2) Solve the communication with the target in the S-Function wrappers."*

Table 17 Finding #5: Cumulating failed step results from different test-cases in one report

Cumulating failed step results from different test-cases in one report	
ID:	#5
Component:	QGen Debugger
Version:	21.1
Type:	Bug
Reported:	No
Fixed:	Ourselves – GPL
Description:	When two or more test-cases failed, within the report file, the failed test steps from previous test-cases are cumulated in the following test-cases reports. This way, it is not fully clear, which test-step failed in which test-case.
Workaround:	None: The issue was fixed manually within the QGen Simulation Framework code which were distributed under the GNU General Public License version 3 or later.

Table 18: Finding #6 Failure in QGen Model Verifier when model contains atan2 function

Failure in QGen Model Verifier when model contains atan2 fucntion	
ID:	#6
Component:	QGen/ QGen Model Verifier
Version:	21.1
Type:	Bug
Reported:	Yes
Fixed:	QGen 22.0w
Description:	When a model contains atan2 function, the Model Verifier fails by stating that file a-ngelfu.adb cannot be found. This file corresponds to the Ada.Numerics.Generic_Elementary_Functions package that QGen uses to implement atan2. QGen initially did not use this compiler while running Model Verifier.

Failure in QGen Model Verifier when model contains atan2 fucntion	
Workaround:	None: update to next version

Table 19: Finding #7 Parameters with bus structure

Parameters with bus datatype	
ID:	#7
Component:	QGen
Version:	From 21.1 (presumably from even previous versions)
Type:	Feature
Reported:	Yes
Fixed:	-
Description:	QGen does not support parameters with bus datatype.
Workaround:	Parameters shall be entered via mask with one of the two following options: • Use a separate parameter for each member of a bus structure. • Use a Matlab variable with structure instead of parameter (this solution is the one used)

Table 20: Finding #8 Lower/Upper case letters mismatch for integrated external code

Lower/Upper case letters mismatch for integrated external code	
ID:	#8
Component:	QGen, QGen Debugger
Version:	21.1, 22.0w, 23.0w
Type:	Bug
Reported:	No
Fixed:	-
Description:	When the external code (e.g.: with enumerators definition) was in included (at the model's level), the folder with generated code included those c-files although its naming was transformed to lower-case letters only. On the other hand, all relevant include statements to those external c-files with enumerator definition did not change its naming and kept original lower-upper-case letters. This created unresolved include statements.
Workaround:	Change manually those unresolved include statements to match the exact file name (lower-case letters only).

Aurora

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