Aurora

Flight SW Autocoding Life-cycle process (Software-in-the-loop)

D4.2

Document Code: AUR-SEN-RP-00032
Document Version: 1.1
Document Date: 02/06/2023
Internal Reference: DOC00231005

sener

POLITÉCNICA

SPACE

esc Aerospace

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
J. Gómez	A. Rodríguez	R.M León	A. López	A. Rodríguez
Date and Signature	Date and Signature	Date and Signature	Date and Signature	Date and Signature

Signature not needed if electronically approved by route

© AURORA Consortium, 2023

Changes Record

Rev	Date	Author	Affected section	Changes
1.0	2022-02-18	J. Gómez	6	Added Software in the Loop Section
1.1	2023-06-02	J. Gómez	1	Removed chapter 7. Clarification added in the introduction section.

© AURORA Consortium, 2023

Index

© AURORA Consortium, 2023

© AURORA Consortium, 2023

1. Introduction

1.1. Purpose

This document describes the Flight SW life cycle for autocoding and the different processes and stages of a model-based process which cover the whole SW life cycle from requirements to qualification.

The procedure is mainly focused for AOCS/GNC SW which has been selected as the primary use case of the project, but it can be adapted to other subsystems as well.

The next steps in the Flight SW life cycle are the Processor-in-the-Loop and Hardware-in-the-loop phases that are described in separated documents, D4.3 and D4.4 respectively.

1.2. Scope

The Flight SW autocoding life-cycle process definition is the main core of the *WP4 Flight SW Autocoding Life-cycle Process Definition* of AURORA, as described in Annex 1 Part A of [AD1]. The document gathers the main process for the SW generation toolchain departing from the System requirements up to complete qualification, detailing it for the different stages of a typical software verification process

This document is an output of the T4.2 activity included in WP4. Future version of this deliverable will be provided as outputs of T4.3 and T4.4.

This document is based on previous output of WP4, document D4.1 which refers to Model in the Loop. This document updates the Software Autocoding Life Cycle with the inclusion of the Software in the Loop Stage. Later deliverables of WP4 will include the Processor in the Loop and Hardware in the Loop phases.

1.3. Document structure

The document has been structured as follows:

- Section 1: this introduction
- Section 2: Related documentation
- Section 3: Overview of the AURORA methodology
- Section 4: Flight SW Autocoding Life-Cycle Process
- Section 5: Model-in-the-loop stage
- Section 6: Software-in-the-loop stage

© AURORA Consortium, 2023

2. Related documentation

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

2.1. Applicable documents

AD #	Title	Project Reference	Issue	Rev
[AD1]	AURORA Grant Agreement	GA number 101004291	-	-
[AD2]	AURORA Consortium Agreement (CA)	CA Nº 101004291 AURORA	-	-

Table 1 Applicable documents

2.2. Reference documents

RD #	Title	Reference	Issue	Rev
[RD1]	Space engineering Software	ECSS-E-ST-40	C	-
[RD2]	Space Software Product Assurance	ECSS-Q-ST-80	C	-
[RD3]	Software Engineering Handbook	ECSS-E-HB-40	A	-
[RD4]	Guidelines for the Automatic Code Generation for AOCS/GNC flight SW Handbook. Vol1 – General concepts	-	1	0
[RD5]	AOCS/GNC Modelling Guidelines	AUR-SAE-RP-0006	1	1
[RD6]	Guidelines for the Automatic Code Generation for AOCS/GNC flight SW Handbook. Vol2 – Mathworks specific guidelines	-	1	1
[RD7]	QGen Evaluation Report	AUR-ESC-RP-0007	4.1	1

Table 2 Reference documents

© AURORA Consortium, 2023

2.3. Acronyms

Acronym	Description
AD	Applicable Document
ATB	Avionics Test Bench
COTS	Commercial Off The Shelf
EBd	Executive Board
ESE	Engineering Simulation Facility
FES	Functional Engineering Simulator
GA	Grant Agreement
GeA	General Assembly
HILF	Hardware-In-the-Loop Facility
HW	Hardware
MIL	Model in the Loop
N/A	Not Applicable or Available
PFM	Proto Flight Model
PIL	Processor in the Loop
RD	Reference Document
SDP	Software Development Plan
SIL	Software in the Loop
SRR	System Requirements Review
SVF	Software Verification Facility
SW	Software
TRB	Test Review Board
WP	Work Package

Table 3 Acronyms

2.4. Terms and definitions

N/A

© AURORA Consortium, 2023

3. Overview

The AURORA WP4 "Flight SW Autocoding Life-cycle Process Definition" [AD1] approaches the definition of a SW Autocoding Life-cycle Process, where Autocoded system refers to any Complex-Models systems that make a full use of MATLAB/Simulink for modelling the algorithms and behavior of the system. The most representative case of such a system in Space missions are the AOCS/GNC systems. In our approach Design and Development and running chained to verification activities and therefore improving the OBSW integration and validation program.

This approach is supported by:

- An early verification of the navigation models.
- Auto-generated source code software following an iterative process.
- Mission requirements Verification at GNC model level and component model.
- Integration phase when OBSW components implement standard interfaces (API).
- Aligned with Space standards and allowing as much as possible the automation of the process.
- Iterative execution of the WP taking inputs from the technology Demonstrator activity.

The Model-in-the-loop (MIL) , Software-In-the-Loop (SIL) and Processor-In-the-Loop (PIL) are key points of the incremental validation in order to verify the behavior of the GNC code in a representative environment and to identify computational resources required through code profiling.

The whole process is iterative. This means that it is applicable several times for each function/mode iteration. The functional iterations are defined e.g., for a subset of functions that can be easily validated independently. For example, an AOCS iteration is associated with an AOCS mode. In the following, the subsystem of choice is the AOCS, but could be any functional chain subsystem expressed with models having Autocode capability (e.g., thermal, power).

This activity enclosed the definition of following In-the-loop steps:

- Model-in-the-loop

  The models have to comply with Aurora modelling standards and guidelines, (QGen framework) and the model simulations demonstrate the feasibility of the preliminary design and the robustness of the selected solutions using Monte Carlo test campaigns. Being able to perform such tests during the preliminary stages of the development allows for efficient iterations at system level, giving valuable contributions for trade-offs that involve other subsystems.

- Software-In-the-Loop

  The auto-coding of the navigation model (QGen framework) will allow testing the Autocoded SW with respect to the algorithms already validated in a MIL environment.

- Component-In-the-Loop

  The SW as an OBSW component has to follow the AURORA standard API, therefore the SW is integrated into a wrapper that implements the API for getting the services provided by the algorithms of the model-based design GNC and reacting to its outputs (CBI component model). The TASTE/QGen tool suite is used to compile, link and execute the components software.

  The TASTE/QGen tool suite is used to compile, link and execute the software.

- Platform-In-the-Loop

  To validate the SW component running in the execution platform connected to an open-loop-environment, typically using an Avionics Test Bench (ATB) equipment.

© AURORA Consortium, 2023

The process is iterative, and any error or change is done at model-level only and implies to iterate previous In-the-loop steps.

© AURORA Consortium, 2023

4. Flight SW Autocoding Life-cycle Process

The space SW generation procedure has traditionally relayed on a linear approach based on manual coding of the SW functions, which departs from the requirements coming from the top-level system, which are derived into SW requirements. From them, a SW architecture is defined, and further requirement levels might be derived. Then, an implementation procedure follows, which is lately checked and verified at the different levels, from unit to integrated architecture, in different facilities. Moreover, the SW is checked for readiness, correctness, maintainability, trying to detect implementation errors beyond those that can be detected by test. This process is tedious and implies a big number of resources.

For AOCS/GNC, the main use case included in AURORA, this traditional process was composed of two parallel workflows with different stages:

- Matlab/Simulink Models:

  This workflow relies on the implementation of Simulink models to define the GNC algorithm for the SC. It consists of the following steps:

  - Definition of requirements, which is common to the other workflow. Departing from the system requirements some requirements are derived to the GNC algorithms.
  - Model prototyping, developing the basic GNC algorithms to cover the mission/system needs. This covers the preliminary design.
  - Model detailed design. This includes the refinement of the models and the formal verification campaign using a representative simulator. This stage finishes the model workflow.

- Manual SW implementation:
  - Definition of requirements, which is common to the other workflow. Departing from the system requirements some requirements are derived to the SW requirements.
  - From the algorithm implementation in the preliminary design phase, the SW requirements are refined to include compatibility with the outlined design.
  - Based on the SW requirements, the manual part of the SW not depending on the GNC algorithms is implemented. Once the preliminary design is over, a first GNC coding is performed and integrated and tested together with the other SW part.
  - After the detailed design phase, the SW is refined introducing some updates and the details coming from GNC algorithms. A SW validation campaign is performed in a representative simulation environment.
  - Then the generated SW is integrated within the system facilities and an extensive verification campaign is run (SIL, PIL, HIL).

© AURORA Consortium, 2023

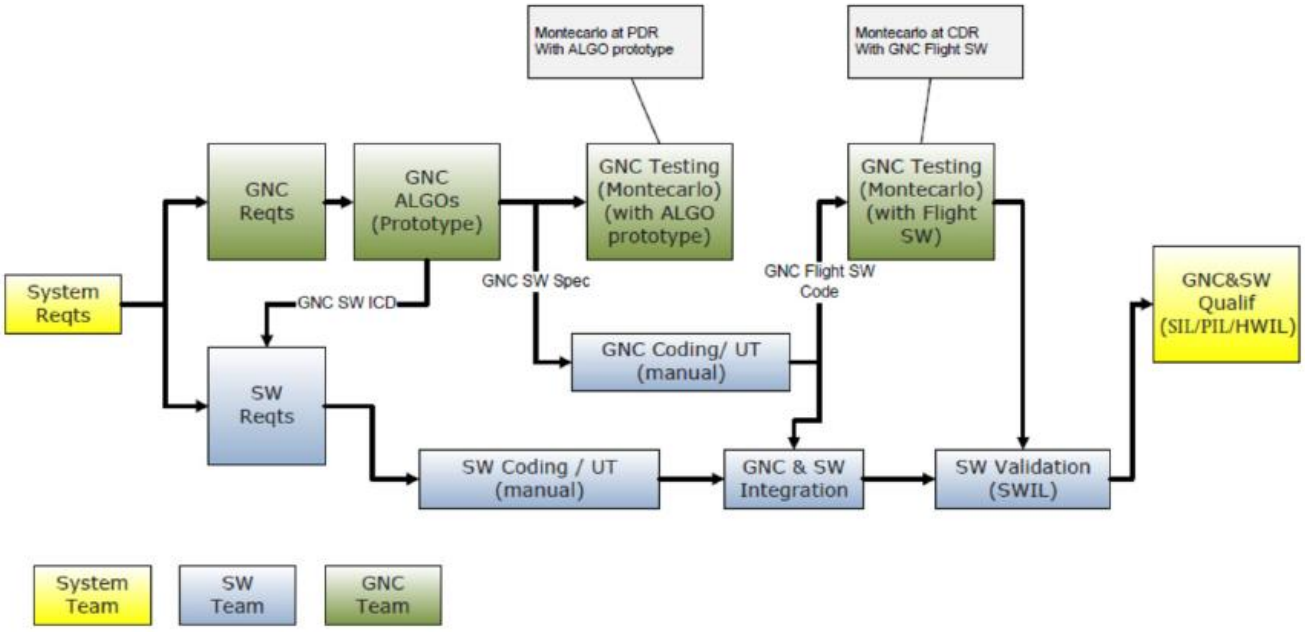

Figure 1 Traditional GNC SW development with manual coding (from [RD4])

This traditional workflow normally takes large implementation times, is prone to human errors which are difficult to track and debug and it is therefore more expensive and less reliable.

An alternative to use this manual based process, relies on autocoding techniques applied to models, in a model-based approach targeting a simplified and more reliable procedure, reducing the implementation times, the number of errors and increasing maintainability, readiness and comprehensiveness.

For AOCS/GNC the use of this model-based approach is the natural evolution of the abovementioned manual procedure, since the models have been already used in the past and can be used as baseline architecture and SW implementation, by using the appropriate autocoding conversion tool.

This document gathers the different processes and stages of this model-based process which cover the whole SW life-cycle from requirements to qualification.

The Table 4: Test facilities definition summarizes the main stages and facilities of AOCS/GNC validation.

Verification Stage	Facility	Comment
MIL	**FES** **Functional Engineering Simulator**	Model of the GNC algorithms implemented in a simulation framework (Matlab/Simulink)
SIL	**FES** **Functional Engineering Simulator**	Software produced from model is connected to a spacecraft simulator to demonstrate that software is still requirement compliant
PIL	**SW Test Bench**	SW is executed on a real OBC, which is connected to a Real Time Simulator (RTS). This stage is done to verify computing budget usage
SVF	**SW Validation Facility**	The AOCS/GNC software is executed with the whole on-board software into a model of the OBC

© AURORA Consortium, 2023 PUBLIC

Verification Stage	Facility	Comment
HWIL	**FUMO (Functional model)** **ATB (Avionics Test Bench)** **PFM (Proto Flight Model)**	Final on-board software is run with some real avionics equipment with some spacecraft simulator, which closes the loop

Table 4: Test facilities definition

The complete software development cycle is presented in Figure 2: Autocoding vs Manual SW development cycle, where the different milestones and documents to be reported are listed at every milestone.

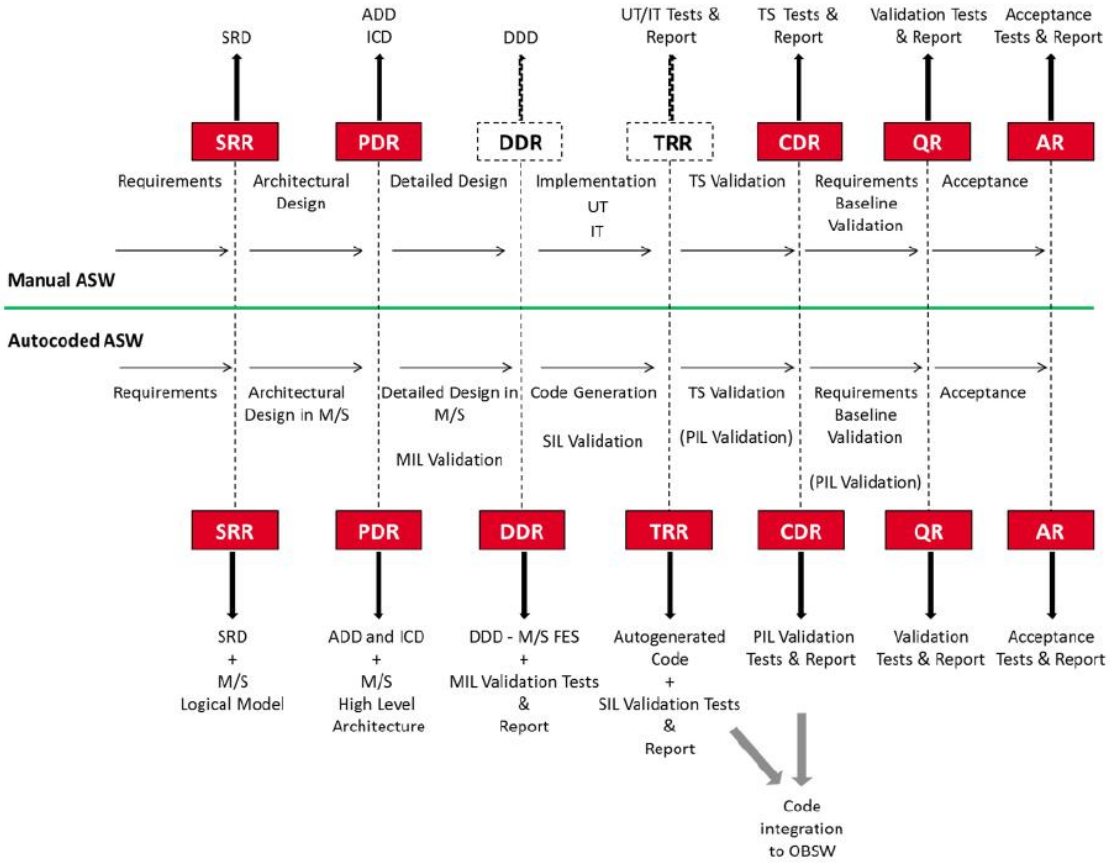

Figure 2: Autocoding vs Manual SW development cycle

© AURORA Consortium, 2023

5. Model-in-the-Loop Stage

This stage is focused on the generation of Matlab/Simulink models, compliant with requirements and early validated with tests that will result in the generation of an automatically generated code with QGen, representative of the original model.

This stage is quite similar with the traditional approach of manual code generation, however some difference in the process is observed due to the earlier availability of the AOCS/GNC SW. At the beginning of the process the following documents shall be prepared:

- ICD: joint work shared between the GNC and SW team in which the data flow and frequency required by the GNC specification is taken into account. Definition of the code generator settings are defined in this document. The GNC engineer is no longer blind to the software side of the process and shall have some insight on the final autogenerated code.
- Model Requirements Specification: document used to design and implement the GNC algorithms based on the GNC requirements specification.

Once that those documents are issued for the SRR milestone (System Requirements Review), the Model- in-the-Loop begins and the generation of the models can start. In this step, the GNC engineer is being supported by a Modelling Guideline Handbook, which gathers industry modelling standards that are recommended to follow for a later easy integration and model maintainability. For a generic Simulink guideline for autocoding model generation, please refer to [RD5] and [RD6].

For Aurora's scope, a custom set of guidelines was generated ([RD5]). These new guidelines are Euclid heritage and were modified to account for QGen limitations i.e., limitations in terms of Simulink block constraints for instance.

The resulting Simulink will apply the algorithms specified in the Model Requirements Specification. In parallel, models representative of the real word, such as DKE models, sensors or actuators shall be developed and ready for performance test.

These model algorithms are then subjected to testing in order to ensure compliance against mission requirements, to identify bugs and to ensure sufficient model coverage. Note that model coverage is not the same as code coverage. Nonetheless, typically, large model coverage implies large code coverage, something to be seek in later stages of software validation. Two different test scenarios are defined:

- Unitary Integration Test of the individual models
- Verification of the AOCS/GNC performance requirements on a validated FES with representative test cases. This campaign typically includes a full Monte Carlo campaign.

5.1. Unitary Integration Test

Testing starts at unitary level, where Unitary Integration Tests are defined by the GNC engineer. This UIT are developed to cover all the functionalities implemented in each function, to test boundaries and to verify requirements allocated to unitary level. These tests can be considered as the classical bottom-up approach in which a set of pre-defined inputs are fed to the model in open-loop simulations.

For each AOCS mode, the UIT campaign will start with the deeper models (leaf models), which are hierarchy tested in the first place. These leaf models are isolated from the rest of the models. Once that the model has been properly tested and its behavior has been properly assessed, the process continues with upper levels, aggregating the previously tested models. Following this procedure, if a top model test fails, it can be safe to assume that the lower models do correctly behave.

The typical procedure of generating the UIT is via test harness, in which the model to test is placed into a model reference block where inputs are fed, and outputs are collected for a final PASS/FAIL evaluation according to the test specification. I/O signals shall be collected for later verification campaigns (SIL/PIL) as those will be used as a confirmation that the autogenerated code behaves as models, that is, same inputs results in same outputs.

© AURORA Consortium, 2023

In this sense, the AOCS models are used as Technical Specification for the auto generated code as the software behavior is validated against model behavior. Library functions used in the model development shall also be unitary tested.

Inputs can be fed via Simulink Signal builder, which allows for easy change of inputs signals and creation of various signal groups. This method allows for an effective and easy execution of the unitary test with easy signal replacement without effectively changing the test harness.

Collection of the I/O can for each test case is typically done via the Signal Logging capabilities of Simulink, where data is automatically stored as a Simulink Dataset variable, although the user is free to choose the most suitable signal save option for their need.

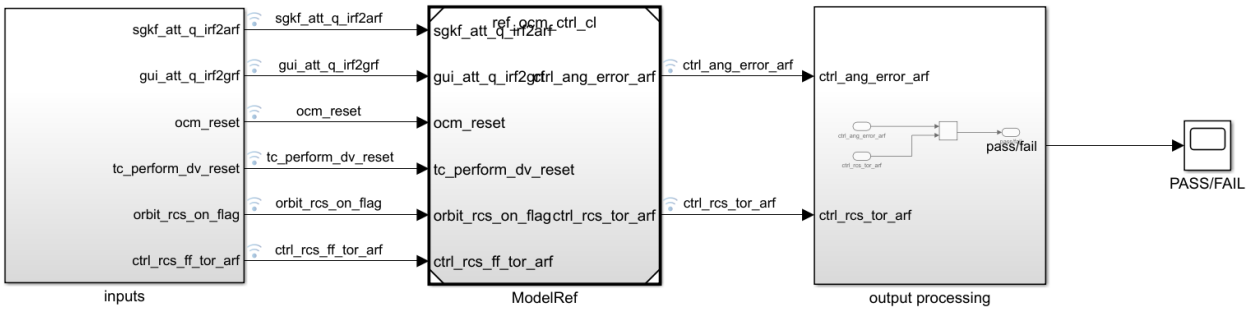

Figure 3: UIT Model harness

Steps

a) Generate UIT specification, defining what inputs and outputs are expected
b) Generate the test harness with the following components
    a. Input block
    b. Model reference block
    c. Output block with PASS/FAIL criteria
c) Run MIL UIT to validate correct test implementation
    a. In case of FAIL, review test case implementation and repeat the MIL execution test
d) Gather I/O signals for MIL-SIL comparison
e) Report results obtained in the corresponding section of the Test Report

In conclusion, UIT are open-loop test cases defined for an early verification of the GNC algorithms and requirements at unitary level with the addition of a preliminary model coverage. Once the model's behavior has been tested at unitary level, then, they can be included inside a simulation architecture for requirements verification.

5.2. Performance cases in FES

A Functional Engineering Simulator is a simulation environment whose purpose is the verification of the AOCS/GNC models. This simulator is in charge of managing the different test and mission scenarios specified, being also a direct support of the software development.

© AURORA Consortium, 2023 PUBLIC

The main components of a FES architecture are:

- *Simulation Engine*: responsible of the definition of a simulation scenario, definition of the mission and the configuration of the models to be simulated. Parameters are configured and pre-processed to obtain simulation parameters, which are set in the mask parameters.
- *Simulation Core*: Simulink templates that is customized for each operation mode. Different Simulink libraries containing the GNC algorithms are present so that the template can replace the adequate models.
- *Monte Carlo Simulation*: functions that manage the configuration and control of Monte Carlo simulations, generating perturbed values of the model parameters and controlling the storage of the raw data
- *Post-processing*: functions to post-process the raw data obtained. This component typically generates representative plots and graphs needed for AOCS/GNC validation.
- *Failure injection*: component in charge of injecting failures in the simulation to check failure conditions or FDIR algorithms. Typical failure comprises of freeze signal, set a signal to a desired value or linear signal behaviour.

It is important to remark that the FES itself must be validated according to a Software Verification and Validation Plan, which complies with the ECSS-E-40 standard.

Unlike UIT, test cases are run in closed loop, including, not only the GNC models generated and unitary tested, but the real word representative models (DKE, sensors and actuators), which were previously validated to ensure good overall performance.

Test cases in a FES are no longer defined as a set of inputs, but as a timeline file that it is read by the simulation engine. This timeline defines the set of initial conditions and the operational timeline, which defines the set of commands to be followed. This timeline is typically defined as an external file, XML file for instance, however, this file is simulator dependent.

Test cases are defined in order to verify that the system is compliant with the requirements specified in the SRR. These tests may include single shots runs with simulator parameters adjusted for adequate testing or Monte Carlo simulations, with the perturbation of relevant parameters.

Once that the results of the test have been formally verified, reported and accepted in Test Reports, the PDR closes this stage.

5.3. Code Generation

Code generation will be further discussed in the next sections as this process belongs to the SIL campaign, nonetheless it is close related to the model development, so a brief insight is presented here.

After running the complete MIL campaign verification, the code generation process starts. Autocoding tools such as Simulink Coder toolbox or QGen can be used to translate the model architecture into C code software files, which can be embedded into a software testing facility for the SIL campaign.

The proposed approach here is that the autogenerated code **shall not** be manually modified at any level. In case of some bugs identified during the software verification process that require correction, the solution shall be applied to the model, being the autocode process regenerated. This is done to ensure that the models and the code generated from them are always align and the AOCS team and Software team can maintain their own process with no major differences. An assessment of the tests to be repeated is done to ensure that the modifications to the models do not imply fail tests.

© AURORA Consortium, 2023

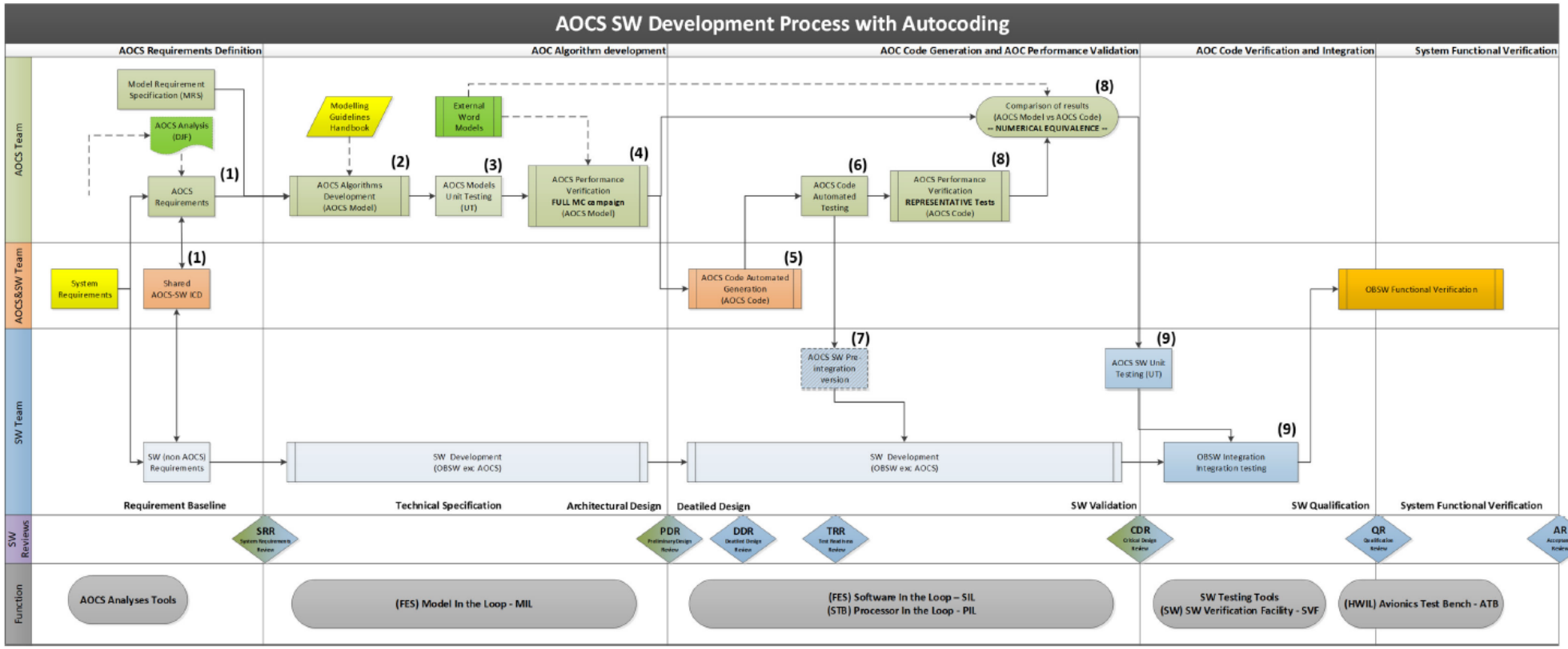

Figure 4 ESA proposed development life-cycle for AOCS/GNC SW (from [RD4])

© AURORA Consortium, 2023

6. Software-in-the-Loop Stage

Software in the loop (SIL) is the next stage of the validation and verification process of a piece of software after Model in the Loop (MIL). This stage comprises the static and dynamic analysis of the autogenerated code, in which metrics about code performance and adherence to standards are extracted. In this document, it is assumed that the code language to analyze and check is C as one of the main languages in the space industry . From now on, every time source code is mentioned, it is understood that it is written in C language. Ada is also a programming language spread in the generation of space software, nonetheless, being Matlab, the most employed programing tool for algorithm generation, written and autogenerating code in C language, this is the one to be analyzed.

Autogenerated code is generated from Matlab/Simulink models which shall follow Savoir Guidelines for Automatic Code Generation Vol 1 [RD4] and Vol 2 [RD6]. The first volume is dedicated to the general concept of development and verification, which can be traced to ECSS E-40 and ECSS Q-80, two industry standards collecting the best software engineering practices. The second volume is dedicated to the AOCS modeling, from general modeling guidelines to configuration of the model for code generation. The whole process of software development, from system definition requirements to the final deployment on the on-board computer is supervised by the DO-178C standard.

In the scope of the Aurora project, modelling guidelines [RD5] have been adapted from Euclid project with modifications related to QGen constraints. This modelling guidelines are defined according to industry standards.

At the time of defining the mission requirements, it is important to define the criticality of the software to be developed. This is crucial, as different level of criticality requires different levels of verification methods or even a tailoring of the guidelines can be considered.

Criticality	Definition
A	Function that if not correctly performed results in catastrophic consequences
B	Function that if not correctly performed results in critical consequences
C	Function that if not correctly performed results in major mission degradation
D	Function that if not correctly performed results in minor mission degradation
E	Function that if not correctly performed, does not alter the correct system behavior, and does not impose additional ground/pilot workload

Table 5: Software Criticality Definition

The definition of the software criticality levels implies the minimum number and types of analysis required to obtain a qualifiable software product. It does not impose any condition on additional test that can be performed on a piece of software.

The SIL campaign can be subdivided into two phases depending on the type of analysis to be performed on the code, either static analysis or a dynamic analysis. It is important to remark that, although the Embedded Coder tool from Mathworks or QGen from AdaCore may ensure some warranties in the final code about coding standards or metrics the code fulfill, the software analysis must be performed.

6.1. Static Analysis

The static analysis is the analysis of the code without executing the application, that is, an analysis of its structure and syntaxis. This phase is typically used to detect security vulnerabilities, performance issues and non-compliance with standards. It is typically done by searching the source code to identify specific coding patterns.

© AURORA Consortium, 2023

Some of the most common metrics to track inside a code are:

- Cyclomatic complexity.
- Nesting level.
- Number of statements.
- Comment frequency.
- Code size.

The cyclomatic complexity is a measure that determines the stability and level of confidence of a software product. Lower cyclomatic complexity implies easier to understand codes and less risky to modify them. This metric was initially created with manual codes in mind, nonetheless, due to how autogenerated codes are structure and written, typically in a more cumbersome way, this metric returns a higher value, and it is treated with less impact on the static analysis phase. The rest of the metrics serve as a way to comply with code metrics defined in the Quality Assurance Plan.

It is important to remark that these metrics do not consider how the code is written, but how it is structured. Some coding rules are required to ensure a quality product. DO-178C imposes the adherence to appropriate coding rules for safety-critical applications, ensuring safe- coding practices that will result in easier and more efficient work. Although DO-178C does not prescribe the adoption of MISRA C rules for coding standard, its use has become so extended, it has become the coding rules standard to follow.

Originally released in 1998, the Motor Industry Software Reliability Association, MISRA C, has evolved and different versions have been published with updates. The current version to be followed is MISRA C:2012, containing 143 rules and 16 directives. Over the years, MISRA C:2012 has evolved with two Amendments (1 & 2), which expands the rules and directives to a total of 158 rules and 17 directives. MISRA coding guidelines can be divided into four categories:

- **Mandatory guidelines**: guidelines which violation is never permitted.
- **Required guidelines**: guidelines which can only be violated when supported by a deviation defining a set of clear restrictions, requirements, and precautions.
- **Advisory guideline**: recommended guidelines to be followed. Violations are identified but are not required to be supported by a deviation.
- **Disapplied guidelines**: advisory guidelines which are ignored.

At the beginning of the project, a re-categorization plan is established as a statement of detailing how the guidelines are being applied to the software product. It is in this plan, approved by customer and supplier, where disapplied guidelines are defined and where some of the required or advisory guidelines are promoted to mandatory based on the relative importance those guidelines have on the scope of the project. The conclusion of this plan is collected into the Guideline Compliance Summary (GCS), as summarized in ***Table 6***.

Misra Category	Compliance levels claimed by the GCS			
Mandatory	Compliant	-	-	-
Required	Compliant	Deviations	-	-
Advisory	Compliant	Deviations	Violations	Disapplied

Table 6: Compliance levels claimed by the GCS

The MISRA C software compliance can be manually checked for smaller projects but, as the project grows, the code quickly becomes hard to track manually and external tools are required. Fortunately, a great variety of tools are able to statically covers all the MISRA C rules and directives. Some examples are:

- Matlab offers Polyspace tool as a way to check MISRA compliance among other standards. Nonetheless, the Embedded Coder tool, capable of generating C code from Simulink models, grants integrated support with MISRA or AUTOSAR standards.
- Cppcheck, a versatile free software under the GNU General Public License.
- Parasoft C/C++ test, which can be integrated as an extension for Visual Studio.

© AURORA Consortium, 2023

- LDRA (Liverpool Data Research Associates), not only offers MISRA compliant checks, but also includes coverage analysis up to level A.

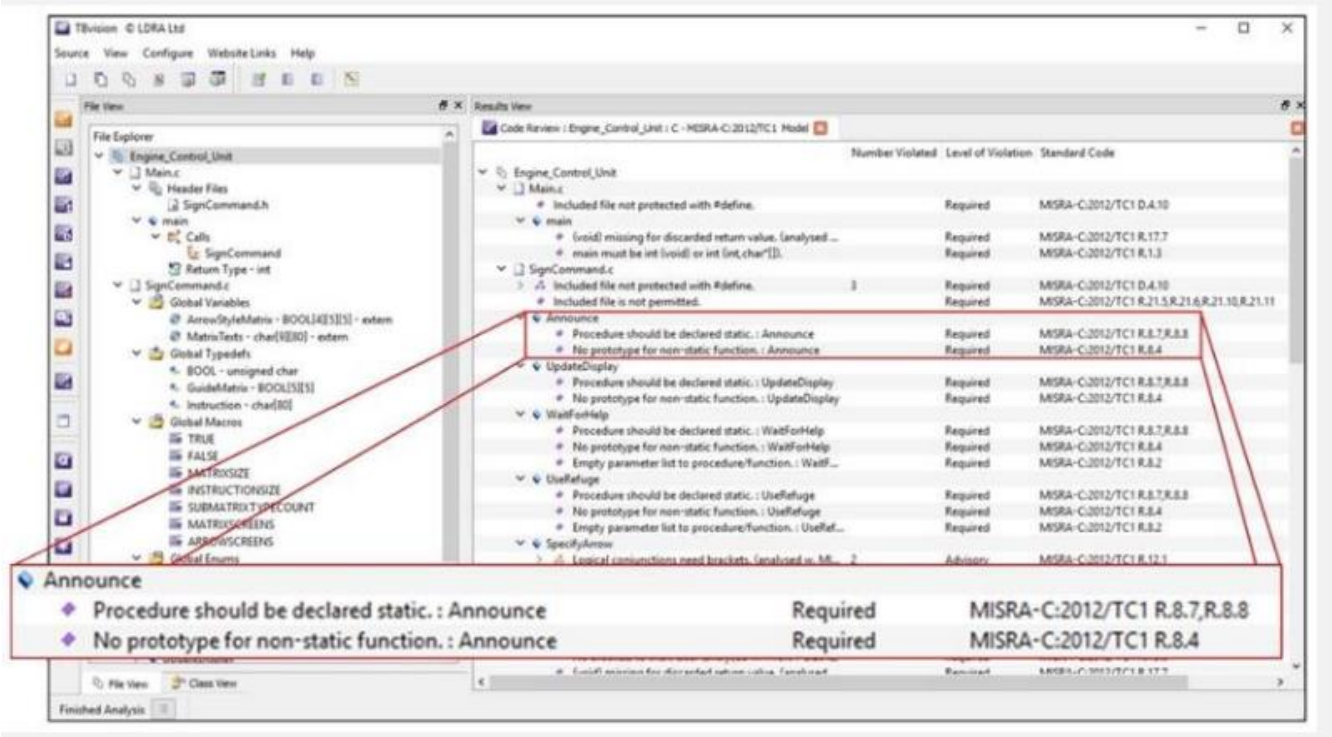

Figure 5: MISRA C checker in LDRA

Of course, the selection of the static analysis tool will vary from project to project, depending on the scope of the project, the availability of funds for licenses, expertise of the company in certain tools, etc. Regarding the AURORA project, the code, generated by QGen has been analyzed by two different tools, cppcheck and LDRA. In addition, AdaCore company uses CodePeer as a static analysis tool for the validation of QGen, although this tool is developed for Ada source code, not C code.

6.2. Dynamic Analysis

The dynamic analysis of the code requires running the code in a software testing environment capable of executing the required test specifications. This is done to ensure correctness in the generation of the code, requirement verification and code coverage analysis.

In classical approaches, the code is manually generated from the models by a software team. Based on the models, the team generates a software piece that shall ensure compliance with requirements. As the software code is manually written, it is also necessary to adapt existing test specifications or even create new ones to ensure requirement compliance.

With the autogenerated code, the process of requirement validation is substituted by a numerical comparison between the results obtained from a model in MIL and the same test run in SIL. As the requirements have already been validated at model level, a numerical comparison ensures whether the code has been correctly generated from the model, leading to a direct requirement compliance.

6.2.1. MIL-SIL comparison

The way to proceed and test the generated code is to first gather the inputs and outputs from the unitary MIL simulation tests. It is assumed that the test specified for the MIL phase verify the model requirements at unitary

© AURORA Consortium, 2023
PUBLIC

level and are considered to be PASSED. Once that you have gathered these I/O, the user can proceed to feed the generated code with the inputs of the unitary tests. Outputs are gathered and compared to the outputs previously obtained in MIL.

Although an exact numerical comparison is possible given a correct configuration of the compiler, the software libraries, or the environment, it is expected some numerical discrepancies between the two different simulation environments. It is the task of the AOCS/GNC engineer to define a numerical threshold to consider that a test has passed and that the discrepancies do not alter the AOCS performance. As a rule of thumb, differences lower than 1E-12 are consider acceptable. Nonetheless, this is case-dependent and should be analyzed by an expert, as the threshold can vary.

The process of running the SIL simulations depends on the selected environment. Matlab/Simulink for instance, offers the possibility to purchase the Embedded Coder license, which allows the generation of the code and SIL, PIL capabilities inside Simulink. This is especially useful as it is easy to run a MIL/SIL comparison given a model harness and two signalbuilders, one for inputs to the SIL model and one for the MIL outputs, which are used for comparison.

In the scope of the Aurora project, the selected environment to run the SIL campaign is GNAT studio, which is an Integrated Development Environment (IDE) for software development. It can be used for testing the autogenerated code by QGen as it is integrated in GNAT. Nonetheless, running the SIL campaign in GNAT is not as straightforward as in the case of using Simulink Embedded Coder since a transition from Simulink to GNAT must be done.

The process to follow for testing the SIL campaign in GNAT starts in Simulink, where a debugger model is generated. This model consists of:

- The original Simulink model, which is placed inside a Model Reference block.
- An input block, in which a Signalbuilder is placed. This block allows for including some predefined signals which serves as input to the model. Also, the data is automatically configured to the right data type and dimensions, based on the input ports of the original model.
- Scope block to graphically view I/O from simulation.

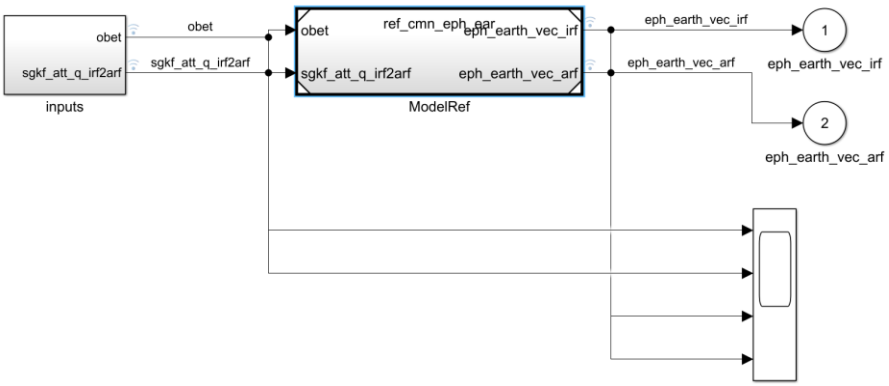

Figure 6: QGen Simulation model

The model is automatically configured foe debugging options, that is, fixed-step solver with discrete continuous times and a fixed step taken from the original model. I/O signals are logged to register and save the data to be later used. Once the inputs are written in the Signalbuilder block, the GNAT studio is launched, code generated, and simulation is run. For a more detailed view of the process, please refer to the QGen Evaluation report [RD7].

Outputs of this SIL campaign is the MIL-SIL comparison between model and code. Again, the required tolerance to consider a PASS criterion is model dependent and shall be studied by the AOCS engineer.

© AURORA Consortium, 2023

6.2.2. Coverage Analysis

The purpose of coverage analysis is to help find areas of a program not exercised by test cases which can potentially contain bug errors, find areas of code which cannot be exercised and creates unnecessary memory allocation and identify redundant test cases.

In Section 5, it is explained the concept of model coverage and how the defined test shall aim for a full coverage. The idea behind defining tests which are targeted for high model coverage is based on the idea of code coverage. There is not a one-to-one correlation between model and code coverage (100% model coverage may not imply 100% code coverage), nonetheless, they are extremely coupled, as the code is generated from the model.

Before digging in industry standard about coverage, some of the definitions for coverage analysis are:

- Condition: it is a Boolean expression. It cannot be broken into simpler Boolean expressions.
- Decision: a Boolean expression composed of conditions and zero or more Boolean operators.
- Statement coverage: measures which of the code lines are executed

$$Statement\ Coverage = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements} \times 100$$

This measurement allows to determine what are the unused or unreachable statements or complete branches of our code, as well as dead code.

- Decision coverage: this measurement reports the true or false expressions. The goal of this metric is to cover and validate all the accessible code by checking and ensuring that each branch of every possible decision point is executed

$$Decision\ Coverage = \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decision\ outcomes} \times 100$$

- Branch coverage: this metric evaluates which every outcome from a code module is tested. That is, ensuring that each decision condition from every branch is executed.

$$Branch\ Coverage = \frac{Number\ of\ executed\ branches}{Total\ number\ of\ brances} \times 100$$

Example: if outcomes of a decision are binary, True and False evaluations are required.

- Condition coverage: this metric evaluates the variables of subexpressions in a conditional statement. The goal is to check individual outcomes for each logical condition.

$$Condition\ Coverage = \frac{Number\ of\ executed\ operands}{Total\ number\ of\ operands} \times 100$$

Example: in the line of code "if (x<y) AND (a>b) THEN", there are 4 possible combinations (TT, FF, TF, FT). Each 4 of them shall be tested for a 100% condition coverage

- Modified Condition/Decision Coverage: every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision's outcome independently by varying just that condition while holding fixed all other possible conditions.

Coverage analysis is required by the industry standards, ECSS-E-ST-40C and DO-178C, as a way of software verification. The level of code coverage analysis depends on the criticality of the system. For software with criticality below D, there is no need to demonstrate code coverage results. For the rest, the Table below shows what must be done in order to certificate a software product:

© AURORA Consortium, 2023

Criticality	Functions	Calls	Blocks	Condition
C	With exists	Yes	Statement blocks	None
B	With exists	Yes	Statement and decision	None
A	With exists	Yes	Statement and decision	MC/DC

Table 7: coverage condition based on criticality levels

The aim of the coverage analysis is to reach 100% coverage for all the coverage types required by the industry according to the criticality level. Results of the analysis are reported in a Software Verification Report in which the code which does not reach 100% shall be properly assessed and agreed with the customer. For instance, when generating a model, safety requirements demand default option for switch cases or if statements, even though this default option may not be possibly exercised.

The coverage analysis for an autogenerated code starts from the definition of test cases for each individual model, that is, in the MIL phase. With the definition of the test cases and the gather of the I/O for those cases the coverage dynamic analysis can start. This process is typically an iterative one as it is not expected to reach 100% in the first iteration.

The process can be described as follows:

1. Define test cases in Model in the Loop. Test cases are defined in a test specification document and must ensure requirement verification at AOCS level.
2. Gather I/O for those cases. Outputs for coverage analysis are not required as the focus of this analysis is not to check outputs of simulation, but which lines of codes and branches are executed.
3. Analyze the coverage of the autogenerated code with a coverage analysis tool.
4. Check which execution paths are not covered and define new test cases to cover those lines.
5. Rerun the coverage analysis (steps 1 to 4) until reaching 100% and justify the results.

It is important to remark that the coverage is analyzed at model/unitary level. The coverage shall be analyzed by unitary test cases defined for a specific module of the final product, that is, the code generated from a unitary model. As an example, if a model A, contains model B with some additional algorithms, the coverage analysis of model A should not include model B, as that model will have its own dedicated set of test cases.

There are multiple tools that measures the coverage results of a given code. The selection of a coverage analysis tool shall be based on costs, the criticality level of the software, interoperability with the test environment, code language, etc.

Some of the most common tools for coverage analysis are

- Gcov: a free open-source coverage analysis and statement by statement profiling tool
- Bullseye Coverage: which includes condition/decision coverage analysis and the ability to merge results from distributed testing
- LDRA: previously mentioned in the static analysis section. This tool, apart from offering condition/decision coverage for software products with criticality level A, also offers the possibility of automatically generate test cases that provide 50-80% coverage.
- Other tools: Coco, Parasoft Jtest, Testwell CTC++

The main driver on the selection of the coverage tool is criticality of the software to be qualified. Higher criticality software products require validation and verification methods that some tools may not offer.

The typical range of prices for a license is 500-800$ yearly. This range of prices does not apply to gcov, since it is a free software, and LDRA, which prince range is between 3000-5000$, depending on the number of seats. This price is understood by the number of options and verification tools LDRA offers, being one of the most complete tools for critical software verification.

© AURORA Consortium, 2023

Statement Coverage Profile

```
================================================================================================================================
LINE NUMBER                                                                                    PREVIOUS  CURRENT
REF. (SOURCE)     STATEMENT                                                                    RUNS      RUN       COMBINED
================================================================================================================================
317 (19)        void                                                                           -         -         -
318               MLFlagnConsecutiveTimes_pZXaSYvB_Init (                                      6         1         7
319               DW_MLFlagnConsecutiveTimes_pZXaSYvB_t * localDW )                            -         -         -
320 (21)          {                                                                            -         -         -
321 (22)          /* InitializeConditions for Delay: 'Delay' ('Math_Library:857') */           -         -         -
322 (23)            localDW -> icLoad = 1U ;                                                   6         1         7
323 (24)          }                                                                            6         1         7
```

Summary	Prev. Runs	Current Run	Combined
Number of Executable Lines	3		
Number Executed	3	3	3
Number not Executed	0	0	0
Statement Coverage (%)	**100**	**100**	**100**

Figure 7: LDRA Statement Coverage Analysis Result

With the completion of the coverage analysis, which results shall be justified in the Software Verification Report, the SIL phase can be considered complete. For a more graphic view of the process, the next Figure illustrate the whole SIL campaign, starting from the original Simulink models to the final report.

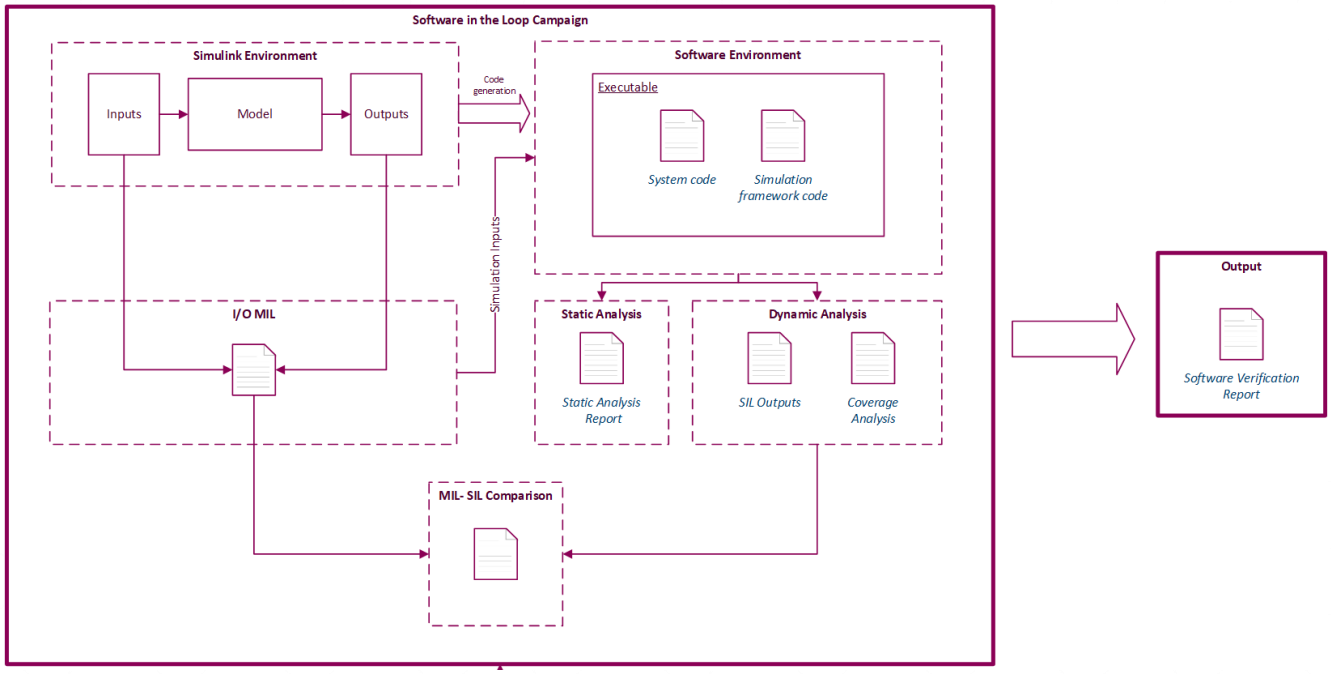

Figure 8: Software in the Loop scheme

Starting from the original Simulink models, Unitary Integration Test are run, and results are being saved in a MIL Input/Output file. The code is generated from the model and embedded in a simulation environment capable of running the code. Additional code, generated by the specific tool to be used, may be required as a framework for the simulation.

© AURORA Consortium, 2023 PUBLIC

Once the code is available, the static analysis tools are run to extract all the relevant information about the code structure as explained in previous sections. This process is followed by the dynamic simulations of the code, which runs the inputs already extracted in MIL and compare the SIL outputs with the expected values. In addition, the code coverage tools extracts which execution paths, functions, statements, etc. are executed.

6.3. Next steps

Once the static and dynamic analysis have been performed on the piece of software in this stage, results are being reported and the go-on by the customer has been consolidated, the next step of the project starts, the Processor in the Loop phase (PIL). It is in this phase where the code is deployed in a representative environment of the real flight processor, either deployed in a testbench with a real processor or deployed in a simulation framework with a representative simulator of the processor.

The main scope of this stage is, consolidate AOCS requirements by a numerical comparison of the I/O from previous phases and check the CPU budget, multithreading execution tasks, jitter, etc. Some features such as the CPU load, maximum execution times, average function execution times can be tested in SIL. Nonetheless, the results are not representative of the final product. That is why this metrics are evaluated in this phase, which will be extended in future releases of this document.

© AURORA Consortium, 2023

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