INtegrated TOol chain for model-based design of CPSs

The INTO-CPS Examples Compendium

Version: 1.3

Date: October 2018

The INTO-CPS Association

http://into-cps.org

Contributors:

Richard Payne, UNEW
Carl Gamble, UNEW
Ken Pierce, UNEW
John Fitzgerald, UNEW
Martin Mansfield, UNEW
Simon Foster, UY
Kangfeng Ye, UY
Casper Thule, AU
Rene Nilsson, AU
Kenneth Lausdahl, AU
Florian Lapschies, VSI
Frederik Foldager, AI

Editors:

John Fitzgerald, UNEW

Document History

Ver	Date	Author	Description
0.1	15-05-2017	Richard Payne	Draft structure of deliverable and initial reframing based on D3.5
0.2	27-10-2017	Martin Mansfield	Added Autonomous Vehicle and Mass Spring Damper pilots
0.3	01-11-2017	Martin Mansfield	Version for internal review
1.0	14-12-2017	Martin Mansfield	Review comments addressed
1.1	26-02-2018	Peter Gorm Larsen	initial version for the INTO-CPS Association
1.2	07-10-2018	John Fitzgerald	Alignment with other Association documents
1.3	02-11-2018	Casper Thule	Updated list of examples and example links

Abstract

This document is intended for users of the INTO-CPS technologies and contains a collection of example and pilot study model descriptions demonstrating the INTO-CPS technology. Each study has a description of the example, the models available for the study and examples of the analyses available with the INTO-CPS Tool Chain. The examples demonstrate the INTO-CPS tools developed during (and after) the INTO-CPS project. This compendium is a living document that has grown with the tool chain. This document is a snapshot of the compendium presently, building on previous versions (in M12 [FGP+15], M24 [PGP+16] and M36 [MGP+17]). The latest version can always be found at `https://github.com/INTO-CPS-Association/training/releases`.

Contents

1 Introduction

This document provides an overview of different public example multi-models that stakeholders who are interested in experimenting with the INTO-CPS technology can use as a starting point. The examples have been developed using the different simulation technologies in INTO-CPS: 20-sim[1]; Overture/VDM-RT[2]; OpenModelica[3]; SysML[4]; and RT-Tester[5]). The examples are comprised of multi-models using the INTO-CPS SysML profile and collections of Continuous Time (CT) and Discrete Event (DE) models elicited from the simulation models. Examples of their use is also given, demonstrating features and analyses made available by the INTO-CPS tool chain.

This deliverable is structured in different sections, each of which provides a brief introduction to each example model. Each example illustrates different aspects of the INTO-CPS tool chain, as summarised here:

- Section 2 presents a Single-tank Water Tank model. The simplest example in the compendium, this is a two-model multi-model, using 20-sim and VDM-RT FMUs. The example has a SysML architecture, can be co-simulated, and has support for Design Space Exploration (DSE).
- Section 3 presents a Three-tank Water Tank model. This study aims to demonstrate the division of CT elements across different FMUs. The study comprises 20-sim and VDM-RT FMUs and demonstrates DSE and Test Automation technologies.
- Section 4 presents a Fan Coil Unit (FCU). Originally presented as a baseline OpenModelica model. This model demonstrates the options for multi-modelling and dividing models into separate FMUs to allow for architecting to be carried out in the SysML architectural model. The example demonstrates the use of co-simulation and DSE.
- Section 5 presents a Line-following Robot. This study has four possible co-simulation multi-models – two using replication offered by 20-sim FMUs (one using 3D visualisation and one without) and another two configurations which are not using FMU replication. The study provides several DSE experiments.
- Section 6 presents a Turn Indicator example. In this study, the behaviour of a car's turn indicator is modelled using parallel state-charts. This model is then used to automatically derive tests and to perform model checking.
- Section 7 presents a single-UAV model, which models the physical dynamics as well as the discrete controller of an Unmanned Aerial Vehicle (UAV). The model contributes a high-fidelity physical model, enabling the multi-model to be used to compare alternative control algorithms.
- Section 8 presents an Ether communication model. This pilot provides an initial demonstration of a model for network communications. This pilot is VDM-RT only,

[1] `http://www.20sim.com`
[2] `http://overturetool.org`
[3] `https://openmodelica.org`
[4] Using the Modelio tool: `https://www.modeliosoft.com`
[5] `https://www.verified.de/products/rt-tester/`

with a simple SysML architecture. Co-simulation takes the form of the demonstration of messages passing through the ether. The intention is that this pilot will be used in the future by others using network communications.

- Section 9 presents a swarm of communicating simplified UAVs. This pilot is a first version of a swarm of UAVs which receive direction from a central controller. The pilot uses FMUs from 20-sim and VDM-RT taking advantage of FMU replication offered by both notations. Co-Simulation and 3D visualisation are supported.
- Section 10 presents an autonomous vehicle model. In this study, a multi-model demonstrates a vehicle traversing a path of defined waypoints. The multi-model comprises two FMUs; The dynamics of the vehicle are modelled by a 20-sim FMU, and the vehicle control is determined by a VDM-RT FMU. The example has a SysML architecture, can be co-simulated, and has support for DSE.
- Section 11 presents a mass spring damper model. The pilot uses models of masses and springs to demonstrate the utility of so-simulation stabilisation using both 20-sim and OpenModelica FMUs.

In order to guide you in what models to consider inspecting, we have created Table 1 illustrating the different characteristics of the different publicly available models.

	INTO-CPS Technology										
Model	Multi-DE model	Multi-CT model	20-Sim (for FMU)	OpenModelica (for FMU)	VDM-RT (for FMU)	INTO-CPS SysML	Co-simulation engine(COE)	DSE support included	Test Automation support	Model checking	Code Generation
Single-tank Water Tank			x	x	x	x	x	x			x
Three-tank Water Tank		x	x		x	x	x	x	x		x
Fan Coil Unit (FCU)		x	x	x	x	x	x	x	x	x	x
Line-following Robot		x	x	x	x	x	x	x	x	x	x
Turn Indicator									x	x	
Single UAV		x	x		x	x	x				
Ether	x				x		x		x	x	
UAV Swarm	x	x	x		x	x	x		x	x	
Autonomous Vehicle			x		x	x	x	x			
Mass Spring Damper			x	x			x				

Table 1: Overview of INTO-CPS technologies used for pilot studies. An unmaintained model is marked with .

2 Single-tank Water Tank

2.1 Example Description

The single-tank water tank pilot study is a simple example that comprises a single water tank which is controlled by a cyber controller. When the water level of the tank reaches a particular level (defined in the controller) the controller sends a command to the tank to empty using an exit valve. A diagram of the example is shown in Figure 1. This pilot is also related to the next pilot in Section 3.

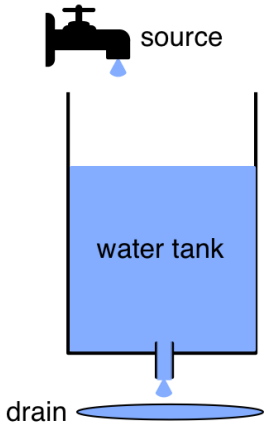


Figure 1: Overview of the single-tank water tank example

2.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-single_watertank` in the *master* branch. There are several subfolders for the various elements: `FMU` contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definition; and `SysML` – contains the SysML model defined for the study.

The `case-study_single_watertank` folder can be opened in the INTO-CPS application to run the various co-simulations as detailed in this document. To run a simulation, expand one of the multi-models and click 'Simulate' for an experiment.

2.3 INTO-CPS SysML profile

The single tank SysML model produced using the INTO-CPS profile comprises two diagrams; an Architecture Structure Diagram (ASD) and a Connections Diagram (CD).

The ASD in Figure 2 shows the system composition in terms of component subsystems from the perspective of multi-modelling.

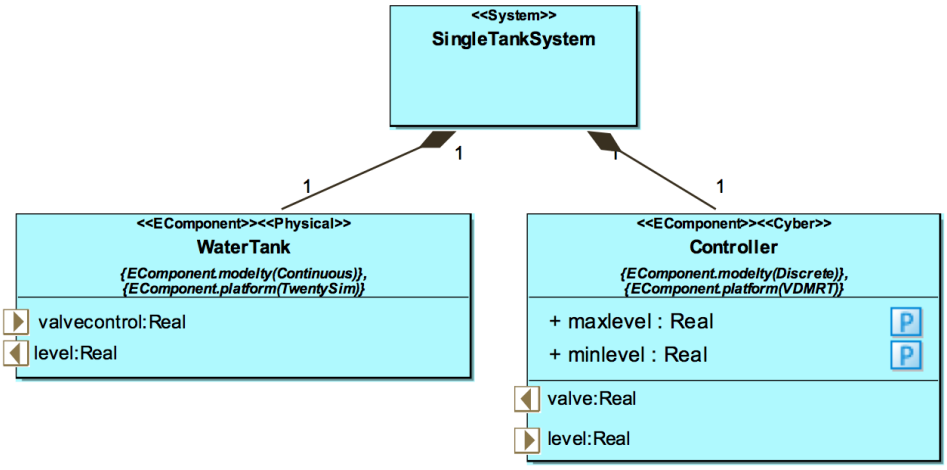


Figure 2: Architecture Structure Diagram defining the Single-tank Water Tank system composition

This *SingleTankSystem* model, comprises a single *WaterTank* physical component and a cyber component *Controller*. Ports are exposed by the *WaterTank* component for outputting the current water level (`level`) and for receiving valve control commands (`valvecontrol`). The *Controller* component has reciprocal ports and also variables to define the permitted minimum and maximum water levels (`minlevel` and `maxlevel` respectively).

The *WaterTank* component is defined as a continuous time model with 20-sim as the target platform, this may be also be defined as OpenModelica. The *Controller* component is a VDM-RT discrete event model.

A single System Block Instance is defined in the model to represent the system configuration. The CD in Figure 3 shows that the *WaterTank* component has two connections with the *Controller* cyber component - regarding the level and valve control.

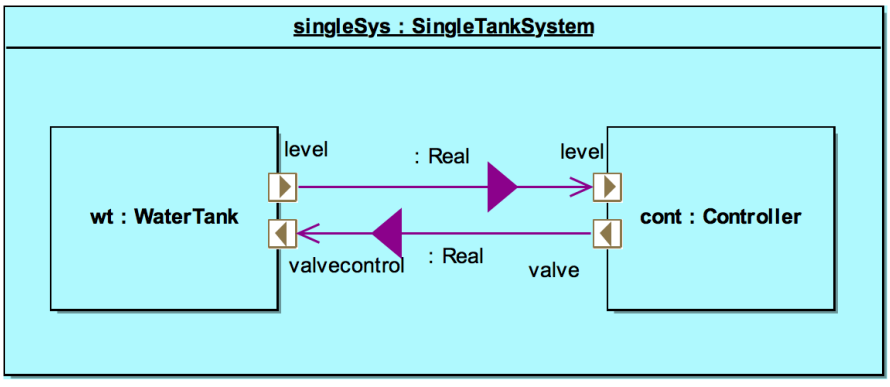


Figure 3: Connections Diagram defining the Single-tank Water Tank system connections

2.4 Multi-model

2.4.1 Models

The SysML model above dictates there are two models: a 20-sim model for the water tank and a VDM-RT model for the controller. This section gives an overview of those models.

Watertank.emx The 20-sim model of the *Water Tank* component, shown in Figure 4, comprises several sub-components. A flow source is connected to a tank, which fills up at a constant rate. The tank reports the current water level on the *level* port. A valve, controlled by the *valvecontrol* port empties water from the tank into a drain.

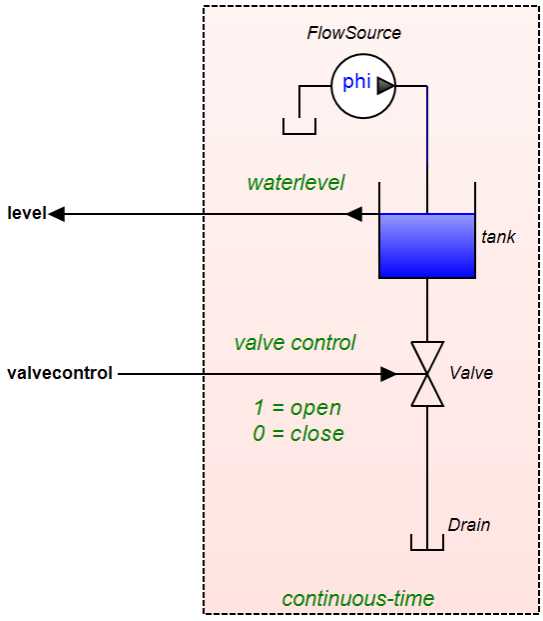


Figure 4: 20-sim Water Tank component

WaterTank.mo *WaterTank.mo* contains a *SingleWaterTank* model, which has the same external interface as the above *SingleWatertank.emx* model – as both comply to the FMI *modeldescription.xml* exported format. The model is defined mainly through equations, and so is not shown in this document.

SingleWT The VDM-RT *SingleWT* controller is a simple model, with an architecture shown in Figure 5. The *System* class owns a *HardwareInterface* instance with RealPorts to receive the sensed water level and send valve control commands. The values are passed to *LevelSensor* and *ValveActuator* objects used by the *Controller* class. The control algorithm compares the level to the *minlevel* and *maxlevel* design parameters and sets the valve control appropriately.

2.4.2 Configuration

Two multi-models are defined:

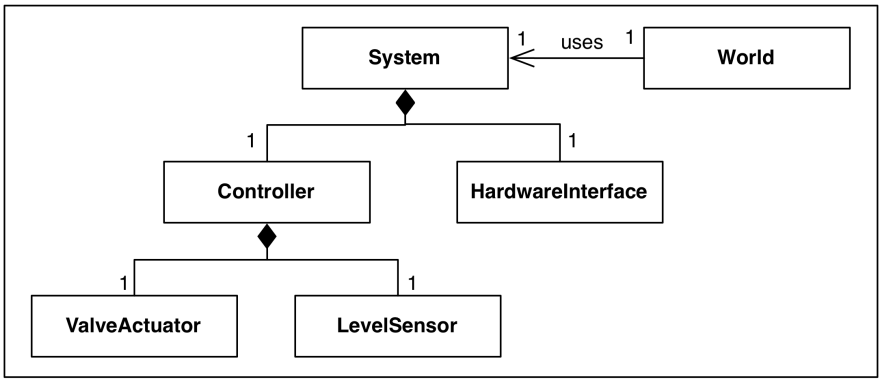


Figure 5: VDM-RT model architecture

mm The multi-model `mm` corresponds to the CD in Section 2.3. Two connections are defined:

- from the *WaterTank* `level` port to the *Controller* `level` port; and
- from the *Controller* `valve` port to the *WaterTank* `valvecontrol` port.

The FMUs used are *singlewatertank-20sim.fmu* and *SingleWT.fmu*

There are two design parameters in the multi-model – `minlevel` and `maxlevel`, which are defined to be 1.0 and 2.0 respectively.

mm-OM An alternative multi-model is defined using the OpenModelica FMU. The connections are identical to the multi-model above, although rather than using the 20-sim FMU, *WaterTank_SingleWaterTank.fmu* is used.

2.5 Co-simulation

A co-simulation experiment is defined for the multi-model – with a runtime of 30 seconds and using the fixed step size of 0.1 seconds. Simulating using this experiment produces the livestream output shown in Figure 6.

The graph shows the water level (orange line) and valve control (blue line) values. The water level rises steadily until it reaches 2.0 (the maximum level), at this point the valve control is set to 1.0 and the water level drops to 1.0 (the minimum level). At the minimum level, the valve is closed and the water rises once again. This behaviour repeats through to the end of the simulation.

2.6 Analyses and Experiments

2.6.1 Design Space Exploration

This pilot supports DSE. We reuse the DSE experiment used in the Three-tank Water Tank Pilot study – and is briefly described here. For discussion on results obtained, see the Three Tank study in Section 3.6.1.

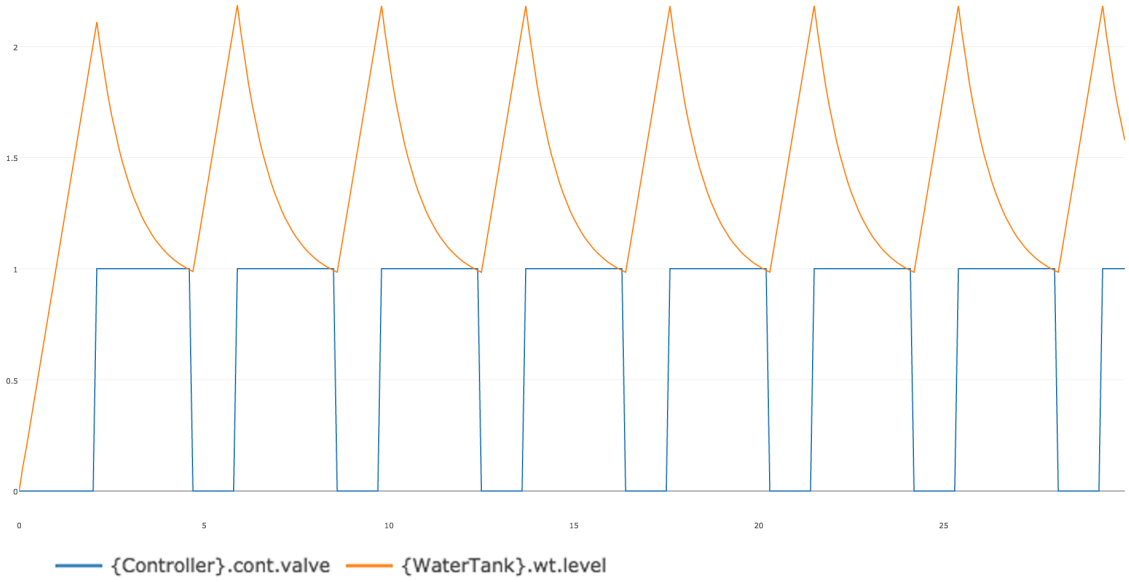


Figure 6: Co-simulation results for Single-tank Water Tank system

The experiment varies the two design parameters of the study – `minlevel` and `maxlevel`. These parameter values may be set between 0.2 and 2.0 in intervals of 0.2. A constraint on the parameters (`Controller.cont.maxlevel` > `Controller.cont.minlevel`) ensures that the maximum water value is always larger than the minimum water level.

Two objectives are defined: *cumulativeDeviation* and *vCount*. The first objective, *cumulativeDeviation*, is to minimise the cumulative deviation from a desired level - set to 1.0. The second objective, *vCount*, is to minimise the number of valve operations – i.e. have a lower number of valve state changes. The analysis uses the Pareto method for ranking.

2.6.2 Code Generation

The VDM-RT model, **SingleWT** can be exported from Overture as a C code FMU, in addition to the tool wrapper FMU as used above. The *watertankController-SourceCode.FMU* included in this pilot is obtained directly from Overture using the "Export Source Code FMU" option. However, this FMU does not contain binaries for co-simulation and so one may use the *FMU Builder* included in the INTO-CPS Application to compile FMUs for Windows, Mac and Linux.

This process has been performed and the resultant FMU is included in the pilot in the FMUs folder; *watertankController-Standalone.FMU*. One example experiment available is to switch this FMU for the tool wrapper version – *SingleWT.FMU* – and compare results.

3 Three-tank Water Tank

3.1 Example Description

The three-tank water tank model is based upon a standard 20-sim example, and is developed to explore the impact on accuracy of multi-modelling across multiple CT models. The example comprises three water tanks which are filled and emptied. The first tank is filled from a source with a valve which may be turned on and off. The outflow of the first tank constitutes the inflow of the second, and so forth. A controller monitors the level of the third tank and controls a valve to a drain.

A key feature of this example is the close coupling required between water tank 1 and 2, and the loose coupling to water tank 3. Water tanks 1 and 2 are tall and thin and are connected by a pipe at the bottom of the tanks (a diagram of the example is shown in Figure 7), and therefore changes to the level of water tank 1 (due to water entering from the source) will quickly affect the level in water tank 2. This effect is not as prevalent between water tank 2 and 3.

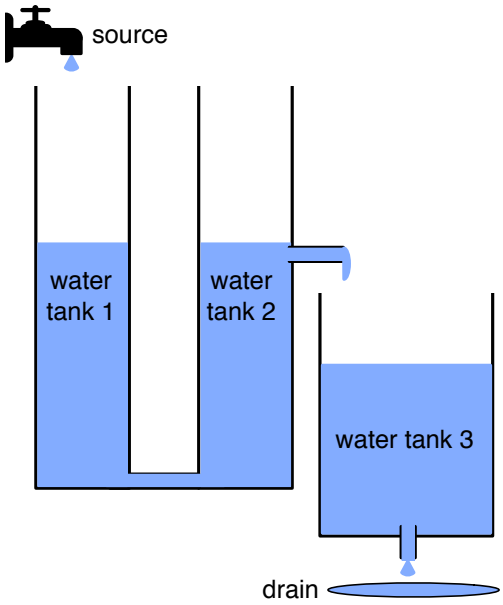


Figure 7: Overview of the three-tank water tank example

This pilot expands that in Section 2.

3.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-three_tank_watertank` in the *master* branch. There are several subfolders for the various elements: `DSEs` - contains work in progress DSE scripts; `FMU` – contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation

technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations; `SysML` – contains the SysML models defined for the study; resources – various images for the purposes of this readme file.

The `case-study_three_tank` folder can be opened in the INTO-CPS application to run the various co-simulations as detailed in this document. To run a simulation, expand one of the multi-models and click 'Simulate' for an experiment.

3.3 INTO-CPS SysML profile

A SysML model produced using the INTO-CPS profile comprises three diagrams and focusses on the structure of the water tank model for multi-modelling; an Architecture Structure Diagram and two Connections Diagrams.

The Architecture Structure Diagram (ASD) in Figure 8 shows the system composition in terms of component subsystems from the perspective of multi-modelling. As discussed in [FGP+15], this architecture differs from a holistic architecture due to the grouping of tanks into the different subsystems.

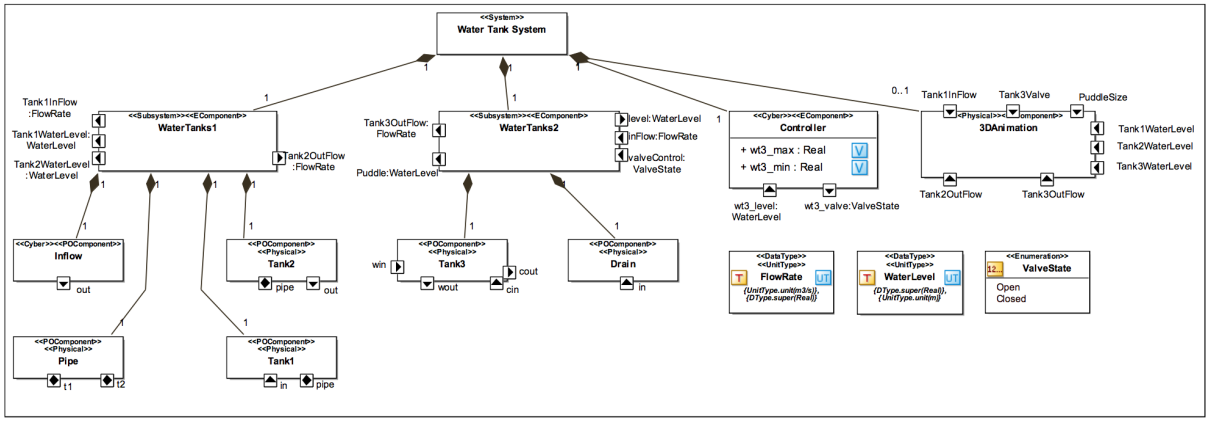


Figure 8: Architecture Structure Diagram defining the Three-tank Water Tank system composition

In this Water Tank system model, the water tanks are split between two subsystems: *WaterTanks1* subsystem contains the *Source*, two *Water Tank* and *Pipe* components; *WaterTanks2* subsystem comprises a single *Water Tank* and *Drain* components; a cyber component *Controller* contains no other components; and the 3D component is available for visualising the behaviour of the system.

To allow the visualisation FMU to depict the internal workings of the system's components, additional ports have been defined for the *WaterTanks1* and *WaterTanks2* blocks. The *WaterTanks1* component exposes: `Tank1InFlow` – corresponding to the rate of water flowing into *Tank1*; `Tank1WaterLevel` – the water level of *Tank1*; and `Tank2WaterLevel` – the water level of *Tank2*. The *WaterTanks2* component exposes the additional ports: `Tank3OutFlow` – corresponding to the rate of water flowing out of *Tank3* and `puddle` – the current volume of water in the drain (or puddle).

The two water tank subsystems are defined as continuous time models, both with 20-sim as the target platform. The controller component is a VDM-RT discrete event

model.

Two System Block Instances are defined in the model to represent alternative system configurations – they are defined in separate Connections Diagrams (CDs). The CD in Figure 9 defines connections as follows: at the subsystem-level, the output of water from the *WaterTanks1* subsystem is input to the *WaterTanks2* subsystem. This subsystem has two connections with the *Controller* cyber component - regarding the level and valve control.

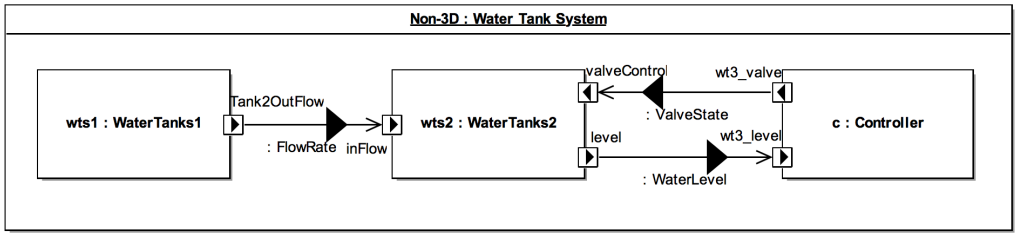


Figure 9: Connections Diagram defining the Three-tank Water Tank system connections

Figure 10 depicts the second CD with several connectors between the system component instances and the 3D visualisation block instance. The connections in Figure 9 are still present, with additional connections sending state information relating to tank water levels, flow rates and controller behaviour to the 3D model.

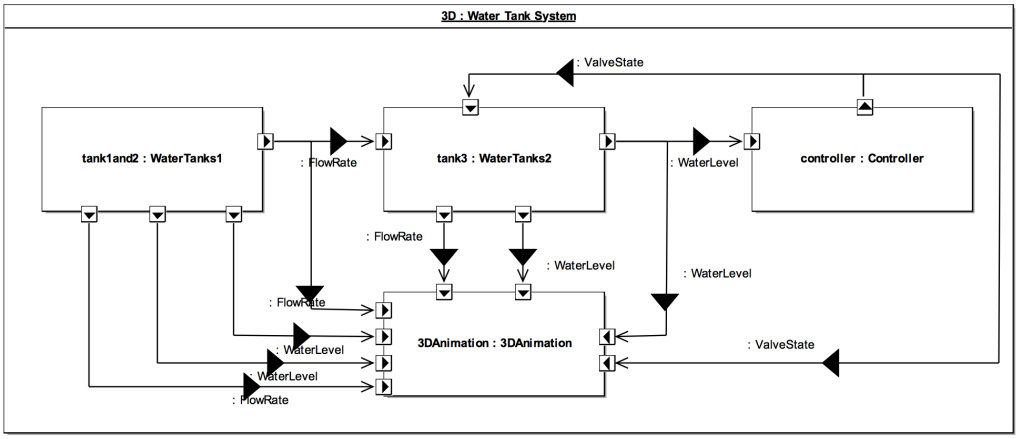


Figure 10: Connections Diagram defining the Three-tank Water Tank system connections and elements for visualisation

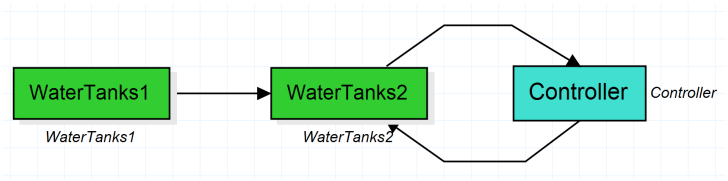
3.4 Multi-model

3.4.1 Models

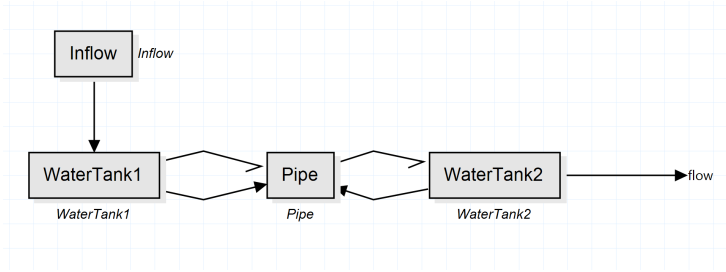
Given the ASD of the SysML model in Section 3.3, three (simulation) models are defined; two 20-sim subsystems and a VDM subsystem as shown in Figure 11(a).

WaterTanks1, WaterTanks2 The partitioning of the 20-sim model is straightforward, with a single signal between the two 20-sim subsystems representing the flow of water between tanks 2 and 3. The rationale behind this split is that the flow rate

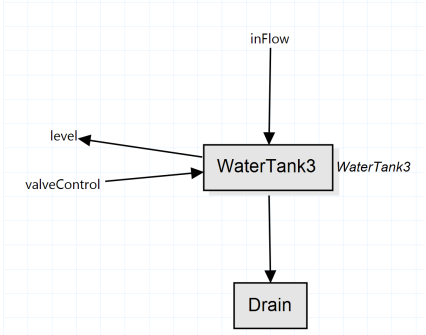
between tank 1 and 2 has a high frequency and amplitude, suggesting that splitting the two tanks would result in erroneous results when time steps are imposed in co-simulation.



(a) Subsystems of Three-tank Water Tank multi-model



(b) WaterTanks1 subsystem



(c) WaterTanks2 subsystem

Figure 11: 20-sim models for the Three-tank Water Tank multi-model

Controller The VDM-RT controller model is a simple controller, which governs *Tank3*. The VDM-RT model contains a *System* class containing *HardwareInterface* and *Controller* objects – *hwi* and *controller*, respectively. The *hwi* object includes the input and output variables of the model and design parameters. The *controller* object is supplied with an instance of the *LevelSensor* (*sensor*) and *ValveActuator* (*valve*) classes – each given access to different parts of the *hwi* object. The *sensor* object represents the sensor that measures the current water level, and *valve* is represents the valve at the bottom of the tank.

The control loop retrieves the current level of water from the sensor and determines whether to set the valve to be open or closed depending on the level compared to some set maximum or minimum value.

3.4.2 Configuration

Two multi-models are defined for the Three Tank Study corresponding to the two System block instances defined in the CDs of the SysML model in Section 3.3.

In the first multi-model (*Non-3D*), there are three FMUs and three connections. The FMUs comprise: WaterTankController, threewatertank1 and threewatertank2 – exported from the VDM-RT and 20-sim models described above. The connections are as follows: firstly between the `flow` port of *WaterTanks1* to the `inFlow` of *WaterTanks2*; secondly between `valveControl` port of the *WaterTanks2* model to the `wt3_valve` of the *Controller*; and finally from the `wt3_level` of the *Controller* to the `level` port of *WaterTanks2*.

In addition, there are two *design parameters* – `wt3_min` and `wt3_max`, both of type `real`.

The complete configuration is given in Figure 12.

```
{
    "fmus":{
        "{c}":"WaterTankController.fmu",
        "{t1}":"threewatertank1.fmu",
        "{t2}":"threewatertank2.fmu"
    },
    "connections":{
        "{c}.controller.wt3_valve":["{t2}.tank2.valveControl"],
        "{t1}.tank1.Tank2OutFlow":["{t2}.tank2.inFlow"],
        "{t2}.tank2.level":["{c}.controller.wt3_level"]
    },
    "parameters":{
        "{c}.controller.wt3_max":1.7,
        "{c}.controller.wt3_min":1.3
    }
}
```

Figure 12: Configuration file for Three-tank Water Tank system

The second multi-model (*3D*) uses the 3D visualisation FMU, and has additional connections to that FMU, as shown in Figure 13.

```
{
    "fmus":{
        "{c}":"WaterTankController.fmu",
        "{t1}":"threewatertank1.fmu",
        "{t2}":"threewatertank2.fmu",
        "{3d}":"3DAnimationFMU.fmu"
    },
    "connections":{
        "{c}.controller.wt3_valve":["{t2}.tank2.valveControl","{3d}.3DAnimationFMU.
            animation.tank3.valve.control"],
        "{t1}.tank1.Tank2OutFlow":["{t2}.tank2.inFlow","{3d}.3DAnimationFMU.animation.
            tank2.outflow"],
        "{t2}.tank2.level":["{c}.controller.wt3_level", "{3d}.3DAnimationFMU.animation.
            tank3.waterlevel"],
        "{t1}.tank1.Tank1InFlow":["{3d}.3DAnimationFMU.animation.tank1.inflow"],
        "{t1}.tank1.Tank1WaterLevel":["{3d}.3DAnimationFMU.animation.tank1.waterlevel"],
        "{t1}.tank1.Tank2WaterLevel":["{3d}.3DAnimationFMU.animation.tank2.waterlevel"],
        "{t2}.tank2.Tank3OutFlow":["{3d}.3DAnimationFMU.animation.tank3.outflow"],
        "{t2}.tank2.puddle":["{3d}.3DAnimationFMU.animation.drain.puddle"]
    },
    "parameters":{
        "{c}.controller.wt3_max":1.7,
        "{c}.controller.wt3_min":1.3
    }
}
```

Figure 13: Configuration file for Three-tank Water Tank system

3.5 Co-simulation

Using the INTO-CPS Co-simulation Engine (COE), we may simulate the three FMU multi-model. We are able to log the water level of tank 3 and the flow rate between tank

2 and 3. These values are shown in the graph in Figure 14, using a fixed step size of *0.05*. A simulation time of at least 20 seconds is recommended so to observe changes in controller behaviour.

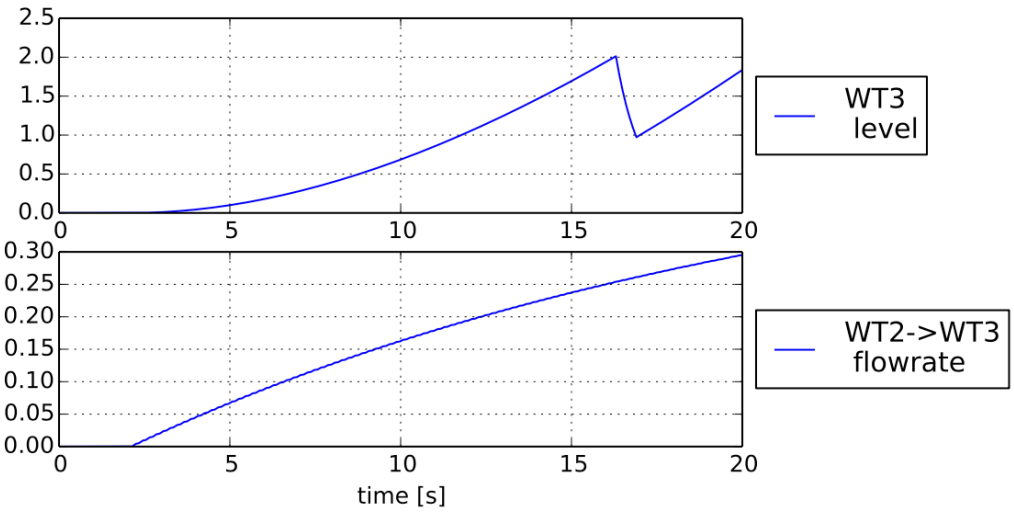


Figure 14: Simulation results using the INTO-CPS COE

The results in the graph correspond closely to those of the baseline Crescendo model illustrated in [FGP+15]. During simulation, the water level raised to the maximum value (2.0 meters) and at 16.3 seconds the tank 3 valve is opened by the VDM-RT controller and the level drops to just below the minimum (1.0 meters) and at 16.9 seconds the valve is closed and the water level begins to rise again.

Co-simulating the 3D multi-model opens a 3D visualisation window as shown in Figure 15 which depicts the state of the Three-tank system as the simulation progresses.

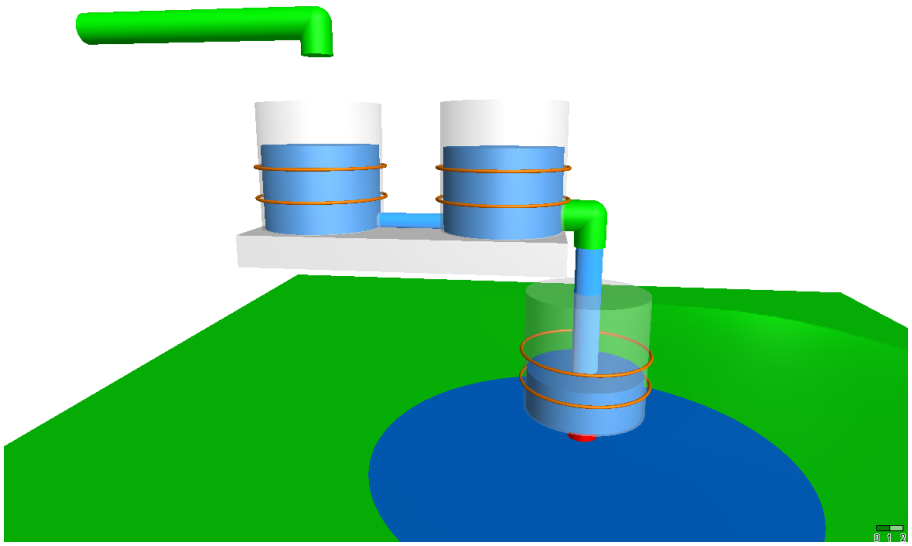


Figure 15: 3D visualisation of the Three-tank Water Tank system

3.6 Analyses and Experiments

3.6.1 Design Space Exploration

A simple DSE experiment is included in the project, which demonstrates the use of the DSE tool support. The experiment varies the two design parameters of the study – `wt3_min` and `wt3_max`. These parameter values may be set between 0.2 and 2.0 in intervals of 0.2. A constraint on the parameters (`controller.controller.wt3_max` > `controller.controller.wt3_min`) ensures that the maximum water value is always larger than the minimum water level.

Two objectives are defined: *cumulativeDeviation* and *vCount*. The first objective, *cumulativeDeviation*, is to minimise the cumulative deviation from a desired level - set to 1.0. The second objective, *vCount*, is to minimise the number of valve operations – i.e. have a lower number of valve state changes. The use of Pareto ranking, minimising both objectives gives the resultant graph in Figure 16.

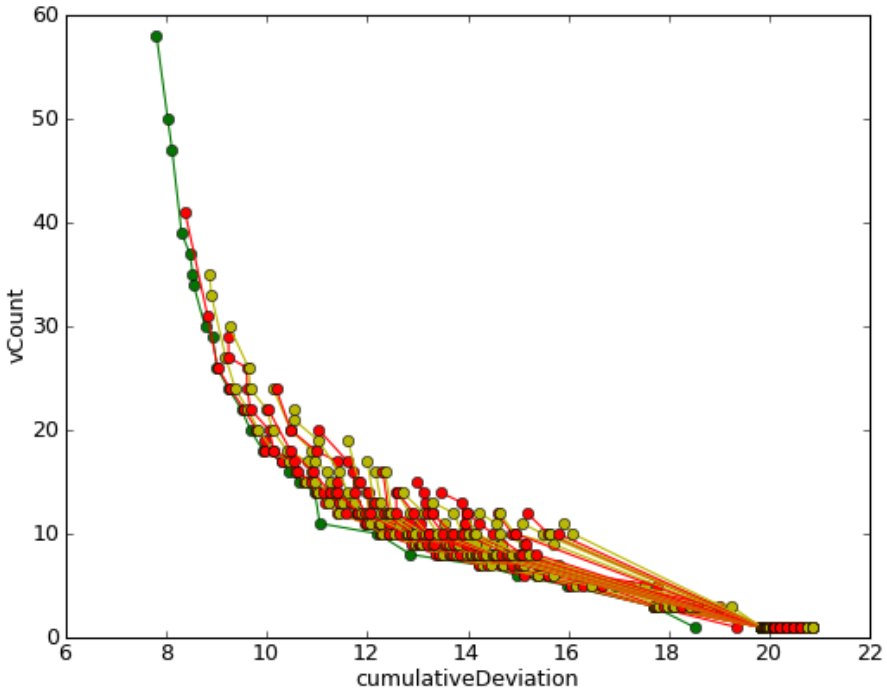


Figure 16: Design Space Exploration Pareto graph of the Three-tank Water Tank system

From the results we see that there is a clear tradeoff to be made between levels which optimise each objective – it is for the engineer to determine which of these is more important. The green line on the graph (the left-most set of results) gives this 'non-dominated' set of results – also given as a table as in Figure 17. In broad terms the ranking shows: levels closer to the desired level (e.g. `wt3_min` = 1.0 and `wt3_max` = 1.05) produce results with a lower cumulative deviation, but higher valve operation count; and a minimum level further from the desired level (e.g. `wt3_min` = 0.2) produces results with a lower valve operation count, but higher cumulative deviation.

Rank	cumulativeDeviation	vCount	controller.wt3_max	controller.wt3_min
1	7.80431475533	58	1.05	1.0
1	8.02778269891	50	1.1	1.0
1	8.11446600624	47	1.05	0.95
1	8.30260988068	39	1.1	0.95
1	8.46667360458	37	1.15	1.0
1	8.52899271589	35	1.15	0.95
1	8.54100674486	34	1.1	0.9
1	8.77136015498	30	1.15	0.9
1	8.92259613716	29	1.2	0.95
1	9.01064334395	26	1.15	0.85
1	9.23326335649	24	1.2	0.85
1	9.49401559283	22	1.25	0.85
1	9.67382697167	20	1.25	0.8
1	9.92779589531	18	1.25	0.75
1	10.3043439504	17	1.35	0.8
1	10.439785803	16	1.3	0.7
1	10.6608278487	15	1.35	0.7
1	10.9554164342	14	1.4	0.7
1	11.0536263724	11	1.5	0.7
1	12.2099586821	10	1.5	0.55
1	12.8636504201	8	1.75	0.55
1	14.2211590453	7	1.55	0.35
1	14.9876347977	6	1.8	0.35
1	15.983226619	5	1.5	0.3
1	17.693808949	3	1.65	0.3
1	18.5082402097	1	1.45	0.2

Figure 17: Design Space Exploration Pareto front table of the Three-tank Water Tank system

3.6.2 Test Automation

Test automation can also be applied to the controller of the three-tank example. A SysML model exists that represents the test model for this system which can be used to produce tests for models and implementations of the controller in RT-Tester[6]. The model consists of a specified System Under Test (SUT), which in this case corresponds to the controller, and the Test Environment (TE) which is the rest of the water tank system, but specifically the water tank the controller is monitoring. A screen shot giving an overview of the test model is shown in Figure 18.

The SUT and TE are specified using the blocks *SystemUnderTest* and *TestEnvironment*, respectively. The SUT block an input flow port called *Stimulation* of type *Interface1* and an output port of type *Interface2*. The TE has the same ports but in the opposite direction. *Interface1* specifies the shared variables that the SUT will read from. In this

[6]Note that this is a different SysML model used for the co-simulation multi-model.

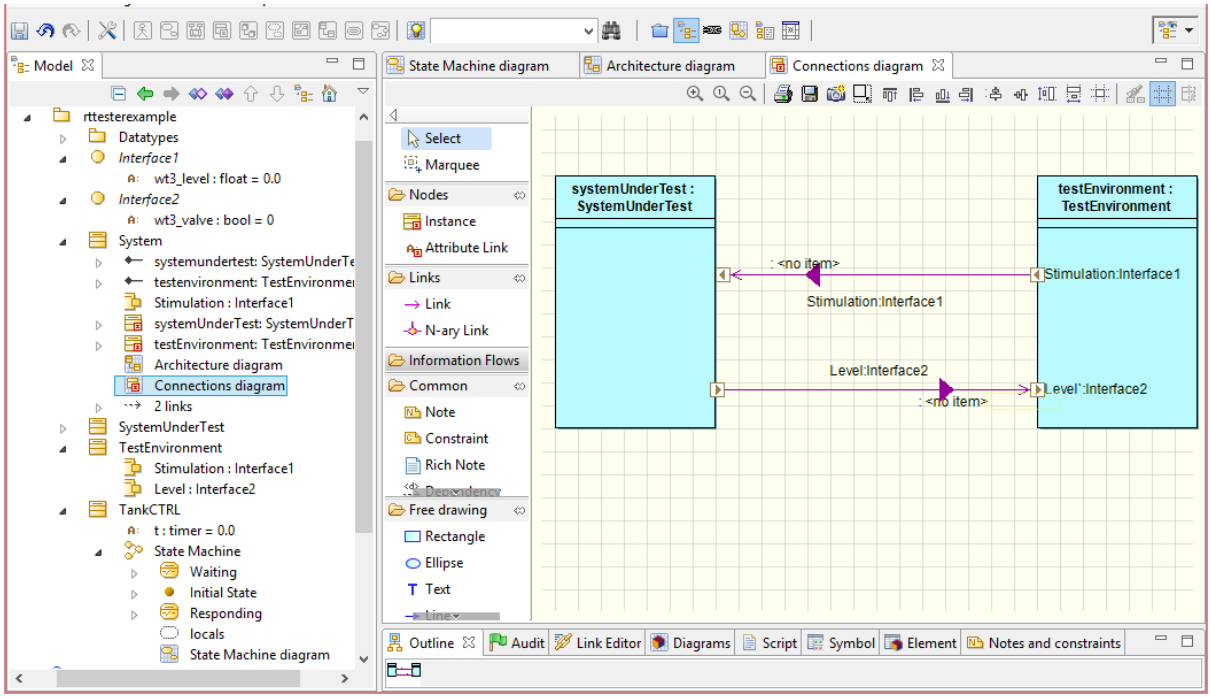


Figure 18: Overview of the three-tank test model in Modelio

case it consists of a single variable *wt3_level*, as seen on the right, which corresponds to the FMU input. Likewise, *Interface2* specifies the shared variable that the SUT will write to, in this case the variable *wt3_valve* which gives the valve status. The two blocks are linked together so that the SUT and TE can communicate on these channels.

In order to generate tests it is necessary to specify an abstract model for the controllers behaviour, which should be modelled using a timed state machine. We thus created the SysML state machine diagram shown in Figure 19. The three-tanks controller is relatively simple and so the state machine has only two states. The **Waiting** state means that the controller is waiting until sufficient time has elapsed to poll the sensors and act accordingly. It has a single outgoing transition with the guard `t.elapsed(1000)`. The variable `t` is a timer for this state machine. It advances in time and can be checked and reset at certain points, rather like a stop-watch. The state machine changes to the **Responding** state once 1000 ms (1 s) has elapsed.

The **Responding** state contains the main decision logic for the controller. It has three outgoing edges with guards and actions (the latter are not shown). If the water level polled on variable *wt3_level* remains within the safe zone of between 1 and 2 then the state machine returns to state **Responding** with no action. If the water level is greater than or equal to 2, the *wt3_valve* variable is set to 0 to shut off the valve, and the controller returns to the **Waiting** state. Otherwise, if the level is less than or equal to 1, then the valve is turned on by setting *wt3_valve* to 1.

This behavioural model must be input into RT-Tester to generate and execute tests. We do this by first exporting XMI from Modelio by selecting the project name, and then the menu item *Import / Export > Export > XMI export*. The model can then be imported from RT-Tester by selecting *Project > Model-based testing > Import model > Import from file*. This currently must be done from the existing *water-tanks* model available in RT-

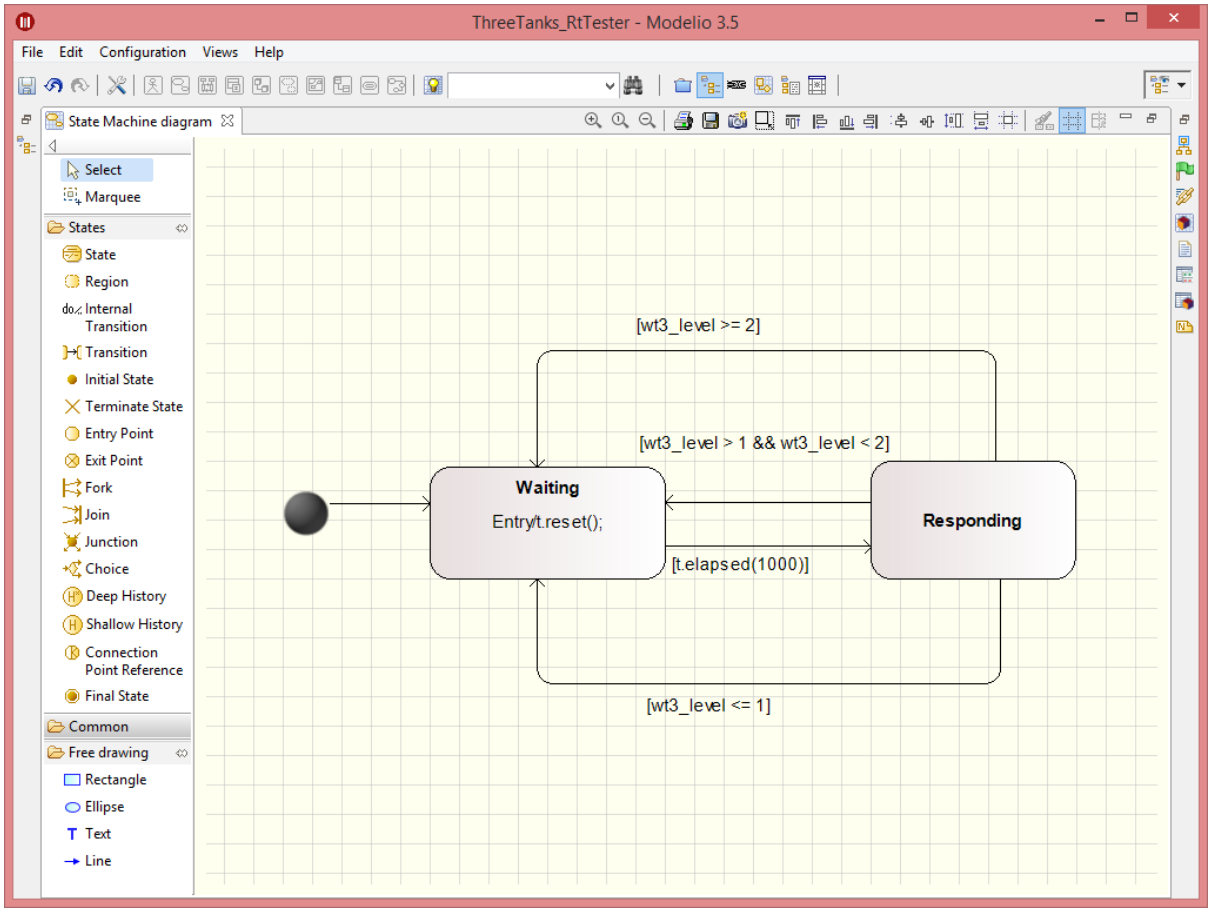


Figure 19: State machine for abstract behaviour of three-tank controller

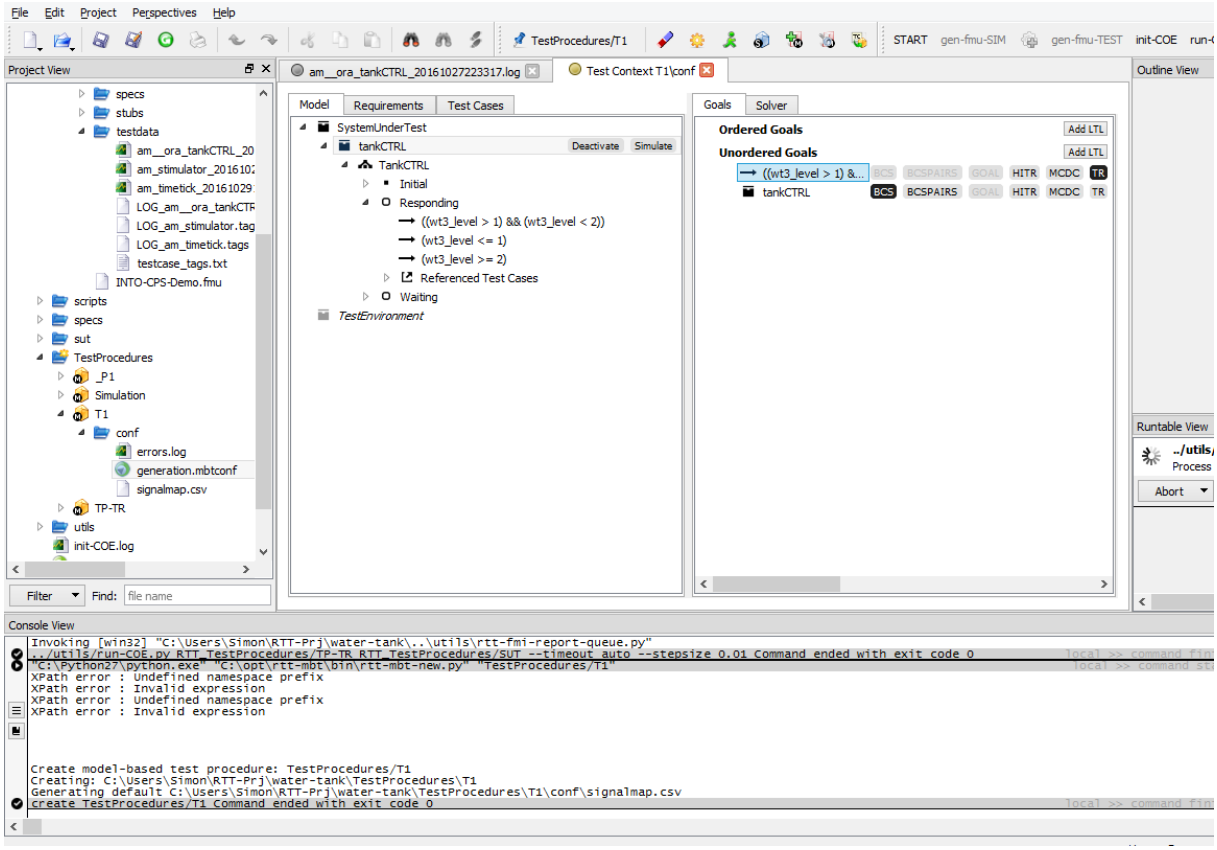


Figure 20: Configuring a test procedure

Tester to ensure that the FMU is correctly set up. One of the standard test procedures can then be run, or a new test procedure can be created by selecting *New* > *MBT Test Procedure* and then using test procedure `TestProcedures/_P1` as a template. Suitable tests can be configured from the *conf* > *generation.mbtconf* file in the new directory as illustrated in Figure 20.

The project can then be prepared for executing the tests through the *init-Project* command that creates the test FMUs. Finally, the test procedure can be executed by starting the COE, and then using the *run-COE* command. This will produce output which is exemplified in Figure 21.

3.6.3 Code Generation

The VDM-RT model, **WaterTankController** can be exported from Overture as a C code FMU, in addition to the tool wrapper FMU as used above. The *WaterTankController-SourceCode.FMU* included in this pilot is obtained directly from Overture using the "Export Source Code FMU" option. However, this FMU does not contain binaries for co-simulation and so one may use the *FMU Builder* included in the INTO-CPS Application to compile FMUs for Windows, Mac and Linux.

This process has been performed and the resultant FMU is included in the pilot in the FMUs folder; *WaterTankController-Standalone.FMU*. One example experiment available is to switch this FMU for the tool wrapper version – *WaterTankController.FMU* – and

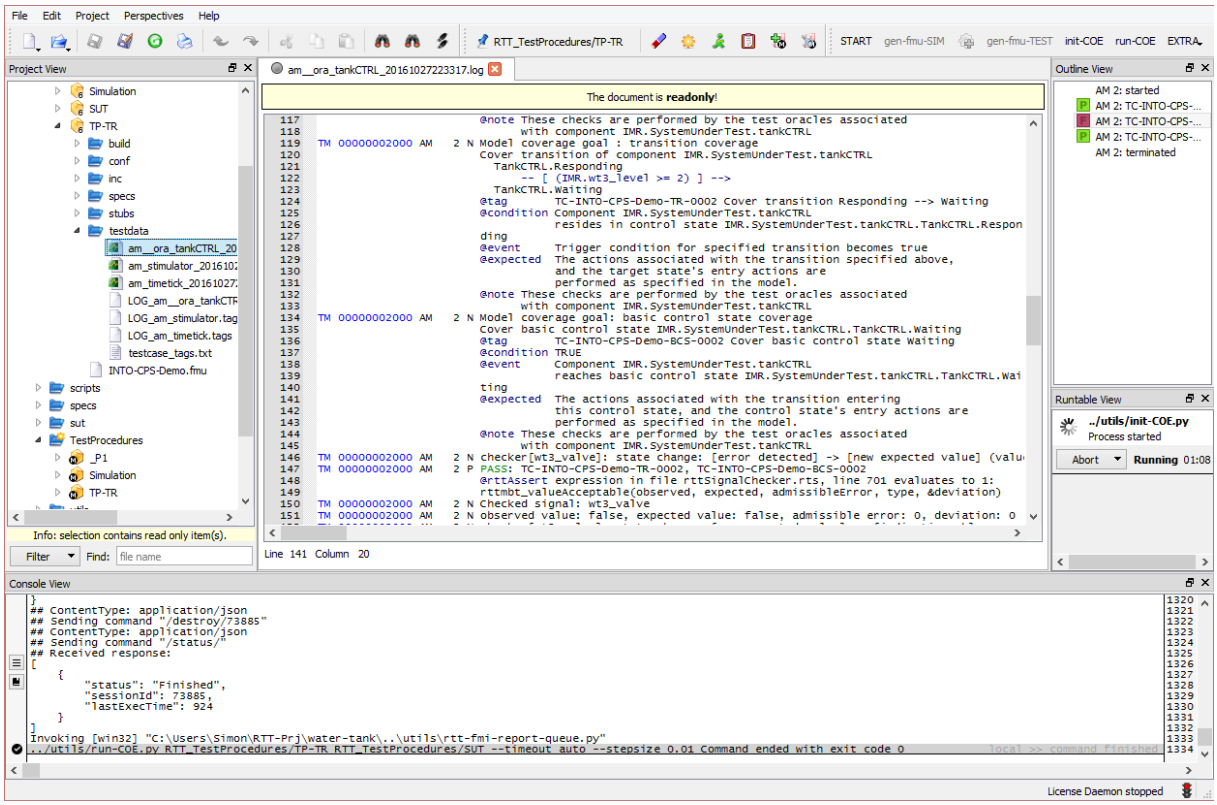


Figure 21: Test procedure output

compare results.

4 Fan Coil Unit (FCU)

4.1 Example Description

This example is inspired by the Heating Ventilation and Air Conditioning (HVAC) industrial case study developed in Task T1.3. The Fan Coil Unit (FCU) aims to control the air temperature in a room through the use of several physical components and software controllers. Water is heated or cooled in a *Heat Pump* and flows to the *Coil*. A *Fan* blows air through the *Coil*. The air is heated or cooled depending upon the *Coil* temperature, and flows into the room. A *Controller* is able to alter the fan speed and the rate of the water flow from the *Heat Pump* to the *Coil*. In addition, the room temperature is affected by the walls and windows, which constitute the environment of the FCU.

The aim of the system is to maintain a set temperature in the single room in which the FCU is located. The system is outlined in Figure 22.

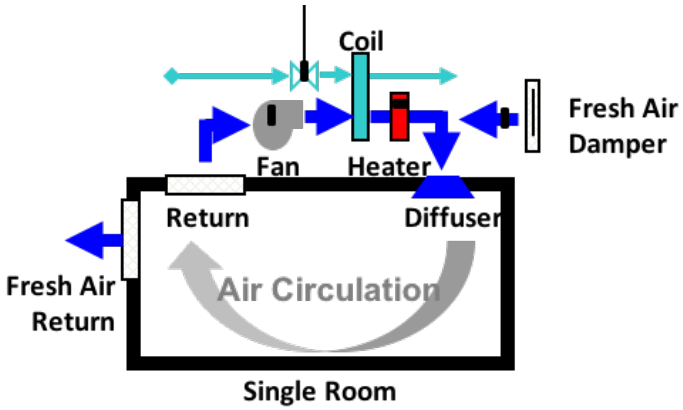


Figure 22: Overview of the fan coil unit (FCU) example

4.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-fcu` in the *master* branch. There are several subfolders for the various elements: DSEs - contains various work in progress DSE scripts to alter CT and DE parameters; `FMU` contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations; `SysML` – contains the SysML model defined for the study; `resources` – various images for the purposes of the readme file; and `userMetricScripts` – contains files for DSE analysis.

The `case-study_fcu` folder can be opened in the INTO-CPS application to run the various co-simulations as detailed in this document. To run a simulation, expand one of the multi-models and click 'Simulate' for an experiment.

4.3 INTO-CPS SysML Profile

Three constituent parts are defined – shown in Figure 23: the *RoomHeating* subsystem, a *Controller* cyber component and the physical *Environment*. The first is a continuous subsystem and comprises the *Room* and *Wall* components. The figure defines the model platform to be 20-sim, however, this could be OpenModelica too. All of the physical elements of the system are contained in a single CT model. The controller subsystem is a cyber element and modelled in VDM-RT.

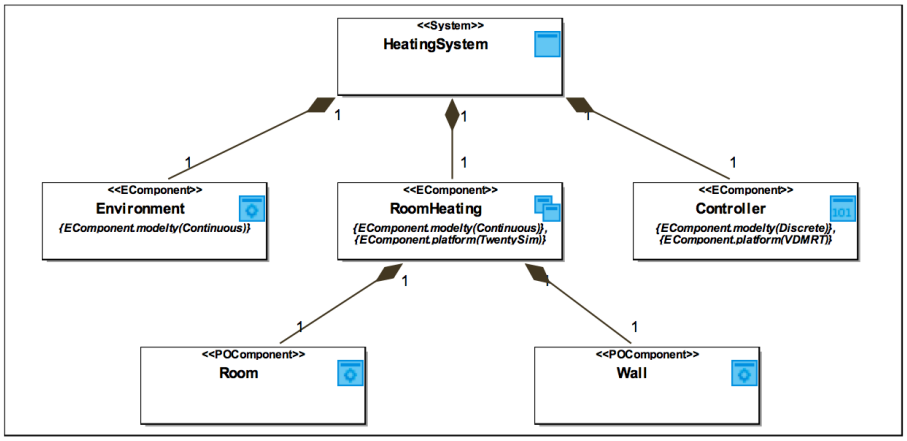


Figure 23: SysML Architecture Structure Diagram using INTO-CPS profile corresponding to baseline models

The connections between components, shown in Figure 24, are similar to those in the baseline CT models, although it should be noted that the subsystem hierarchy is shown, with the *Room* component supplying and receiving the flows of the *RoomHeating* subsystem. The connections between CT and DE models show the interface that is managed during the co-simulation. Specifically, the Room Air Temperature (*RAT*) from the CT system is communicated to the controller, which sets the fan speed *fanSpeed* and the valve open state *valveOpen* used by the Room component model *r*, with the aim of achieving the Room Air Temperature Set Point *RATSP* provided by the user in the *Environment*.

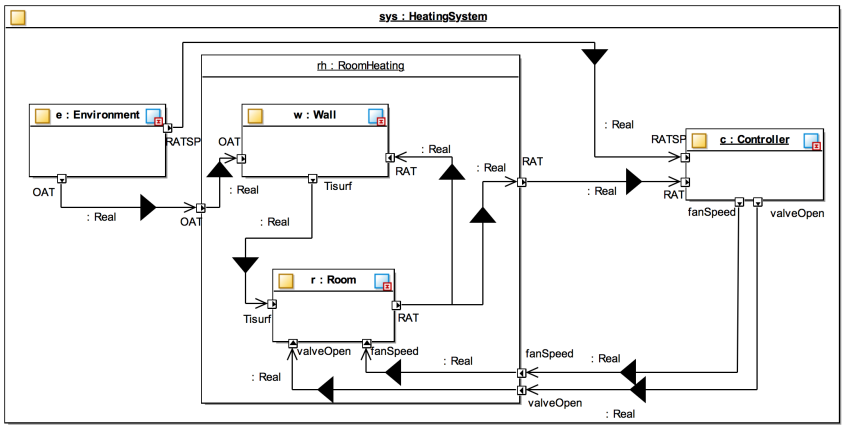


Figure 24: SysML Connection Diagram using INTO-CPS profile corresponding to baseline models

4.4 Multi-model

4.4.1 Models

This pilot comprises two 20-sim models: `RoomHeating` and the `Environment`; an OpenModelica `RoomHeating_OM` model; and a `Controller` VDM-RT model.

RoomHeating.emx Figure 25 shows the *RoomHeating* subsystem with blocks for the room and the wall. The model takes inputs for the required room temperature, outside air temperature, fan speed and valve control. The model outputs the current room air temperature.

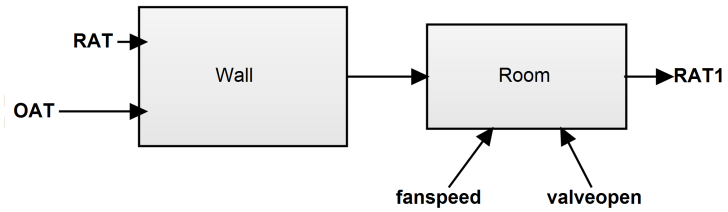


Figure 25: RoomHeating model

RoomHeating_OM.mo The OpenModelica version of the *RoomHeating* subsystem is similar to that of the 20-sim version – it also comprises blocks for the room and wall, with the same interface. The block diagram is shown in Figure 26.

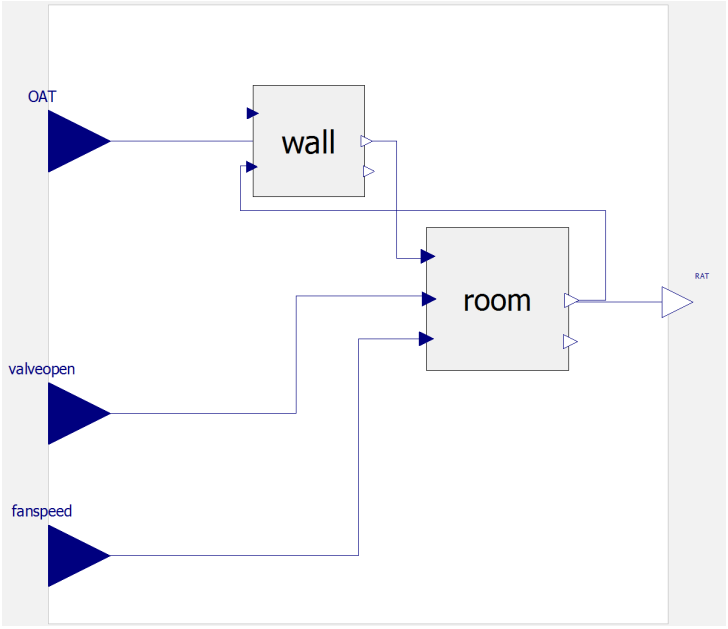


Figure 26: RoomHeating model

Environment.emx The *Environment* model, in Figure 27, provides data on the environment outside air temperature and scenario data based on change of room temperature set point.

ControllerFCU The VDM controller model comprises a *Sensor* class, which provides access to the current room temperature, and a *LimitedActuator* class, which pro-

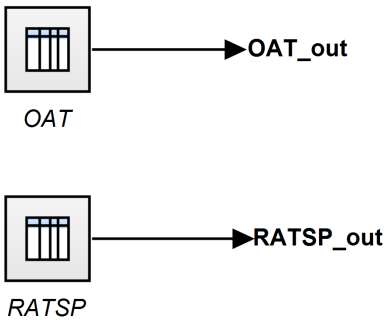


Figure 27: Environment model

vides output for the valveOpen and fanSpeed values. The actuator is limited such that values fall only between the real values 1.0 and 0.0000001.

4.4.2 Configuration

The multi-model comprises three FMUs and five connections. The FMUs – `FCUController.fmu`, `RoomHeating.fmu` and `Environment.fmu` – are exported from the VDM-RT and 20-sim models.

The connections are as follows:

- from the *EnvironmentFMUs* `RAT_OUT` port to the *ControllerFMU* `RATSP` port;
- from the *EnvironmentFMUs* `OAT_OUT` port to the *RoomHeatingFMU* `OAT` port;
- from the *ControllerFMUs* `valveOpen` port to the *RoomHeatingFMU* `valveopen` port;
- from the *ControllerFMUs* `fanSpeed` port to the *RoomHeatingFMU* `fanspeed` port; and
- from the *RoomHeatingFMU* `RAT` port to the *ControllerFMUs* `RAT` port.

There are three parameters to set: *lambdaWall* and *rhoWall* which define the `Wall` thermal conductivity and density respectively, and *controllerFrequency*, which defines the frequency of the `Controller`. The standard parameters for these are 1.1192, 1312 and 1000000000 respectively. These may be adjusted for the purposes of DSE.

4.5 Co-simulation

Co-simulation of the full scenario (outside air temperature and room set point) has a duration of 6800 seconds. Running the two multi-models produces the same results. The results as displayed in the INTO-CPS application are shown in Figure 28, and values of note sent between FMUs are shown separately in Figure 29.

The results in Figure 29 show that the set point (top left) is toggled between 20 and 0, with the fan (and valve) are adjusted to achieve the set point. The bottom right graph shows the ultimate result of the simulation – that the Room Air Temperature (RAT)

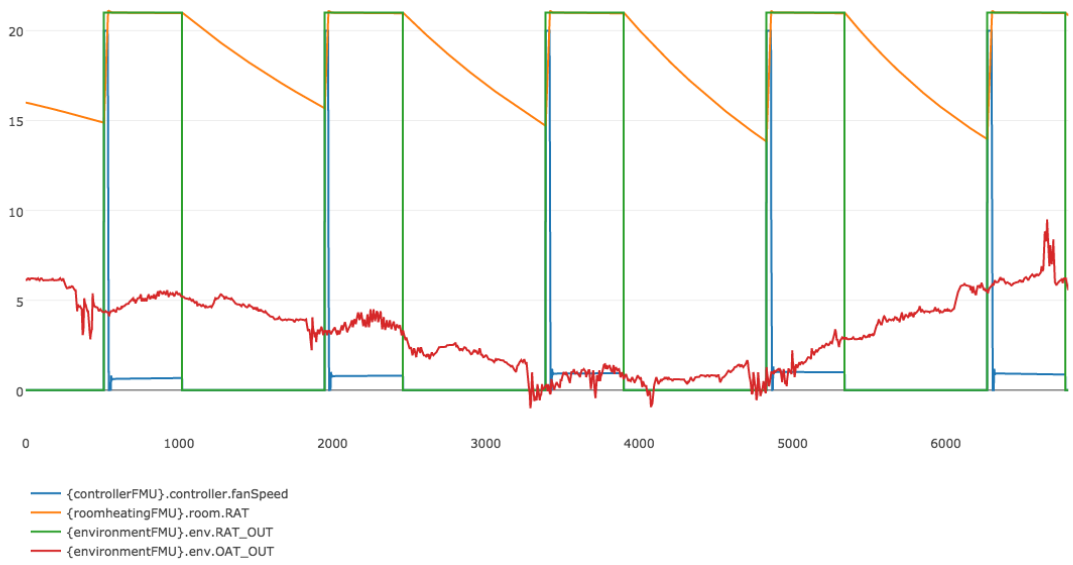


Figure 28: Co-simulation results as shown in lNTO-CPS application live stream

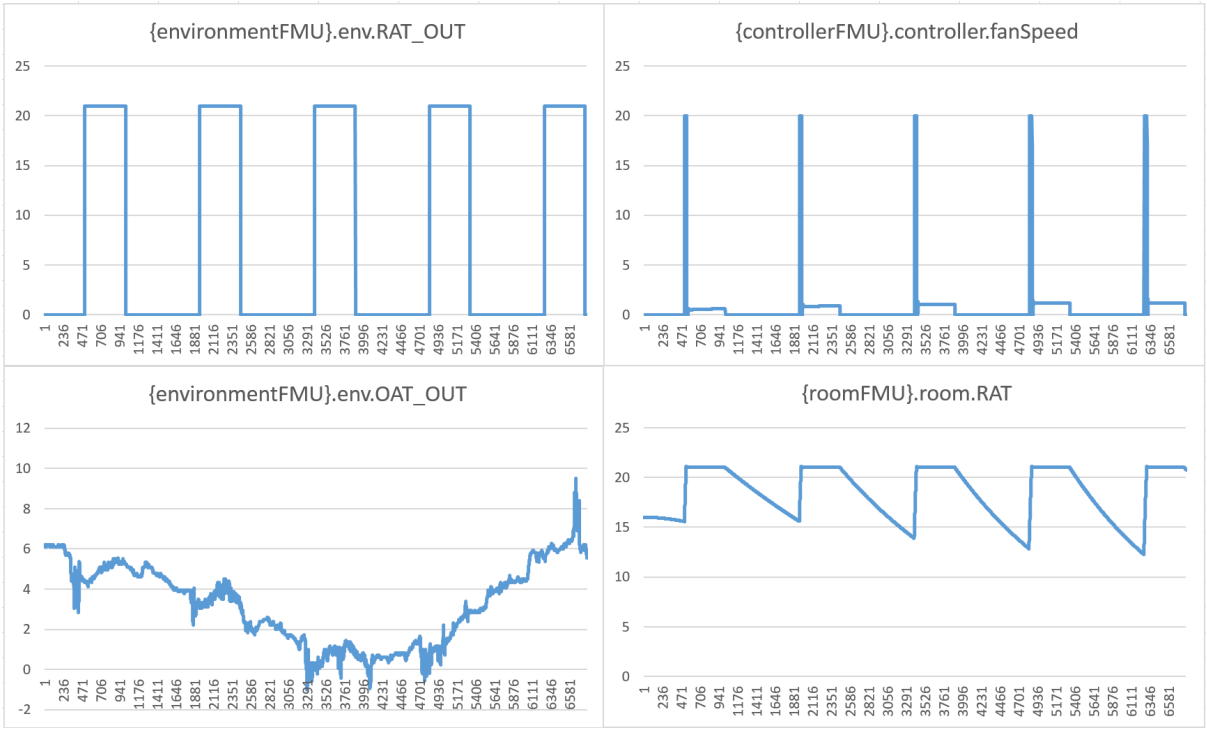


Figure 29: Co-simulation results as shown in graphs from result log files

meets the set point, maintains that temperature whilst required and then slowly drop in temperature until the set point returns to 20.

As mentioned in Section 4.4.2, the *lambda_ Wall*, *rho Wall* and *controllerFrequency* design parameters may be altered to test different wall properties and their effect on the overall CPS.

4.6 Analyses and Experiments

4.6.1 Design Space Exploration

Detailed in Section 4.4.2, the multi-model has three design parameters, *lambda_ Wall*, *rhoWall* and *controllerFrequency* which define the wall thermal conductivity, wall density and controller frequency respectively, which may be altered to perform DSE.

This example has 2 DSE experiments:

fcu-walls: The parameter values for *lambda_ Wall* ranges from 0.1192 to 10.1192 in intervals of 0.25, and the *rhoWall* value may be either 1312.0 or 1400.0. In this experiment a wide range of *lambda_ Wall* values (40 in total) provides a 80-model design space – no constraints are defined. Two objectives are defined, using internal DSE functions, *energyConsumed* and *averageTemperature*. The first objective is to retrieve the maximum energy usage value, this is essentially the final value of the energy port. The second is to return the mean RAT – that is the average temperature of the RAT, which shows how the room heats and cools over time depending upon the different wall values.

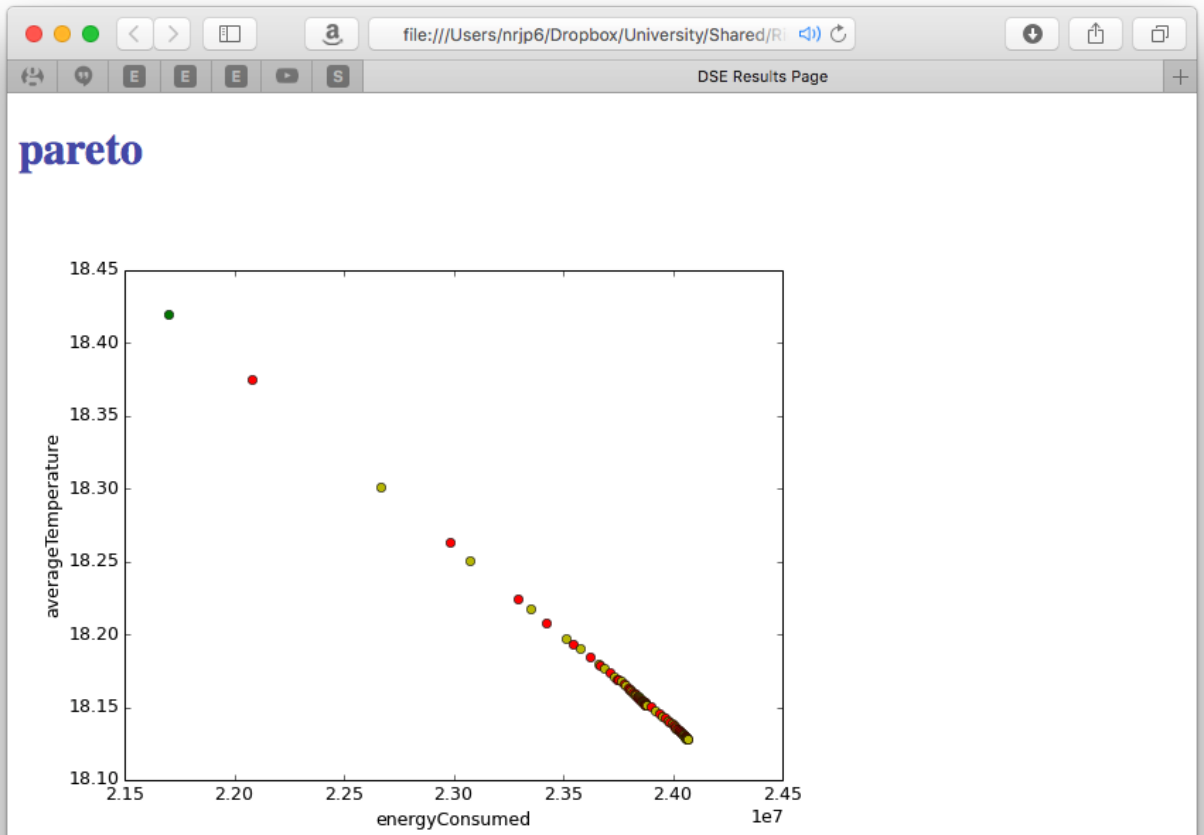
The Pareto ranking seeks to maximise the average temperature and minimise the energy consumed. Figure 30 shows that there is one experiment at rank 1 (the green dot), which indicates that (as is intuitive) the best design is that with the lowest thermal conductivity and greater density.

fcu-walls-controller: The second experiment varies the *lambda_ Wall* and *rhoWall* as earlier, but with a different collection of values: *lambda_ Wall* ranges from 0.1192 to 10.1192 in intervals of 1.0, and the *rhoWall* value ranging from 1300.0 to 1700.0 in intervals of 1000.0. In addition, the *controllerFrequency* parameter is varied between 800000000 and 1200000000 in intervals of 100000000. A space of 250 designs is defined. The same objectives and rankings are used as the above DSE experiment.

Figure 31 shows that there are five experiments at rank 1 (the green dots), which indicates that (as is intuitive) the best design is that with the lowest thermal conductivity and greater density, with the controller frequency providing only marginal impact. It is interesting to note that the frequency does have an impact on both the *energyConsumed* and *averageTemperature* objectives.

4.6.2 Test Automation and Model Checking

In this pilot study, test automation is applied to the system, including not only the discrete FCU controller but also the continuous room and wall. A test model in SysML models both discrete and continuous parts together. Accordingly, we developed two SUT implementations that have different scope. The first SUT has a controller implemented in C, and uses two 20-Sim models for the *Room* and the *Wall* separately. While the second SUT has all (the *Controller*, *Wall* and *Room*) implemented in C in the same program. With the two SUTs and an additional SUT called `Simulation` that is automatically generated from the test model in RT-Tester, we can construct three test automation configurations. And each configuration corresponds to one SUT.



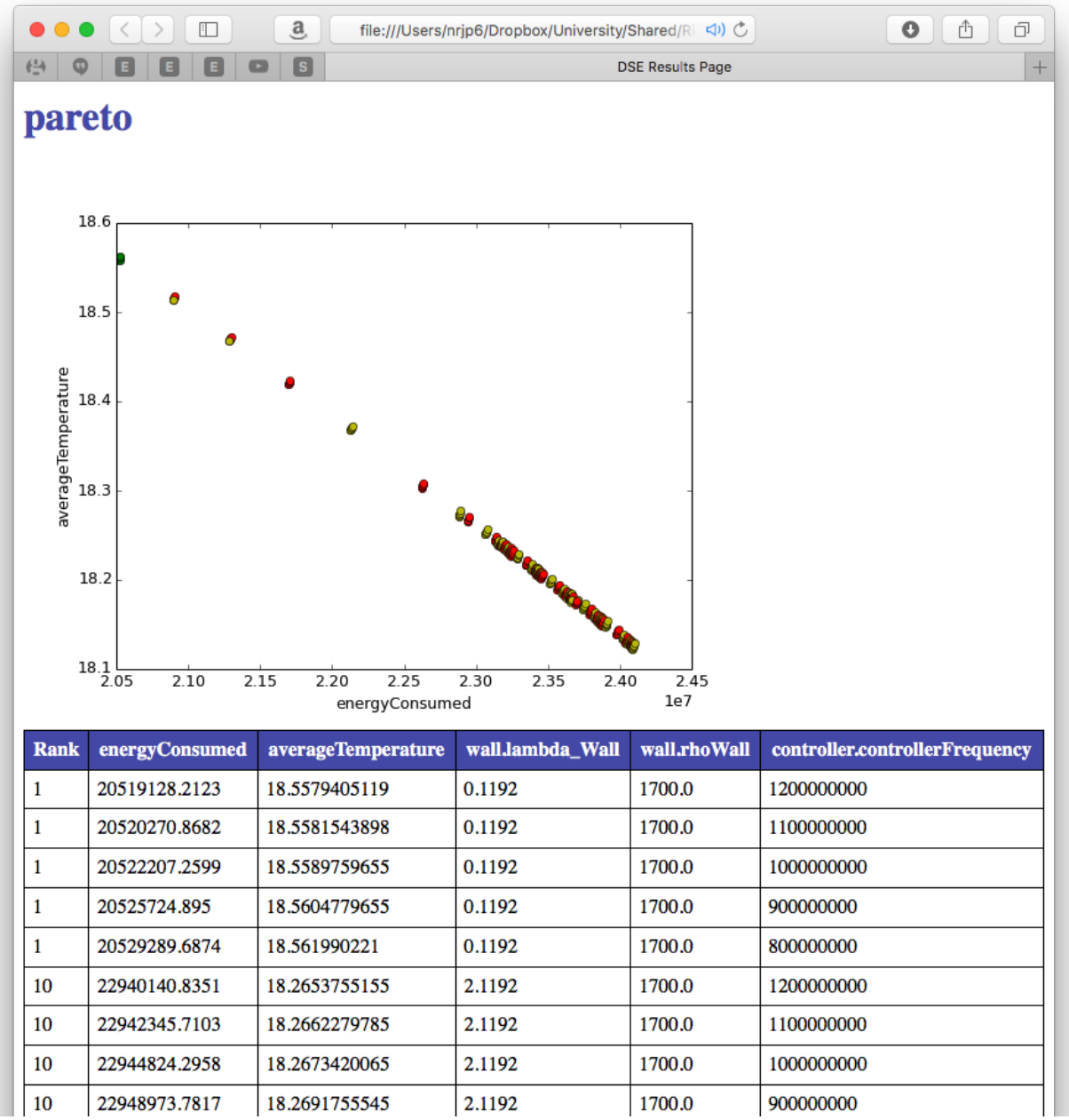
Rank	energyConsumed	averageTemperature	wall.lambda_Wall	wall.rhoWall	controller.controllerFrequency
1	21700698.1034	18.4202118552	0.1192	1400.0	1000000000
10	23547082.9842	18.1930154625	0.8692	1312.0	1000000000
11	23575608.7888	18.1900573495	1.6192	1400.0	1000000000
12	23623004.341	18.1844796461	1.8692	1400.0	1000000000
13	23659648.1983	18.1801463718	2.1192	1400.0	1000000000
14	23665812.5533	18.1785840922	1.1192	1312.0	1000000000
15	23688772.6006	18.1766743987	2.3692	1400.0	1000000000
16	23712445.4096	18.1738250828	2.6192	1400.0	1000000000

Figure 30: DSE results for **fcu-walls** experiment

Subsequently, we present the test model in SysML, then the two SUT implementations, and finally test results for three different configurations.

Test Model The developed test model consists of a *Controller*, a *Room* model, and a *Wall* model. We use the Euler method to model continuous parts *Wall* and *Room*.

Similar to the test model for *Three-tank Water Tank* in Section 3.6.2, the SysML model consists a SUT and a TE. They are represented by the blocks *SystemUnderTest* and *TestEnvironment*. Both of them have two ports with *Observables* and *Stimuli* interfaces respectively. From the SUT perspective, the port of *Observables* is an output port, and the port of *Stimuli* is an input port. However, their corresponding ports in the TE are in the opposite direction. For brevity, we omit the ASD and CD diagrams of the *System*



Rank	energyConsumed	averageTemperature	wall.lambda_Wall	wall.rhoWall	controller.controllerFrequency
1	20519128.2123	18.5579405119	0.1192	1700.0	1200000000
1	20520270.8682	18.5581543898	0.1192	1700.0	1100000000
1	20522207.2599	18.5589759655	0.1192	1700.0	1000000000
1	20525724.895	18.5604779655	0.1192	1700.0	900000000
1	20529289.6874	18.561990221	0.1192	1700.0	800000000
10	22940140.8351	18.2653755155	2.1192	1700.0	1200000000
10	22942345.7103	18.2662279785	2.1192	1700.0	1100000000
10	22944824.2958	18.2673420065	2.1192	1700.0	1000000000
10	22948973.7817	18.2691755545	2.1192	1700.0	900000000

Figure 31: DSE results for **fcu-walls-controller** experiment

in this document.

Inputs and Outputs The *SystemUnderTest* receives the following inputs (stimuli) from the *TestEnvironment*:

- OAT: Outside Air Temperature;
- $RATSP$: Room Air Temperature Set Point.

The *SystemUnderTest* provides the following observable outputs to the *TestEnvironment*:

- RAT_out: Room Air Temperature.

SystemUnderTest The *SystemUnderTest* consists of a FCU controller, a wall model, and a room model.

The architecture structure diagram of *SystemUnderTest* is shown in Figure 32.

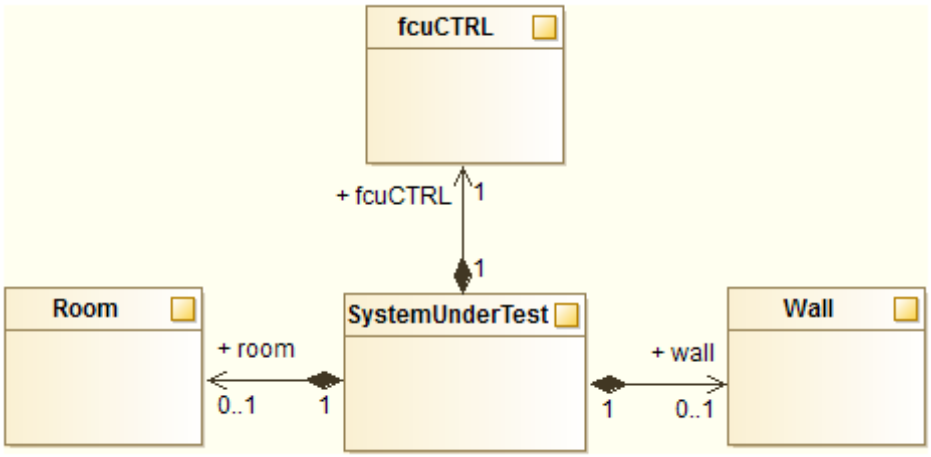


Figure 32: Architecture Structure Diagram of SUT

FCU Controller The *fcuCTRL* receives the following inputs (stimuli):

- *RAT_out*: Room Air Temperature from the *Room*;
- *RATSP*: Room Air Temperature Set Point from the *TestEnvironment*.

The *fcuCTRL* provides the following outputs to the *Room*:

- *fanSpeed*: Fan Speed;
- *valveOpen*: Valve Open State.

The state machine diagram of the *fcuCTRL* is illustrated in Figure 33.

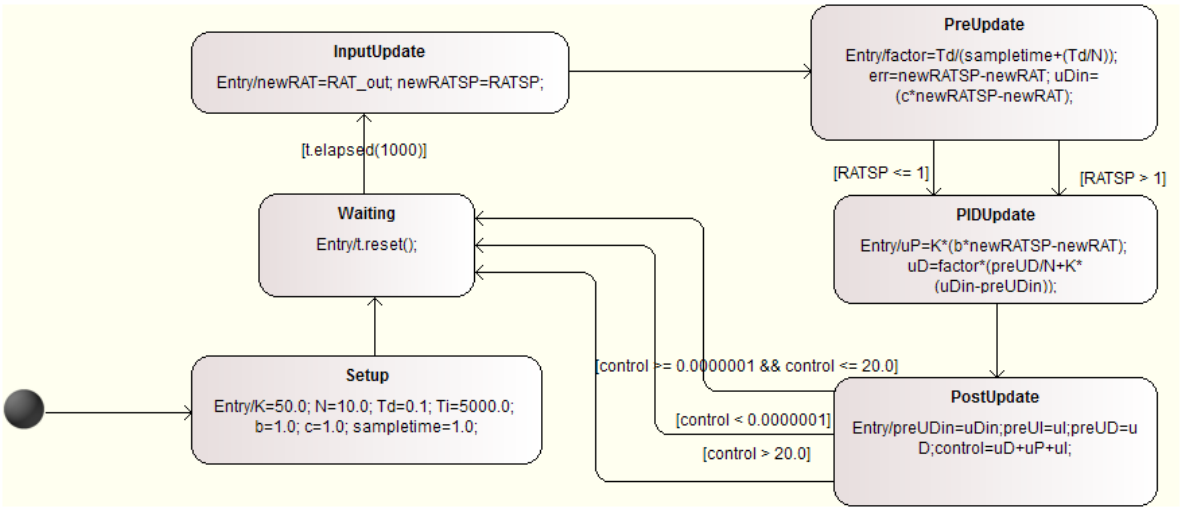


Figure 33: State Machine Diagram of fcuCTRL

In the diagram,

- `Setup` is used to initialize constant variables;
- After `Setup`, the state machine resides in the `Waiting` state, most of the time;
- Then every second, it starts to calculate PID parameters again:
  - At first, the inputs RAT and $RATSP$ are copied to local variables to make sure all subsequent computations refer to same values of RAT and $RATSP$;
  - Then, uP, uD, uI, and other local variables are computed for future use,
  - Next, the summation of uP, uD, and uI is calculated and assigned to *control*,
  - Finally, we limit *control* and assign it to $fanSpeed$ and *valveSpeed* respectively in the transitions from `PostUpdate` to `Waiting`.

Wall The *Wall* receives the following inputs (stimuli):

- RAT_out: Room Air Temperature from the *Room*;
- OAT: Outside Air Temperature from the *TestEnvironment*.

And the *Wall* provides the following outputs to *Room*:

- $Tisurf$: Wall Internal Surface Temperature.

The state machine diagram of the *Wall* is illustrated in Figure 34.

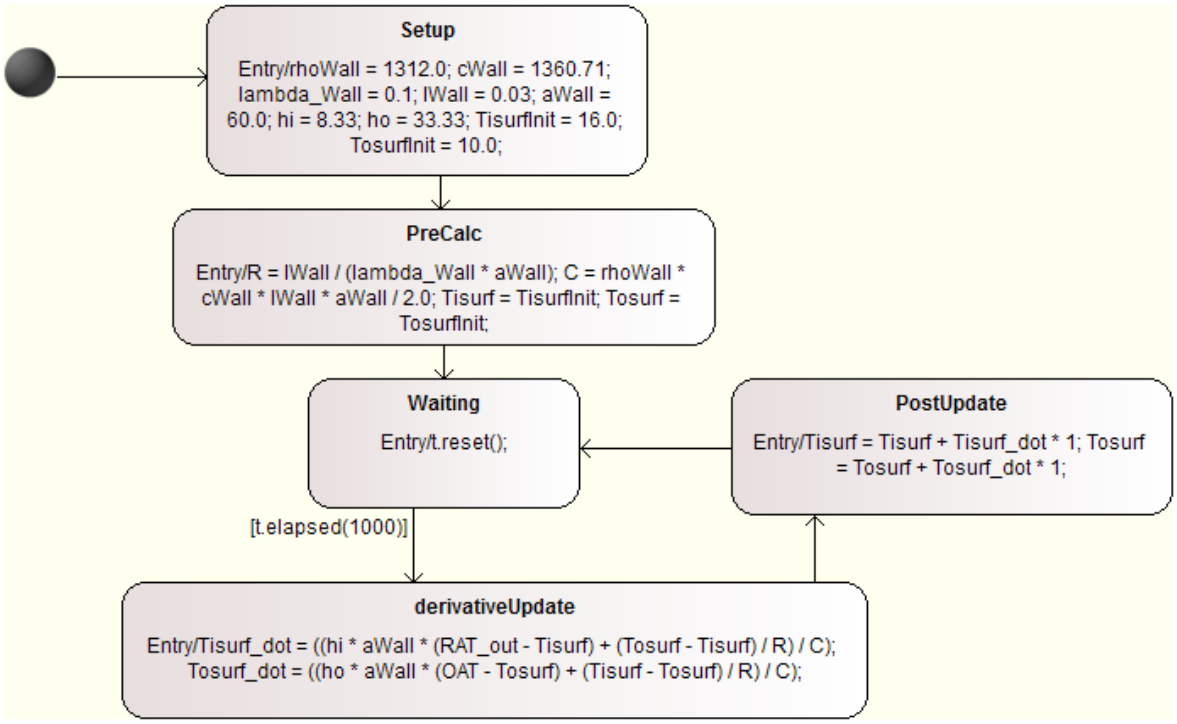


Figure 34: State Machine Diagram of Wall

In the diagram,

- `Setup` is used to initialize constant variables;
- After `Setup`, we calculate the constants R and C, and initialize $Tisurf$ and $Tosurf$ in `PreCalc` according to other constants that are initialized in `Setup`.
- Then the state machine resides in the `Waiting` state, most of the time. And every second, it starts to calculate and update $Tisurf$ and $Tosurf$ again:
  - The derivatives of $Tisurf$ and $Tosurf$ are calculated in `derivativeUpdate`,
  - Then, they are used in `PostUpdate` to update $Tisurf$ and $Tosurf$ according to the Euler Methods.

Room The *Room* receives the following inputs (stimuli):

- $fanSpeed$: Fan Speed from the *fcuCTRL*;
- $valveOpen$: Valve Open State from the *fcuCTRL*;
- $Tisurf$: Wall Internal Surface Temperature from the *Wall*.

The *Room* provides the following outputs to the *fcuCTRL* and *TestEnvironment*:

- RAT_out: Room Air Temperature.

The state machine diagram of the *Room* is illustrated in Figure 35.

In the diagram,

- Similarly, `Setup` is used to initialize constant variables;
- After `Setup`, we set RAT_out to $RATinit$ in `PreCalc`.
- Then the state machine resides in the `Waiting` state, most of the time. And every second, it starts to calculate and update RAT_out:
  - Validality of the inputs $fanSpeed$ and $valveOpen$ are checked at first,
  - Then, the derivatives of Qin (heat transferred to the air in the coil) and $Qout$ (heat lost through room and walls) are calculated,
  - Subsequently, the derivative of RAT is calculated, and intermediate SAT (supply air temperature) and EWT (entering water temperature) are updated as well.
  - In the end, the output RAT_out is updated by the Euler method.

Test Input Simulation The automatically generated test cases from a test model using RTT-MBT in RT-Tester is hardly realistic for a real life scenario. In order to provide reasonable inputs to Co-simulation, we use a real input sequence for OAT and $RATSP$ in `Multi-models`. However, it is not possible for RTT-MBT to give this as input since RTT-MBT uses a SMT solver to generate test cases, or uses a state machine to specify the input sequence in `TestEnvironment` of the test model. We use the second way to specify the input sequence by defining a state machine in a block `TESim` of `TestEnvironment`. The diagram is shown in Figure 36. The specified sequence is shown as SUT_OAT

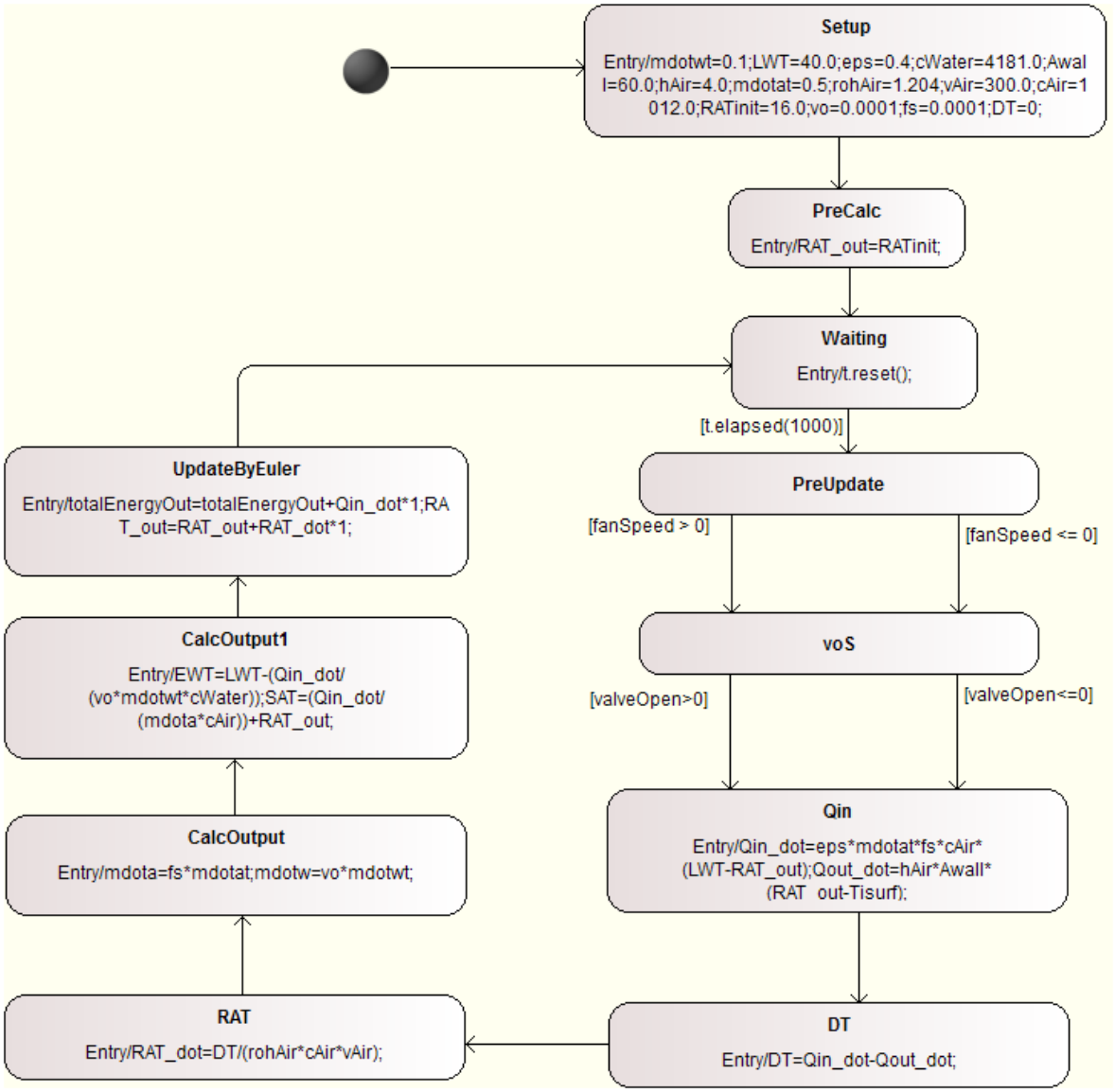


Figure 35: State Machine Diagram of Room

and SUT_RATSP in Figure 38. Basically, OAT changes slightly but $RATSP$ vibrates between 10 and 35 °C.

Manual Implementations of SUT Two SUTs are implemented. The first one is a SUT with the *Controller* in C, and then reuses the *Wall* and *Room* models in 20-Sim or Modelica. And the second one is a standalone C program which has all three parts together. By using a Python script named "build-suts.py", we can build, assemble, and wrap them into FMUs in RT-Tester automatically. The corresponding FMU for the first implementation is named `SUT_FCU_Ctrl.fmu` while that of the second one is named `SUT.fmu`.

Test Results Three test configurations are provided for test automation.

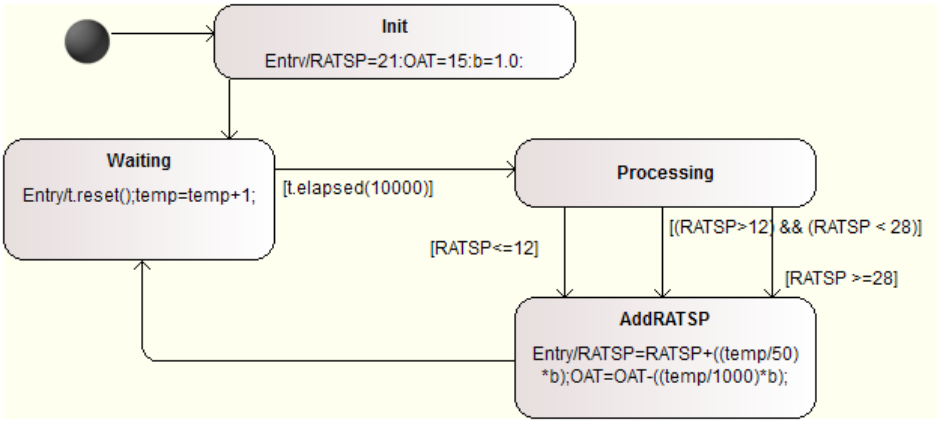


Figure 36: State machine diagram of TESim

SIM `TP` (a test procedure in RT-Tester that provides test inputs and checks expected results) against `Simulation` (a SUT created from the test model),

SUT `TP` against `SUT.fmu` (a SUT having all in the same program and FMU),

SUT_RoomWall `TP` against `SUT_FCU_Ctrl.fmu` and `RoomHeating.FMU` (a FMU from 20-Sim), therefore three FMUs altogether.

We use user defined test cases by LTL formulas in RT-Tester to define a test goal that simulation time is at least 2000 seconds long as shown in Figure 37. In the test goal configuration, `TESim` is activated to simulate the test input sequence as specified in its state machine chart. Then the solver will generate a test data generation report that includes test goals, implicitly covered test coverage cases, signal configurations, and test stimulations and expected behaviour. This test covers all basic control state coverage test cases (25), 45 basic control state pairs coverage test cases in 242, 1 MCDC coverage test case in 3, and 11 transition coverage test cases in 12.

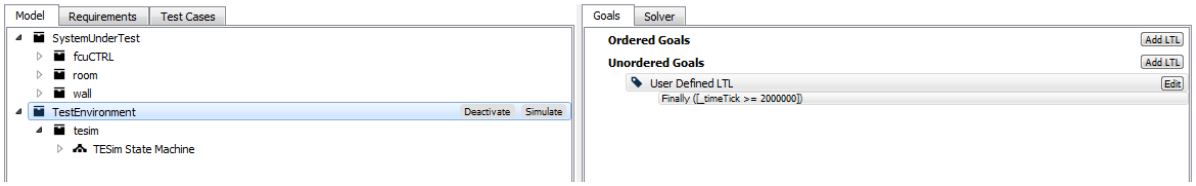


Figure 37: Test Goal Configuration of FCU

Our test results show all these implicitly covered test cases are with `PASS` or `INCONCLUSIVE` verdicts. A comparison of test results for three test configurations with the same test input sequence is displayed in Figure 38.

From the diagram, we can see that

- the change of set points will be reflected in the output *RAT* though there are delays for the output to follow,
- the time to decrease temperature is longer than that to increase temperature,
- the fan speed is occasionally high, and most of time, the fan is inactive,

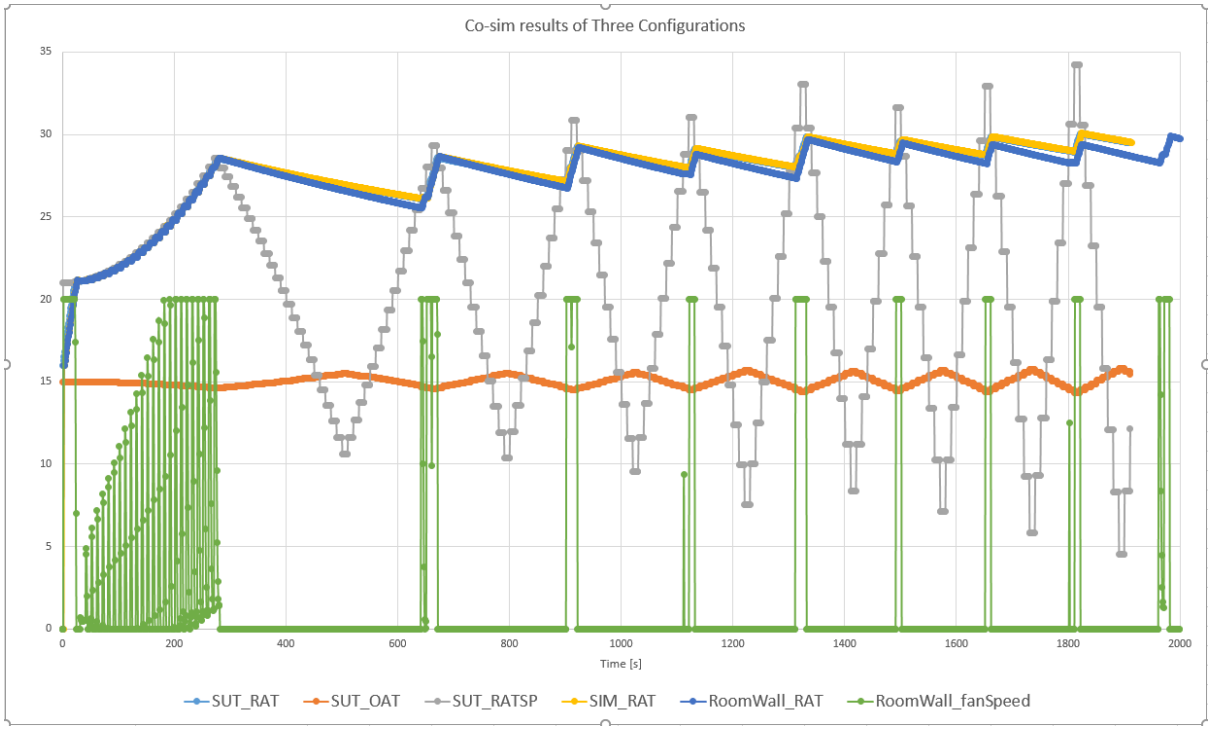


Figure 38: Test Results of Three Configurations

- the output *RAT* from `SUT` almost overlaps with that of `SIM` (because both of them uses the Euler method for the *Room* and *Wall*),
- the output *RAT* from `SUT_RoomWall` is slightly different from those of `SIM` and `SUT` (because the `RoomHeating` 20-Sim model uses the Runge Kutta 4 method).
  - For `SUT_RoomWall`, the room air temperature is changed faster than other two configurations.

4.6.3 Code Generation

The VDM-RT model, **FCUController** can be exported from Overture as a C code FMU, in addition to the tool wrapper FMU as used above. The *FCUController-SourceCode.FMU* included in this pilot is obtained directly from Overture using the "Export Source Code FMU" option. However, this FMU does not contain binaries for co-simulation and so one may use the *FMU Builder* included in the INTO-CPS Application to compile FMUs for Windows, Mac and Linux.

This process has been performed and the resultant FMU is included in the pilot in the FMUs folder; *FCUController-Standalone.FMU*. One example experiment available is to switch this FMU for the tool wrapper version – *FCUController.FMU* – and compare results.

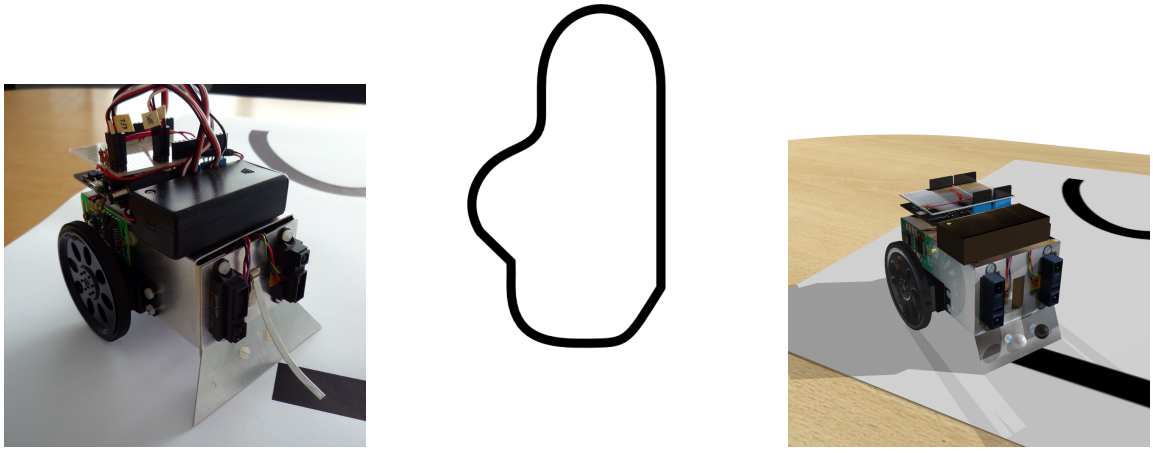
5 Line-following Robot

5.1 Example Description

This example, originally developed in the DESTECS project and presented in [IPG+12]. The model simulates a robot that can follow a line painted on the ground. The line contrasts from the background and the robot uses a number of sensors to detect light and dark areas on the ground. The robot has two wheels, each powered by individual motors to enable the robot to make controlled changes in direction. The number and position of the sensors may be configured in the model. A controller takes input from the sensors and encoders from the wheels to make outputs to the motors.

Figure 39 provides an overview of different aspects of the example: the real robot; an example path the robot will follow; and a 3D representation in 20-sim.

The robot moves through a number of phases as it follows a line. At the start of each line is a specific pattern that will be known in advance. Once a genuine line is detected on the ground, the robot follows it until it detects that the end of the line has been reached, when it should go to an idle state.



(a) A line-following robot (b) A line-follow path (c) 3D representation of the line-following robot

Figure 39: The line-following robot

5.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-line_follower_robot` in the *master* branch. There are several subfolders for the various elements: DSEs - contains various work in progress DSE scripts to alter CT and DE parameters; `FMU` – contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations – with 3D and non-3D options, and also with

and without the use of replicated FMUs; `SysML` – contains the SysML models defined for the study; `resources` – various images for the purposes of the readme file; and `userMetricScripts` – contains files for DSE analysis.

The `case-study_line_follower_robot` folder can be opened in the INTO-CPS application to run the various co-simulations as detailed in this document. To run a simulation, expand one of the multi-models and click 'Simulate' for an experiment.

5.3 INTO SysML profile

Non replicated sensors

The multi-model architecture, defined in the INTO-CPS SysML profile, shows that the *Robot* system is comprised of up to 5 components, as shown in the Architecture Structure Diagram in Figure 40. This comprises the following components: *Body*, *Sensor1* and *Sensor2* physical components, a *Controller* cyber component and a *3DVisualisation* visualisation component.

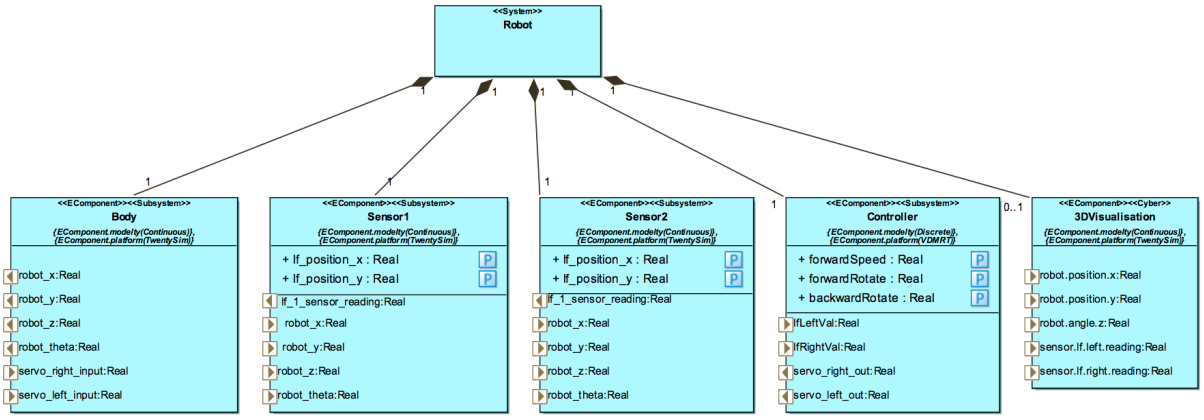


Figure 40: The line-following robot Architecture Structure Diagram

Two Connection Diagrams are defined. The first, in Figure 41, shows connections only between the *Controller*, *Body*, *Sensor1* and *Sensor2* component instances. Broadly speaking: the *Controller* receives sensor readings from both *Sensor1* and *Sensor2* components; the *Controller* in turn sends servo commands to the *Body* component; and finally the *Body* sends the robot position to both sensor components.

Figure 42 shows the alternative CD in which the *3DVisualisation* component is used. In this diagram, the *3DVisualisation* component receives data from the *Body* on the robot position, and the sensor readings from the two sensors. Unlike other examples using the visualisation component type, no additional internal data is required to be exposed by the existing components.

Replicated sensors

An alternative architecture has also been defined in which we use replication offered by 20-sim and OpenModelica FMUs. The ASD in Figure 43 demonstrates the use of two

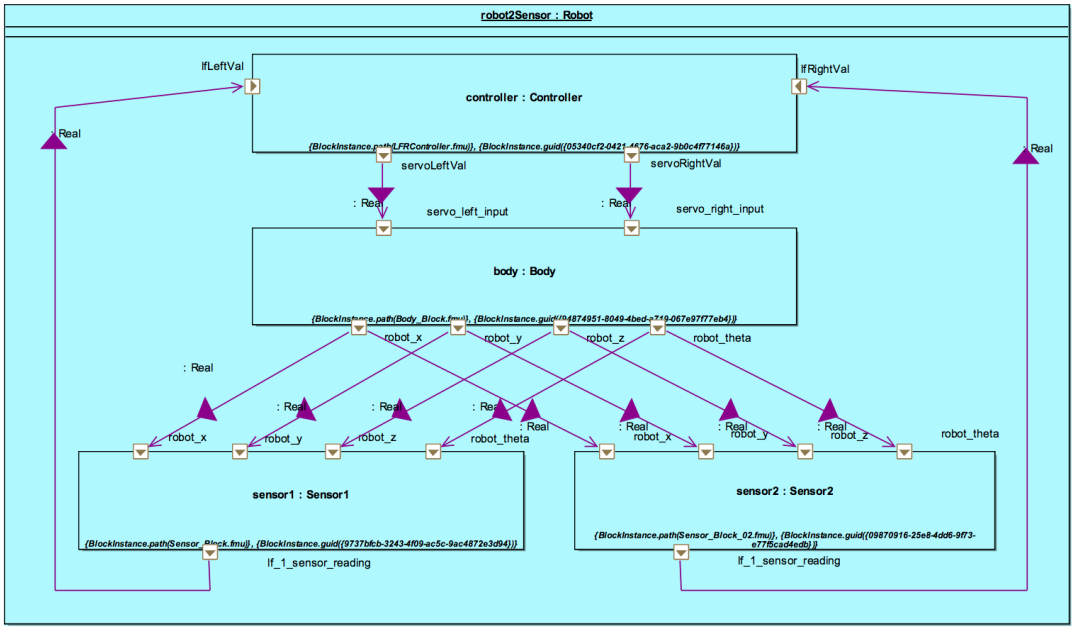


Figure 41: The line-following robot Connections Diagram

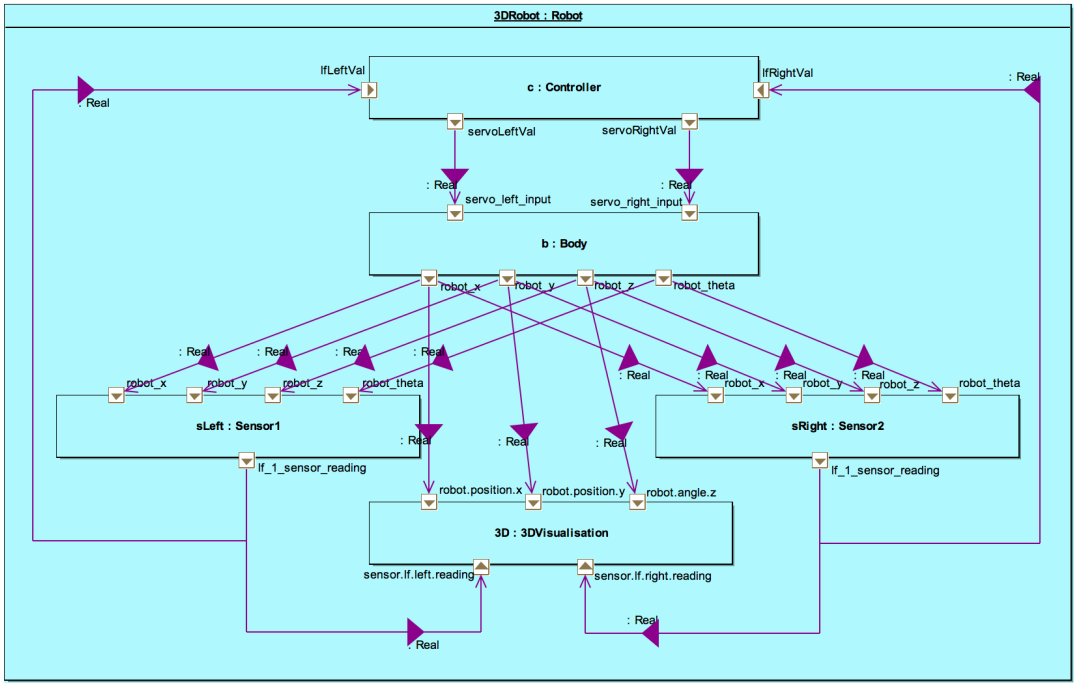


Figure 42: The line-following robot Connections Diagram

instances of the same *Sensor* component type.

The CDs for this SysML model are similar to the non-replicated sensor model. The only difference is the change of sensor types for the two sensor instances – this is shown in Figures 44 and 45.

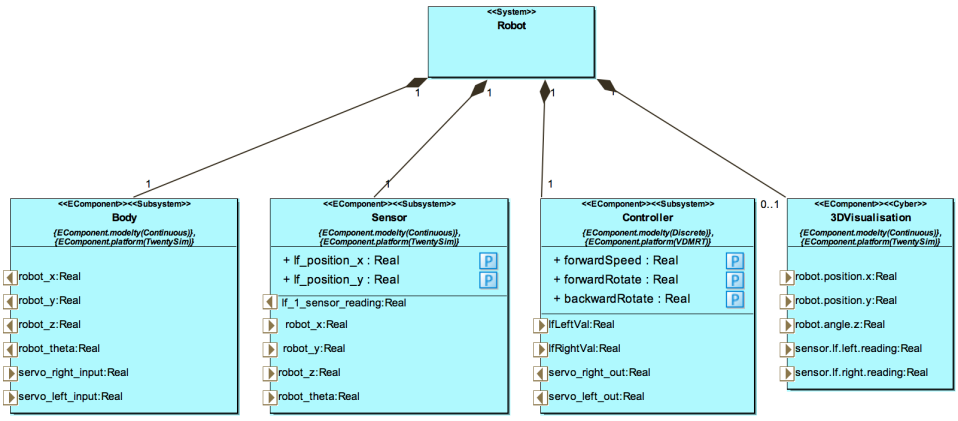


Figure 43: The line-following robot Architecture Structure Diagram with replicated sensors

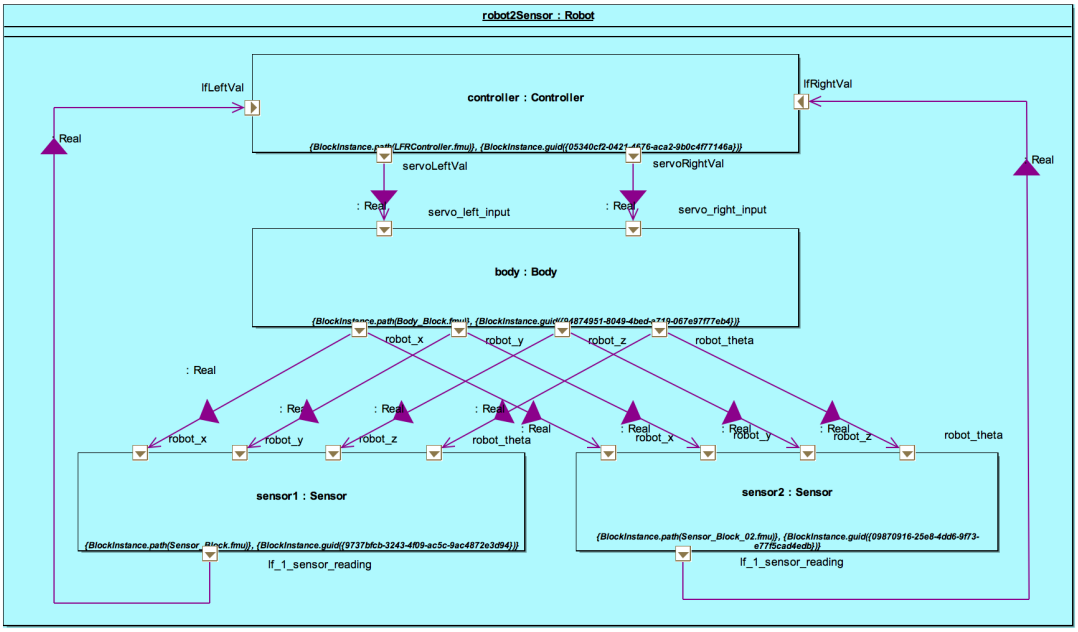


Figure 44: The line-following robot Connections Diagram

5.4 Multi-model

5.4.1 Models

Based upon the two SysML models, we define four different simulation models: a 20-sim *Body* model; a VDM-RT *Controller* model; a 20-sim *Sensor* models; and one OpenModellica *Sensor* model.

Body To define the 20-sim Body subsystem, Figure 46, we first define a top-level decomposition with a *Body_Block* and a block to represent the body's *Environment*.

Decomposing the *Body_Block* further, the 20-sim model is defined as in Figure 47. Blocks are defined for servos, encoders, wheels, the battery and the body itself. A collection of input and output ports are defined: ports to output the robot position (*robot_x*, *robot_y*, *robot_z* and *robot_theta*); ports to output wheel rotation values

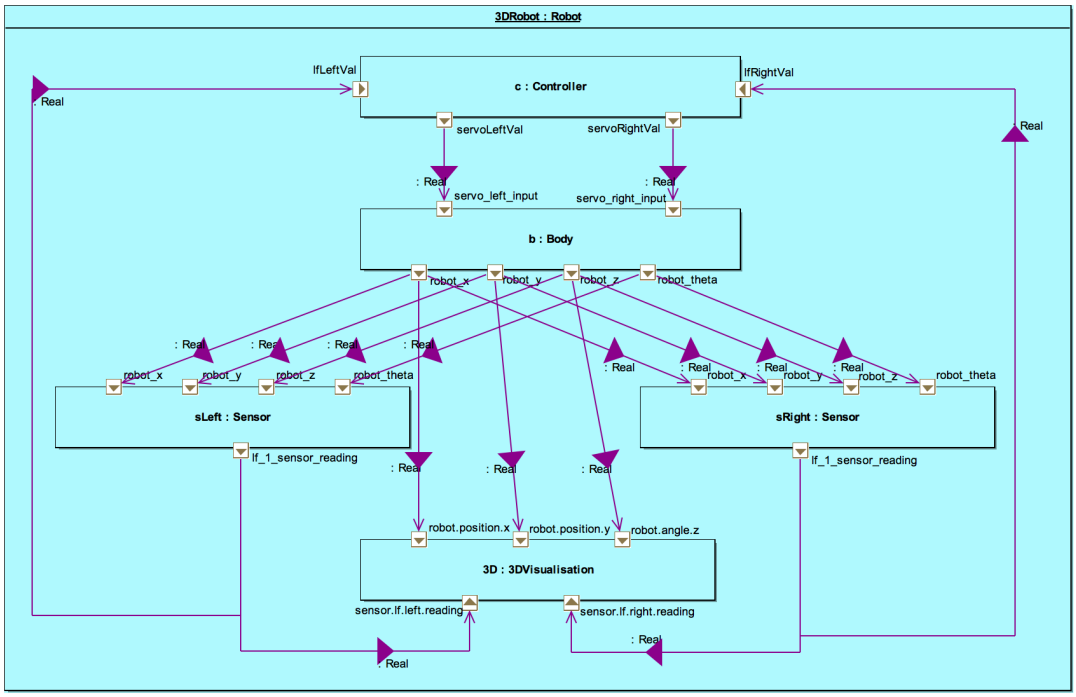


Figure 45: The line-following robot Connections Diagram

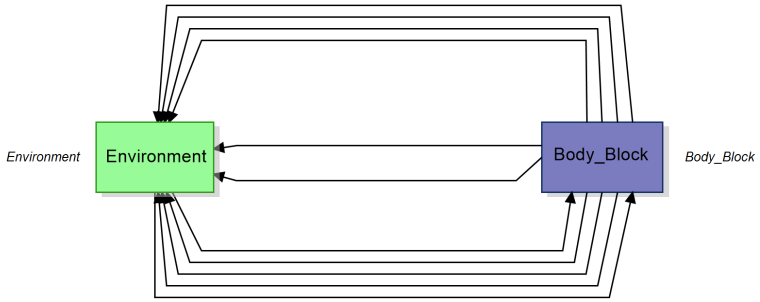


Figure 46: Top-level 20-sim model of the line-following robot *Body*

(*wheel_left_rotation* and *wheel_right_rotation*); a port to output the battery usage (*total_energy_used*); and ports for inputting servo power values (*servo_left_input* and *servo_right_input*).

20-sim_Sensor The *Sensor_Block* is shown in Figure 48. For the non-replicated version, we change the names of the *Sensor_Block* to generate different FMUs.

Decomposing the *Sensor_Block*, we see the internal elements of the sensor – shown in Figure 49. The sensor receives the robot position from its environment, calculating its position in the world using the *line_follow_x* and *line_follow_y* design parameters. This position information is passed to the *map* block, which takes a sample of values and passes a raw reading back to the sensors. The sensors then convert this to an 8-bit value, taking into account realistic behaviours: ambient light levels, a delayed response to changes, and A/D conversion noise. The final sensor reading is output on the *lf_1_sensor_reading* port.

OM_Sensor The OpenModelica version of the sensor is provided in the *LineFollower*

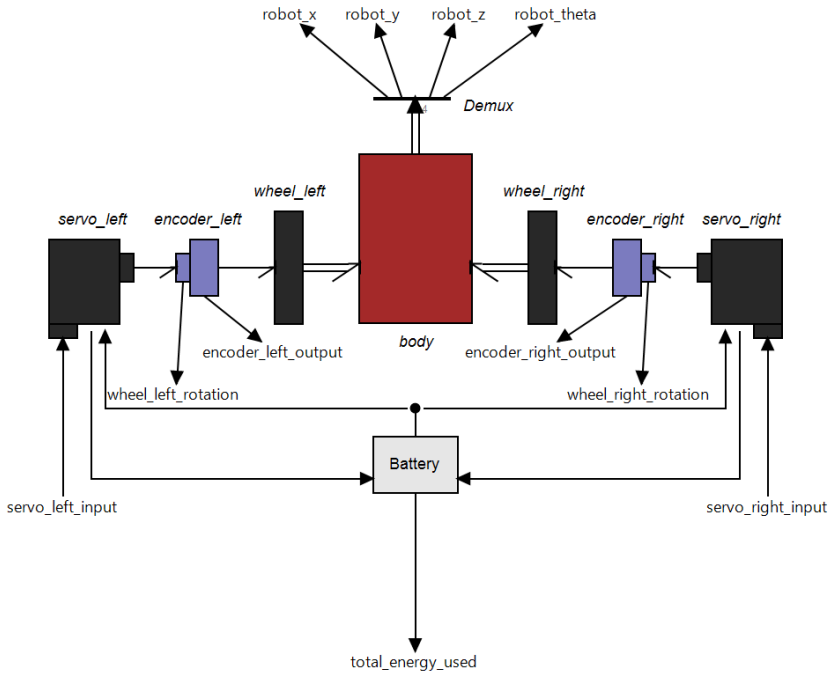


Figure 47: 20-sim model of the line-following robot *Body*

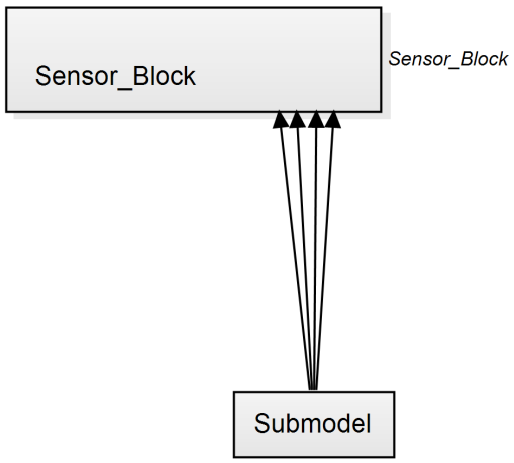


Figure 48: Top-level 20-sim model of the line-following robot *Sensor*

package. The *SensorBlock1.mo* element, shown in Figure 50 corresponds to the 20-sim model shown in Figure 49. The model has the same interface, with internal elements for reflectivity, ambient light, and A/D conversion noise.

Controller The VDM-RT controller model is conceptually unchanged from the original Crescendo controller. The architecture of this model is in Figure 51. The *Controller* model comprises a *System* class which contains a *HardwareInterface* instance which contains references to the inputs, outputs and design parameters. The *System* class also contains a *Controller* class which makes the control decisions. The decisions are based upon sensor readings obtained from two instances of the *RobotSensor* class, and decisions are sent to the two *RobotServo* instances. In this model, a simple algorithm is used: when both sensors see a black line the robot moves forward

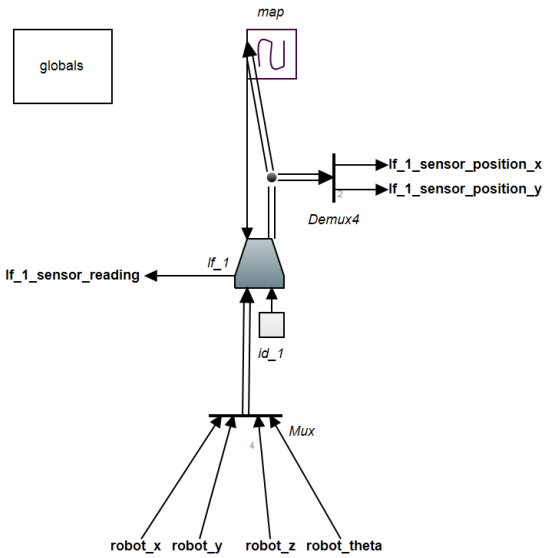
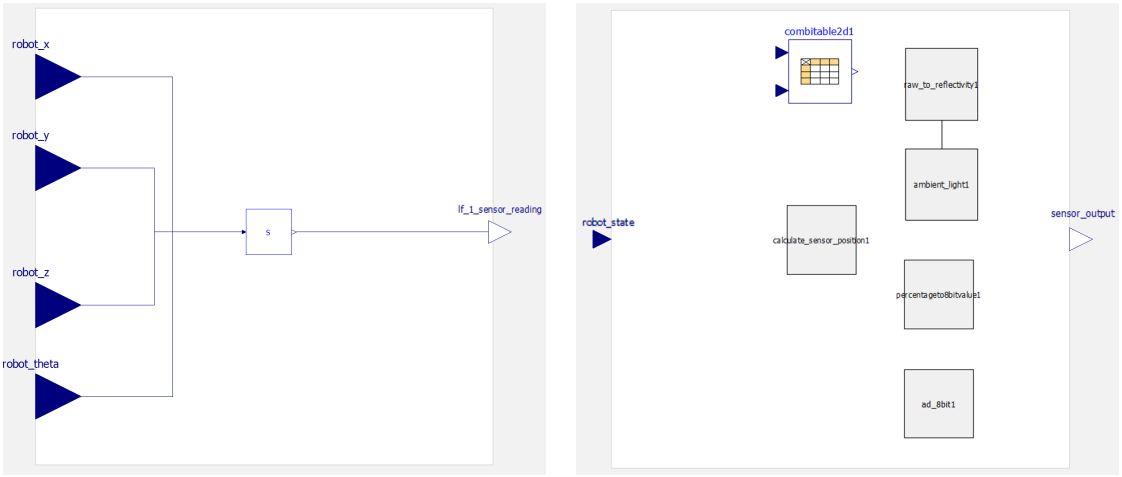


Figure 49: 20-sim model of the line-following robot *Sensor*



(a) Top-level view (b) Low-level view

Figure 50: OpenModelica model of the line-following robot *Sensor*

(both servos are set to the same value), when only the right sensor sees the black line the robot moves left – and vice-versa.

5.4.2 Configuration

There are several connections between the models in the multi-model.

The first collection of connections is between the Body 20-sim model and the Controller VDM-RT model. In this collection, there are two connections corresponding to signals for the actuators that power the motors for the wheels:

- from the *Controller* `servoLeftVal` variable of type real to the `servo_left_input` port of the *Body*; and

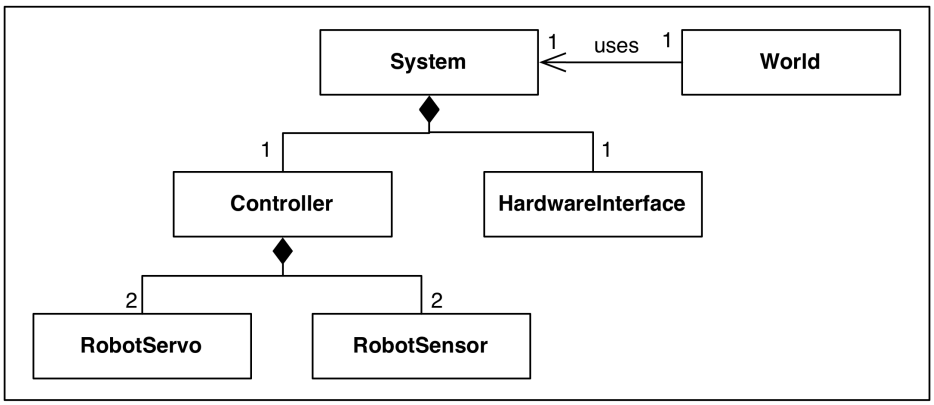


Figure 51: UML representation of line-following robot *Controller* model

- from the *Controller* `servoRightVal` variable of type real to the `servo_right_input` port of the *Body*.

The second collection of connections is between the Sensor models and the Controller VDM-RT model. For each sensor there is one connection to the controller to represent inputs from line-following sensors that can detect areas of light and dark on the ground. Therefore for a two-sensor model there are two connections:

- from the *Sensor1*[7] `lf_1_sensor_reading` port to the *Controller* `lfLeftVal` variable; and
- from the *Sensor2*[8] `lf_1_sensor_reading` port to the *Controller* `lfRightVal` variable.

A third collection of connections exist between the body and the sensors related to the robot position:

- from the *Body* `robot_x` port to the *Sensor* `robot_x` port;
- from the *Body* `robot_y` port to the *Sensor* `robot_y` port;
- from the *Body* `robot_z` port to the *Sensor* `robot_z` port; and
- from the *Body* `robot_theta` port to the *Sensor* `robot_theta` port.

A collection of multi-models is provided in the study for combinations of 20-sim and OpenModelica models.

Several shared design parameters are present also: the separation of the line-following sensors from the centre line, in metres (*line_follow_x*); and the distance forward of the line-following sensors from the centre of the robot, in metres (*line_follow_y*). In addition, design parameters are set for the controller: the *forwardSpeed* and values for rotation – *forwardRotate* and *backwardRotate*.

[7]Sensor1 is an instance of either the non-replicated *Sensor_Block1* or the replicated *Sensor_Block*
[8]Sensor2 is an instance of either the non-replicated *Sensor_Block2* or the replicated *Sensor_Block*

5.5 Co-simulation

For all these multi-models, co-simulations require approximately 25-30 seconds of simulation to traverse the full map, using a fixed step size of 0.01 seconds. The example has co-simulation set ups for each multi-model and the non-3D models have live stream enabled for the sensed values from sensor1 and sensor2.

5.6 Analyses and Experiments

Below we detail some useful experiments to demonstrate features of the INTO-CPS tool chain.

5.6.1 Change FMUs/parameters

The case study has several Sensor FMUs. In the multi-model configuration it is possible to swap the FMU allocated to each sensor instance of the multi-model. We can therefore compare the results of co-simulation using a combination of 20-sim sensors (the replicated *Sensor_Block.fmu*, or *Sensor_Block1.fmu* and *Sensor_Block2.fmu*) and OpenModelica sensors (replicated, or some combination of *LineFollower_Examples_SensorBlock1.fmu* and *LineFollower_Examples_SensorBlock2.fmu*).

In addition, there are parameters defined for the two sensors (an x and y position `lf_position_x` and `lf_position_y`), and of the controller (forward and rotational speeds *forwardSpeed*, *forwardRotate* and *backwardRotate*). Experiments may be carried out by defining different values for these to model different placement of the sensors on the robot and altering the robot speeds,.

5.6.2 Simulations due to previous results

Simulations can be replayed using different design parameter values to change the position of the robot sensors. Changing these parameters amounts to changing the design of the robot - some value pairs will produce robots which perform in some way better (e.g. have a faster lap time) and others will result in robots which cannot follow the line.

5.6.3 Design Space Exploration

Several Design Space Exploration experiments have been included in this pilot. They are described in more detail in Deliverable D5.3e [Gam17], and we give an overview of the different experiments here.

lfr-2sensorPositions: This experiment uses four design parameters, but only varies one. The `lf_position_y` of *sensor1* may be either 0.07 or 0.13. Four objective scripts are used: *meanSpeed*, *lapTime*, *maxCrossTrackError* and *meanCrossTrackError*. The Pareto ranking only uses the *lapTime* and *meanCrossTrackError* objectives; these determine the time taken for the robot to perform one lap of the map, and also the mean error the robot makes when following the line.

lfr-8controllerValues: This experiment uses and varies three parameters. These parameters are all on the cyber-side of the multi-model – effecting the robot speeds set by the controller. For each (`forwardSpeed`, `forwardRotate` and `backwardRotate`), two possible values are defined - giving a design space of 8 designs. The same four objective scripts are used as above with the same Pareto ranking.

lfr-16sensorPositionsConstrained: This experiment is a more complex version of the **lfr-2sensorPositions** experiment in that it varies all four design parameters – the `lf_position_x` and `lf_position_y` coordinates of both *sensor1* and *sensor2*. Two possible values are defined for each parameter – giving a 16-design space. A constraint is defined for the parameters, which limits this design space to include only those designs which have the same y coordinate and the same absolute x coordinate. The same four objective scripts are used as above with the same Pareto ranking.

lfr-216controllerValues: This expands the **lfr-8controllerValues** experiment, providing 6 speed values for the Controller parameters (`forwardSpeed`, `forwardRotate` and `backwardRotate`), producing a design space of 216 designs. The same four objective scripts are used as above with the same Pareto ranking.

lfr-2187ControllerAndSensors: The final experiment combines DSE on both DE and CT models. Providing an insight into the possibility to trade-off between development effort on the cyber or physical side. In this experiment there are 3 possible positions for each of the the `lf_position_x` and `lf_position_y` coordinates of both *sensor1* and *sensor2*, and also 3 values for the Controller parameters (`forwardSpeed`, `forwardRotate` and `backwardRotate`). This produces a design space of 2187 designs. The same four objective scripts are used as above with the same Pareto ranking.

5.6.4 Test Automation and Model Checking

In this pilot study, we apply test automation only to the controller, instead of the system in the FCU study. The continuous body and sensors are rather complex in order to be modelled in a test model in SysML in Modelio by state machine diagrams. Therefore, only the discrete part *Controller* will be tested. In addition, the *Controller* is a simplified version which is similar to the VDM model in *LFRController.fmu*. In this model, the parameters $forwardSpeed$, $forwardRotate$ and $backwardRotate$ are fixed to 4.0, 5.0 and 1.0 respectively. We implement a SUT manually in C. Then model checking is applied to check a couple of properties of the test model, and finally test automation to test whether the SUT is a correct implementation of the test model.

Test Model The overall architecture and connection diagrams of this test model are omitted for brevity. The *SystemUnderTest* includes only one block LFR_CTRL.

Inputs and Outputs The *SystemUnderTest* receives the following inputs (stimuli) from the *TestEnvironment*:

- $sensorLeftVal$

- $sensorRightVal$

The *SystemUnderTest* provides the following observable outputs to the *TestEnvironment*:

- $servoLeft$
- $servoRight$

Constant Variables and Local Variables Preliminarily, we intend to define three constant variables $forwardSpeed$, $forwardRotate$ and $backwardRotate$ of which the access model property is set as `Read`. Then in the state machine diagram of LFR_CTRL, we use these variables to refer to constants. However, these constant variables are not supported in RT-Tester. Their initial values are all set to 0 when parsing the test model. Alternatively, we can initialise them to constants in the first state just after the `Init` state, or just use hard-coded constants in the test model (the way used in this study).

Another two local variables $preServoLeft$ and $preServoRight$ are declared to store previously output values of $servoLeft$ and $servoRight$ separately. And their access model property is set to `Read/Write`.

State Machine Diagram The state machine diagram of FCU_CTRL is given in Figure 52. It is worth noting that

- This model is based on a very basic variation of the controller and it only provides four possible outputs.
- The texts in blue actually are not a part of the model and they are annotations only to illustrate associated expression for each guarded transition (because the expressions cannot be seen from the diagram directly).
- When both $sensorLeftVal$ and $sensorRightVal$ are larger than or equal to 150, our model outputs the previously stored $preServoLeft$ and $preServoRight$. It means the outputs have not changed.

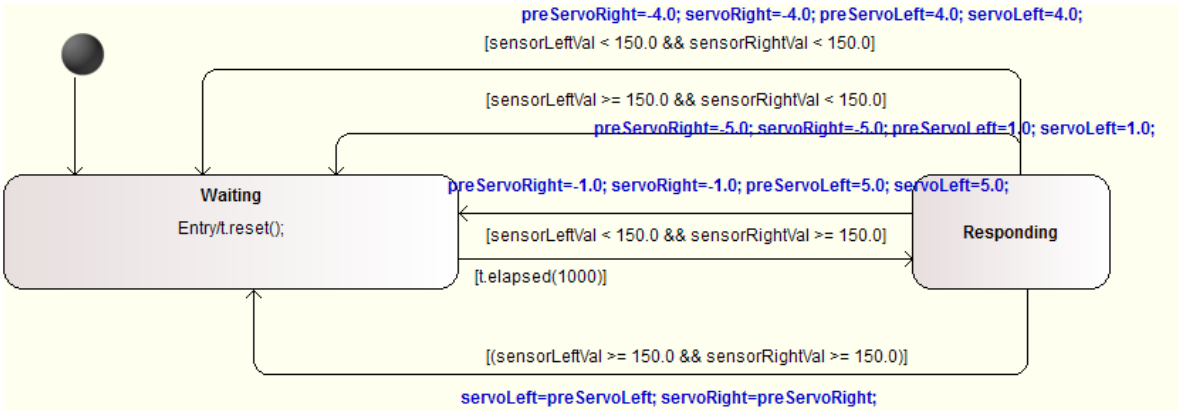


Figure 52: State Machine Diagram of LFR Controller

A Manual Implementation of SUT in C The source code is listed as follows.

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include "lfr_ctrl.h"

#define FORWARDSPEED    4.0f
#define FORWARDROTATE   5.0f
#define BACKWARDROTATE  1.0f

#define SENSORVAL       150.0f

#define _ms( t ) ( (t) * 1000 )
float preServoRight = 0;
float preServoLeft = 0;

VSTimer_t t;

void reset( VSTimer_t* timer)
{
  struct timeval now;
  ti_gettimeofday( &now, NULL );
  *timer = now.tv_sec * 1000000 + now.tv_usec;
}

BOOLEAN elapsed( VSTimer_t* timer, long usec )
{
  struct timeval now;
  long long usec_now;

  ti_gettimeofday( &now, NULL );
  usec_now = now.tv_sec * 1000000 + now.tv_usec;
  return ( ( usec_now - (*timer) ) >= usec );
}

int firstCall = 1;

/** Initialize SUT */
void sut_init()
{
    preServoLeft    = 0;
    preServoRight   = 0;
    reset(&t);
}

/** Run SUT (one step) */
void sut_run(float sensorLeftVal, float sensorRightVal,
    float* servoLeft, float* servoRight)
{
    if(firstCall)
    {
        if(!elapsed(&t, _ms(1002)))
        {
            *servoRight = preServoRight;
            *servoLeft = preServoLeft;
            return;
        }
    }
```

```
    else
    {
        if(!elapsed(&t, _ms(1000)))
        {
            *servoRight = preServoRight;
            *servoLeft = preServoLeft;
            return;
        }
    }

    firstCall = 0;
    reset(&t);

    if(sensorLeftVal < SENSORVAL)
    {
        if(sensorRightVal < SENSORVAL)
        {
            preServoRight = *servoRight = -FORWARDSPEED;
            preServoLeft = *servoLeft = FORWARDSPEED;
        }
        else //if(sensorRightVal >= SENSORVAL)
        {
            preServoRight = *servoRight = -BACKWARDROTATE;
            preServoLeft = *servoLeft = FORWARDROTATE;
        }
    }
    else // if(sensorLeftVal >= SENSORVAL)
    {
        if(sensorRightVal < SENSORVAL)
        {
            preServoRight = *servoRight = -FORWARDROTATE;
            preServoLeft = *servoLeft = BACKWARDROTATE;
        }
        else // if(sensorRightVal >= SENSORVAL)
        {
            *servoRight = preServoRight;
            *servoLeft = preServoLeft;
        }
    }

    return;
}
```

We define the *reset* and *elapsed* functions to reset the time variable t and check where the specified time has elapsed or not. The controller is mainly idle (it could not process inputs but just return previously stored outputs). But every one second, it starts to process inputs and return corresponding outputs. It is worth noting that in the first call of the *sut_run*, we have additional two ms delay. The reason is given later.

Model Checking Model checking can be applied to the test model to check if it satisfies the properties. In order to use bounded model checking in the INTO-CPS app, we set the bounded steps "BMC steps" to 50. The properties below are checked to hold in 50 steps.

P0 Livelock property by the **Check Mode** function on RT-Tester.

```
Check static model semantics ...done.
- IMR.SystemUnderTest.lfrCTRL...  [PASS]
- IMR.TestEnvironment... [PASS]

================================================================
|              Livelock report                                 |
================================================================
IMR.SystemUnderTest.lfrCTRL.....................................[PASS]
IMR.TestEnvironment.............................................[PASS]
```

P1 Check all outputs are always within their valid range.

$$Globally\left(\left[\begin{array}{l} -5 \leq IMR.servoLeft \wedge IMR.servoLeft \leq 5 \wedge \\ -5 \leq IMR.servoRight \wedge IMR.servoRight \leq 5 \end{array}\right]\right)$$

P2 The value of $servoLeft$ is always larger than or equal to 0 but that of $servoRight$ is always less than or equal to 0.

$$Globally\left(\begin{array}{l} [\_timeTick == 0] \vee \\ [\_timeTick > 0 \wedge IMR.servoLeft \geq 0 \wedge IMR.servoRight \leq 0] \end{array}\right)$$

P3 If we use constant variables rather than hard-coded constants, we can check they will never be changed.

$$Globally\left(\begin{array}{l} [\_timeTick == 0] \vee \\ [\_timeTick > 0 \wedge forwardSpeed == 4.0] \end{array}\right)$$

Test Results

User Defined Test Cases We use user defined test cases by LTL formulas in RT-Tester to define a test goal that all combinations of inputs shall be covered. This goal is illustrated in Figure 53. Then the solver will generate a test data generation report that includes test goals, explicitly and implicitly covered test coverage cases, signal configurations, and test stimulations and expected behaviour. This test covers all basic control state coverage test cases (3), all transition coverage test cases (5), and 4 MCDC coverage test cases in 12. The generated test input sequence is shown in Figure 54.

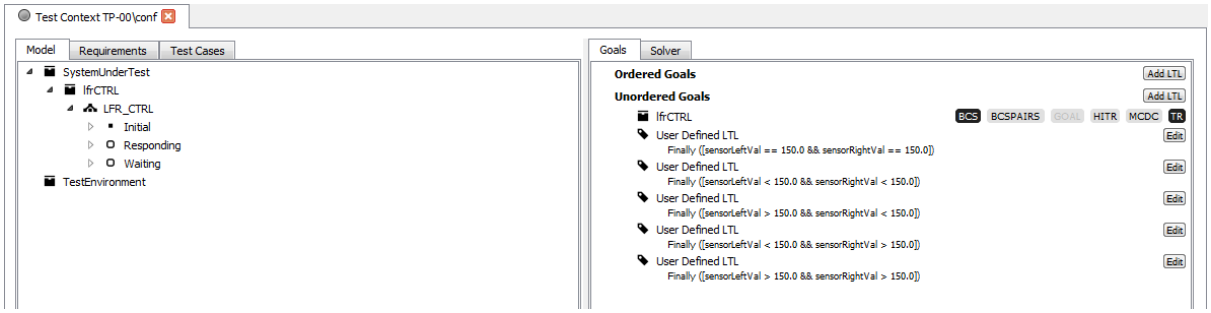


Figure 53: Test Goal Configuration of LFR

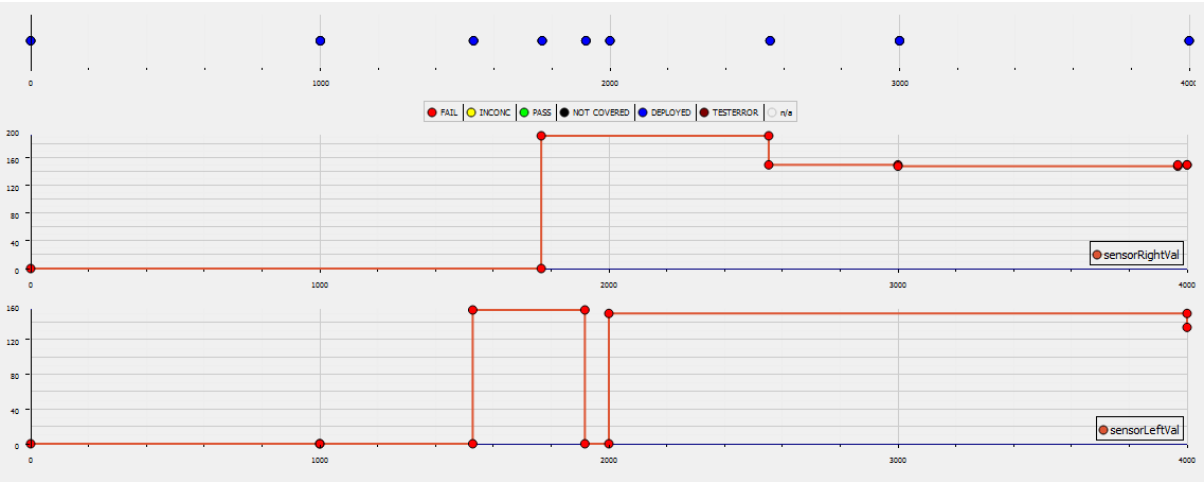


Figure 54: Test Input Sequence of LFR

Firstly, the test result of `TP` against `Simulation` is displayed in Figure 55. All test cases should be `PASS` or `INCONCLUSIVE` because both `TP` and `Simulation` are test procedures generated from the exactly same test mode. However, three `FAIL`s are seen from the figure. This is because there is a very small delay in arrival time of inputs between `TP` and `Simulation`. For instance, $sensorLeftVal$ is changed to 150 from 0 at 2000 ms (as shown in Figure 54) which is the time for `TP` to output it to `Simulation` as well as check of expected behaviour. But the actual time when `Simulation` gets the latest value 150 would be later (provided the Co-Simulation step is 1ms, and then there is 1ms delay). Finally, at 2000 ms, `TP` uses 150 to calculate the expected behaviour but `Simulation` still uses the old value 0 for $sensorLeftVal$ to give its output, which leads to mismatch of actual outputs with expected outputs.

Test Case Coverage — Component RTT_TestProcedures/TP-00

Test Case	Baseline	Verdict	Reference (Origin)
TC-INTO-CPS-Demo-BCS-0002	@BASELINE@	FAIL	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000002007]
TC-INTO-CPS-Demo-TR-0001	@BASELINE@	INCONCLUSIVE	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000001000]
TC-INTO-CPS-Demo-BCS-0003	@BASELINE@	INCONCLUSIVE	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000001000]
TC-INTO-CPS-Demo-TR-0002	@BASELINE@	PASS	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000001001]
TC-INTO-CPS-Demo-TR-0005	@BASELINE@	FAIL	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000002007]
TC-INTO-CPS-Demo-TR-0004	@BASELINE@	FAIL	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000003006]
TC-INTO-CPS-Demo-TR-0003	@BASELINE@	PASS	testdata\am__ora_lfrCTRL_20170807230959.log [TM 00000004000]

Figure 55: Test Case Summary (TP vs. SIM)

In order to correct it, we can put additional 2 ms offset manually in SUT. It is shown in the source code. Eventually, the test result of `TP` against `SUT` is displayed in Figure 56 in which all test cases are `PASS` or `INCONCLUSIVE`.

Test Case Coverage — Component RTT_TestProcedures/TP-00

Test Case	Baseline	Verdict	Reference (Origin)
TC-INTO-CPS-Demo-BCS-0002	@BASELINE@	PASS	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000000000]
TC-INTO-CPS-Demo-TR-0001	@BASELINE@	INCONCLUSIVE	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000001000]
TC-INTO-CPS-Demo-BCS-0003	@BASELINE@	INCONCLUSIVE	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000001000]
TC-INTO-CPS-Demo-TR-0002	@BASELINE@	PASS	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000001003]
TC-INTO-CPS-Demo-TR-0005	@BASELINE@	INCONCLUSIVE	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000002000]
TC-INTO-CPS-Demo-TR-0004	@BASELINE@	PASS	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000003003]
TC-INTO-CPS-Demo-TR-0003	@BASELINE@	PASS	testdata\am__ora_lfrCTRL_20170807130238.log [TM 000000004003]

Figure 56: Test Case Summary (TP vs. SUT)

5.6.5 Code Generation

The VDM-RT model, **LFRController**, can be exported from Overture as a C code FMU, in addition to the tool wrapper FMU as used above. The *LFRController-SourceCode.fmu* included in this pilot is obtained directly from Overture using the "Export Source Code FMU" option. However, this FMU does not contain binaries for co-simulation and so one may use the *FMU Builder* included in the INTO-CPS Application to compile FMUs for Windows, Mac and Linux.

This process has been performed and the resultant FMU is included in the pilot in the FMUs folder; *LFRController-Standalone.fmu*. One example experiment available is to switch this FMU for the tool wrapper version – *LFRController.fmu* – and compare results.

6 Turn Indicator

6.1 Example Description

The turn indicator model discussed here is an adaption of a model originally designed with an industrial partner from the automotive domain[9]. The model specifies the behaviour of a turn indication controller, which essentially supports left and right flashing as well as emergency flashing. The functionality is modelled using three inputs (the voltage, the control lever and the emergency flash button) and two outputs (the states of the left and right turn indication lights, respectively). The model can then be used to automatically generate test cases for a system that shall implement the specified behaviour. In addition, desired safety properties of the system can also be verified using model checking. Both these activities are performed using the RT-Tester Model-Based Test Case Generator (RTT-MBT) [Ver15] and are described in more detail in Deliverable D5.3a [GMB17] and Deliverable D5.3c [BH17], respectively.

A key feature of this example is that it combines several features which are important for effective modelling of system specifications using SysML state charts: It uses variables of different types (voltage is real-valued, the other ones are integral), it uses hierarchical state machines and concurrent components.

6.2 Usage

The example is available at `https://github.com/INTO-CPS-Association/example-turn_indication` and can be downloaded as an example project directly from within the INTO-CPS application. After that, the example can be used for test automation and model checking activities.

In addition the *VSI tools* release bundle installs a pre-configured RT-Tester project in the directory `C:\Users\<USER>\RTT-Prj\turn-ind\`. The sub-folder `sut\` contains a C implementation of the system under test. The associated FMU `RTT\_TestProcedures\SUT\TurnIndicationController_sut.fmu` can be run in co-simulated test run against a generated test driver.

6.3 SysML

The model has been developed in Modelio by means of hierarchic parallel state-charts. Furthermore, architecture diagrams are used to structure components and ports, and connections are used to express data flow between parallel components.

Figure 57 depicts the structure of the overall `System` which is decomposed in a `SystemUnderTest` and a `TestEnvironment` component. The `SystemUnderTest` encompasses the desired behaviour of the system under test and has therefore been annotated with the *SUT* stereotype. The `TestEnvironment` represents the operational environment to the system under test and is annotated with the *TE* stereotype.

[9]The detailed model is described in [PVL11].

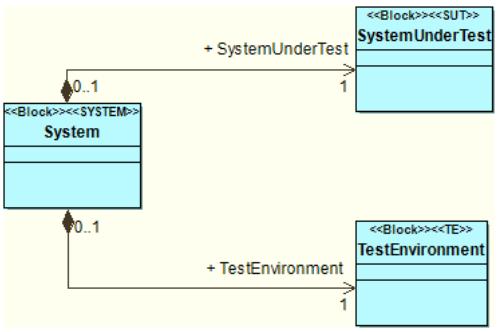


Figure 57: Top-level architecture diagram of the turn indicator model.

The system under test receives inputs from the environment and provides outputs. For both of these, data flow interfaces specify the involved variables and have been associated with the stereotypes *SUT2TE* and *TE2SUT*, respectively. The system under test receives the following inputs from the environment:

- `TurnIndLvr`: The position of the turn indicator lever, which can either be neutral, left flashing, or right flashing.
- `EmerSwitch`: The on/off status of the emergency flashing switch.
- `voltage`: The voltage of the car's battery.

The `SystemUnderTest` produces the following observable outputs:

- `LampsLeft`: The of/off status of the indication lights on the left side.
- `LampsRight`: The of/off status of the indication lights on the right side.

The connection diagram in Figure 58 connects the system under test with the test environment using the described interfaces.

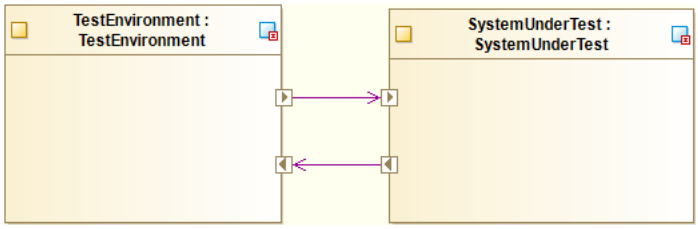


Figure 58: Top-level connection diagram of the turn indicator model.

Note, that the correct stereotype annotations of the components are important for test case generation using RTT-MBT.

In this example, the `TestEnvironment` does not constrain the input variables in any way (RTT-MBT automatically ensures that the values assigned during test case generation are within the specified range). The relevant logic is thus implemented in `SystemUnderTest`, which is divided into two hierarchical state charts called `FLASH_CTRL` and `OUTPUT_CTRL` as expressed by the class diagram in Figure 59. The component `FLASH_CTRL` is responsible for deciding whether the left or the right side has to flash depending on the turn-indicator

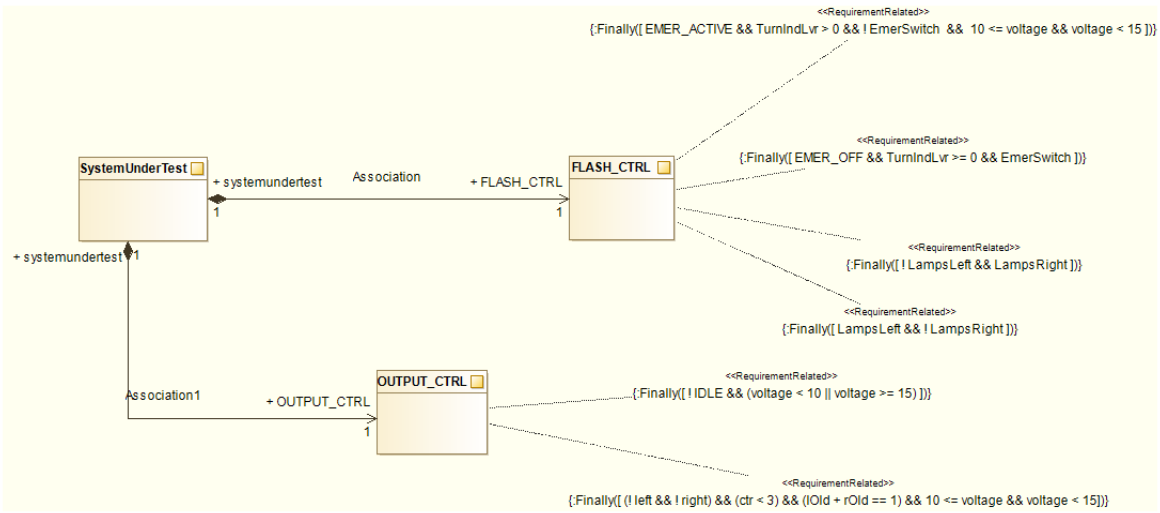


Figure 59: System under test architecture diagram of the turn indicator model.

lever and the emergency switch. This general decision for the two sides is fed into the `OUTPUT_CTRL` which is responsible for periodically turning the lights on and off. This data flow between the two components is expressed by the connection diagram in Figure 60.

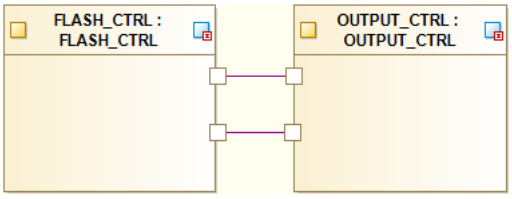


Figure 60: System under test connection diagram of the turn indicator model.

The `FLASH_CTRL` state machine in Figure 61 controls the impact of operating the turn indicator and emergency flashing switch. If the emergency switch is not pressed, the state machine simply enables flashing on a specific side if the turn indicator lever is in the respective position. If the emergency switch is pressed, the composite state in Figure 62 decides whether both sides should flash. Observe, that using the turn indicator lever while the emergency switch is pressed can override emergency flashing. The lamps resume flashing on both sides when the turn indicator level is returned to the neutral position.

`OUTPUT_CTRL` depicted in Figure 63 implements two modes for setting the outputs. It can be in either idle or flashing mode, where the flashing mode itself is implemented as composite state that can switch from off to on and vice versa. It does so in a regular interval if the system has enough power and a lever or the emergency button has been used. The state machine also implements a counter that ensures that left or right flashing is still flashing for three times if the turn indicator level is only operated for a short duration.

Observe that certain states and transitions in the model have been annotated with requirements. For example, the transition from state `TURN_IND_OVERRIDE` → `EMER_ACTIVE` in

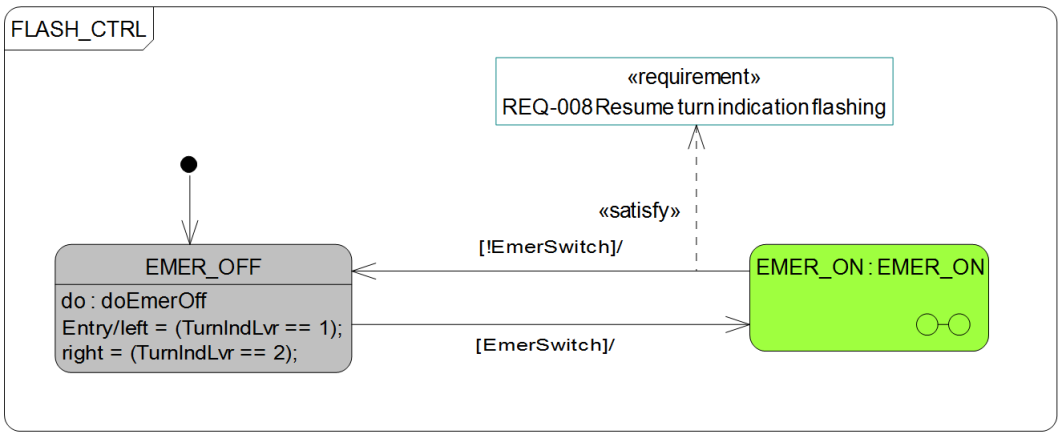


Figure 61: The `FLASH_CTRL` state machine.

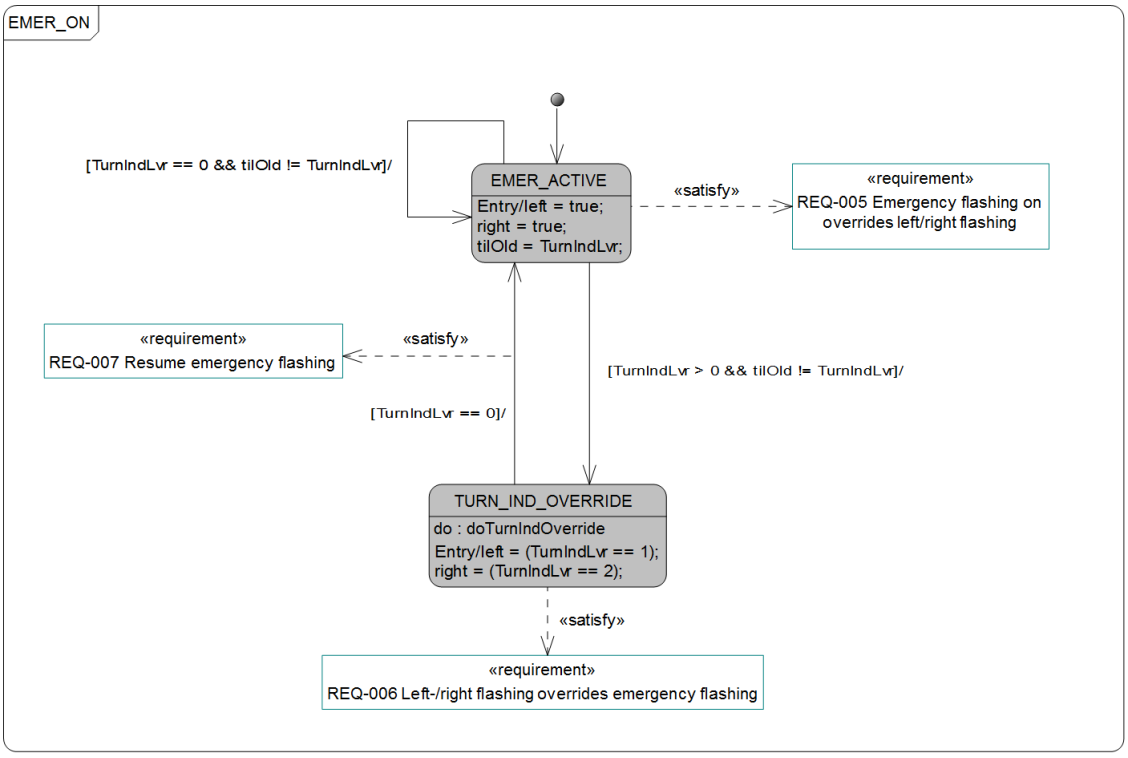


Figure 62: The `EMER_ON` composite state in the `FLASH_CTRL` state machine.

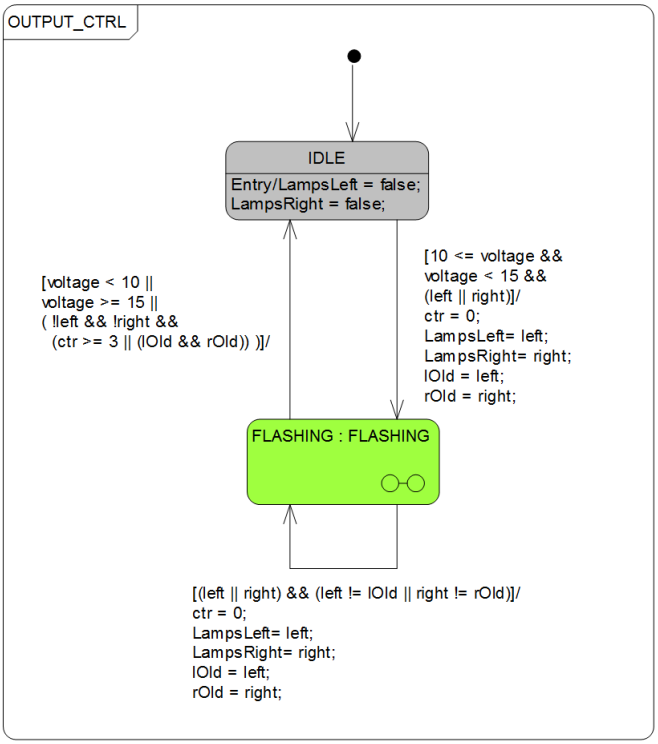


Figure 63: The `OUTPUT_CTRL` state machine.

Figure 62 has been linked to requirement REQ-007 via a *satisfy relation*. Likewise, state `TURN_IND_OVERRIDE` has been linked to requirement REQ-006. Linking requirements in that way specifies that the associated structural elements of the model help to realise the given requirement.

Furthermore, some requirements are attached to classes in conjunction with an LTL formula as can be seen in Figure 59. The LTL formula specifies an abstract execution trace that could serve as a witness to demonstrate that the requirement is fulfilled by an implementation.

6.4 Analyses and Experiments

6.4.1 Test Automation and Model Checking

As mentioned earlier, this pilot can be used to automatically generate test cases for a system that shall implement the specified behaviour. In addition, desired safety properties of the system can also be verified using model checking. Both these activities are performed using the RT-Tester Model-Based Test Case Generator (RTT-MBT) and are described in more detail in Deliverable D5.3a [GMB17] and Deliverable D5.3c [BH17], respectively.

7 Unmanned Aerial Vehicle

This example is no longer maintained or updated. You are welcome to contact the example owner in case of interest. See `https://github.com/INTO-CPS-Association/example-single_uav` for contact details.

7.1 Example Description

This pilot study originates from a master thesis study presented in [GN16]. The study models the physical dynamics as well as the discrete controller of an Unmanned Aerial Vehicle (UAV). Focus on the details of the model contribute to a high model fidelity, enabling the multi-model to be used to compare alternative control algorithms.

Figure 64 shows a 3D model of the UAV and some of its main components, including the `airframe`, which is the main body the UAV, the propulsion system consisting of `rotors`, `motors`, and `electronic speed controllers`, along with the `battery` and the `controller` platform. Additionally, a UAV have a range of different `sensors` and a `telemetry system` used to communicate with a pilot or a ground control center.

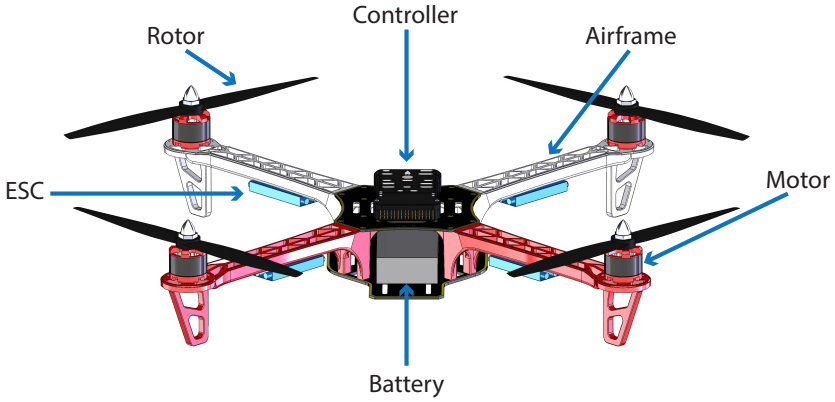


Figure 64: UAV 3D model

7.2 Usage

The example is available at `https://github.com/INTO-CPS-Association/example-single_uav` in the *master* branch. There are several subfolders for the various elements: `FMU` – contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations – with 3D and non-3D options; and `SysML` – contains the SysML model defined for the study.

In the *abstract_intocps* branch, the original discrete event control model have been replaced with an abstract control model in order to enable high-level control prototyping. A prototype of an autonomous vertical waypoint controller is exemplified. The same model is found in the *abstract_crescendo* branch, where DESTECS technology is used instead of INTO-CPS technology. This can be used to compare the two technologies [TN16].

7.3 INTO-CPS SysML profile

The INTO-CPS SysML profile is used to create an Architecture Structure Diagram (ASD) and a Connections Diagram (CD), shown in Figure 65 and Figure 66 respectively. The ASD expresses that the system `UAV` is a composition of a cyber part `ArduPilot`, a physical part `ArduCopter`, and an optional `3D animation`.

`ArduPilot` is a discrete event controller described in VDM-RT. It takes a number of sensor inputs and outputs a control signal for each of the four motors of the UAV. By adjusting the motor setpoints, it is able to make the UAV fly to predefined waypoints, taking into account feedback from sensors.

`ArduCopter` is a model of the physical dynamics of the UAV described in 20-sim. The inputs to the `ArduCopter` model are the four motor setpoints. Based on these, it calculates the angular position described with roll, pitch, and yaw angles, and the spatial position described with a latitude, longitude, and altitude (X,Y,Z), and the velocities and accelerations of the UAV. Additionally, corresponding sensor outputs are simulated for a 3-axis accelerometer, a 3-axis gyroscope and a GPS.

Angular and spatial positions are used by the `3D animation`.

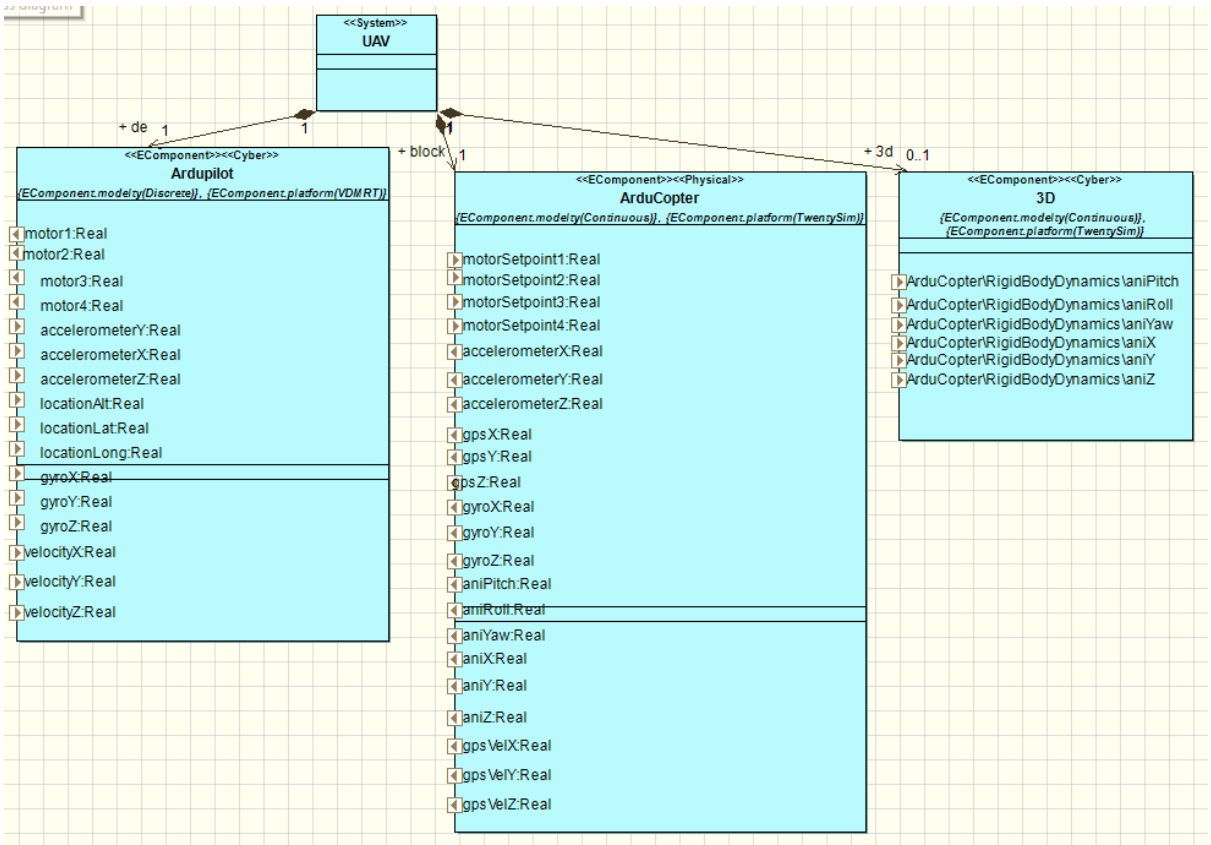


Figure 65: UAV Architecture Structure Diagram

The connection between the constituent models is a one-to-one mapping between `ArduPilot` and `ArduCopter`, with the exception that the `3D animation` is connected to `ArduCopter` as well, as shown in Figure 66.

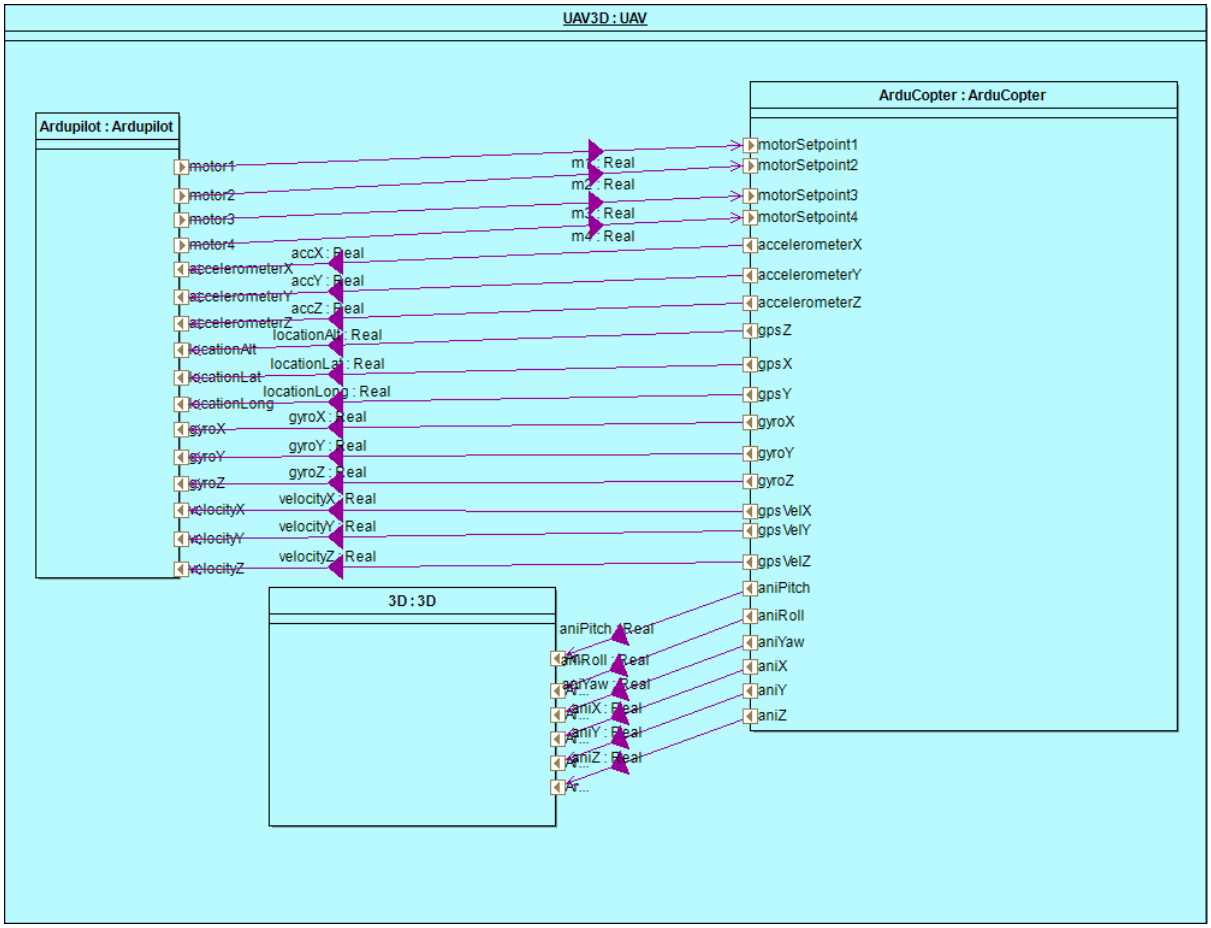


Figure 66: UAV Connections Diagram

7.4 Multi-model

7.4.1 Models

The system comprises a continuous-time (CT) model `ArduCopter` and a discrete event (DE) model `ArduPilot`.

ArduCopter: The physical dynamics of the UAV is described in 20-sim. In Figure 67 an overview of the `Quadcopter` model is shown. It includes the rigid body dynamics of the `airframe`, the aerodynamics of the `rotors`, the electronics and mechanics of the `motors` and the `electronic speed controllers`, as well as `sensor` noise, inaccuracies, and rounding errors.

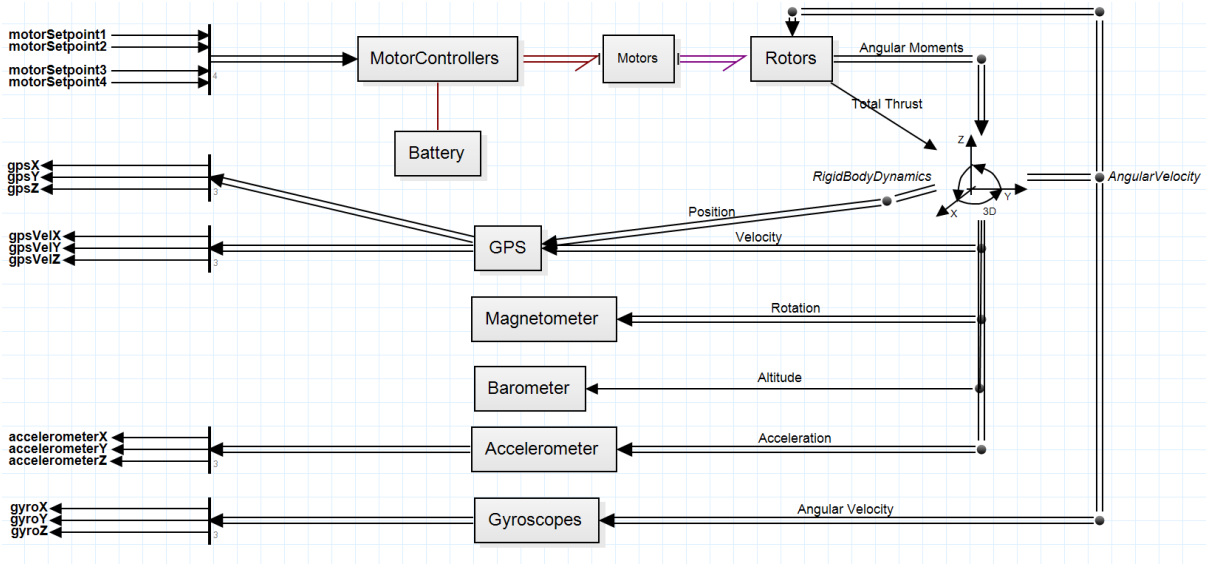


Figure 67: 20-sim model of the UAV

ArduPilot: Figure 68 shows an overview of the `ArduPilot` model, which is described in VDM-RT. The main class of the model `ArduPilot` starts a `Scheduler` and a `Flight controller`. To improve model fidelity, sensor values are updated periodically by the scheduler to emulate the real update frequencies of the various sensors. The flight controller takes input from a pilot and from sensor data, on which sensor fusion is performed, and is responsible for calculating desired accelerations for the UAV. These accelerations are translated, by the `Motors` class, into motor setpoints for each motor. The translation involves a complex tradeoff between roll, pitch, and yaw accelerations and total thrust. The `MotorsQuad` class is shown to illustrate that the `Motors` class can be extended to support any number or configuration of motors.

The `Flight Control` class contains the core functionality of the model and is probably also the most complex part. It can operate in multiple flight modes, which make use of either an attitude controller or both an attitude and a position controller. The attitude controller is capable of obtaining and maintaining any given attitude, whereas the position controller is capable of controlling the altitude. Both the attitude and position controllers depend on a number of low level controllers, such as Proportional-Integral-Derivative (PID) controllers and a number of different

filters to remove unwanted noise and vibrations caused by the fast spinning rotors.

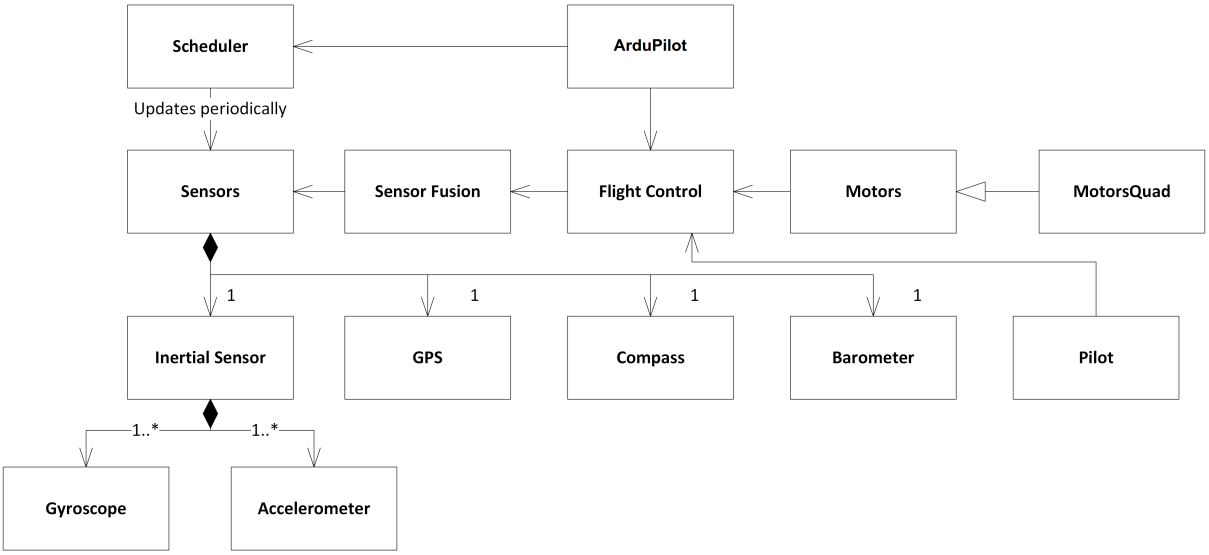


Figure 68: ArduPilot model overview

7.4.2 Configuration

This pilot study does not use any parameters and the connections should be self explanatory from Figure 66.

7.5 Co-simulation

Two multi-models are defined for this pilot study. The only difference between the two is whether the 3D animation is included or not.

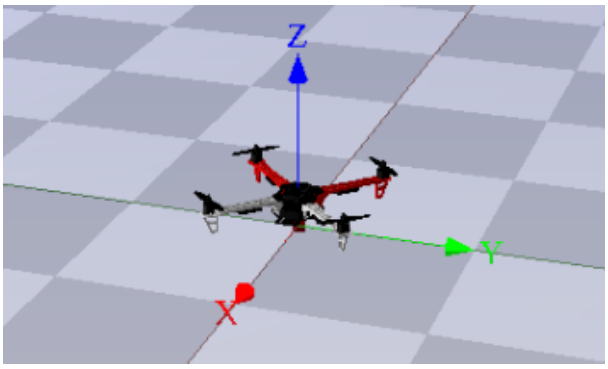


Figure 69: 3D visualization of the UAV

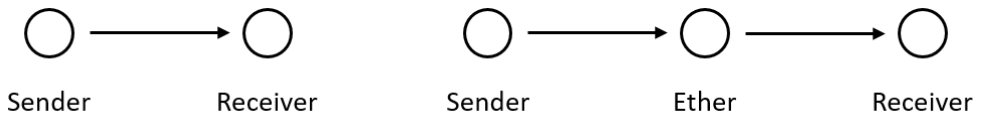
8 Ether

8.1 Example Description

This example explores ways to model network communications between controllers. This involves passing messages —VDM values encoded as strings— from a model called *Sender* to a model called *Receiver*. This is either done directly, as illustrated in Figure 70(a), or via a third model called the *Ether*, as illustrated in Figure 70(b).

While this example demonstrates that direct connection is possible, for multi-models with large numbers of connected controllers (for example swarms or cooperative vehicles) it becomes unwieldy. This example includes an Ether model, which represents an abstract communication mechanism, that handles message passing between the Sender and Receiver. This Ether can be used in other models.

The introduction of a model of communications also permits a range of realistic and faulty behaviours to be introduced, such as message delay, duplication, and loss. The ether pattern which this example implements is described in greater detail in Deliverable D3.3a [FGP17], and includes a discussion of realistic and faulty behaviour.



(a) Sender and Receiver connected directly

(b) Sender and Receiver connected via the Ether

Figure 70: Topology of the 'Direct' and 'Ether' multi-models.

8.2 Usage

The example is available from the INTO-CPS application menu at *File > Import Example Project* or at `https://github.com/INTO-CPS-Association/example-ether` in the *master* branch. There are several subfolders for the various elements: `FMU` – contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations – with Direct and Ether configurations.

The `case-study_ether folder` can be opened in the INTO-CPS application to run various co-simulation experiments. To run a simulation, expand the emphDirect or emphEther models, then open the *co-sim_ direct* or co-sim_ether experiments and click *Simulate*. Section 8.5 below gives some suggestions of how to explore the multi-model.

8.3 Multi-model

8.3.1 Models

There are three models in this example, all of which are DE models written in VDM:

Sender: This model has a single output port called `out`, of type `String`. The `Controller` class generates random messages every 0.1 seconds, and passes them to the output port. Each message consists of three integers in the range (0,10), and are converted to a string representation using the `VDMUtil` standard library. The time and content of each message is printed to the console.

Receiver: This model has a single input port called `iin`, of type `String`. The `Controller` class listens for messages on the input port and attempts to convert message strings back to a VDM type representation using the `VDMUtil` standard library. Empty strings are often received at the start of co-simulation, and these are ignored. If conversion is successful, the time and content of the message is printed to the console.

Ether: This model represents an abstract communication medium. It has an input port called `sender` and an output port called `receiver`, both of type `String`. The `Ether` class passed strings from the `sender` port to the `receiver` port. Although in this example only one message will be received at a time, in general there may be multiple messages for the same destination during a single update, so the `Ether` class collects messages in a list and passes on a string of strings that the destination (i.e. in this case Receiver) must decode.

The connections in the `Ether` class are determined by the values passed to the constructor, which is called in the `System` class. The constructor takes a map of named input ports, a map of named output ports and a set of pairs indicating which inputs are connected to which outputs. The `Ether` class does not examine the value of the messages that it passes. This model can be reused in other multi-models where message-based communications between models are required.

8.3.2 Configuration

There are two multi-model configurations included in the example:

Direct In this configuration, the `Sender.out` port is connected directly to the `Receiver.in` port.

Ether In this configuration, the `Sender.out` port is connected to the `Ether.sender` port and the `Sender.reciever` port is connected to the `Receiver.in` port.

These two different configurations allow exploration of the consequences of introducing the Ether FMU. The Sender model does not need to know if it is connected to the Ether or not. Since the Ether allows one-to-many communications, it passes lists of values, so the Receiver must check whether it received a single value from the Sender directly, or a list containing a single value via the Ether. This could be avoided in this example by letting the Ether assume that there is only one connection, but would make the Ether less general. The included implementation means that the Ether can be used directly in other multi-models — only the `HardwareInterface` class and called to the constructor of `Ether` would need to be changed for the context of the new model.

8.4 Co-simulation

The simulation time for the multi-model is set to one second, which is sufficient to see the behaviour of the system. During this time, 10 messages are sent from the Sender. Not all messages are received by the Receiver since at least one extra update cycle is needed to process the final message, or final two messages in the case of communication via the Ether.

8.5 Analyses and Experiments

8.5.1 Sample Experiments

The following experiments are instructive in demonstrating the effects on introducing the Ether and the effects of the relative speeds up the *Sender*, *Ether* and *Receiver* on messages. All three FMUs print messages and times to the COE console.

1. Run the *Direct* co-simulation and observe that messages arrive at Receiver one step (0.1 seconds) later. Run the *Ether* co-simulation and observe that messages arrive two steps (0.2 seconds) later.
2. Change the frequency of the Ether to 20Hz or higher. Run the *Ether* co-simulation and observe that messages arrive at Receiver one step (0.1 seconds) later. This is because the Ether can now update in between steps of the Receiver.
3. Set the Ether back to 10Hz and change the frequency of the Sender to 20Hz or higher. Run the *Ether* co-simulation and note how messages are not lost because they are changed by the Sender before they are read by the Ether.
4. Set the Sender back to 10Hz and change the frequency of the Receiver to 20Hz or higher. Run the *Ether* co-simulation and note how messages are now duplicated because the Receiver is reading them twice or more.

If elimination of message loss or duplication is important in a multi-model, the description of the ether pattern in Deliverable D3.3a [FGP17] gives some initial guidance on overcoming such problems by introducing extra logic to give quality of service (QoS) guarantees in message-passing.

8.5.2 Test Automation and Model Checking

Our model for test automation is different from that for Multi-model and analysis in the following aspects.

- Our test model aims for model network communications between multiple controllers through an intermediate *Ether*. The *Ether* is regarded as a shared medium between controllers to dispatch packages from multiple sources to multiple targets. Therefore, we do not need all controllers connected to each other directly and the direct connection mode between controllers is not taken into account in our test model.

- Current Multi-model configuration **Ether** includes only one `Sender` and one `Receiver`. In order to be used as a communication medium in the UAV swarm pilot study in Section 9, the *Ether* model should support more senders and receivers. Our test model supports two controllers to be connected through the *Ether*, and each controller is composed of a sender and a receiver to allow it to communicate with others bidirectionally.
- For Co-Simulation, these controllers could be in different FMUs. Then the *Ether* should provide the capability to dispatch packages from sources to destinations according to their targets. For instance, one central controller would send a command to the controller one at a specific time, then later it would send another command to the controller two. We introduce a new package structure (as shown in Section 8.5.2) which is composed of an address part and a payload part. Therefore, the *Ether* is able to dispatch packages by the addresses that are encoded in the packages. In particular, it supports broadcast and unicast.
- The model for Multi-model and analysis has messages encoded in strings. But due to the fact that RT-Tester is not able to parse string or array in test models, we are not able to model variable-length strings in test models. The encoding mechanism in Section 8.5.2 is an example to encode a string (up to 5 Base64 characters) into a 32-bit integer. We allow different applications can have different encoding mechanisms. As an example, the test model in the UAV swarm study uses three doubles for its payload and each double means a position or velocity in one of three axes (X, Y and Z). However, for this *Ether* pilot study, the test model will not look into the inside of payloads and it just uses an integer for illustration.

Subsequently, we present the test model in SysML. Then model checking is applied to check a number of properties of the test model. Finally a SUT is implemented manually and test automation is applied to test whether the SUT is a correct implementation of the test model.

Test Model [Ether mode] Our test model connected two controllers (or FMUs, here we call it *End*) through a *Ether*. The interfaces between the *TestEnvironment* and the *SystemUnderTest* are the packages that two *Ends* would send and receive.

String Encoding A package is composed of two parts: address and payload. Each of them is encoded in one integer (32-bit).

- Address (32-bit integer): the high 24 bits are always 0, the low 8 bits are split into two 4-bit parts (one for the destination port and one for the source port)
  - 0xF: for broadcast;
  - 0x1-0xE: port 1 - port 15;
  - 0x0: no message signal (used for the polling method to identify a valid package).
- Payload (32-bit integer): the highest two bits are reserved, other 30 bits consist of five 6-bit parts (each encodes a Base64 character)

  - Base64: 64 characters (26 uppercase alphabet letters [A-Z], 26 lowercase alphabet letters [a-z], 10 digits [0-9], '+' and '/'),
  - up to five characters in a string.

Finally, it supports

- up to 15 incoming ports and 15 outgoing ports, and each FMU (or end) has a paired incoming and outgoing ports,
- up to 15 FMUs connected,
- the paired ports have the same and unique id,
- the destination address of an input pair can be its paired port (send a message to itself),
- up to 5 characters in the payload of each package.

Inputs and Outputs Inputs and outputs of the test model are shown in Figure 71.

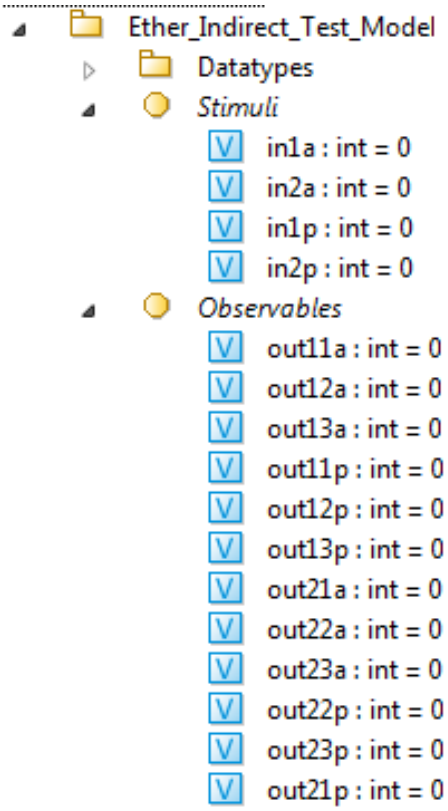


Figure 71: Input and Output Variables of Ether

The *SystemUnderTest* receives the following inputs (stimuli) from the *TestEnvironment*:

- $in1a$ and $in1p$: address and payload of the incoming package in *Ether* from End 1,
- $in2a$ and $in2p$: address and payload of the incoming package in *Ether* from End 2.

The *SystemUnderTest* provides the following observable outputs to the *TestEnvironment*:

- *out*11*a* and *out*11*p*: address and payload of the first outgoing package in *Ether* to End 1,
- *out*12*a* and *out*12*p*: address and payload of the second outgoing package in *Ether* to End 1,
- *out*13*a* and *out*13*p*: address and payload of the third outgoing package in *Ether* to End 1,
- *out*21*a* and *out*21*p*: address and payload of the first outgoing package in *Ether* to End 2,
- *out*22*a* and *out*22*p*: address and payload of the second outgoing package in *Ether* to End 2,
- *out*23*a* and *out*23*p*: address and payload of the third outgoing package in *Ether* to End 2.

Other Interfaces Other interfaces, as shown in Figure 72, are used in the *SystemUnderTest* for connections between the *Ends* and the *Ether*.

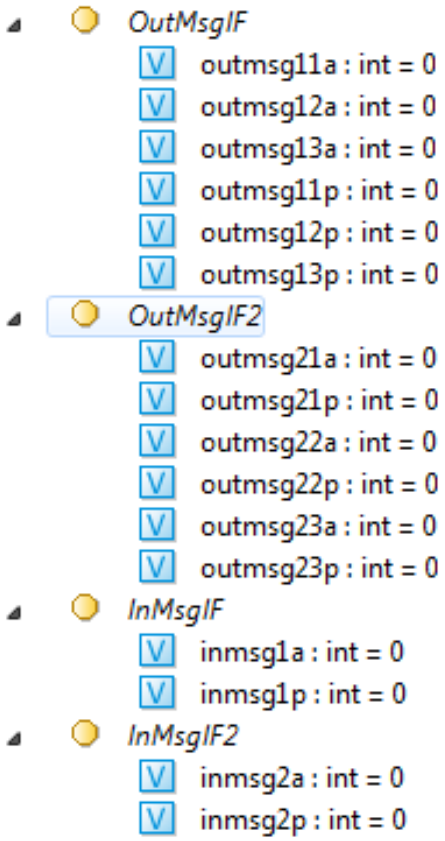


Figure 72: Other Interfaces of Ether

- $InMsgIF$ and $OutMsgIF$: the incoming and outgoing interfaces between the *End1* and the *Ether*,

- $InMsgIF2$ and $OutMsgIF2$: the incoming and outgoing interfaces between the *End2* and the *Ether*.

SystemUnderTest Architecture Diagram The system architecture diagram of the *SystemUnderTest* is shown in Figure 73. It consists of two *Ends*: *End1* and *End2*, and one *Ether*.

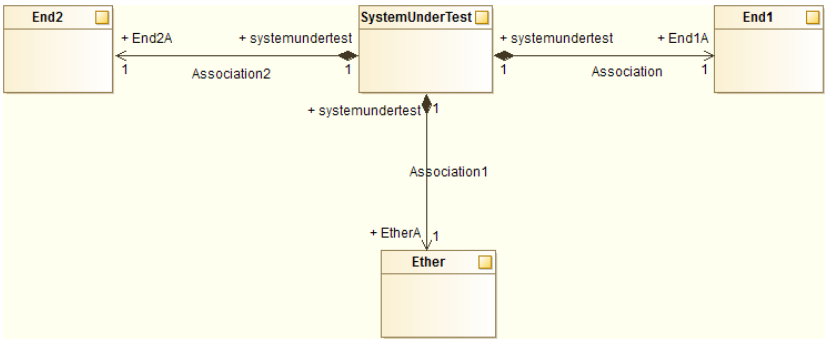


Figure 73: Ether Architecture Diagram

SystemUnderTest Connection Diagram The connection diagram of the *SystemUnderTest* is illustrated in Fig 74. Both *End1* and *End2* are connected to the *Ether*, but they are not connected directly. Therefore, a package that is sent from the *End1* to the *End2* would transmit to the *Ether* at first, then dispatch to the *End2* by the *Ether*.

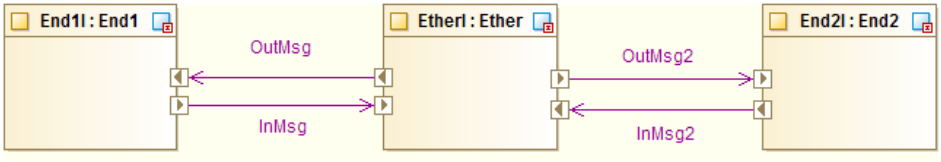


Figure 74: Ether Connection Diagram

End1 State Machine Diagram The behaviour of the *End1* is specified in the state machine diagram shown in Figure 75. It merely copies the input packages from the *TestEnvironment* to the *Ether* and the output packages from the *Ether* to the *TestEnvironment*.

Figure 75: End1 State Machine Diagram

End2 State Machine Diagram Similarly, the state machine diagram of the *End2* is displayed in Figure 76.

Figure 76: End2 State Machine Diagram

Ether State Machine Diagram The behaviour of the *Ether* is specified in the state machine diagram shown in Figur 77. Basically, it checks the incoming packages from the *End1* and the *End2*, and then dispatches them to their corresponding output ends according to the address in each package.

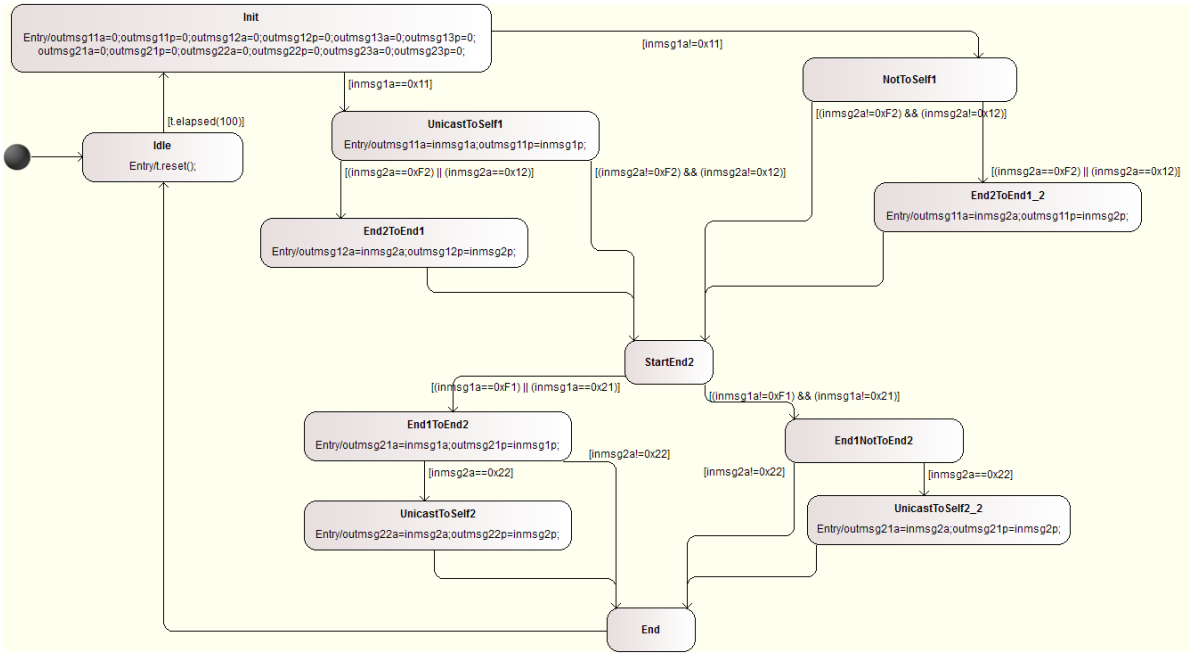


Figure 77: Ether State Machine Diagram

Test Input Simulation We use a state machine to specify the input sequence in the `TestEnvironment` of the test model. The diagram is shown in Figure 78. This state machine simulates all combinations of incoming packages from the *End1* and *End2* in terms of package addresses.

Model Checking Model checking can be applied to the test model to check if it satisfies several properties. In order to use bounded model checking in the INTO-CPS app, we set the bounded steps "BMC steps" to 50. The properties below are checked to hold in 50 steps.

P0 Livelock property by the **Check Mode** function on RT-Tester.

```
Check static model semantics ...done.
- IMR.SystemUnderTest.End1I... [PASS]
- IMR.SystemUnderTest.End2I... [PASS]
- IMR.SystemUnderTest.EtherI... [PASS]
- IMR.TestEnvironment.tesimulator... [PASS]
```

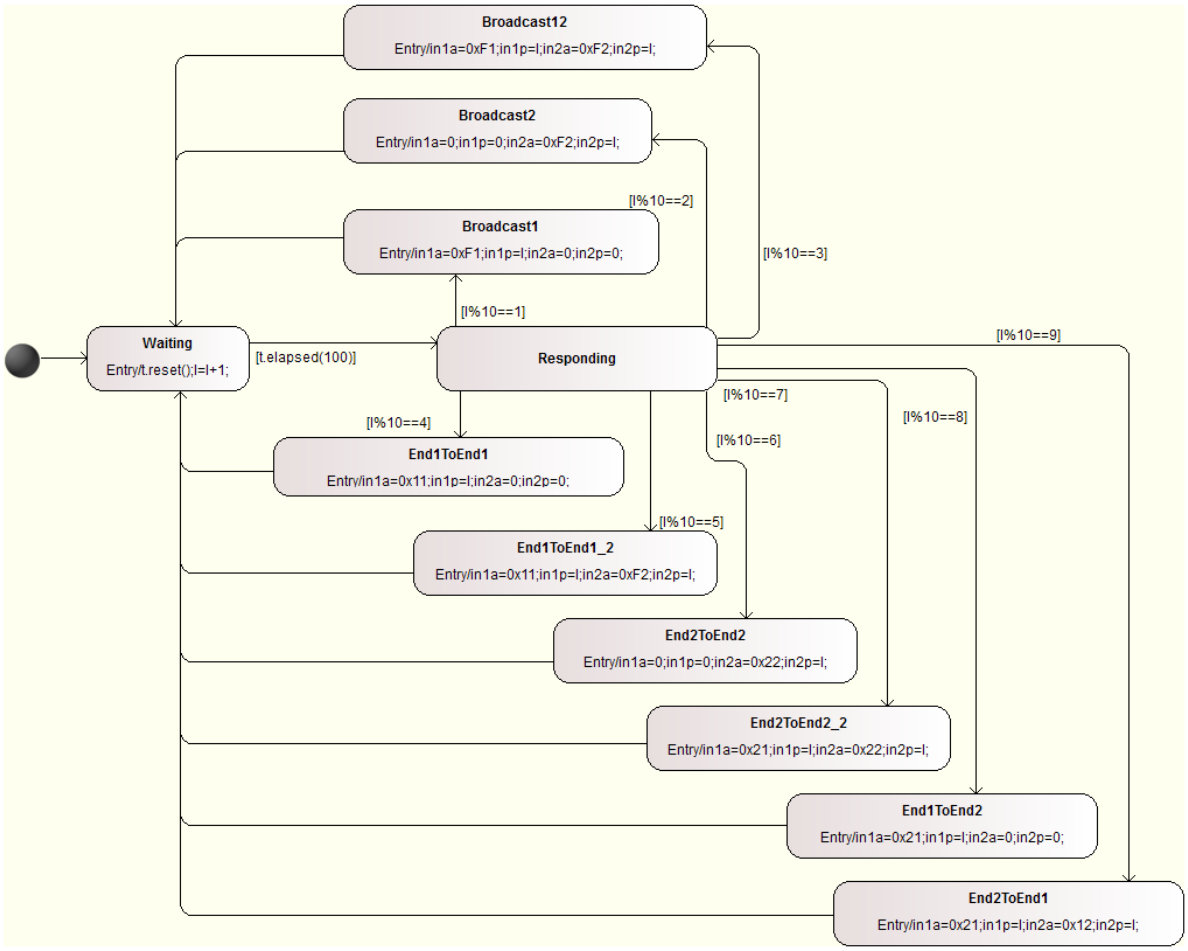


Figure 78: Test Input Simulation State Machine Diagram

P1 An *End* would never receive a package with a wrong destination address.

```
Globally ([IMR.out11a != 33] &&
    [IMR.out11a != 34] &&
    [IMR.out21a != 17] &&
    [IMR.out21a != 18])
```

P2 All controllers could be in the *Idle* state at the same time.

```
Finally ([IMR.SystemUnderTest.End1I.End.Idle
    && IMR.SystemUnderTest.EtherI.Ether.Idle
    && IMR.SystemUnderTest.End2I.End.Idle])
```

P3 A broadcast package would never be received by its source port.

```
Globally ([IMR.out11a != 241] && [IMR.out21a != 242])
```

A Manual Implementation of SUT in C A SUT is implemented in C. And it is finally built and wrapped into a SUT FMU.

Test Results

Test Goals Our test goals configuration is displayed in Figure 79. All basic control state coverage test cases and transition coverage test cases for three controllers are included, in the context of the *TestEnvironment* simulation enabled. In addition, a user defined test case is presented to require the test lasting at least 1.5 seconds. The generated and solved input signals for the test goals are shown in Figure 80 and Figure 81.

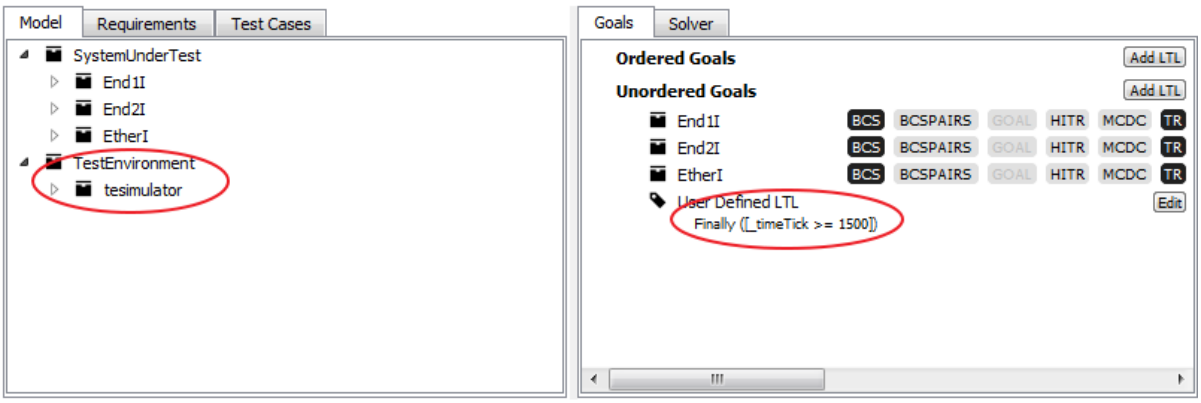


Figure 79: Test Goals Configuration

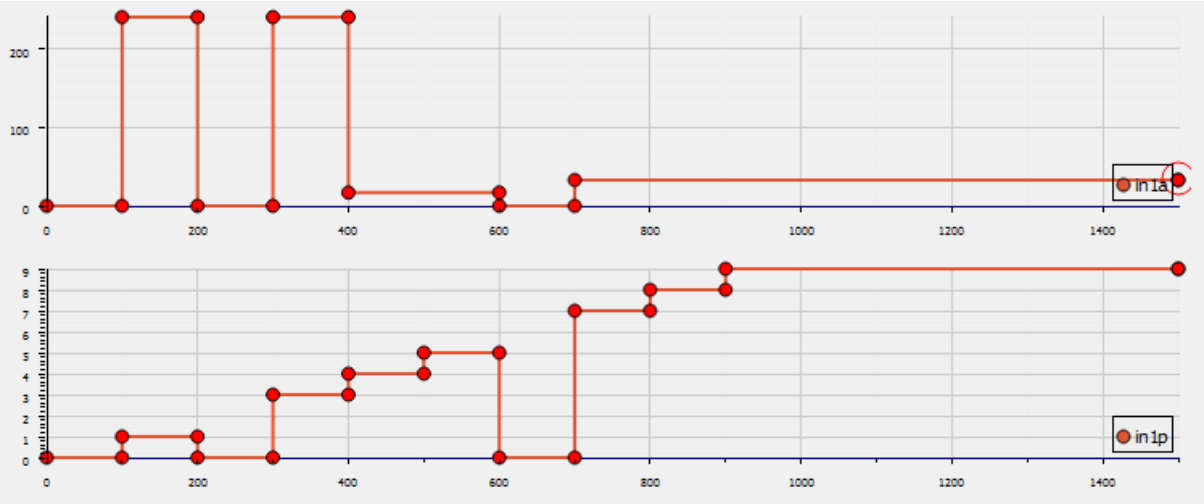


Figure 80: Input Signals of *End1*

The input address signals for both *End1* and *End2* (where the payloads are excluded) are marked and illustrated in Figure 82.

TP-00* vs. *Simulation The *TP-00* is a test procedure in RT-Tester that provides test inputs and checks expected results, while *Simulation* is a SUT test procedure created from the same test model as the *TP-00*. Our test results show all these test cases are with `PASS` or `INCONCLUSIVE` verdicts.

TP-00* vs. *SUT This is to test if our manually implemented SUT is a correct implementation of the test model or not. Our test results show all these test cases are with `PASS` or `INCONCLUSIVE` verdicts. Corresponding to the inputs shown in Figure 80 and

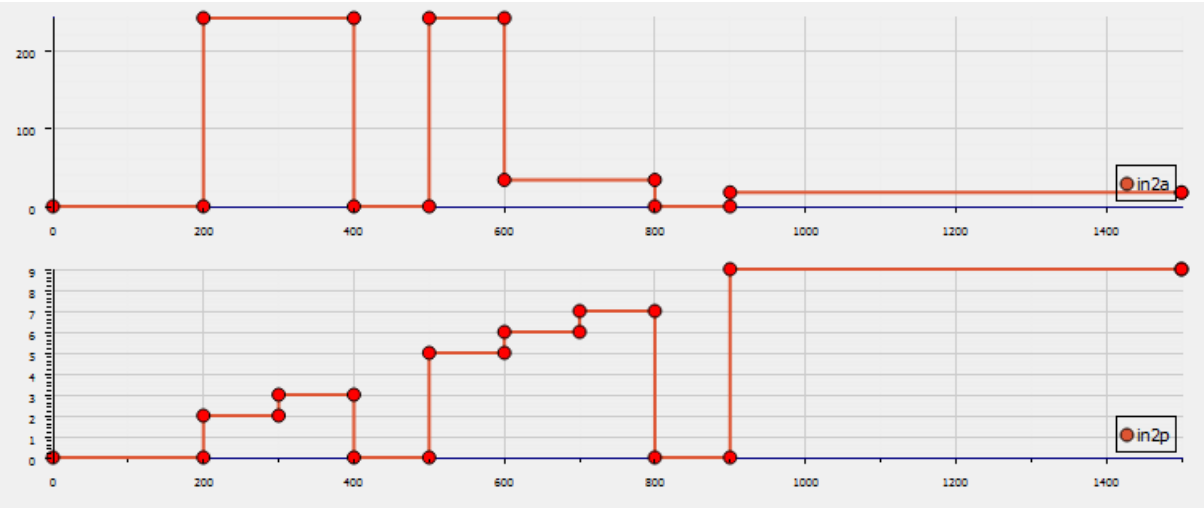


Figure 81: Input Signals of *End2*

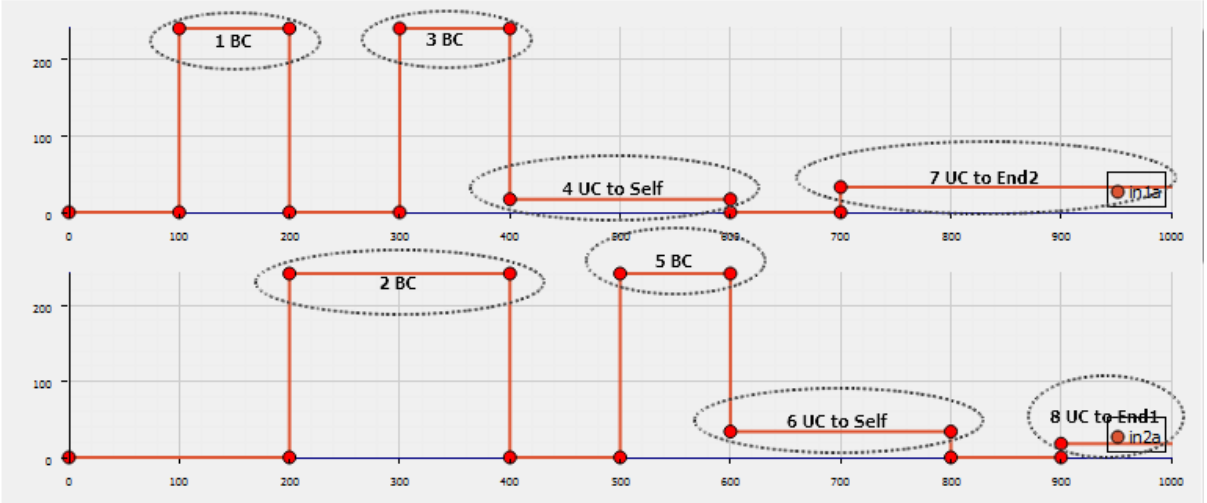


Figure 82: Input Address Signals

Figure 81 as well as the marked input signals in Figure 82, the output signals for the *End1* and the *End2* are illustrated in Figure 83 and Figure 84 respectively.

Our test results show all these test cases are with `PASS` or `INCONCLUSIVE` verdicts. These signals are described below with numbering.

- 1. Broadcast from *End1* ([100ms, 200ms]),
  - as shown in the outputs of *End2* ([300, 400]),
  - therefore, 200ms delay.
- 2. Broadcast from *End2* ([200, 400]),
  - as shown in the outputs of *End1* ([400, 600]).
- 3. Broadcast from *End1* ([300, 400]),
  - as shown in the outputs of *End2* ([500, 600]).

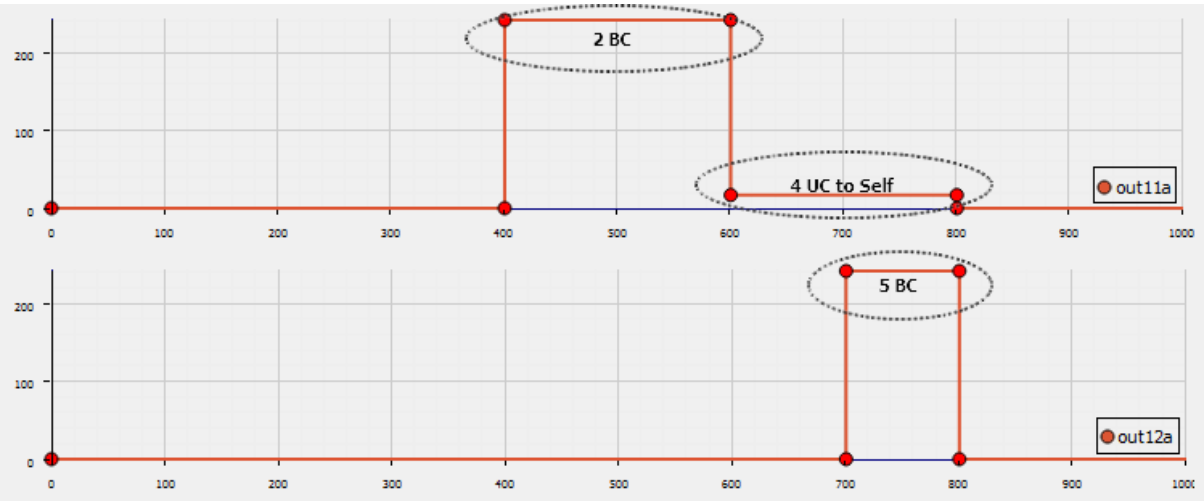


Figure 83: Output Signals of *End1*

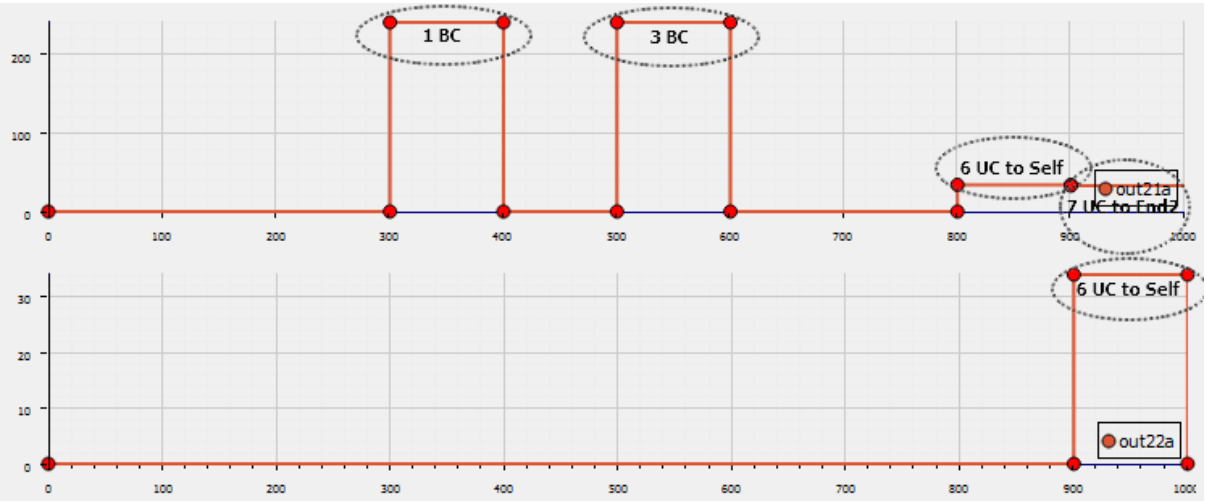


Figure 84: Output Signals of *End2*

- 4. Unicast from *End1* to *End1* ([400, 600]),
  - as shown in the output (`out11a`) of *End1* ([600, 800]).
- 5. Broacast from *End2* ([500, 600]),
  - as shown in the output (`out12a`) of *End1* ([700, 800]).
- 6. Unicast from *End2* to *End2* ([600, 800]),
  - as shown in the output (`out21a`) of *End2* ([800, 900]),
  - and the output (`out22a`) of *End2* ([900, 1000]).
- 7. Unicast from *End1* to *End2* ([700, ...]),
  - as shown in the output (`out21a`) of *End2* ([900, 1000]).
- 8. Unicast from *End2* to *End1* ([900, 1000]), ...

These behaviours are expected in our test model.

9 Swarm of UAV

9.1 Example Description

The Unmanned Aerial Vehicle (UAV) Swarm pilot study is concerned with a collection of UAVs that communicate in order to achieve some global behaviour. The pilot study is related to the previous UAV study in Section 7, however this does not focus on the low-level physical details. The pilot uses a simplified physical model to allow simulation of multiple UAVs simultaneously communicating.

In this pilot, each UAV is able to adjust its pitch, yaw and roll to move in 3D space using rotors. Figure 85 shows a single UAV with its motors, rotors and battery – each UAV has a controller which is able to communicate with its environment. In a swarm, the UAVs may cooperate in order to avoid collide, to achieve some predefined topology, or collaborate to provide some functionality. In this study, we demonstrate the use of a central controller to dictate the desired movements of the UAVs comprising the swarm.

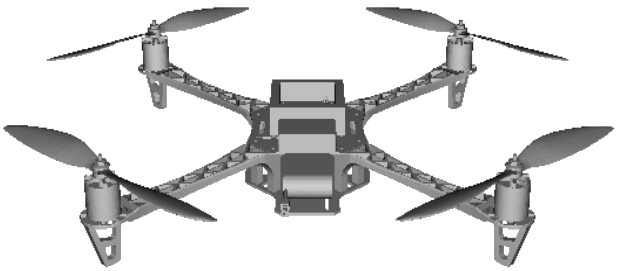


Figure 85: UAV 3D model

9.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-uav_swarm` in the *master* branch. There are several subfolders for the various elements: `FMU` – contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations – with 3D and non-3D options; and `SysML` – contains the SysML model defined for the study.

The `case-study_uav_swarm folder` can be opened in the INTO-CPS application to run various co-simulation experiments. To run a simulation, expand one of the multi-models and click 'Simulate' for an experiment.

9.3 INTO-CPS SysML profile

Using the INTO-SysML profile, we model a subset of the pilot study. The reason for modelling only a small subset will become clear in this section. In Figure 86, we show an Architecture Structure Diagram (ASD) depicting 3 UAVs – each comprised of a *UAVController* component and a *UAV* component, a component for 3D visualisation *UAV 3D*, and a *UAV Global Controller* component. Each component has a large number of inputs and outputs. Briefly the *UAV* has inputs for setting the pitch, yaw, roll and throttle, and outputs for the position, velocity and battery status. The *UAV Controller* has the same ports, but with reversed direction and also ports to receive commands from the global controller. The optional *UAV 3D* takes as input the position, pitch, yaw and roll of each UAV. Finally, the *UAV Global Controller* has a collection of ports for target locations for each UAV.

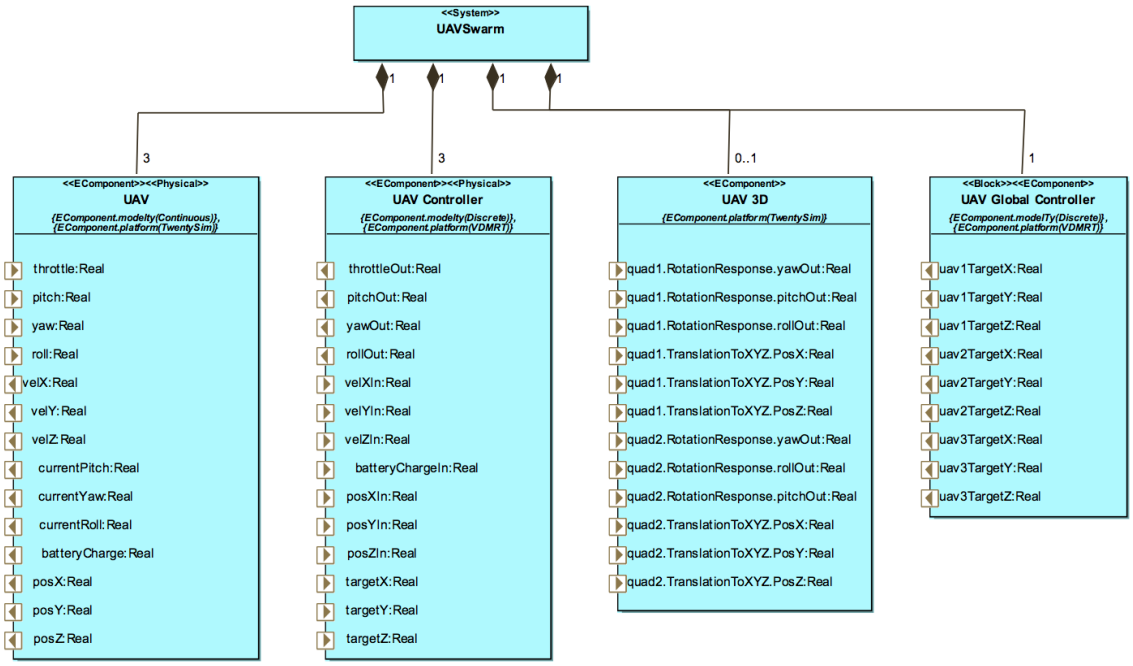


Figure 86: UAV Swarm Architecture Structure Diagram

Due to the large number of ports and connectors in this pilot, it is not appropriate to represent them all on the same diagram. As such, we create a single Connections Diagram (CD) per UAV each containing the same system instance (*swarm : UAVSwarm*). Combining all CDs produces a complete configuration for that system instance. The CD in Figure 87 depicts connections between the *UAVController* component and a *UAV* component of UAV1, and the connections to the *UAV 3D* and *UAV Global Controller* components related to UAV1. Note that the *UAV 3D* and *UAV Global Controller* instances have a subset of their ports shown.

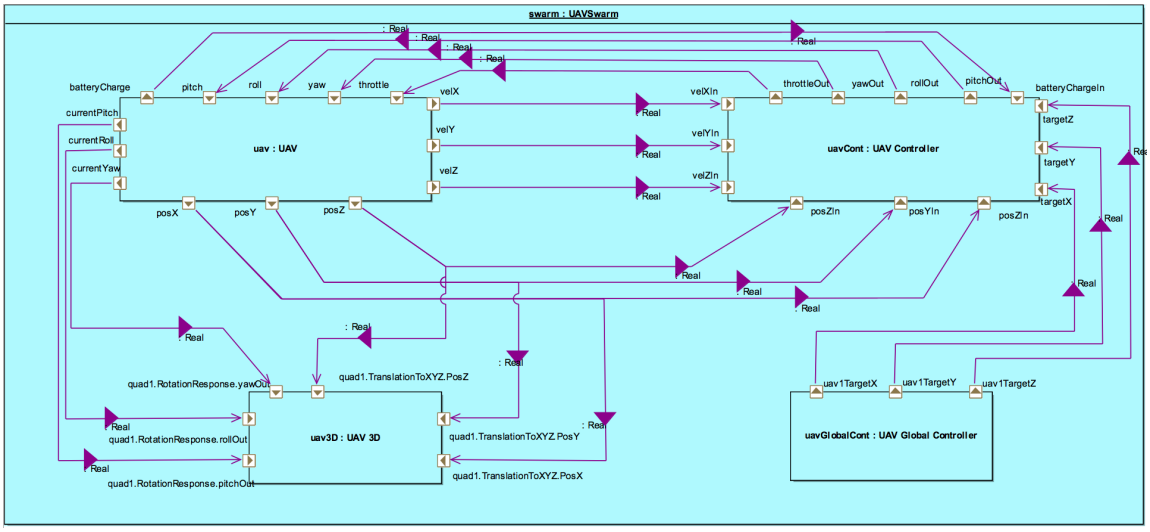


Figure 87: UAV Swarm Connections Diagram showing connections relating to UAV1

9.4 Multi-model

9.4.1 Models

There are three models defined here (we do not include a description of the UAV 3D model.

UAV: The physical model – *UAV* is defined in 20-sim. The *UAV* model represents the motors, rotors, battery and implicitly the frame of the UAV. The top-level structure of the model is shown in Figure 88.

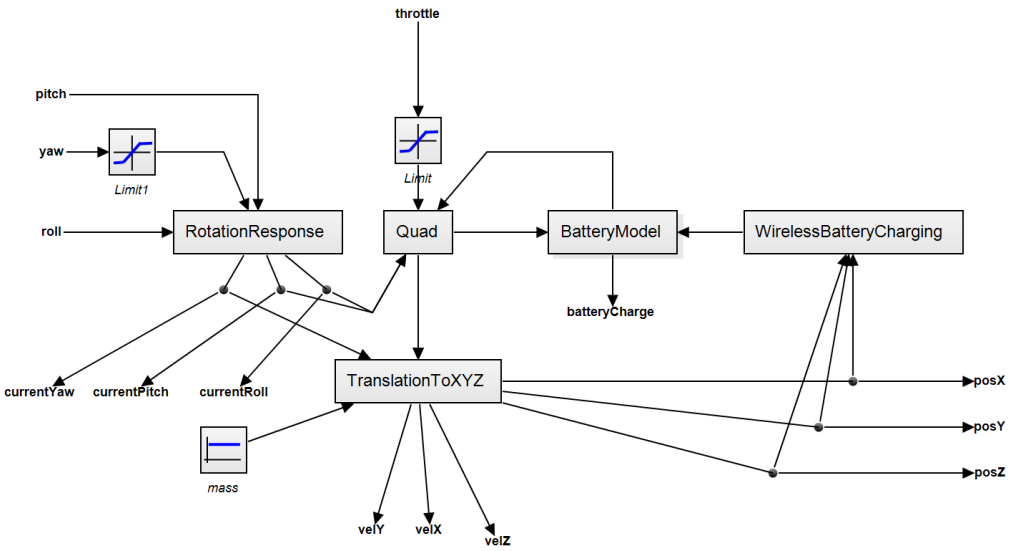


Figure 88: 20-sim model for UAV taking part in a Swarm

As can be seen from Figure 88, there are several input and output signal ports – these correspond to those ports defined in the SysML model above – for setting the pitch, yaw and roll and reporting various aspects for control and visualisation

purposes. The model is simplified and more abstract from that in Section 7, in that it is better suited to modelling shallow pitch and roll angles and contains several simplifications. The inputs from the controller enter the UAV model in two places with the pitch, roll and yaw entering the *RotationResponse* block while the throttle setting enters the *quad* block. This model of the UAV abstracts away the fact of there being four distinct rotors and so a simplified model of their effects on the orientation of the UAV, so instead of torques from the four rotors being applied to the UAV body and the acceleration calculated, the *RoationResponse* considers the difference between the current and requested pitch, roll and yaw rate an simulates a damped transition from the current to the requested. The resulting orientations are sent to the *quad* where they are combined with the throttle setting and battery voltage to give thrust vectors along the vertical, front and side axis of UAV. These vectors along with the UAV yaw are sent to the *translationToXYZ* where the thruse vectors are mapped onto the X,Y and Z axis and drag, accelerations, velocities and positions computed. The final two blocks *BatteryModel* and *WirelessBatteryChargin* calculate the charge going into and out of the battery so that its working voltage may be output to the *quad* model.

UAVController: The first of two VDM-RT models – *UAVController* – has a similar architecture to other pilots (e.g. the line follower robot in Section 5), in that the *System* has an instance of a *HardwareInterface* class containing ports for the inputs and outputs of the model. The *System* class also has an instance of the *Controller* class, which owns several instances which sense or make changes to the environment – they are *Actuators*, *Sensors* and *Commands*. The controller has a collection of operations that aim to control the movement of the underlying UAV – the top-level control loop takes the commands from the environment with target locations. Given the X, Y and Z coordinate the UAV should meet, the controller calculates the throttle, yaw and roll to achieve the change form the current position to that target. Those values are sent on the output ports.

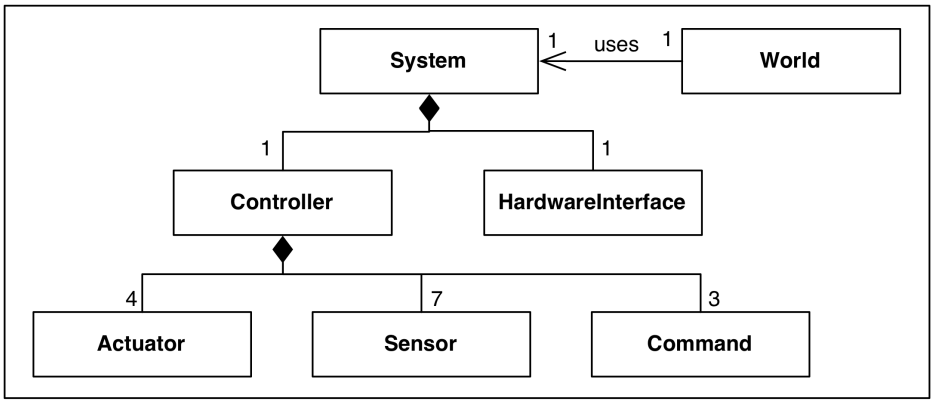


Figure 89: UAV Controller architecture

GlobalUAVController: The global controller for the UAV Swarm study is a simple model, which sends target locations to the different UAVs in the study at differing times. The *System* class contains the *HardwareInterface* class with ports for outputs of the model, and a *Controller* class which owns instances of objects which set those output values. In addition, the *Controller* includes a *Command* class that contains

the main control loop. This loop will change the target coordinates of each UAV at different time steps, which are sent as outputs to the correct UAV.

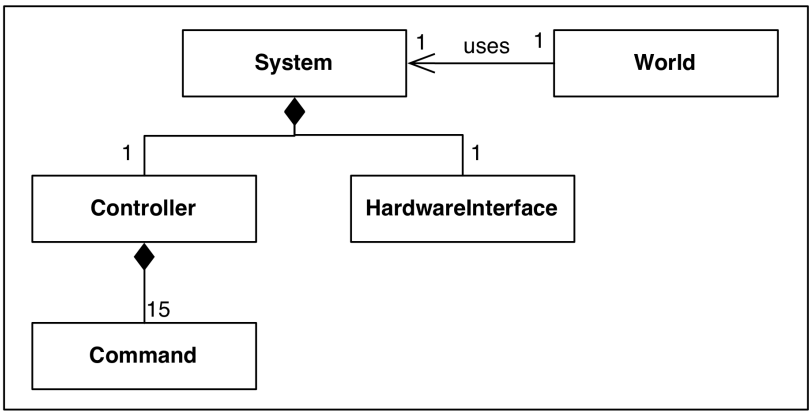


Figure 90: Global UAV Controller architecture

9.4.2 Configuration

The multi-model configuration comprises a collection of connections between the FMUs. We do not use parameters in the example. The connections may be grouped into three classes: between *UAV* and *UAVController*, between *UAVController* and *GlobalController* and *UAV* to *3DUAV*. These groups are repeated for each UAV in the pilot, therefore we only describe one set of connections – see Section 9.5 for different co-simulation experiments.

The first set is between the *UAV* and *UAVController*:

- from the *UAV* `velX` port to the *UAVController* `velXIn` port;
- from the *UAV* `velY` port to the *UAVController* `velYIn` port;
- from the *UAV* `velZ` port to the *UAVController* `velZIn` port;
- from the *UAV* `batteryCharge` port to the *UAVController* `batteryCharge` port;
- from the *UAV* `posX` port to the *UAVController* `posXIn` port;
- from the *UAV* `posY` port to the *UAVController* `posYIn` port;
- from the *UAV* `posZ` port to the *UAVController* `posZIn` port;
- from the *UAVController* `throttleOut` port to the *UAV* `throttle` port;
- from the *UAVController* `pitchOut` port to the *UAV* `pitch` port;
- from the *UAVController* `yawOut` port to the *UAV* `yaw` port; and
- from the *UAVController* `rollOut` port to the *UAV* `roll` port.

The second set is between the *UAVController* and *GlobalController*:

- from the *UAVGlobalController* `uavTargetX` port to the *UAVController* `targetX` port;

- from the *UAVGlobalController* `uavTargetY` port to the *UAVController* `targetY` port; and
- from the *UAVGlobalController* `uavTargetZ` port to the *UAVController* `targetZ` port.

The final set is between the *UAV* and *3DUAV*:

- from the *UAV* `currentPitch` port to the *3DUAV* `quad.RotationResponse.pitchOut` port;
- from the *UAV* `currentYaw` port to the *3DUAV* `quad.RotationResponse.yawOut` port;
- from the *UAV* `currentRoll` port to the *3DUAV* `quad.RotationResponse.rollOut` port;
- from the *UAV* `posX` port to the *3DUAV* `quad.TranslationToXYZ.PosX` port;
- from the *UAV* `posY` port to the *3DUAV* `quad.TranslationToXYZ.PosY` port; and
- from the *UAV* `posZ` port to the *3DUAV* `quad.TranslationToXYZ.PosZ` port.

9.5 Co-simulation

Four multi-model configuration variations are defined to enable different co-simulation experiments. Those different multi-models vary by the number of UAVs (the number of *UAV* and *UAVController* FMU instances) and the inclusion of the *3DUAV* FMU for visualisation:

3-UAV-3D This multi-model comprises 4 FMUs: 3 instances of *UAV.fmu*; 3 of *UAVController.fmu*; 1 instance of *3DanimationFMU.fmu*; and 1 instance of *UAVGlobalController.fmu*. The multi-model has a co-simulation experiment with a run time of 10 seconds and with a variable step size. The 3D visualisation shows the flight paths of the UAVs.

3-UAV-Non-3D The 3-UAV non-3D multi-model is the same as the above study, however without the *3DanimationFMU.fmu* instance. Without the 3D view, livestream values are enabled for the x-position of the 3 UAVs – this may be changed.

5-UAV-3D The 5-UAV-3D multi-model is the same as in the 5-UAV-3D version, however with 5 instances of *UAV.fmu* and *UAVController.fmu*.

5-UAV-Non-3D Again, the 5-UAV-Non-3D multi-model is the same as in the 5-UAV-Non-3D version, however with 5 instances of *UAV.fmu* and *UAVController.fmu*.

The 3D visualisation offered by those 3D multi-models opens a 3D visualisation window as shown in Figure 91 which depicts the state of the swarm as the simulation progresses. It should be noted that the FMU has a fixed number of UAV objects, therefore the FMU must be extended to handle a greater number of UAVs in a swarm.

Figure 91: UAV Swarm visualisation

9.6 Analyses and Experiments

9.6.1 Test Automation and Model Checking

Unlike multi-model that has three or five UAVs in a swarm, we use two UAVs in the test model. This is because multiple instances are not supported in the test model. For each instance, we have to create a new block and have all similar local variables and state machine diagrams in this block. More instances will complicate the test model as well as solving of test cases in the SMT solver. Eventually, we choose two UAVs to keep the test model relatively simple and still more than one UAV.

In addition, communication between controllers through the Ether is also introduced. Therefore, the system covered in our test model is composed of

- two physical *UAVs*,
- two discrete *UAV Controllers* (one controller for each UAV),
- a discrete *UAV Global Controller*,
- a *Ether* model through which
  - *Global Controller* is able to send target positions to *UAV Controllers* by unicast, and
  - *UAV Controllers* are able to broadcast their current positions to other *UAV Controllers* and *Global Controller*.

Test Model Our test model aims to model the discrete part of the UAV swarm study. To be more specific, it includes two *UAV Controllers*, a *Global Controller* and a *Ether* which is described in Section 8. The physical *UAVs* are regarded as the test environment of the model.

Inputs and Outputs The *SystemUnderTest* receives the following inputs (stimuli) from the *TestEnvironment*:

- *targetX*1, *targetY*1, and *targetZ*1: target positions on X, Y and Z respectively for the first *UAV* (*UAV1*),
- *targetX*2, *targetY*2, and *targetZ*2: target positions on X, Y and Z respectively for the second *UAV* (*UAV2*).
- *velX*1, *velY*1, and *velZ*1: current velocity of *UAV1* on the X, Y and Z axes respectively,
- *posX*1, *posY*1, and *posZ*1: current position of *UAV1* on the X, Y and Z axes respectively,
- *batteryCharge*: battery status of *UAV1*,
- *velX*2, *velY*2, and *velZ*2: current velocity of *UAV2* on the X, Y and Z axes respectively,
- *posX*2, *posY*2, and *posZ*2: current position of *UAV2* on the X, Y and Z axes respectively,
- *batteryCharge*: battery status of *UAV2*.

The *SystemUnderTest* provides the following observable outputs to the *TestEnvironment*:

- *pitchOut*1 and *pitchOut*2: pitch set point for *UAV1* and *UAV2* respectively,
- *throttleOut*1 and *throttleOut*2: throttle set point for *UAV1* and *UAV2* respectively,
- *rollOut*1 and *rollOut*2: roll set point for *UAV1* and *UAV2* respectively.

Interfaces In addition to the interfaces between *SystemUnderTest* and *TestEnvironment* for inputs and outputs, six extra interfaces, shown in Figure 92, are defined for connections between all controllers and *Ether*.

- *GlobalEtherInIF* and *GlobalEtherOutIF*: interfaces for connections between *Global Controller* and *Ether*. In the name of the interfaces, *in* and *out* mean inputs to and outputs from *Ether*. Therefore, *GlobalEtherInIF* is an interface from *Global Controller* to *Ether*, and *GlobalEtherOutIF* is an interface from *Ether* to *Global Controller*. For the variable names in the interfaces, *a* denotes the address part and *p* denotes the payload part. *GEin*1*a*, *GEin*1*p*1, *GEin*1*p*2 and *GEin*1*p*3 in *GlobalEtherInIF* compose a package in which there are one integer for the address and three doubles for the payload. But *GlobalEtherOutIF* contains two packages, which means *Ether* can send two packages to *Global Controller* once but it can only receive one package from *Global Controller*.
- *UAV1EtherInIF* and *UAV1EtherOutIF*: interfaces for connections between *UAV1 Controller* and *Ether*.
- *UAV2EtherInIF* and *UAV2EtherOutIF*: interfaces for connections between *UAV2 Controller* and *Ether*.

SystemUnderTest *SystemUnderTest* is composed of two *UAV Controllers*, a *Global Controller* and a *Ether*. The architecture structure diagram of *SystemUnderTest* is shown

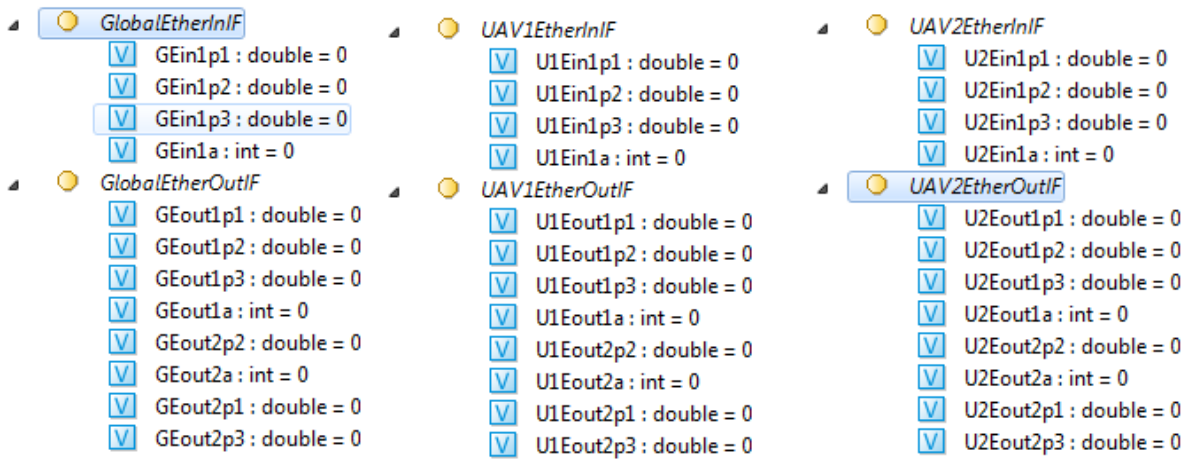


Figure 92: Interfaces in *SystemUnderTest*

in Figure 93 and the connection diagram is illustrated in Figure 94. From the connection diagram, we can see that all controllers are connected to *Ether* directly and each of them has two connections to *Ether* in two directions. Therefore they are able to communicate each other in two directions via *Ether*.

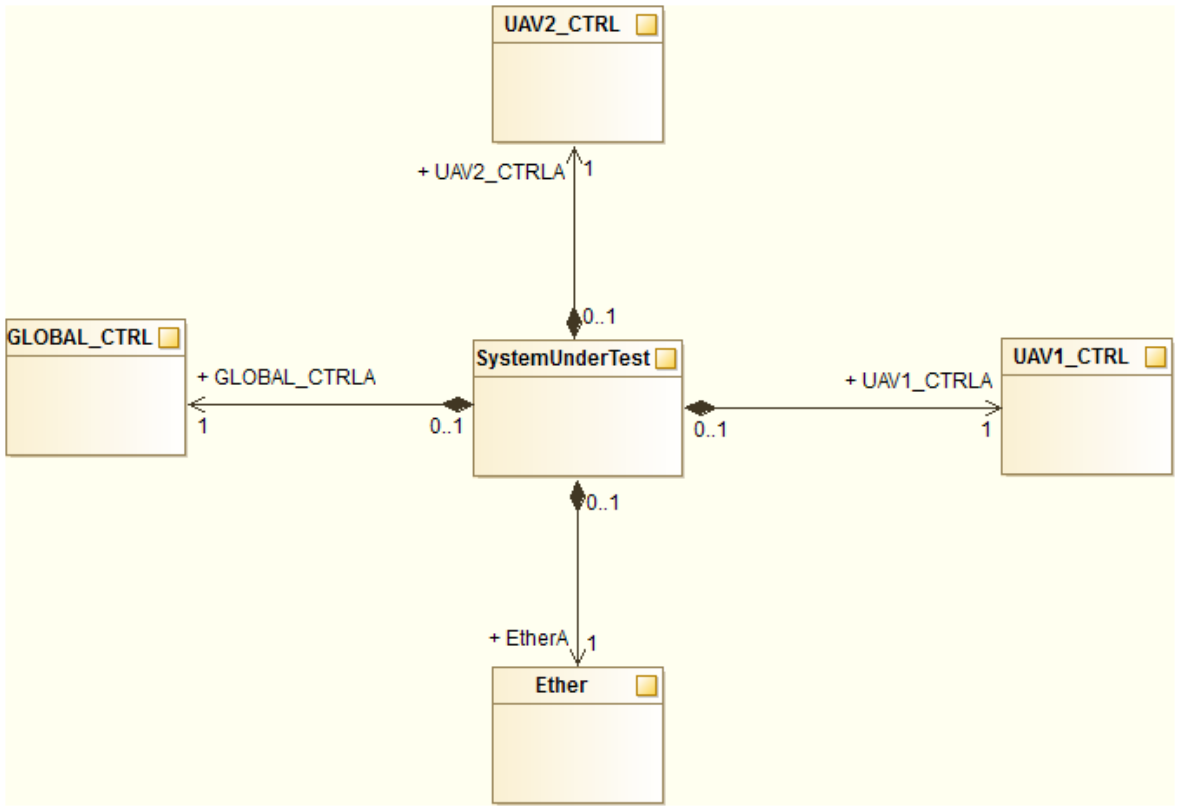


Figure 93: Architecture Structure Diagram of *SystemUnderTest*

Ether In order to contain messages for communication into the payload part of packages, the package structure is revised based on that given in Section 8.5.2.

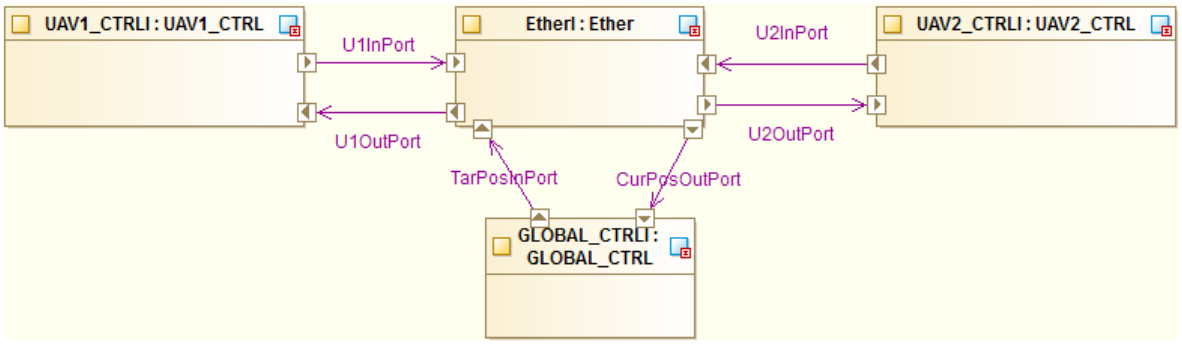


Figure 94: Connection Diagram of *SystemUnderTest*

A package is composed of two parts: address and payload. The address part is encoded in one integer and the payload part is in three doubles.

- Address (32-bit int): high 24 bits are always 0, low 8 bits are split into two 4-bit parts (one for the destination port and one for the source port).
  - 0xF: for broadcast;
  - 0x1-0xE: port 1 - port 15;
  - 0x0: no message signal (used for the polling method to identify a valid package);
  - In this case, 0x1 and 0x2 are used to identify *UAV1 Controller* and *UAV2 Controller*, 0xE for *Global Controller*, and 0xF for broadcast.
- Payload (three doubles): for the position or velocity on X, Y, and Z axes.

The state machine diagram of Ether is illustrated in Figure 95. After the `Init` state, it enters the `IDLE` state with the timer variable t reset. Then if 10 ms is elapsed, the transition to `PreUpdate` fires. It checks all input packages from *Global Controller* and *UAV Controllers*, then copies them into output packages of the expected ports. In the end, the state machine returns to `IDLE` and waits for the next loop.

UAV1 Controller Constant variables and local variables of *UAV1* are shown in Figure 96.

It is worth noting that three local variables ($curPosX2$, $curPosY2$, and $curPosZ2$) are used to store current positions of *UAV2*. In other words, *UAV1* is aware of the position of *UAV2*. These variables are updated from the packages broadcast from *UAV2* through *Ether*.

The state machine diagram is illustrated in Figure 97. Most of the time, the state machine resides in the `Waiting` state. And every 40 ms it starts to process.

- Firstly, it is going to update its own target position from the *Global Controller* and current positions of *UAVs* according to incoming packages from *Ether*. That is specified in the states between `PreUpdateTargetPos` and `PostUpdateTarCurPos`,

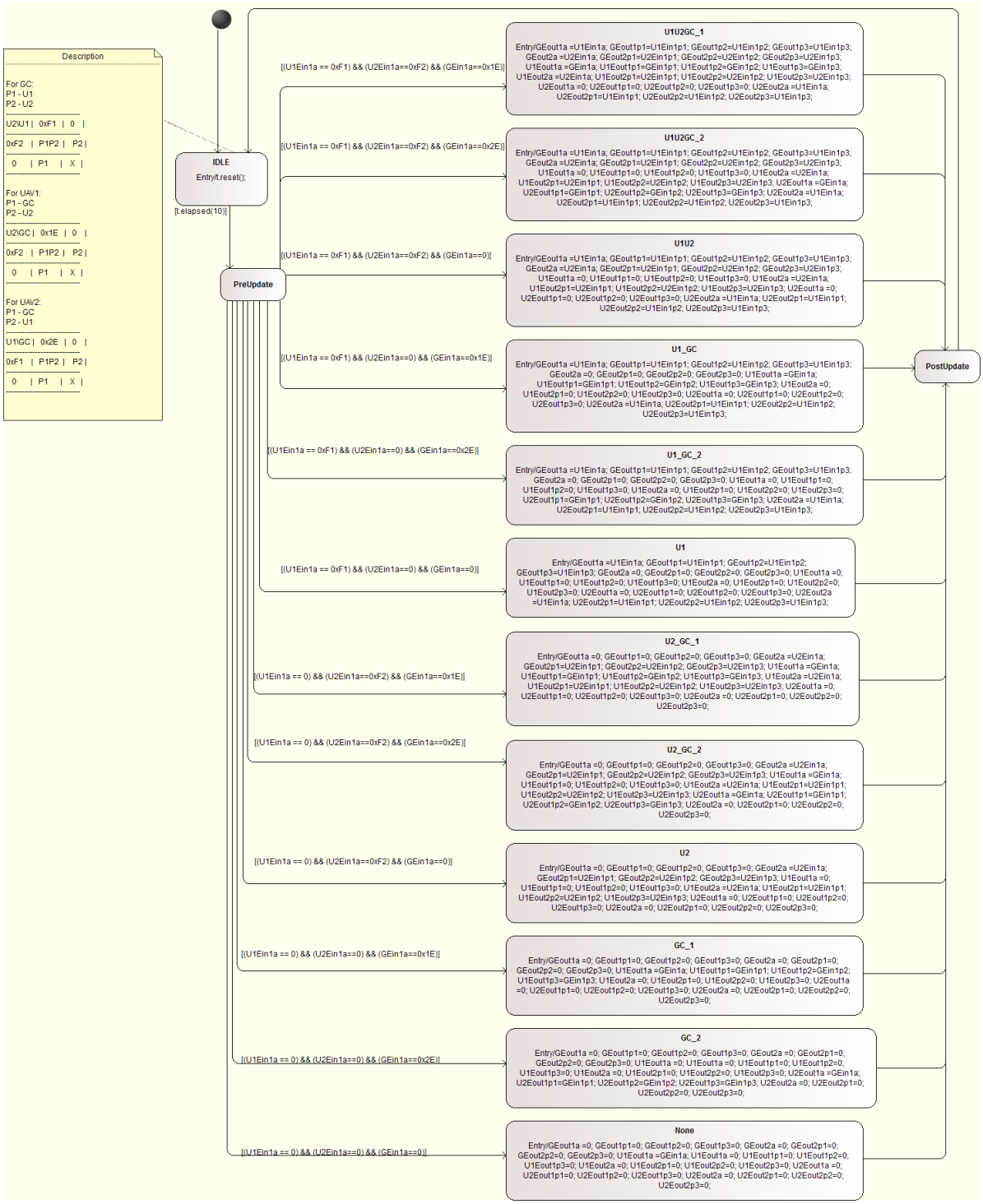


Figure 95: State Machine Diagram of Ether

- Then, it broadcasts its own current position in the `PostUpdateTarCurPos` state through *Ether*,
- Afterwards, it starts to update the parameters of the *PID* controller. The entry of the state `PreThrottle` calculates the *throttleIpart* and *throttlePpart* while the transition from it to the state `PostThrottle` updates *throttleDpart*. Finally, the entry of `PostThrottle` sets the output variable *throttleOut*. The states `PrePitch`

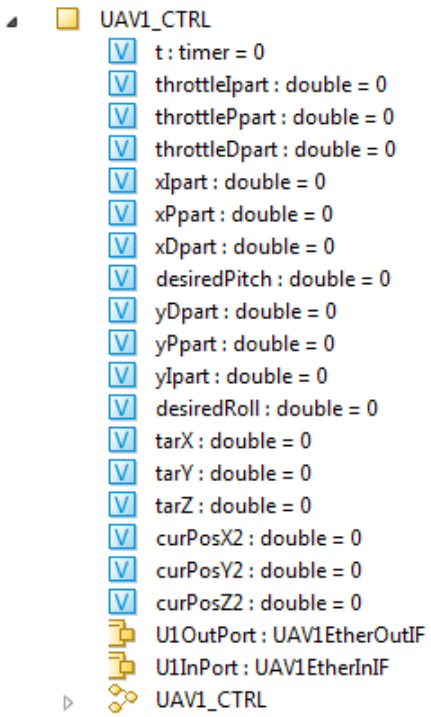


Figure 96: Variables of UAV Controller

and `PostPitch`, and the states `PreRoll` and `PostRoll` are similar.

- Eventually, the state machine returns back to the `Waiting` state and waits for the next loop.

UAV2 Controller The *UAV2 Controller* model is very similar to that of the *UAV1 Controller*.

Global Controller The *Global Controller* model has six local variables ($(curPosX1$, $curPosY1$, $curPosZ1$, $curPosX2$, $curPosY2$, and $curPosZ2)$ to keep current positions of *UAV1* and *UAV2*. They are updated from broadcast packages of *UAV1* and *UAV2*.

The *Global Controller* intends to send target positions to both *UAV1* and *UAV2* in each loop (40 ms). Then from the *UAV Controllers* aspect, they get their target position almost at the same time because they start to process inputs every 40 ms as well. Furthermore, since one controller is able to send only one package each time to *Ether* and the processing interval of *Ether* is 10 ms, we need to send target positions to *UAV1* and *UAV2* separately with an interval of more than 10 ms. In the end, we use timing shown in the state machine diagram of *Global Controller* in Figure 98. Its state machine loops every 40 ms too. *Global Controller* sends the target position to *UAV1* at 15 ms, the target position to *UAV2* at 30 ms, and check incoming packages from *Ether* at 40 ms.

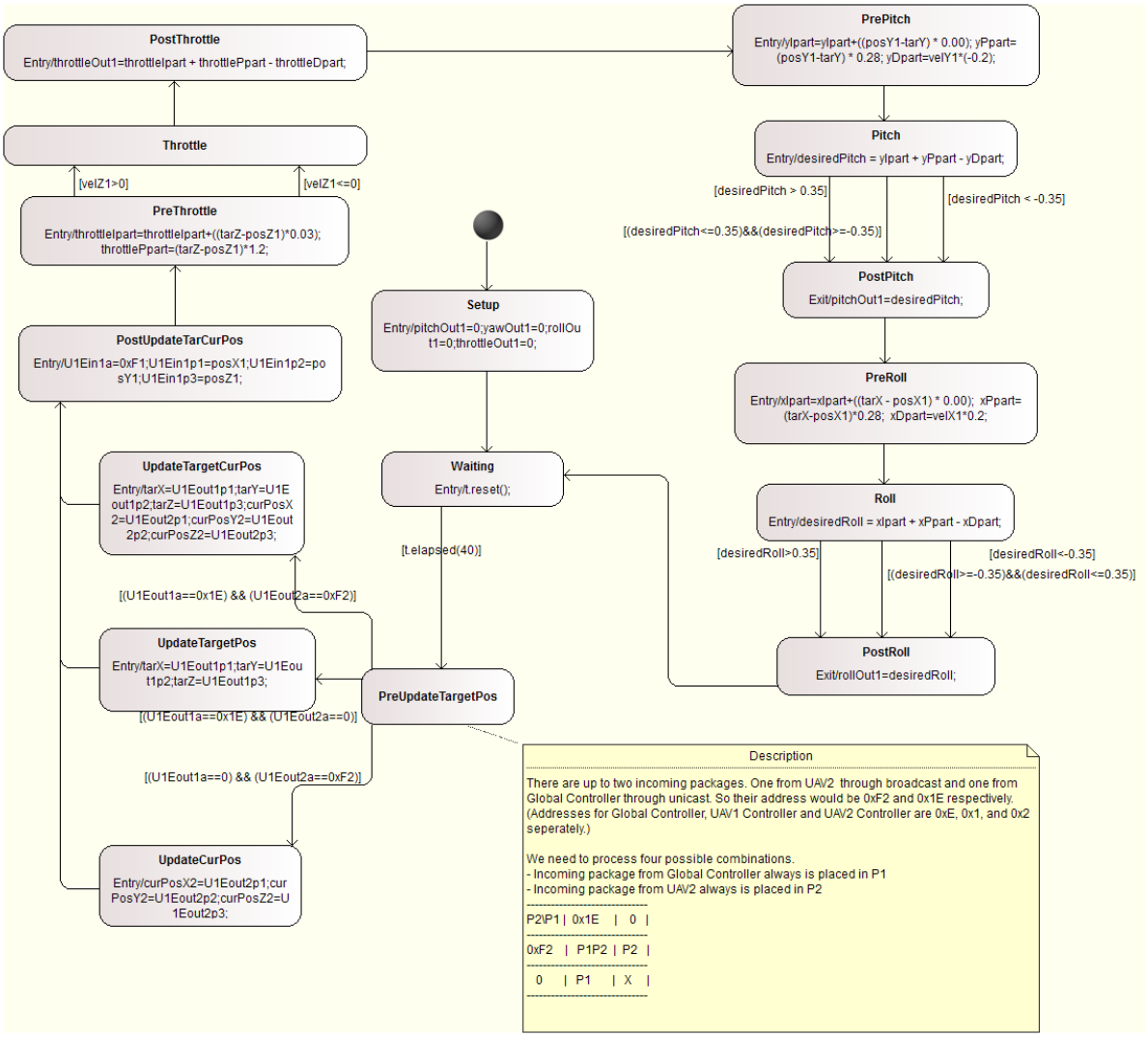


Figure 97: State Machine Diagram of UAV_CTRL

TestEnvironment We use a state machine diagram of a block *TESim* in *TestEnvironment* to simulate input sequences of target positions to *SystemUnderTest*, which is showed in Figure 99.

The target positions of *UAV1* and *UAV2* specified in the state machine diagram are illustrated in Figure 100 and Figure 101.

Model Checking We set "BMC steps" to 50. The properties below are checked to hold in 50 steps.

P0 Livelock property by the **Check Mode** function on RT-Tester.

```
Check static model semantics ...done.
- IMR.SystemUnderTest.EtherI... [PASS]
- IMR.SystemUnderTest.GLOBAL_CTRLI...
[LIVELOCK_CHECK] Could not get value for the lower bound
of the symbol IMR.SystemUnderTest.GLOBAL_CTRLI.curPosX2.
```

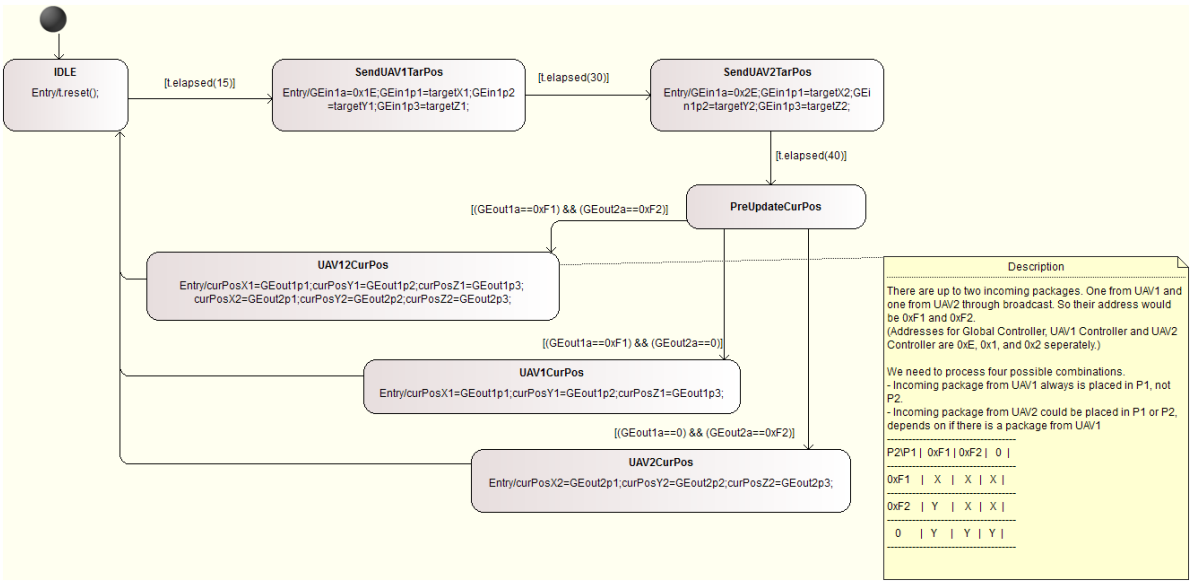


Figure 98: State Machine Diagram of GLOBAL_CTRL

- because in the test model, these variables are given a "RESTRICTED" stereotype but minimum and maximum values are not specified,
- after removing this stereotype for each variable (except input and output variables), it is fixed and the check is passed.

```
Check static model semantics ...done.
- IMR.SystemUnderTest.EtherI...  [PASS]
- IMR.SystemUnderTest.GLOBAL_CTRLI... [PASS]
- IMR.SystemUnderTest.UAV1_CTRLI... [PASS]
- IMR.SystemUnderTest.UAV2_CTRLI... [PASS]
- IMR.TestEnvironment.tesim... [PASS]
```

P1 *Global Controller* sends packages only to *UAV1 Controller* or *UAV2 Controller*.

```
Globally ([IMR.out11a != 33] && [IMR.out11a != 34] &&
    [IMR.out21a != 17] && [IMR.out21a != 18])
```

P2 All controllers could be in `Idle` at the same time.

```
Finally ([IMR.SystemUnderTest.End1I.End.Idle
    && IMR.SystemUnderTest.EtherI.Ether.Idle
    && IMR.SystemUnderTest.End2I.End.Idle])
```

P3 A broadcast package would never be received by its source port.

```
Globally ([IMR.out11a != 241] && [IMR.out21a != 242])
```

A Manual Implementation of SUT This SUT is composed of four modules (*Global Controller*, *Ether*, *UAV1 Controller*, and *UAV2 Controller*). And each of them corresponds to one controller in the test model and is wrapped in its own FMU in testing.

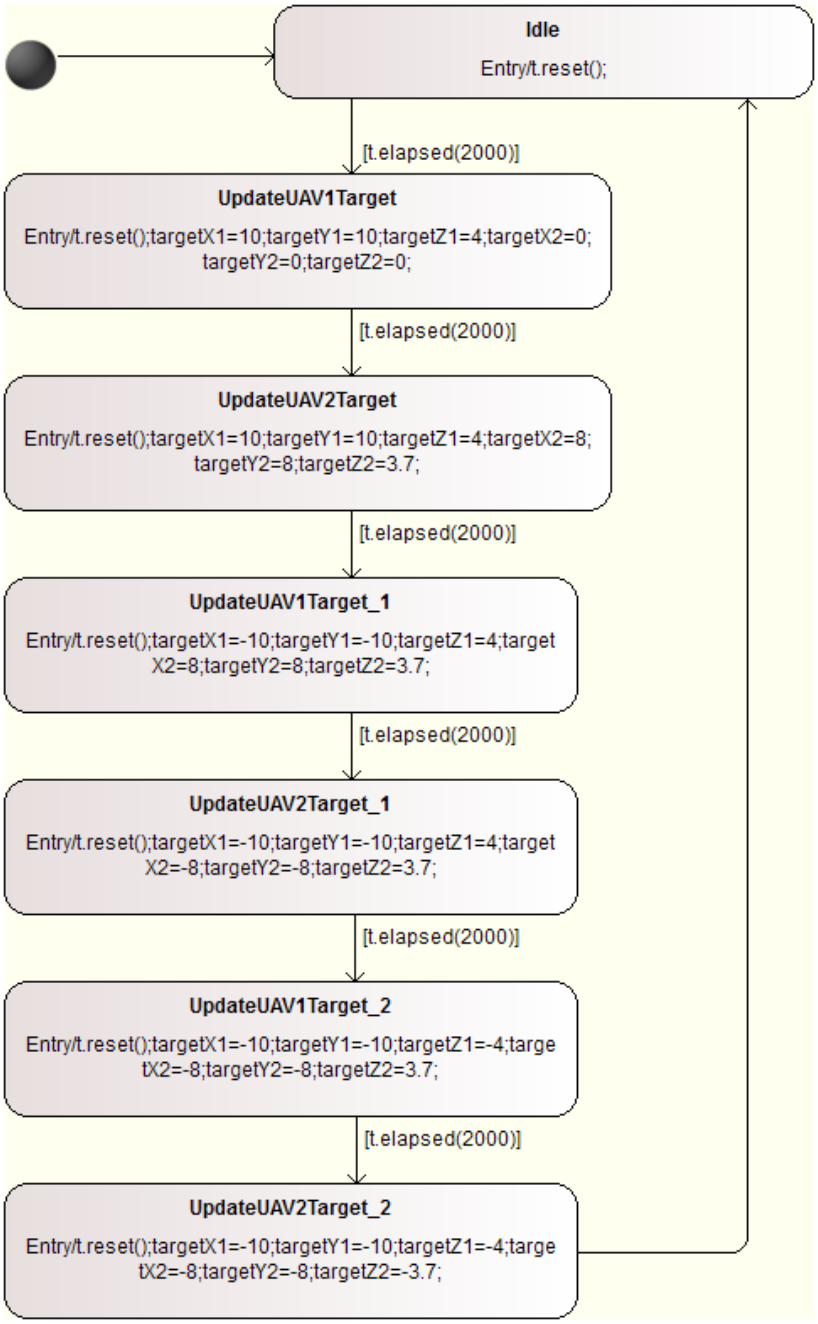


Figure 99: State Machine Diagram of *TESim*

Test Results Two tests are carried out for this study according to the configuration whether continuous *UAV* model is included in the test or not. One configuration is to connect all outputs from the test procedure *TP-00* to inputs of SUT (*UAV* in *TestEnvironment* is simulated), and another is to connect SUT's outputs to inputs of two *UAVs* (20-sim models) as well as inputs of *TP-00* and connect outputs of two *UAVs* to inputs of SUT. In the second configuration, the test shall be `FAIL` because inputs to SUTs are partially from *UAVs*, rather than all from *TP-00*. Therefore, expected results are different from actual results. But this configuration provides a more realistic Co-Simulation result because the physical model is included in Co-Simulation. And its result could provide further analysis for SUT.

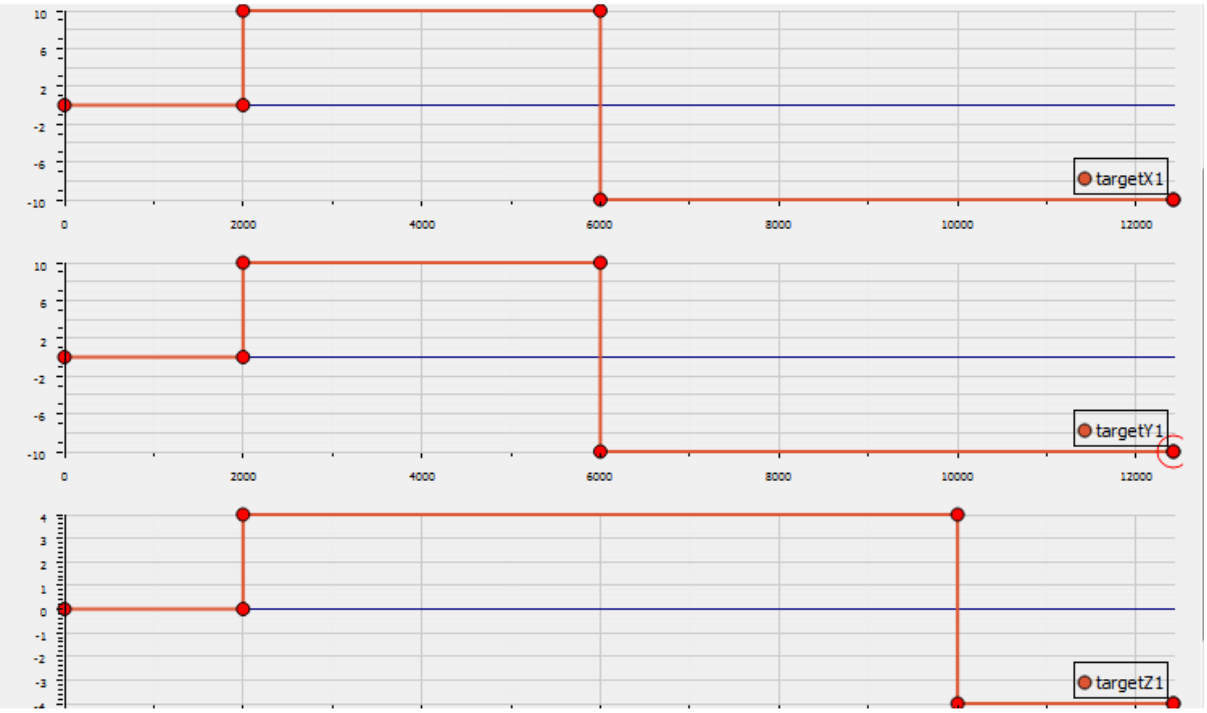


Figure 100: Simulated Target Positions for *UAV1*

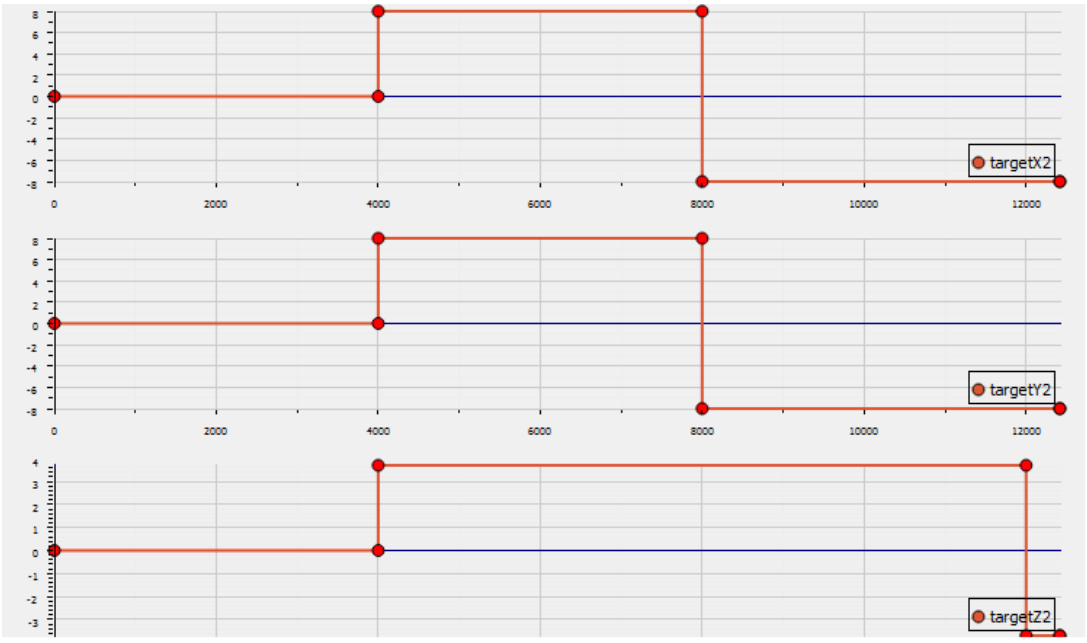


Figure 101: Simulated Target Positions of *UAV2*

Test Goals The test goal is defined by a user defined test case through the LTL formula: $Finally[\_timeTick \geq 25000]$. The *Admissible Error* and *Latency* in "signalmap.csv" are set to 0.0000001 and 5 ms respectively.

Co-Simulation without UAVs This test configuration is used to check if the SUT is a right implementation of the test model. The connection of five FMUs is configured as shown in Figure 102. This configuration does not include *UAVs*. Instead, current

positions and velocities from *TP-00* are connected to corresponding ports in *UAV Controllers*, and the UAV control signals (*throttle*, *roll*, *yaw*, and *pitch*) are connected from *UAV Controllers* to *TP-00*. Therefore, *TP-00* has all outputs and inputs connected to the SUT. Finally, test results are able to check if behaviour between the SUT and the test model (in *TP-00*) is consistent. Our test with used defined test case shows that all test cases are either `PASSED` or `INCONCLUSIVE`, and no `FAIL`.

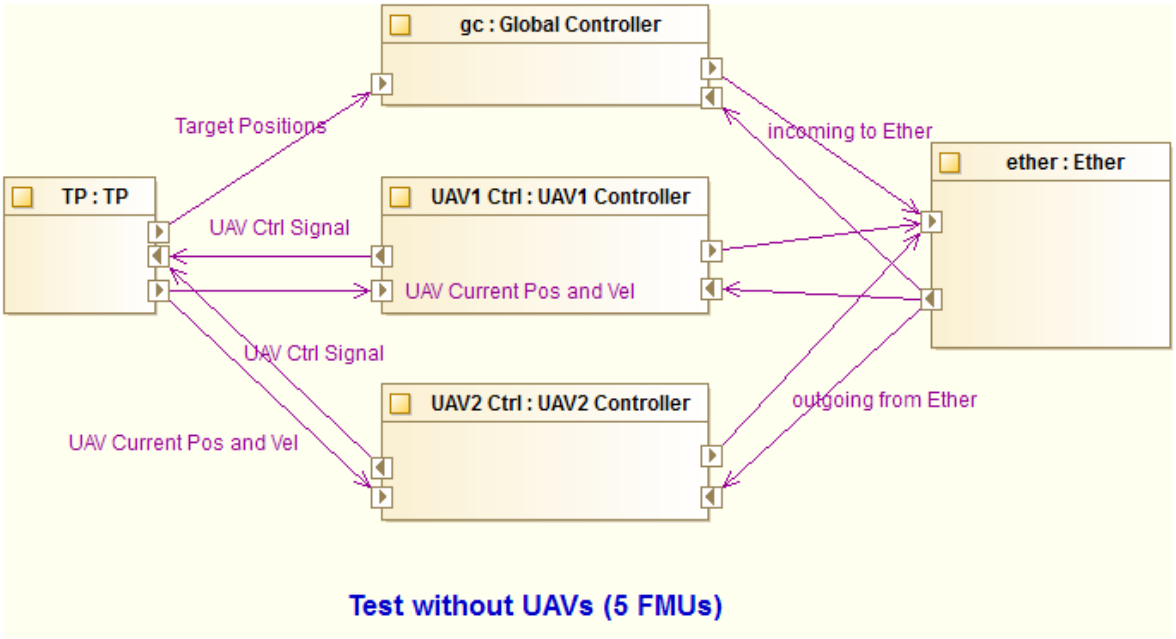


Figure 102: Connection for Test without UAVs

Co-Simulation with UAVs This test configuration is not intended for checking SUT against the test model, but for Co-Simulation with physical *UAVs* (20-sim models). The connection of seven FMUs is configured as shown in Figure 103. There are two *UAVs* and each of them is connected to corresponding *UAV Controller*.

Current positions and velocities to *UAV Controllers* are from *UAVs* and those from *TP-00* are left unconnected. But we still connect UAVs control signals from *UAV Controllers* to both *TP-00* and *UAVs*. This test is not able to check behaviour of the SUT against the test model because input sequences in expected results from *TP-00* and actual results from SUT are different. The test results show the RT Tester reports many *FAILs*, almost all FAILs after initial match with zero.

However, we can use this test configuration with Multi-model on the INTO-CPS application to co-simulate the UAV swarm in a context that all target positions are given from the test model by a state machine diagram of TestEnvironment.

Co-Simulation results for *UAV1* are displayed in Figure 104. This chart illustrates target positions in the X and Z axes (blue and orange respectively), current positions and velocities in the X and Z axes (green-yellow, grey, pink, and brown), and control signals from UAV controller (*throttle* - green, *pitch* - red, and *roll* - purple). Several observations can be seen from this chart.

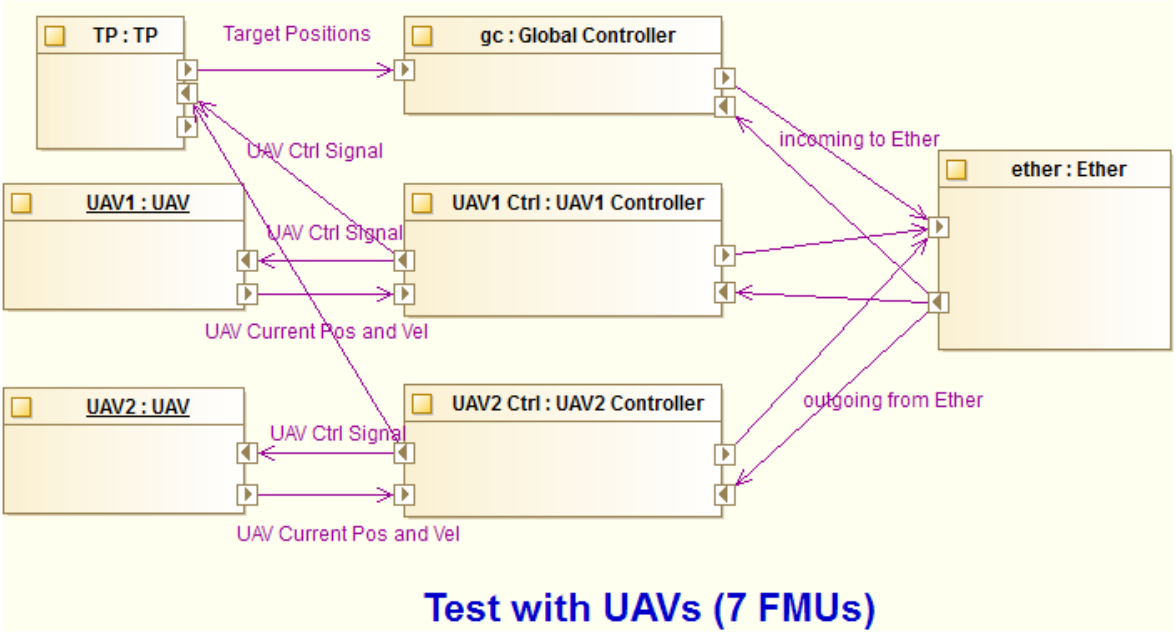


Figure 103: Connection for Test with UAVs

- basically, current position ($posX1$ - green-yellow) of *UAV1* follows its target position ($curPosX1$ - blue) though there is a delay,
- current position ($posZ1$ - grey) of *UAV1* follows its target position ($curPosZ1$ - orange) when it is larger than 0, but current position won't go below 0. So *UAVs* might have this restriction,
- velocities in the X and Z axes reflect movement of the *UAVs* in the axes as well,
- and the changes of positions, directions and velocities are also driven by the changes in control signals (*throttle*, *picth*, and *roll*).

Co-Simulation results for *UAV2* is displayed in Figure 105. This chart illustrates similar observations as Figure 104.

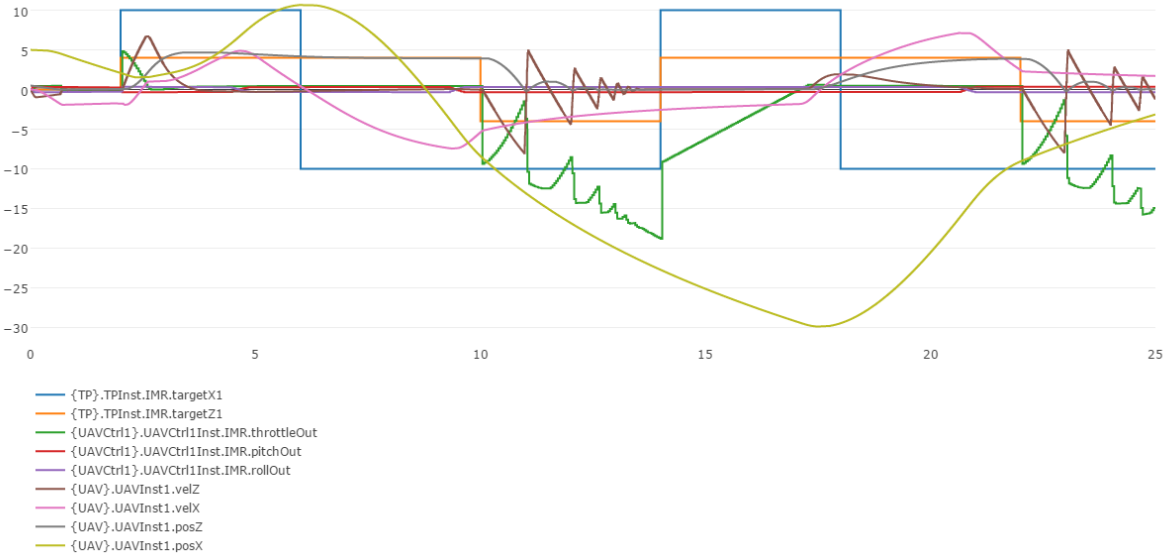


Figure 104: Co-Simulation Results of *UAV1*

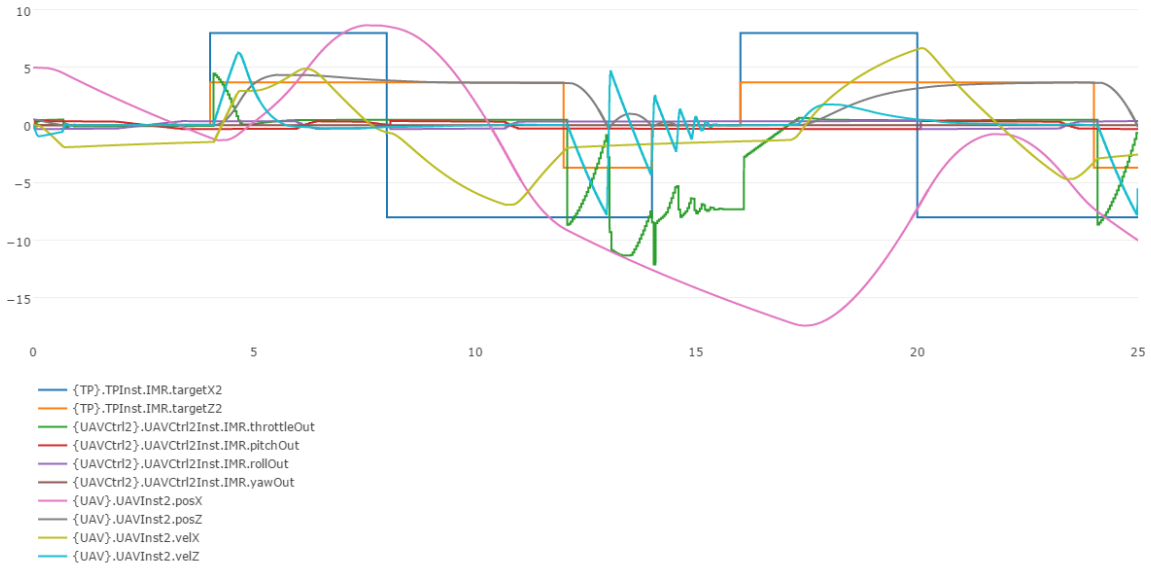


Figure 105: Co-Simulation Results of *UAV2*

10 Autonomous Vehicle

10.1 Example Description

This pilot study is an example of a vehicle driving a route defined from a set of way points. The multi-model consist of a steering controller and a model representing the dynamics of a vehicle. The steering controller contains the desired route of the vehicle. The vehicle is steered on the front wheels.

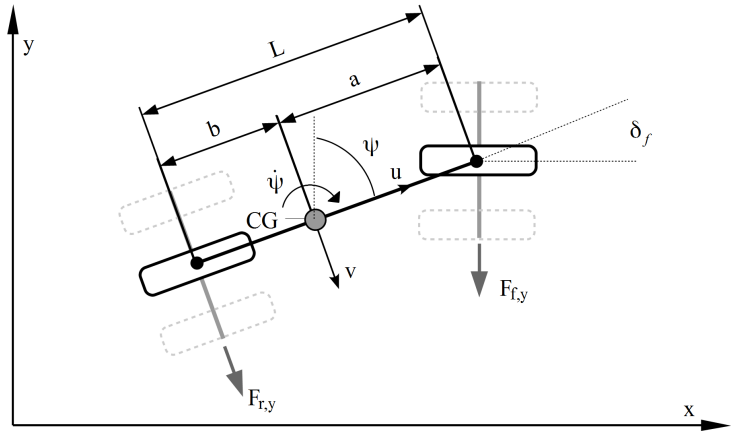


Figure 106: Overview and notation of the vehicle

10.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-autonomous_vehicle` in the *master* branch. The autonomous vehicle pilot study is an example of a vehicle driving through a path of specified way points.

The folder `FMUs` contains `vehicle.fmu` and `steering_controller.fmu`. The folder `Models` contains the VDM-RT project of the steering controller and the 20sim model of the dynamic description of the vehicle, `BicycleModel.emx`. The folder `Multi-models` contains the multi-model definition.

To run a simulation, open the INTO-CPS application, expand the `basic-sim` folder and open `Simulation-1`. Launch the COE and hit 'Simulate'. By default, the simulation integrates with a fixed time step size of 0.1 second for 100 seconds.

After a finished simulation, visualize the trajectory by running the file `plot_trajectory.py` using Python 2.x. That illustrates the trajectory of the latest simulation in the folder `\Multi-models\simulation-1`.

10.3 INTO-CPS SysML profile

The autonomous vehicle study SysML model is constructed using the the INTO-CPS profile. The Architecture Diagram of the co-simulation is shown in Figure 107,the Connection Diagram is shown in Figure 108. In the *RouteFollowingSystem* contains the *vehicle* component and component *Controller*. Position and orientation of the vehicle is passed between the two components. From *Vechicle* the current position is passed to the controller in where the desired orientation is determined based on the route and converted to a steering angle `delta_f`.

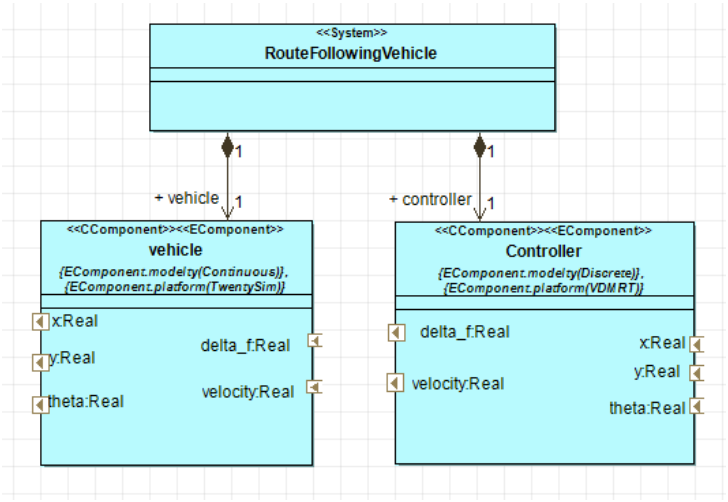


Figure 107: Architecture Diagram

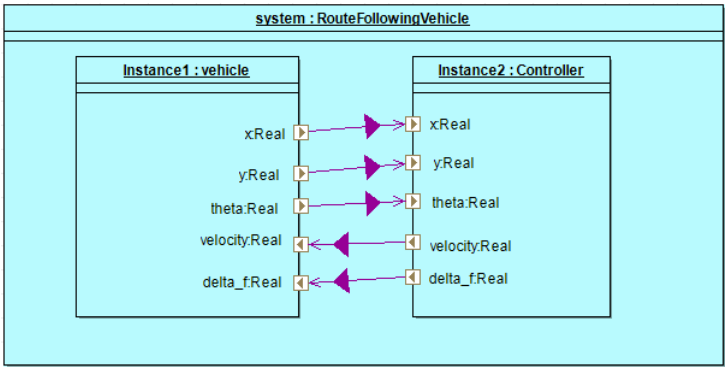


Figure 108: Connection Diagram

10.4 Multi-model

The parameters of the simulation can be changed in the INTO-CPS application or through the `mm.json` file.

10.4.1 Models

The SysML model above shows that the multi-model consists of two separate models. A 20sim model of the dynamic behaviour of the vehicle and a steering controller that keeps the vehicle on the specified path.

BicycleModel.emx The 20-sim model of the *vehicle* component, shown in Figure 109, contains several components.

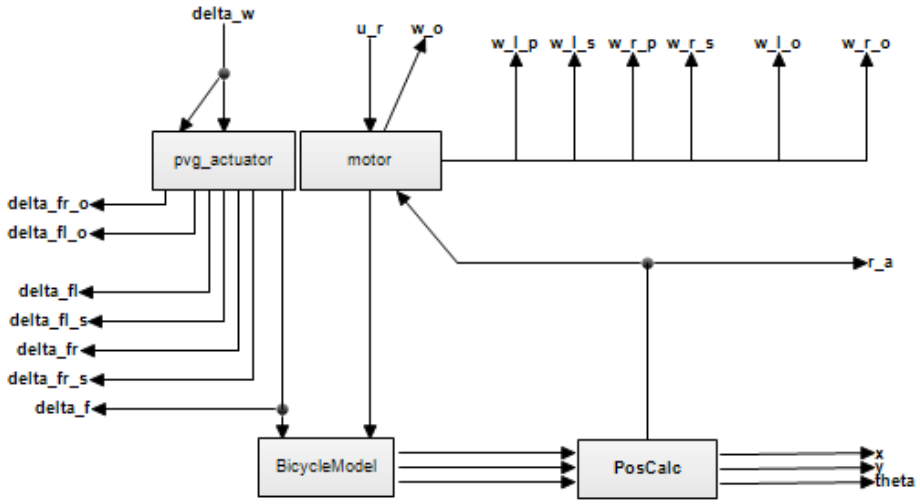


Figure 109: 20-sim vehicle model

The actuation block, `pvg_actuator` converts the desired steering angle from the controller to an steering angle compliant with the mechanical system. The actuation is shown in Figure 110. `delta_fl_w` and `delta_fr_w` are the desired steering angles. `delta_f` is the angle applied in the vehicle dynamics calculations. `delta_f` is calculated as the average of the two angle components from left and right side.

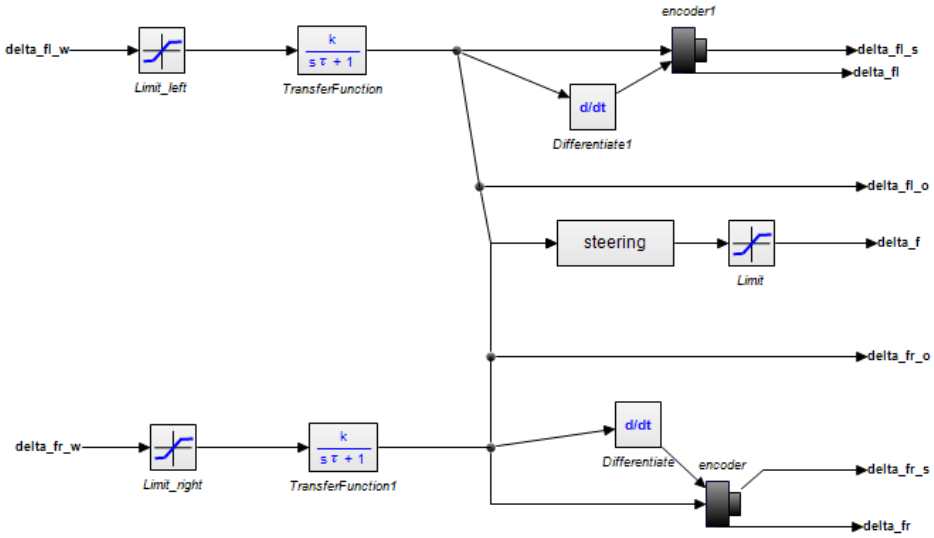


Figure 110: Actuation system

The `BicycleModel` components holds the calculations of the dynamic bicycle model. The `BicycleModel` is shown in Figure 111. The inputs to the model are the the forward velocity of the vehicle `u_r` and the `delta_f` which is the steering angle as shown in Figure 106. The outputs are the velocity components in x and y direction and yaw-rate `d_x_v`, `d_y_v` and `r`.

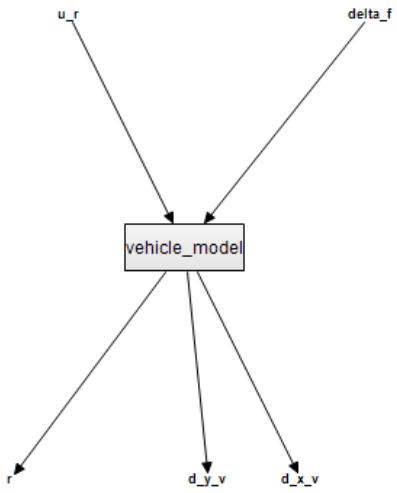


Figure 111: BicycleModel

The `vehicle_model` contains the following initial equations determining the weight on the wheels, the position of the centre of gravity (CG) and the moment of inertia as shown in Figure 112. An important feature of vehicle dynamics is the estimation

```
initialequations
    L = cg_a+cg_b;              // Wheel base
    a = L-(cg_b-dist_b);        // Distance from CG to front axel
    b = L-(cg_a+dist_b);        // Distance from CG to rear axel

    I_zz = (m/4)*((a+b)^2);     // Moment of inertia
    W_f = (m*g)*(b/L);          // Weight on the front axel
    W_r = (m*g)*(a/L);          // Weight on the rear axel
    h_s_f = (W_f*mu)/2;
    h_s_r = (W_r*mu)/2;
```

Figure 112: `initialequations`

of the lateral tire forces. These are in this case calculated with a non-linear tire model. The calculation of the slipangles and the lateral tire forces is determined as shown in Figure 113.

steering_controller is a VDM-RT project. The *System* class contains a *HardwareInterface* with RealPorts where the position and orientation of the vehicle is passed into the control calculations. The control signal is calculated as the product of a gain (p) and the angular misalignment of the current heading of the vehicle and the

```
equations   // Caclulating the slip angles and tire forces

   if abs(u_i) > 0.025 then
      if(u_i > 0) then
         alpha_f = ((v+a*r)/abs(u_i))-delta_f_i;
      else
         alpha_f = ((v+a*r)/abs(u_i))+delta_f_i;
      end;
      alpha_r = ((v-b*r)/abs(u_i));
   else
      alpha_f = 0;
      alpha_r = 0;
   end;

   f_y_f = -C_af*tan(alpha_f);       // Normal load on the front
   f_y_r = -C_ar*tan(alpha_r);       // Normal load on the rear

   // Applying the nonlinear tire model on the front
   if abs(f_y_f) > h_s_f then
      F_y_f = -mu*W_f*sign(alpha_f)*(1-((mu*W_f)/(4*C_af*abs(tan(alpha_f)))));
   else
   // Applying the linear tire model
      F_y_f = f_y_f;
   end;
   // Applying the nonlinear tire model on the rear
   if abs(f_y_r) > h_s_r then
      F_y_r = -mu*W_r*sign(alpha_r)*(1-((mu*W_r)/(4*C_ar*abs(tan(alpha_r)))));
   else
   // Applying the linear tire model
      F_y_r = f_y_r;
   end;

   if abs(u_i) > 0.025 then
      d_v = (-m*u_i*r+F_y_f+F_y_r)/m;      // Lateral acceleration
      d_r = (F_y_f*a-F_y_r*b)/I_zz;        // Yaw acceleration
   else
      d_v = 0;
      d_r = 0;
   end;
```

Figure 113: `Tire model`

desired heading towards the next way point. An illustration of the steering controller is shown in 114 where x and y is the current position. The route is specified

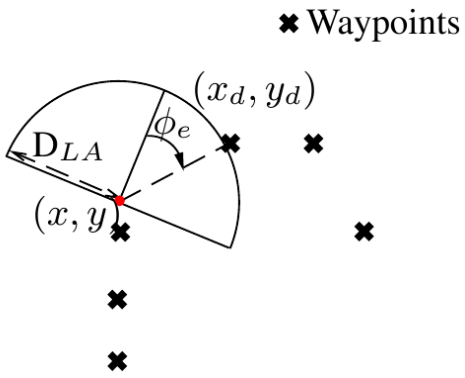


Figure 114: Sketch of steering control, (x, y) is the current position of the lawn mower, (x_d, y_d) is the desired heading, and ϕ_e is the deviation.

in a `.csv` file in the folder `...\Models\steering_controller`.

10.4.2 Configuration

One multi-models is defined: The model corresponds to the connection diagram shown in Figure 107.

10.5 Co-simulation

A co-simulation is performed through the INTO-CPS application for 180 seconds using a fixed time-step of 0.1 sec. The parameters of simulation is defined in the mm.json as :

```
"{fmu1}.Steering_control.speed_ref":  1,
"{fmu1}.Steering_control.look_ahead_dist":  1,
"{fmu1}.Steering_control.control_parameter":  1.
```

Running the simulation in the INTO-CPS application, the results are presented in Figure 115. The graphs `vehicle.x` and `vehicle.y` represents the relative position in meters compared to the initial point of the vehicle. `Steering_control.speed` is the forward velocity of the vehicle in m/s and `Steering_control.delta_f` is the steering angle in radians.

By plotting the x and y components, the trajectory reveals, as shown in Figure 116. The blue dashed line is the desired route, and the red graph is the simulated trajectory.

10.6 Analyses and Experiments

The performance, e.g. how well the vehicle track the desired route, can be investigated by changing the parameters in the `mm` file. Hereby, the influence of the control parameter p and the look ahead distance of the controller be evaluated under different velocities.

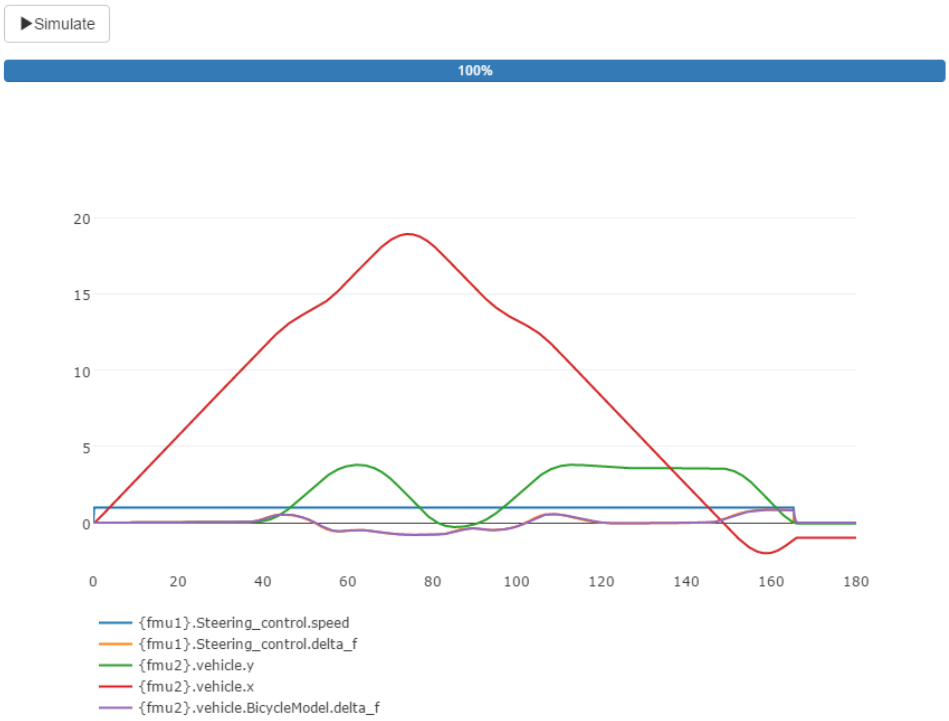


Figure 115: Simulation results in the the INTO-CPS app

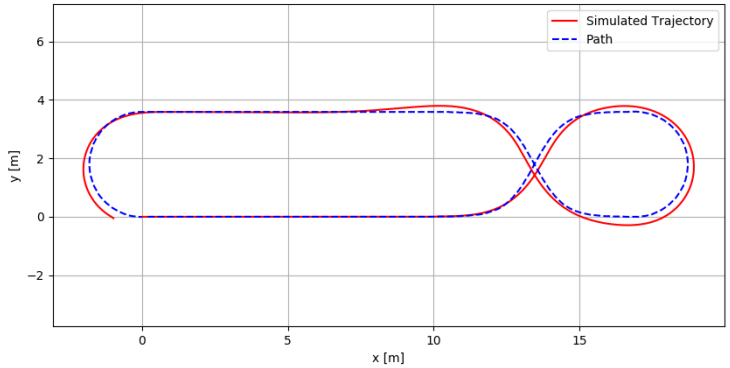


Figure 116: Simulation results in the the INTO-CPS application

10.6.1 Design Space Exploration

The DSE features are available for this pilot study. An objective script has been reused from the Line-following Robot pilot study, the *meanCrosstrackError*. For further information, see Section 5. In this example, a simple DSE is presented. The DSE evaluates the mean cross track error as a function of the velocity, gain and look-ahead parameter. In this example, an exhaustive search is applied for exploiting the design space in a number of predefined values. The parameters applied here is defined in the dse.json file as follows:

```
"{fmu1}.Steering_control.speed_ref":  [0.5, 1.0]
"{fmu1}.Steering_control.look_ahead_dist":  [0.5, 1.0]
"{fmu1}.Steering_control.control_parameter":  [0.5, 1.0],
```

giving a total of $2^3 = 8$ simulations. The results are ranked only based on the mean

cross track error as shown in Figure 117 from the `results.html` file. The column `meanCrossTrackError2` is forced to `1.0` since the designs has only been evaluated based on the mean cross track error. This is forces through the file `one_function.py` in the userMetricScripts folder.

Rank	meanCrossTrackError	meanCrossTrackError2	Steering_control.control_parameter	Steering_control.speed_ref	Steering_control.look_ahead_dist
1	0.0378415647323	1.0	1	0.5	1
2	0.0405744725804	1.0	1	0.5	0.5
3	0.0591024260981	1.0	0.5	0.5	1
4	0.0624341590598	1.0	0.5	0.5	0.5
5	0.0913897954769	1.0	1	1.0	0.5
6	0.107500796286	1.0	1	1.0	1
7	0.220008000086	1.0	0.5	1.0	0.5
8	0.354105008919	1.0	0.5	1.0	1

Figure 117: Ranking

Which illustrates that the best combination of parameters in terms of tracking the route is the combination of look-ahead and control parameter p is [1.0, 1.0] at a velocity of 0.5 m/s.

To run the DSE script perform the following:

- Download the DSE scripts using the download manager
- Open the terminal and write the following (Requires Python 2.x): `python Algorithm_selector.p Relative_path_to_dse_config Relative_path_to_coe_config`

11 Mass Spring Damper

11.1 Example Description

The mass spring damper pilot study demonstrates a stabilization technique for co-simulation. The study comprises two mass spring dampers as illustrated in Figure 118.

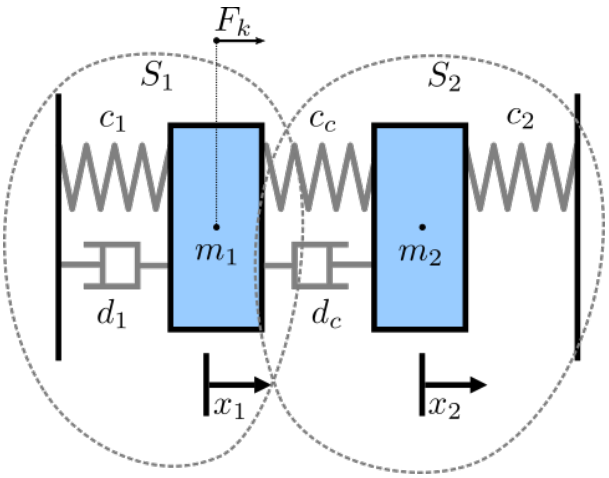


Figure 118: Illustration of two mass spring damper system

There are two simulators included in the study, each representing a mass spring damper system. The first simulator calculates the mass displacement and speed of *m1* for a given force *Fk* acting on mass *m1*. The second simulator calculates force *Fk* given a displacement and speed of mass *m1*.

By coupling these simulators, the evolution of the position of the two masses can be computed by a standalone numerical solver operating directly on the differential equations, to give the expected behaviour illustrated in Figure 119.

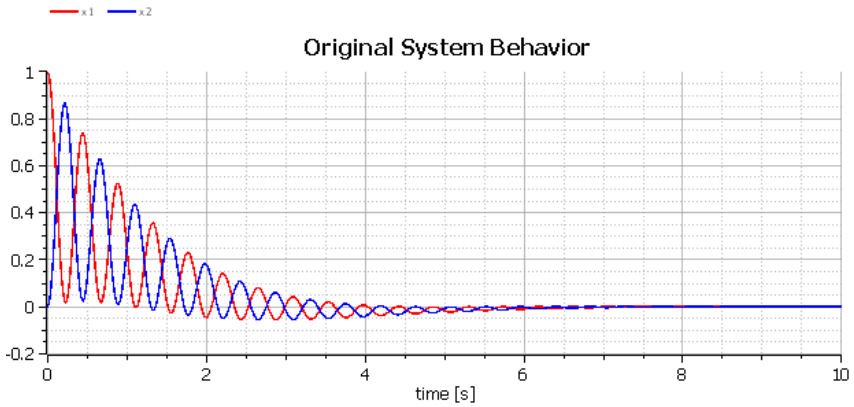


Figure 119: The expected behaviour of the modelled masses

11.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/INTO-CPS-Association/example-mass_spring_damper` in the *master* branch. There are several subfolders for the various elements: `FMUs` contains the various FMUs of the study; `Models` – contains the constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains two multi-model configurations, for both stable and unstable simulation.

The `case-study_mass_springer_damper` folder can be opened in the INTO-CPS application to run the various co-simulations as detailed in this document. To run a simulation, expand one of the multi-models and click 'Simulate' for an experiment.

11.3 Multi-model

11.3.1 Models

There are several models included in this pilot: a 20-sim model and three OpenModelica models.

MassSpringDamper.emx The 20-sim model of the two spring damper example, shown in Figure 120, comprises a sub-component for each spring damper. Each sub-component describes the dynamics of the spring damper, and the two sub-components are connected. *MassSpringDamper1* calculates displacement and velocity of a mass for a given force acting on it. *MassSpringDamper2* calculates the resulting force from a given a mass displacement and velocity.

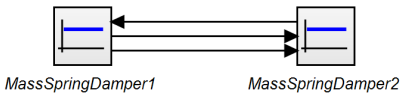


Figure 120: 20-Sim model of Mass Spring Damper System

MassSpringDamper1.mo *MassSpringDamper1.mo* describes the same dynamics expressed in *MassSpringDamper1* in the 20-sim model *MassSpringDamper.emx*.

MassSpringDamper2.mo *MassSpringDamper2.mo* describes the same dynamics expressed in *MassSpringDamper2* in the 20-sim model *MassSpringDamper.emx*.

MassSpringDampersScenario.mo *MassSpringDampersScenario.mo* links MassSpringDamper1.mo and MassSpringDamper2.mo and defines parameters for simulation.

11.3.2 Configuration

A single multi-model is defined:

mm The multi-model `mm` uses the *20-sim/MassSpringDamper1.fmu* and *20-sim/MassSpringDamper2.fmu* FMUs.

There are several design parameters defined the multi-model configuration.

11.4 Co-simulation

Two co-simulation variations are defined for the multi-model. Both simulation configurations have identical parameters, including a runtime of 10 seconds and use a fixed step size of 0.001 seconds. The only difference between the two configurations is the application of COE stabilisation (successive substitution).

Unstable This co-simulation configuration does not enable stabilisation. Co-simulating this multi-model configuration produces the output plotted in Figure 121, which shows two trajectories increasing indefinitely. This differs greatly from the expected output illustrated in Figure 119 where the two trajectories tend to 0 over time.

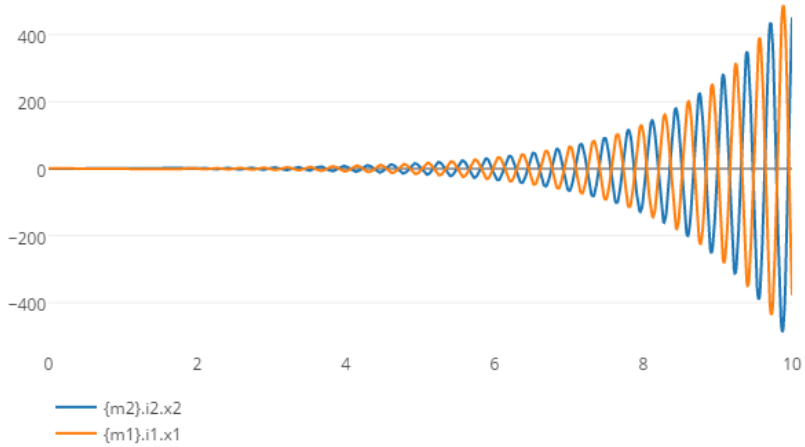


Figure 121: Co-simulation output of unstable configuration

Stable This co-simulation configuration does enable stabilisation. Co-simulating this multi-model configuration produces the output plotted in Figure 122, which shows the expected behaviour as illustrated in Figure 119.

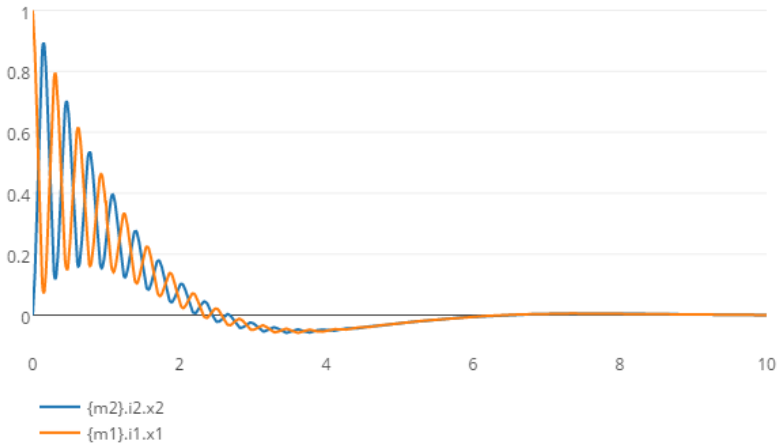


Figure 122: Co-simulation output of stable configuration

References

[BH17] Jörg Brauer and Miran Hasanagic. Implementation of a model-checking component. Technical report, INTO-CPS Deliverable, D5.3c, December 2017.

[FGP+15] John Fitzgerald, Carl Gamble, Richard Payne, Ken Pierce, and Jörg Brauer. Examples Compendium 1. Technical report, INTO-CPS Deliverable, D3.4, December 2015.

[FGP17] John Fitzgerald, Carl Gamble, and Ken Pierce. Method Guidelines 3. Technical report, INTO-CPS Deliverable, D3.3a, December 2017.

[Gam17] Carl Gamble. Comprehensive DSE Support. Technical report, INTO-CPS Deliverable, D5.3e, December 2017.

[GMB17] Carl Gamble, Oliver Möller, and Victor Bandur. Test automation module in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D5.3a, December 2017.

[GN16] Ivan Grujic and René Nilsson. Model-based development and evaluation of control for complex multi-domain systems: Attitude control for a quadrotor uav. Technical Report 23, Department of Engineering, Aarhus University, January 2016.

[IPG+12] Claire Ingram, Ken Pierce, Carl Gamble, Sune Wolff, Martin Peter Christensen, and Peter Gorm Larsen. Examples compendium. Technical report, The DESTECS Project (INFSO-ICT-248134), October 2012.

[MGP+17] Martin Mansfield, Carl Gamble, Ken Pierce, John Fitzgerald, Simon Foster, Casper Thule, and Rene Nilsson. Examples Compendium 3. Technical report, INTO-CPS Deliverable, D3.6, December 2017.

[PGP+16] Richard Payne, Carl Gamble, Ken Pierce, John Fitzgerald, Simon Foster, Casper Thule, and Rene Nilsson. Examples Compendium 2. Technical report, INTO-CPS Deliverable, D3.5, December 2016.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Nasa Formal Methods, Third International Symposium, NFM 2011*, pages 298–312, Pasadena, CA, USA, April 2011. NASA, Springer LNCS 6617.

[TN16] Casper Thule and René Nilsson. Considering Abstraction Levels on a Case Study. In Peter Gorm Larsen, Nico Plat, and Nick Battle, editors, *The 14th Overture Workshop: Towards Analytical Tool Chains*, pages 16–31, Cyprus, Greece, November 2016. Aarhus University, Department of Engineering. ECE-TR-28.

[Ver15] Verified Systems International GmbH, Bremen, Germany. *RT-Tester Model-Based Test Case and Test Data Generator – RTT-MBT: User Manual*, 2015. `https://www.verified.de/products/model-based-testing/`, Doc. Id. Verified-INT-003-2012.