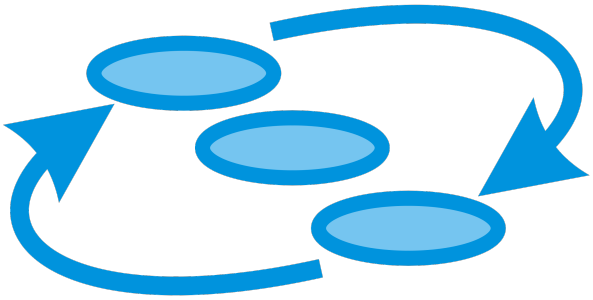
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Examples Compendium 2

Deliverable Number: D3.5

Version: 1.0

Date: December 2016

Public Document

Contributors:

Richard Payne, UNEW
Carl Gamble, UNEW
Ken Pierce, UNEW
John Fitzgerald, UNEW
Simon Foster, UY
Casper Thule, AU
Rene Nilsson, AU
Florian Lapschies, VSI

Editors:

Richard Payne, UNEW

Reviewers:

Peter Gorm Larsen, AU
Adrian Pop, LIU
Etienne Brosse ST

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	10-05-2016	Richard Payne	Draft structure of deliverable and initial reframing based on D3.4
0.2	16-06-2016	Richard Payne	Allocation of responsibilities and provisional dates for inclusion
0.3	01-11-2016	Richard Payne	Version for internal review
1.0	13-12-2016	Richard Payne	Final version with UAV And Turn Indicator pilots added and review comments addressed

Abstract

This deliverable 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 and of the models available for the study. The examples demonstrate the INTO-CPS tools developed during the first two years of the INTO-CPS project. The deliverable also lays out a roadmap for the final 12 months of case study and example development to test and demonstrate upcoming INTO-CPS technologies.

Contents

1 Introduction **7**

2 Single-tank Water Tank **9**
2.1 Example Description . . . 9
2.2 Usage . . . 9
2.3 INTO-CPS SysML profile . . . 9
2.4 Multi-model . . . 11
2.5 Co-simulation . . . 12
2.6 Analyses and Experiments . . . 12

3 Three-tank Water Tank **14**
3.1 Example Description . . . 14
3.2 Usage . . . 14
3.3 INTO-CPS SysML profile . . . 15
3.4 Multi-model . . . 16
3.5 Co-simulation . . . 18
3.6 Analyses and Experiments . . . 20

4 Fan Coil Unit (FCU) **26**
4.1 Example Description . . . 26
4.2 Usage . . . 26
4.3 INTO-CPS SysML Profile . . . 27
4.4 Multi-model . . . 28
4.5 Co-simulation . . . 29
4.6 Analyses and Experiments . . . 31

5 Line-following Robot **34**
5.1 Example Description . . . 34
5.2 Usage . . . 34
5.3 INTO SysML profile . . . 35
5.4 Multi-model . . . 37
5.5 Co-simulation . . . 42
5.6 Analyses and Experiments . . . 42

6 Turn Indicator **44**
6.1 Example Description . . . 44
6.2 Usage . . . 44
6.3 SysML . . . 44
6.4 Analyses and Experiments . . . 48

7 UAV **49**
7.1 Example Description . . . 49
7.2 Usage . . . 49
7.3 INTO-CPS SysML profile . . . 50
7.4 Multi-model . . . 52
7.5 Co-simulation . . . 53

8 Ether **54**
8.1 Example Description . . . 54
8.2 Usage . . . 54
8.3 Multi-model . . . 55
8.4 Co-simulation . . . 56
8.5 Analyses and Experiments . . . 56

9 Swarm of UAV **57**
9.1 Example Description . . . 57
9.2 Usage . . . 57
9.3 INTO-CPS SysML profile . . . 58
9.4 Multi-model . . . 59
9.5 Co-simulation . . . 62

10 Smart Grid **64**
10.1 Example Description . . . 64
10.2 Usage . . . 64
10.3 INTO-CPS SysML profile . . . 64
10.4 Multi-model . . . 66
10.5 Co-simulation . . . 71

11 Roadmap for Pilot Studies **72**
11.1 Future INTO-CPS Technology Demonstration Needs . . . 72

1 Introduction

This deliverable 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. The document concludes by laying out a roadmap for the final 12 months of case study and example development to test and demonstrate upcoming INTO-CPS technologies.

This deliverable is structured in different sections, each of which provides a brief introduction to each example model. The examples each illustrate 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 illustrates 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.

[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/`

- 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, 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 illustrates a smart grid multi-model. The final study is a preliminary study presenting only the architecture and constituent models in 20-sim and VDM-RT. FMUs have not yet been produced, therefore co-simulation has yet to be demonstrated.

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 multi-models.

	INTO-CPS Technology										
Multi-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			
Line-following Robot		x	x	x	x	x	x	x			x
Turn Indicator									x	x	
Single UAV		x	x		x	x	x				
Ether	x				x		x				
UAV Swarm	x	x	x		x	x	x				
Smart Grid	x	x	x		x	x					

Table 1: Overview of INTO-CPS technologies used for pilot studies

Section 11 presents a roadmap for the final 12 months of pilot case study development. We identify the various INTO-CPS technologies in production over the final year of the project that the pilots should seek to demonstrate.

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.

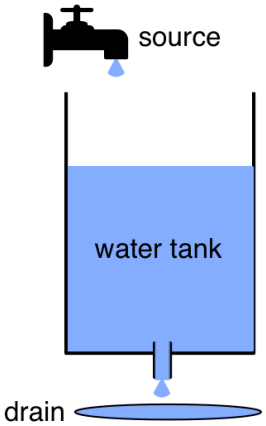


Figure 1: Overview of the single-tank water tank example

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/case-study_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.

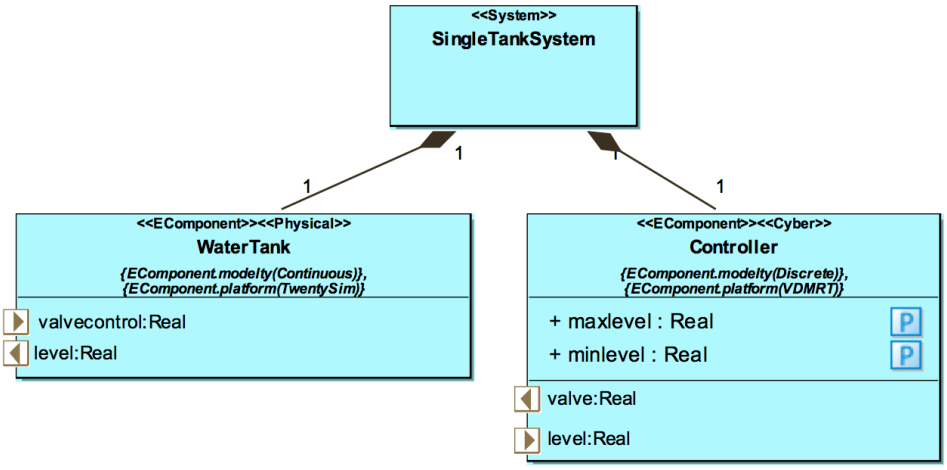


Figure 2: Architecture Structure Diagram defining the Single-tank Water Tank system composition

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.

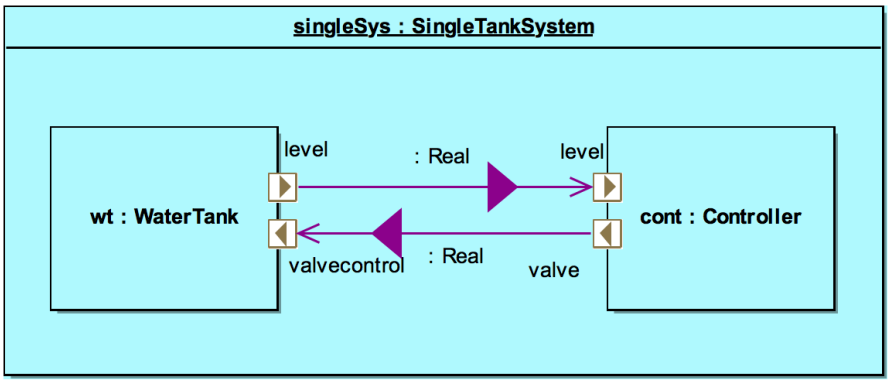


Figure 3: Connections Diagram defining the Single-tank Water Tank system connections

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.

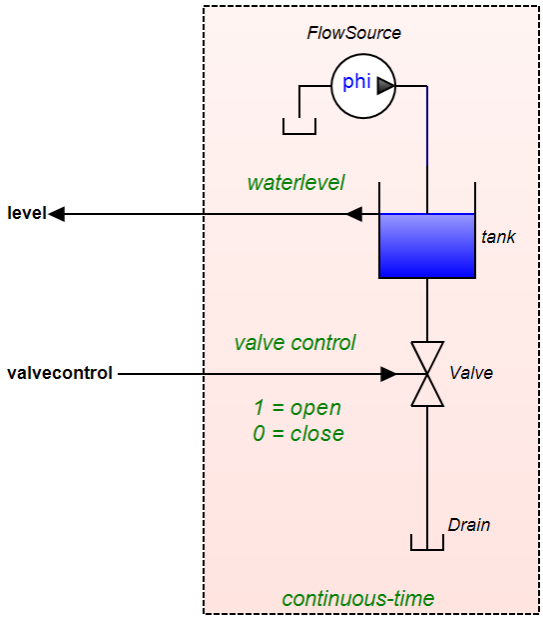


Figure 4: 20-sim Water Tank component

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:

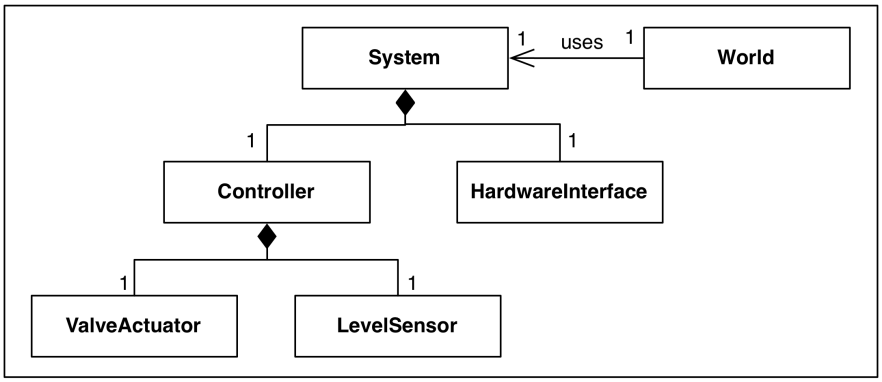


Figure 5: VDM-RT model architecture

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.

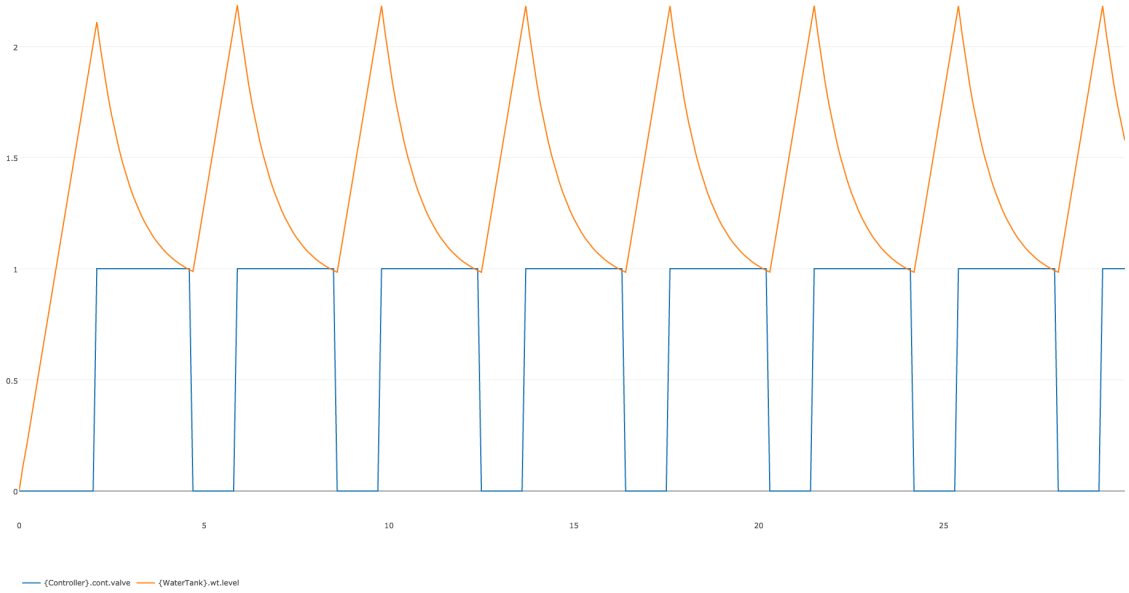


Figure 6: Co-simulation results for Single-tank Water Tank system

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.

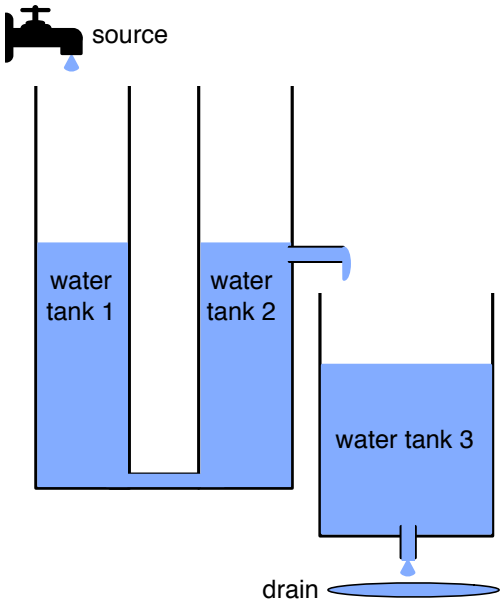


Figure 7: Overview of the three-tank water tank example

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/case-study_three_tank` 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.

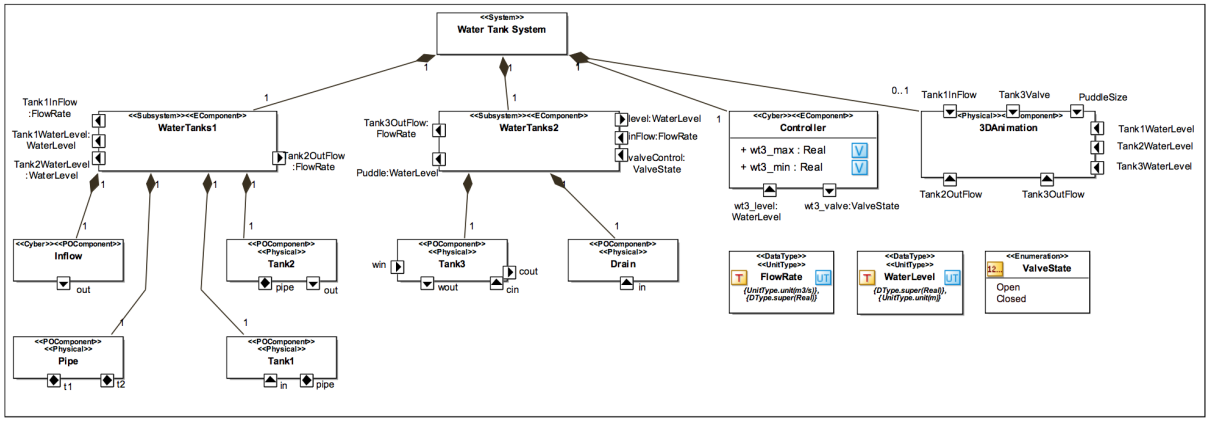


Figure 8: Architecture Structure Diagram defining the Three-tank Water Tank system composition

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.

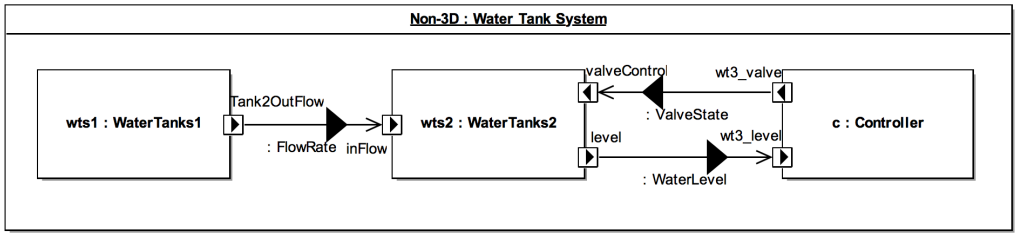


Figure 9: Connections Diagram defining the Three-tank Water Tank system connections

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.

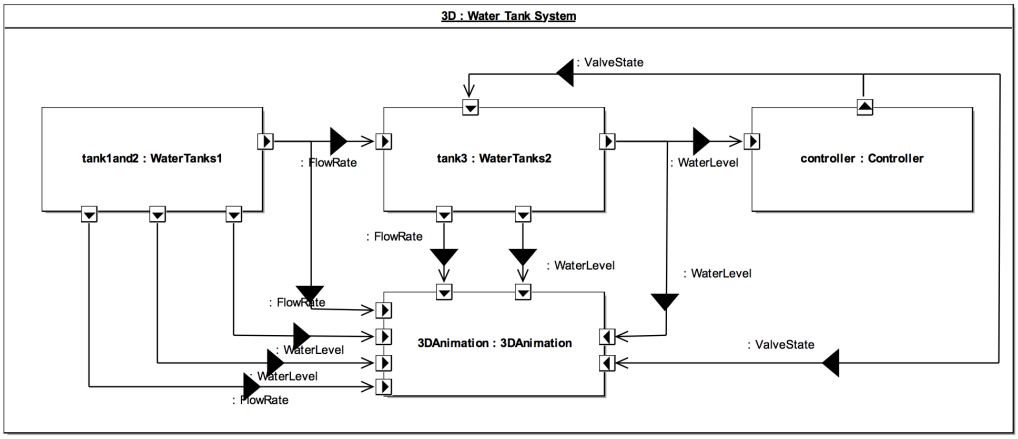


Figure 10: Connections Diagram defining the Three-tank Water Tank system connections and elements for visualisation

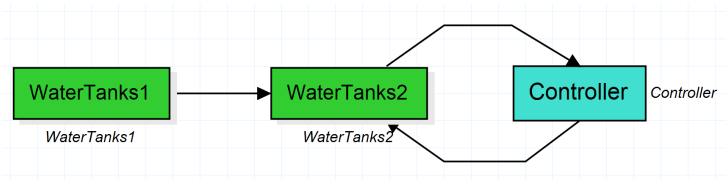
3.4 Multi-model

3.4.1 Models

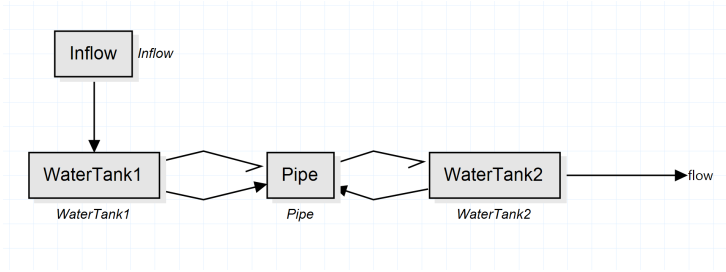
Given the ASD of the SysML model in Section 3.3, three (simulation) models are defined; two 20-sim subsystems and a VDM subsystem as shown in Figure 11(a).

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

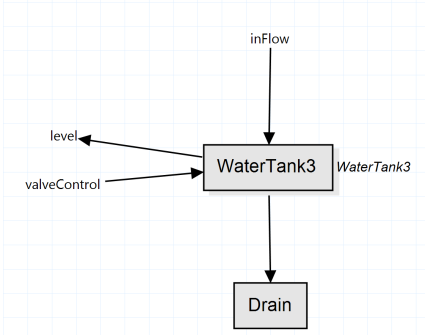
between tank 1 and 2 has a high frequency and amplitude, suggesting that splitting the two tanks would result in erroneous results when time steps are imposed in co-simulation.



(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.

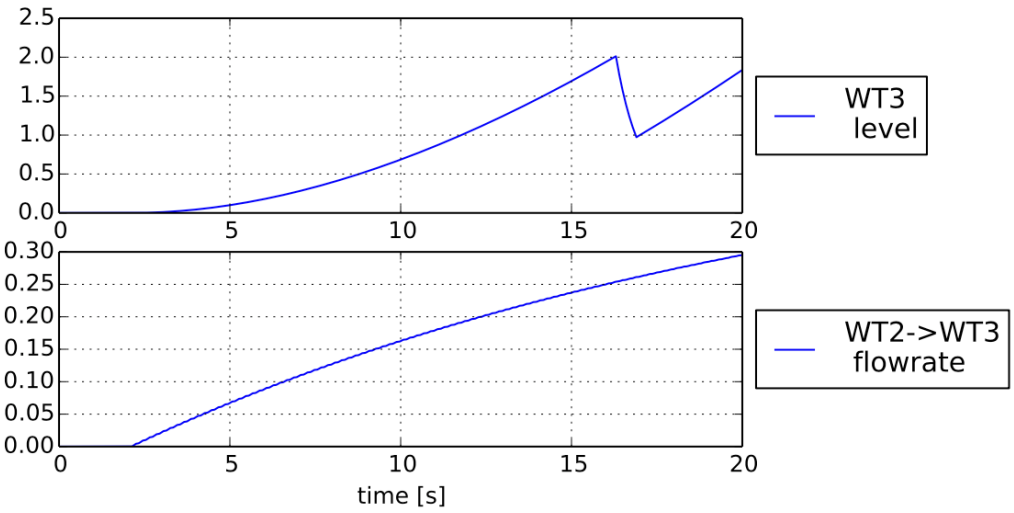


Figure 14: Simulation results using the INTO-CPS COE

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.

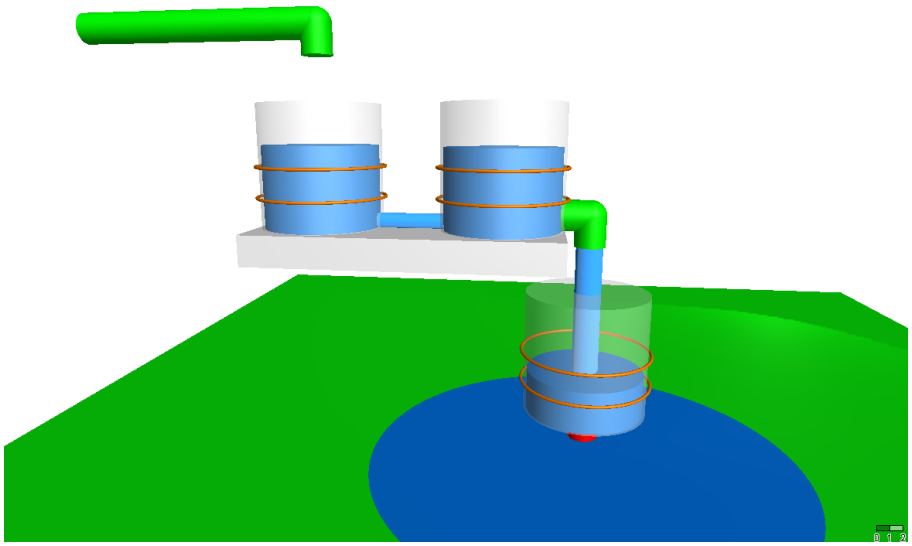


Figure 15: 3D visualisation of the Three-tank Water Tank system

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.

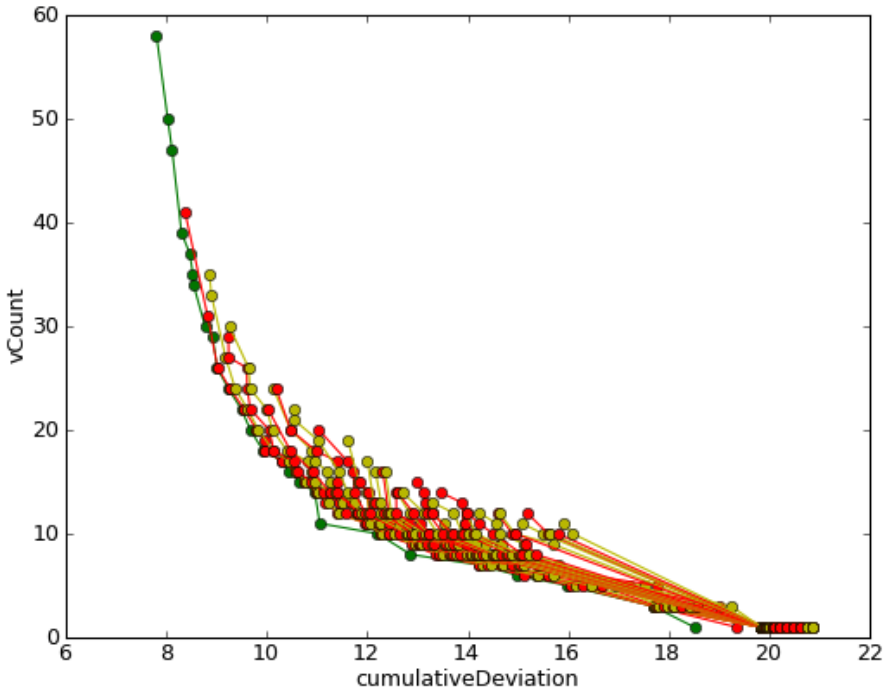


Figure 16: Design Space Exploration Pareto graph of the Three-tank Water Tank system

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 TE and SUT 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.

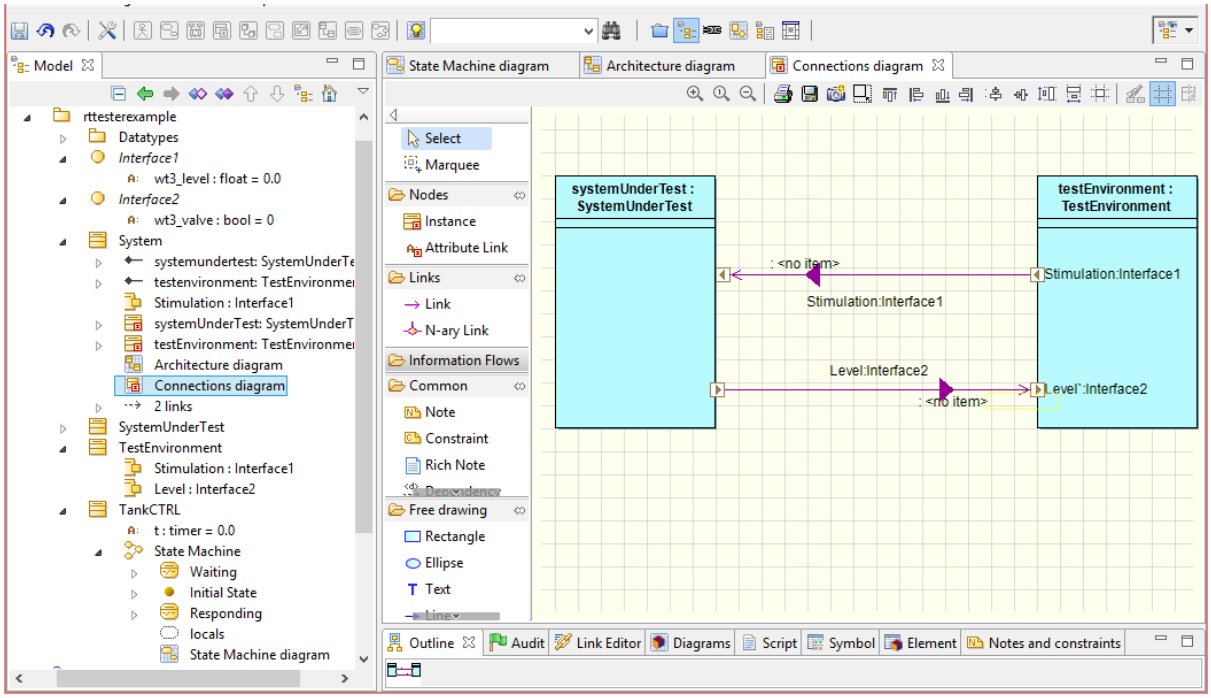


Figure 18: Overview of the three-tank test model in Modelio

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 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-Tester

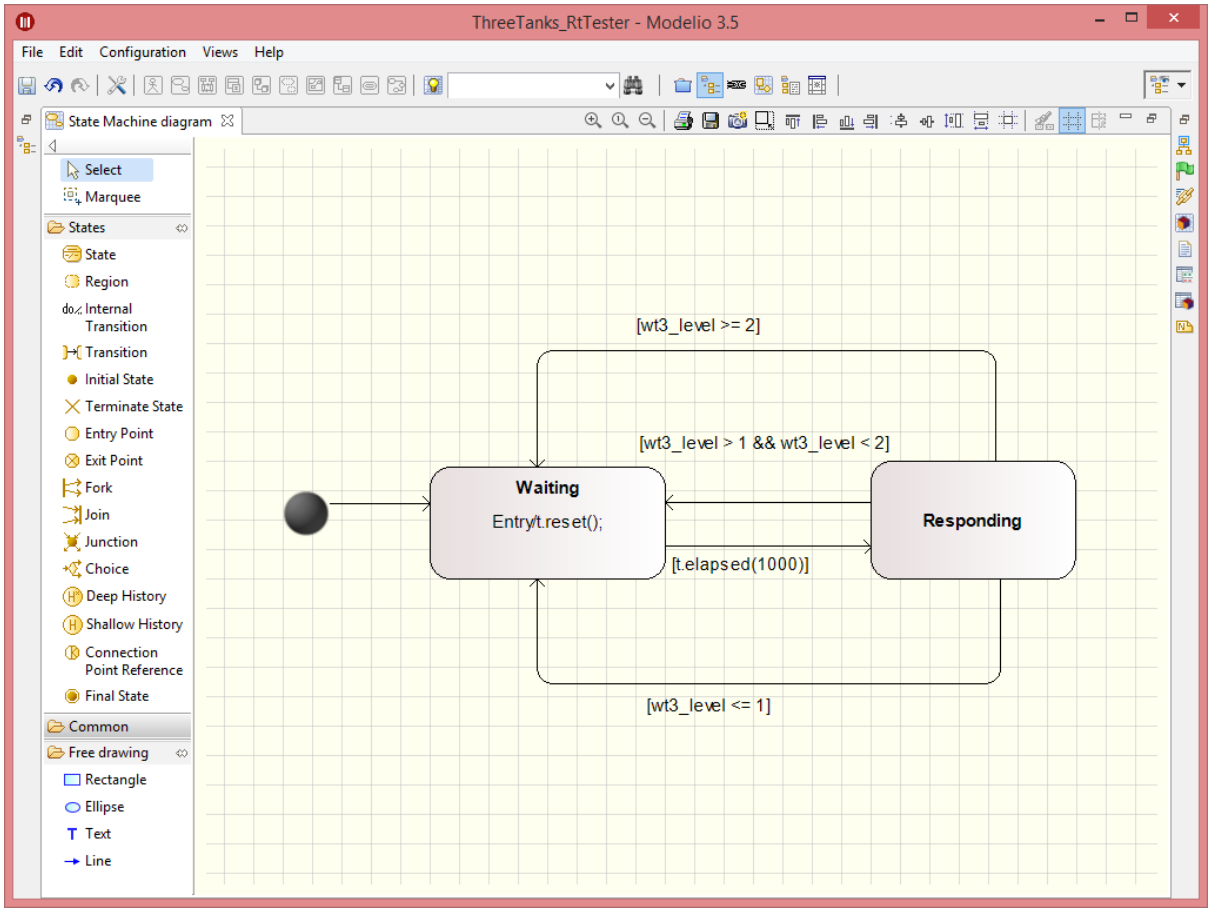


Figure 19: State machine for abstract behaviour of three-tank controller

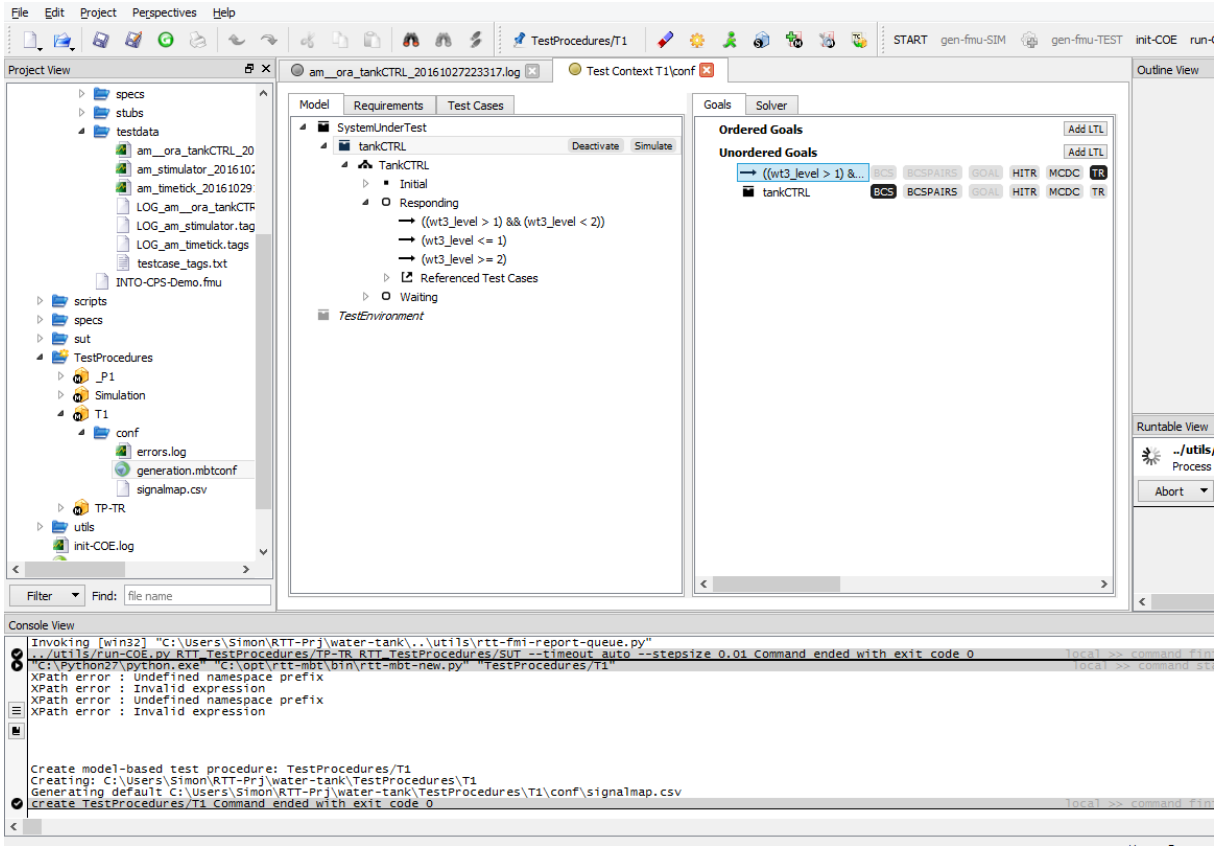


Figure 20: Configuring a test procedure

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

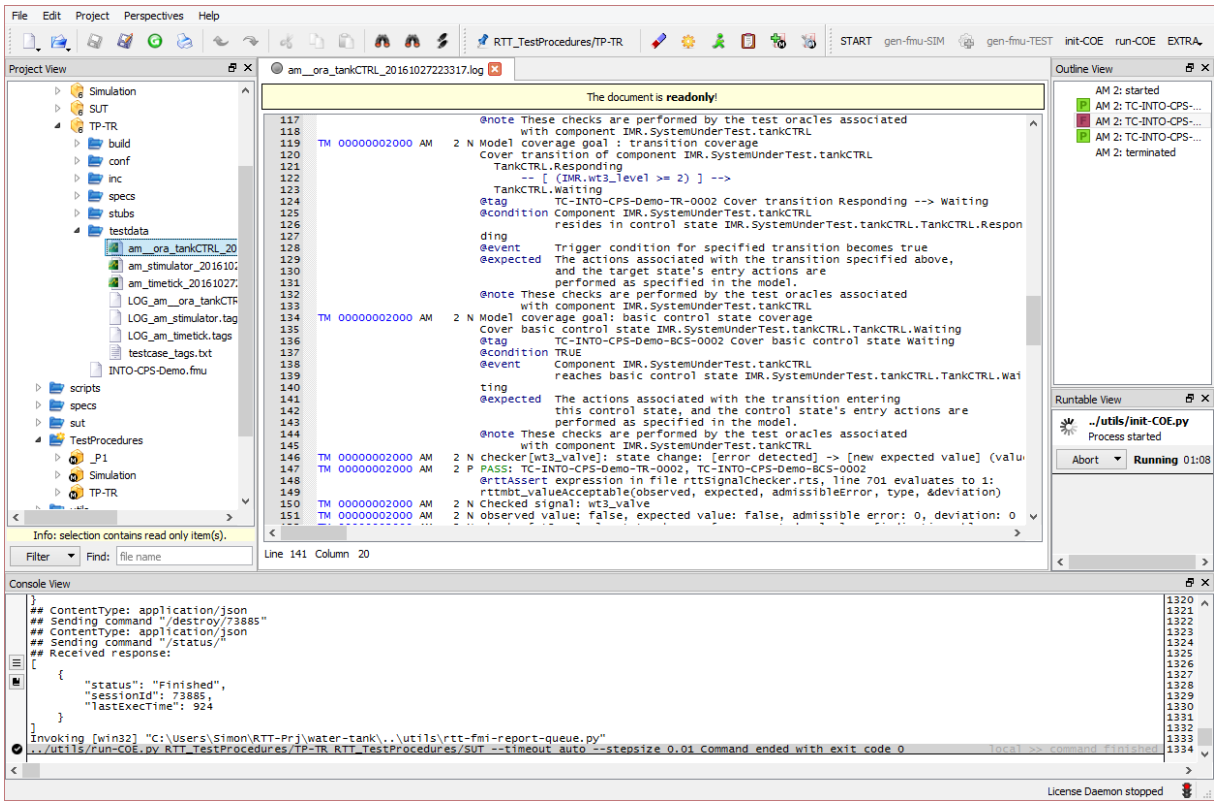


Figure 21: Test procedure output

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.

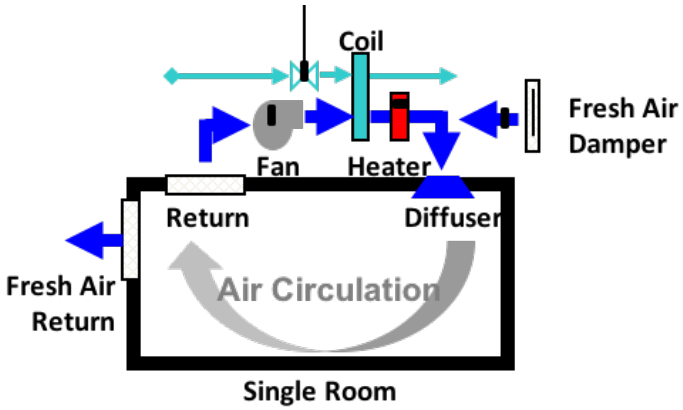


Figure 22: Overview of the fan coil unit (FCU) example

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/case-study_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.

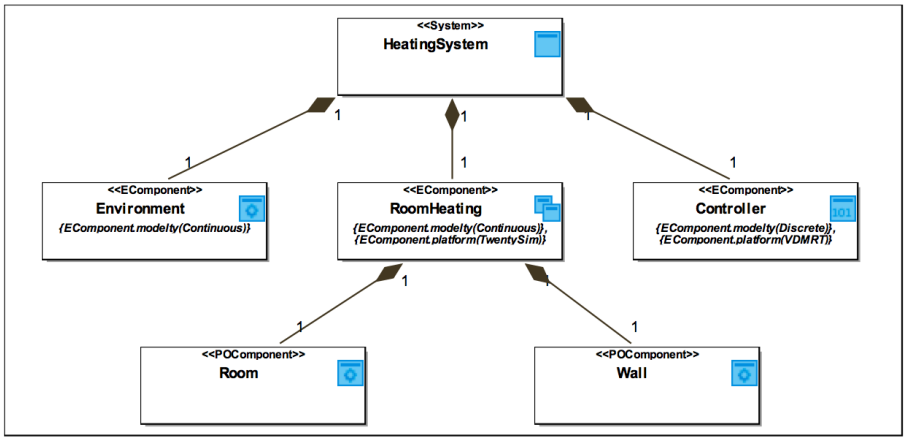


Figure 23: SysML Architecture Structure Diagram using INTO-CPS profile corresponding to baseline models

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*.

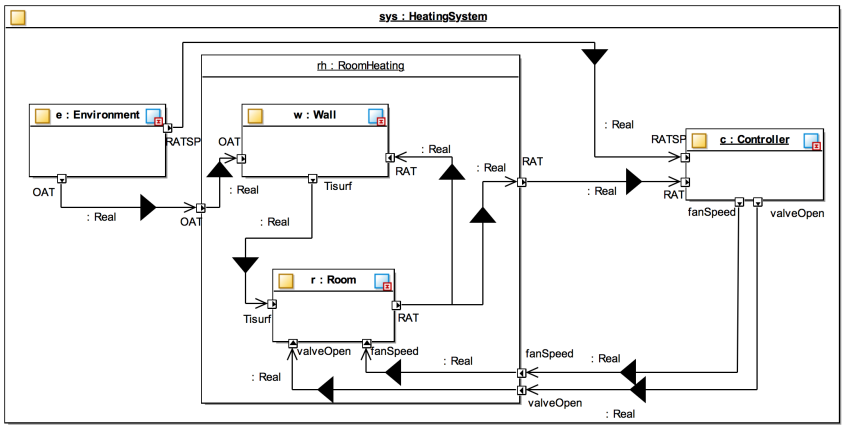


Figure 24: SysML Connection Diagram using INTO-CPS profile corresponding to baseline models

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.

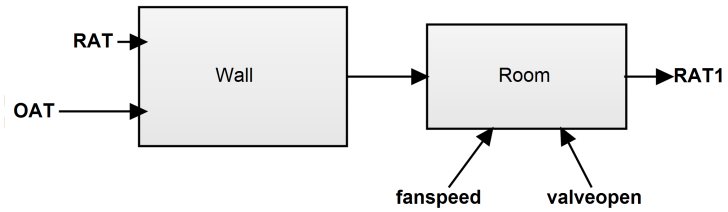


Figure 25: RoomHeating model

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.

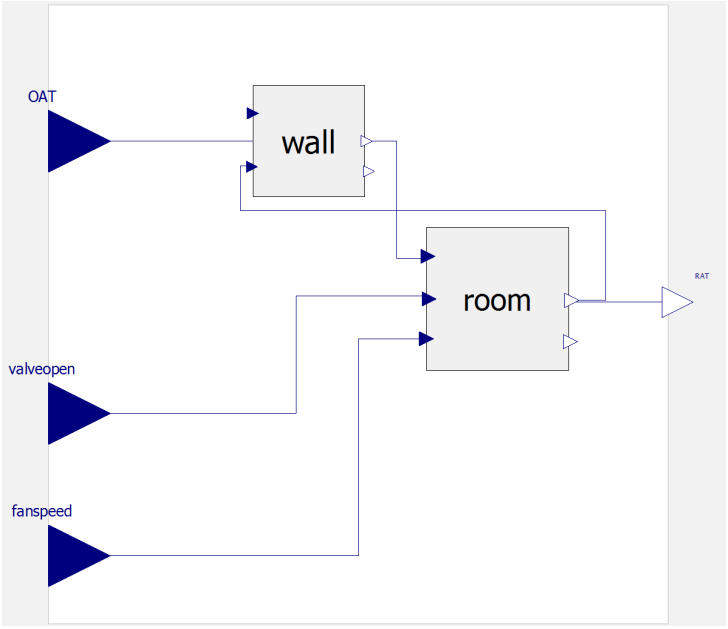


Figure 26: RoomHeating model

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-

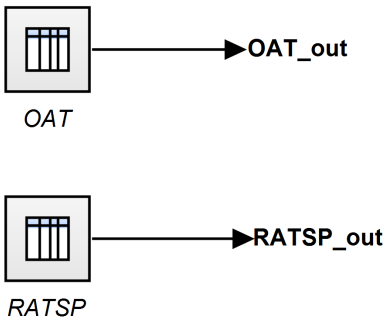


Figure 27: Environment model

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 3 FMUs and 5 connections. The 3 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)

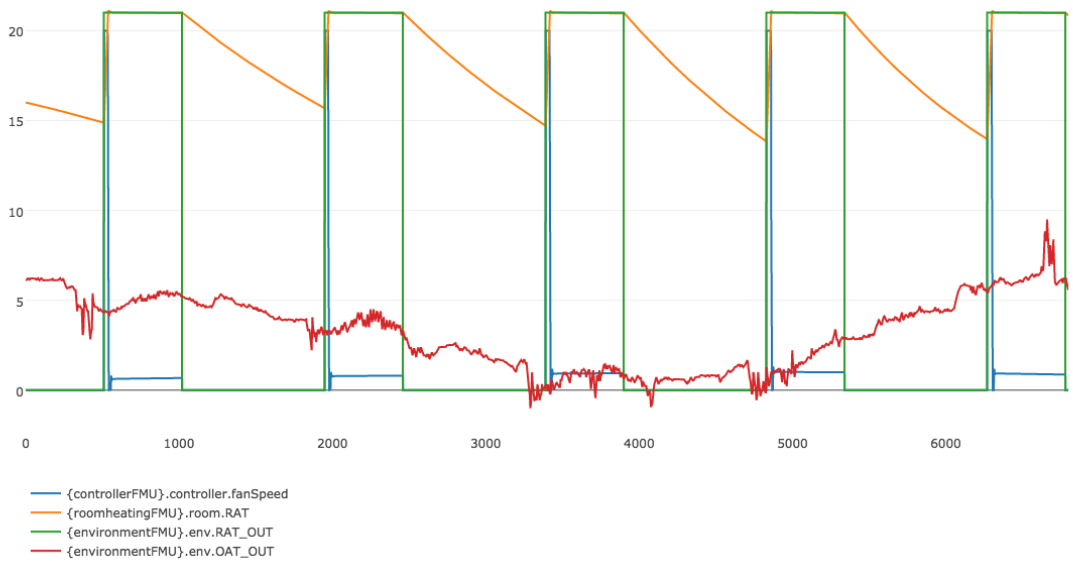


Figure 28: Co-simulation results as shown in lINTO-CPS application live stream

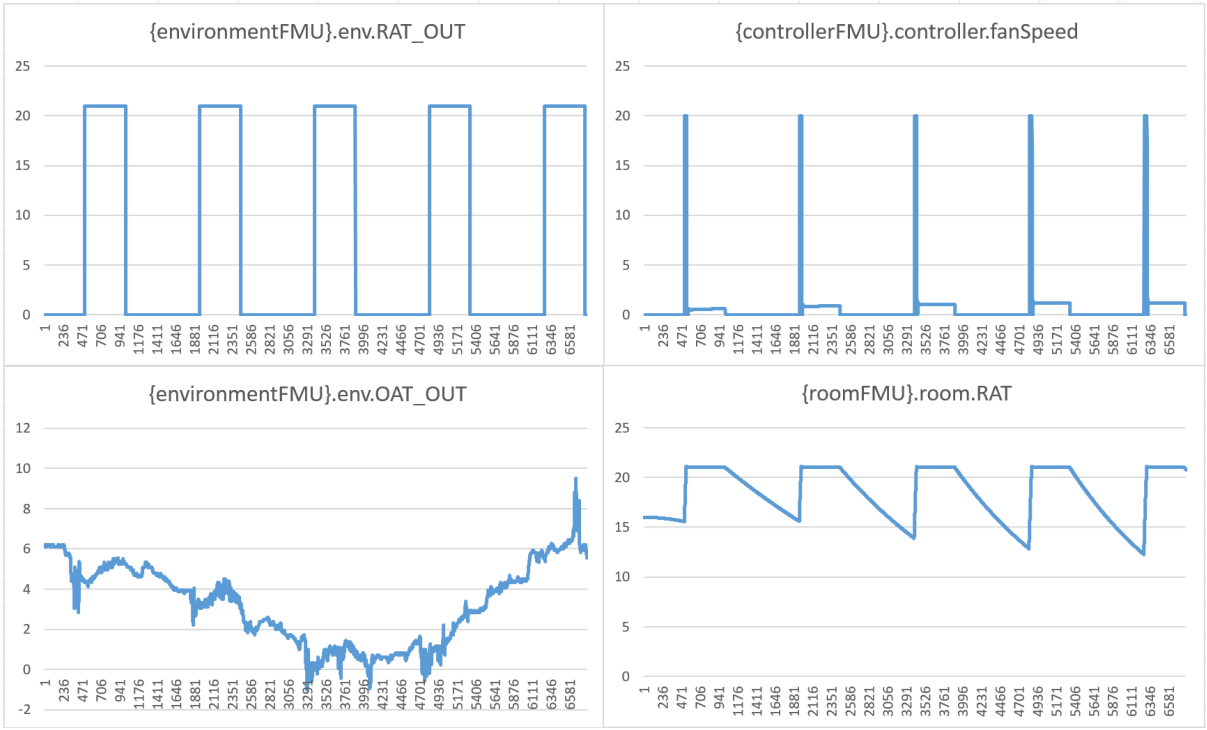


Figure 29: Co-simulation results as shown in graphs from result log files

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 the previous section, 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

As mentioned above, the multi-model has three design parameters, *lambda_ Wall*, *rho Wall* and *controllerFrequency* which define the wall thermal conductivity, wall density and controller frequency respectively, which may be altered to perform DSE.

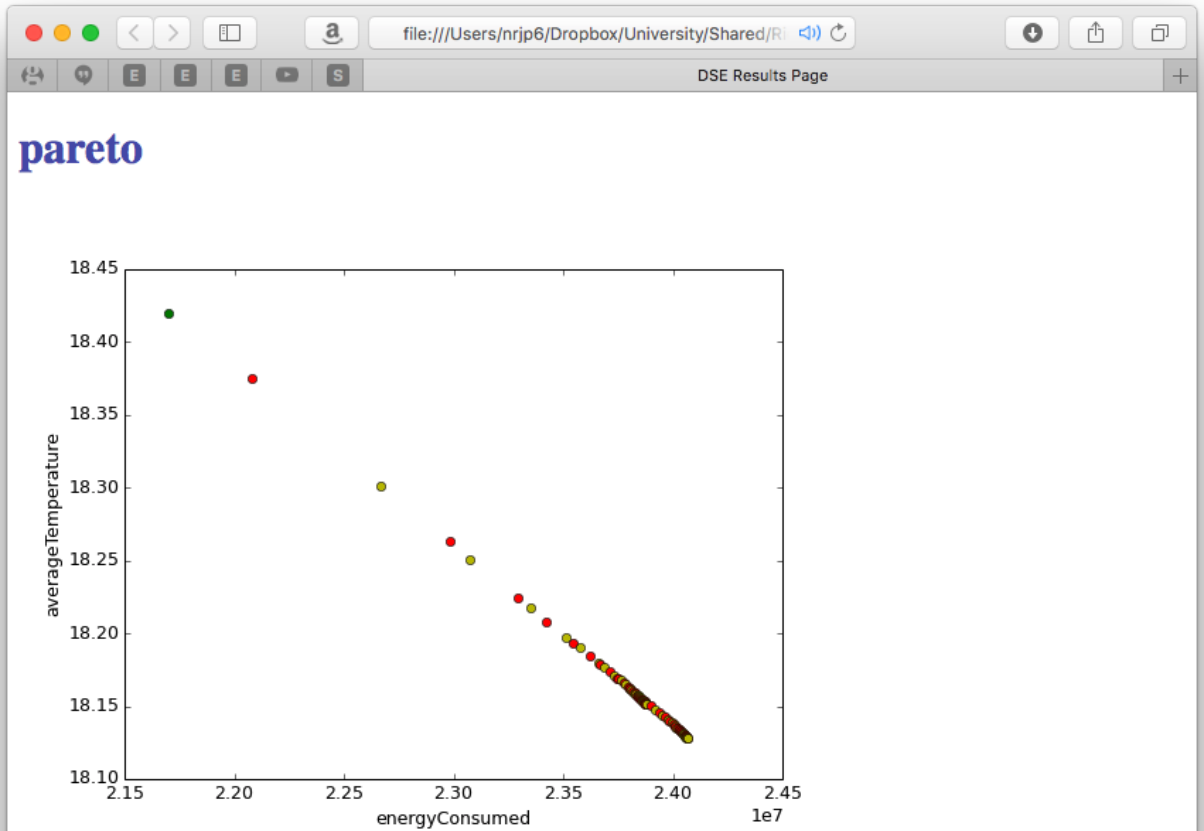
This example has 2 DSE experiments:

fcu-walls: The parameter values for *lambda_ Wall* ranges from 0.1192 to 10.1192 in intervals of 0.25, and the *rho Wall* 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.

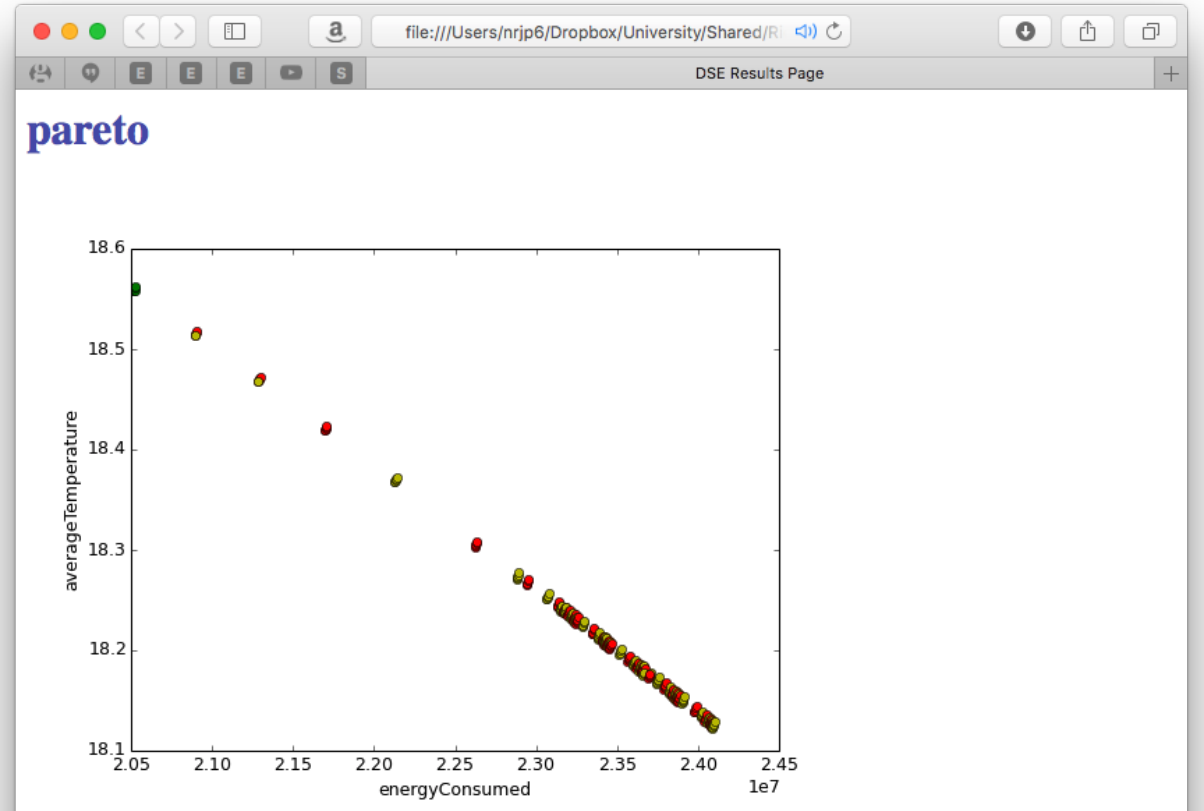
fcu-walls-controller: The second experiment varies the *lambda_ Wall* and *rho Wall* as earlier, but with a different collection of values: *lambda_ Wall* ranges from 0.1192 to 10.1192 in intervals of 1.0, and the *rho Wall* value ranging from 1300.0 to 1700.0 in intervals of 1000.0. In addition, the *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 5 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.



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



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

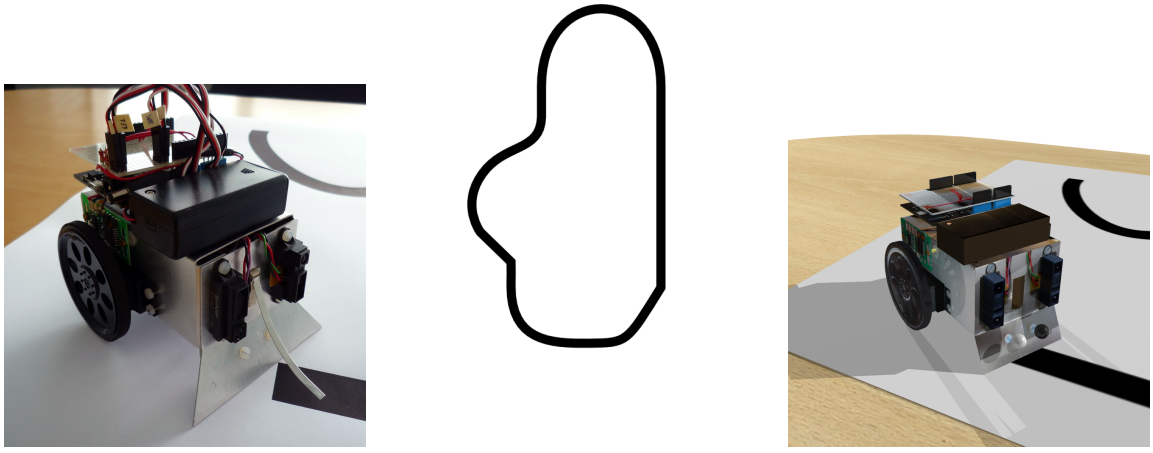
5 Line-following Robot

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 32 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 32: 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/case-study_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 33. This comprises the following components: *Body*, *Sensor1* and *Sensor2* physical components, a *Controller* cyber component and a *3DVisualisation* visualisation component. For simplicity, we omit the lower-level component types from the SysML model in Deliverable D3.4 [FGP+15].

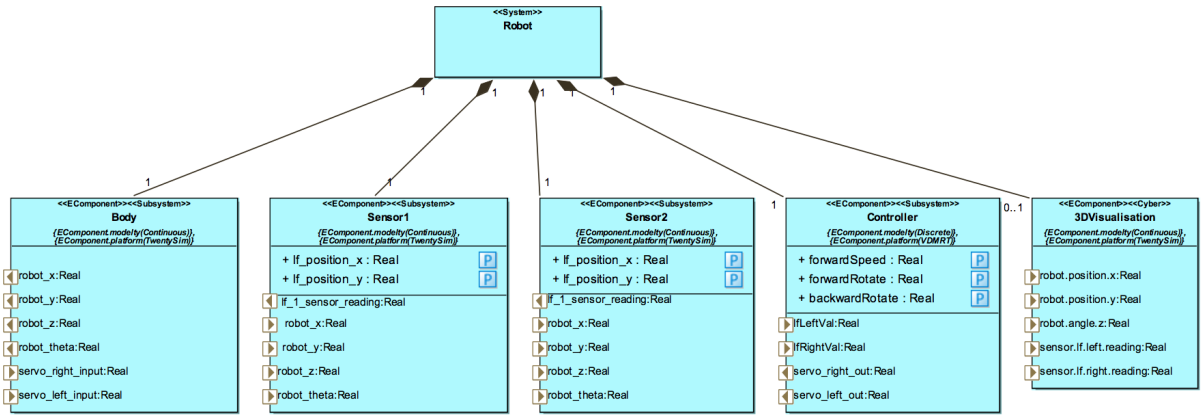


Figure 33: The line-following robot Architecture Structure Diagram

Two Connection Diagrams are defined. The first, in Figure 34, 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 35 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.

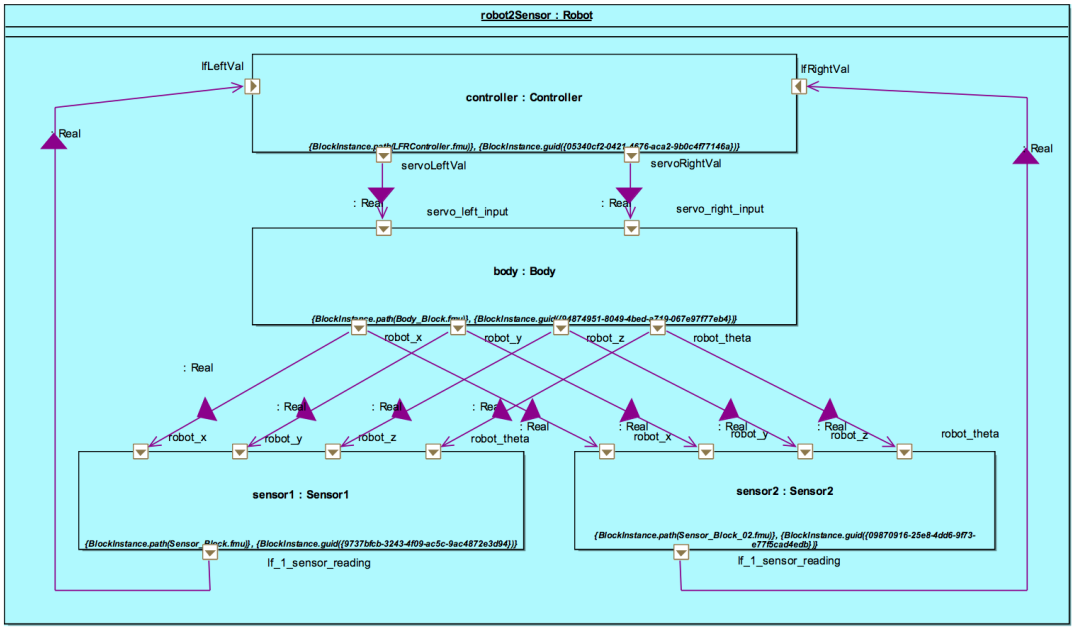


Figure 34: The line-following robot Connections Diagram

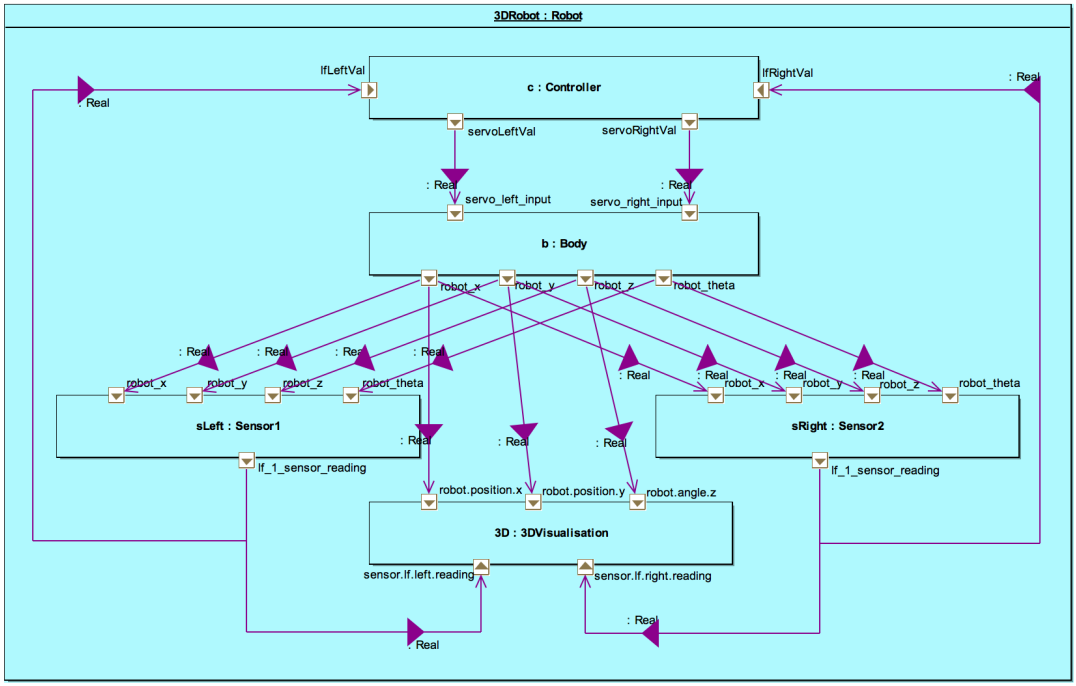


Figure 35: The line-following robot Connections Diagram

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 36 demonstrates the use of two 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

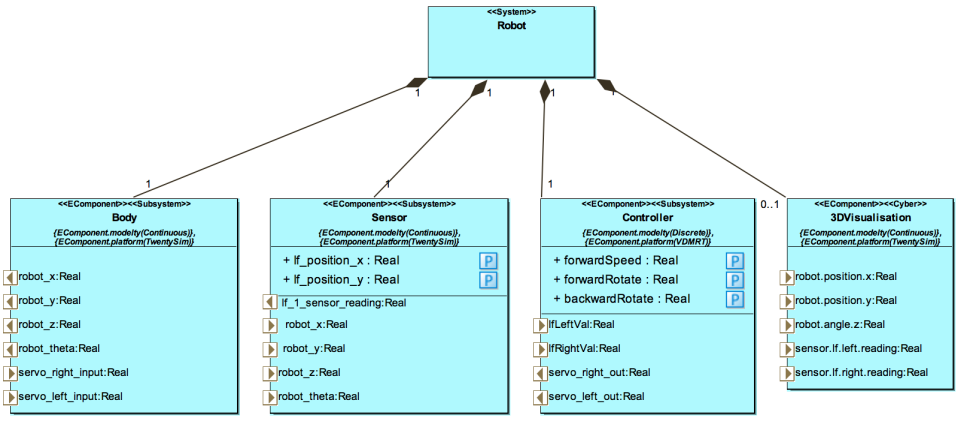


Figure 36: The line-following robot Architecture Structure Diagram with replicated sensors

Figures 37 and 38.

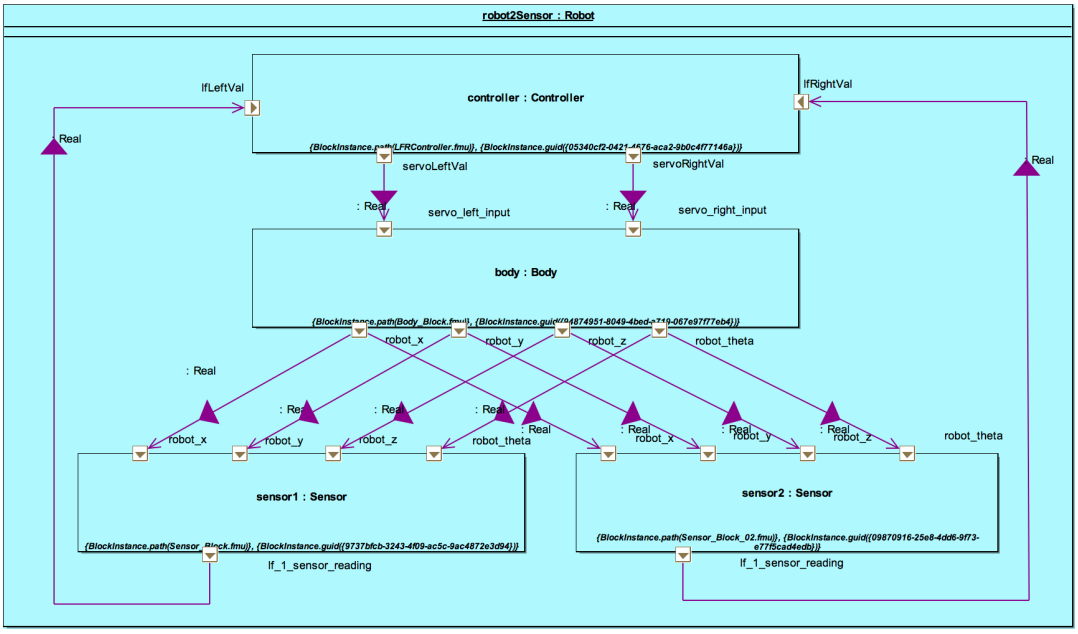


Figure 37: The line-following robot Connections Diagram

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 39, we first define a top-level decomposition with a *Body_Block* and a block to represent the body's *Environment*.

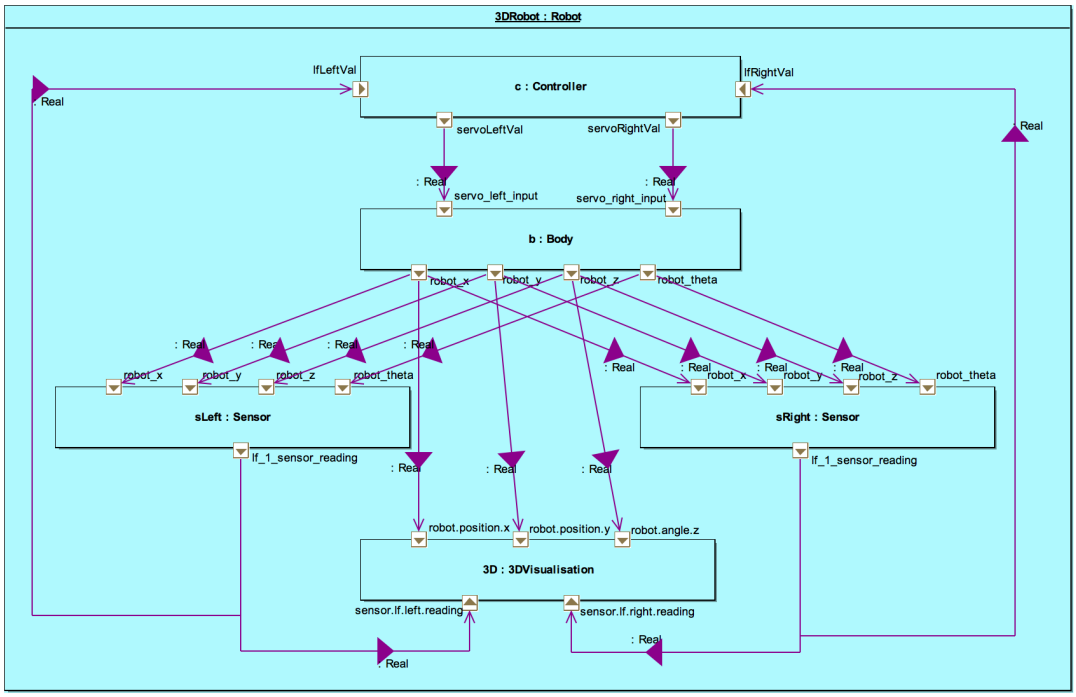


Figure 38: The line-following robot Connections Diagram

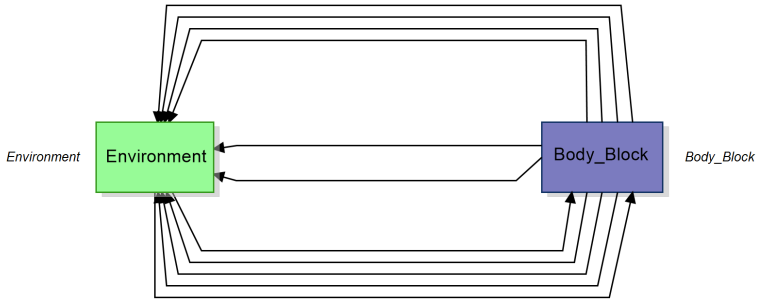


Figure 39: Top-level 20-sim model of the line-following robot *Body*

Decomposing the *Body_Block* further, the 20-sim model is defined as in Figure 40. 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 (*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 41. 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 42. 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

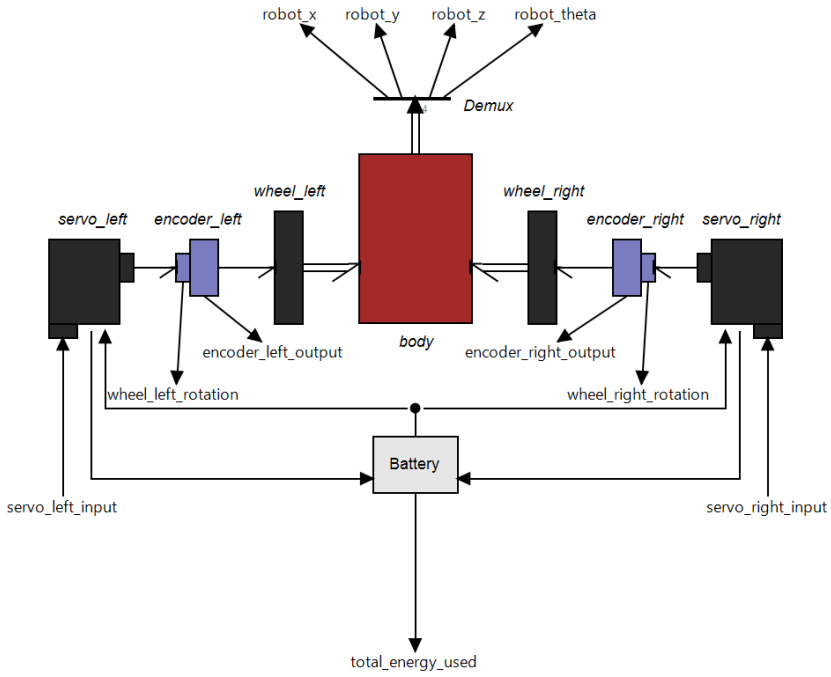


Figure 40: 20-sim model of the line-following robot *Body*

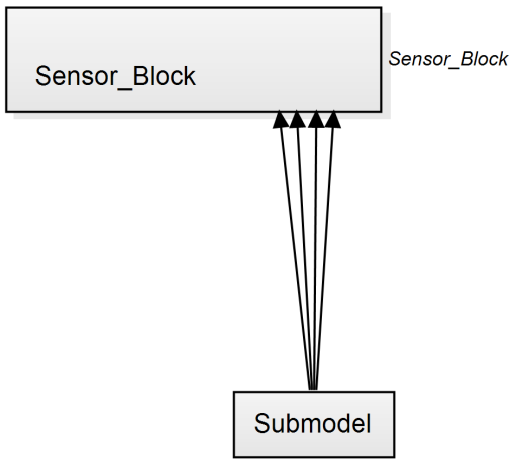


Figure 41: Top-level 20-sim model of the line-following robot *Sensor*

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* package. The *SensorBlock1.mo* element, shown in Figure 43 corresponds to the 20-sim model above. 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 44. 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

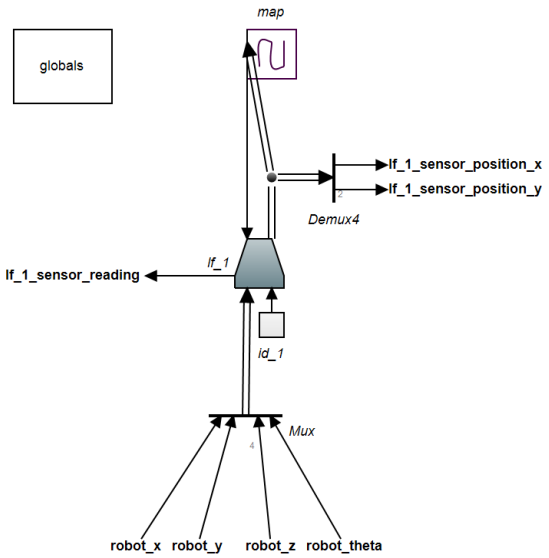
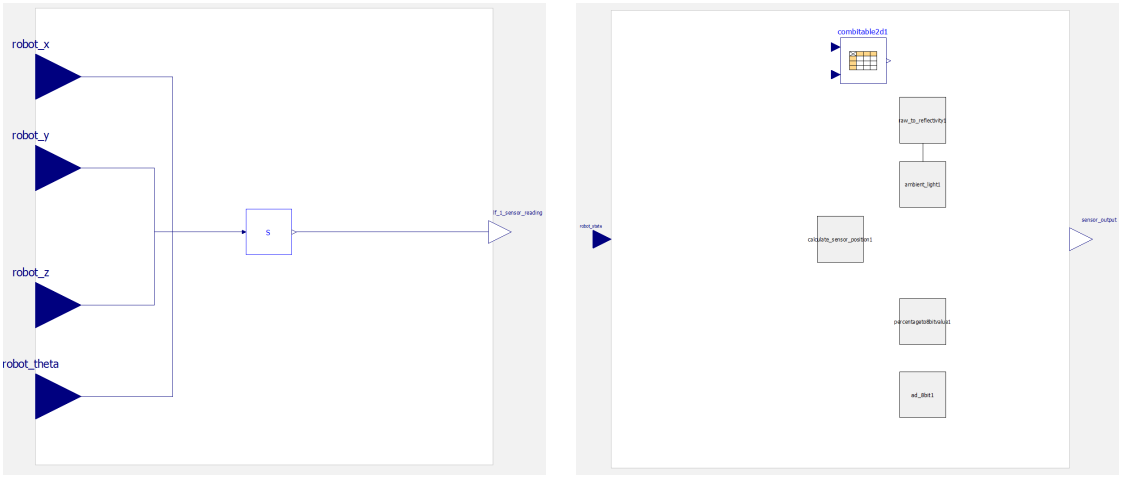


Figure 42: 20-sim model of the line-following robot *Sensor*



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

Figure 43: OpenModelica model of the line-following robot *Sensor*

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 (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

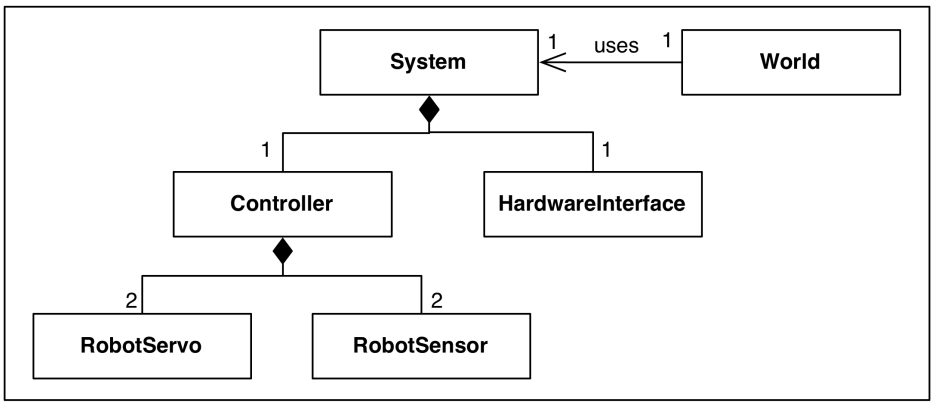


Figure 44: UML representation of line-following robot *Controller* model

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
- 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

[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*

addition, design parameters are set for the controller: the *forwardSpeed* and values for rotation – *forwardRotate* and *backwardRotate*.

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 can't 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 D5.2d [Gam16], 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 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 D5.2a [PLM16] and Deliverable D5.2b [BLM16], 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/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 45 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].

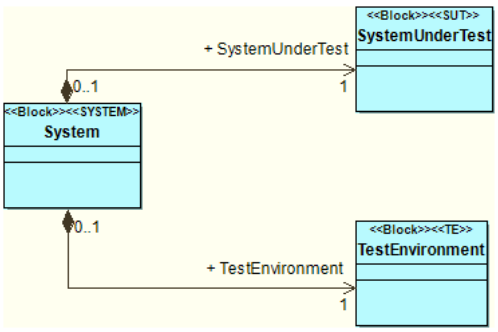


Figure 45: Top-level architecture diagram of the turn indicator model.

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 46 connects the system under test with the test environment using the described interfaces.

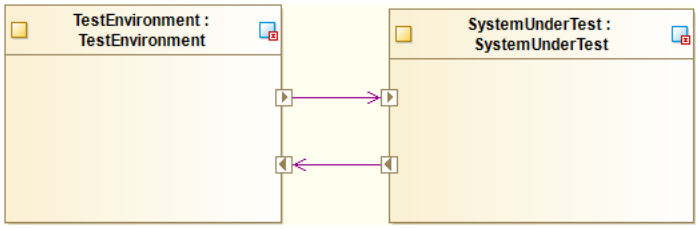


Figure 46: Top-level connection diagram of the turn indicator model.

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 47. The component `FLASH_CTRL` is responsible for deciding whether the left or the right side has to flash depending on the turn-indicator

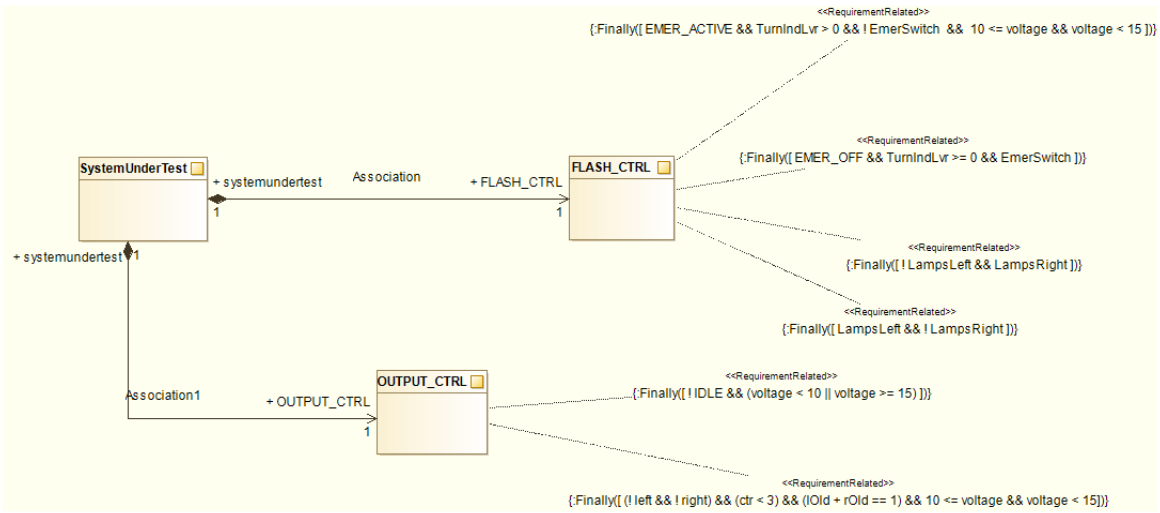


Figure 47: System under test architecture diagram of the turn indicator model.

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 48.

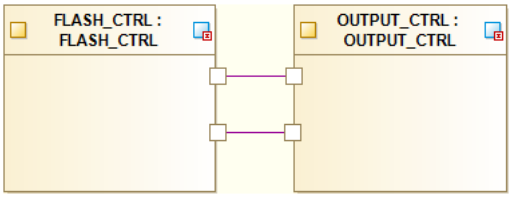


Figure 48: System under test connection diagram of the turn indicator model.

The `FLASH_CTRL` state machine in Figure 49 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 50 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 51 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

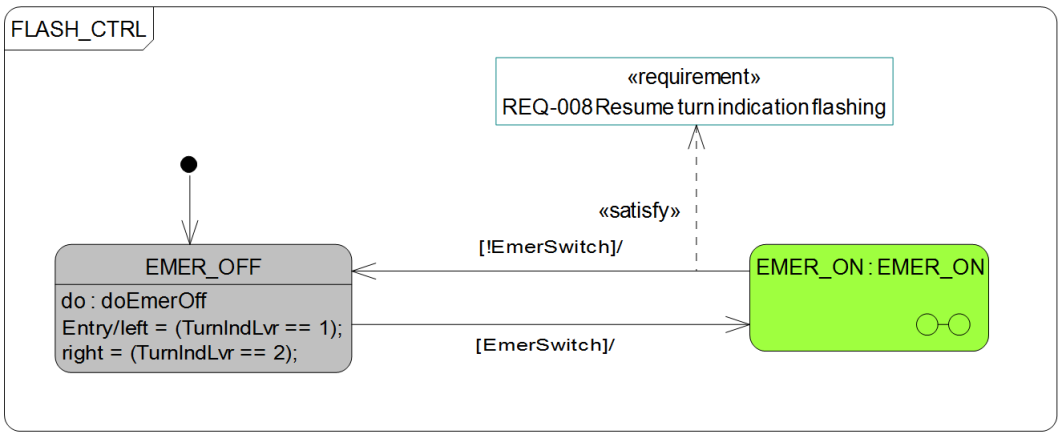


Figure 49: The `FLASH_CTRL` state machine.

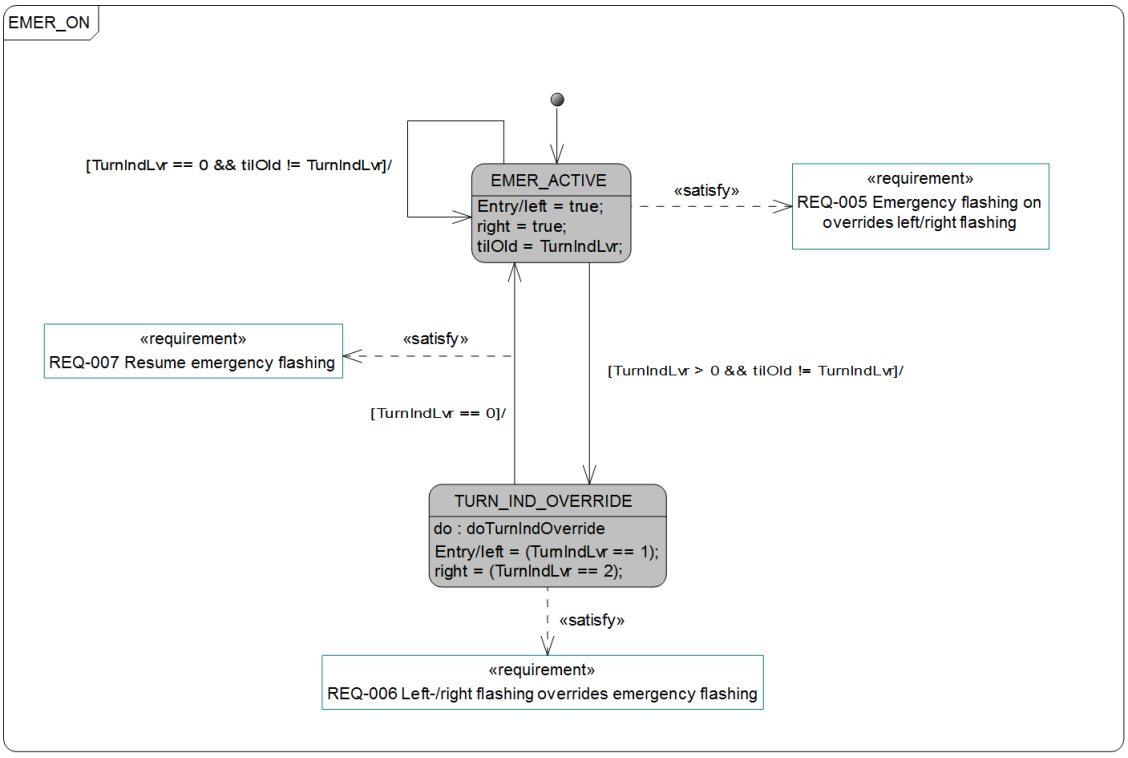


Figure 50: The `EMER_ON` composite state in the `FLASH_CTRL` state machine.

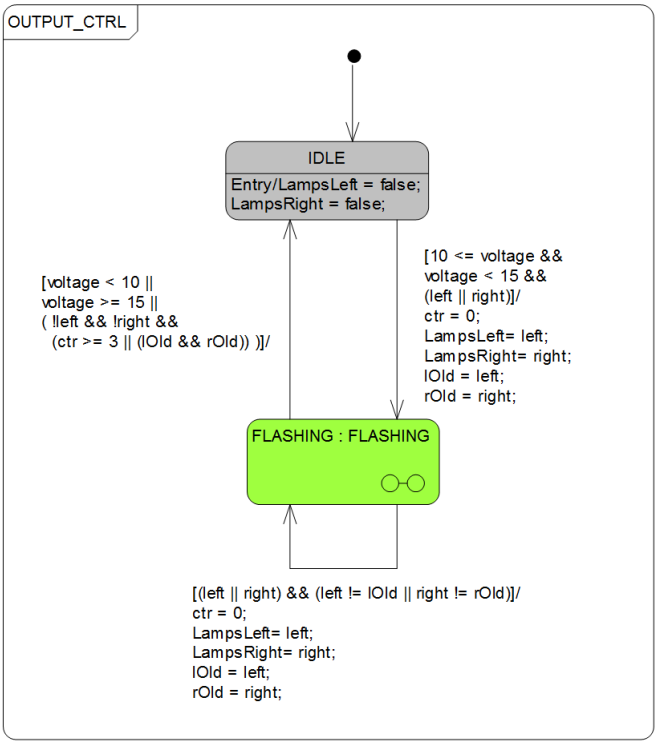


Figure 51: The `OUTPUT_CTRL` state machine.

Figure 50 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 47. 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 D5.2a [PLM16] and Deliverable D5.2b [BLM16], respectively.

7 UAV

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 52 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.

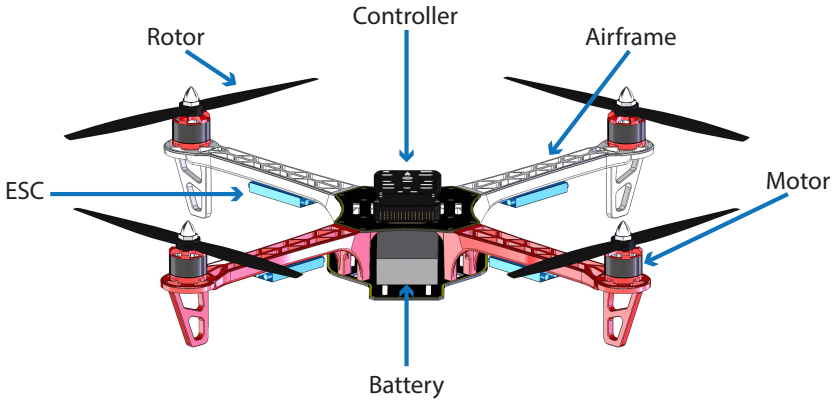


Figure 52: UAV 3D model

7.2 Usage

The example is available from the INTO-CPS application menu at *File>Import Example Project* or at `https://github.com/into-cps/case-study_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 53 and Figure 54 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`.

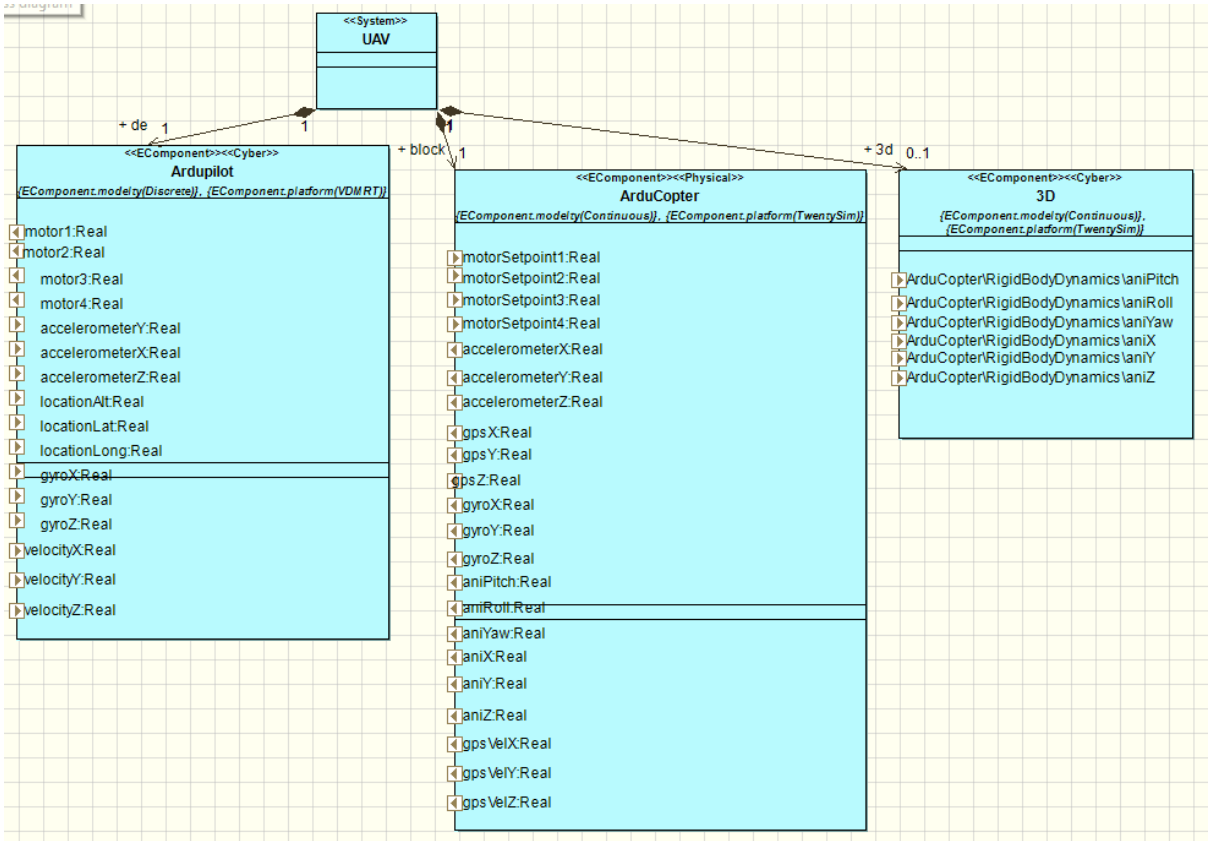


Figure 53: UAV Architecture Structure Diagram

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 54.

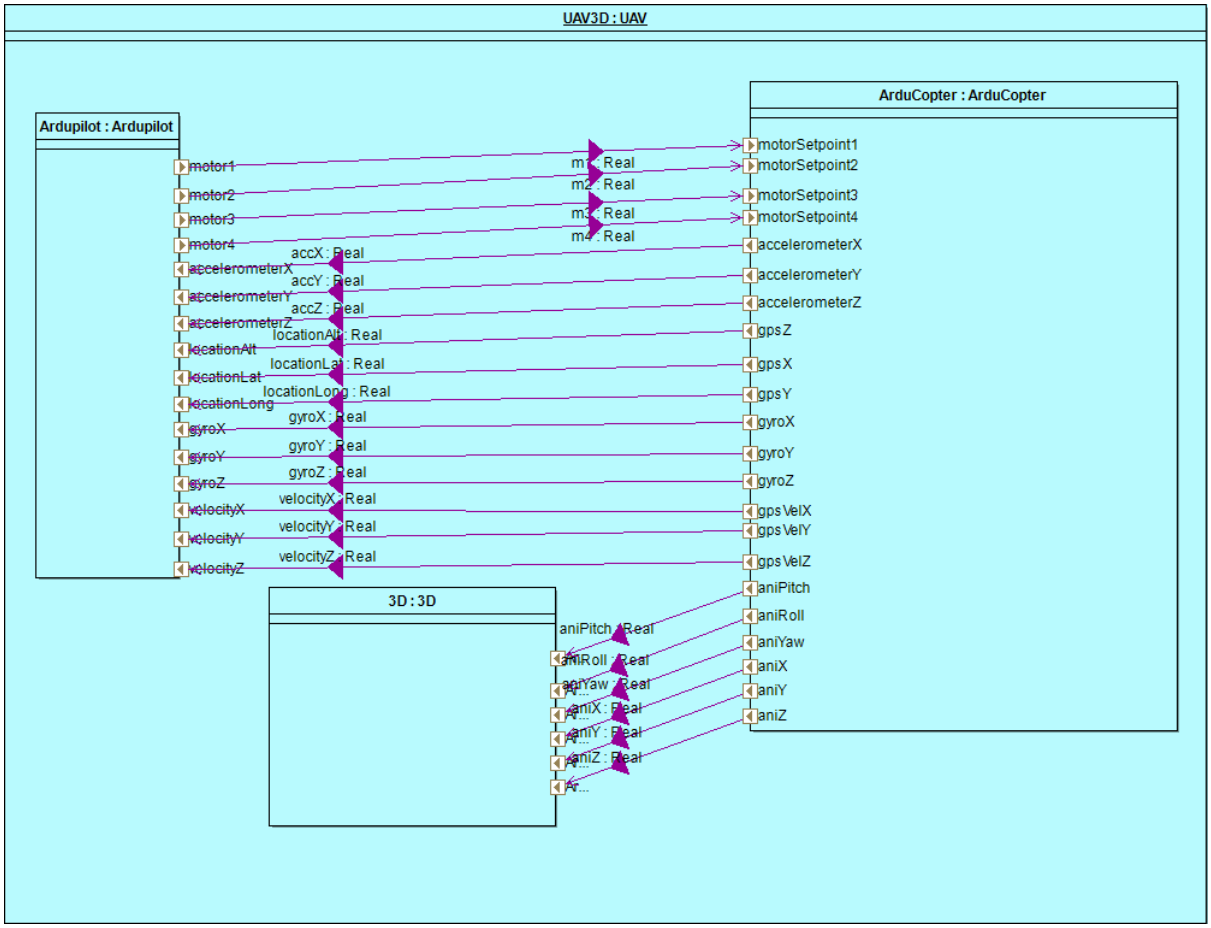


Figure 54: UAV Connections Diagram

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 55 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.

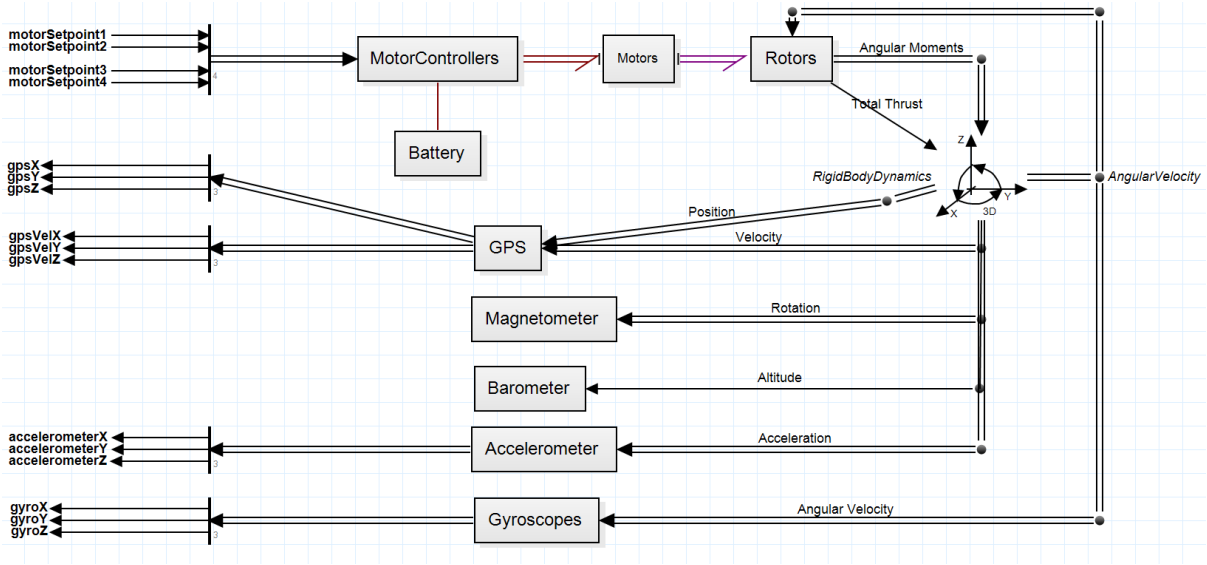


Figure 55: 20-sim model of the UAV

ArduPilot: Figure 56 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.

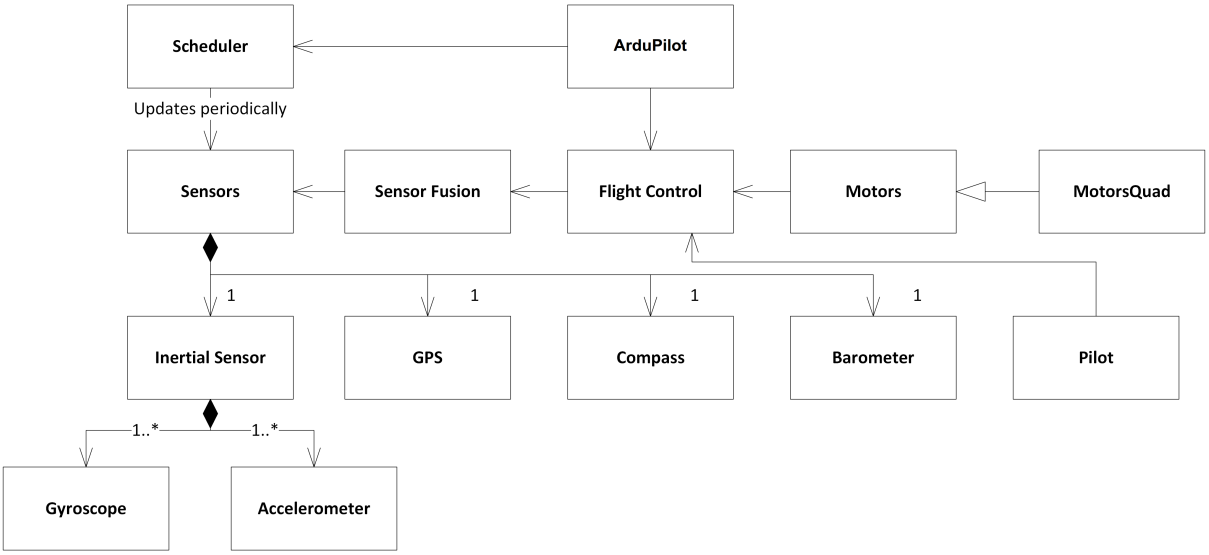


Figure 56: ArduPilot model overview

7.4.2 Configuration

This pilot study does not use any parameters and the connections should be self explanatory from Figure 54.

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.

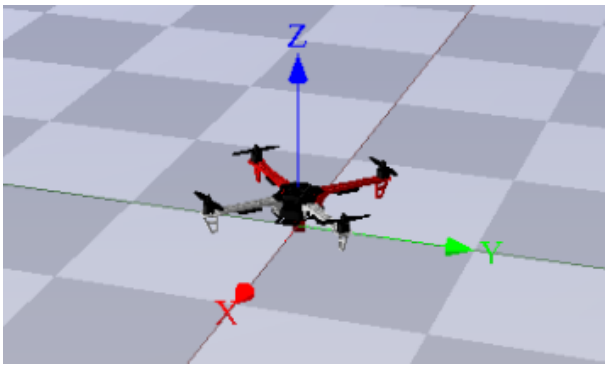


Figure 57: 3D visualization of the UAV

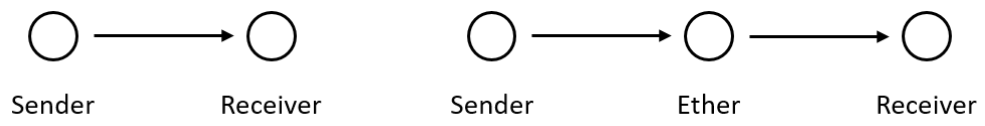
8 Ether

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 58(a), or via a third model called the *Ether*, as illustrated in Figure 58(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 D3.2a [FGPP16], and includes a discussion of realistic and faulty behaviour.



(a) Sender and Receiver connected directly

(b) Sender and Receiver connected via the Ether

Figure 58: 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/case-study_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.

Warning! Values printed from Overture FMUs are not visible in the COE status window in the INTO-CPS Application version 2.1.0 or below.

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

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 D3.2a [FGPP16] gives some initial guidance on overcoming such problems by introducing extra logic to give quality of service (QoS) guarantees in message-passing.

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 59 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.

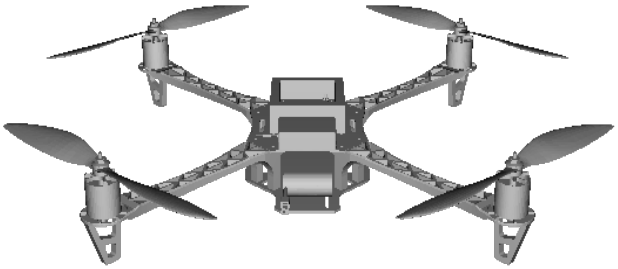


Figure 59: UAV 3D model

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/case-study_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 60, 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.

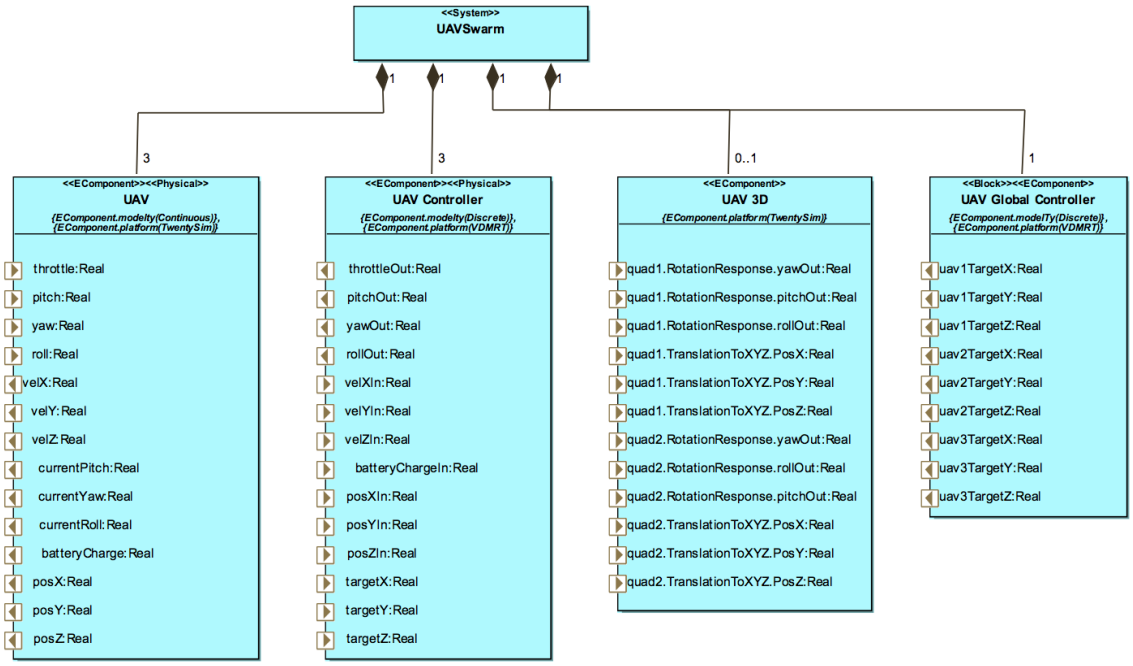


Figure 60: UAV Swarm Architecture Structure Diagram

Due to the large number of ports and connectors in this pilot, it 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 61 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.

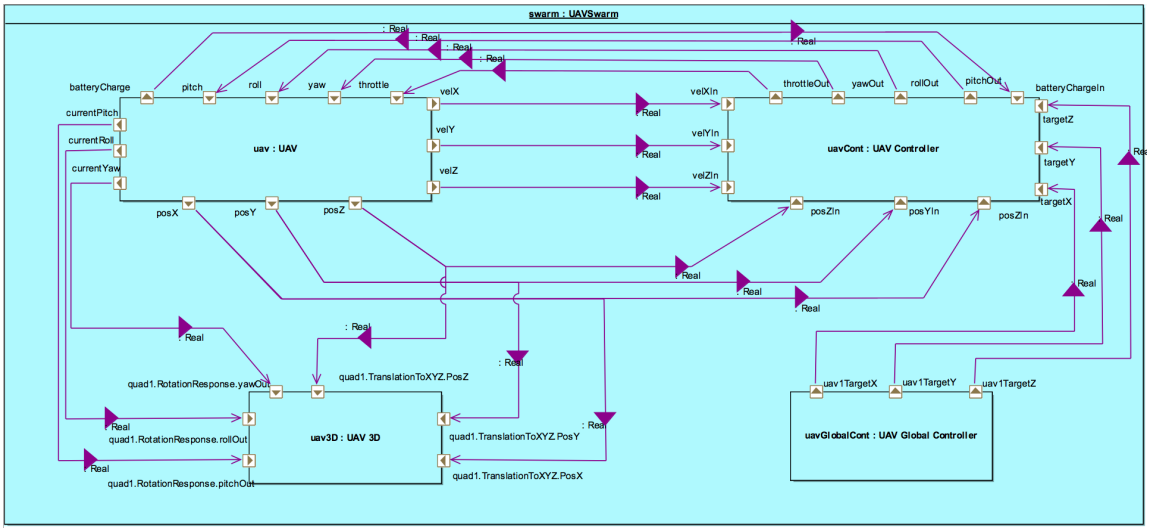


Figure 61: UAV Swarm Connections Diagram showing connections relating to UAV1

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 62.

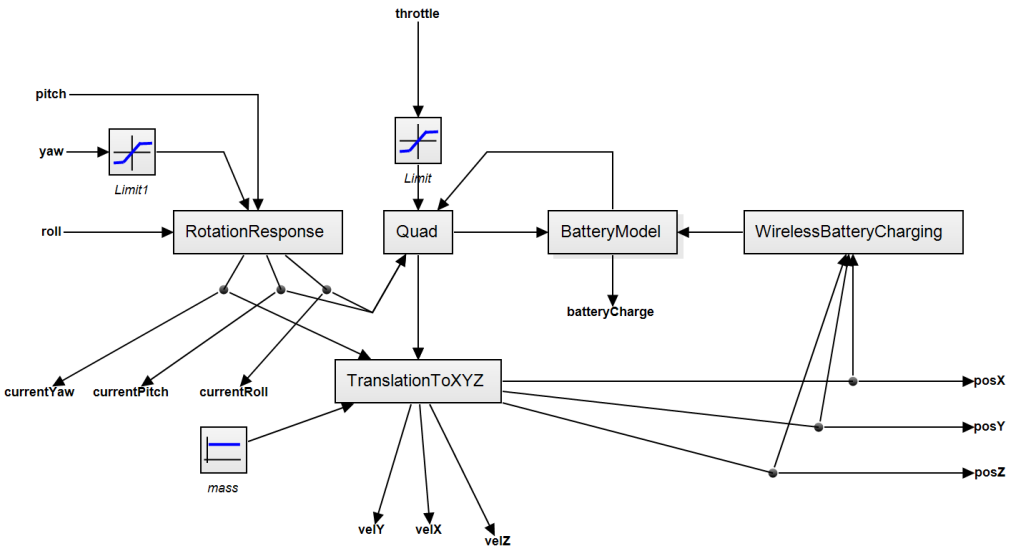


Figure 62: 20-sim model for UAV taking part in a Swarm

As can be seen from Figure 62, 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 and 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.

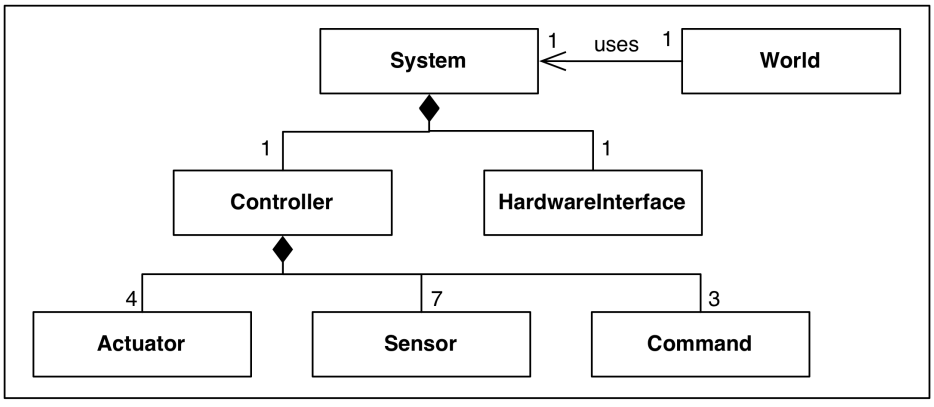


Figure 63: UAV Controller architecture

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.

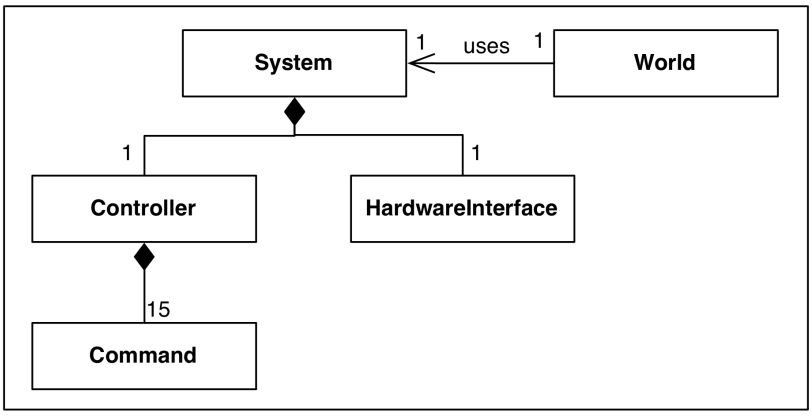


Figure 64: Global UAV Controller architecture

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 65 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 65: UAV Swarm visualisation

10 Smart Grid

10.1 Example Description

A smart grid is an electricity power grid where integrated ICT systems play a role in the control and management of the electricity power supply. cite? Such ICT elements include distributed control in households, control of renewable energies and networked communications.

In this section we outline a Smart Grid model to explore different design decisions in the cyber control of an electricity power grid. The model presented here is a small illustrative example, which omits complexities of a real Smart Grid. For example, the change from three-phase AC power to one-phase DC power allowing us to use simpler physical models. A second simplification is in the number of houses present in the grid model. We model only five houses, assumed to be in a small local area supplied by a single substation. We do not consider the remainder of the grid. To ensure that any effect due to changes in the power consumption by those properties are observed by the other houses, we skew the resistance of the transmission lines between the power generation and substation, and substation to houses.

10.2 Usage

This is a work-in-progress pilot (available at `https://github.com/into-cps/case-study_smart_grid`) and not currently intended to be used for co-simulation; the *master* branch therefore contains no models or FMUs. The work-in-progress artefacts are available in the *development* branch. There are several subfolders for the various elements: `FMU` – contains the various FMUs of the study; `Models` – contains the in development constituent models defined using the INTO-CPS simulation technologies; `Multi-models` – contains the multi-model definitions and co-simulation configurations; and `SysML` – contains the SysML models defined for the study.

10.3 INTO-CPS SysML profile

The SysML model of the Smart Grid comprises an Architectural Structure Diagram (ASD) and a single Connections Diagram (CD). The ASD in Figure 66, shows that the multi-model is composed of four EComponents: *FiveHouseGrid*, *SubstationController*, *DataNetwork* and *HouseController*. We model one CT physical element – the grid itself – in 20-sim, and three DE models for the ICT control and communication features.

The CD in Figure 67 shows that the *FiveHouseGrid* model sends voltage and current details to both the *SubstationController* and each of the five *HouseController* model instances. This is intended to represent sensed meter readings at different parts of the grid. The *SubstationController* and each of the*HouseControllers* communicate via the *DataNetwork* model – all having a set of input and output connections.

It should be noted, in Deliverable D3.1a [FGPP15], we presented two SysML models for a Smart Grid CPS. In these models we split the *FiveHouseGrid* into a collection of

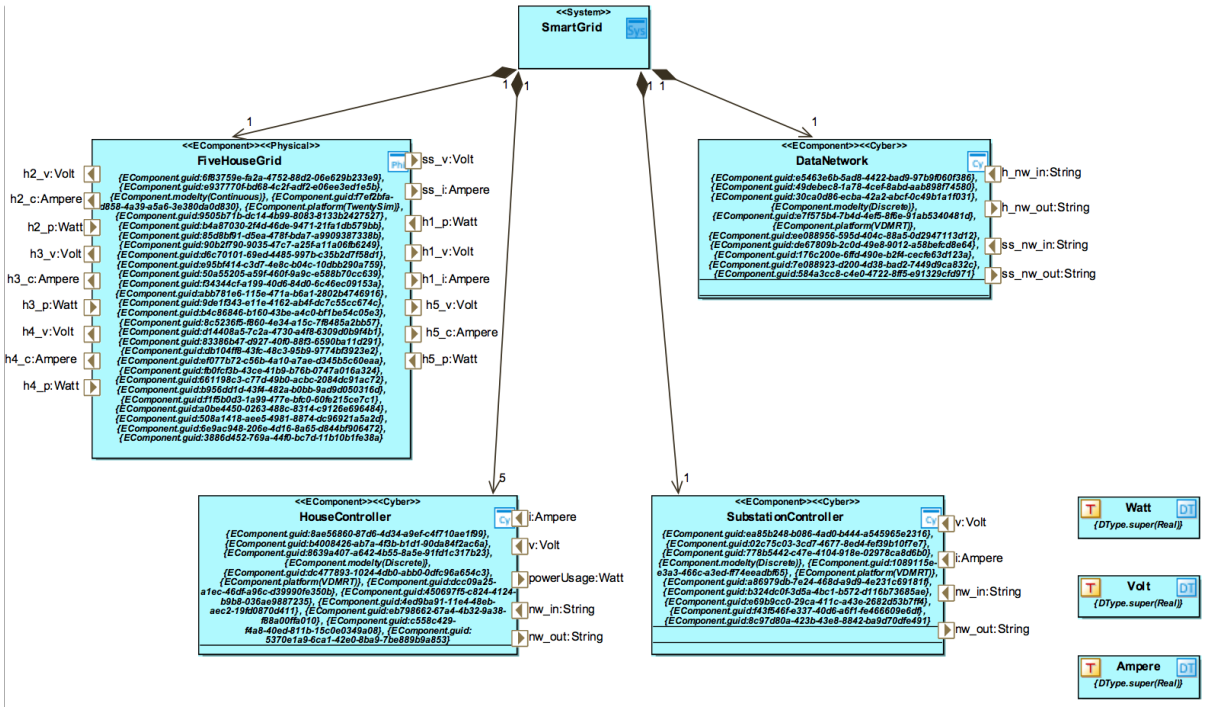


Figure 66: Architecture Structure Diagram for Smart Grid multi-model

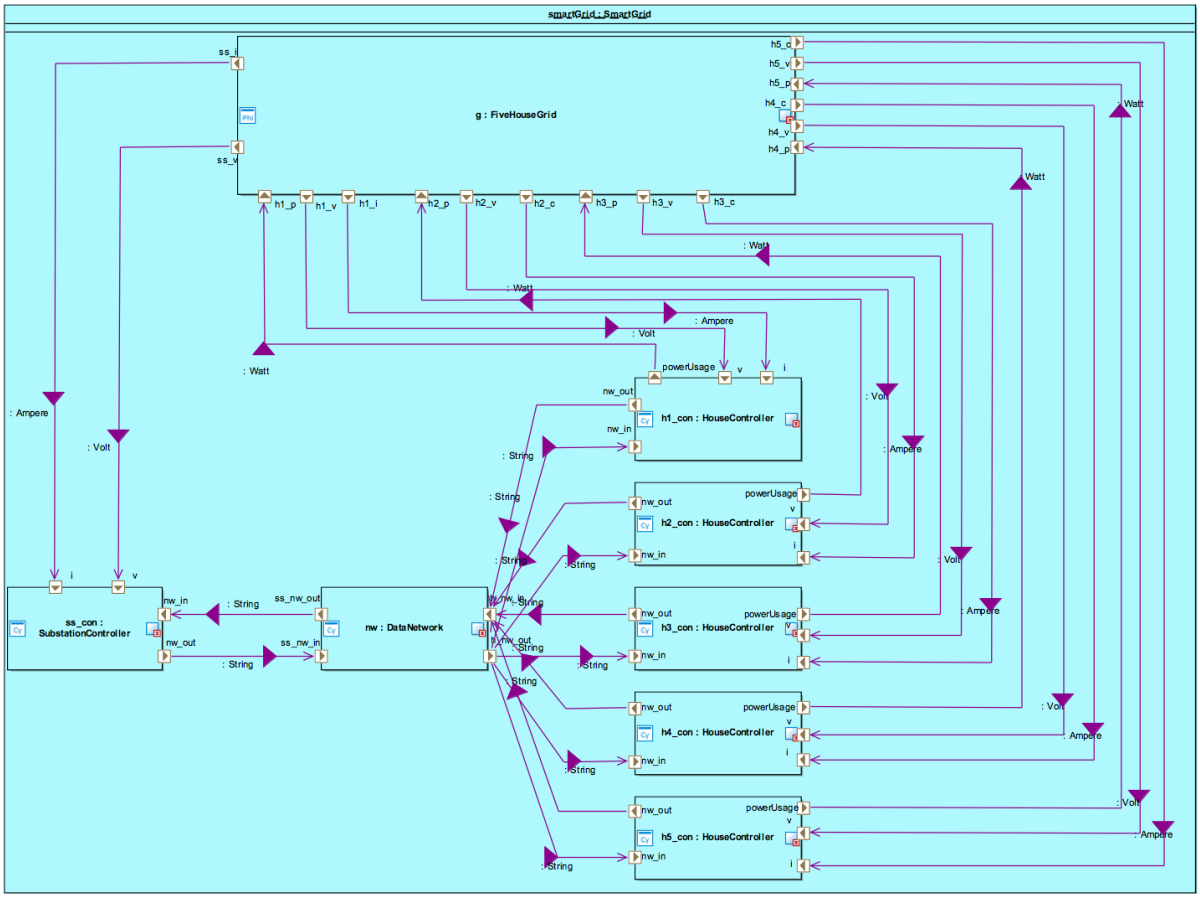


Figure 67: Connection Diagram for Smart Gridmulti-model

separate EComponents to represent the different subparts of the physical infrastructure. At present, this is not feasible due to constraints on algorithmic loops formed between the elements. We will continue developing this in Year 3 of the project.

10.4 Multi-model

10.4.1 Models

The Smart Grid multi-model comprises four 'simulation' models: a single 20-sim model corresponding to the electrical grid and three VDM models for the substation controller, house controller and data network.

As alluded to in Section 10.3, the physical aspects of the Smart Grid CPS is contained in a single physical model. The use of bonds to connect the various internal elements (such as *Power Generation*, *Transmission* and *Step-down Transformer*) limits the ability to spilt the physical model into smaller models.

FiveHouseGrid: In the CT model we begin by defining a top-level block diagram in the 20-sim tool. This allows the composition of a model in terms of the different physical elements of the Smart Grid CPS, and also to identify connections to the DE controller. This is shown in Figure 68. It is important to note that in 20-sim, we model both *effort* and *flow* – corresponding to *voltage* and *current* as we are modelling the electrical domain.

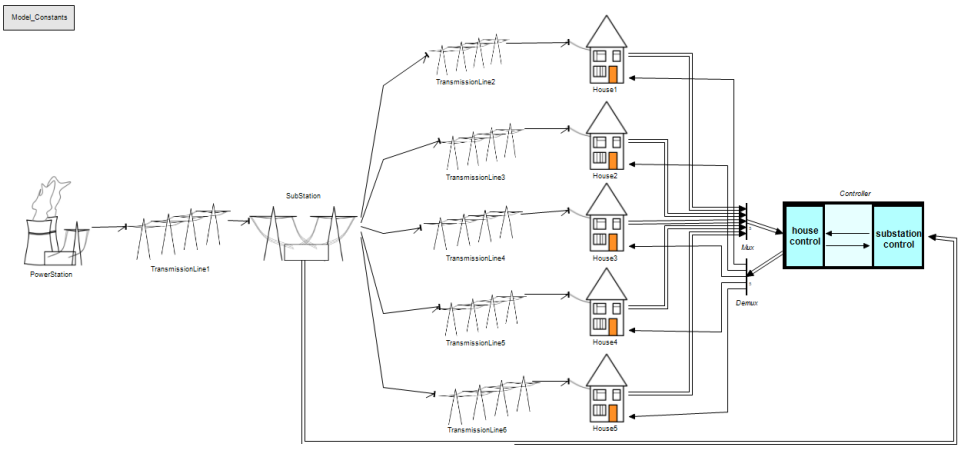


Figure 68: Top-level 20-sim block diagram connected by signals and energy bonds

The top-level block diagram is subsequently decomposed into different physical constituent elements: *Power Generation*, *Transmission Lines*, *Substation* and *Houses*. Each of these 20-sim blocks is given a graphical icon and may be 'exploded' to show their internal structure in terms of 20-sim elements.

Defining the *Power Generation* in Figure 69, we take an abstract view, comprising; a source of effort (`SE` in the model) representing a constant power source. For the purposes of this model, the voltage (effort) is defined as being 11000 volts.

The *Transmission Line*, shown in Figure 70 is modelled simply as a resistor – corresponding to the power drop experienced over such transmission lines. The

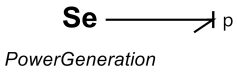


Figure 69: 20-sim elements comprising the Power Generation

resistance defined for the *Transmission Lines* differs between the high voltage line between *Power Generation* and *Substation* and the lower voltage *Transmission Line* between the *Substation* and the *Houses*.

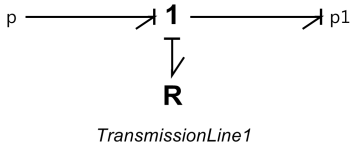


Figure 70: 20-sim elements comprising the Transmission Lines

A *Substation*, shown in Figure 71 and as defined in the SysML architectural model, has two physical parts. Firstly, the *Step-down Transformer* alters the voltage from the high transmission voltage to the target local voltage of 230 volts. The *Substation Meter* is modelled as a smart meter, in Figure 72 using *effort* and *flow meters* (an 'e' and 'f' surrounded by a circle) to monitor the voltage and current observed in the substation.

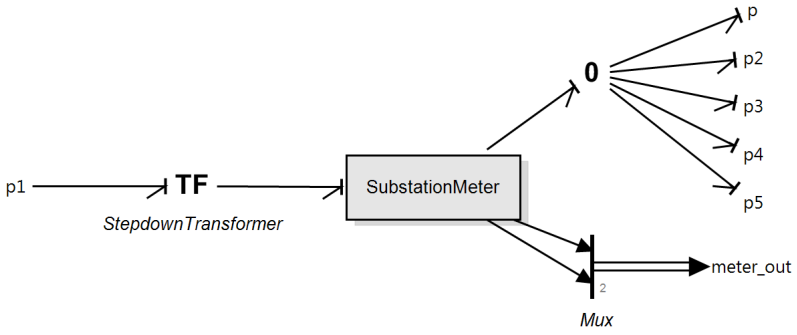


Figure 71: 20-sim elements comprising the Substation

Finally, the *Houses*, shown in Figure 73, are each modelled as containing a *House Meter* and a set of devices. The *House Meters* are modelled in the same way as the *Substation Meter*. The *Devices* are defined as being a variable resistor. The resistance is calculated using a variable *powerUsage*. This value is provided by a source external to the model.

The final 20-sim artefact to consider is the link to the DE controller. This element receives an array input (the voltage and current provided by the *Substation Meter*) and a matrix of *House Meter* readings (again voltage and current). The controller returns an array of values corresponding to the power usage of the different houses. The software controller itself is defined in a VDM-RT model, described in the next section.

HouseController The *HouseController* VDM-RT model dictates, and aims to manage, the power usage of each house. The power usage takes two forms: unmanaged profiles of usage – where devices are used solely at the behest of the user; and managed

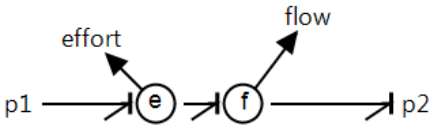


Figure 72: 20-sim elements comprising a Smart Meter

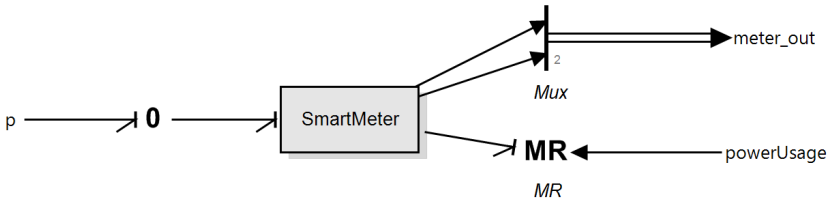


Figure 73: 20-sim elements comprising a House

power use – where the controller may manage the power use. The architecture is shown in Figure 74. The *System* class contains an instance of the *Controller*, *HardwareInterface* and *IOFactory* classes. The *HardwareInterface* class contains input and output ports for the model – communication with the *FiveHouseGrid* model (inputs are the house voltage and current, with the house power usage as output), and with the *DataNetwork* (with input and outputs for data communication).

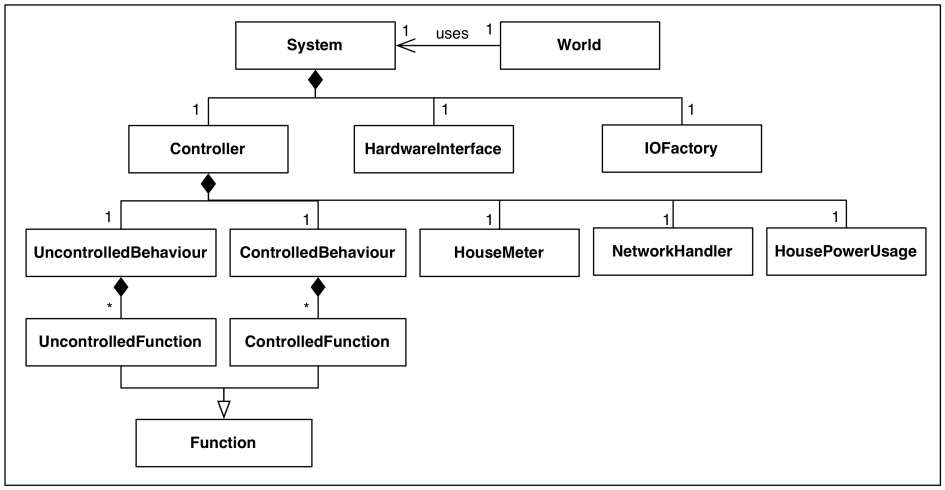


Figure 74: Architecture of HouseController model

In the case of unmanaged power usage, this may mean that the home owner turns devices and appliances on and off at will, or where some appliances change power usage at set times through the day (heating, for example). In the VDM-RT model, we define a collection of appliance and device power use profiles, a map of time to power use, which will always occur during simulation. Each house is initialised with a collection of device power profiles.

Managed power sources are modelled in a similar manner to unmanaged sources, apart from the fact they have a shorter profile, and no fixed start time. The House-

Controller may, therefore, change the start time of a managed device depending upon a policy defined in the controller. At present, once a device has started, it can not be paused, however this may be changed in future.

The HouseController has several control mechanisms for managing the controlled devices in that household. These include:

No controlled devices: In this mode the house controller cancels all controllable devices from starting. This can be reversed, however at present if the start time passes whilst cancelled, the device will not turn on in the future. This mode does not stop already running devices.

No management: The controlled devices operate as requested – the controller does not alter the start time.

Local control: The decision to start a controlled device is based upon the local meter readings at a set time before the device is scheduled to start. Each house has a voltage threshold – which dictates the minimum voltage permitted for a smart device to be turned on. If the voltage is below this threshold, the starting time of the smart device is increased by a set time.

Request: The HouseController makes an explicit request to the substation to check if controlled devices may start.

Additional control mechanisms may be added at a later time.

SubstationController: The *SubstationController* VDM-RT model architecture is shown in Figure 75. The SubstationController aims to manage the power across a set of houses. The controller takes, as input, the voltage and current at the substation and also receives meter readings from all the houses it is managing. A network may be modelled for the transmission of these readings, allowing for the modelling of a faulty network. In this project we abstract from the network and assume a perfect network.

The SubstationController is able to make decisions and influence the behaviour of the individual HouseControllers based upon the meter reading values at any point. The SubstationController has two control mechanisms available to a policy designer:

Reporting: This is the simplest mode, whereby it receives readings from each of the houses but takes no action. In this mode, however, the SubstationController may still respond to requests made by the different HouseControllers.

Substation control: In this mode, the SubstationController takes control of controlled devices of all houses in the neighbourhood. Modelling this is beyond the scope of the project, however we could envisage multiple subtypes of this control mode depending on how the engineer decides upon which devices to allow to start, for example depending on financial incentives.

DataNetwork Not yet defined – in Year 3 of the project, we shall implement this model using the Ether pilot study – see Section 8.

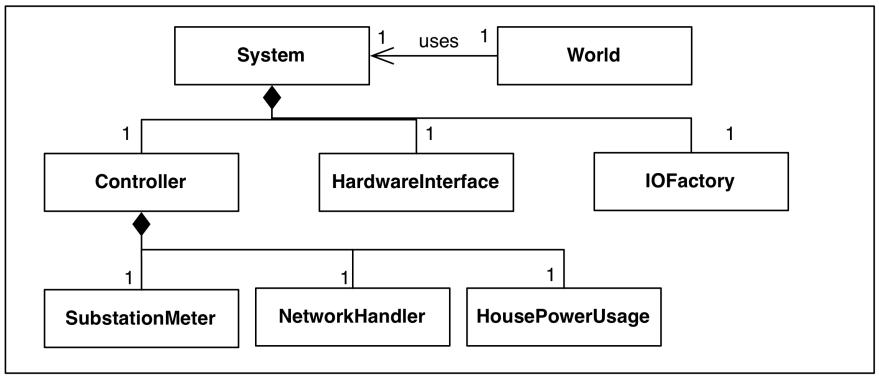


Figure 75: Architecture of SubstationController model

10.4.2 Configuration

In this pilot, we see four groups of connections.

The first, between the physical *FiveHouseGrid* and the cyber *SubstationController*, relate to the metering of voltage and current at the substation of the grid. The connections are:

- from the *FiveHouseGrid* `ss_i` port to the *SubstationController* `i` port, and
- from the *FiveHouseGrid* `ss_v` port to the *SubstationController* `v` port.

Next, connections exist between the physical *FiveHouseGrid* and the each of the five cyber *HouseControllers*, relating to the metering of voltage and current at the substation of the grid. In addition, there is a connection carrying the current power usage commands from the controller to the grid. Due to the number of connections, we consider only one house instance here. The connections are:

- from the *FiveHouseGrid* `h_i` port to the *HouseController* `i` port,
- from the *FiveHouseGrid* `h_v` port to the *HouseController* `v` port, and
- from the *HouseController* `powerUsage` port to the *FiveHouseGrid* `h_p` port.

The next collection contains two connections between the *DataNetwork* and *SubstationController* to represent the flow of data into and out of the *SubstationController*.

- from the *DataNetwork* `ss_nw_out` port to the *SubstationController* `nw_in` port, and
- from the *SubstationController* `nw_out` port to the *DataNetwork* `ss_nw_in` port.

Finally, we present two connections between the *DataNetwork* and *HouseController* to represent the flow of data into and out of the *HouseController*. Again, we consider only one house instance here.

- from the *DataNetwork* `h_nw_out` port to the *HouseController* `nw_in` port, and
- from the *HouseController* `nw_out` port to the *DataNetwork* `h_nw_in` port.

10.5 Co-simulation

FMUs have not yet been generated from the models of this study and therefore co-simulation has not yet been performed. This shall follow in Year 3 of INTO-CPS.

11 Roadmap for Pilot Studies

In the final 12 months of the project, the pilot studies will demonstrate the range of INTO-CPS technologies as they become available during the final year. In addition, future pilot studies must continue to exhibit *network communication* – a key property of CPSs. Initial efforts have been demonstrated in the Ether case study in Section 8, the UAV swarm (Section 9) and Smart Grid (Section 10) studies are natural candidates for deploying the Ether.

11.1 Future INTO-CPS Technology Demonstration Needs

In the final 12 months, those technologies developed in the project should be demonstrable by the pilot studies. As a part of producing a roadmap, the technology developers in the project were asked for some properties required of future studies. In this section, we briefly outline characteristics the future studies should target. We do not aim to identify which studies may test these specific areas, only that they should be targets.

Code Generation Code generation has been used in three pilots. In the coming year, we aim to apply code generation for additional pilots. Additional language data structures will be supported as necessary.

Design-Space Exploration Pilot studies demonstrating DSE should use the DSE SysML profile (as detailed in Deliverable D3.2a [FGPP16]. Pilots should also demonstrate the use of both exhaustive, genetic algorithms and other search algorithms as appropriate. All ranking mechanisms should be demonstrated.

Test Automation The Test Automation feature has been demonstrated on only two studies. In the coming year, we aim to cover more studies.

Model Checking Model checking has been demonstrated in the turn indicator pilot only, we seek better coverage in the final deliverable.

Traceability The use of traceability links and queries for impact analysis should be supported in pilots. The line follow robot pilot has example traces defined in Deliverable D3.2a [FGPP16] – these should be implemented in the final pilot study deliverable.

INTO-SysML profile The INTO-SysML profile has been extended in the latter part of this year. The architectural models of the pilot studies should be altered to take advantage of new stereotypes, and where appropriate use the new DSE and TA (see above) profiles. In addition, for the purposes of Task T2.1, we require: a connection diagram that is different from the port-dependency graph (in that there may be connections that do not lead to a dependency) and showing algebraic loops.

References

[BLM16] Jörg Brauer, Florian Lapschies, and Oliver Möller. Implementation of a Model-Checking Component. Technical report, INTO-CPS Deliverable, D5.2b, December 2016.

[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.

[FGPP15] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Method Guidelines 1. Technical report, INTO-CPS Deliverable, D3.1a, December 2015.

[FGPP16] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Method Guidelines 2. Technical report, INTO-CPS Deliverable, D3.2a, December 2016.

[Gam16] Carl Gamble. DSE in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D5.2d, December 2016.

[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.

[PLM16] Adrian Pop, Florian Lapschies, and Oliver Möller. Test automation module in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D5.2a, 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.