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

D4.3

Document Code: AUR-SAE-RP-0033

Document Version: 1.0

Document Date: 10/10/2022

Internal Reference: DOC00305299

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. Zurera H. Valente M. Kurowski	M. de Miguel	R. Talavera	A. López	A.I. Rodriguez
Date and Signature	Date and Signature	Date and Signature	Date and Signature	Date and Signature
Signature not needed if electronically approved by route				

© AURORA Consortium, 2022

Changes Record

Rev	Date	Author	Affected section	Changes
0.1	2022-08-31	J. Zurera	All	Initial version.
0.2	2022-08-31	J. Zurera	All	UPM content.
0.3	2022-10-07	M. Kurowski	All	TASTE related content.
1.0	2022-10-10	A.I Rodriguez	Section 3	Introduction to Component-in-the-Loop stage.

© AURORA Consortium, 2022

Index

1. Introduction 6
1.1. Purpose 6
1.2. Scope 6
1.3. Document structure 6
2. Related documentation 7
2.1. Applicable documents 7
Table 1 Applicable documents 7
2.2. Reference documents 7
Table 2 Reference documents 7
2.3. Acronyms 8
Table 3 Acronyms 9
2.4. Terms and definitions 9
3. Component-in-the-Loop stage 10
3.1. TASTE 10
3.1.1. Overview 10
3.1.2. Runtime 11
3.1.3. Automatically generated GUI 11
Figure 1 Example automatically generated TASTE GUI in action 12
3.1.4. Function Tester 12
Figure 2 Example TASTE system with a Component hosting a Simulink model 13

© AURORA Consortium, 2022

Figure 3 Choice of TASTE Runtimes for the Function Tester 13

Figure 4 Configuration of Linux Runtime helper for result retrieval 14

3.1.5. Simulator 14

Figure 5 Trivial demo system running in TASTE Simulator 15

3.1.6. Model checker 15

Figure 6 Main GUI of the TASTE Model Checker 15

3.2. MIL and SIL metrics, test, and results 16

Figure 7 UPMSat-2 ACS Simulink high level model view 16

3.2.1. Unitary Integration Tests 16

Figure 8 UPMSat-2 ACS Simulink low level model view 17

3.2.2. Performance cases in FES 17

Figure 9 UPMSat-2 simulation environment for FES 17

Figure 10 TASTE ACS Interface View 18

3.2.3. Static analysis 19

3.2.4. Dynamic analysis 19

Figure 11 SIL test and results 19

3.3. Implementation of the component model (CBI provider) over TASTE functions 20

Figure 12 cFS as a TASTE middleware. Figure adapted from cFS-B&O 20

3.3.1. Conceptual cFS model 21

3.3.2. Transformation process 21

Figure 13 cFS application autogenerating toolchain 22

3.3.3. Graphical extension 22

3.4. Integration of the UPM-Sat2 TASTE model into a cFS architecture 23

Figure 14 ACS TASTE new graphic extension 23

© AURORA Consortium, 2022

3.5. Guidelines for Component-in-the-Loop testing using TASTE 24

3.5.1. Testing using static test vectors 24

3.5.2. Testing using dynamic environment 25

3.5.3. Summary 25

4. Next steps for Platform in the Loop 27

4.1. PIL using cFS as CBI provider 27

© AURORA Consortium, 2022

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.

This activity focuses on the definition of Flight SW Autocoding Life-cycle Process, where the Autocoded system refers to any Complex-Models systems that make a full use of MATLAB/Simulink for modelling the algorithms and behaviour of the system.

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. 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. In this particular case the system developed has been the UPMSat-2 Attitude Control System (ACS). The Guidance, Navigation and Control (GNC) software of UPMSat-2.

This document is an output of the T4.3 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 and D4.2 Software in the Loop. This document updates the Software Autocoding Life Cycle with the inclusion of the Component in the Loop Stage. Later deliverables of WP4 will include the Platform in the Loop phase.

1.3. Document structure

The document has been structured as follows:

- Section 1: Introduction. AURORA life cycle.
- Section 2: Related documentation.
- Section 3: Component in the Loop.
- Section 4: Next steps for Platform in the Loop.

© AURORA Consortium, 2022

PUBLIC

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]	D5.1 CBI Requirement Specification	AUR-UPM-SP-0003	-	4
[RD2]	D5.3 AURORA Component Model Report	AUR-UPM-RP-0010	-	8
[RD3]	D6.2 Evidences for the assessment report	AUR-ESC-RP-0014	-	0.5
[RD4]	cFS Background and Overview	cFS-B&O	-	-
[RD5]	TASTE webpage	https://taste.tools/	-	-
[RD6]	D5.2 AURORA CBI Technical Architecture	AUR-N7S-RP-0001	1	2
[RD7]	D5.4 AURORA Interface Specification	AUR-N7S-RP-0002	1	3
[RD8]	RTEMS SMP QDP	https://rtems-qual.io.esa.int/	-	-

Table 2 Reference documents

© AURORA Consortium, 2022

2.3. Acronyms

Acronym	Description
HW	Hardware
MIL	Model in the Loop
N/A	Not Applicable or Available
PIL	Processor in the Loop
RD	Reference Document
SIL	Software in the Loop
SW	Software
WP	Work Package
cFS	core Flight System
UIT	Unitary Integration Test
ACS	Attitude Control System
CIL	Component in the Loop
AKA	Also Known As
OBC	On Board Computer
FES	Functional Engineering Simulator
OBSW	On Board Software
CBI	Component Based Interfaces
AADL	Architectural and Analysis Design Language
XML	Extensible Markup Language
SAVOIR	Space AVionics Open Interface aRchitecture
RTEMS	Real-Time Executive for Multiprocessor Systems
POSIX	Portable Operating System Interface
SPARC	Scalable Processor ARChitecture
VHDL	Very High-speed Hardware Description Language
SLOC	Source Lines Of Code
CSV	Comma-Separated Values
SDL	Specification and Description Language
SEDS	Space onboard interface services Electronic Data Sheet
MBSE	Model Based Software/System Engineering
ASN.1	Abstract Syntax Notation 1
BLOB	Binary Large OBject
FDIR	Failure Discovery, Isolation and Recovery

© AURORA Consortium, 2022
PUBLIC

Acronym	Description
GUI	Graphical User Interface
MSC	Message Sequence Chart

Table 3 Acronyms

2.4. Terms and definitions

N/A

© AURORA Consortium, 2022

3. Component-in-the-Loop stage

This section describes the Component in the Loop Stage of the Flight SW autocoding life-cycle process:

- Section 3.1 describes the TASTE tool-chain that is the proposed development framework for the Component Based Interface (CBI) SW generation and the Function Tester, a dedicated testing tool developed within the scope of the AURORA project to test the code generated for a TASTE QGenC Function.
- From a Simulink model, the section 3.2 describes the development of a SW system in TASTE following MIL and SIL phases, as defined in the Model-in-the-Loop (MIL) and Software-in-the-loop (SIL) stages of the Flight SW Autocoding Life-cycle Process, deliverables D4.1 and D4.2 respectively.
- Section 3.3 describes the Implementation of the component model (CBI provider) over TASTE functions. The SW as an OBSW component must follow the AURORA standard CBI API.
- Section 3.4 describes the integration of Simulink-QGenC autogenerated functions using cFS as middleware.
- Section 3.5 describes the new TASTE functionality implemented within the scope of AURORA project thar provides additional capabilities for unit-testing Simulink functions using predefined test vectors, making it easier to detect issues when transitioning from Simulink environment to the target system, e.g., using additional tools such as QGen.

3.1. TASTE

3.1.1. Overview

TASTE [RD5] is “a tool-chain targeting heterogeneous embedded systems, using a model-based development approach”. At its core, it uses AADL and ASN.1 to define software architecture and data models. The functionality embedded in the logical architecture can be defined via a variety of means, including, but not limited to:

- C/C++ code
- Ada code
- SDL models
- Simulink models.

Additionally, there are tools allowing import/migration of other languages/standards (e.g., SEDS) into TASTE. TASTE automatically translates high-level models (such as SDL or Simulink) into low-level code, usually using tools such as OpenGEODE, MATLAB or QGen. The physical architecture model supported by TASTE allows to deploy a coherent logical architecture onto multiple heterogeneous, interconnected nodes. Together with the natively supported technologies, these features make TASTE a good integration platform.

A fully defined TASTE system can be then compiled directly for one of the supported target platforms (such as x86 (for Linux), ARM SAMV71 (using FreeRTOS) or SPARC Leon3 (using RTEMS)), thus making TASTE a complete end-to-end MBSE solution.

In TASTE, a Component (as in Component Based Interfaces) is defined as a Function (in the meaning of a “functionality”, not a mathematical function, or a function in a programming language). A TASTE Interface is an endpoint for sending/receiving data of a given format and potentially triggering the given behavior. This contrasts with e.g., an interface in a language such as Java, which is an aggregate of method signatures. TASTE Interface of a Function can be Provided (implemented) or Required (used). It can be also Sporadic (asynchronous), Protected (synchronous synchronized), Unprotected (synchronous unsynchronized) or Cyclic (automatic, parameterless, asynchronous periodic). A CBI Component in the context of TASTE is therefore a Function with a

© AURORA Consortium, 2022

set of Interfaces. In particular, a Simulink model usually maps to a Function with a single Provided Protected Interface. [RD7] describes the Component Model and compilation process.

The following subchapters will briefly describe TASTE features that are most useful in the context of Component-in-the-Loop development.

3.1.2. Runtime

In order to create a full, executable system, the following items are required:

- Code responsible for user functions
- Wrappers and glue code responsible for communication between user functions
- Threading and messaging facilities
- Low-level drivers.

Code responsible for user functions is provided either directly or via models (such as SDL or Simulink), which are then translated into C or Ada via code generators (such as OpenGEODE or QGen). The rest is provided by a TASTE runtime for the given platform, part of which is delivered directly as code libraries (low-level drivers, real-time operating system) and part of which is automatically generated from the architecture description using templates. The process is described in both [RD5] and [RD7].

GR712RC RTEMS SMP QDP TASTE Runtime was developed within the scope of the AURORA project for the purpose of further PIL testing. It uses a version of RTEMS provided in ESA's Qualification Data Package for Symmetric Multiprocessing.

3.1.3. Automatically generated GUI

A TASTE Function may be set to be implemented as "GUI". In such case, TASTE automatically generates a Python GUI which exposes all interfaces to the user – both for sending and reception of data. The data received on the "provided" interfaces of a GUI Function may be plotted for user convenience (see Figure 1).

Due to the internal dependencies (including Python and Qt framework), the GUI functions may be deployed only on Linux hosts. As the data exchanges between the GUI and partition are done via queues, only sporadic (asynchronous) interfaces are allowed in the GUI Functions.

In the context of testing, the GUI has the capability to both record and replay message exchanges in the form of Message Sequence Charts. These MSCs can be easily converted into Python scripts for test automation.

© AURORA Consortium, 2022

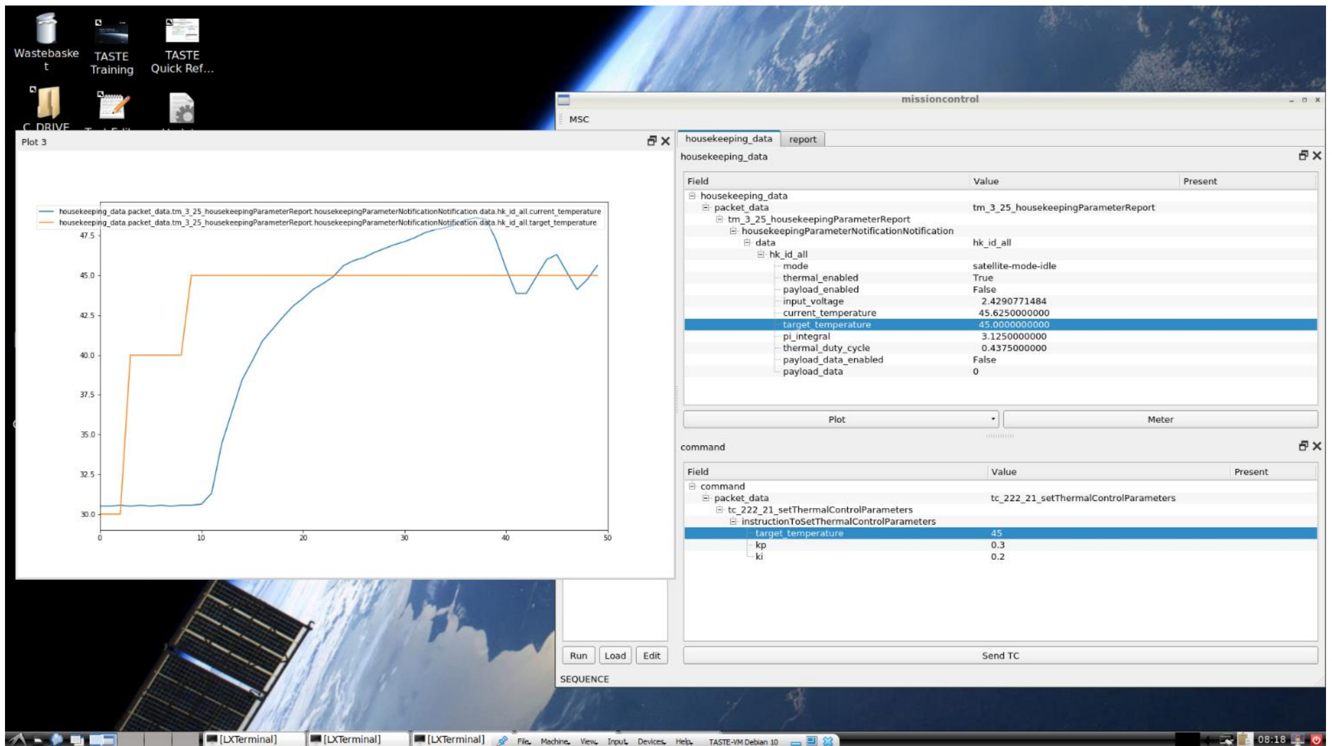

Figure 1 Example automatically generated TASTE GUI in action

3.1.4. Function Tester

Function Tester is a dedicated testing tool developed within the scope of the AURORA project to satisfy the following requirements [RD1]:

- [AUR-CBI-INT-0220] It shall be possible to test the code generated for a TASTE QGenC Function within TASTE host environment by providing a set of reference inputs and outputs.
- [AUR-CBI-INT-0230] It shall be possible to test the code generated for a TASTE QGenC Function within the target environment by providing a set of reference inputs and outputs.

As such it is limited to testing Functions with a Provided Protected Interface. However, it can be used to test both Functions implemented in Simulink (and translated to C via QGen), and Functions written in C or Ada.

As the input, Function tester needs the Interface to be tested (e.g., “Calculate” Provided Interface of “QGenHost” Function in Figure 2), a test vector (given via a CSV file) and target platform choice (selected from the list of available TASTE Runtimes, as depicted in Figure 3). Function Tester extracts the Function-under-Test into a separate test system and generates an additional test-driver Function based on the given test vector. The system is then configured for the selected target platform, compiled and executed. Execution is managed via a debugger, such as GDB, which is used to detect the end of the test scenario and download the results. As the results are dumped as a raw binary object, the configuration needed to interpret it is derived both from the system’s ASN.1 data model and the additional binary layout configuration (as illustrated in Figure 4). Data retrieval via debugger is relatively slow, but it is independent from any additional communication drivers and hardware (dongles, cables, ports, etc.).

The capability to select the target platform, for an example, the GR712RC RTEMS6 SMP QDP, allows Function Tester to test Components both on the host development hardware, and on the target embedded system. Observed behavior of code that depends on the accuracy of mathematical functions may differ between compilers, compiler settings and the underlying floating-point hardware (or software emulation thereof).

© AURORA Consortium, 2022

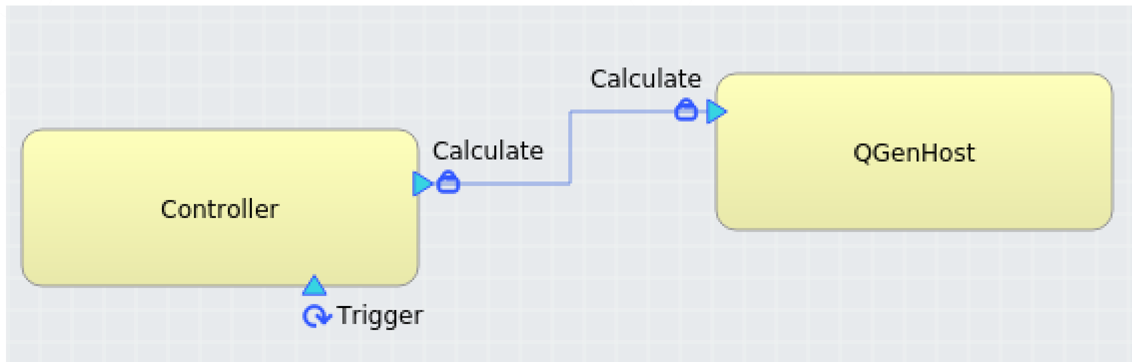

Figure 2 Example TASTE system with a Component hosting a Simulink model

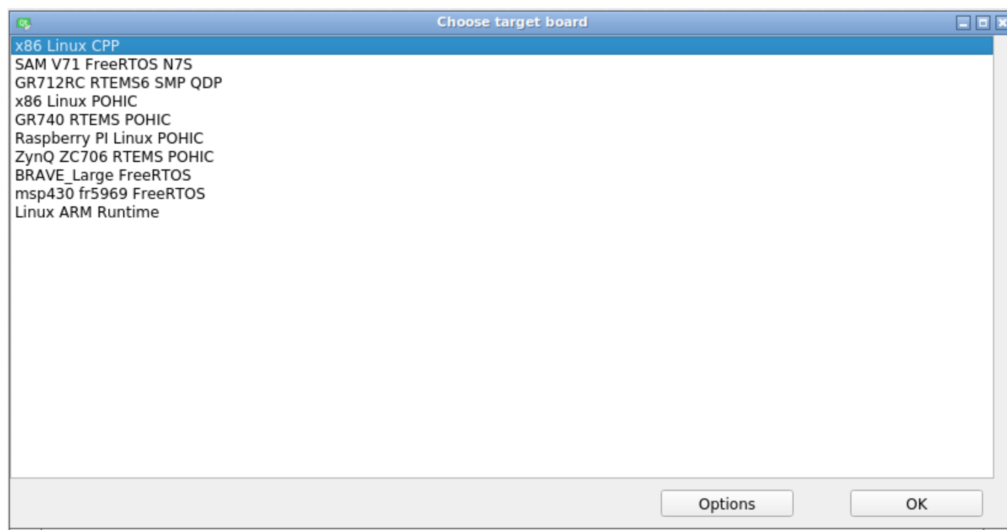

Figure 3 Choice of TASTE Runtimes for the Function Tester

© AURORA Consortium, 2022

Board options

Script path: ›cal/share/QtProject/QtCreator/getResults.gdb [Select]
Client: gdb
Client params: -batch -x $SCRIPT_PATH
Server: gdbserver
Server params: localhost:1234 $BIN_PATH
Byte order: Little Endian
Stack size (bytes): 5000

Type layout

	Size	Padding
INTEGER	8	0
BOOLEAN	1	7
REAL	8	0

OK

Figure 4 Configuration of Linux Runtime helper for result retrieval

The design of the Function Tester is included in [RD6].

3.1.5. Simulator

TASTE contains a built-in simulator, which can be used to manually test the functionality of a system step-by-step. In contrast to simply running the system on a Linux host, the simulator allows to fully control the ordering of events and exposes the internal state of the system, including intra-partition messaging and SDL process state changes (see Figure 5). Simulator can be therefore used both to explore (potentially unlikely) edge cases and debug problematic scenarios which cause issues during runtime or model checking. While the simulator cannot be used to debug Simulink models, it can be used to verify the behavior of SDL functions which may be responsible for e.g., state management in the system.

© AURORA Consortium, 2022

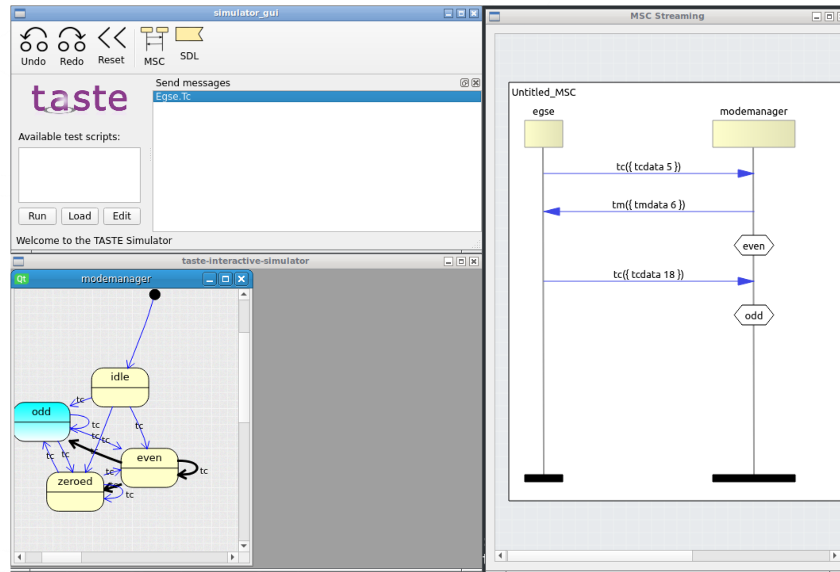

Figure 5 Trivial demo system running in TASTE Simulator

3.1.6. Model checker

TASTE incorporates model checking facilities, based on IF and Spin (under development) model checkers. They can be used to verify properties of SDL models integrated within a TASTE system – formally defined desired or undesired behaviors, system invariants and desired outcomes. Although they are not applicable to Simulink models, which are the focus of the Flight SW autocoding activity, they can be used for verification of some (SDL model based) Components, which may interact with the Simulink based Components. Such SDL Components may be used e.g., for mode management, telecommand handling or FDIR.

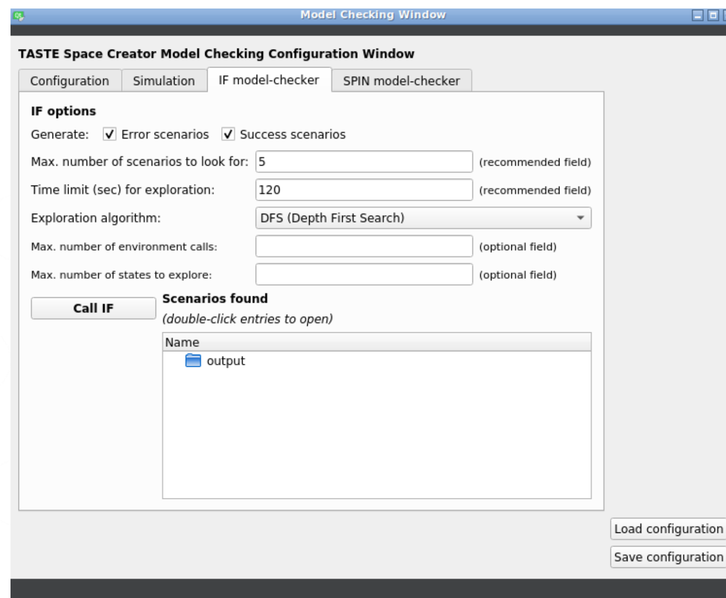

Figure 6 Main GUI of the TASTE Model Checker

© AURORA Consortium, 2022

3.2. MIL and SIL metrics, test, and results

The changes that have to be made to solve some problem in the functioning or behavior of both the modeled algorithm and the system as a whole must be made at the first level, at the model level. That is why in all the phases of the in the loop process, the previous phases must be taken into account.

The UPMSat-2 ACS has been designed and tested using the MATLAB Simulink Model Debugger tool. A set of models have been developed to describe the functioning of the system, which, in essence, determines the satellite's attitude, that is, its orientation relative to the Earth's surface. To do so, the satellite is equipped with three magnetometers, each measuring all three axes, and three magnetic torquers (AKA magnetorquers), one per axis. Also, to check the correct operation of the UPMSat-2 ACS, a Simulink model is created that simulates a real environment. It is connected using sockets to the ACS system and sends data to the sensors of it. These sensors send the data to the algorithm which makes its calculations and returns a certain value.

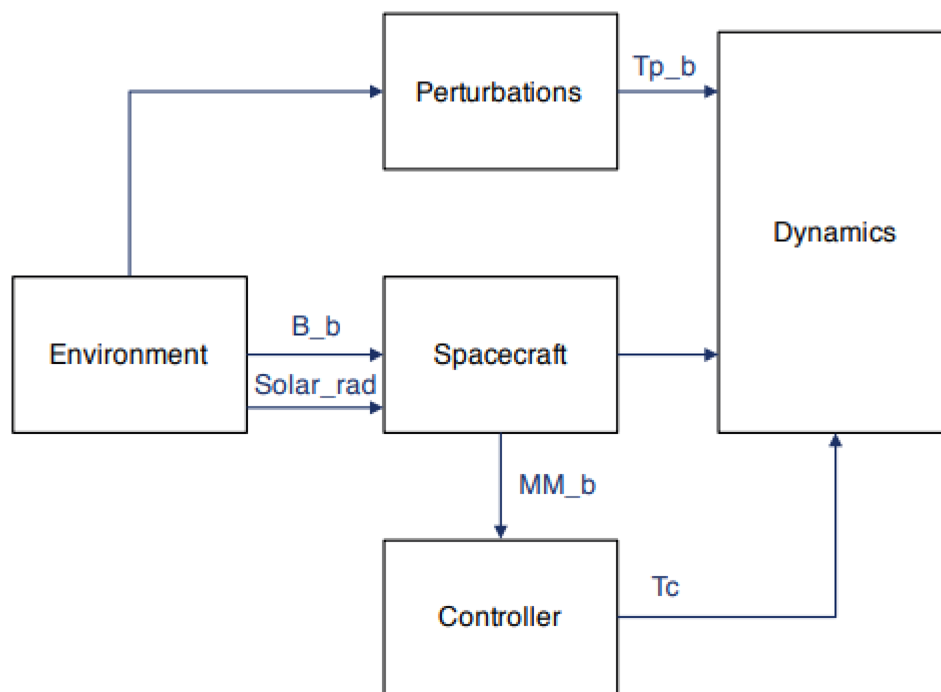

Figure 7 UPMSat-2 ACS Simulink high level model view

3.2.1. Unitary Integration Tests

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

© AURORA Consortium, 2022

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.

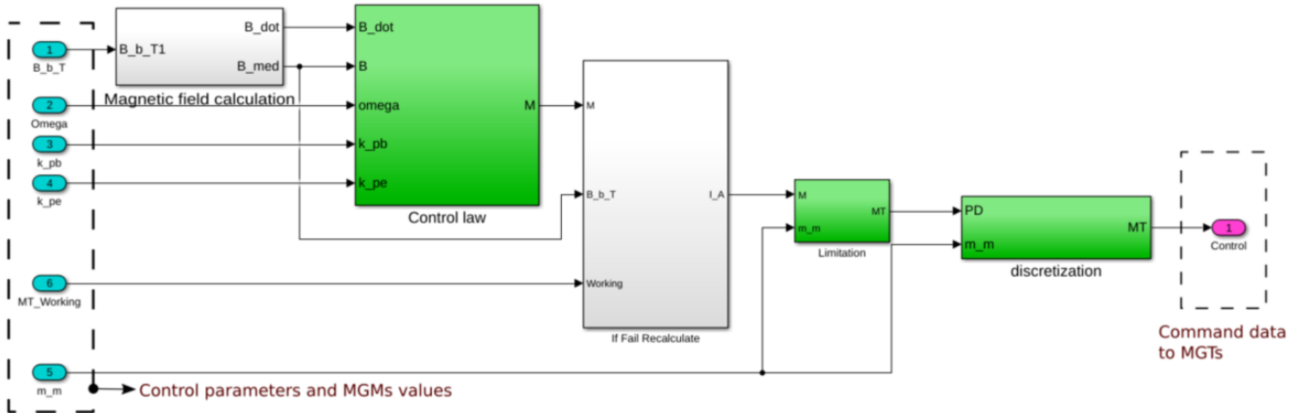

Figure 8 UPMSat-2 ACS Simulink low level model view

3.2.2. 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.

The UPMSat-2 ACS Simulink model contains blocks that simulate the satellite environment (Earth and Sun), the On-Board Computer (OBC), and relevant equipment for the ACS such as magnetometers and magnetorquers. The entire model was developed following a hierarchical structure. The following figure depicts the internals of the simulation environment for a FES performance case.

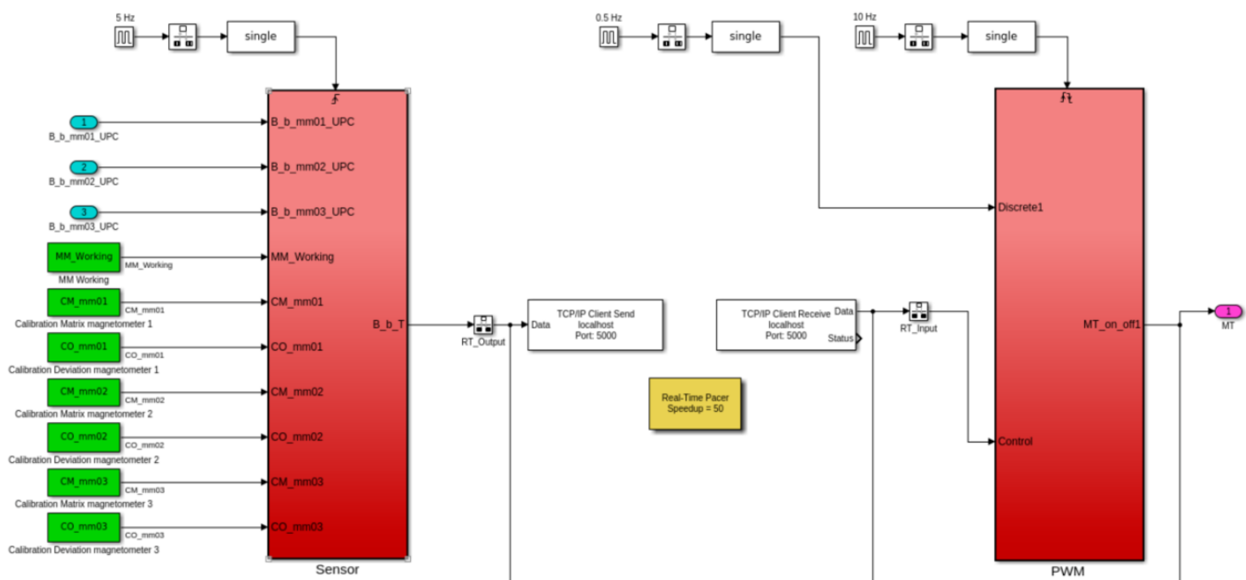

Figure 9 UPMSat-2 simulation environment for FES

© AURORA Consortium, 2022
PUBLIC

These models have been integrated into a TASTE project that will contains the control algorithm and communicate with the simulated environment. This integration is detailed in:

https://taste.tuxfamily.org/wiki/index.php?title=QGen .

The result is the auto-coding of the navigation model, using QGenC SW Generator. This will allow testing the autocoded SW with respect to the algorithms already validated in a MIL environment. The TASTE/QGenC tool suite is used to compile, link and execute the software. This behavior is represented in the following figure.

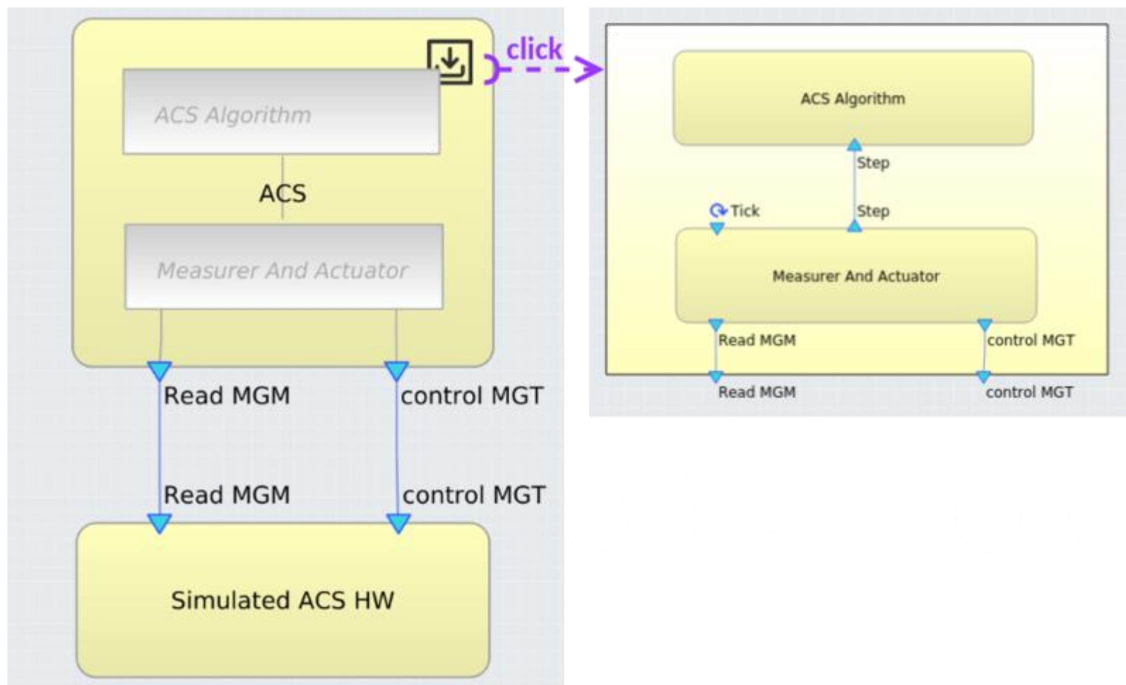

Figure 10 TASTE ACS Interface View

This TASTE project will contain the control algorithm and communicate with the simulated environment. These two functional aspects suggest the following decomposition for the Interface View:

- Simulated ACS HW: This TASTE Function is in charge of the communication with the simulated environment utilizing a TCP/IP socket. It will act as the server end of the communication and offer two Provided Interfaces (PI): Read_MGM to read the magnetometers, and Control MGT to send the magnetorquers commands.
- ACS: This is a composite TASTE function that will implement the ACS algorithm; thereby, it is composed of:
  - ACS Algorithm, this is a QGenC component that holds the control algorithm previously described. Therefore, it offers the Step unprotected PI that receives the six input signals as input parameters and returns the Control signal through an output parameter.
  - Measurer And Actuator, this is the active component and orchestrates the traditional control steps: read, control, actuate. This sequence is performed by the Tick cyclic PI.

© AURORA Consortium, 2022

3.2.3. 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.

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

- Cyclomatic complexity: 3.72 mean, and 40 maxima.
- Nesting level: 1.1
- Number of statements: 19999
- Comment frequency: 20 %
- Code size: 40476 SLOC and 262 Files.

3.2.4. Dynamic analysis

To visually inspect the correctness of the algorithm, have a look at the angular velocity scope. The setpoint established in the control loop was specified in the omega parameter, one of the inputs parameters of the ACS algorithm and modeled in the “Tick” Provided Interface of ACS Algorithm. As you can see in the following figure, the angular velocity on the Z-axis is 1 rad/sec approx. and 0 rad/sec on the X and Y-axis. This is the test that has to be carried out in the endo of the CIL phase to check if, once provided the CBI capabilities to the components, the results of this test with the same omega are compliant with the SIL results.

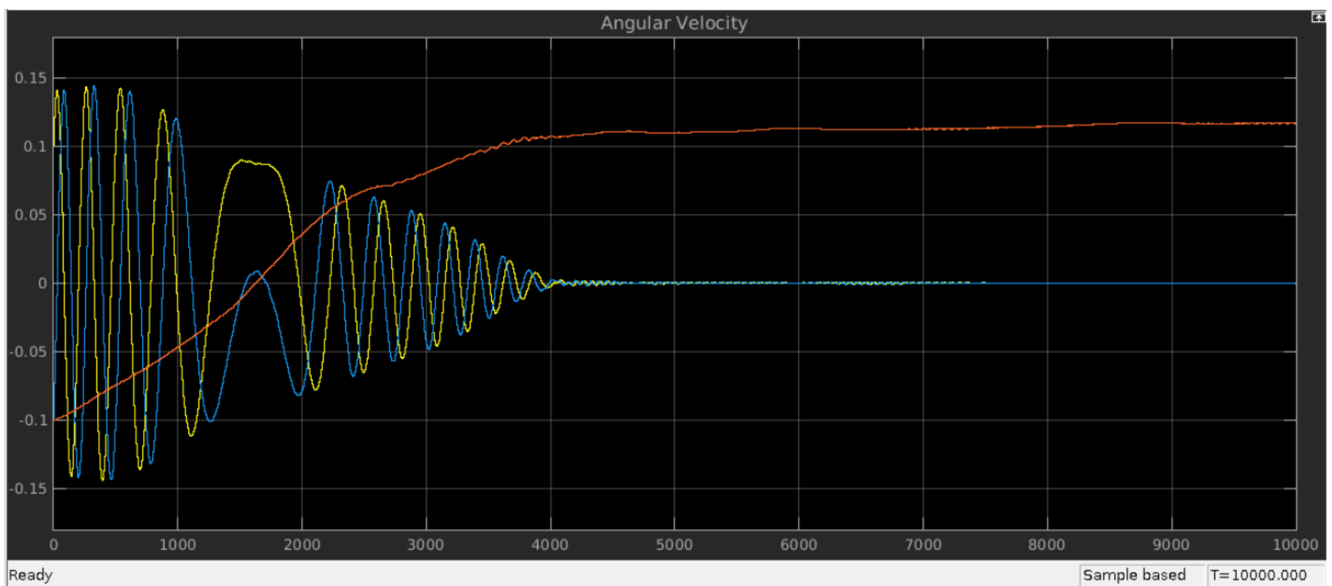

Figure 11 SIL test and results

© AURORA Consortium, 2022

3.3. Implementation of the component model (CBI provider) over TASTE functions

The SW as an OBSW component must follow the AURORA standard API, the Component-Based Interfaces (CBI), therefore the SW is integrated into a wrapper that implements the CBI for getting the services provided by the algorithms of the model-based design Guidance, Navigation and Control SW. In this case, the UPM-Sat2 ACS.

The CBI were defined in the AUR-UPM-SP-0003. These requirements can be summarized in 5 different blocks:

- N to M asynchronous communication: the component model shall provide to the components the capability to become publishers of some message following some determined protocol and to other components the capability of subscribing to those messages. Due to the asynchronism the message delivery time is not known.
- Data Store: aim at defining a Data Interface which can be implemented in those components that the model decides that needs storing capabilities. This Data interface provides a set of methods for reading, writing, updating, and deleting information.
- Fault Detection: Detecting a failure in a subsystem and informing all others to avoid communication or dependencies when an error has occurred.
- Events: The component model shall provide to some components the capability to register event listeners so they can be notified when an event occurs, and to others the capability of sending event notifications.
- Component Management: The component model interface provides a set of methods for starting, stopping, suspending, resuming, and changing power modes and management.

The objective of the CIL is to provide these capabilities to the TASTE functions no matter the implementation they have. All AADL systems, which are the direct translation from TASTE functions, must be wrapped into a piece of code that provide them with the requirements enumerated before. These capabilities are not present in TASTE as it stands at the beginning of the AURORA project. This is mainly due to the use of PolyORB as middleware environment. This middleware presents a client-server communication methodology in which many clients connect to one server. For this project N to M communications are a requirement and to achieve so a publication-subscription methodology is required. The publication and subscription communication method will be used to fulfill the communication needs of the rest of the requirements such as event communications or data store interfaces.

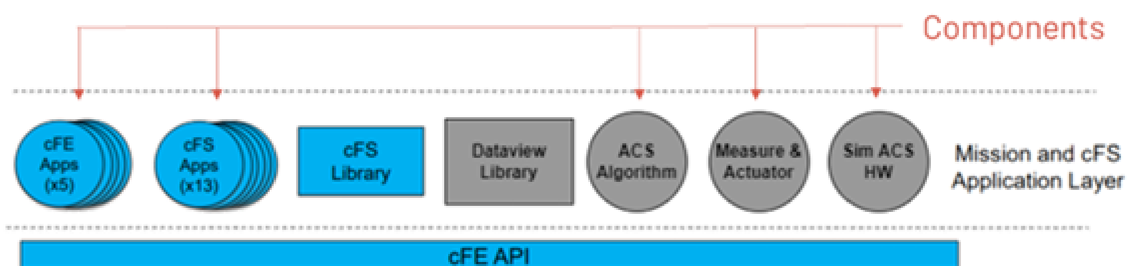

Figure 12 cFS as a TASTE middleware. Figure adapted from cFS-B&O

In order to provide each TASTE function, the CBI wrapper core Flight System has been selected as middleware for TASTE. The aim is to design a model of re-usable, scalable, and configurable components, which will facilitate concept, interface specification and behavior definition exchanges across projects, project phases and domains. Standardizing every functional block such as cFS applications, manual coded C functions or autogenerated QGenC functions into AADL systems. These AADL system as components using cFS as middleware API provider. Each of the TASTE functions would be wrapped into a cFS application that would represent the component model.

© AURORA Consortium, 2022

From a TASTE Interface View architecture with embedded QGenC autogenerated functions and other TASTE functions completing the architecture functionality such as sensors, management or store functions, a compilation is performed, which would be detailed in an example in the next section, in which these functions are converted into cFS applications and integrated in a cFS built environment. This built environment is compiled and linked into a set of executables depending on the target in which the executions is performed. To check the CBI characteristics, and following the recommendations of the Grand Agreement, the Component in the Loop phase shall be carried out in the same environment than the Software in the Loop was performed, that means the same environment than the TASTE/QGenC tool suite.

The process of generation of the Component Model and its integration on a cFS architecture is described in the AUR-UPM-RP-0010 summarized in three transformation processes which are presented below.

3.3.1. Conceptual cFS model

A conceptual model that identified the properties that were necessary to be supported by Space Creator, the TASTE graphical user interface. This conceptual model supports the basic modelling concepts such as:

- Function: The function is the main modelling element in Space Creator and represents a component. In AADL the function properties are attached to the system element.
- Data Store: A datastore provides the ability to manage data that survives a processor reset. In AADL the datastore properties are attached to the system element.
- Interface: Interfaces are used to allow interaction between components. In AADL the interface properties are attached to the subprogram element.
- Event: Events are notices sent as a response to some command or error. In AADL the event properties are attached to the subprogram element.
- Message: Messages are the way in which data is sent between components. In AADL the message properties are attached to the subprogram element.

3.3.2. Transformation process

TASTE relies on AADLConverter and Kazoo for the code generation. These are the tools that need to be modified in order to support the new conceptual model and generate cFS compliant source code. By adding new templates and extending the parsing capabilities of Kazoo, we can generate code skeletons for a new platform, cFS.

- XML to AADL: mapping between the XML elements and the AADL elements. To extend the elements supported by AADL, a new property set named CFS was created.

© AURORA Consortium, 2022

- AADL to C: template files that contain two different types of text: the static text that will be rendered and the handles which reference dynamic text that will be changed based on the properties specified in the model. The Kazoo templating engine takes as input the code generation templates and replaces the handles with the values present in an associative table.

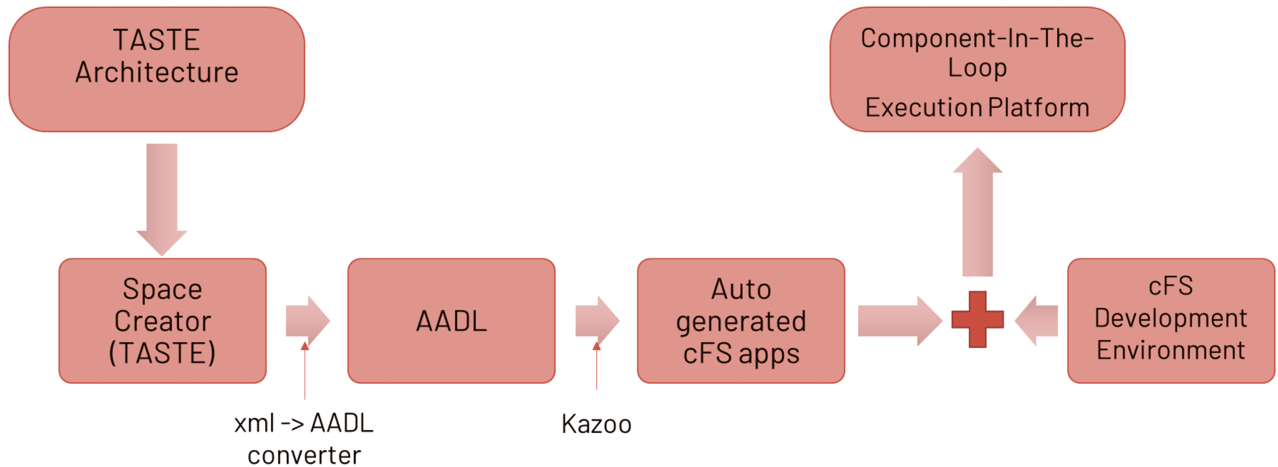

Figure 13 cFS application autogenerating toolchain

3.3.3. Graphical extension

Changes have been made to the graphical user interface to represent the new cFS elements. In this way it is easier for the user to study and understand the system. The icons used to represent messages and events was based on the SAVOIR notation. The changes allow to represent the data flow between the publisher and subscriber, send events and represent a Data Store component.

© AURORA Consortium, 2022

3.4. Integration of the UPM-Sat2 TASTE model into a cFS architecture

In this section is integrated an instance of the already presented ACS TASTE architecture with a complex Simulink-QGenC autogenerated function ACS Algorithm and two other AADL systems Measurer And Actuator and Simulated ACS HW. The generated code consists of application specific files that are generated for each component based on the properties of the models and of global configuration files that specify the startup information required to load the component on startup. This code acquires the CBI characteristics using cFS as middleware. In the next figure the architecture is shown using the new graphical extension detailed in the previous section.

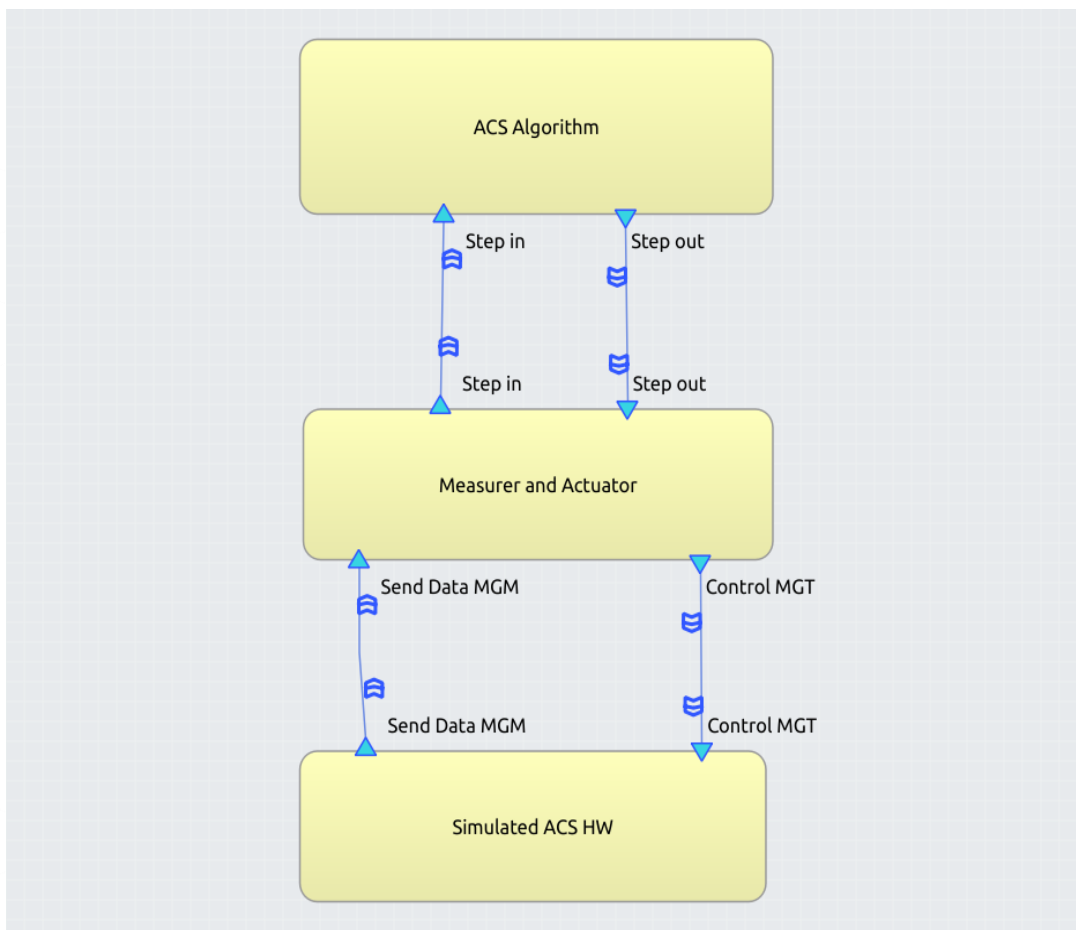

Figure 14 ACS TASTE new graphic extension

This architecture is divided into three components, one for each TASTE function. The ACS subsystem has two TASTE functions, ACS Algorithm and Measurer And Actuator which are integrated into cFS applications. Simulated ACS HW simulates an environment and shall not be executed on the same board. This is because this simulated environment consumes an amount of processing capacity that, in the case of using the same board, would handicap its operation, obtaining invalid results. For the case that concerns this deliverable, CIL, the tests

© AURORA Consortium, 2022

must be executed on the same system as those carried out in the SIL section, so software will be executed on the same PC-Linux on which TASTE is being executed.

- ACS Algorithm. This is the component that has embedded the main ACS algorithm. This is generated autonomously from your model in Simulink using QGenC.
  This tool generates a certain number of files with functions that perform the attitude calculation operation. For their integration and use of the cFS features it is only needed to combine them with a standard cFS application that: first subscribes to Measurer And Actuator, receives the correct data array (Step in) from it and inserts it using the functions generated by QGenC in the algorithm, collect the computed value and publish it back to the component it received it from (Step out). In order to be able to insert the received value to the algorithm, all the files autogenerated by QGENC have to be inserted as source code of the cFS application. The entry point is the only method that the application shall call inserting a value and receiving the result.
- Measurer And Actuator. This component acts as a message broker between the simulated environment and the ACS algorithm. The component setups four different communication paths: subscription to Simulated ACS HW (Send Data MGM), publication to ACS Algorithm sending environmental raw data (Step in), subscription from ACS Algorithm (Step out) and publication to Simulated ACS HW with the attitude value computed and its effect on the environment (Control MGT).

- Simulated ACS HW: This component generates a set of values that the sensors would receive in a real deployment (Send Data MGM) and sends them by publication to the Measurer And Actuator. It receives the actuation of the system (Control MGT) depending on the value computed in ACS Algorithm through subscription to Measurer And Actuator.

Initially, cFS core services and core Flight Executive will be started, which provide the CBI features for which cFS has been selected as CBI provider. The startup order of the different cFS applications will follow the priority determined in their configuration file. These applications will carry out those functionalities for which they have been developed in TASTE. In a system log you can see the different actions of the same and the use of the different interfaces provided by cFS such as the use of publication subscription methods for communications, capacity to manage memory with reading and writing capabilities, publication of different types of events and so on.

3.5. Guidelines for Component-in-the-Loop testing using TASTE

Based on the capabilities of TASTE, both pre-existing and developed within the scope of the AURORA project, and sourcing the needs from the experience reported from the UPM-Sat2 use case, there are two general approaches to testing Components (Functions) using TASTE:

- testing using static test vectors
- testing using dynamic environment

The approaches are complementary and do not exclude each other.

3.5.1. Testing using static test vectors

Testing using static test vectors can be done in several ways. The primary recommended method is to prepare a set of test vectors containing both the input and output values. Such data should be available from earlier phases of the development that include Simulink modelling (e.g., Model-in-the-Loop). The test vectors should be provided in the form of a CSV file, which can be then used by the Function Tester to exercise the selected Interface of a Function implemented via the Simulink model. Such approach can verify whether the code generated during TASTE compilation process (which may include custom code generators, such as QGen) produces the same output as the model in its native environment (e.g., Simulink), thus detecting possible translation issues. If the

© AURORA Consortium, 2022

tests are run on the target platform instead of the development host, additional issues related to differences in compilers, mathematical libraries and floating-point hardware may be also revealed.

As the test vectors are provided as a constant input and reference, they are useful for repeatable, semi-automated testing and can be used to detect any possible regressions. The main drawback of the approach is that the test vectors need to be available in the first place, which may not be the case if the development process does not include a proper MIL phase. Another issue is that due to the perfect repeatability of the test vectors, they cover only the given, potentially small, subset of possible inputs and outputs, which may be insufficient to fully exercise the system. Careful selection of representative test vectors is necessary to ensure high testing quality. Code (line and decision) coverage can be used to measure the quality of the choice of such test vectors.

Another approach to testing using static test vectors is to manually create a test system which includes the Component-under-Test and a connected GUI Function. If the Component-under-Test exposes only protected interfaces, as is the case for Simulink based Functions, another Proxy component is needed to bridge the asynchronous queries and responses from/to the GUI and the synchronous queries of the tested Function. Such a system can be compiled for the host platform and exercised manually. The results of manual testing can be then inspected and saved in the form of an MSC scenario, which can be then replayed as needed to detect any possible regressions. This approach relies on the manual inspection of the first set of results, making it time consuming and error prone. However, further testing passes can be automated.

3.5.2. Testing using dynamic environment

While static testing provides a reliable way to test for regressions and translation/transpilation issues, the preparation of test vectors may be time consuming or infeasible. This however can be solved by testing using a dynamic environment, in which the Component-under-Test is fed data from e.g., a concurrently running simulation. Depending on the construction of the system – for example depending on whether the used time step is fixed or measured/simulated (and thus allowing some jitter) – such testing may reveal issues related to timings, which, in rare cases of incorrectly written algorithms, may lead to instabilities. While the exact desired numeric outputs may be unknown, success conditions may be formally defined in terms of e.g., output convergence to a set value or an allowed amplitude of oscillations.

In order to perform testing of a Component using a dynamic environment, it may need to be placed in a manually created test harness, which, in addition to the Component-under-Test, may include:

- Verification Function, which decides on a test pass or fail status
- Environment Function, which generates inputs and reacts to the outputs of the Component-under-Test

In particular, instead of a single Environment Function, the necessary environment simulation can be distributed into several Functions, responsible e.g., for physic simulations, as well as sensor and actuator mocks. Such Functions may be distributed over multiple nodes (both Linux-based and embedded) and, if necessary, can be implemented via complementary Simulink models. The Verification Function may be implemented as a GUI for visual inspection, but for test reliability and automation, especially in the context of a Continuous Integration environment, algorithmically defined constraints are recommended. If the data needs to be reviewed, either for a regular inspection or for issue investigation, the Verification Function may include logging facilities.

3.5.3. Summary

The TASTE capability to generate executable code, autogenerate GUI and include multiple Simulink models, potentially distributed over multiple nodes, allows to construct testing harnesses for Components hosting Simulink models. The tests can be both automated (in case of formally defined test/pass criteria) and manual (if output inspection is needed).

New TASTE functionality implemented within the scope of AURORA project provides additional capabilities for unit-testing Simulink functions using predefined test vectors, making it easier to detect issues when transitioning from Simulink environment to the target system, e.g., using additional tools such as QGen.

Both testing using static test vectors and dynamic system simulation is recommended for ensuring code quality.

© AURORA Consortium, 2022

The currently implemented simulation and model-checking capabilities of TASTE, while useful for SDL models, are not directly applicable to Simulink models. Possibility to integrate Simulink models into TASTE simulator and model checkers should be investigated.

© AURORA Consortium, 2022
PUBLIC

4. Next steps for Platform in the Loop

4.1. PIL using cFS as CBI provider

The objective of this section is to present the further steps to develop a platform to represent the real behavior of the autogenerated SW. Validate the SW component running in the execution platform connected to an open-loop-environment.

To analyze the behavior of the subsystem ACS UPMStat-2 using cFS as CBI provider a new execution platform has to be developed. At the time of the development of this document cFS is ready to be implemented in a variety of different platforms. From the GitHub platform, referred in the cFS-B&O, these platforms are pc-Linux, pc-RTEMS and mcp750-VxWorks. The objective here is to create a new LEON3-RTEMS platform adding new OS abstraction and support packages.

RTEMS is an Open-Source Real-Time Operating System that is available for a wide range of processor families and boards. SPARC V8 processors are supported, and it is available for several computer systems based on ERC32, LEON2, LEON3, and LEON4 processors. It is widely used in the aerospace domain and has a POSIX interface that makes RTEMS a good choice for building a platform to execute cFS applications.

A specific On-Board Computer (OBC) has been designed as a hardware platform. This hardware platform includes a LEON3 processor that has SPARC v8 architecture. This On-board Computer was developed from the VHDL GRLIB IP cores library, and it is the flight version of the UPMSat-2 microsatellite that was launched on September 3, 2020, on the flight VV16 of the Vega rocket.

Due to the layered structure of cFS, once the platform is developed, the applications that were generated from the AADL modeling language using the auto code generation tool Kazoo in the CIL stage can be directly executed and tested in the new platform without any changes.

© AURORA Consortium, 2022

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