Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Method Guidelines 3

Deliverable Number: D3.3a

Version: 1.4

Date: December 2016

Public Document

http://into-cps.au.dk

Contributors:

John Fitzgerald, UNEW
Carl Gamble, UNEW
Martin Mansfield, UNEW
Richard Payne, UNEW
Ken Pierce, UNEW

Editors:

Ken Pierce, UNEW

Reviewers:

Christian König, TWT
Etienne Brosse, ST
Frederik Foldager, AI

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	14-02-2017	Ken Pierce	Initial document structure
0.2	24-10-2017	Ken Pierce	Revised document structure
0.3	26-10-2017	Ken Pierce	First draft of materials
0.4	31-10-2017	Ken Pierce	Added SysML chapter
0.5	01-11-2017	Ken Pierce	Draft for comment
1.0	07-12-2017	Ken Pierce	First revisions based on comments
1.1	12-12-2017	Ken Pierce	Further updates based on comments
1.2	14-12-2017	Ken Pierce	Revised SysML chapter
1.3	18-12-2017	Carl Gamble	Traceability chapter completed
1.4	18-12-2017	Carl Gamble	DSE chapter completed

Abstract

This document is the final methods guidance document for the INTO-CPS technologies. It is aimed at end users of the technologies, and complements the User Manual, Deliverable D4.3a [1], by helping to describe the why to complement the how. This document presents: a concepts base, which describes the terminology used within INTO-CPS; information on getting started with the technologies, and the variety of workflows they support; a description of the traceability features of the tool chain, and why these must be considered at the beginning of development to realise them fully; guidelines on incorporating requirements engineering in a cyber-physical systems (CPS) context; a description of the INTO-SysML profile and its use; guidance on discrete-event first (DE-first) modelling as a way to begin multi-modelling; guidance on modelling networks in multi-models; and guidelines for the use of design space exploration (DSE) features of the INTO-CPS tool chain.

Contents

Part I

Introductory Material

Chapter 1

Introduction

The INTO-CPS tool chain brings together a variety of technologies to allow engineers to undertake collaborative, model-based based design of Cyber-Physical Systems (CPSs). Each technology has its own culture, abstractions, and approaches to problem solving that inform how they are used. Many of these things are tacit and tend to be discovered only after trying to combine them. The guidance in this document aims to help the reader overcome these challenges, and to understand how best to use these technologies.

This document complements the tools User Manual (Deliverable D4.3a [1]) —which gives detail on how to use the features of the tool chain— by providing information on when and why you might use these features. The guidance in this document has been distilled from experience gained in a series of pilot studies and applications of INTO-CPS technologies to real industrial case studies. These pilot studies now appear as examples that can be opened directly from the INTO-CPS Application, supported by descriptions in the Examples Compendium (Deliverable D3.6 [2]). Industrial applications can be read about in the Case Studies report (Deliverable D1.3a [3]).

1.1 How to Use This Document

Since this document is aimed at both new and experienced users of the INTO-CPS technologies, it has been divided into two parts. Part I, Chapters 1–3, covers introductory material including this introduction, the terminology used in INTO-CPS, and the various activities that INTO-CPS enables. Part II, Chapters 4–9, covers more advanced topics that require a basic familiarity with the INTO-CPS technologies.

While the chapters in the Part II are ordered primarily based on a start-to-end "work flow" of system development with INTO-CPS, it is not necessary to read the advanced chapters in order. While experienced users may read any chapter on which they require further guidance, new users are recommended to:

- Read the introductory material in Part I.
- Follow the first tutorial to experience using the INTO-CPS Application.
- Import one or two examples from the Examples Compendium (Deliverable D3.6 [2]) into the INTO-CPS Application and interact with them.
- As you start your own multi-modelling, return to this document as and when you require guidance on a particular area.

1.2 Overview of Sections

Chapter 2: Concepts and Terminology This chapter is an introduction to the concepts and terminology used in INTO-CPS. It explains many terms from the various baseline technologies, as well as other model-based design terminology. In parts this involved reconciling terms used differently in different areas, and finding common, agreed-upon terms for similar concepts. These concepts are applicable for all documents produced by INTO-CPS (this document, user manuals, deliverables, and publications).

Chapter 3: Getting Started with INTO-CPS This chapter suggests how to get started with the INTO-CPS tool chain, trying out core features by following tutorials, which puts the other range of activities in context. It also describes the full range of activities that the INTO-CPS tool chain enables.

Chapter 4: Traceability and Provenance This chapter explains how to approach the INTO-CPS tool chain in order to make used of the machine-assisted traceability features included in the INTO-CPS Application and baseline tools. It also describes the set of included queries that can be run over traceability data sets, and how further queries can be written.

Chapter 5: Requirements Engineering This chapter focuses on a key initial activity for CPS design, specifically requirements engineering (RE) in a CPS context, and the specification and documentation of requirements placed upon a CPS. This section describes an approach called SoS-ACRE in the context of INTO-CPS, and includes descriptions of how this approach can be realised using tools identified as useful by the industrial partners (specifically SysML and Excel). By following these guidelines, engineers can bridge the gap between natural language requirements and multi-models.

Chapter 6: SysML and Multi-modelling This chapter describes the various roles of SysML in INTO-CPS. SysML can be used for architectural modelling of CPSs, while INTO-CPS provides additional SysML profiles that can be used to describe the architecture of multi-models and provide machine-assisted configuration of co-simulations and other analyses. This section provides a description of these profiles, how standard SysML can be used within INTO-CPS, and the relationship between these two uses.

Chapter 7: Initial Multi-modelling This chapter looks at producing an initial multi-model through the creation of abstract, discrete-event FMUs. These simplified FMUs can then be replaced by higher-fidelity versions in more appropriate tools such as 20-sim. This is referred to as a “DE-first” approach [4].

Chapter 8: Modelling Networks in Multi-models This chapter describes how to also model realistic communications between controllers in an FMI setting. This chapter describes one approach: introducing an FMU that represents an abstract communication mechanism, the *ether*. Guidance on the consequences of adopting such an approach is included, as well as extensions to cover quality-of-service modelling.

Chapter 9: Design Space Exploration This chapter gives guidance on DSE, including the types of search algorithms that can be used to explore a design space, and how the SysML profile extensions help in the design of experiments.

Differences from Previous Versions

This document builds on previous versions Deliverables D3.1a [5] and D3.2a [6]). Some material is retained and updated, while other material is entirely new. The following list gives an overview of new and updated material for each section:

Concepts and Terminology appeared in the previous version. The concepts base has been stable in the final year of the project.

Getting Started with INTO-CPS has been heavily revised from previous "workflows" section in response to end user interactions and feedback.

Traceability and Provenance is entirely new.

Requirements Engineering appeared in the previous version.

SysML and Multi-modelling has been updated significantly to present a comprehensive overview of SysML in the INTO-CPS context, using new and revised material.

Initial Multi-modelling appeared in the previous version.

Modelling Networks in Multi-models appeared previously.

Design Space Exploration has been revised to include description of how to select the algorithm to use and an outline of an iterative search approach.

Chapter 2

Concepts and Terminology

This section introduces the basic concepts used in the INTO-CPS project. CPSs bring together domain experts from diverse backgrounds, from software engineering to control engineering. Each discipline has developed their own terminologies, principles and philosophy for years — in places they use similar terms for quite different meanings and different terms that have the same meaning. In addition, the INTO-CPS project aims to produce a tool chain for CPS engineering resulting in the need for common tool-based terminology. INTO-CPS requires experts from diverse fields to work collaboratively, so this section gives some core concepts of INTO-CPS that will be used throughout the project. We divide the concepts into several broad areas in the remainder of this section.

2.1 Systems

A ***System*** is defined as being "a combination of interacting elements organized to achieve one or more stated purposes" [7]. Any given system will have an ***environment***, considered to be everything outside of the system. The behaviour exhibited by the environment is beyond the direct control of the developer [8]. We also define a ***system boundary*** as being the common frontier between the system and its environment. The definition of the system boundary is application-specific [8].

Cyber-Physical Systems (CPSs) refer to "ICT systems (sensing, actuating, computing, communication, etc.) embedded in physical objects, interconnected (including through the Internet) and providing citizens and businesses with a wide range of innovative applications and services" [9, 10].

A ***System of Systems (SoS)*** is a "collection of constituent systems that pool their resources and capabilities together to create a new, more complex system which offers more functionality and performance than simply the sum of the constituent systems" [11]. CPSs may exhibit the characteristics of SoSs.

2.2 Models

In the INTO-CPS project, we concentrate on "model-based design" of CPSs. A ***model*** is a potentially partial and abstract description of a system, limited to those components and properties of the system that are pertinent to the current goal [11]. A model should be "just complex enough to describe or study the phenomena that are relevant for our problem context" [12]. Models should be abstract "in the sense that aspects of the product not relevant to the analysis in hand are not included" [13]. A model "may contain representations of the system, environment and stimuli" [14][1].

In a CPS model, we model systems with cyber, physical and network elements. These components are often drawn from different domains, and are modelled in a variety of languages, with different notations, concepts, levels of abstraction, and semantics, which are not necessarily easily mapped one to another. This heterogeneity presents a significant challenge for simulation in CPSs [11]. In INTO-CPS we use ***continuous time (CT)*** and ***discrete event (DE)*** models to represent physical and cyber elements as appropriate. A CT model has state that can be changed and observed *continuously* [12] and is described using either explicit continuous functions of time either implicitly as a solution of differential equations. A DE model has state that can be changed and observed only at fixed, *discrete*, time intervals [12]. The approach used in the DESTECS project was to use *co-models* – "a model comprising a DE model, a CT model and a contract" [8]. In INTO-CPS we propose the use of ***multi-models*** – "comprising multiple ***constituent*** DE and CT models". Related to this is a ***Hybrid Model***, which contains both DE and CT elements.

A ***requirement*** may impose restrictions, define system capabilities or identify qualities of a system and should indicate some value or use for the different stockholders of a CPS. ***Requirements Engineering (RE)*** is the process of the specification and documentation of requirements placed upon a CPS. Requirements may be considered in relation to different ***contexts*** – that is the point of view of some system component or domain, or interested stakeholder.

We cover the main features of the notations used in INTO-CPS in Section 2.5. Here we consider some general terms used in models. A ***design parameter*** is a property of a model that can be used to affect the model's behaviour, but remains constant during a given simulation [8]. A ***variable*** is feature of a model that may change during a given simulation [8]. ***Non-functional properties (NFPs)*** pertain to characteristics other than functional correctness. For example, reliability, availability, safety and performance of specific functions or services are NFPs that are quantifiable. Other NFPs may be more difficult to measure [15].

The activity of creating models may be referred to as ***modelling*** [14] and related terms include ***co-modelling*** and ***multi-modelling***. A ***workflow*** is a sequence of ***activities*** performed to aid in modelling. A workflow has a defined purpose, and may cover a subset of the CPS engineering development lifecycle.

The term ***architecture*** has many different definitions, and range in scope depending upon the scale of the product being 'architected'. In the INTO-CPS project, we use the simple definition from [16]: "an architecture defines the major elements of a system, identifies the relationships and interactions between the elements and takes into account process. Those elements are referred to as ***components***. An architecture involves both a definition of structure and behaviour. Importantly, architectures are not static but must evolve over time to reflect the change in a system as it evolves to meet changes to its requirements". In a CPS architecture, components may be either ***cyber components*** or ***physical components*** corresponding to some functional logic or an entity of the physical world respectively.

[1]Further discussion is required in the final year of INTO-CPS regarding the definition of aspects of models in particular; environment models, test models in RT-Tester and their correspondence in the INTO-CPS SysML profile.

In INTO-CPS we consider both a ***holistic architecture*** and a ***design architecture***. An example of their use is given in Chapter 6. The aim of a holistic architecture is to identify the main units of functionality of the system reflecting the *terminology and structure of the domain of application*. It describes a conceptual model that highlights the main units of the system architecture and the way these units are connected with each other, taking a holistic view of the overall system. The design architectural model of the system is effectively a multi-model. The INTO-CPS SysML profile [17] is designed to enable the specification of CPS design architectures, which emphasises a decomposition of a system into ***subsystems***, where each subsystem is an assembly of cyber and physical components and possibly other subsystems, and modelled separately in isolation using a special notation and tool designed for the domain of the subsystem. ***Evolution*** refers to the ability of a system to benefit from a varying number of alternative system components and relations, as well as its ability to gain from the adjustments of the individual components' capabilities over time (Adjusted from SoS [18]).

Considering the interactions between components in a system architecture, an ***interface*** "defines the boundary across which two entities meet and communicate with each other" [11]. Interfaces may describe both digital and physical interactions: digital interfaces contain descriptions of operations and attributes that are *provided* and *required* by components. Physical interfaces describe the flow of physical matter (for example fluid and electrical power) between components.

There are many methods of describing an architecture. In the INTO-CPS project, an ***architecture diagram*** refers to the symbolic representation of architectural information contained in a model. An ***architectural framework*** is a "defined set of viewpoints and an ontology" and "is used to structure an architecture from the point of view of a specific industry, stakeholder role set, or organisation. [11]. In the application of an architecture framework, an ***architectural view*** is a "work product (for example an architecture diagram) expressing the architecture of a system from the perspective of specific system concerns" [16].

The INTO-CPS SysML profile comprises diagrams for architectural modelling and ***design space exploration*** specification. There are two architectural diagrams. The ***Architecture Structure Diagram (ASD)*** specialises SysML block definition diagrams to support the specification of a system architecture described in terms of a system's components. ***Connections Diagrams (CDs)*** specialise SysML internal block diagrams to convey the internal configuration of the system's components and the way they are connected. The system architecture defined in the profile should inform a co-simulation multi-model and therefore all components interact through connections between flow ports. The profile permits the specification of ***cyber*** and ***physical*** components and also components representing the ***environment*** and ***visualisation*** elements. The INTO-CPS SysML profile includes three design space exploration diagrams: a ***parameters diagram***; an ***objective diagram***; and a ***ranking diagram***. See Section 2.4 for concepts relating to design space exploration.

2.3 Tools

The ***INTO-CPS tool chain*** is a collection of software tools, based centrally around FMI-compatible co-simulation, that supports the collaborative development of CPSs. The ***INTO-CPS Application*** is a front-end to the INTO-CPS tool chain. The application allows the specification of the co-simulation configuration, and the co-simulation execution itself. The application also provides access to features of the tool chain without an existing user interface (such as design space exploration and model checking). Central to the INTO-CPS tool chain is the use of the Functional Mockup Interface (FMI) standard.

The ***Functional Mockup Interface (FMI)*** is a tool-independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code [19]. Part of the FMI standard for model exchange is specification of a ***model description*** file. This is an XML file that supplies a description of all properties of a model (for example input/output variables). A ***Functional Mockup Unit (FMU)*** is a tool component that implements FMI. Data exchange between FMUs and the synchronisation of all simulation solvers [19] is controlled by a ***Master Algorithm***.

Co-simulation is the simultaneous, collaborative, execution of models and allowing information to be shared between them. The models may be CT-only, DE-only or a combination of both. The ***Co-simulation Orchestration Engine (COE)*** combines existing co-simulation solutions (FMUs) and scales them to the CPS level, allowing CPS multi-models to be evaluated through co-simulation. This means that the COE implements a ***Master Algorithm***. The COE will also allow real software and physical elements to participate in co-simulation alongside models, enabling both Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation.

In the INTO-CPS Application, a ***project*** comprises: a number of FMUs, optional source models (from which FMUs are exported); a collection of ***multi-models***; and an optional SysML architectural model. A multi-model includes a list of FMUs, defined instances of those FMUs, specified connections between the inputs/outputs of the FMU instances, and defined values for design parameters of the FMU instances. For each multi-model a ***co-simulation configuration*** defines the step size configuration, start and end time for the co-simulation of that multi-model. Several configurations can be defined for each multi-model.

Code generation is the transformation of a model into generated code suitable for compilation into one or more target languages (e.g. C or Java).

The INTO-CPS project considers two tool-supported methods for recording the rationale of design decisions in CPSs. ***Traceability*** is the association of one model element (e.g. requirements, design artefacts, activities, software code or hardware) to another. ***Requirements traceability*** "refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" [20]. ***Provenance*** "is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" [21]. In INTO-CPS traceability between model elements defined in the various modelling tools is achieved through the use of ***OSLC messages***, handled by a traceability ***daemon tool***. This supports the ***impact analysis*** and general ***traceability queries***.

Two broad groups of users are considered in the INTO-CPS project. A ***Tool Chain User*** is an individual who uses the INTO-CPS Tool Chain and its various analysis features. A ***Foundations Developer*** is someone who uses the developed foundations and associated tool support (see Section 2.6) to reason about the development of tools.

2.4 Analysis

Design-Space Exploration (DSE) is "an activity undertaken by one or more engineers in which they build and evaluate [multi]-models in order to reach a design from a set of requirements" [8]. "The ***design space*** is the set of possible solutions for a given design problem" [8]. Where two or more models represent different possible solutions to the same problem, these are considered to be ***design alternatives***. In INTO-CPS design alternatives are defined using either a range of parameter values or different multi-models. Each choice involves making a selection from alternatives on the basis of an ***objective*** – criteria or constraints that are important to the developer, such as cost or performance. The alternative selected at each point constrains the range of design alternatives that may be viable next steps forward from the current position. Given a collection of alternatives with corresponding objective results, a ***ranking*** may be applied to determine the 'best' design alternative.

Test Automation (TA) is defined as the machine assisted automation of system tests. In INTO-CPS, we concentrate on various forms of ***model-based testing*** – centering on testing system models, against the requirements on the system. The ***System Under Test (SUT)*** is "the system currently being tested for correct behaviour. An alias for system of interest, from the point of view of the tester" [11]. The SUT is tested against a collection of ***test cases*** – a finite structure of input and expected output [22], alongside a ***test model***, which specifies the expected behaviour of a system under test [23]. TA uses a ***test suite*** – a collection of ***test procedures***. These test procedures are detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases [24].

INTO-CPS considers three main types of test automation: ***Hardware-in-the-Loop (HiL)***, ***Software-in-the-Loop (SiL)*** and ***Model-in-the-Loop (MiL)***. In ***HiL*** there is (target) hardware involved, thus the FMU is mainly a wrapper that interacts (timed) with this hardware; it is perceivable that realisation heavily depends on hardware interfaces and timing properties. In ***Software-in-the-Loop (SiL)*** testing the object of the test execution is an FMU that contains a software implementation of (parts of) the system. It can be compiled and run on the same machine that the COE runs on and has no (defined) interaction other than the FMU-interface. Finally, in ***Model-in-the-Loop (MiL)*** the test object of the test execution is a (design) model, represented by one or more FMUs. This is similar to the SiL (if e.g., the SUT is generated from the design model), but MiL can also imply that running the SUT-FMU has a representation on model level; e.g., a playback functionality in the modelling tool could some day be used to visualise a test run.

Model Checking (MC) exhaustively checks whether the model of the system meets its specification [25], which is typically expressed in some temporal logic such as ***Linear Time Logic (LTL)*** [26] or ***Computation Tree Logic (CTL)*** [27]. As opposed to testing, model checking examines the entire state space of the system and is thus able to provide a correctness proof for the model with respect to its specification. In INTO-CPS, we can concentrate on ***Bounded Model Checking (BMC)*** [28, 29, 30], which is based on encodings of the system in propositional logic, for a timed variant of LTL. The key idea of this approach is to represent the semantics of the model as a Boolean formula and then apply a ***Satisfiability Modulo Theory (SMT)*** [31] solver in order to check whether the model satisfies its specification. A powerful feature of model checking is that, if the specification is violated, it provides a counterexample trace that shows exactly how an undesired state of the system can be reached [32].

2.5 Existing Tools and Languages

The INTO-CPS tool chain uses several existing modelling tools. ***Overture***[2] supports modelling and analysis in the design of discrete, typically, computer-based systems using the ***VDM-RT*** notation. VDM-RT is based upon the ***object-oriented*** paradigm where a model is comprised of one or more ***objects***. An object is an instance of a ***class*** where a class gives a definition of zero or more ***instance variables*** and ***operations*** an object will contain. Instance variables define the identifiers and types of the data stored within an object, while operations define the behaviours of the object.

The ***20-sim***[3] tool can represent continuous time models in a number of ways. The core concept is that of connected ***blocks***. ***Bond graphs*** may implement blocks. Bond graphs offer a domain-independent description of a physical system's dynamics, realised as a directed graph. The vertices of these graphs are idealised descriptions of physical phenomena, with their edges (***bonds***) describing energy exchange between vertices. Blocks may have input and output ***ports*** that allow data to be passed between them. The energy exchanged in 20-sim is the product of ***effort*** and ***flow***, which map to different concepts in different domains, for example voltage and current in the electrical domain.

OpenModelica[4] is an open-source ***Modelica***-based modelling and simulation environment. Modelica is an "object-oriented language for modelling of large, complex, and heterogeneous physical systems" [33]. Modelica models are described by ***schematics***, also called ***object diagrams***, which consist of connected components. Components are connected by ports and are defined by sub components or a textual description in the Modelica language.

Modelio[5] is an open-source modelling environment supporting industry standards like UML and SysML. INTO-CPS will make use of Modelio for high-level system architecture modelling using the ***SysML*** language and proposed extensions for CPS modelling. The systems modelling language (SysML) [34] extends a subset of the UML to support modelling of heterogeneous systems.

2.6 Formalisms

The ***semantics*** of a language describes the meaning of a (grammatically correct) program [35] (or model). There are different methods of defining a language semantics: ***structural operational semantics***; ***denotational semantics***; and ***axiomatic semantics***.

A structural operational semantics (SOS) describes how the individual steps of a program are executed on an abstract machine [36]. An SOS definition is akin to an interpreter in that it provides the meaning of the language in terms of relations between beginning and end states. The relations are defined on a per-construct basis. Accompanying the relations are a collection of semantic rules which describe how the end states are achieved. Where an operational semantics defines how a program is executed, a denotational approach defines a language in terms of denotations, in the form of abstract mathematical objects, which represent the semantic function that maps over the inputs and outputs of a program [37].

[2]`http://overturetool.org/`
[3]`http://www.20sim.com/`
[4]`https://www.openmodelica.org/`
[5]`http://www.modelio.org/`

The Unifying Theories of Programming (UTP) [38] is a technique to for describing language semantics in a unified framework. A theory of a language is composed of an ***alphabet***, a ***signature*** and a collection of ***healthiness conditions***.

The Communicating Sequential Processes ***CSP*** notation [39] is a formal process algebra for describing communication and interaction. ***INTO-CSP*** is a version of CSP, which will be used to provide a model for the SysML-FMI profile, FMI, VDM-RT and Modelica semantics. It is a front end for a UTP theory of reactive concurrent continuous systems customised for the needs of INTO-CPS. ***Hybrid-CSP*** is a continuous version of CSP defined originally by He Jifeng [40]. It will be used as a basis to inform the design of INTO-CSP.

Several forms of verification are enabled through the use of formally defined languages. ***Refinement*** is a verification and formal development technique pioneered by [41] and [42]. It is based on a behaviour preserving relation that allows the transformation of an abstract specification into more and more concrete models, potentially leading to an implementation. ***Proof*** is the process of showing how the validity of one statement is derived from others by applying justified rules of inference [43].

For the purposes of verification in INTO-CPS, and in particular the work of WP2, we make use of the Isabelle/HOL theorem prover and the FDR3 refinement checker. These are not considered part of the INTO-CPS tool chain, and are used in the INTO-CPS project primarily to support the development of foundation work.

Chapter 3

Getting Started with INTO-CPS

This chapter should help you become familiar with the possibilities for collaborative, model-based design offered by the INTO-CPS tool chain. It does this by explaining the types of activities that can be undertaken with support of one or more of the INTO-CPS technologies, and hopefully putting some of the concepts from the previous chapter in context.

Performing one or more of these activities in order, possibly with iterations, forms a "workflow" for using the INTO-CPS technologies. There are many potential workflows, which depend on the users background and intended use for the tools. A key aspect of most workflows is to produce a multi-model, therefore this chapter includes some guidance.

3.1 Activities Enabled by INTO-CPS

The following activities are all enabled by one or more of the INTO-CPS technologies. They are grouped into broad categories and include both existing, embedded systems activities and activities enabled by INTO-CPS, since INTO-CPS extends traditional embedded systems design capabilities towards CPS design. The choice of granularity for defining these activities naturally affects the size of such a list. The level chosen is instructive for describing workflows, but one that does not make the described workflows overly long.

In the following descriptions (and corresponding summary in Table 3.1), we identify the tools that support the activities, where applicable, using the following icons:

The INTO-CPS Application, COE and its extensions.
Modelio.
The Overture tool.
RT-Tester.
OM OpenModelica.
20-sim.

Descriptions of these tools can be found in the concepts base at the beginning of this document in Section 2.5. Those activities in *italics* can be recorded by the traceability features of INTO-CPS, which is described in Chapter 4.

Requirements and Traceability Writing *Design Notes* ( ) includes documentation about what has been done during a design, why a decision was made and so on. *Requirements* ( ) includes requirements gathering and analysis. *Validation* ( ) is any form of validation of a design or implementation against its required behaviour.

Architectural Modelling INTO-CPS primarily supports architectural modelling in SysML. *Holistic Architectural Modelling* ( ) and *Design Architectural Modelling* ( ) are described in Section 6. The former focuses on a domain-specific view, whereas the latter targets multi-modelling using a special SysML profile. The *Export Model Descriptions* ( ) activity indicated passing component descriptions from the Design Architectural Model to other modelling tools.

Modelling The *Import Model Description* (OM) activity means taking a component interface description from the Design Architectural Model into another modelling tool. *Cyber Modelling* ( ) means capturing a "cyber" component of the system, e.g. using a formalism/tool such as VDM/Overture. *Physical Modelling* (OM) means capturing the "physical" component of the system, e.g. in 20-sim or OpenModelica. Collectively, these can be referred to as *Simulation Modelling* (OM) to distinguish from other forms, such as *Architectural Modelling* ( ). *Co-modelling* ( ) means producing a system model with one DE and one CT part, e.g. in Crescendo. *Multi-modelling* ( ) means producing a system model with multiple DE or CT parts with several tools.

Design *Supervisory Control Design* means designing some control logic that deals with high-level such as modal behaviour or error detection and recovery. *Low Level Control Design* means designing control loops that control physical processes, e.g. PID control. *Software Design* is the activity of designing any form of software (whether or not modelling is used). *Hardware Design* means designing physical components (whether or not modelling is used).

Analysis In INTO-CPS, the RT-Tester tool enables the activities of *Model Checking* ( ), *Creating Tests* ( ) and creating a *Test Oracle* ( ) FMU. The *Create a Configuration* ( ) activity means preparing a multi-model for co-simulation. The *Define Design Space Exploration Configurations* ( ) activity means preparing a multi-model for multiple simulations. *Export FMU* (OM) means to generate an FMU from a model of a component. *Co-simulation* ( ) means simulating a co-model, e.g. using Crescendo baseline technology or the COE.

Prototyping *Manual Code Writing* means creating code for some cyber component by hand. *Generate Code* (OM) means to automatically create code from a model of a cyber component. *Hardware-in-the-Loop (HiL) Simulation* ( ) and *Software-in-the-Loop (HiL) Simulation* ( ) mean simulating a multi-model with one or more of the models replaced by real code or hardware.

Table 3.1: Activities in existing embedded systems design workflows or enhanced INTO-CPS workflows.

Requirements Engineering	
Stakeholder Documents	●
Requirement Definition	●
Validation	●
Architectural Modelling	
Holistic Architectural Modelling	●
Design Architectural Modelling	●
Export Model Descriptions	●
Modelling	
Import a Model Description	● 20 sim OM
Physical Modelling (Simulation Modelling)	20 sim OM
Cyber Modelling (Simulation Modelling)	●
Co-modelling	●
Multi-modelling	●
Design	
Supervisory Controller Design	
Low Level Controller Design	
Software Design	
Hardware Design	
Analysis	
Create Tests	Verified
Model Checking	Verified
Create Test Oracle	Verified
Create a Configuration	●
Define Design Space Exploration Configurations	●
Export FMU	● 20 sim OM
Co-simulation	● ●
Prototyping	
Generate Code	● 20 sim OM
Hardware-in-the-Loop (HiL) Simulation	●
Software-in-the-Loop (SiL) Simulation	●
Manual Code Writing	

3.2 Configuring Multi-Models

As discussed in Chapter 2, a multi-model is a collection of FMUs with a configuration file that: defines instances of those FMUs, specifies connections between the inputs/outputs of the FMU instances, defines values for design parameters of the FMU instances, and defines other simulation settings such as a start, end time, and Master algorithm settings. As seen above, creating a multi-model is a key part of using the INTO-CPS tool chain as it is a pre-requisite for many of the analysis techniques that INTO-CPS can perform.

The INTO-CPS Application supports a project, a view of a folder containing source models, generated FMUs, and configuration files for co-simulation (multi-models) as well as configuration files for other analyses (design space exploration, model checking, test automation). Multi-model configurations can be created in three ways:

1. Created manually using the GUI of the INTO-CPS Application; or
2. Generated from a SysML model created in Modelio; or
3. Created manually by editing JSON configuration files

All three approaches produce the same configuration file, so the choice of which to use depends on the engineer's background. Those comfortable with SysML may find it best to follow the SysML route, but this is not required. So those unfamiliar with SysML can use the Application directly. These two approaches are covered in the second and third tutorials in Part III. Manually editing the JSON configuration is an advanced topic that is not covered in the tutorials, but since JSON is human-readable, not complicated with some experimentation.

3.3 First Steps for Users

In this final section of this chapter, and of Part I, we consider a how different types of users might approach the INTO-CPS technologies. As described in Section 1, all new users are recommended to:

- Follow the first tutorial (see Part III) to experience the INTO-CPS Application.
- Import one or two examples from the Examples Compendium (Deliverable D3.6 [2]) into the INTO-CPS Application and interact with them.
- Return to Part II document as and when you require guidance on a particular area.

After initial familiarisation, the following list provides hints on next steps for different types of users, and where to find further information. As a reminder, tutorials are found in Part III[1].

Students Bachelor and Masters students wishing to build multi-models should follow the first few tutorials on adding exporting and adding FMUs. The SysML tutorial can be skipped if desired. Further guidance on exporting FMUs from different tools can be found in the User Manual, Deliverable D4.3a [1].

Individual Engineers Engineers should follow the first few tutorials on adding and exporting FMUs. The SysML tutorial is also recommended. Further guidance on exporting FMUs from different tools can be found in the User Manual, Deliverable D4.3a [1]

Engineering Teams Teams requiring traceability must read Chapter 4 first (and Chapter 5 is also recommended), as traceability must be considered from the outset. The SysML

[1]Updated tutorials supporting newer versions of the tool can be found at `https://github.com/INTO-CPS-Association/training/releases`

tutorial is mandatory, because traceability links begin with requirements and architectural models in Modelio.

Those with Legacy Models A primary goal is to generate an FMU from the tool for your existing models. These can be incorporated into multi-models as described in the second tutorial.

Those wishing to run Design Space Exploration It is necessary to build a multi-model first in order to run a DSE, so the first tutorials should be followed. The SysML tutorial is optional, though useful as the SysML profile includes extensions to help configure DSE analyses. The later tutorials cover DSE, with further guidance in the user manual, Deliverable D4.3a [1], and Deliverable D5.3e [44] (for more technical details).

Those interested in model checking The User Manual, Deliverable D4.3a [1], provides useful insight, with in-depth information found in Deliverable D5.3c [45].

Those interested in formal semantics and analysis The collection of D2.3deliverables [46, 47, 48, 49] provides in-depth information on these aspects of the tool chain, including mechanisation efforts in Isabelle.

Part II

Advanced Topics

Chapter 4

Traceability and Provenance

The technologies in the INTO-CPS tool chain are able to automatically capture traceability information as activities are performed using the various parts in the tool chain. This includes information about who created or modified an artefact (model, simulation result etc.) and which requirements it is linked to. The traceability features of the INTO-CPS tool are powerful, but require a specific workflow to be followed in order to make best use of them. This chapter explains the steps in this workflow.

This chapter appears first in this advanced material as the following chapters, in particularly Chapters 5 and 6, provide key guidance on the first part of the workflow that must be followed in order for traceability to be realised. Those not wishing to use the traceability features can read chapters in any order, driven by their needs or interest. This chapter should be used in conjunction with the User Manual (Deliverable D4.3a [1]), which covers details of how to enable traceability recording in the INTO-CPS Application and baseline tools[1]. Readers interested in detailed specifications of the traceability and provenance features are directed to Methods Progress Report (Deliverable D3.3b [50]), while the tool implementation is described in Deliverables D4.2d [51] and D4.3d [52].

4.1 Traceability Workflow

The INTO-CPS tool chain builds a graph of traceability relations, as there can be multiple relationships between different artefacts. The graph is however tree-like in the sense that there must be some root node(s) to trace from or back too. These root nodes are *requirements*. To use fully the machine-assisted traceability features, it is necessary to initialise the traceability graph by using Modelio from the beginning of the development process. This means that it is necessary to follow these steps:

1. Define requirements through some requirements process (see guidance in Chapter 5);
2. Create a Requirements Diagram (RD) in Modelio representing these requirements;
3. Create an Architecture Structure Diagram (ASD) and Connections Diagram (CD) describing the multi-model;
4. Link each requirement to one «`EComponent`» (FMU);
5. Export model descriptions for each «`EComponent`»;
6. Import model descriptions into baseline tools; and

[1]Traceability is turned off by default as it can be intrusive if the right workflow is not followed.

7. Generate a multi-model configuration from the CD.

After these steps, the traceability graph will then be updated by the baseline tools as models are created from the model descriptions, FMUs are exported and so on, and co-simulation runs and results will be recorded by the INTO-CPS Application. Therefore, by following this workflow it is possible to take advantage of the machine-assisted traceability within INTO-CPS. By performing the required manual input of requirements and links to SysML elements, it is then possible to automatically trace forward to models, FMUs and simulation results, and to trace backwards from these artefacts to individual requirements.

4.2 What Artefacts are Traced?

Traceability in the INTO-CPS tool chain is based upon a study of the actions performed when using the INTO-CPS tool chain, the artefacts that are used and produced and a combination of two existing standards, the W3C's Prov [2] and the OMG's OSLC [3]. The combination of these resulted in the INTO-CPS traceability ontology that captures in detail all elements in the INTO-CPS workflow and describes the relationships between them. The complete ontology is presented in deliverable D3.3b [50] and a summary is presented here.

Traceability data is inherently a graph based structure based upon nodes and the connections between them, and Prov provides basic types for those nodes along with list of relationships that may exist between them. The three types of nodes are: Entities, things that may be produced or used during a development process; Activities, are things that act upon and make use of entities; and Agents, objects that have responsibility for entities and activities. The Prov relations then allow then connection of nodes such as an activity may use an entity, and an entity may be generated by an activity.

The combination of the Prov nodes and relations supports the representation of the processes that lead to the generation of a particular entity, but it does not support connection of those entities to requirements. OSLC contains a set of specifications, each of which defines a list of relations that it supports between entities. In the case of the INTO-CPS traceability, parts of the OSLC architecture management and requirements management specifications are employed, these allow the connection of entities to requirements via a 'satisfies' relation indicating the entity attempts to address the needs of the requirement, additionally it allows the connection of simulation results to requirements via a 'verifies' relation indicating that the requirement has been met.

The INTO-CPS traceability ontology breaks the INTO-CPS workflow down into activities that, while not atomic if we consider a user's interaction with a particular tool, could be considered atomic when viewing the process of developing a CPS. Figure 4.1 shows the traceability links recorded during one step in the development of a line following robot. In this example, the requirements, R1 & R2, already exist in the architecture models and the user has created an ASD to decompose the proposed robot into components. The user has, at the same time, associated the blocks within the ASD with the the requirements that each block aims to satisfy. When the user saves the updates architecture model, the Modelio tool records the user's 'Architecture Modelling' activity, along with references to the ASD, the blocks it contains and the newly created links between the blocks and the requirements. Here the *used*, *wgb* (short for 'was

[2] `https://www.w3.org/TR/prov-overview/`
[3] `http://open-services.net/`

generated by'), *assoc* (short for 'associated with') and *attrib* (short for 'attributed to') are links that come from the Prov standard. The *OSLC_Sat* (short for 'satisifies') comes from the OSLC requirements management specification.

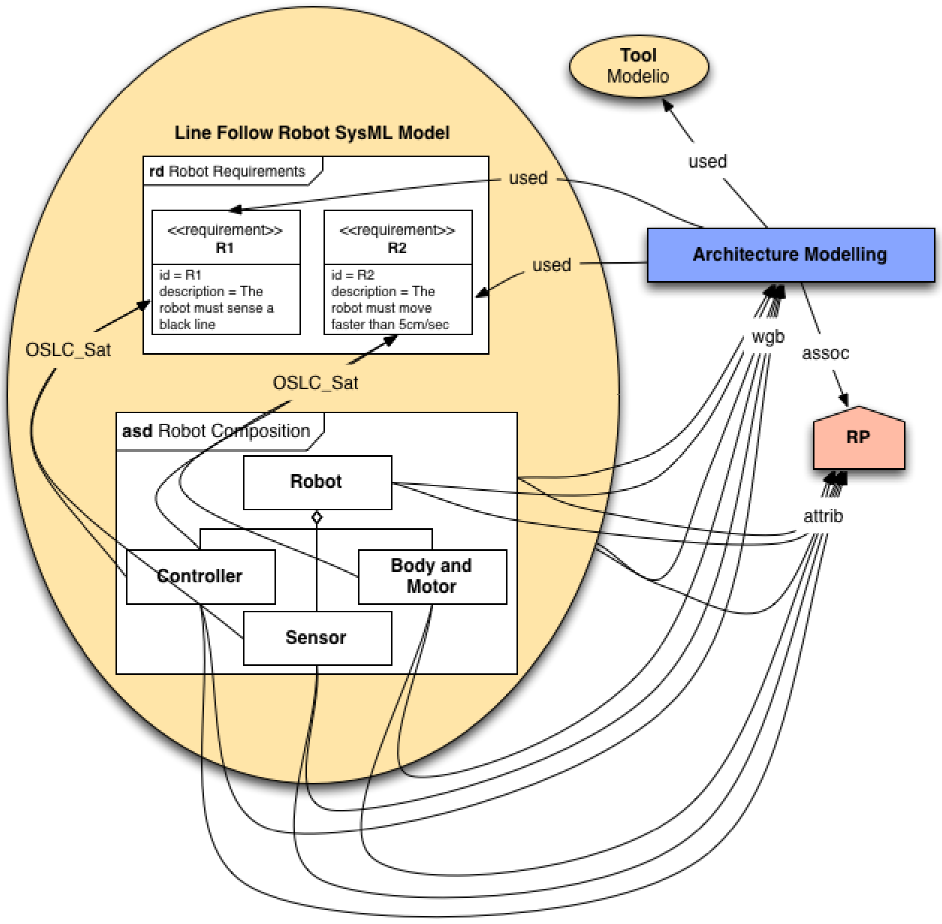


Figure 4.1: Traceability links captured during the production of an ASD for a line following robot.

A development project will likely consist of many instances of the activities identified in the ontology being performed and together they form a traceability graph. Figure 4.2 shows a simplified view of a traceability graph with some steps removed for brevity. At the top of the graph we see the architecture modelling step described previously, that produces an architecture model. From the architecture, model description files are exported to start the production of the simulation models. In turn the simulation models are exported as FMUs and the FMUs are used to produce simulation results. Key to the traceability graph then are the 'used' and 'wgb' connections that can be used by a query to determine from where each entity was generated. By following these links back from any entity to the individual blocks within the architecture model, it is possible to determine which requirement(s) each should satisfy. Finally when simulation results are output, these may be linked back to the relevant requirements, stating whether a requirement was verified or violated by that result.

The traceability ontology captures the significant activities and entities that form the INTO-CPS workflow. For example a development project might see the following activities recorded in the traceability graph: *Requirements Management*, *Architecture Modelling*, *Architecture Configuration Creation Model Description Export*, *Simulation Modelling*, *FMU Export*, *Configuration*

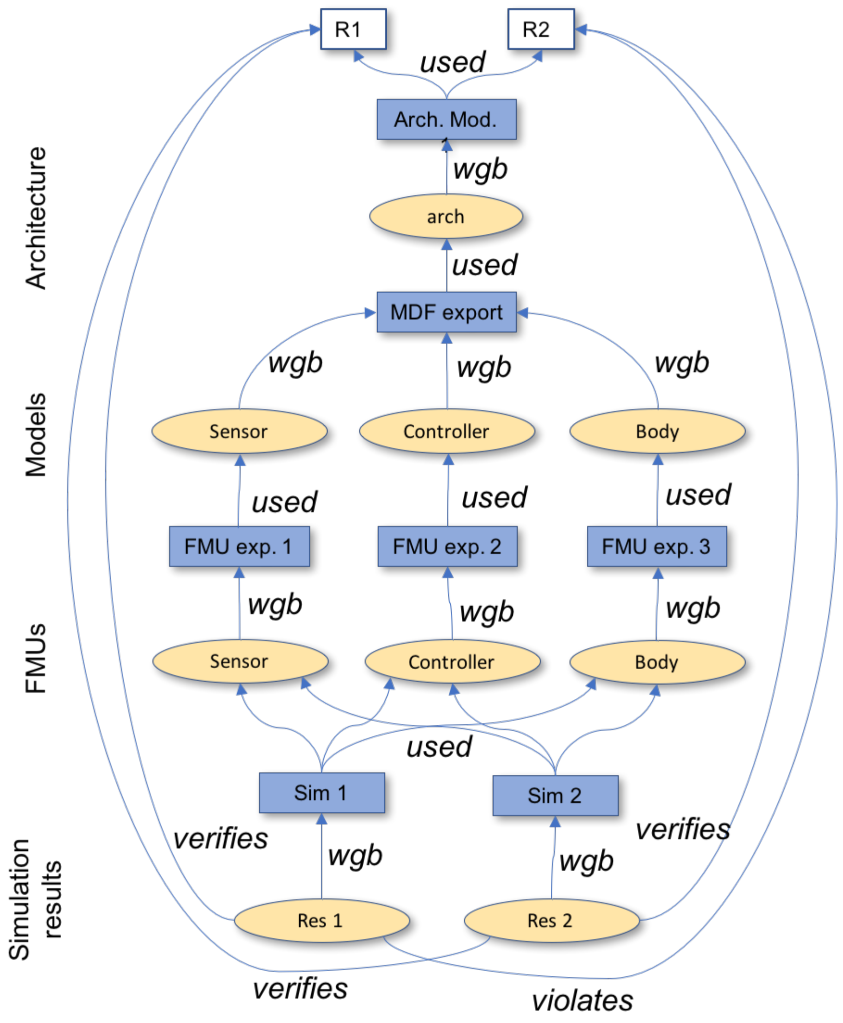


Figure 4.2: Traceability links captured during the production of an ASD for a line following robot.

Creation, *Simulation Configuration Creation* and then *Simulation*. These activities are connected in the workflow by the entities they create and use, so the example would see the traceability graph containing records of: *Archtecture Structure Diagram*, *Architecture SubSystem*, *Architecture Connection Diagram*, *Model Description File*, *Simulation Model*, *FMU*, *Multi-model Configuration*, *Simulation Configuration* and *Simulation Result*. Alongside these will be records of the agent(s), who are both associated with activities and have entities attributed to them.

4.3 Traceability Queries

The traceability graph created by the INTO-CPS tool chain uses a graph database tool called *Neo4J*. Once a graph has been built, queries can be executed over the graph to perform both forwards and backwards traceability. Below are some types of queries that can be executed over the graphs. The INTO-CPS Application supports some of these queries with the GUI, and the rest through inline access to the Neo4J console.

1. Impact analysis
   - Forward traceability (from requirements to entities)
   - Backwards traceability (from FMU to requirements)
   - Backwards traceability (from components to requirements)
2. Simulation sources
   - Find all simulations
   - Find sources and sinks for a simulation
3. Coverage
   - Requirements without architecture elements
   - Requirements without simulation models
   - Requirements without FMUs
   - Requirements without positive simulation results
   - Requirements without any simulation results
4. Code sources
   - Find all generated source code entities
   - Find the models for a given source code entity
5. User impact
   - Find all users in the database
   - Find all artefacts influenced by a user
   - Find all activities performed by a user

Queries are written in Cypher, a query language built into Neo4J. Advanced users or those developing extensions to INTO-CPS can build their own queries in Cypher[4] and execute them using Neo4J directly as described in the User Manual (Deliverable D4.3a [1]).

[4] `https://neo4j.com/developer/cypher-query-language/`

Chapter 5

Requirements Engineering

In this chapter, we consider the requirements engineering (RE) activities for the design of CPSs. Specifically, we consider the specification and documentation of requirements placed upon a CPS. These requirements may, for example, impose restrictions, define system capabilities or identify qualities of a system. The requirements should indicate some value or use for the different stockholders of a CPS.

As described in the previous chapter, traceability needs requirements to be defined as early as possible in a development process, and these must be recorded in Modelio for the machine-assisted traceability information to be recorded accurately. It is therefore appropriate to consider requirements processes for such developments at this stage.

In this remainder of this chapter, we discuss the needs for requirements engineering in CPS development, in particular based on the experience of the industrial partners for INTO-CPS. We describe one possible approach to RE for CPS, specifically adapting the SoS-ACRE approach for systems-of-systems (SoSs) to CPS. Note however that this approach is not mandatory, and in general RE processes and tools vary widely across organisations and domains. For this reason, tool support for traceability in INTO-CPS begins once requirements have been defined and can be added to Modelio. The diagrams described in the example are not part of INTO-CPS SysML specification. Therefore, this chapter should truly be treated as guidance, primarily serving to highlight the nature of RE for CPS, which may be of use for both new and more experienced CPS teams.

5.1 Requirements Engineering and Cyber-Physical Systems

The main issue of concern for RE in CPSs is that of differing domain contexts [53]. In addition, it has been noted that there are overlaps in challenges in CPSs and SoSs [54]— especially independence, evolution and, increasingly, distribution. As described by Lewis et al. [55], as system architectures become more complex, there is often a need to consider requirements and structural architectures during the RE process. The authors suggest that an engineer should identify the system needs, component interactions and stakeholders, and map those needs onto those interested parties.

As research in RE in CPS is a nascent field, we suggest one approach is to adopt RE processes from the SoS world, rather than defining an approach specifically for CPSs. In chapter, we consider SoS-ACRE (System of Systems Approach to Context-based Requirements Engineering) [56], as an example. This approach was adapted from standard systems engineering, and tailored for SoSs— enabling the identification and reasoning about requirements across constituent systems of an SoS and understanding multi-stakeholder contexts. We suggest it might be useful to organisations trying to approach RE for CPS.

INTO-CPS industry partners and RE

At the beginning of the INTO-CPS project, the four industrial partners were surveyed about their use of various technologies and methods, including requirements engineering [57]. Microsoft Excel was quoted as being used by three partners (UTRC, TWT and CLE), IBM Rational Doors used by one partner (UTRC), and Microsoft Word by one partner (AI).

Issues raised by industrial partners include:

- Language/terminology of the requirements not consistent;
- Different people involved in the workflow do not have common understandings of requirements;
- Requirements traceability is considered to be highly inefficient and time consuming;
- Different people have to meet together and generate proofs among each other to validate dependable requirements; and
- Stakeholders do not have a clear vision about the product and tend to disagree on the objectives.

As can be seen, the above issues may be due to not having a rigorous RE approach, but also due to the challenges in CPSs— that of different domains. In this section, we consider how a context-based approach to RE (SoS-ACRE) may be incorporated into the INTO-CPS tool chain, in particular using both the INTO-CPS technologies and the industrial partners' baseline technologies.

5.2 The SoS-ACRE View of Requirements

We first consider the collection of views defined in SoS-ACRE, and their applicability to CPS engineering and the INTO-CPS tool chain. These views could be represented as diagrams in SysML[1], or as we describe, could equally be represented in other tools where these are already used (e.g. Excel). Examples of each view are shown in Figures 5.3, 5.4, 5.1 and 5.2.

Source Element View (SEV) The SEV defines a collection of source materials from which requirements are derived. In SoS-ACRE, a SysML block definition diagram is considered. In INTO-CPS, this view could also be represented using an Excel table or Word document (with each source having a unique identifier), or by simply referring to source documents using OSLC traces.

Requirement Description View (RDV) The RDV is used to define the requirements of a system and forms the core of the requirement definition. SoS-ACRE suggests the use of SysML requirements diagram or in tabulated form, such as through the use of Excel. In addition, specifying requirements in Doors would support this view.

[1]Note that SoS-ACRE is not specifically supported as a Modelio plug-in, but other equivalent diagrams could be used.

Context Definition View (CDV) The CDV is a useful view for CPS engineering in order to explicitly identify interested stakeholders and points of context in the system development, including customers, suppliers and system engineers themselves. In SoS-ACRE, they are defined using SysML block definition diagrams, and could also be represented using an Excel table or Word document (with each context having a unique identifier). This diagram type could be useful when identifying the divide in CT/DE and cyber-physical elements of a system.

Requirement Context View (RCV) In SoS-ACRE, a RCV is defined for each constituent system context identified in CDVs. This is appropriate when there is a set of diverse system owners, which is typical for SoSs and increasingly CPSs. A **Context Interaction View (CIV)** is then defined to understand the overlap of contexts and any common/conflicted views on requirements. In a CPS, however, there may not be such a clear delineation between the owners of constituent system components. However, if we consider the different domains (e.g. CT/DE or cyber/physical divides) as different contexts, then this approach would be useful. In SoS-ACRE, RCVs and CIVs are both defined with SysML use case diagrams. Excel could be used if unique identifiers are defined for contexts and requirements as described earlier.

Validation View (VV) VVs, defined as SysML sequence diagrams in SoS-ACRE, describe validation scenarios for a SoS to ensure each constituent system context understands the correct role of the requirements in the full SoS. This is not an obvious fit in CPS engineering, and therefore not necessarily required.

5.3 The SoS-ACRE RE Process

The SoS-ACRE requirement engineering process may be useful for organisations wishing to better understand requirements for CPSm, particularly across multiple domains. It is a lightweight process, and therefore suitable for small- to medium-sized enterprises. Organisations with established may not feel the need to radically alter their existing practice, but may find it instructive to consider how their current processes might be updated or revised to consider better CPS requirements.

A SoS-ACRE process for CPS should include the following steps:

1. Identify and record source elements. This would be using a SEV, or simply recording paths to relevant files or documents.

2. Record system-level functional and non-functional requirements. Requirements may be derived using RDVs, and we could consider domain-specific requirements (e.g. cyber or physical), or analysis-specific requirement types (e.g. DSE or testing requirements).

3. Model initial System structure using INTO-CPS ASD. This will identify the cyber and physical elements and the domain/phenomena of the CPS. This may also give initial idea of component functionalities, which may lead to a repeat of Step 2 above[2].

4. Define the various contexts in CDVs – both external stakeholders, and if appropriate, contexts for the different components. If only a single system context is defined, then a

[2]In the process of architectural modelling, it may also be necessary to redefine contexts depending on whether different simulation tools, or indeed different components of a model, are better able to provide the requirements of the CPS.

single RCV is defined. However, if multiple contexts are defined for a CPS, then several RCVs are to be defined, along with a CIV to explore requirements from multiple contexts.

5. Trace the requirements through INTO-CPS tool chain models and results. This was covered in the previous chapter, however we revisit it below in the context of requirements.

5.4 Using technologies with SoS-ACRE

As INTO-CPS does not specifically support SoS-ACRE. Indeed INTO-CPS does not mandate and specific approach to RE, because of the wide variety of approaches in industry. We conclude this chapter with an example of how a SoS-ACRE (or other RE process) could be integrated into an INTO-CPS development. We describe a range of permutations of the use of models and documents for recording the requirements engineering process described above. In addition, we include discussions on the links between requirements and architectural models—identified above as a key method for requirements engineering in CPSs.

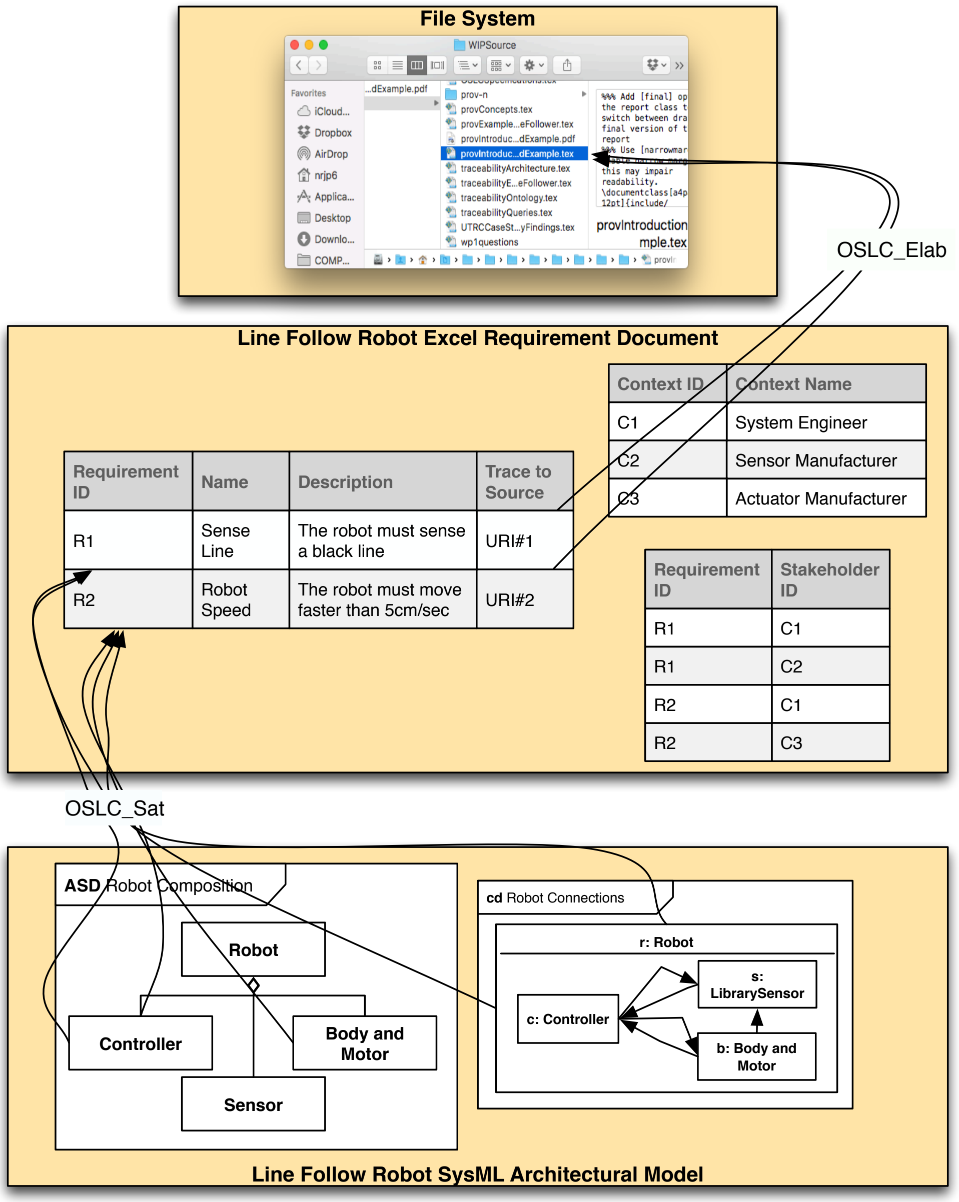
URI, Excel and SysML We first consider an approach using URIs for the source elements, an Excel document (or a collection of Excel tables) for the RDV, CDV, RCV and CIV of SoS-ACRE. A SysML model in Modelio can be used to define the architecture of the multi-model. Internal tracing in Excel can be achieved using identifiers referenced between sheets. Excel requirements can be replicated in Modelio then traced to elements in the INTO-CPS tool chain automatically. Figure 5.1 presents an example with URI, Excel and SysML models and OSLC links between the artefacts.

Excel and SysML The next approach uses Excel to define the SEV and RDV of SoS-ACRE, a SysML model to define the context-oriented views (CDV, RCV and CIV) and a separate architectural model to define the CPS architecture. The Excel requirements can then be mirrored in a Modelio model, and linked to the architectural model. The INTO-CPS traceability features can trace the requirement artefacts to the architectural model. Additional OSLC links could be added manually to link elements of the Excel requirements and context views in a SysML. Figure 5.2 presents an example with URI, Excel and two SysML models with OSLC links between the artefacts.

Single SysML model The next permutation is to use a single SysML model for both requirements engineering and architectural modelling. Such a model will contain all SoS-ACRE views[3] (SEV, RDV, CDV, RCV and CIV), in addition to diagrams defined using the INTO-CPS profile for the CPS composition and connections. Modelling in this way enables trace links to be defined inside a single SysML model. Figure 5.3 presents an example SysML model with trace relationships.

SysML requirements and SysML architectural models The final permutation is to use SysML for both requirements engineering and architectural modelling, however to use two separate models for the two activities (one containing the RE views (SEV, RDV, CDV, RCV and CIV) and another for architectural diagrams (ASD and CD)). We consider this permutation with two SysML models in addition to the single SysML model, because the requirements engineering and architectural modelling activities are often considered separately, with different engineering teams comprised of engineers with specialist skills. As

[3]Note that Modelio does not currently provide an extension for SoS-ACRE, but these views can be realised using existing SysML stereotypes.



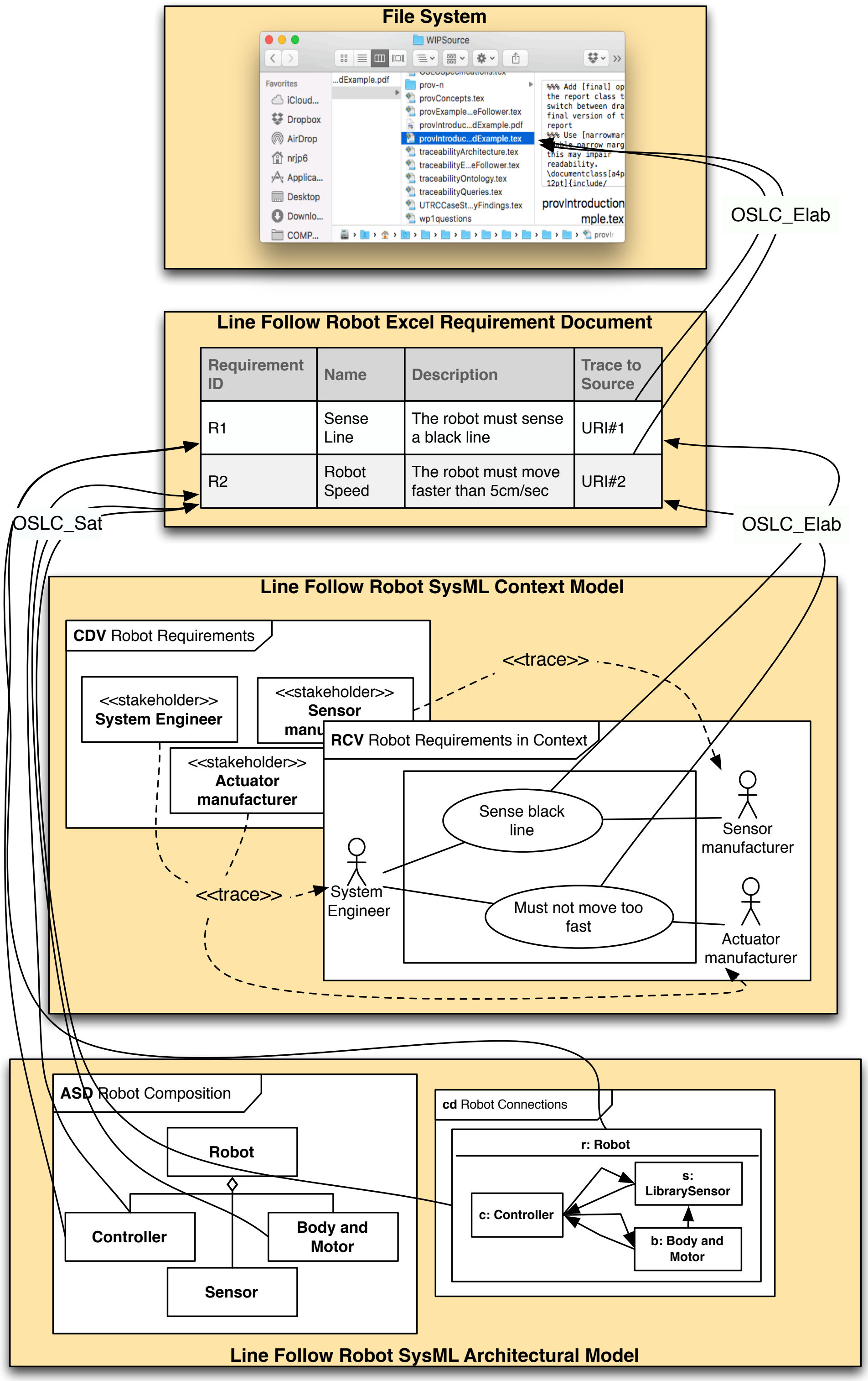
Context ID	Context Name
C1	System Engineer
C2	Sensor Manufacturer
C3	Actuator Manufacturer

Requirement ID	Name	Description	Trace to Source
R1	Sense Line	The robot must sense a black line	URI#1
R2	Robot Speed	The robot must move faster than 5cm/sec	URI#2

Requirement ID	Stakeholder ID
R1	C1
R1	C2
R2	C1
R2	C3

Figure 5.1: URI, Excel and SysML – model overview

such we can assume there are cases where these teams have ownership of different models. Trace links may be used within each individual model (for example, tracing from source elements to requirements in a RE model), and OSLC links defined to trace between requirements elements and architectural elements. Figure 5.4 presents an example with two SysML models with trace relationships and OSLC links between the models.



Requirement ID	Name	Description	Trace to Source
R1	Sense Line	The robot must sense a black line	URI#1
R2	Robot Speed	The robot must move faster than 5cm/sec	URI#2

Figure 5.2: Excel and SysML – model overview

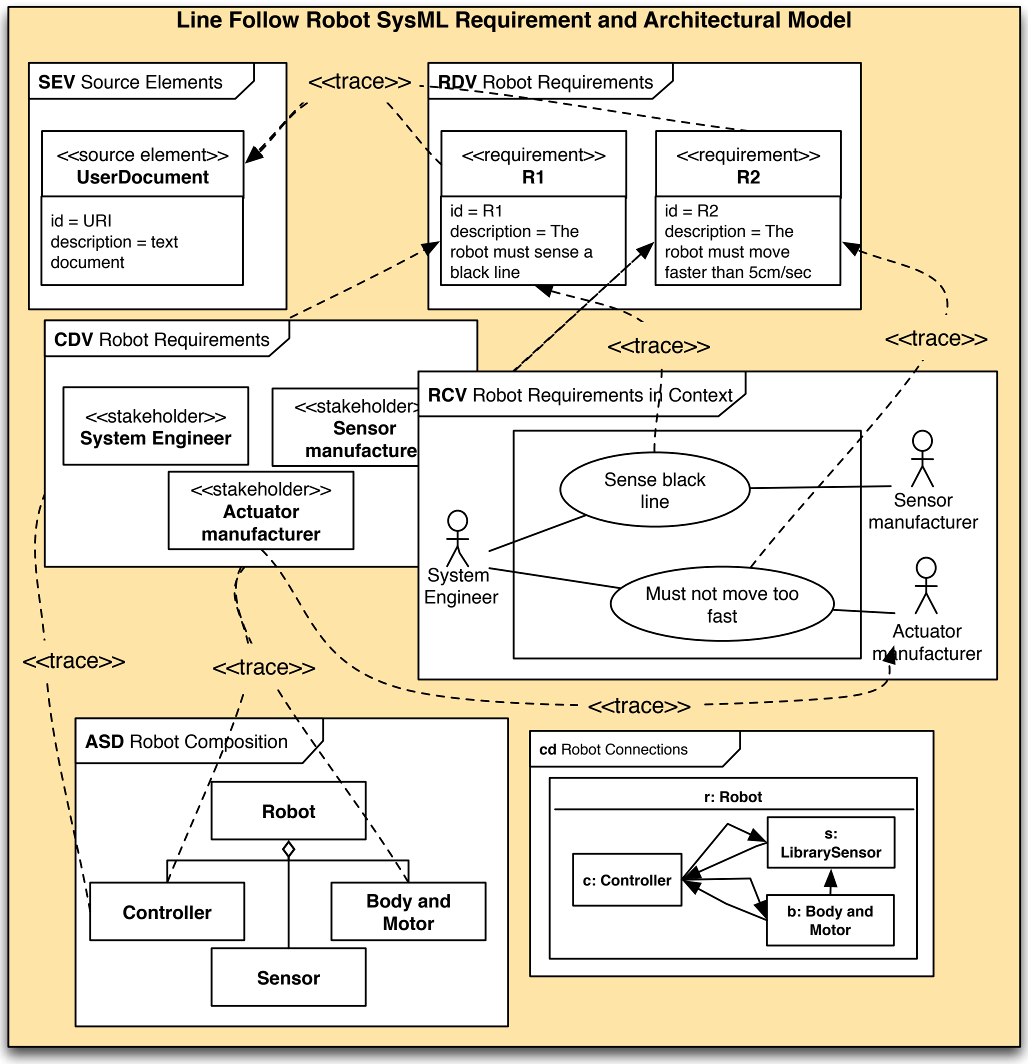


Figure 5.3: Single SysML model – model overview

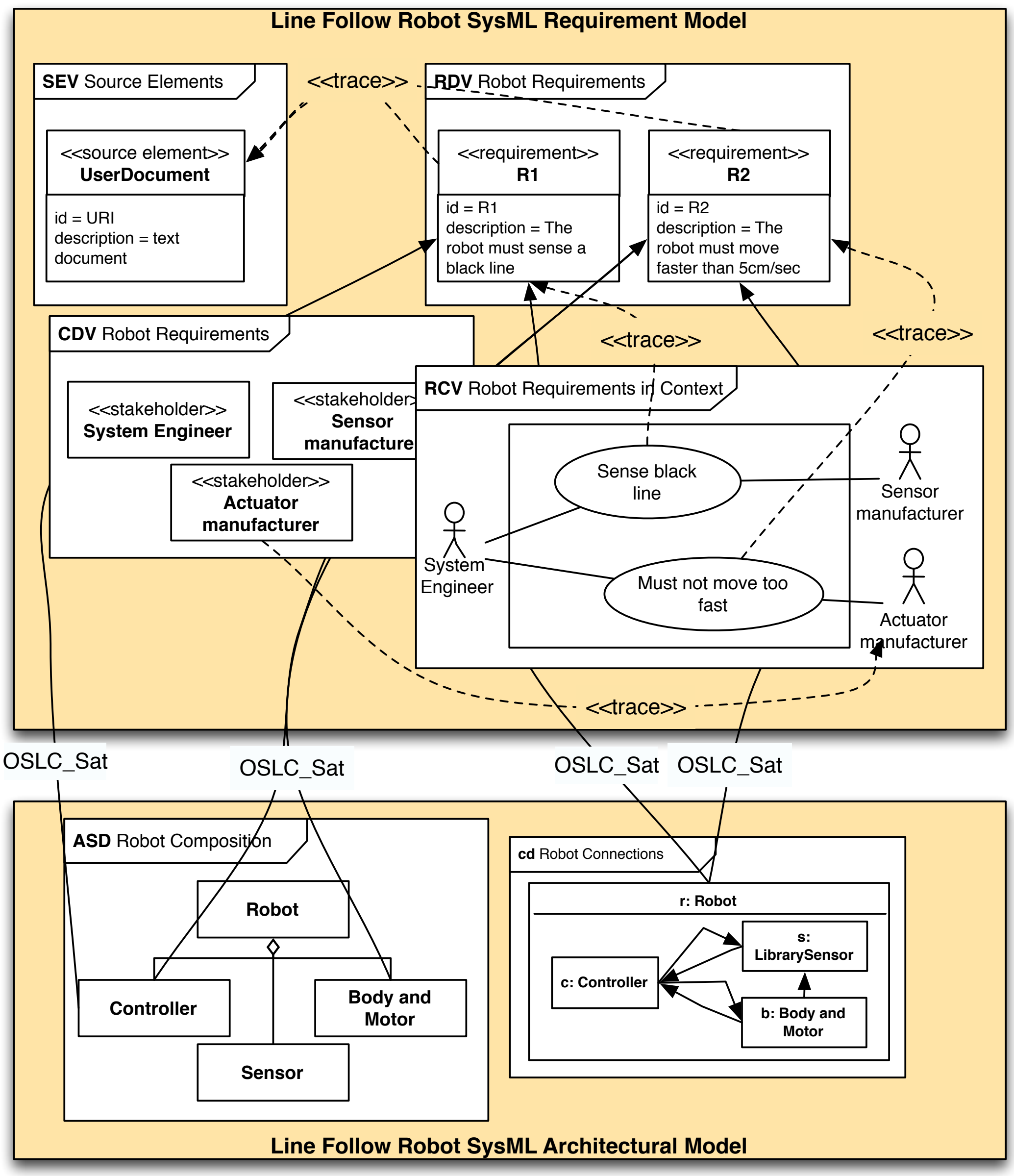


Figure 5.4: SysML requirements and SysML architectural models – model overview

Chapter 6

SysML and Multi-modelling

This chapter describes the use of SysML with the INTO-CPS tool chain. As described previously in Chapter 5, standard SysML can be used as part of a development process to build a model of a system and link elements to requirements. The INTO-CPS tool chain also provides an extended SysML profile that help users to *configure multi-models for co-simulation* and *configure design space exploration (DSE) analysis* [17, 58, 46, 59, 60, 61]. For ease explanation, we describe these separately below, however all the diagrams described are part of a single extended SysML profile.

This chapter summarises the diagrams provided in the two profiles and describe their use in Sections 6.1 and 6.2. The diagrams presented are illustrative, showing the main elements of a diagram; they are not full definitions of the meta-model, which can be found in the documents cited above. All diagrams are supported by the Modelio tool, and we refer readers to the user manual, Deliverable D4.3a [1], for further information on how to use Modelio to draw these diagrams and generate configurations for use in the INTO-CPS Application.

The chapter concludes with an example of the relationship between a *holistic* model created using standard SysML and a *design* model using the INTO-CPS profile, and concludes with a discussion on how to represent non-design elements (such as FMUs that only perform visualisation) in the INTO-CPS profile in Section 6.4.

6.1 SysML Diagrams Describing Multi-models

The multi-modelling SysML profile defines two diagrams for configuring a co-simulation. The INTO-CPS Application can run a co-simulation based on a configuration file, using the JSON format to describe the FMUs, their parameters and connections between them. These can be created manually in a text editor, or from the INTO-CPS Application itself. Alternatively, a configuration can be generated by Modelio from the diagrams defined in this profile. There are two types diagram, the *Architectural Structure Diagram* describing the static structure of FMUs, and the *Connections Diagram* describing their instantiation and connections. These are shown in Figure 6.1.

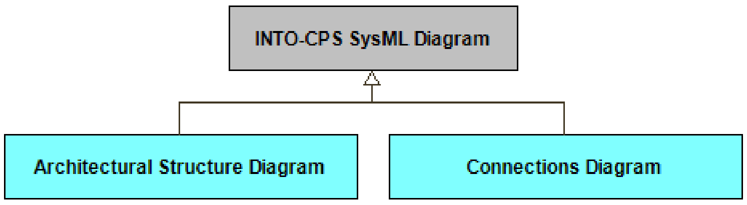


Figure 6.1: Diagrams in the multi-modelling SysML profile

6.1.1 Architectural Structure Diagram

The *Architecture Structure Diagram* (ASD) specialises SysML block definition diagrams (BDDs) to support the specification of a multi-model architecture described in terms of a systems components, which will be represented by FMUs. As shown in Figure 6.2 this diagram must include a «`System`» which is then broken down into zero or more «`Component`» blocks.

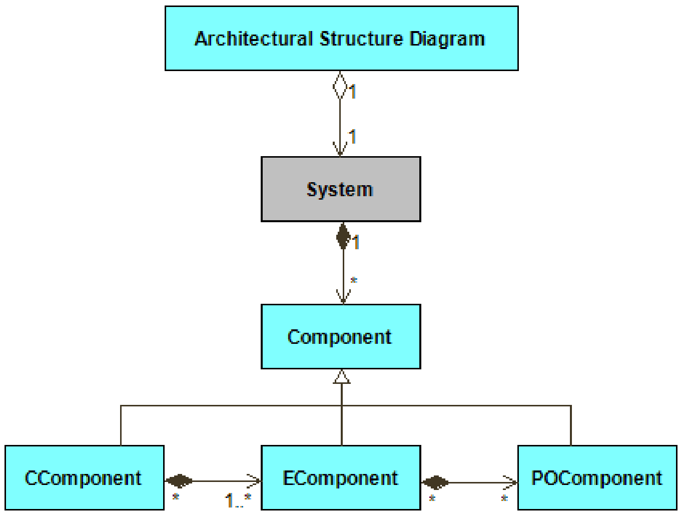


Figure 6.2: *Architectural Structure Diagram* describing FMUs (EComponents) and their hierarchies

There are three types of component block. The «`EComponent`» (encapsulating component) represents a part of a system that will be represented by a single FMU. These blocks have properties indicating which modelling language and tool will be used: `modelType` (*discrete* or *continuous*) and `platform` (*VDMRT*, *TwentySim*, *OM*, and *other*).

An «`EComponent`» can be broken down logically into «`PComponent`» (part-of component) representing an internal element of an «`EComponent`». Both «`EComponent`» and «`PComponent`» blocks can define *variables* and *FlowPorts* that an FMU will have.

The third type of component is a «`CComponent`» (collection component) that allows other components to be grouped logically (it has no ports or behaviours). These can be used to separate design elements within a diagram, as described in Section 6.4. All component blocks have a *kind* that marks their purpose in the model (*cyber*, *physical*, *environment*, *visualisation*).

FMUs are connected by *ports*, and may also present internal state through externally visible *variables*, which can be monitored on a live graph, for example. Both «`EComponent`» and

«PComponent» blocks can define *FlowPort* and *Variable* attributes, as shown in Figure 6.3, which will form the interface of the FMU and are added to the "model description" exported by Modelio.

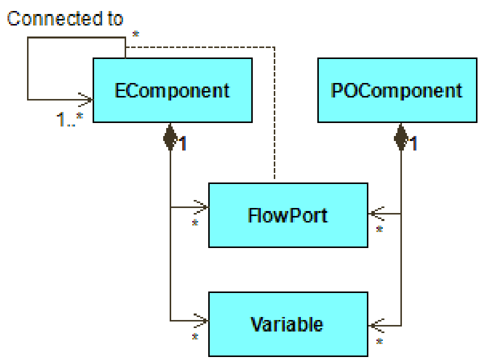


Figure 6.3: Component blocks may define variables and ports

6.1.2 Connections Diagram

The *Connections Diagram* (CD) specialises SysML internal block diagrams to convey the internal configuration of the systems components. Specifically, it describes which FMUs are instantiated (i.e. which «EComponent»s form the ASD), and how the ports are connected. This diagram is used by Modelio to generate multi-model configurations.

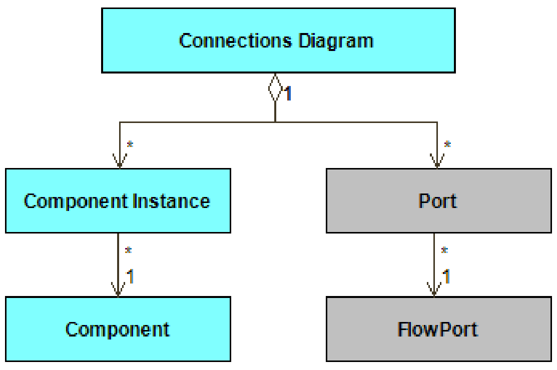


Figure 6.4: *Connections Diagram* describing the static structure of FMUs

6.2 SysML Diagrams Describing Design Space Exploration

The design space exploration (DSE) SysML profile is an addition to the multi-modelling SysML profile described above. As with single co-simulation, the INTO-CPS Application can run a DSE based on a JSON configuration file. These can be created manually in a text editor or edited in the INTO-CPS Application. Alternatively, a configuration can be generated by Modelio, from a set of diagrams defined in the profile. There are five diagram types, which are described below. Further guidance on DSE can be found in Chapter 9.

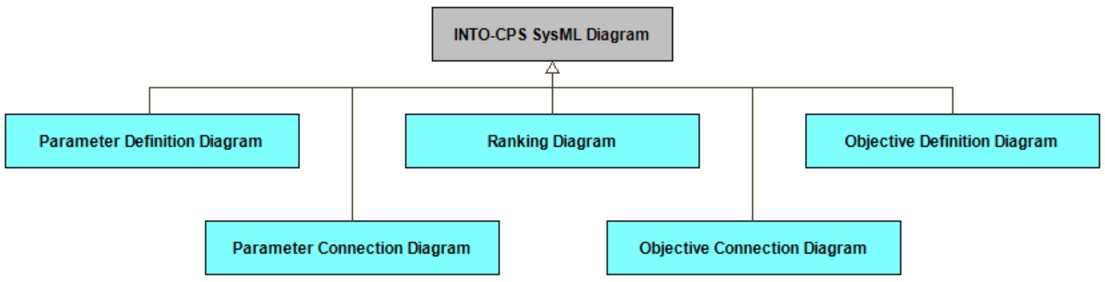

INTO-CPS SysML Diagram
Parameter Definition Diagram
Ranking Diagram
Objective Definition Diagram
Parameter Connection Diagram
Objective Connection Diagram

Figure 6.5: Diagrams in the DSE SysML profile

6.2.1 Objective Definition Diagram

The *Objective Definition Diagram* is used to define the objectives for use during a DSE. Objectives are characterising measures of performance that may be used to determine the relative benefits of competing designs. They are defined as metrics over the results of a co-simulation of a specific design and are used to judge its quality for use in later processing e.g. ranking.

Objectives are described in terms of a name, a script file that will be used to compute them, and the ports that will provide the data they require. As with the *Architectural Structure Diagram* above (Section 6.1.1), this diagram gives the static structure of the objectives; instances of these definitions are created using the *Objective Connection Diagram* below (Section 6.2.2).

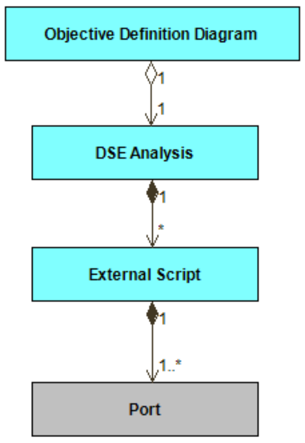


Figure 6.6: *Objective Definition Diagram* describing objectives in a DSE

6.2.2 Objective Connection Diagram

The *Objective Connection Diagram* is used to instantiate objectives defined in the *Objective Definition Diagram* above (Section 6.2.1). The diagrams allow the ports of each instance of the objective to be linked to a data source: either a static value, or a value from data exchanged in the multi-model.

6.2.3 Parameter Definition Diagram

The *Parameter Definition Diagram* is used to define the parameters that will changed for each co-simulation in a DSE. Parameters are described in terms of a name, and a set of values that

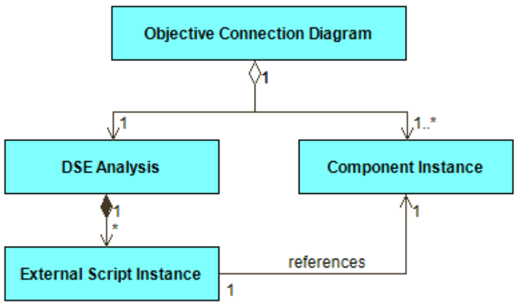


Figure 6.7: *Objective Connections Diagram* linking objectives to data sources

we wish to test. The product of the cardinalities of the set of values for each parameter gives the size of the design space— the total number of simulation required for an exhaustive search. As with the *Architectural Structure Diagram* above (Section 6.1.1), this diagram gives the static structure of the parameters; instances of these definitions are created using the *Parameter Connection Diagram* below (Section 6.2.4).

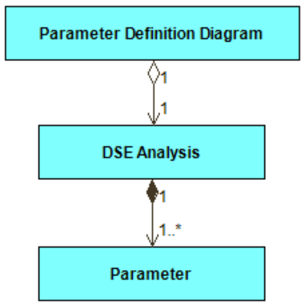


Figure 6.8: *Parameter Definition Diagram* defining parameters and their values

6.2.4 Parameter Connection Diagram

The *Parameter Connection Diagram* is used to instantiate parameters defined in the *Parameter Definition Diagram* above (Section 6.2.3). The diagram allows the parameters to be linked to those provided by the FMUs in the multi-model.

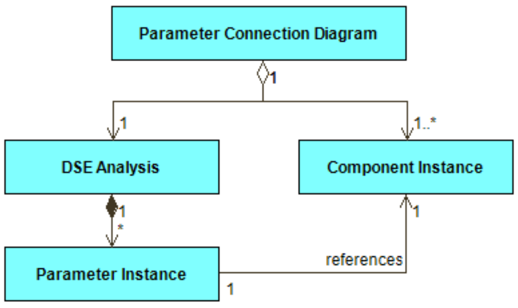


Figure 6.9: *Parameter Connections Diagram* linking parameters to FMUs

6.2.5 Ranking Diagram

The *Ranking Diagram* is used to declare which of the objectives should be used to compare competing designs, and whether lower or higher values for each the objectives is better (i.e. whether to maximise or minimise a value).

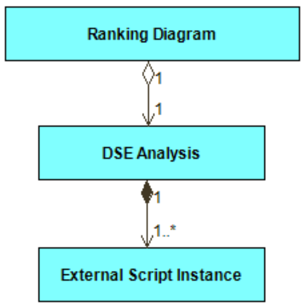


Figure 6.10: *Ranking Diagram* defining how to rank designs based on objectives

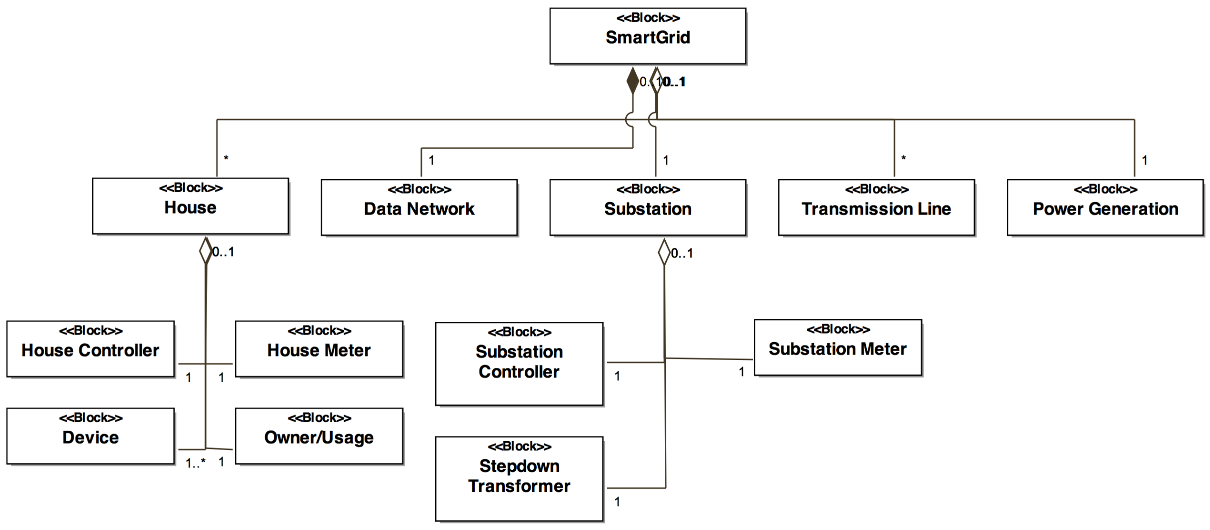
6.3 Holistic and Design Architectural Modelling

A system architecture defines the major components of a system, and identifies their relationships, behaviour and interactions. A model of the architecture is potentially partial (representing some or all of the system) and abstract, limited to those elements pertinent to the modelling goal. In CPS engineering, this goal may include understanding the system in terms of the application domain (a *holistic* model), or capturing the system components in a way that targets multi-modelling (a *design* model).

The diagrams in the two profiles described above divide architectural models into subsystems composed of cyber or physical components. Defining an architecture this way may not be the best approach when designing a system ab initio, with systems comprising entities across different domains requiring diverse domain expertise. Following on from Chapter 5, this section uses a smart grid example to show both holistic and design architectural modelling approaches, and provide some commentary and guidance on how to model in a way which is natural for domain experts, and how to move from holistic to design models when multi-modelling.

Example Introduction

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. 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 5 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.



<<Block>> SmartGrid
0..1
0..1
<<Block>> House
<<Block>> Data Network
<<Block>> Substation
<<Block>> Transmission Line
<<Block>> Power Generation
*
1
1
*
1
0..1
0..1
<<Block>> House Controller
<<Block>> House Meter
<<Block>> Device
<<Block>> Owner/Usage
<<Block>> Substation Controller
<<Block>> Substation Meter
<<Block>> Stepdown Transformer
1
1
1..*
1
1
1
1

Figure 6.11: Block Definition Diagram of Smart Grid

Holistic Architectural Model

A Block Definition Diagram (BDD) of the Smart Grid is given in Figure 6.11. The figure shows that the *Smart Grid* system comprises two top-level physical elements: *Power Generation* and *Transmission Lines*; a single top-level cyber elements: the *Data Network*; and two cyber-physical systems: a *Substation* and several *Houses*. The two elements may be further decomposed. The *Substation* elements is composed of a cyber *Substation Controller* and physical *Substation Meter* and *Step-down Transformer*. The *House* element comprises: a cyber *House Controller*, physical *House Meter* and *Devices*, and an *Owner/Usage Profile*.

An Internal Block Diagram (IBD) of the Smart Grid is given in Figure 6.12. The diagram shows there are two main connection types in the model, corresponding to the physical power connections and the cyber data connections. The model also shows the connections between the cyber and physical parts of the models – currently modelled using data-type connections.

The first type of connection —the physical power connections— show a flow of *Power* from the Power Generation, through the Transmission Lines to the Houses, via the Substation. In the Substation, the Stepdown Transformer is connected to the Substation Meter. Similarly, in each House (only one is shown in the figure), the Power flows through the House Meter to each Device (again only one is shown for readability). The data connections exist between the Substation Controller and House Controllers. The Data Network is explicitly modelled and links the various controllers. Finally, there are links between the cyber controllers and the physical systems. In this model, the Substation Controller is connected to the Substation Meter, and the House Controller is linked to the House Meter and Devices.

Design Architectural Model

Looking at the holistic architecture defined in Figures 6.11 and 6.12 and moving towards a multi-model, we use the INTO-CPS SysML profile to define the architecture of the Smart Grid from the perspective of multi-model. This yields the ASD in Figure 6.13. This structure removes all subsystem structures such that each component is to be realised in a single FMU. Each element is defined as either a *physical* or *cyber* component, with the model type and

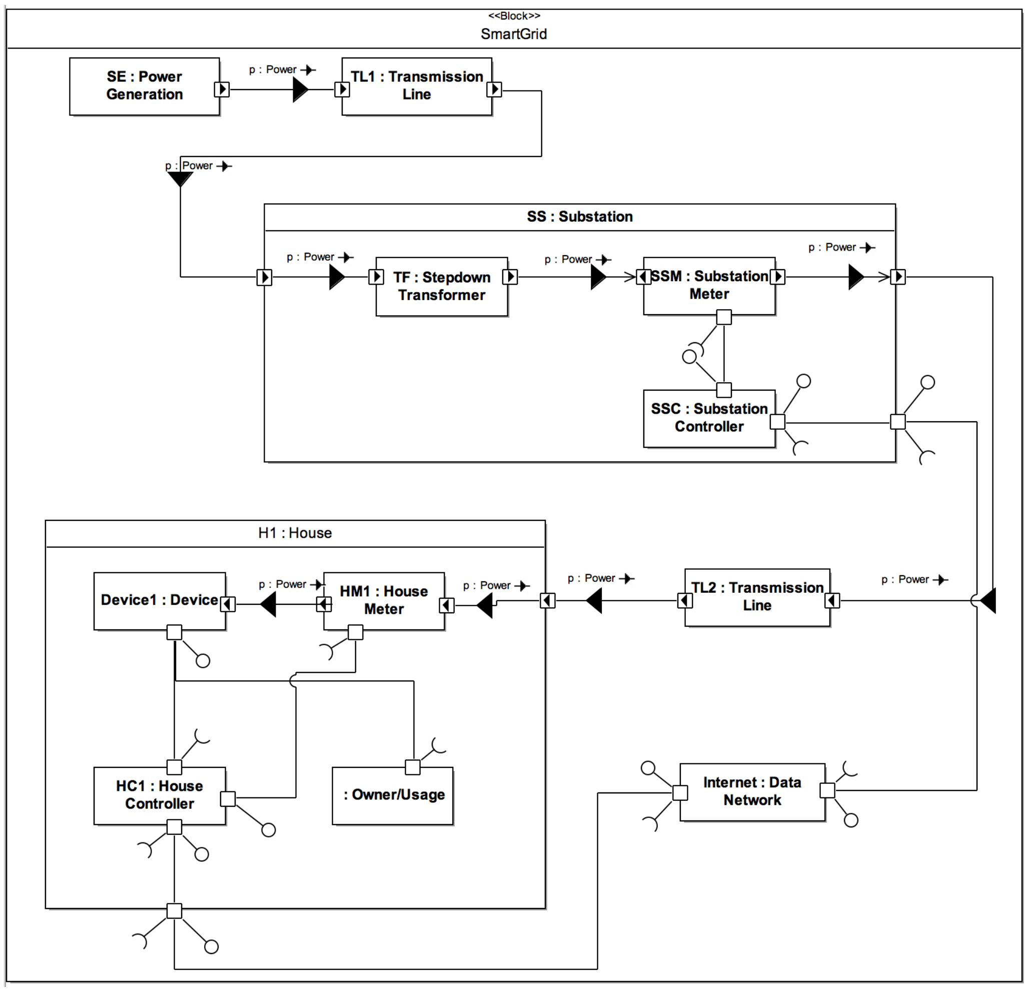


Figure 6.12: Internal Block Diagram of Smart Grid

platform identified.

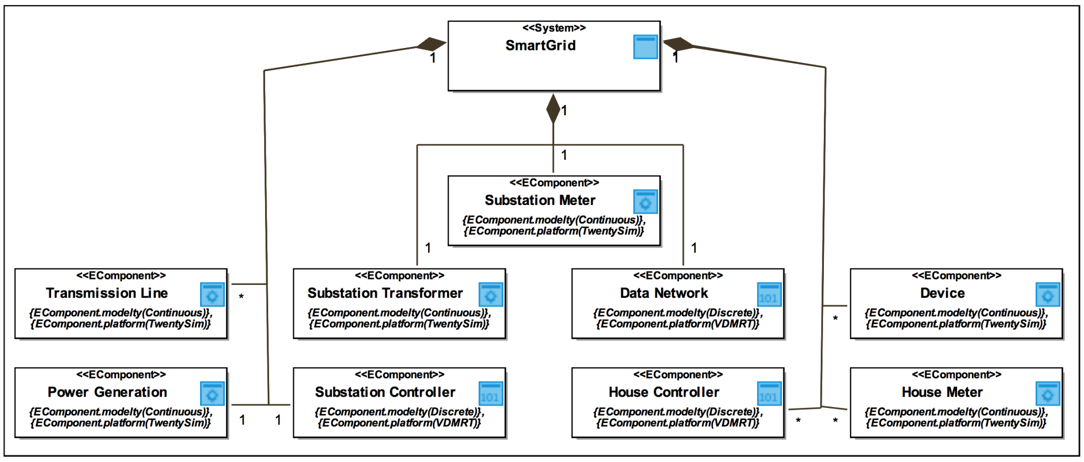

<<System>>
SmartGrid
<<EComponent>>
Substation Meter
{EComponent.modelty(Continuous)}, {EComponent.platform(TwentySim)}
<<EComponent>>
Transmission Line
{EComponent.modelty(Continuous)}, {EComponent.platform(TwentySim)}
<<EComponent>>
Substation Transformer
{EComponent.modelty(Continuous)}, {EComponent.platform(TwentySim)}
<<EComponent>>
Data Network
{EComponent.modelty(Discrete)}, {EComponent.platform(VDMRT)}
<<EComponent>>
Device
{EComponent.modelty(Continuous)}, {EComponent.platform(TwentySim)}
<<EComponent>>
Power Generation
{EComponent.modelty(Continuous)}, {EComponent.platform(TwentySim)}
<<EComponent>>
Substation Controller
{EComponent.modelty(Discrete)}, {EComponent.platform(VDMRT)}
<<EComponent>>
House Controller
{EComponent.modelty(Discrete)}, {EComponent.platform(VDMRT)}
<<EComponent>>
House Meter
{EComponent.modelty(Continuous)}, {EComponent.platform(TwentySim)}

Figure 6.13: Architecture Structure Diagram for multi-model of Smart Grid

The connections between the components are defined in the Connections Diagram (CD) in Figure 6.14. The interface between subsystems is defined as the interaction points between cyber and physical components (FMUs).

Discussion

Contrasting the architectures shown in the initial model (Figures 6.11 and 6.12) to that in the multi-model (Figures 6.13 and 6.14), whilst the same base components are present in both, some of the intuitive domain-specific structures are lost when moving to a multi-model. For example, it is now not clear where the *substation* or *house* elements are in the multi-model.

An important issue here is in the reason behind producing different architectural models. Using SysML diagrams in a *holistic* approach, a CPS engineer describes the model using a structure natural to the application domain. As such, the *reason* for modelling is not in the ultimate analysis to perform, but to define and understand the structure and behaviour of a system. In contrast, the *design* approach is necessary to configure INTO-CPS multi-models from SysML.

Figure 6.15 presents an overview of the relationships between the different types of models. The figure shows that the 'real' system may be modelled in different forms: the holistic and design architectures and the multi-model.

As illustrated in the figure, one approach can inform another. In some cases this may be a natural process; for example in the Smart Grid example, isolating each of the lowest level components in Figure 6.11 to be individual FMUs in a multi-model is an evolution which will likely result in a feasible model. By creating a domain-specific holistic architecture first, then transforming these models into a design architecture for multi-modelling, design teams will likely gain the most benefit.

6.4 Representing Non-Design Elements in SysML

Using the INTO-CPS tool chain, we generate co-simulation configurations using an architectural model defined with the INTO-SysML profile. This model defines the structure of a system

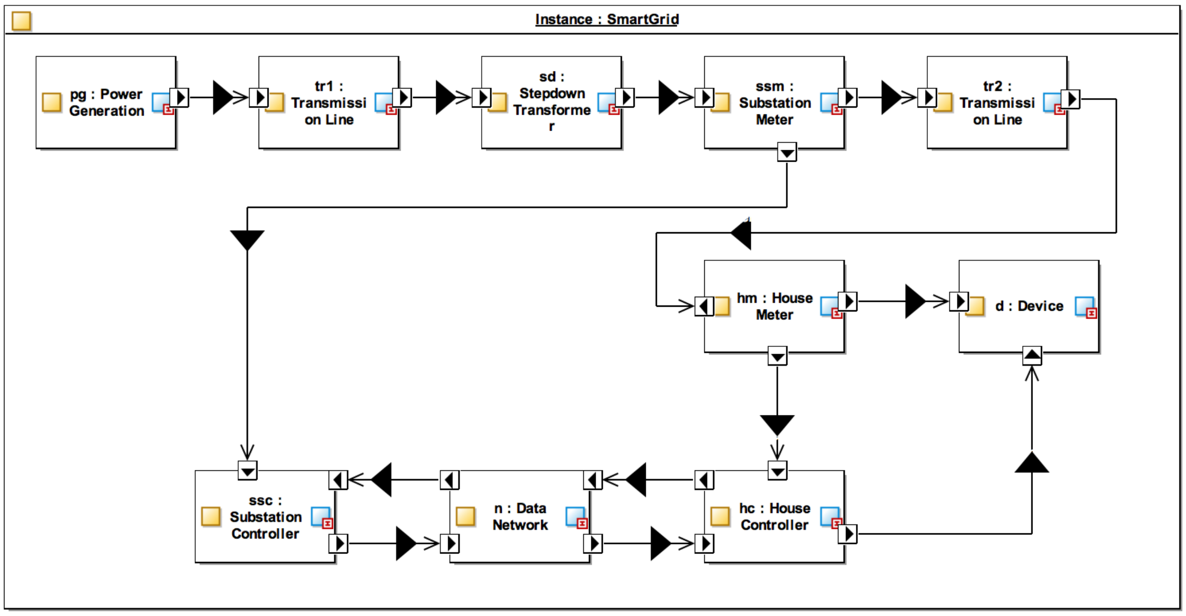


Figure 6.14: Connections Diagram for multi-model of Smart Grid

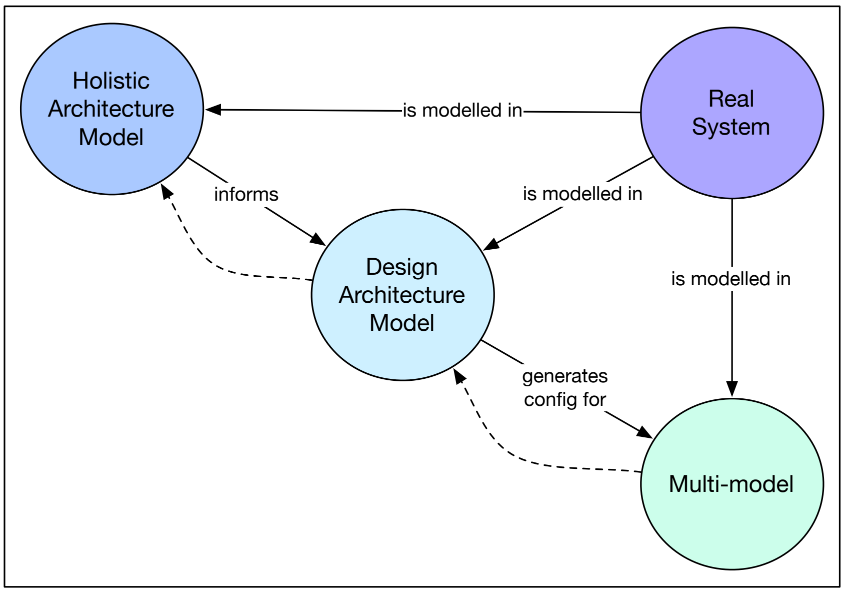


Figure 6.15: Relating holistic and design architectures

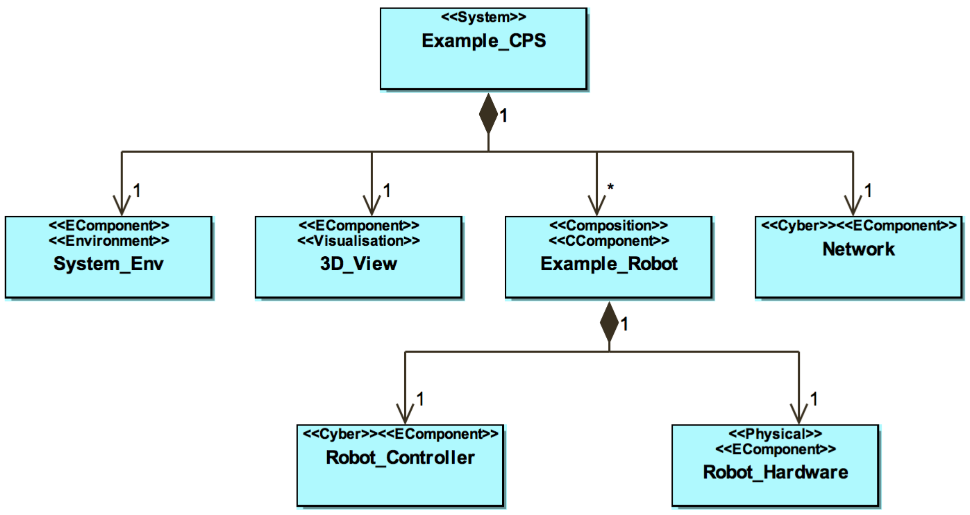


Figure 6.16: Example Architecture Structure Diagram of robot system

in terms of the composition of its components and their connections. There are however circumstances where elements in the multi-model are not part of the design of the final system, for example where an FMU is used purely for visualisation. This FMU must be connected to the system components, however is not itself a system component. This is also true when considering the environment of the system.

Here we present a small example of the use of these extensions, using a simple robot example (based on the line-following robot pilot study, see Deliverable D3.6 [2]) to illustrate the use of «`CComponent`»s and the *kind* of components (*cyber*, *physical*, *environment*, *visualisation*) described in Section 6.1.1 above.

The architecture structure diagram in Figure 6.16 shows: a *System_Env* block, an «`EComponent`» defined as an `Environment` FMU; a *3D_View* block an «`EComponent`», defined as an `Visualisation` FMU; and an *Example_Robot* block, an «`EComponent`» defined as an `composition` of two FMUs.

The example has two connection diagrams. The first is shown in Figure 6.17, it contains only those connections with respect to the system and its constituent components . This diagram shows a block instance *cps1* containing the environment (*e*) and the example robot (*r*) which contains two the controller and hardware components.

The second is shown in Figure 6.18, it depicts the use of the block instance *3D* of type *3D_View*. In this diagram, we show additional ports of the original block instances to output internal model details and connect these to the *3D* instance. The diagram includes the system connectors as shown in Figure 6.17.

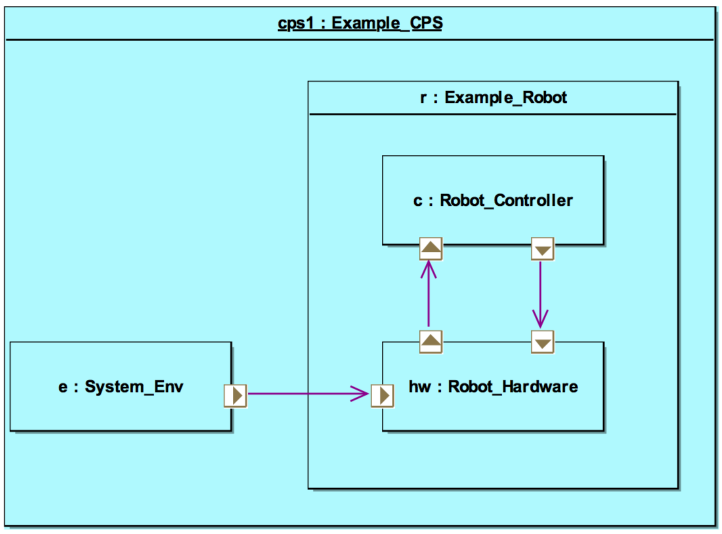


Figure 6.17: Connections Diagram for robot showing only system and environment connectors

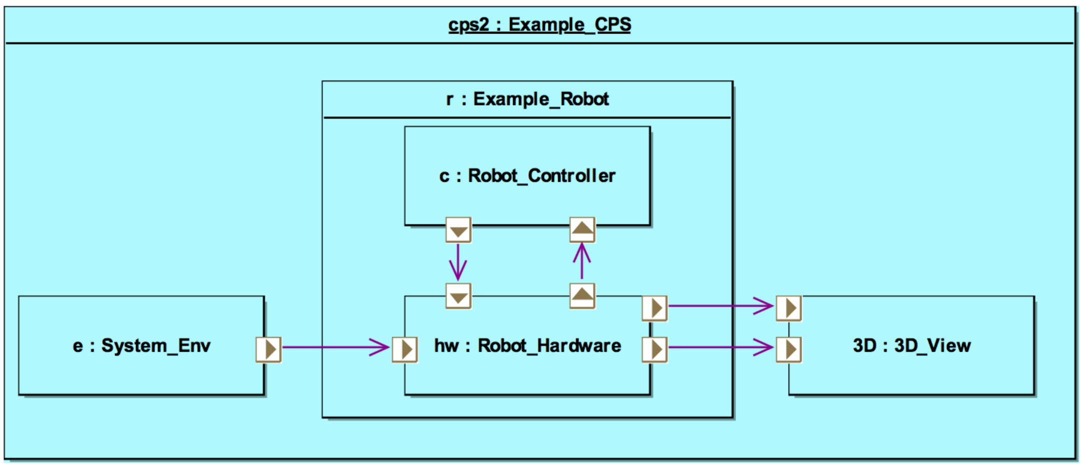


Figure 6.18: Connections Diagram for the robot system showing the system *and* visualisation components

Chapter 7

Initial Multi-Modelling using VDM

In this section we provide guidance on producing initial multi-models from architectural descriptions produced using the INTO-CPS SysML profile. We focus on using discrete-event (DE) models to produce initial, abstract FMUs that allow integration testing through co-simulation before detailed modelling work is complete. This is called a "DE-first" approach [4, 62]. We describe the use of VDM and the Overture tool, with FMI export plug-in installed, for this approach. The principles outlined in this section can be applied in other modelling tools. This approach can work with or without the SysML profile.

7.1 The DE-first Approach

After carrying out requirements engineering (RE), as described in Chapter 5, and design architectural modelling in SysML, as described in Chapter 6, the engineering team should have the following artifacts available:

- One or more Architecture Structure Diagrams (ASDs) defining the composition of «`EComponent`»s (to be realised as «`Cyber`» or «`Physical`» FMUs) that will form the multi-model.
- Model descriptions exported for each «`EComponent`».
- One or more Connections Diagrams (CDs) that will be used to configure a multi-model.

The next step is to generate a multi-model configuration in the INTO-CPS Application and populate it with FMUs, then run a first co-simulation. This however requires the source models for each FMU to be ready. If they already exist this is easy, however they may not exist if this is a new design. In order to generate these models, the model descriptions for each «`EComponent`» can be passed to relevant engineering teams to build the models, then FMUs can be passed back to be integrated.

It can be useful however to create and test simple, abstract FMUs first (or in parallel), then replace these with higher-fidelity FMUs as the models become available. This allows the composition of the multi-model to be checked early, and these simple FMUs can be reused for regression testing. This approach also mitigates the problem of modelling teams working at different rates.

Where these simple FMUs are built within the DE formalism (such as VDM), this is called a *DE-first* approach. This approach is particularly appropriate where complex DE control be-

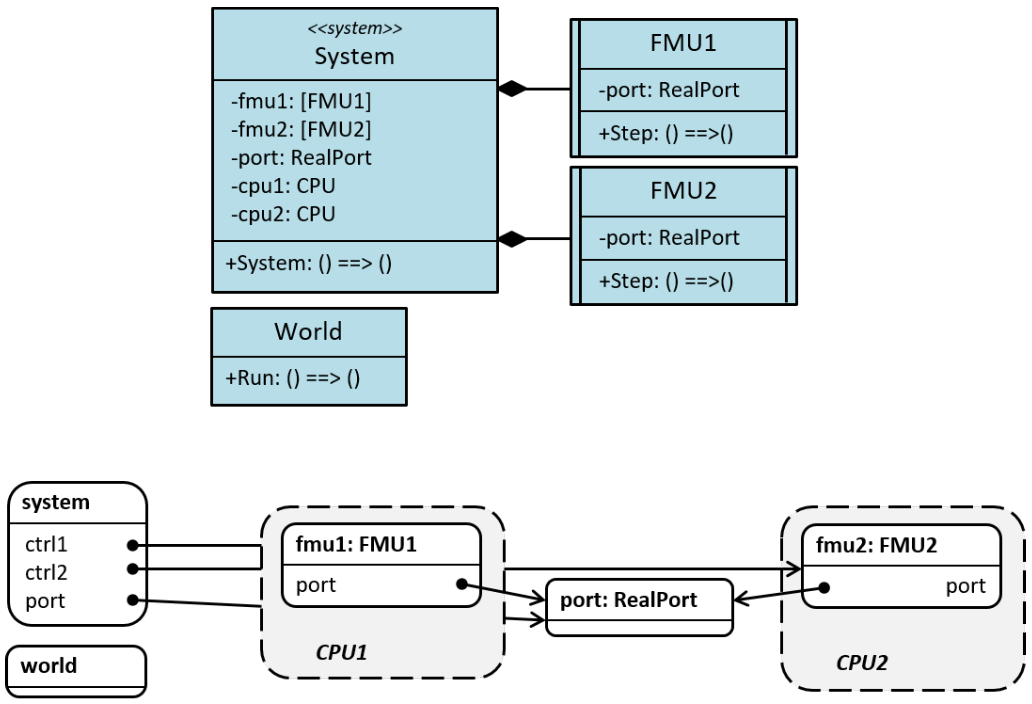


Figure 7.1: Class diagram showing two simplified FMU classes created within a single VDM-RT project, and an object diagram showing them being instantiated as a test.

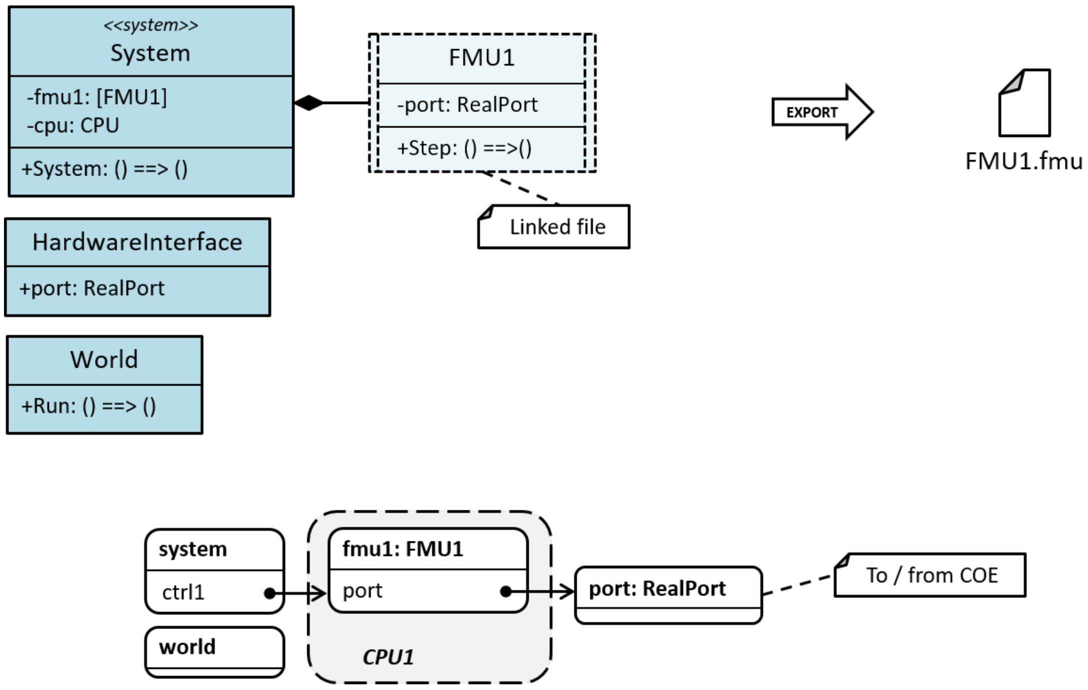


Figure 7.2: Class and object diagrams showing a linked class within its own project for FMU creation.

haviours —such as supervisory control or modal behaviours— are identified as a priority or where the experience of the modelling team is primarily in the DE domain [14].

Guidance on how to produce DE approximations for use in multi-modelling, and in particular approximations of CT behaviour, can be found in material describing the Crescendo baseline technology [62], which is also available via the Crescendo website[1].

7.2 DE-first within INTO-CPS

Given an architectural structure diagram, connections diagram and model descriptions for each «`EComponent`», the suggested approach is to begin by building a single VDM-RT project in Overture with the following elements:

- A class for each «`EComponent`» representing an FMU. Each class should define port-type instance variables (i.e. of type `IntPort`, `RealPort`, `BoolPort`, or `StringPort`) corresponding to the model description and a constructor to take these ports as parameters. Each FMU class should also define a thread that calls a `Step` operation, which should implement some basic, abstract behaviour for the FMU.
- A `system` class that instantiates port and FMU objects based on the connections diagram. Ports should be passed to constructor of each FMU object. Each FMU object should be deployed on its own CPU.
- A `World` class that starts the thread of each FMU objects.

Class and object diagrams giving an example of the above is shown in Figure 7.1. In this example, there are two «`EComponent`»s (called *FMU1* and *FMU2*) joined by a single connection of type real. Such a model can be simulated within Overture to test the behaviour of the FMUs. This approach can be combined with the guidance in Chapter 8 to analyse more complicated networked behaviour. Once the behaviour of the FMU classes has been tested, actual FMUs can be produced and integrated into a first multi-model by following the guidance below.

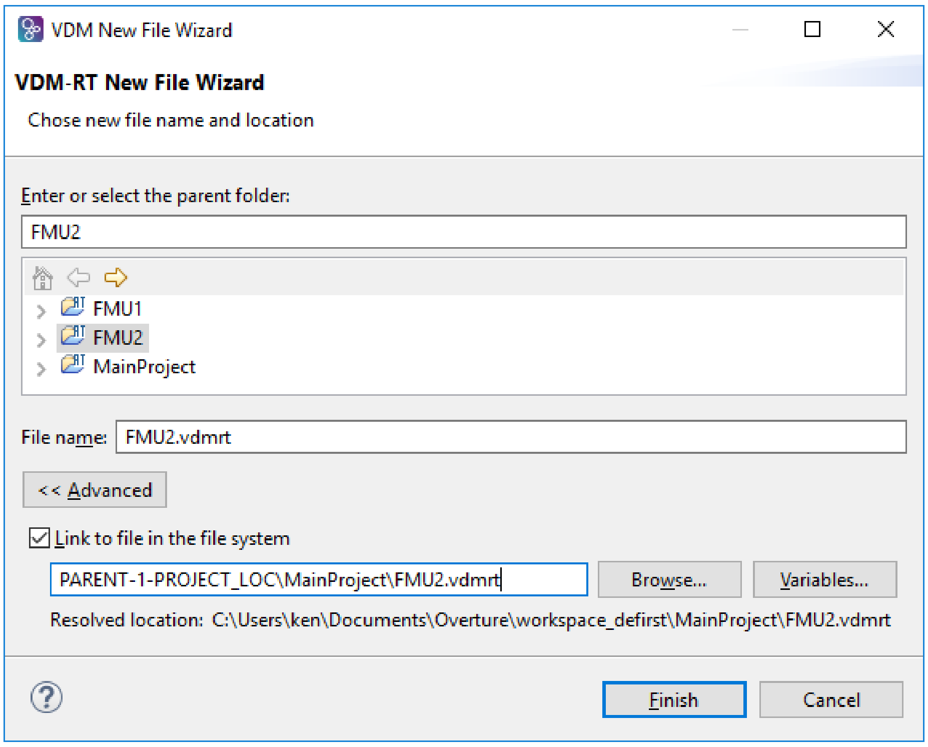
7.3 FMU Creation

The steps outlined below assume a knowledge of FMU export in Overture, which can be found in the User Manual, Deliverable D4.3a [1], in Section 5.1. To generate FMUs, a project must be created for each «`EComponent`» with:

- One of the FMU classes from the main project.
- A `HardwareInterface` class that defines the ports and annotations required by the Overture FMU export plug-in, reflecting those defined in the model description.
- A `system` class that instantiates the FMU class and passes the port objects from the `HardwareInterface` class to its constructor.
- A `World` class that starts the thread of the FMU class.

The above structure is shown in Figure 7.2. A skeleton project with a correctly annotated `HardwareInterface` class can be generated using the model description import feature of the Overture FMU plug-in. The FMU classes can be linked into the projects (rather than hard copies being made) from the main project, so that any changes made are reflected in both

[1]See `http://crescendotool.org/documentation/`


VDM New File Wizard

VDM-RT New File Wizard

Chose new file name and location

Enter or select the parent folder: FMU2

FMU1
FMU2
MainProject

File name: FMU2.vdmrt

<< Advanced

☑ Link to file in the file system

PARENT-1-PROJECT_LOC\MainProject\FMU2.vdmrt

Browse... Variables...

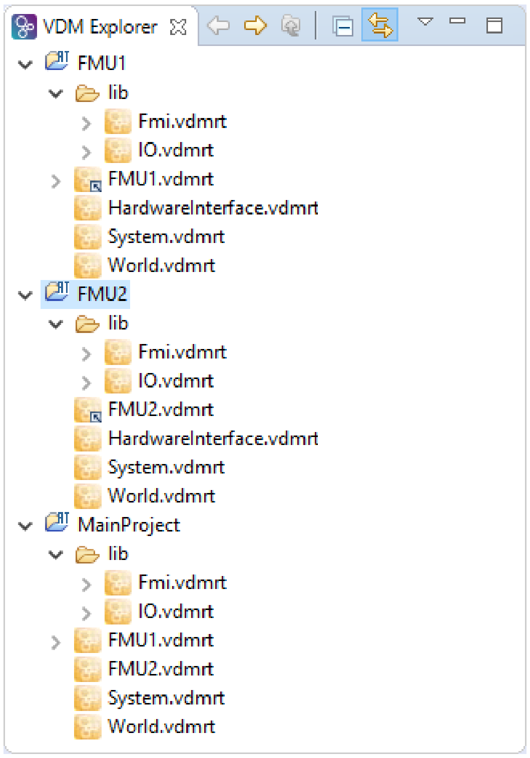
Resolved location: C:\Users\ken\Documents\Overture\workspace_defirst\MainProject\FMU2.vdmrt

Finish Cancel

Figure 7.3: Linking files in the *New* > *Empty VDM-RT File* dialogue.

the main project and the individual FMU projects. These links can be created by using the *Advanced* section of the *New* > *Empty VDM-RT File* dialogue, using the `PROJECT-1-PARENT_LOC` variable to refer to the workspace directory on the file system (as shown in Figure 7.3). Note that if the FMU classes need to share type definitions, these can be created in a class called `Types` in the main project, then this class can be linked into each of the FMU projects in the same way.

From these individual project, FMUs can be exported and co-simulated within the INTO-CPS tool. These FMUs can then be replaced as higher-fidelity versions become available, however they can be retained and used for regression and integration testing by using different multi-model configurations for each combination.



```
FMU1
  lib
    Fmi.vdmrt
    IO.vdmrt
  FMU1.vdmrt
  HardwareInterface.vdmrt
  System.vdmrt
  World.vdmrt
FMU2
  lib
    Fmi.vdmrt
    IO.vdmrt
  FMU2.vdmrt
  HardwareInterface.vdmrt
  System.vdmrt
  World.vdmrt
MainProject
  lib
    Fmi.vdmrt
    IO.vdmrt
  FMU1.vdmrt
  FMU2.vdmrt
  System.vdmrt
  World.vdmrt
```

Figure 7.4: Project structure of an Overture workspace showing a main project and two projects used for generating FMUs from linked class files.

Chapter 8

Modelling Networks with VDM in Multi-models

In this section, we address the problem of modelling networked controllers in multi-models, presenting a solution using VDM. When modelling and designing distributed controllers, it is necessary to model communications between controllers as well. While controller FMUs can be connected directly to each other through for co-simulation, this quickly becomes unwieldy due to the number of connections increasing exponentially. For example, consider the case of five controllers depicted in Figure 8.1. In order to connect each controller together, 20 connections are needed (i.e. for a complete bidirected graph). Even with automatic generation of multi-model configurations, this is in general not a feasible solution.

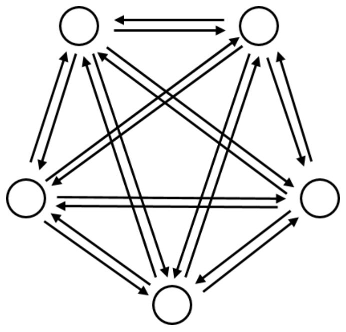


Figure 8.1: Topology of five controllers connected to each other

We suggest employing a pattern described initially as part of the Crescendo technology [14], in which a representation of an abstract communications medium called the 'ether' is introduced. In the INTO-CPS setting, the ether is an FMU that is connected to each controller that handles message-passing between them. This reduces the number of connections needed, particularly for large numbers of controllers such as swarms. For five controllers, only 10 connections are needed, as shown in Figure 8.2.

In the remainder of this section, we describe how to pass messages between VDM FMUs using string types, how the ether class works, some of the consequences of using the ether pattern, and finally some extensions for providing quality of service (QoS) guarantees. An example multi-model, called *Case Study: Ether*, is available from the INTO-CPS Application. It is also described in the Examples Compendium, Deliverable D3.6 [2].

8.1 Representing VDM Values as Strings

Connections between FMUs are typically numerical or Boolean types. This works well for modelling of discrete-time (DT) controllers and continuous-time (CT) physical models, however one of the advantages of VDM is the ability to represent more complex data types that better fit the abstractions of supervisory control. Therefore, in a multi-modelling setting, it is advantageous if VDM controllers can communicate with each other using data types that are not part of the FMI specification.

This can be achieved by passing strings between VDM FMUs (which are now supported by the Overture FMU export plug-in) and the `VDMUtil` standard library included in Overture, which can convert VDM types to their string representations and back again.

The `VDMUtil` library provides a (polymorphic) function called `val2seq_of_char`, that converts a VDM type to a string. It is necessary to tell the function what type to expect as a parameter in square brackets. For example, in the following listing, a 2-tuple is passed to the function, which will produce the output "`mk_(2.4, 1.5)`":

```
VDMUtil`val2seq_of_char[real*real](mk_(2.4, 1.5))
```

The above can be used when sending messages as strings. In the model receiving message, the inverse function `seq_of_char2val` can be used. This function returns two values, a Boolean value indicating if the conversion was successful, and the value that was received:

```
let mk_(b,v) = VDMUtil`val2seq_of_char[real*real](msg) in
  if b then ...
```

In the first few steps of co-simulation, empty or invalid strings are often passed as values, so it is necessary to check if the conversion was successful (as in the above listing) before using the value.

Note that currently (as of Overture 2.4.0), the `VDMUtil` library is called in the default scope, meaning that it does not know about custom types defined in the model. Therefore, it is recommended to pack values in a tuple (as in the above example) for message passing, then convert to and from any custom types in the sending and receiving models.

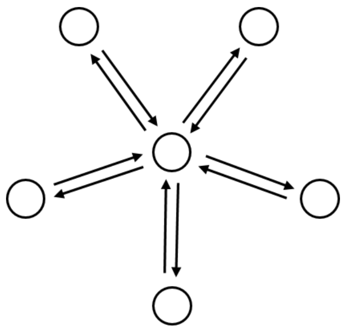


Figure 8.2: Topology of five controllers connected via a shared medium

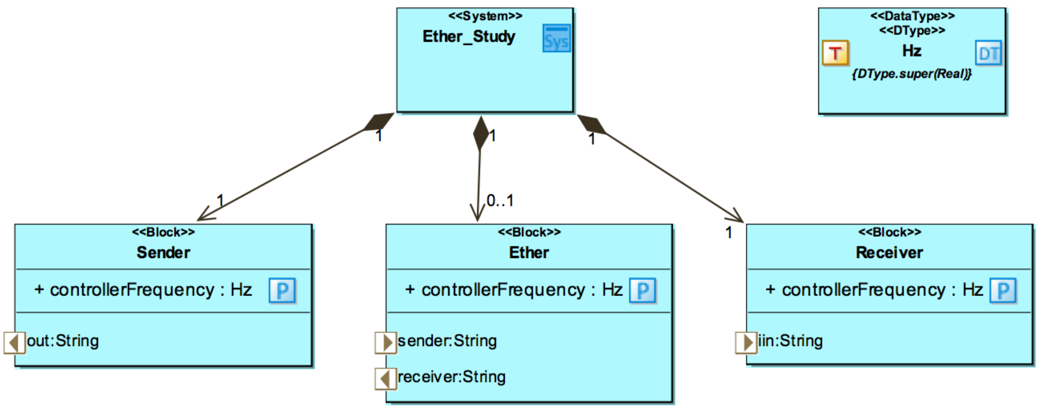


Figure 8.3: *Case Study: Ether* example

8.2 Using the Ether FMU

By encoding VDM values as strings, it is possible to define a simple broadcast ether that receives message strings on its input channel(s) and sends them to its output channel(s). As a concrete example, we consider the *Case Study: Ether* (see Deliverable D3.5 [63]), which contains a `Sender`, a `Receiver` and an `Ether`, as depicted in Figure 8.3. In this example, the three FMUs have the following roles:

Sender Generates random 3-tuple messages of type `real * real * real`, encodes them as strings using the `VDMUtil` library and puts them on its output port.

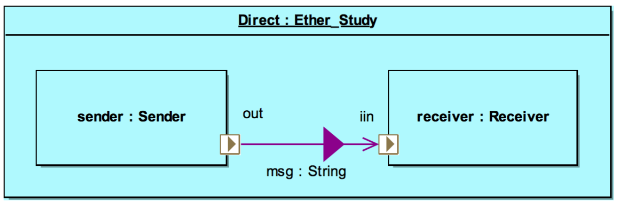
Receiver Receives strings on its input port and tries to convert them to single messages of type `real * real * real` or to a sequence of messages of type of type `seq of (real * real * real).`

Ether Has an input port and output port, each assigned a unique identifier, i.e. as a `map Id to StringPort`. It also has a mapping of input to output ports as a set of pairs: `set of (Id * Id).` It has a list that holds messages for each output destination, because multiple messages might arrive for one destination. It gathers messages from each input and passes them to the outputs defined in the above mapping.

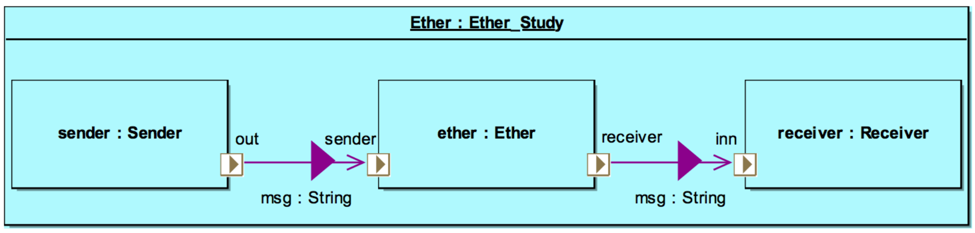
In this simple example, the sender and receiver are asymmetrical, but in more complicated examples controllers can be both senders and receivers by implementing both of the behaviours described above.

The *Case Study: Ether* example contains two multi-models that allow the sender and receiver to be connected directly (connection diagram shown in Figure 8.4(a)), or to be connected via the ether (connection diagram shown in Figure 8.4(b)). The description in the Examples Compendium, Deliverable D3.5 [63], explains how to run the two different multi-models. This approach shows that the use of string ports and the `VDMUtil` library can be useful even without the ether for message passing between controllers in simple topologies.

For the sender, this connection is transparent, it does not care whether it is connected to the ether or not. For the receiver, in the direct connection it will receive single messages, whereas when receiving from the ether it will receive a list of messages (even for a single value). So the receiver is able to deduce when it is directly connected or connected via the ether.



(a) Connection diagram of the *Direct* multi-model in the *Case Study: Ether* example



(b) Connection diagram of the *Ether* multi-model in the *Case Study: Ether* example

Figure 8.4: Alternative multi-models in the *Case Study: Ether* example

The ether defined in this example is intended to be generic enough that it can be used in other case studies that need a simple broadcast ether without guarantees of delivery. To use it, you can:

1. Import the *Ether* model from the *case-study_ether/Models* directory into Overture;
2. Update the `HardwareInterface`[1] class to provide input and/or output ports for all controllers that will be connected to the ether.
3. Update the `System` class to assign identifiers to all input and output ports; and
4. Update the set of identifier pairs that define connections.

8.3 Consequences of Using the Ether

The ether as presented above is fairly basic. In each update cycle, it passes values from its input variables to their respective output variables. This essentially replicates the shared variable style of direct FMU-FMU connections, which means that the relative update speeds of the FMUs may lead to the following:

Values may be delayed The introduction of an intermediate FMU means that an extra update cycle is required to pass values from sender to ether and ether to receiver. This may delay messages unless the ether updates at least twice as fast as the receiver.

Values may not be read If a value is not read by the receiver before it is changed, then that value is lost.

Values may be read more than once If a value is not changed by the sender before the receiver updates, then the value is read twice. In the simple ether, the receiver cannot distinguish an old message from a new message with the same values.

In the Examples Compendium, Deliverable D3.5 [63], the *Case Study: Ether* example is described along with some suggested experiments to see the effects of the above examples by changing the controller frequency parameters of the sender, ether and receiver. In the final part

[1] A class that provides annotated definitions of the ports for a VDM FMU.

of this section we outline ways to overcome such problems if it is necessary to guarantee that messages arrive and are read during a co-simulation.

8.4 Modelling True Message Passing and Quality of Service

The key to achieving a true message-passing is to overcome the problem of distinguishing old messages from new messages with the same values. This can be done by attaching a unique identifier to each message, which could be, for example, an identifier of the sender plus a message number:

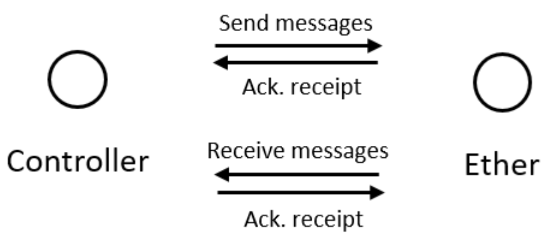


Figure 8.5: Topology of controller to *Ether* connection with dedicated channels for messages and acknowledgements

```
instance variables

id: seq of char := "a";
seqn: nat1 := 1;

...

VDMUtil`val2seq_of_char[seq of char*real*real](
  mk_(id ^ [seqn], 2.4, 1.5));
seqn := seqn + 1
```

The advantage of assigning an identifier to each controller is that messages could also contain destination addresses, instead of the broadcast model presented above. In order to achieve these, some changes are needed to allow for acknowledging receipt of messages. Controllers should:

1. Send a queue of messages on their output channel along with message identifiers of (recently received) messages;
2. Expect to receive a queue of messages along with message identifiers of successfully sent messages; and
3. Senders should remove messages from their output queue once their receipt has been acknowledged.

The `Ether` class must be extended to:

1. Inspect the message identifier (and destination if required) using `VDMUtil`;
2. Pass message identifiers back to senders to acknowledge receipts; and
3. Listen for message identifiers from receivers to know when to remove messages from the queue.

A dedicated channel for acknowledging messages could also be introduced, which would simplify the above. Therefore, each controller would have four connections to the ether: send and acknowledge, receive and acknowledge, as depicted in Figure 8.5.

The advantages of guaranteed message delivery as described here are that realistic and faulty behaviour of the communication medium can be studied. An ether can be produced that provides poorer quality of service (delay, loss, repetition, reordering). These behaviours could be parameterised and explored using DSE (see Chapter 9). By controlling for problems introduced by the nature of co-simulation, any reduction in performance of the multi-model can be attributed to the realistic behaviour introduced intentionally into the model of communications.

Chapter 9

Design Space Exploration

In this section, we outline guidelines for DSE over co-models of CPSs that: (a) support decision management by helping engineers to articulate clearly the parameters, objectives and metrics of a DSE analysis (Section 9.1); and (b) enable the tuning of DSE methods for given domains and systems of interest (Section 9.2).

9.1 Guidelines for Designing DSE in SysML

9.1.1 Rationale

Designing DSE experiments can be complex and tied closely to the multi-model being analysed. The definitions guiding the DSE scripts should not just appear with no meaningful links to the any other artefacts in the INTO-CPS Tool chain. There are two main reasons for this, firstly there is no traceability back to the requirements from which we might understand why the various objectives (measures) were being evaluated or why they were included in the ranking definition. Secondly, if DSE configurations are created manually for each new DSE experiment it is easy to imagine that the DSE analysis and ranking might not be consistent among the experiments.

Engineers need, therefore, to be able to model at an early stage of design how the experiments relate to the model architecture, and where possible trace from requirements to the analysis experiments. Here we describe the first step towards this vision: a SysML profile for modelling DSE experiments. The profile comprises five diagrams for defining *parameters*, *objectives* and *rankings*.

We take the same approach to defining the SysML profile for DSE as that used to define the INTO-SysML profile. A metamodel is defined (see Deliverable D3.2b [64]) and the collection of profile diagrams that implement this metamodel are defined in Deliverable D4.2c [60].

In this section, we present an illustrative example of the use of the DSE-SysML profile – from requirements engineering through defining parameters and objectives in the DSE-SysML profile to the final DSE JSON configuration files. We present result of the execution of DSE for the defined configuration.

As an example, we use the line follower robot pilot study. More details can be found in Deliverable D3.5 [63].

9.1.2 Requirements

We propose the use of a subset of the SoS-ACRE method detailed in Chapter 5 (as this section concentrates on the application of the DSE-SysML profile, we don't consider the full SoS-ACRE process). In the Requirements Definition View in Figure 9.1, the following five requirements are defined:

1. The robot shall have a minimal cross track error
2. The cross track error shall never exceed **X** mm
3. The robot shall maximise its average speed
4. The robot shall have a minimum average speed of **X** ms^{-1}
5. The robot sensor positions may be altered to achieve global goals

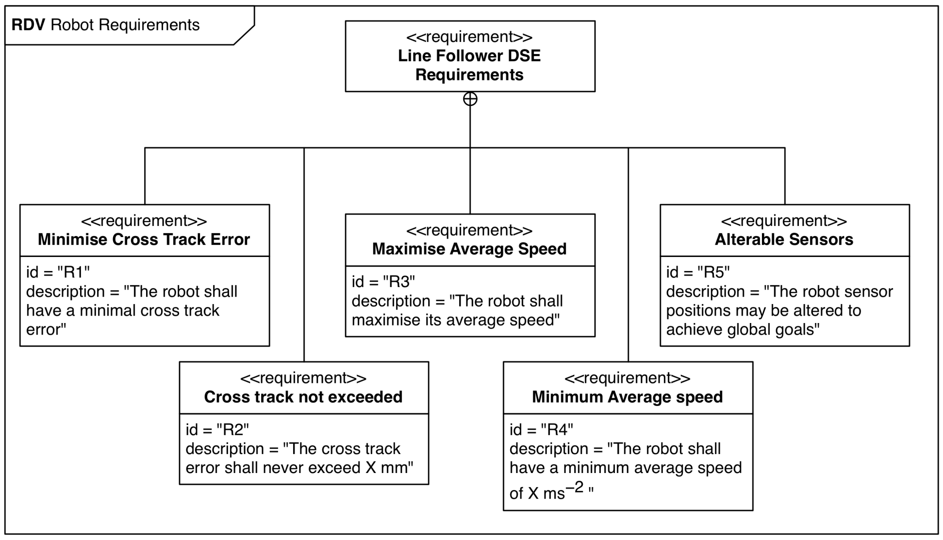

RDV Robot Requirements
<<requirement>>
Line Follower DSE Requirements
<<requirement>>
Minimise Cross Track Error
id = "R1"
description = "The robot shall have a minimal cross track error"
<<requirement>>
Maximise Average Speed
id = "R3"
description = "The robot shall maximise its average speed"
<<requirement>>
Alterable Sensors
id = "R5"
description = "The robot sensor positions may be altered to achieve global goals"
<<requirement>>
Cross track not exceeded
id = "R2"
description = "The cross track error shall never exceed X mm"
<<requirement>>
Minimum Average speed
id = "R4"
description = "The robot shall have a minimum average speed of X ms^{-2}"

Figure 9.1: Subset of the Requirements Definition View for requirements of the Line Following Robot

9.1.3 Objectives from Requirements

Based upon the requirements above, we define two objectives: the calculation of deviation from a desired path, and the speed of the robot.

Deviation The deviation from a desired path, referred to as the cross track error, is the distance the robot moves from the line of the map, as shown in Figure 9.2.

To compute cross track error we need some model of the desired path to be followed and the actual path taken by the robot. Each point on the actual path is compared with the model of the desired path to find its distance from the closest point, this becomes the cross track error. If the desired path is modelled as a series of points, then it may be necessary to find shortest distance to the line between the two closest points.

Speed The speed may be measured in several ways depending on what data is logged by the COE and what we really mean by speed, indicated in Figure 9.3.

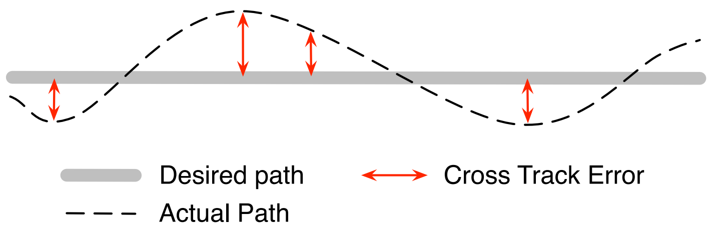


Figure 9.2: Cross track error at various points for a robot trying to follow a desired line

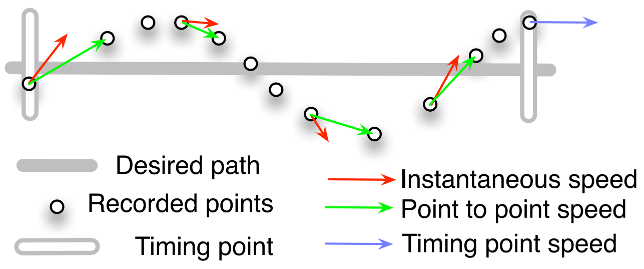


Figure 9.3: Cross track error at various points for a robot trying to follow a desired line

Inside the CT model there is a bond graph flow variable that represents the forwards motion of the robot. This variable is not currently logged by the COE but it could be and this would result in snapshots of the robot speed being taken when simulation models synchronise. In this example, we take the view that speed is referring to the time taken to complete a lap.

9.1.4 SysML Representation of Parameters, Objectives and Ranking

We next consider the use of the upcoming DSE profile to define the DSE parameters, objectives and desired ranking function. In the following SysML diagrams, we explicitly refer to model elements as defined in the architectural model of the line follower study, presented in Deliverable D3.5 [63].

Parameters In the requirements defined above, we see that the position of the line follower sensors may be varied. In real requirements, we may elicit the possible variables allowed. Figure 9.4 is a DSE Parameter Definition Diagram and defines four parameters required: *S1_X*, *S1_Y*, *S2_X* and *S2_Y*, each a set of real numbers. The DSE experiment in this example is called *DSE_Example*.

Figure 9.5 identifies the architectural model elements themselves (the `lf_position_x` and `lf_position_y` parameters of *sensor1* and *sensor2*) and the possible values each may have (for example the `lf_position_x` parameter of *sensor1* may be either 0.01 or 0.03). The diagram (or collection of diagrams if there is a large number of design parameters) should record all parameters for the experiment.

Objectives The objectives follow from the requirements as mentioned above. Figure 9.6 shows the DSE Objectives Definition Diagram with four objectives: *meanSpeed*, *lapTime*, *maxCrossTrackError* and *meanCrossTrackError*. Each have a collection of inputs – defined either as constants (e.g. parameter *p1* of *meanSpeed*), or to be obtained for the multi-model.

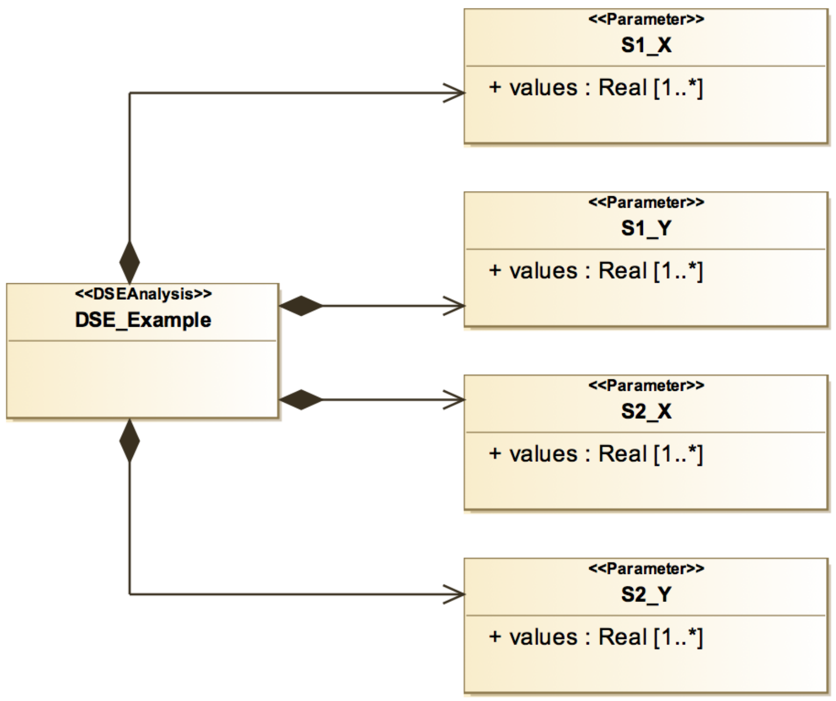


Figure 9.4: DSE-SysML Parameter Definition Diagram of Line Following Robot example

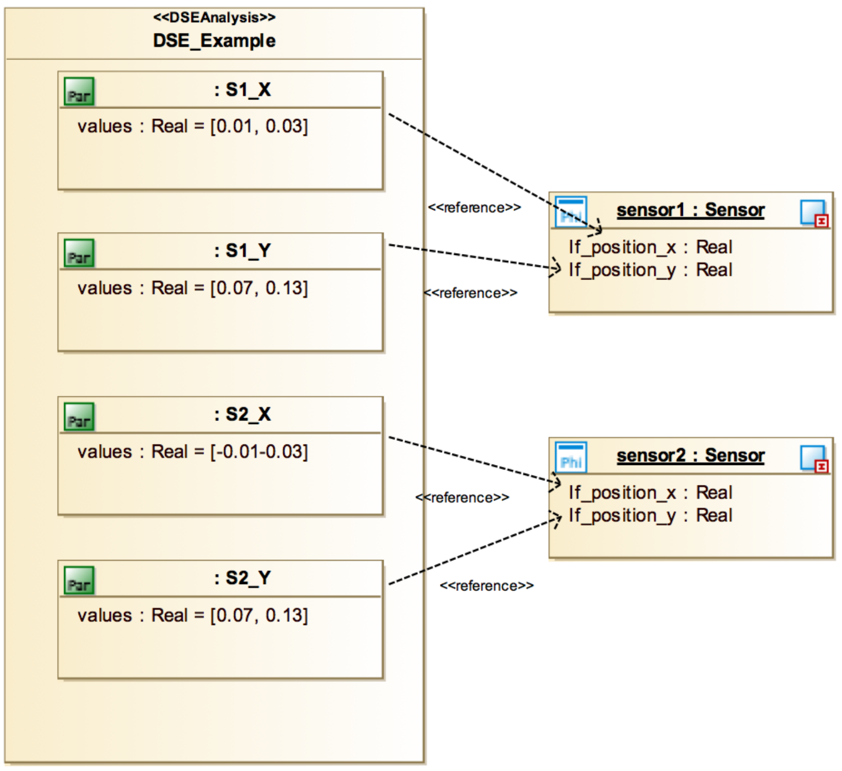


Figure 9.5: DSE-SysML Parameter Connection Diagram of Line Following Robot example

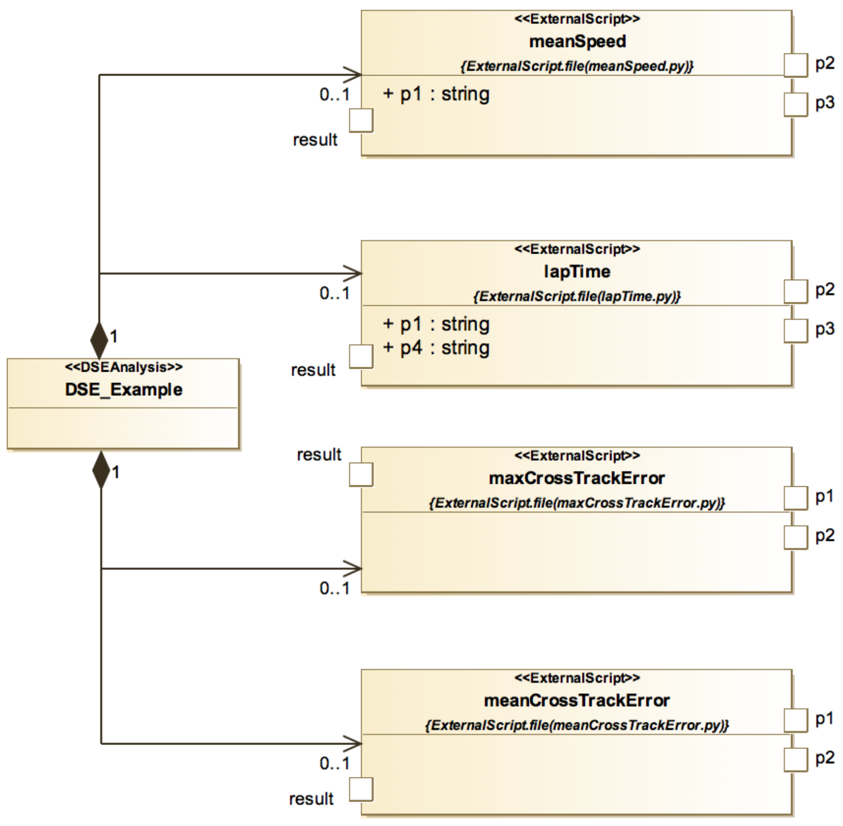


Figure 9.6: DSE-SysML Objective Definition Diagram of Line Following Robot example

The objective definitions are realised in Figure 9.7. The *meanSpeed* requires the step-size of the simulation (this is obtained from the co-simulation results, rather than defined here) and the `robot_x` and `robot_y` position of the robot body. The *lapTime* objective requires the time at each simulation step (again, obtained directly from the co-simulation output), the `robot_x` and `robot_y` position of the robot body and the name of the map. Both the *maxCrossTrack-Error* and *meanCrossTrackError* objectives require only the `robot_x` and `robot_y` position of the robot body.

Ranking Finally, the DSE Ranking Diagram in Figure 9.8 defines the ranking to be used in the experiment. This diagram states that the experiment uses the Pareto method, and is a 2-value Pareto referring to the *lapTime* and *meanCrossTrackError* objectives.

9.1.5 DSE script

These diagrams may then be translated to the JSON config format required by the DSE tool. The export of the configuration is performed in the Modelio tool and the subsequent movement of the resulting configuration file is performed in the INTO-CPS application (see the INTO-CPS User Manual, Deliverable D4.3a [1] for more details). Figure 9.9 shows the corresponding DSE configuration file for the line follower experiments. Note that where we refer to model elements of the architecture (such as model parameters), we now use the same conventions used in the co-simulation orchestration engine configuration.

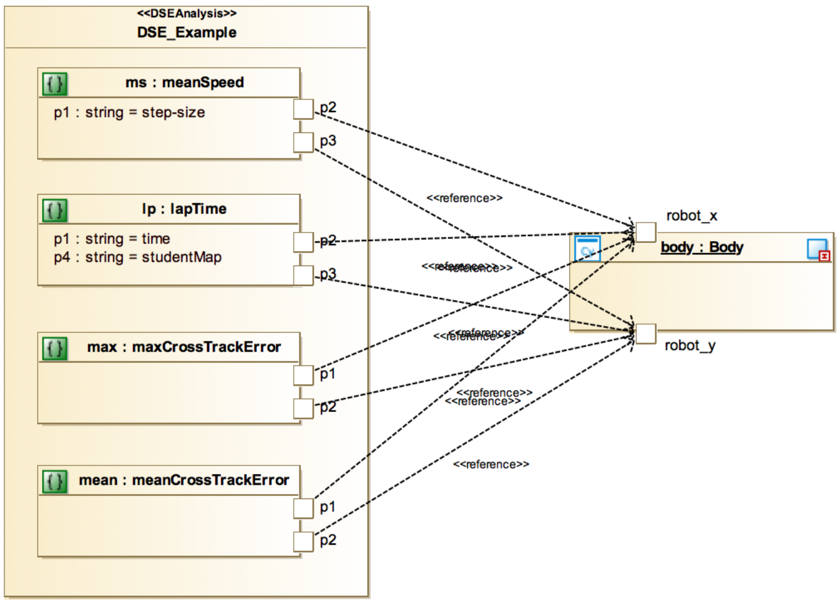


Figure 9.7: DSE-SysML Connection Objective Diagram of Line Following Robot example

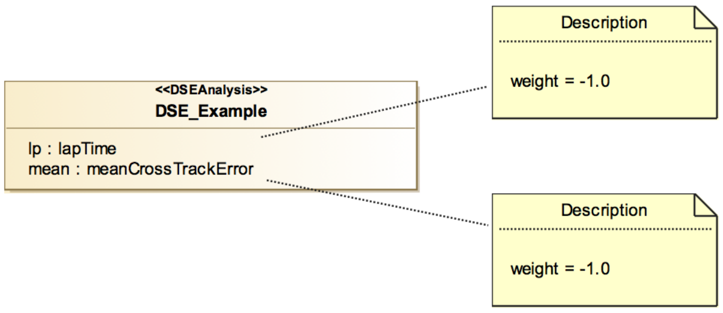
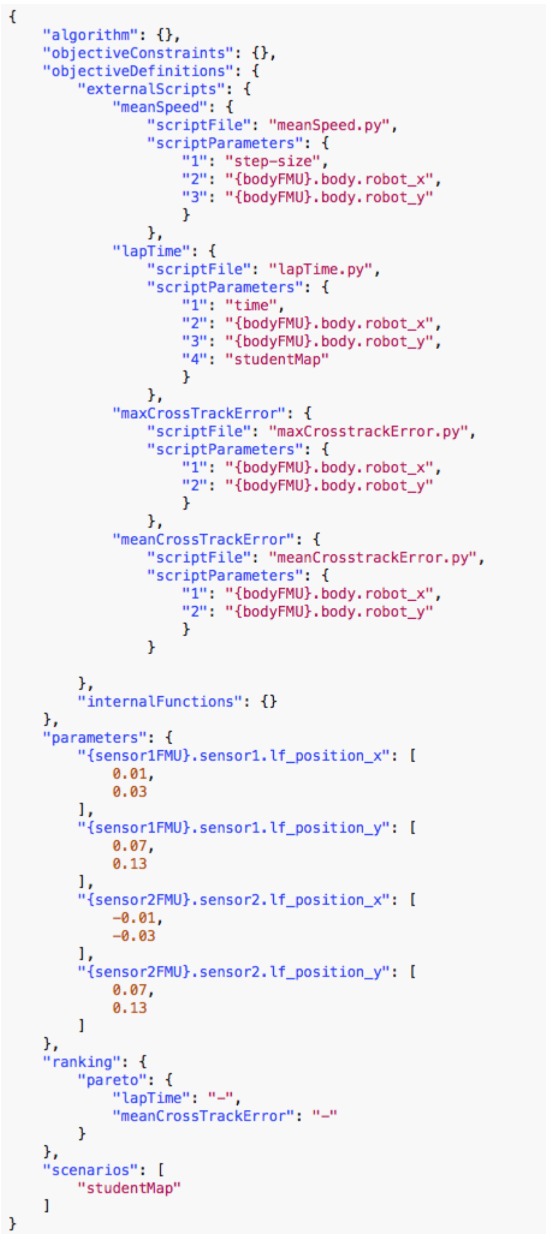


Figure 9.8: Example DSE-SysML Ranking Diagram of Line Following Robot example



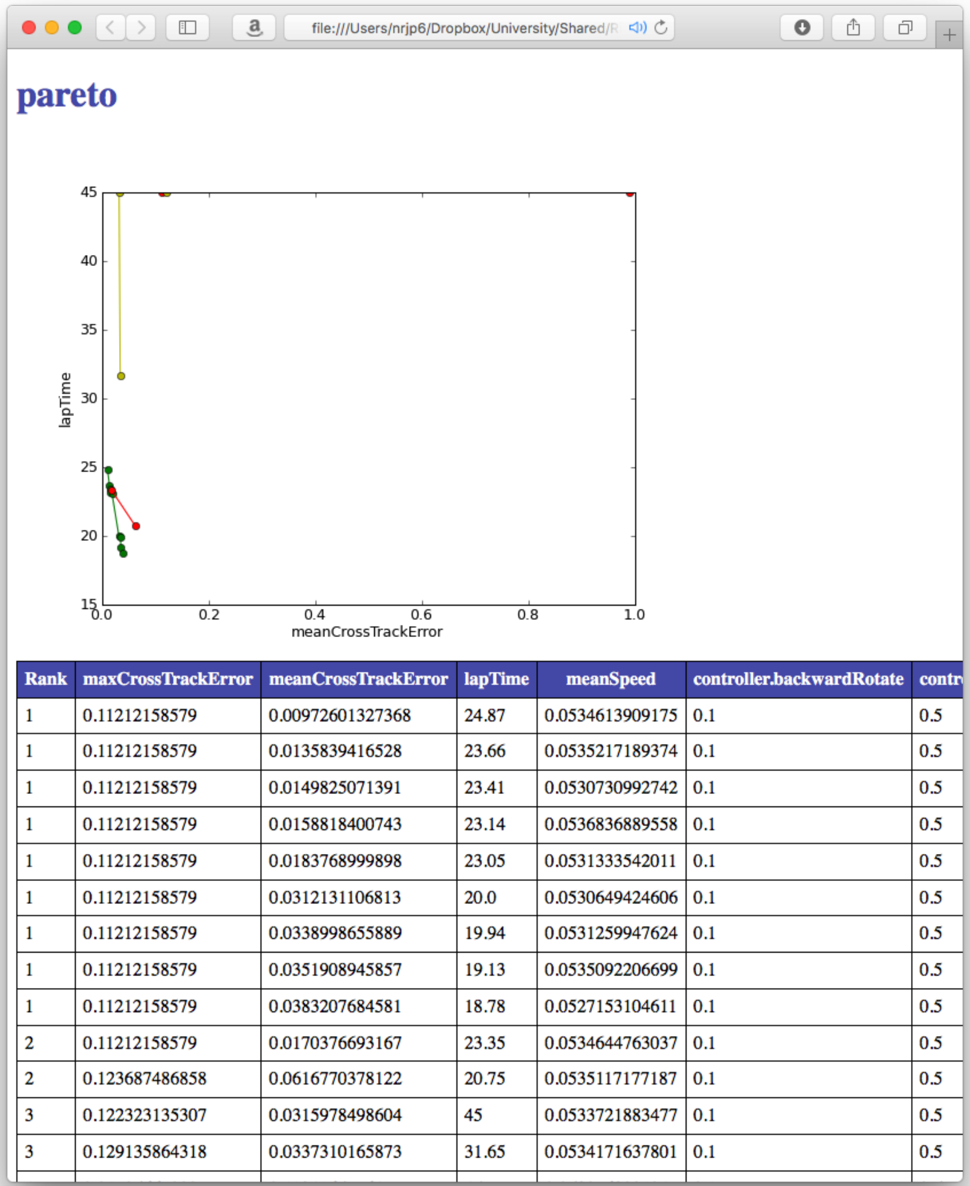
```
{
    "algorithm": {},
    "objectiveConstraints": {},
    "objectiveDefinitions": {
        "externalScripts": {
            "meanSpeed": {
                "scriptFile": "meanSpeed.py",
                "scriptParameters": {
                    "1": "step-size",
                    "2": "{bodyFMU}.body.robot_x",
                    "3": "{bodyFMU}.body.robot_y"
                }
            },
            "lapTime": {
                "scriptFile": "lapTime.py",
                "scriptParameters": {
                    "1": "time",
                    "2": "{bodyFMU}.body.robot_x",
                    "3": "{bodyFMU}.body.robot_y",
                    "4": "studentMap"
                }
            },
            "maxCrossTrackError": {
                "scriptFile": "maxCrosstrackError.py",
                "scriptParameters": {
                    "1": "{bodyFMU}.body.robot_x",
                    "2": "{bodyFMU}.body.robot_y"
                }
            },
            "meanCrossTrackError": {
                "scriptFile": "meanCrosstrackError.py",
                "scriptParameters": {
                    "1": "{bodyFMU}.body.robot_x",
                    "2": "{bodyFMU}.body.robot_y"
                }
            }

        },
        "internalFunctions": {}
    },
    "parameters": {
        "{sensor1FMU}.sensor1.lf_position_x": [
            0.01,
            0.03
        ],
        "{sensor1FMU}.sensor1.lf_position_y": [
            0.07,
            0.13
        ],
        "{sensor2FMU}.sensor2.lf_position_x": [
            -0.01,
            -0.03
        ],
        "{sensor2FMU}.sensor2.lf_position_y": [
            0.07,
            0.13
        ]
    },
    "ranking": {
        "pareto": {
            "lapTime": "-",
            "meanCrossTrackError": "-"
        }
    },
    "scenarios": [
        "studentMap"
    ]
}
```

Figure 9.9: A complete DSE configuration JSON file for the line follower robot example

9.1.6 DSE results

DSE is performed in the DSE tool (again, see the INTO-CPS User Manual, Deliverable D4.3a [1] for more detail) by processing the DSE configuration using scripts that contain the required algorithms. The main scripts contain the search algorithm that determines which parameters to use in each simulation, the simplest of these is the exhaustive algorithm that methodically runs through all combinations of parameters and runs a simulation of each. The log files produced by each simulation are then processed by other scripts to obtain the objective values defined in the previous section. Finally, the objective values are used by a ranking script to place all the simulation results into a partial order according to the defined ranking. The ranking information is used to produce tabular and graphical results that may be used to support decisions regarding design choices and directions.

Figure 9.10 shows an example of the DSE results from the line follower robot where the lap time and mean cross track error were the objectives to optimise. These results contain two representations of the data, a graph plotting the objective values for each design, with the Pareto front of optimal trade-offs between the key objectives highlighted, here in blue. The second part of the results presents the data is tables, indexed by the ranking position of each result. This permits the user to determine the precise values for both the measured objectives and also the design parameters used to obtain that result.



Rank	maxCrossTrackError	meanCrossTrackError	lapTime	meanSpeed	controller.backwardRotate	contr
1	0.11212158579	0.00972601327368	24.87	0.0534613909175	0.1	0.5
1	0.11212158579	0.0135839416528	23.66	0.0535217189374	0.1	0.5
1	0.11212158579	0.0149825071391	23.41	0.0530730992742	0.1	0.5
1	0.11212158579	0.0158818400743	23.14	0.0536836889558	0.1	0.5
1	0.11212158579	0.0183768999898	23.05	0.0531333542011	0.1	0.5
1	0.11212158579	0.0312131106813	20.0	0.0530649424606	0.1	0.5
1	0.11212158579	0.0338998655889	19.94	0.0531259947624	0.1	0.5
1	0.11212158579	0.0351908945857	19.13	0.0535092206699	0.1	0.5
1	0.11212158579	0.0383207684581	18.78	0.0527153104611	0.1	0.5
2	0.11212158579	0.0170376693167	23.35	0.0534644763037	0.1	0.5
2	0.123687486858	0.0616770378122	20.75	0.0535117177187	0.1	0.5
3	0.122323135307	0.0315978498604	45	0.0533721883477	0.1	0.5
3	0.129135864318	0.0337310165873	31.65	0.0534171637801	0.1	0.5

Figure 9.10: DSE results

9.2 An Approach to Effective DSE

Given a "designed" design space using the method detailed above, we use the INTO-CPS Tool Chain to simulate each design alternative. The initial approach we took was to implement an algorithm to exhaustively search the design space, and evaluate and rank each design. Whilst this approach is acceptable on small-scale studies, this quickly becomes infeasible as the design space grows. For example, varying n parameters with m alternative values produces a design space of m^n alternatives. In the remainder of this paper, we present an alternative approach to exploring the design space in order to provide guidance for CPS engineers on how to design the exploration of designs for different classes of problems.

9.2.1 A Genetic Algorithm for DSE

Inspired by processes found in nature, genetic algorithms "breed" new generations of optimal CPS designs from the previous generation's best candidates. This mimics the concept of survival of the fittest in Darwinian evolution. Figure 9.11 represents the structure of a genetic algorithm used for DSE. Several activities are reused from exhaustive DSE: simulation; evaluation of objectives; rank simulated designs; and generate results. The remaining activities are specific to the genetic approach and are detailed in this section.

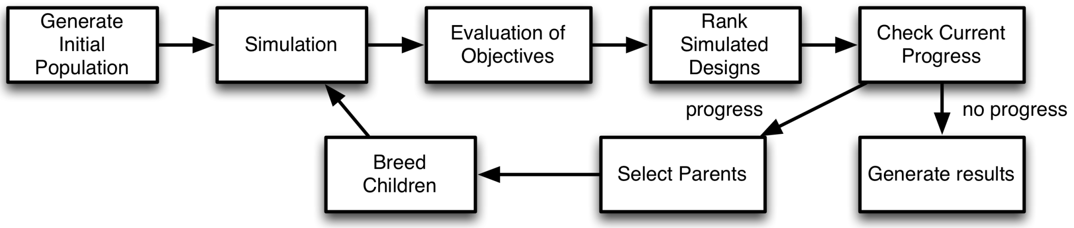


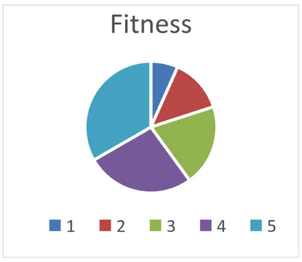
Figure 9.11: High-level process for DSE Genetic Algorithm

Generating initial population: Two methods for generating an initial population of designs are supported: randomly, or uniformly across the design space. Generating an initial design set which is distributed uniformly could allow parts of the design space to be explored that would otherwise not be explored with a random initial set. This could give us greater confidence that the optimal designs found by the genetic algorithm are consistent with the optimal designs of the total design space.

Selecting parents: Two options for parent selection are supported: random and distributed. Random selection means that two parents are chosen randomly from the non-dominated set (NDS). There is also a chance for parents to be selected which are not in the NDS, potentially allowing different parts of the design space to be explored due to a greater variety of children being produced.

An intelligent approach involves calculating the distribution of each design's objectives from other designs in the NDS. One of the parents chosen is the design with the greatest distribution, enabling us to explore another part of the design space which may contain other optimal designs. Picking a parent that has the least distribution suggests that this parent is close to other optimal designs, meaning that perhaps it is likelier to produce optimal designs.

Figure 9.12(a) shows the fitness roulette by which how much a design solution in Figure 9.12(b) satisfies the requirements. It can be seen that there exists a relationship where the greater the fitness value a design has, the more likely it is to be selected as a parent. The probability P of design d being selected as a parent can be calculated by:



(a) Example fitness roulette

Solution	Fitness
1	25
2	5
3	40
4	10
5	20

(b) Fitness of designs

Figure 9.12: Genetic Algorithm fitness selection

Breeding children: After the parents are selected, the algorithm creates two new children using a process of crossover. Figure 9.13 shows this process. Mutation could also occur, where a randomly chosen parameter's value is replaced by another value defined in the initial DSE configuration, producing new designs to explore other parts of the design space.

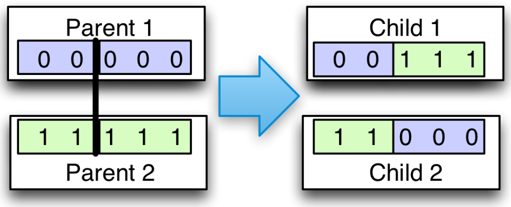


Figure 9.13: Depiction of genetic crossover

Checking current progress: Progression is determined by the change in the NDS on each iteration. It is possible to tune the number of iterations without progress before termination.

9.2.2 Measuring Effectiveness

To provide guidance on selection and tuning of a specific algorithm to a DSE situation it is necessary that there is a means for experimenting with the algorithm parameters and also means for evaluating the resulting performance. To this end an experiment was devised that supports exploration of these parameters using a range of design spaces as the subject. The experiment is based upon generating a ground truth for a set of design spaces such that the composition of each Pareto front is known and we may assess the cost and accuracy of the genetic algorithm's attempt to reach it. A limiting factor for these design spaces is that they must be exhaustively searched and so there are current four of these all based upon the line follow robot: an 81-point and a 625-point design space where the sensor positions are varied and a 216-point and 891-point design spaces where the controller parameters are varied. There are three measures applied to each result that target the tension between trading off the cost of running a DSE against the accuracy of the result

Cost: The simplest of the measures is the cost of the performing the search and here it is measured by the number of simulations performed to reach a result. For the purposes of comparison across the different design spaces, this cost is represented as a proportion of the total number of designs

$cost = \frac{|Simulations\ Run|}{|Design\ Space|}$

Accuracy: The ground truth exhaustive experiments provide us with the Pareto Front for that design space and each DSE experiment returns a non-dominated set of best designs found. Here the accuracy measure considers how many of the designs in the genetic non-dominated set are actually the best designs possible. It is measured by finding the proportion of points in the genetic NDS that are also found in the ground truth Pareto front.

$accuracy = \frac{|GeneticNDS \cap ExhaustiveNDS|}{|GeneticNDS|}$

Generational Distance: The accuracy measure tells us something about the points in the genetically found NDS that are also found in the exhaustive NDS (Van Veldhuizen & Lamont, 2000). The generational distance gives us a figure indicating the total distance between the genetic NDS and the exhaustive NDS. It is calculated by computing the sum of the distance between each point in the genetic NDS and its closest point in the exhaustive NDS and dividing this by the total number of points.

$generational\ distance = \frac{\sqrt{(\sum_{i=1}^{n} d_i^2)}}{n}$

9.2.3 Genetic DSE Experiments and Results

The DSE experiments involved varying three parameters of the genetic algorithm and repeating each set of parameters with each design space five times. The parameters of the genetic algorithm varied were:

Initial population size: The initial population size took one of three values. All design spaces were tested using an initial population of 10 designs, they were also tested with initial populations equal to 10% of the design space and 25% of the design space. These are represented on the left hand graphs by the 10, 10% and 25% lines.

Progress check conditions: The number of rounds the genetic algorithm would continue if there was no progress observed was tested with three values, 1, 5 and 10. These are represented on the right hand graphs with the 1, 5 and 10 lines.

Algorithm options: There are two variants of the genetic algorithm, phase 1 with random initial population and random parent selection, and phase 3 which give an initial population distributed over the design space and where parent selection is weighted to favour diverse parents. The phase one experiments are on the left hand side of the graphs, with points labelled '<design space size>-p1' while the phase three experiments are on the right labelled '<design space size>-p3'.

The results of the simulations are shown in graph form below. Each point graphed is the averaged result of the five runs of each set of parameters. Figure 9.14, shows the graphs of cost of running the DSEs. Encouragingly there is a slight trend of the cost of DSE reducing as a proportion of the design space as the design space size increases. As expected the cost was greater with larger initial populations but the cost did not vary when changing the progress check condition as much as expected.

Figure 9.15 shows the graphs of DSE accuracy. There is again a slight downward trend as design space size increases, meaning that there is a slight increase in the number of points in the genetic NDS that are not truly optimal. As expected the larger initial population generally resulted in more accurate NDS, this was also true of using the largest value for the progress check condition.

Figure 9.16 presents the generational distance results. Here we find that the results are generally low, with the exception of the 891-point design space which is significantly worse, the reason for this is still to be determined. The largest initial design space resulted in the lowest (best) values as did using a progress check condition value of five.

9.2.4 Selecting Approaches based on Design Space

The choice of which search algorithm to use when performing DSE is dependant on one factor, and that is a comparison of the 'simulations required' compared to the 'simulation budget', before describing what to do with the comparison, it is first necessary to explain those terms.

The number of 'simulations required' is dependant on the number of different design alternatives that the DSE is supposed explore, but there also other factors, specifically repetitions and scenarios. The 'design' part of the number of simulations required is determined by multiplying together the number of values each parameter may adopt, since this gives the number of unique designs. If the DSE configuration includes parameter constraints then the number of valid designs will be lower since some parameter combinations will fail to meet the constraints. For example, A line follower robot with two sensors, where each sensor has three possible x position values and three possible y position values, would have a design space size of 81, however if those parameters are constrained so designs must be symmetrical, i.e. the x and y values of each sensor must be identical, then the design space only has nine points.

If a simulation model contains random elements, such a noisy inputs to sensors or models of dropped messages on a network, then this leads to the simulation results being non-deterministic. In this case, it will be necessary to perform repeated simulations of the same design with the same starting conditions to account for the random variation.

'Scenarios' refer to the environment around the actual system-under-test in the simulation, for example, in the case of a line following robot, the environment could include the map that the robot is to follow along with other factors such as the intensity of the ambient light. If there is a desire to perform simulations under different scenarios, then the number of scenarios must also be taken into account when determining the required number of simulations. The final required number of simulations then is the product of the design space size, the number of repetitions and the number of scenarios.

The simulation budget term refers to the maximum number of simulations that a user may perform as part of DSE experiment. It is a matter for the user to determine the value for this budget, but it could be determined by determining the amount of CPU time allocated to the DSE and dividing it by the time to run a simulation and compute the objective values.

The decision of whether to use the exhaustive search algorithm or a closed loop search can be made by comparing the number of simulations required with the simulation budget. If

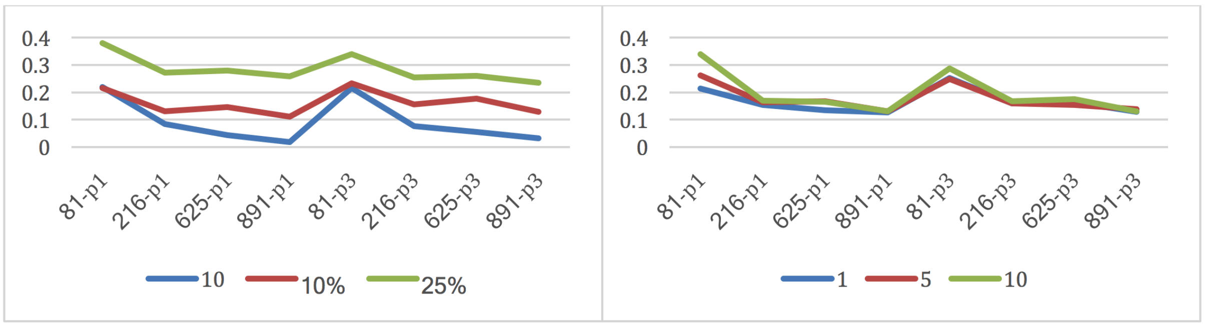


Figure 9.14: Cost of DSE, number of simulations run as proportion of the total design space.

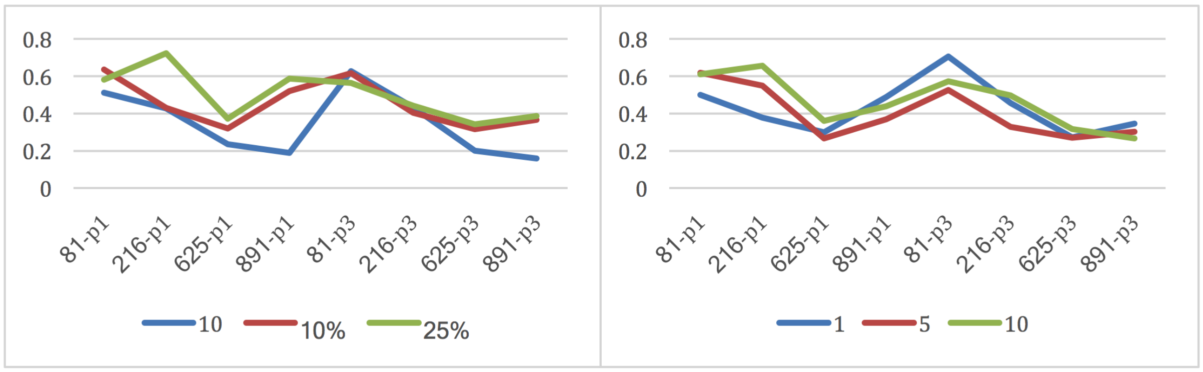


Figure 9.15: Accuracy of DSE, proportion of genetic NDS found in exhaustive NDS.

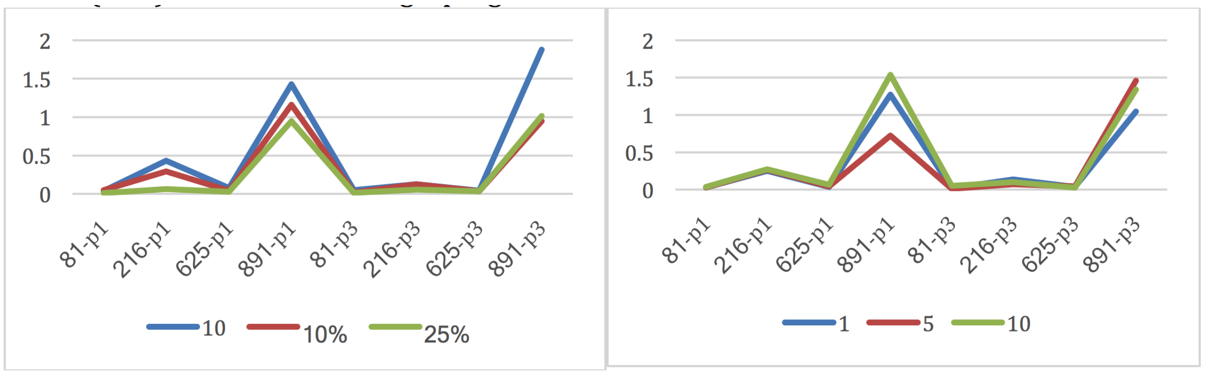


Figure 9.16: Gap between Genetic NDS and Exhaustive NDS. The vertical axis has no meaningful units, a smaller number is better.

the simulation budget is greater than or equal to the required number of simulations, then an exhaustive search should be used as this guarantees to find the optimal designs given the design parameter values, if the budget is less than the required number of simulations required then a closed loop approach is needed.

Parameters for a Genetic Search

If the decision is made to perform a genetic search, then it is important to note that there are two parameters that affect how the algorithm behaves and will have an effect on the outcome. The first of these parameters is the initial population size, this defines how many random designs are generated at the start of the search as a seed for the process. A general rule for this initial population size is that it should be 10 times the number of dimensions (parameters) [65], with the caveat that as the number of dimensions increases, this multiplier must also also increase.

The second parameter is the termination condition, or the number generations the algorithm will continue without seeing progress before it terminates. The genetic algorithm measures progress by looking at the designs that make up the non-dominated set of the Pareto analysis. The only way membership of this set can change between two generations is if better designs, according to the objective measures, have been found, so if membership changes then the search is making progress towards finding better designs. It is not unusual for the algorithm to breed new designs that are not better than those currently in the non-dominated set and so to have a generation that does not show progress, but then to make progress in a subsequent generation. Thus, the number of generations without progress parameter is used to relax the termination condition to permit generations without progress without stopping. Increasing this value will increase the probability that the search will not become stuck in some local optima in the results and may progress to find better designs. There is a cost associated with increasing this value since, as the algorithm produces two new designs per generation, there will be two times the number of generations without progress simulations run at the end of the process that do not lead to better results [66].

Iterative Exhaustive Search

An alternative to the genetic search, which is automated, is to use repeated exhaustive searches to home in on better regions of the design space. In this approach the user would plan to perform multiple DSE experiments, each using some portion of their total simulation budget. The first DSE experiment is used to cover the whole range of the design space, but not including all values for each parameter. In this way the first DSE is used to locate regions of interest within the design space. The regions of interest are areas of the design space that produced the better designs according to the ranking results, with the bounds of the 'area' defined by the parameter values that produced good results. The user then divides up their remaining simulation budget between the one or more areas of interest and perform further DSE on those areas. Figure 9.17 shows an example initial search, with the blue dots indicating simulations performed and the green areas giving the best results. The user then divides their remaining simulation budget among the three green areas of interest, searching each with a higher resolution in an attempt to extract the best results from each, Figure 9.18.

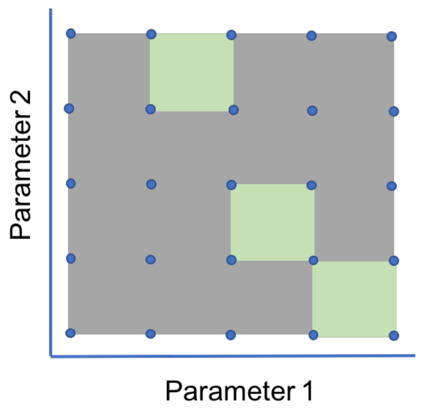


Figure 9.17: Step 1 of an iterative search. The best results being found in the green regions

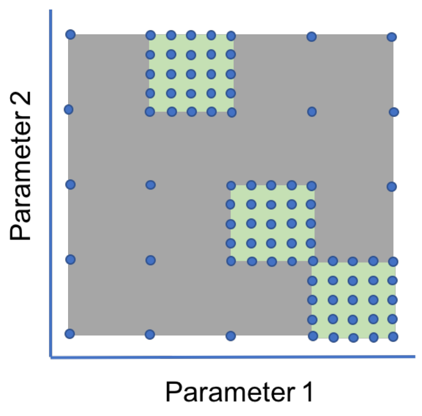


Figure 9.18: Step 2 of an iterative search. The green regions are searched with a higher resolution to find the best results

Bibliography

[1] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Casper Thule, Anders Franz Terkelsen, Carl Gamble, Adrian Pop, Etienne Brosse, Jörg Brauer, Florian Lapschies, Marcel Groothuis, Christian Kleijn, and Luis Diogo Couto. INTO-CPS Tool Chain User Manual. Technical report, INTO-CPS Deliverable, D4.3a, December 2017.

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

[3] Julien Ouy, Thierry Lecomte, Frederik Forchhammer Foldager, Andres Villa Henriksen, Ole Green, Stefan Hallerstede, Peter Gorm Larsen, Luis Diogo Couto, Pasquale Antonante, Stylianos Basagiannis, Sara Falleni, Hassan Ridouane, Hajer Saada, Erica Zavaglio, Christian König, and Natalie Balcu. Case Studies 3, Public Version. Technical report, INTO-CPS Public Deliverable, D1.3a, December 2017.

[4] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, and Marcel Verhoef. A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. *Mathematical Structures in Computer Science*, 23(4):726–750, 2013.

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

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

[7] INCOSE. Systems Engineering Handbook. A Guide for System Life Cycle Processes and Activities, Version 4.0. Technical Report INCOSE-TP-2003-002-04, International Council on Systems Engineering (INCOSE), January 2015.

[8] Jan F. Broenink, John Fitzgerald, Carl Gamble, Claire Ingram, Angelika Mader, Jelena Marincic, Yunyun Ni, Ken Pierce, and Xiaochen Zhang. Methodological guidelines 3. Technical report, The DESTECS Project (INFSO-ICT-248134), October 2012.

[9] Haydn Thompson, editor. *Cyber-Physical Systems: Uplifting Europe's Innovation Capacity*. European Commission Unit A3 - DG CONNECT, December 2013.

[10] Lipika Deka, Zoe Andrews, Jeremy Bryans, Michael Henshaw, and John Fitzgerald. D1.1 definitional framework. Technical report, The TAMS4CPS Project, April 2015.

[11] J. Holt, C. Ingram, A. Larkham, R. Lloyd Stevens, S. Riddle, and A. Romanovsky. Convergence report 3. Technical report, COMPASS Deliverable, D11.3, September 2014.

[12] Job van Amerongen. *Dynamical Systems for Creative Technology*. Controllab Products, Enschede, Netherlands, 2010.

[13] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[14] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.

[15] Richard J. Payne and John S. Fitzgerald. Evaluation of Architectural Frameworks Supporting Contract-based Specification. Technical Report CS-TR-1233, School of Computing Science, Newcastle University, December 2010.

[16] Simon Perry, Jon Holt, Richard Payne, Jeremy Bryans, Claire Ingram, Alvaro Miyazawa, Luís Diogo Couto, Stefan Hallerstede, Anders Kaels Malmos, Juliano Iyoda, Marcio Cornelio, and Jan Peleska. Final Report on SoS Architectural Models. Technical report, COMPASS Deliverable, D22.6, September 2014. Available at http://www.compass-research.eu/.

[17] Nuno Amalio, Richard Payne, Ana Cavalcanti, and Etienne Brosse. Foundations of the SysML profile for CPS modelling. Technical report, INTO-CPS Deliverable, D2.1a, December 2015.

[18] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Model-based engineering of systems of systems. *Submitted to ACM Computing Surveys*, June 2013.

[19] Torsten Blochwitz. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads`, July 2014.

[20] Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. 1st Intl. Conf. on Requirements Engineering*, pages 94–101, April 1994.

[21] Luc Moreau and Paul Groth. PROV-Overview. Technical report, World Wide Web Consortium, 2013.

[22] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 2006.

[23] Joey Coleman, Anders Kaels Malmos, Luis Couto, Peter Gorm Larsen, Richard Payne, Simon Foster, Uwe Schulze, and Adalberto Cajueiro. Third release of the COMPASS tool — symphony ide user manual. Technical report, COMPASS Deliverable, D31.3a, December 2013.

[24] RTCA SC-167/EUROCAE WG-12. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178B, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 1992.

[25] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[26] Amir Pnueli. The Temporal Logic of Programs. In *18th Symposium on the Foundations of Computer Science*, pages 46–57. ACM, November 1977.

[27] E. M. Clarke and A. E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *IBM Logics of Programs Workshop*, volume LNCS 131. Springer Verlag, 1981.

[28] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[29] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[30] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[31] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[32] E.M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2003.

[33] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[34] OMG Systems Modeling Language (OMG SysML^{TM}). Technical Report Version 1.3, SysML Modelling team, June 2012. http://www.omg.org/spec/SysML/1.3/.

[35] Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications – A Formal Introduction*. John Wiley & Sons Ltd, 1992.

[36] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[37] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer language. Technical Report PRG-6, Oxford Programming Research Group Technical Monograph, 1971.

[38] Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.

[39] Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.

[40] He Jifeng. A classical mind. chapter From CSP to Hybrid Systems, pages 171–189. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[41] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[42] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[43] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

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

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

[46] Frank Zeyda, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. SysML Foundations: Case Study. Technical report, INTO-CPS Deliverable, D2.3a, December 2017.

[47] Ana Cavalcanti, Simon Foster, Bernhard Thiele, Jim Woodcock, and Frank Zeyda. Final Semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.3b, December 2017.

[48] Frank Zeyda, Simon Foster, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. A Mechanised FMI Semantics. Technical report, INTO-CPS Deliverable, D2.3c, December 2017.

[49] Ana Cavalcanti, Simon Foster, Jim Woodcock, and Frank Zeyda. Multi-Model Linking Semantics. Technical report, INTO-CPS Deliverable, D2.3d, December 2017.

[50] John Fitzgerald, Carl Gamble, and Ken Pierce. Methods Progress Report 3. Technical report, INTO-CPS Deliverable, D3.3b, December 2017.

[51] Kenneth Lausdahl, Peter Niermann, Jos Höll, Carl Gamble, Oliver Mölle, Etienne Brosse, Tom Bokhove, Luis Diogo Couto, and Adrian Pop. INTO-CPS Traceability Design. Technical report, INTO-CPS Deliverable, D4.2d, December 2016.

[52] Christian König, Kenneth Lausdahl, Peter Niermann, Jos Höll, Carl Gamble, Oliver Mölle, Etienne Brosse, Tom Bokhove, Luis Diogo Couto, and Adrian Pop. INTO-CPS Traceability Implementation. Technical report, INTO-CPS Deliverable, D4.3d, December 2017.

[53] Stefan Wiesner, Christian Gorldt, Mathias Soeken, Klaus-Dieter Thoben, and Rolf Drechsler. Requirements engineering for cyber-physical systems - challenges in the context of "industrie 4.0". volume 438 of *IFIP Advances in Information and Communication Technology*, pages 281–288. Springer, 2014.

[54] Birgit Penzenstadler and Jonas Eckhardt. A Requirements Engineering content model for Cyber-Physical Systems. In *RESS*, pages 20–29, 2012.

[55] Grace A. Lewis, Edwin Morris, Patrick Place, Soumya Simanta, and Dennis B. Smith. Requirements Engineering for Systems of Systems. In *Systems Conference, 2009 3rd Annual IEEE*, pages 247–252. IEEE, March 2009.

[56] Jon Holt, Simon Perry, Richard Payne, Jeremy Bryans, Stefan Hallerstede, and Finn Overgaard Hansen. A model-based approach for requirements engineering for systems of systems. *IEEE Systems Journal*, 9(1):252–262, 2015.

[57] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Methods Progress Report 1. Technical report, INTO-CPS Deliverable, D3.1b, December 2015.

[58] Nuno Amalio, Ana Cavalcanti, Alvaro Miyazawa, Richard Payne, and Jim Woodcock. Foundations of the SysML for CPS modelling. Technical report, INTO-CPS Deliverable, D2.2a, December 2016.

[59] Etienne Brosse and Imran Quadri. COE Contracts from SysML. Technical report, INTO-CPS Deliverable, D4.1c, December 2015.

[60] Etienne Brosse and Imran Quadri. SysML and FMI in INTO-CPS. Technical report, INTO-CPS Deliverable, D4.2c, December 2016.

[61] Etienne Brosse. SysML and FMI in INTO-CPS. Technical report, INTO-CPS Deliverable, D4.3c, December 2017.

[62] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2013.

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

[64] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Methods Progress Report 2. Technical report, INTO-CPS Deliverable, D3.2b, December 2016.

[65] Pedro A Diaz-Gomez and Dean F Hougen. Initial population for genetic algorithms: A metric approach. In *GEM*, pages 43–49, 2007.

[66] John Fitzgerald, Carl Gamble, Richard Payne, and Benjamin Lam. Exploring the Cyber-Physical Design Space. In *Proc. INCOSE Intl. Symp. on Systems Engineering*, volume 27, pages 371–385, Adelaide, Australia, 2017.

Part III

Tutorials

Latest Versions

These tutorials are a snapshot at the end of the project, but will become out-of-date as the technology continues to evolve as part of the INTO-CPS Association. The latest versions can be found at the link below along with the source projects referred to in the tutorials.

`https://github.com/INTO-CPS-Association/training/releases`

Tutorial 1 — First Co-simulation

Overview

This first INTO-CPS tutorial will show you how to:

1. Open a project in the INTO-CPS App
2. Run a co-simulation
3. View a 3D plot (Windows only)

Requirements

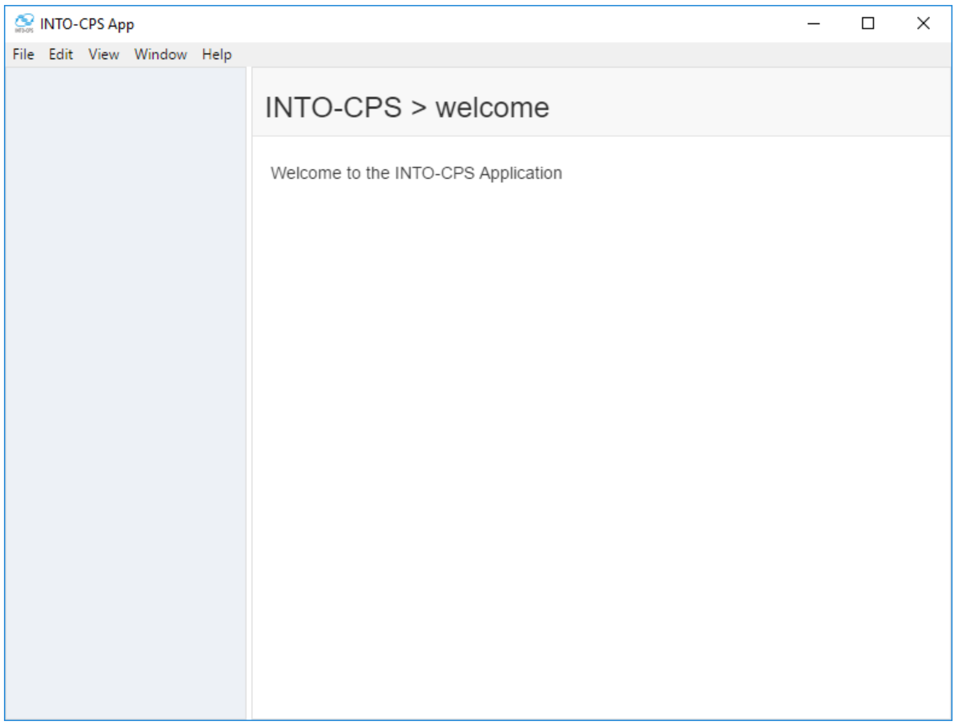
This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- INTO-CPS Application
- COE (Co-simulation Orchestration Engine) accessible to the Application

You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window* > *Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

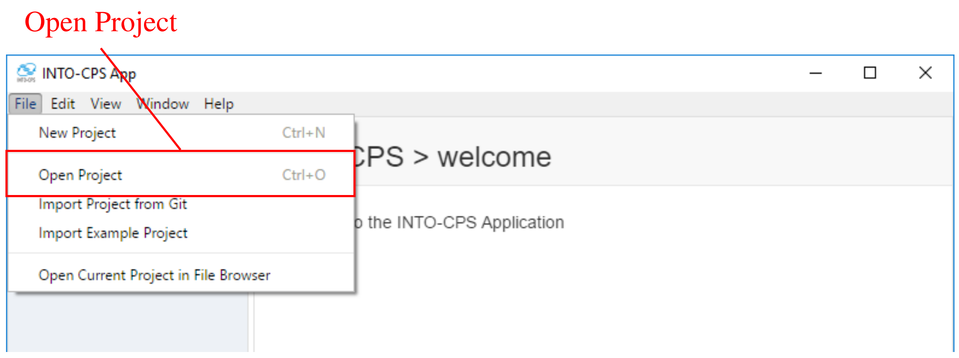
1 Opening a Project

Step 1. Launch the *INTO-CPS Application*. On first loading, it will look like the screenshot below. If you have opened a project previously, that project will be opened automatically.

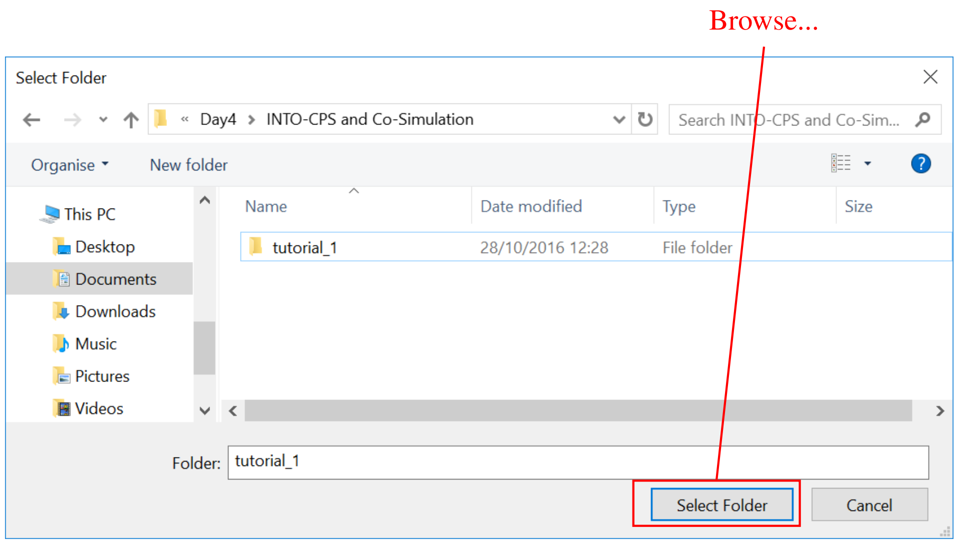


Welcome to the INTO-CPS Application

Step 2. To open a project, select *File > Open Project*.



Step 3. Find *Tutorials/tutorials_1*, select it and press *Select Folder*.



Step 4. Once the project is opened, you will see that project browser on the left of the INTO-CPS Application window is now populated. The entries in the project browser correspond to folders and files in the *Tutorials/tutorials_1* folder.

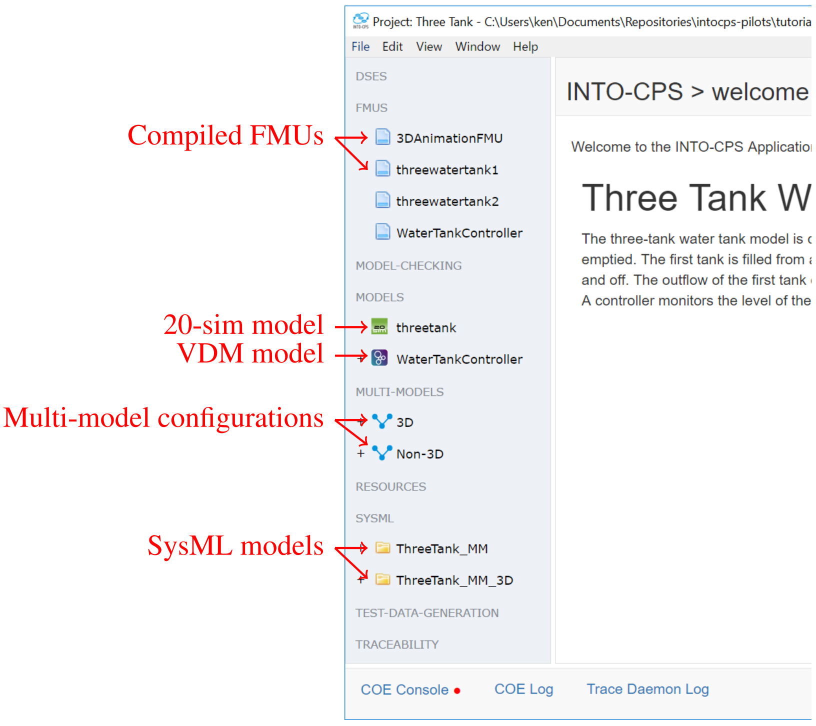
The elements in the *tutorial_1* project are:

FMUs Compiled FMUs (with file extension .fmu) that are used in co-simulation.

Models Source models used to generate the FMUs. The icon of each entry shows which tool created the model. In this case Overture and 20-sim.

Multi-models Used to configure co-simulations, including which FMUs are used and other co-simulation settings.

SysML Architectural models that are used to create model and multi-model descriptions.

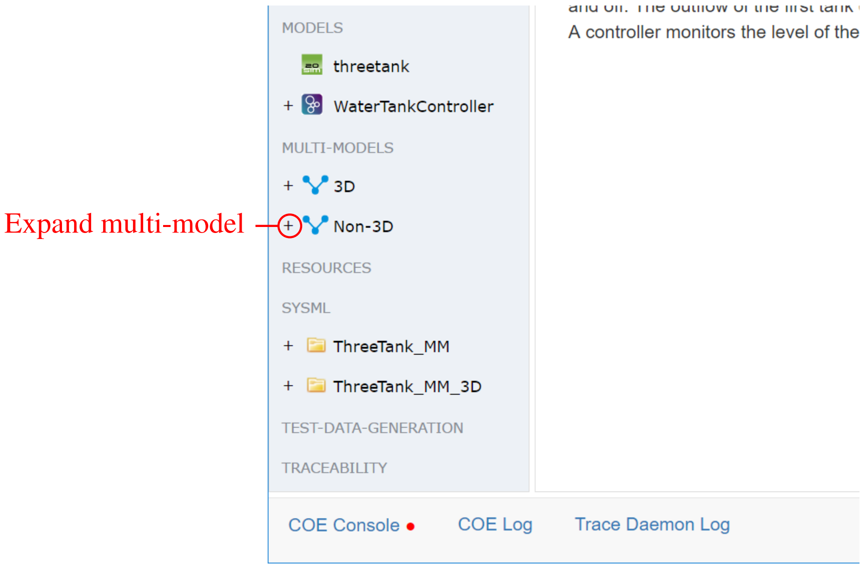


The three-tank water tank model is c
emptied. The first tank is filled from a
and off. The outflow of the first tank
A controller monitors the level of the

2 Running a Co-simulation

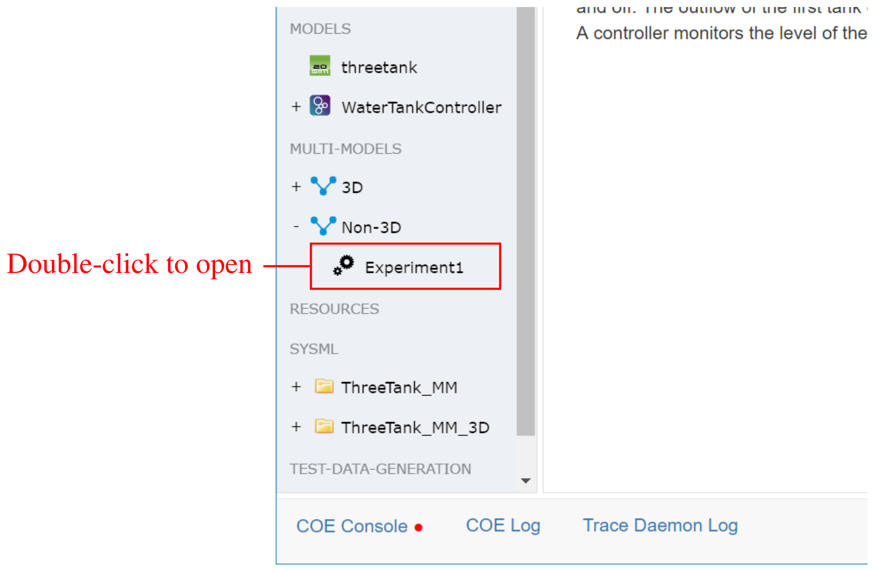
To run a co-simulation we must use one of the multi-model configurations. We'll start with the *Non-3D* multi-model (since it works on all platforms).

Step 5. Click the + symbol next to *Non-3D* multi-model to expand it.

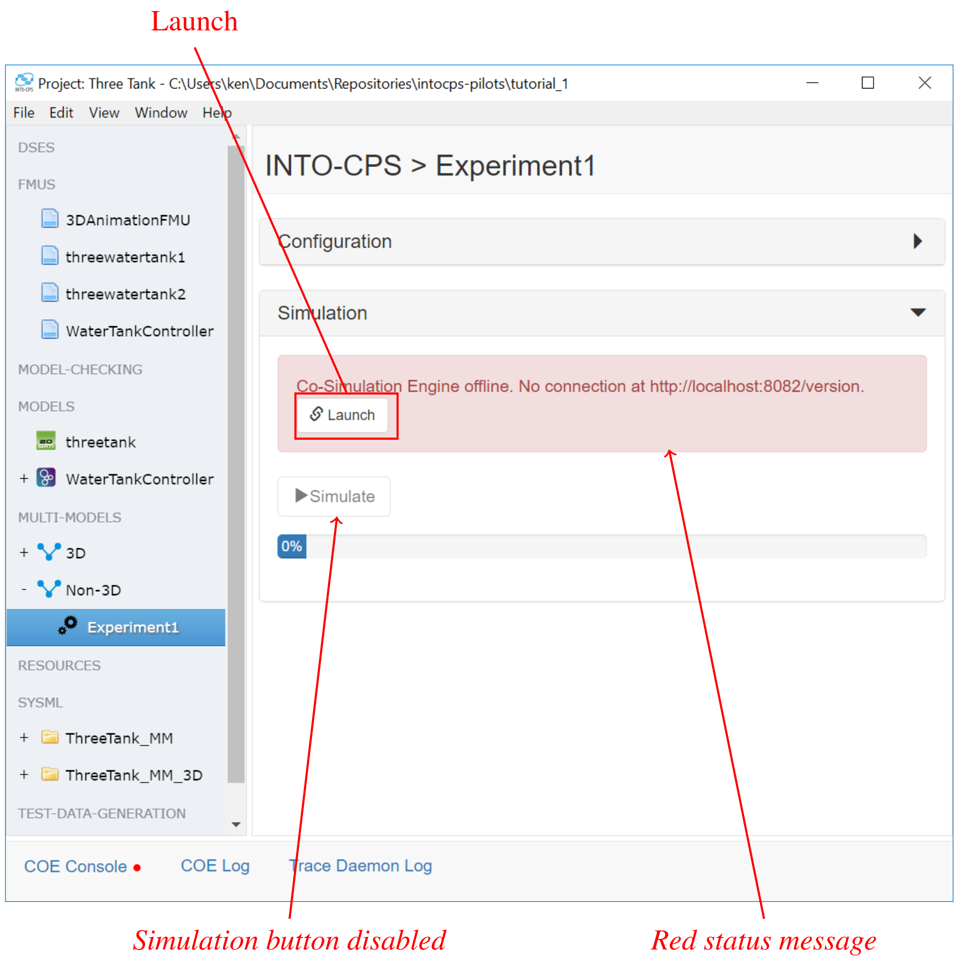


and off. The outflow of the first tank
A controller monitors the level of the

Step 6. There is one co-simulation configuration in this multi-model called *Experiment1*. Double-click this to open this configuration.

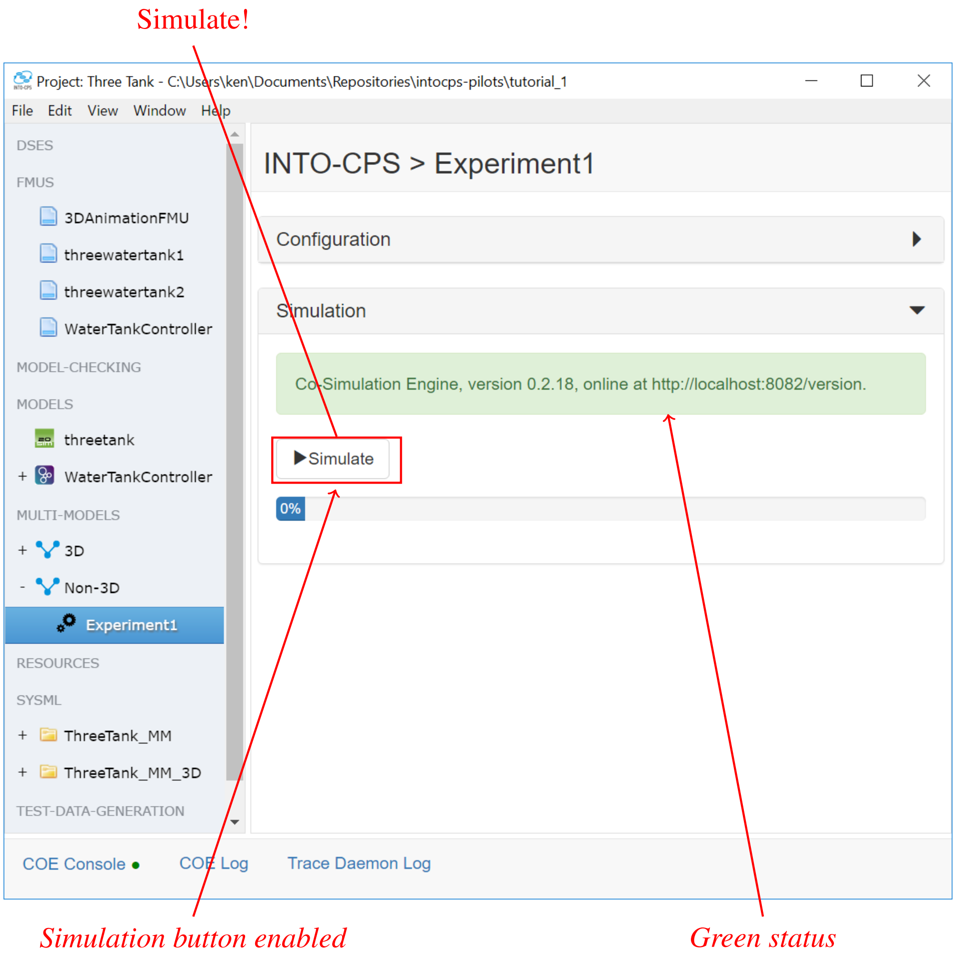


Step 7. Once the *Experiment1* co-simulation configuration is open, you will see the following screen. The COE (Co-simulation Orchestration Engine) is a separate tool from the INTO-CPS Application. This screen gives the status, which is offline. To launch it, press *Launch*.



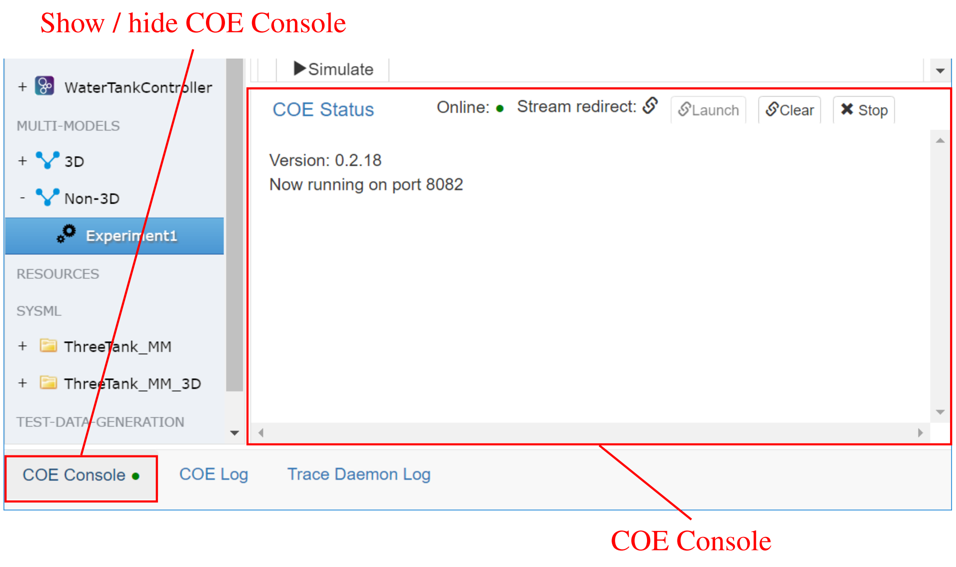
Co-Simulation Engine offline. No connection at http://localhost:8082/version.

Step 8. Once the COE is online, the status message will become green and the *Simulate* button will be enabled. Press *Simulate* to run a co-simulation.



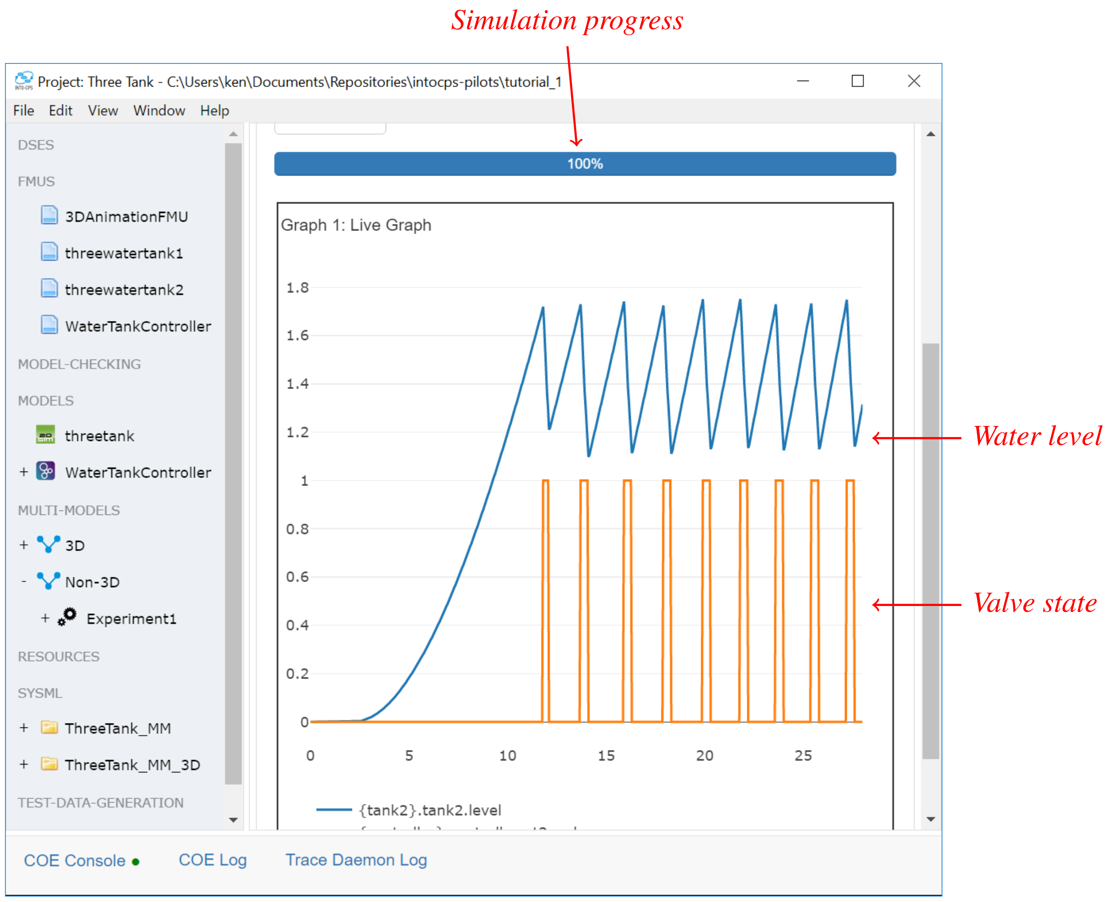
Co-Simulation Engine, version 0.2.18, online at http://localhost:8082/version.

You can also see the status of the COE in the bottom left corner. Pressing here brings up the COE Console, which shows output from the COE and allows you to stop and launch the COE. Pressing the button again will hide the console.



```
Version: 0.2.18
Now running on port 8082
```

Step 9. When simulating, you may see a Java console windows appearing, status information will appear in the *COE Console*, and a *live plot* will show variables in the model across time.



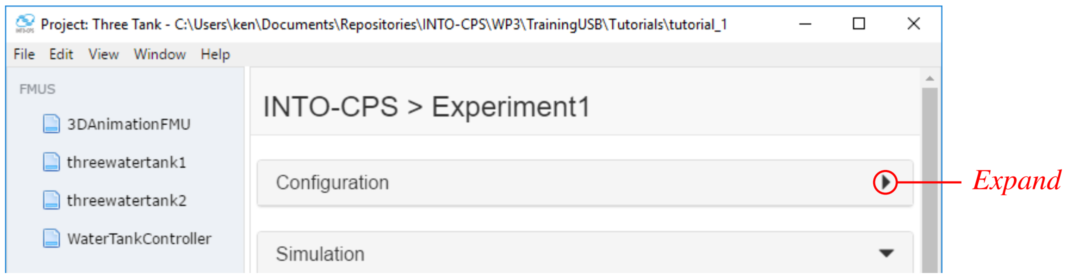
This multi-model is of a water tank system. The live plot shows the water level in one of the tanks that is constantly being filled, and the state of the valve (1 = open, 0 = closed) that allows water to flow out of the tank. You can see the water level rise and fall as the controller opens and closes the valve. The water tank was modelled in 20-sim and the controller modelled in VDM.

Congratulations!

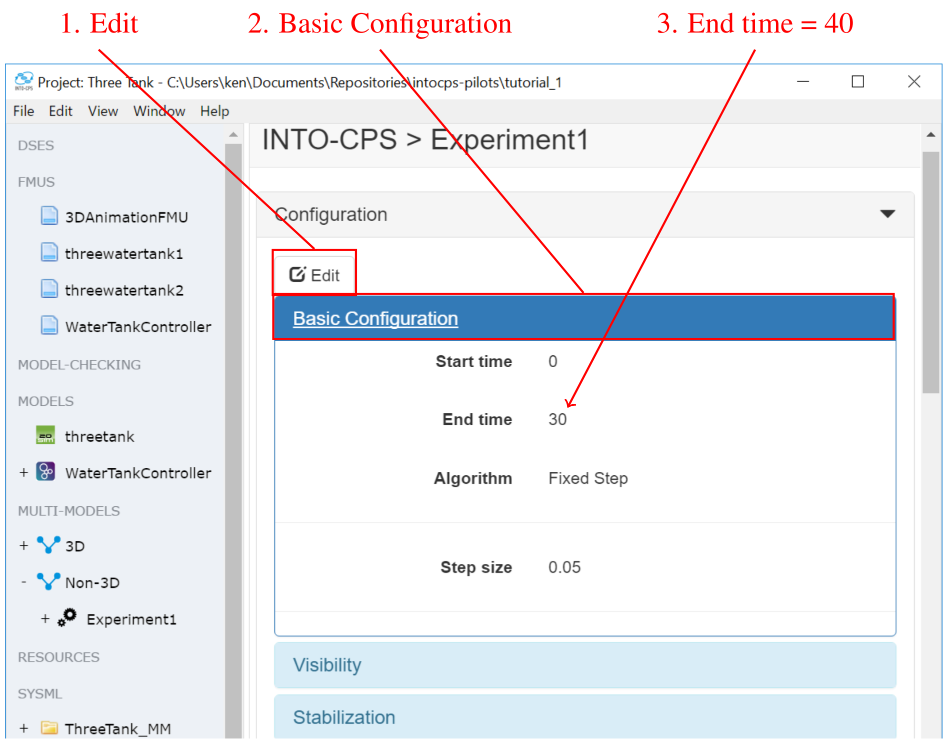
You have completed your first co-simulation with the INTO-CPS Application.

3 Changing Co-simulation Parameters

Step 10. We can change the length of the co-simulation, the parameters of the master algorithm, and the variables that are plotted using the *Configuration* pane of the co-simulation configuration. Expand the pane by pressing the triangle.

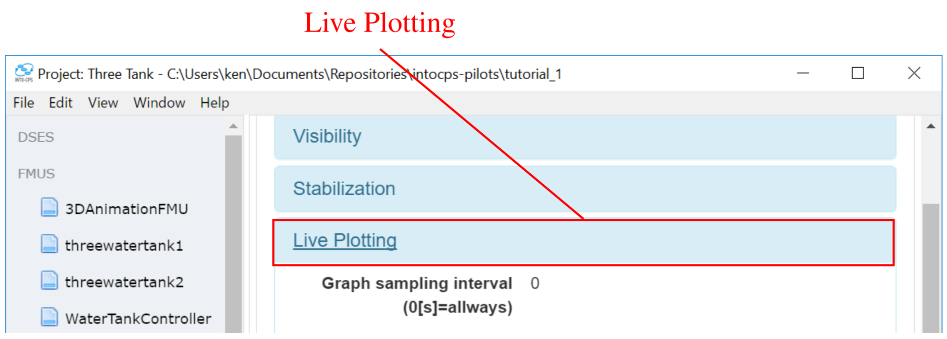

Project: Three Tank - C:\Users\ken\Documents\Repositories\INTO-CPS\WP3\TrainingUSB\Tutorials\tutorial_1
File Edit View Window Help
FMUS
3DAnimationFMU
threewatertank1
threewatertank2
WaterTankController
INTO-CPS > Experiment1
Configuration
Simulation
Expand

Step 11. Press the *Edit* button, which allows you to make changes, then press *Basic Configuration*. Set the *End time* to 40 (seconds).



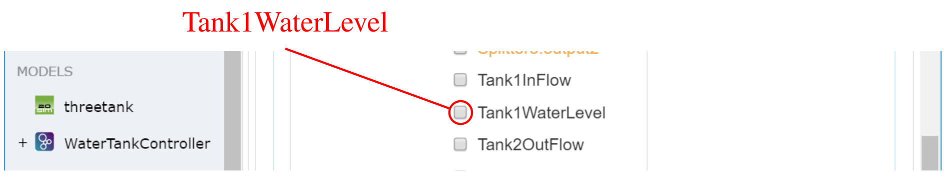
Start time	0
End time	30
Algorithm	Fixed Step
Step size	0.05

Step 12. Press *Live Plotting*.

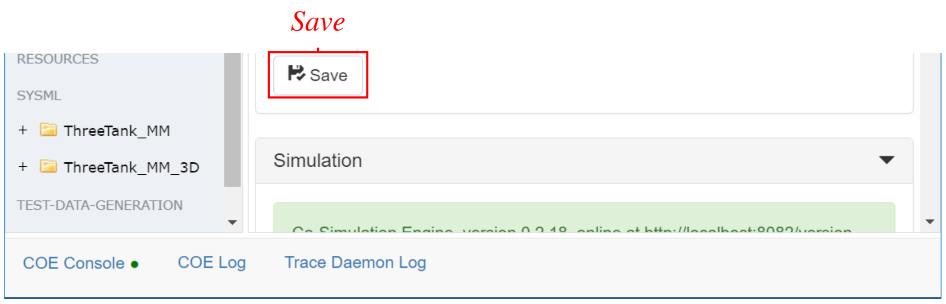


Graph sampling interval (0[s]=allways) 0

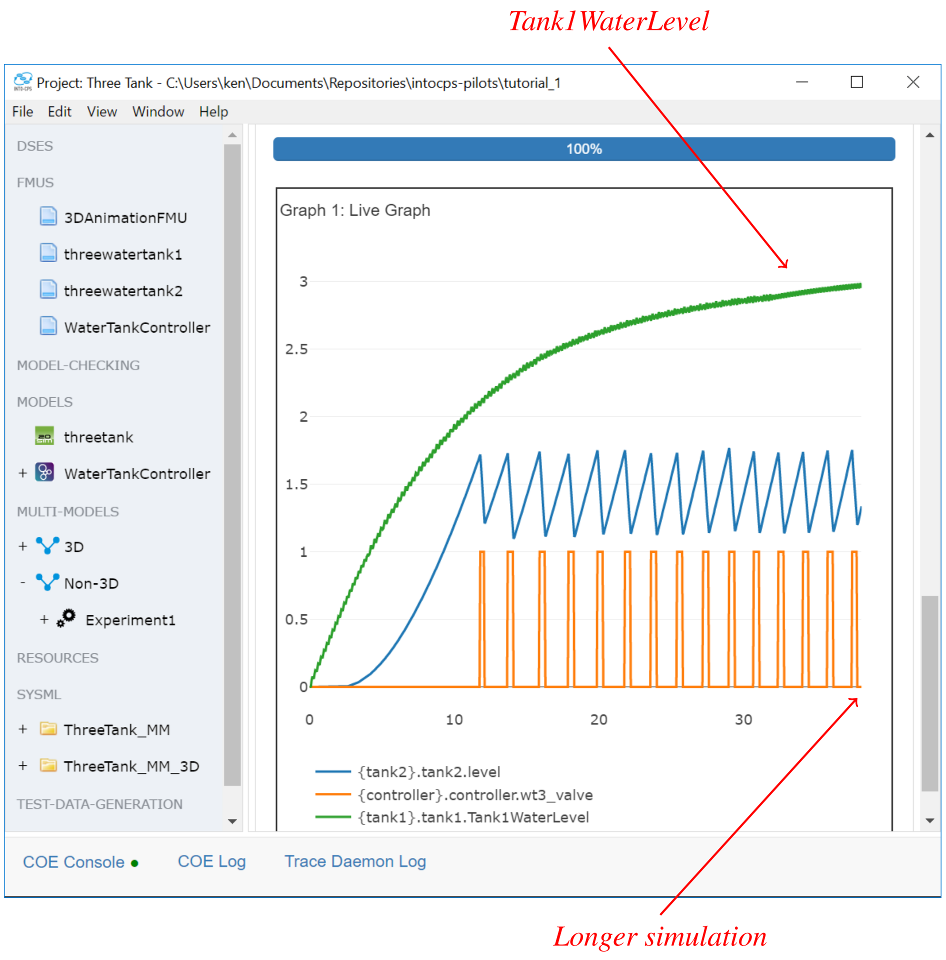
Scroll down to find **tank1.tank1** and check *Tank1WaterLevel*.



Step 13. Press the *Save* button.



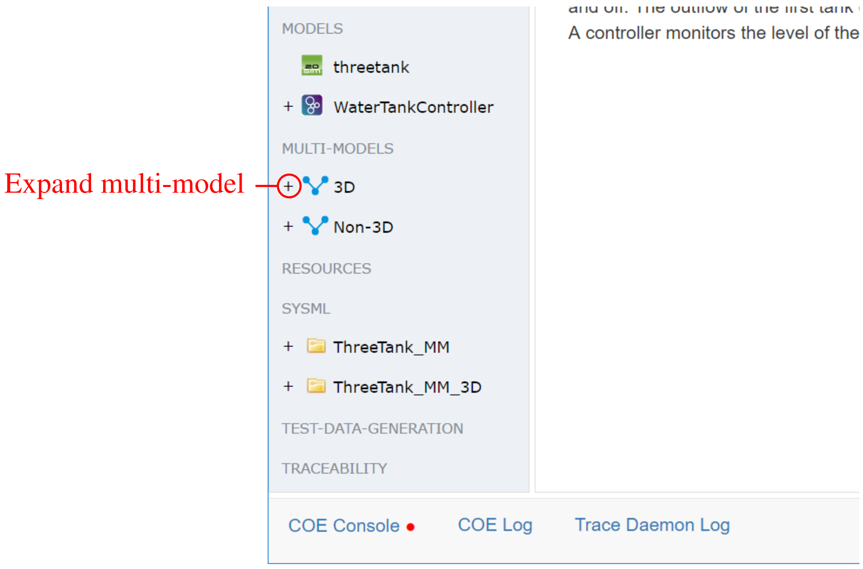
Step 14. Run the co-simulation again with the *Simulate* button. You will see a new variable on the graph, and that the co-simulation runs for a further 10 (simulated) seconds than before.



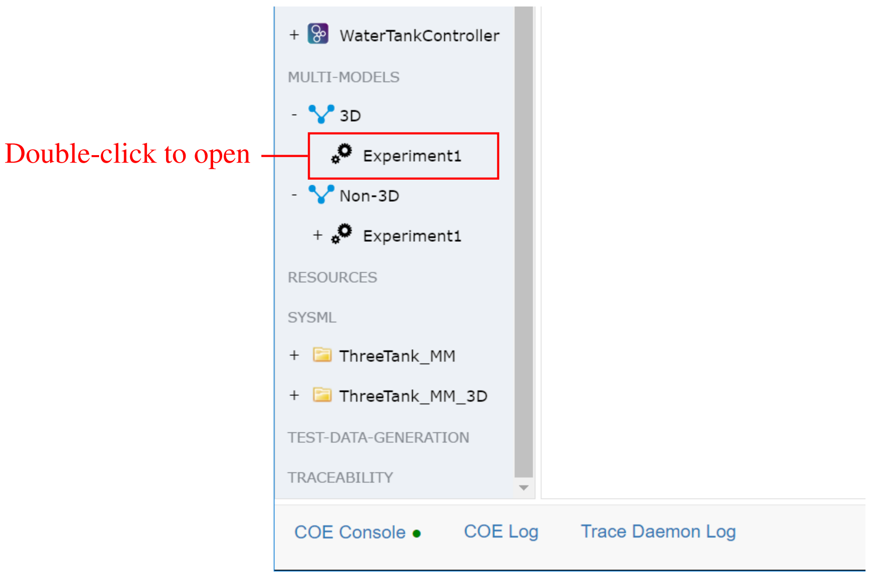
4 Viewing a 3D Plot (Windows Only)

The 20-sim tool is able to create 3D visualisations of simulations, which are linked to variables in a model. These can be included within an FMU generated by 20-sim, however *this feature is currently only available on the Windows platform*.

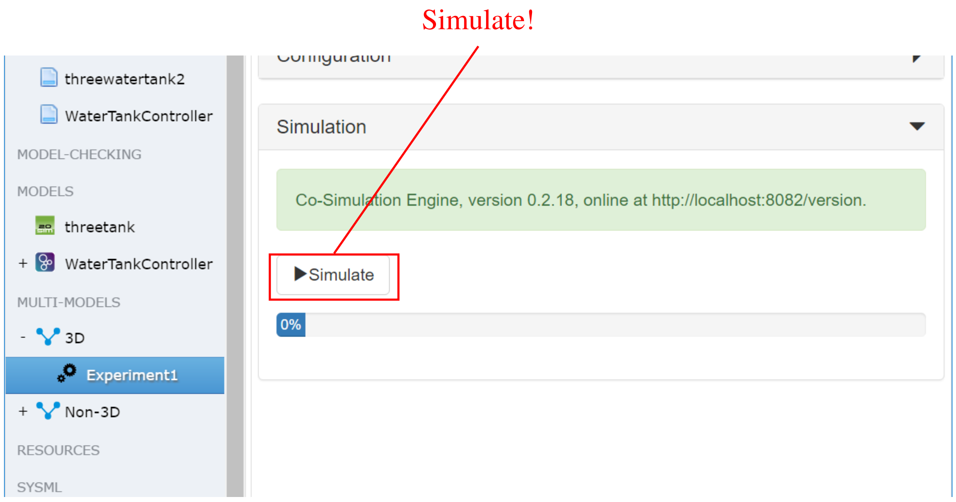
Step 15. The *3DAnimationFMU* is included in the multi-model configuration called *3D*. Click the + symbol next to the *3D* multi-model to expand it.



Step 16. There is one co-simulation configuration in this multi-model, also called *Experiment1*. Double-click this to open this configuration.

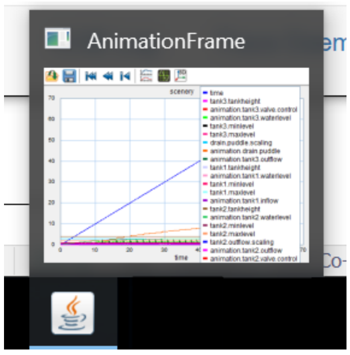


Step 17. Click *Simulate*. See Step Step 7. if the *Simulate* button is disabled or the COE is offline.

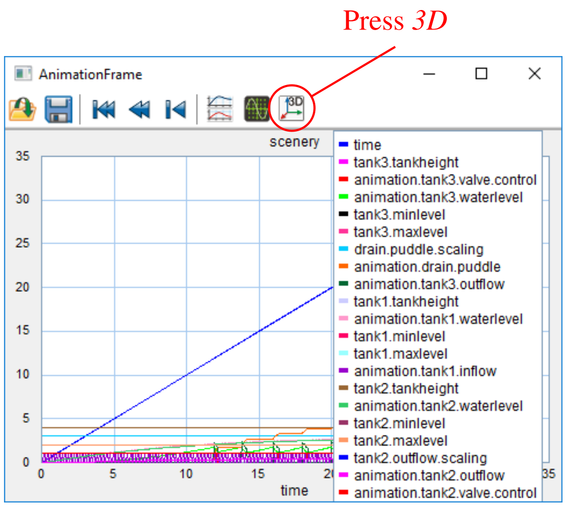


Co-Simulation Engine, version 0.2.18, online at http://localhost:8082/version.

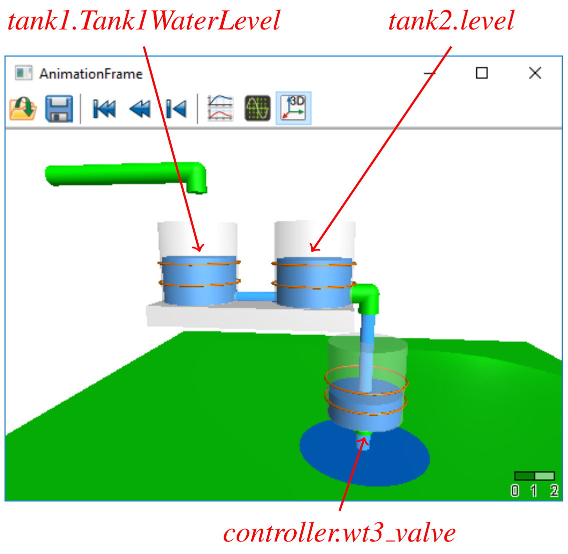
Step 18. The *3DAnimationFMU* launches as a Window called *AnimationFrame*.



To see the 3D visualisation, you must press the button called *3D*.



Step 19. The *AnimationFrame* window should now show you a 3D scene with water levels changing as seen on the live plot. The valve empties water on to a puddle on the floor.



Warning: The *3DAnimationFMU* will crash the COE if the *AnimationFrame* does not have focus when the simulation ends. If this happens, simply relaunch the COE as covered in Step Step 7..

Tutorial 2 — Adding FMUs

Overview

This second tutorial will show you how to:

1. Edit a multi-model configuration
2. Add a new FMU to a multi-model configuration
3. Execute a co-simulation using the new multi-model configuration

Requirements

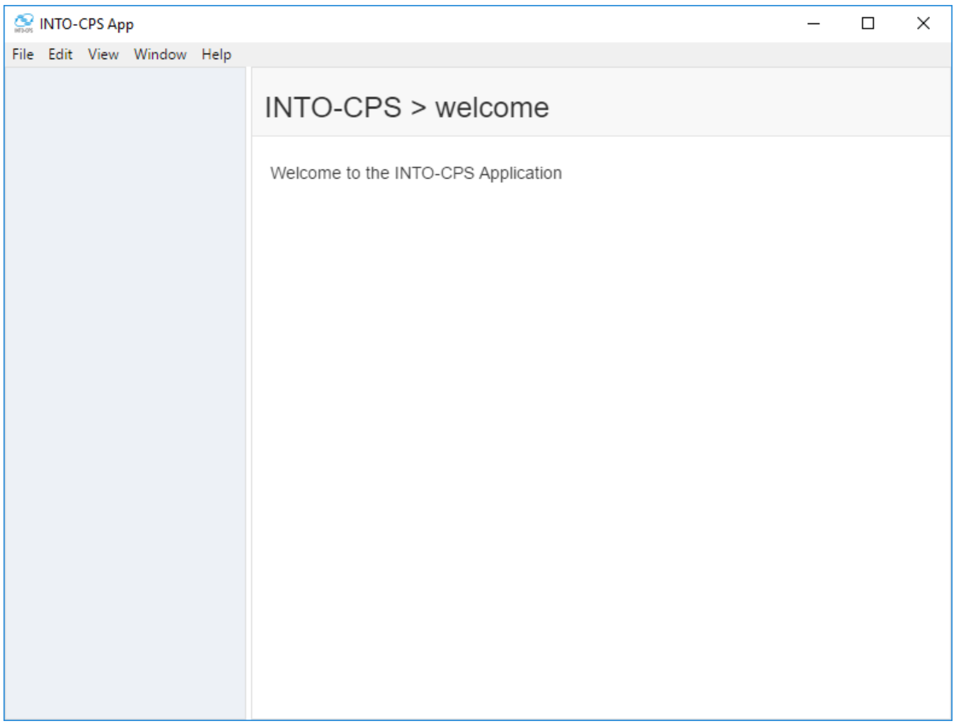
This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- INTO-CPS Application
- COE (Co-simulation Orchestration Engine) accessible to the Application

You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window > Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

1 Opening a Project

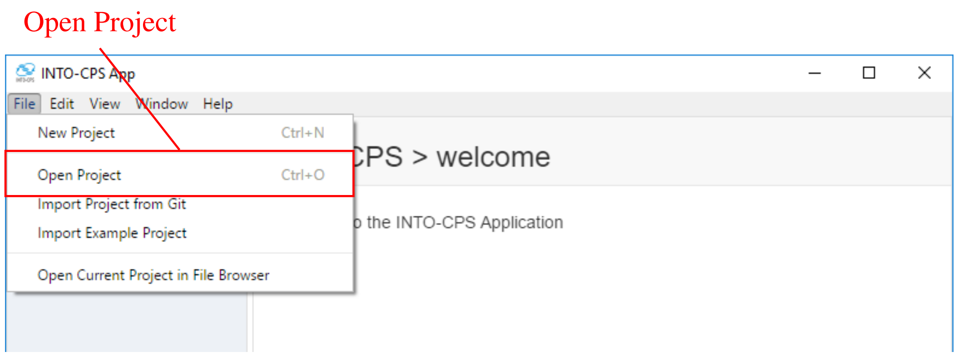
Step 1. Launch the *INTO-CPS Application*. On first loading, it will look like the screenshot below. If you have opened a project previously, that project will be opened automatically.



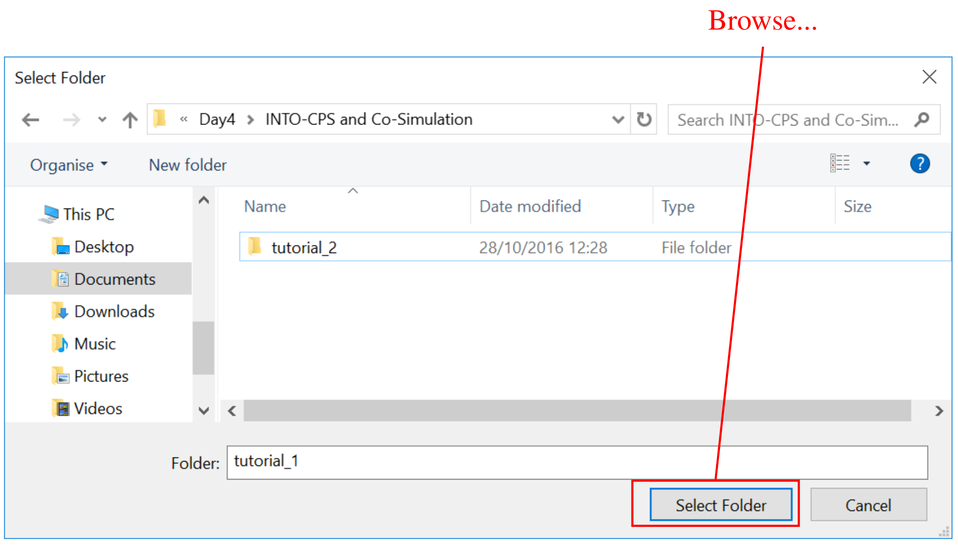
INTO-CPS > welcome

Welcome to the INTO-CPS Application

Step 2. To open a project, select *File > Open Project*.



Step 3. Find *Tutorials/tutorials_2*, select it and press *Select Folder*.



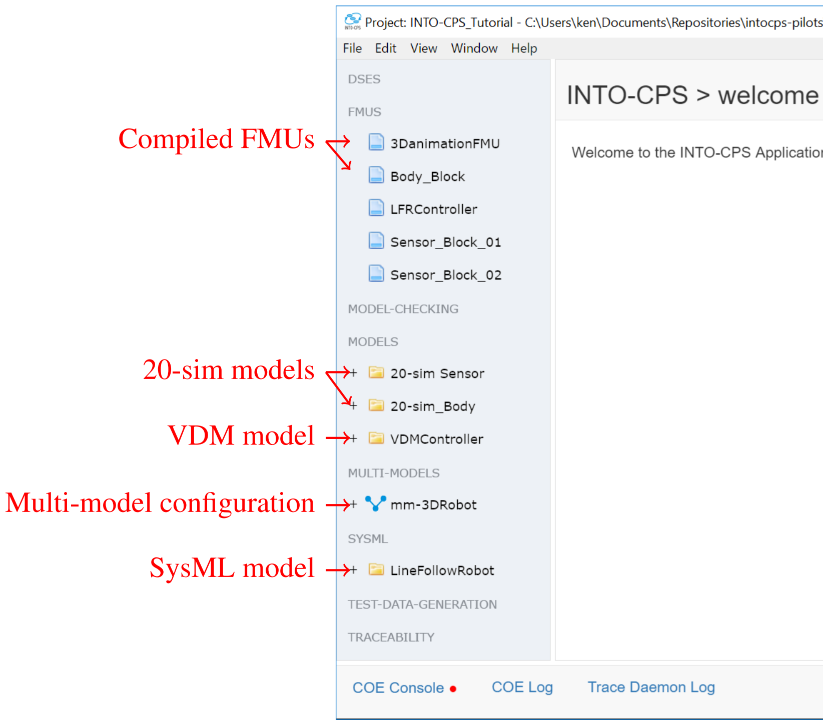
Step 4. Once the project is opened, you will see that project browser on the left of the INTO-CPS Application window is now populated. The entries in the project browser correspond to folders and files in the *Tutorials/tutorials_2* folder. These are:

FMUs Compiled FMUs (with file extension .fmu) that are used in co-simulation.

Models Source models used to generate the FMUs. The icon of each entry shows which tool created the model. In this case Overture and 20-sim.

Multi-models Used to configure co-simulations, including which FMUs are used and other co-simulation settings.

SysML Architectural models that are used to create model and multi-model descriptions.

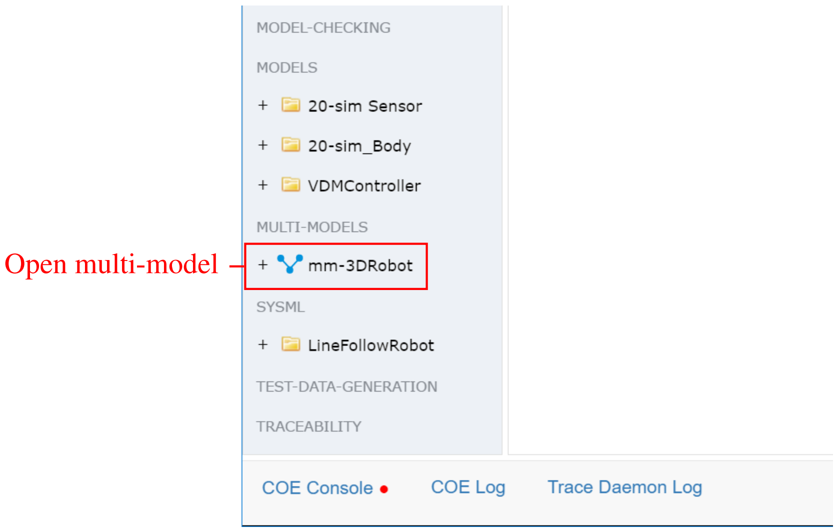


This multi-model is a line-following robot, which contains a model of the body (including wheels and motors), a model of the sensors, and a controller which reads the sensors and controls the motors to make the robot follow the line.

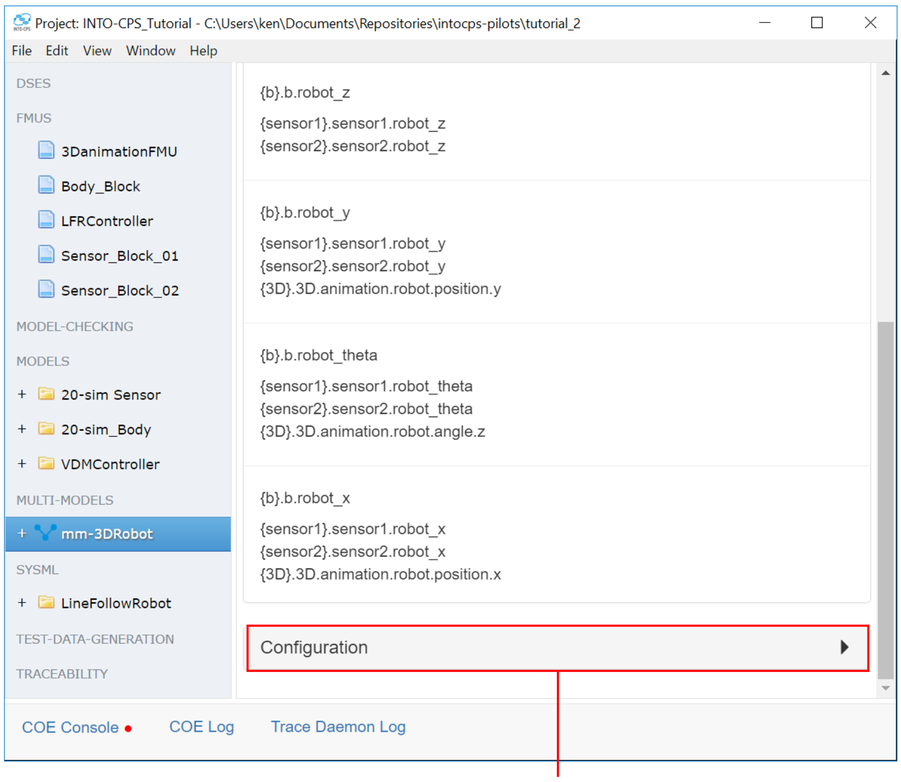
2 Editing a Multi-model

In this tutorial, the controller FMU has not been added to the multi-model, and multi-model parameters are missing. We will add the FMU, set the parameters, then run a co-simulation.

Step 5. Double-click the *mm-3DRobot* to open it.



Step 6. Click on *Configuration* to expand it.



{b}.b.robot_z
{sensor1}.sensor1.robot_z
{sensor2}.sensor2.robot_z

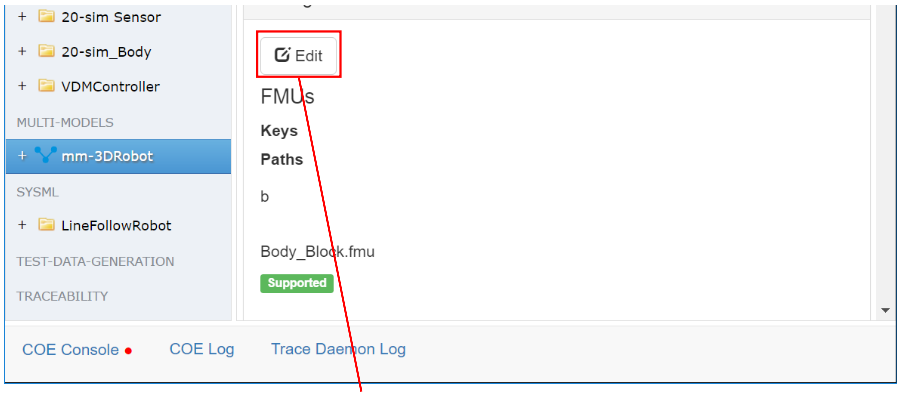
{b}.b.robot_y
{sensor1}.sensor1.robot_y
{sensor2}.sensor2.robot_y
{3D}.3D.animation.robot.position.y

{b}.b.robot_theta
{sensor1}.sensor1.robot_theta
{sensor2}.sensor2.robot_theta
{3D}.3D.animation.robot.angle.z

{b}.b.robot_x
{sensor1}.sensor1.robot_x
{sensor2}.sensor2.robot_x
{3D}.3D.animation.robot.position.x

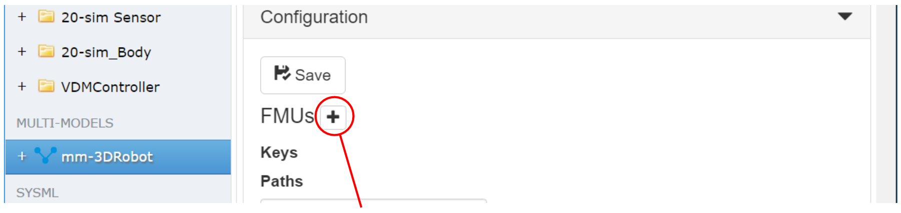
Configuration

Step 7. Click *Edit*.



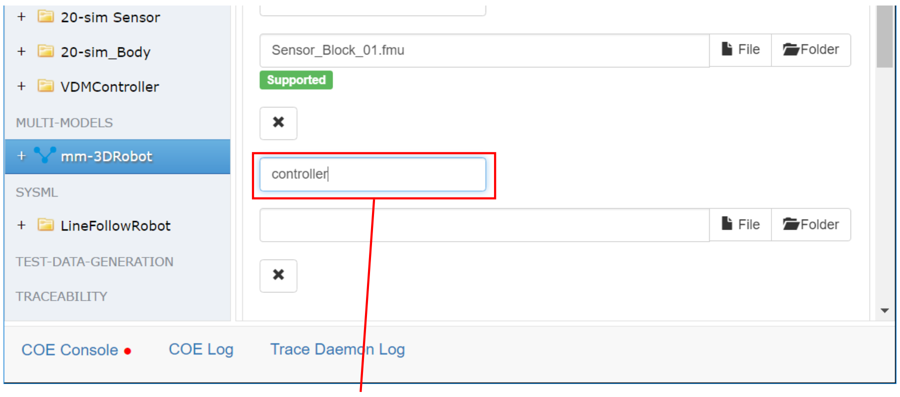
Edit

Step 8. Click the + icon to add a new FMU entry.



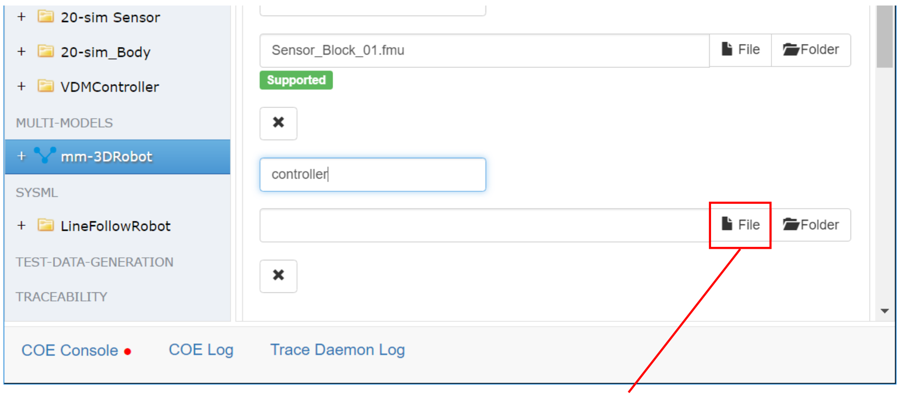
Add FMU

Step 9. The new entry is named *FMU*. Rename it to *controller*.



Rename to *controller*

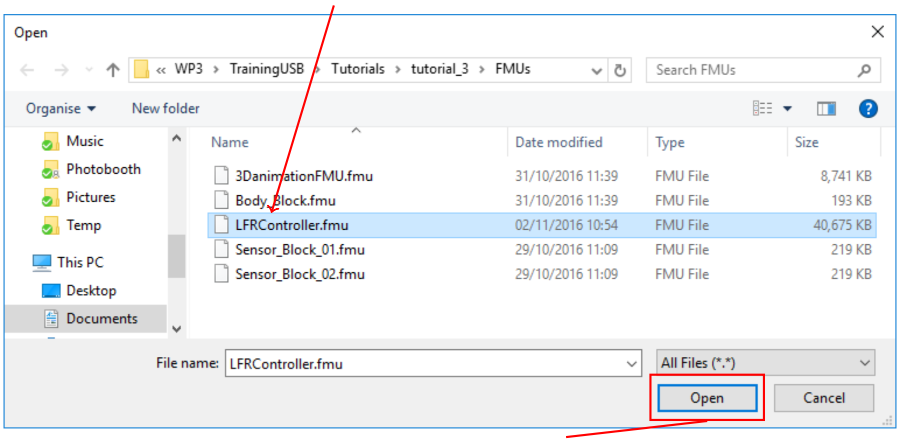
Step 10. We need to associate an FMU with this entry. To do this, click the *File* button.



File

Step 11. A file browser window will open and show five FMUs (if the file browser does not show the FMUs, navigate to *tutorials_2/FMUs*). Select *FMUController.fmu* and click *Open*.

1. Locate and select *FMUController.fmu*

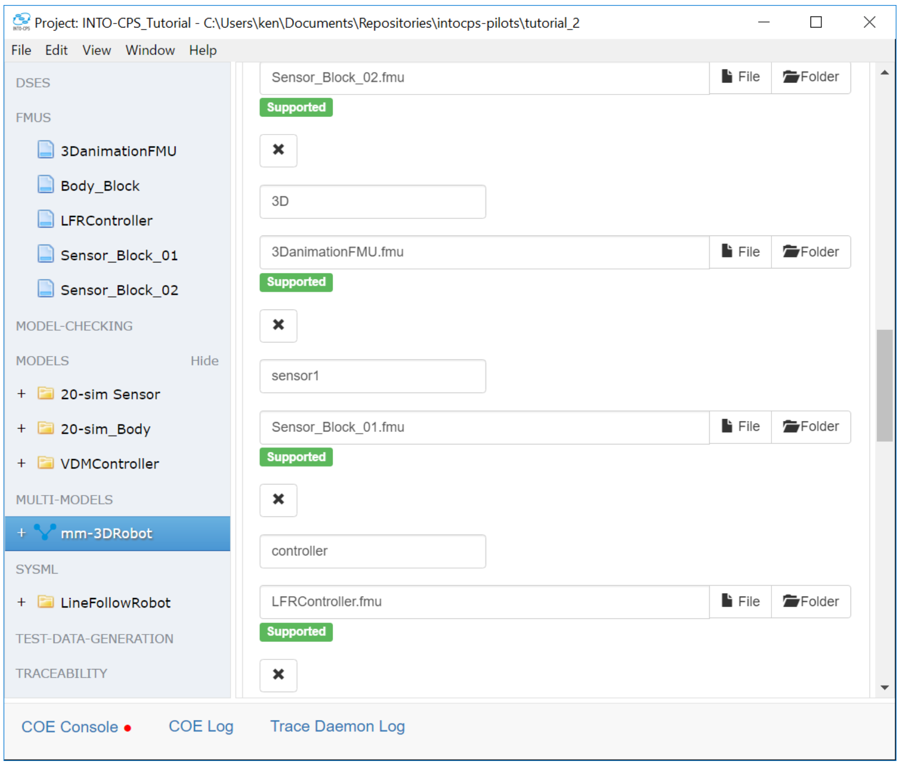


Name	Date modified	Type	Size
3DanimationFMU.fmu	31/10/2016 11:39	FMU File	8,741 KB
Body_Block.fmu	31/10/2016 11:39	FMU File	193 KB
LFRController.fmu	02/11/2016 10:54	FMU File	40,675 KB
Sensor_Block_01.fmu	29/10/2016 11:09	FMU File	219 KB
Sensor_Block_02.fmu	29/10/2016 11:09	FMU File	219 KB

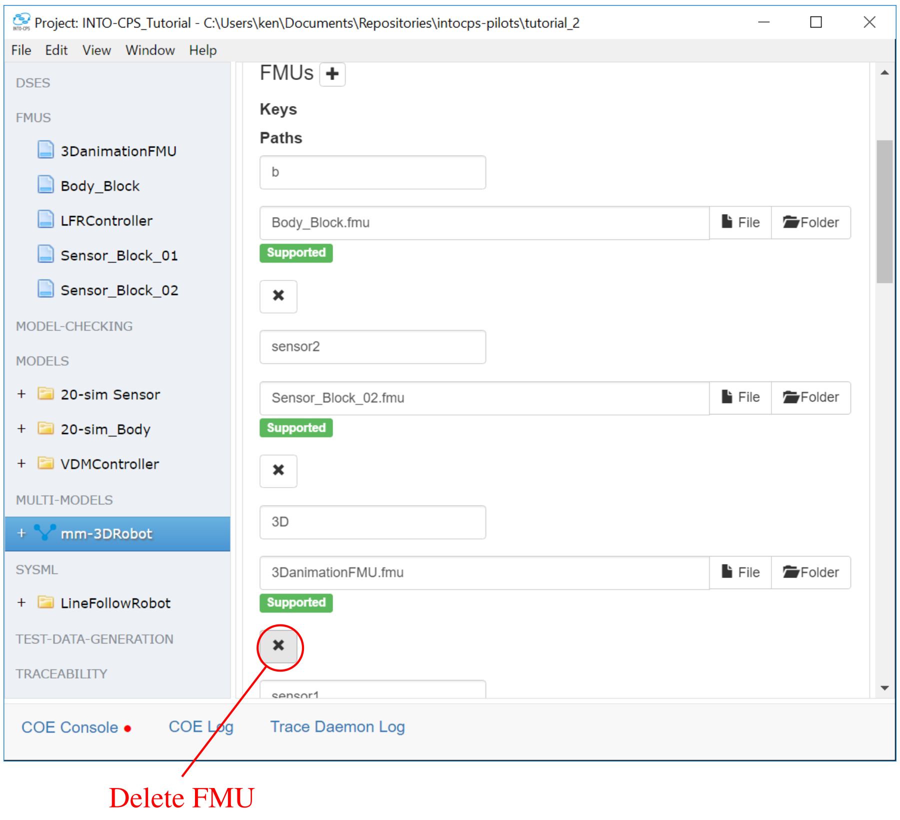
File name: LFRController.fmu

2. Click *Open*

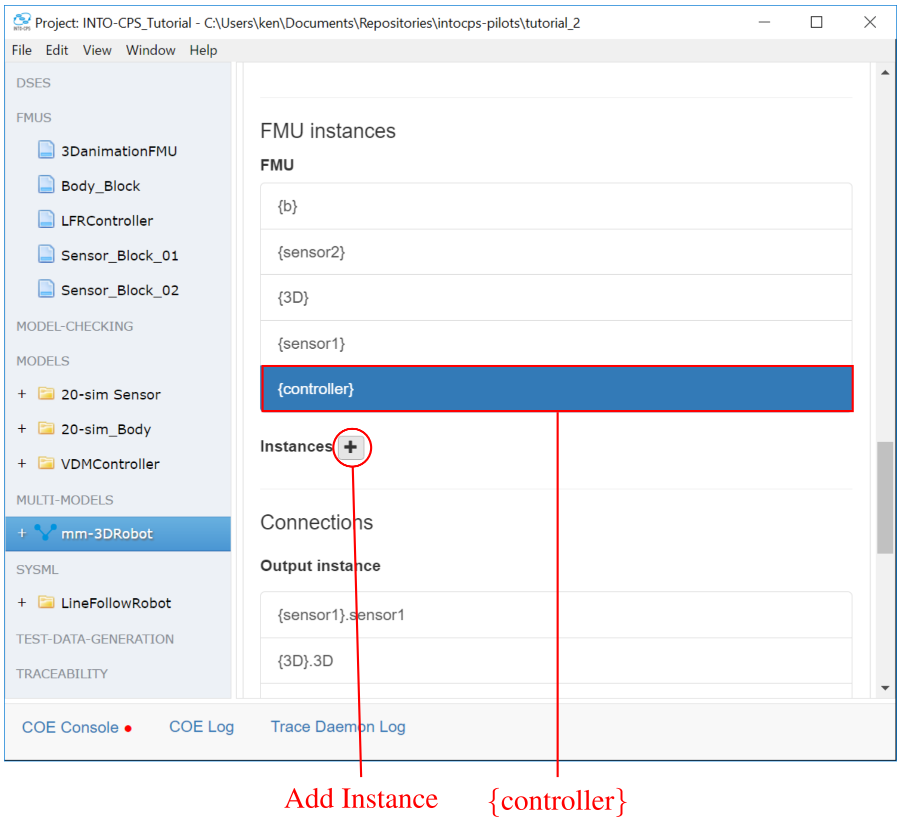
Step 12. The controller FMU is now associated.



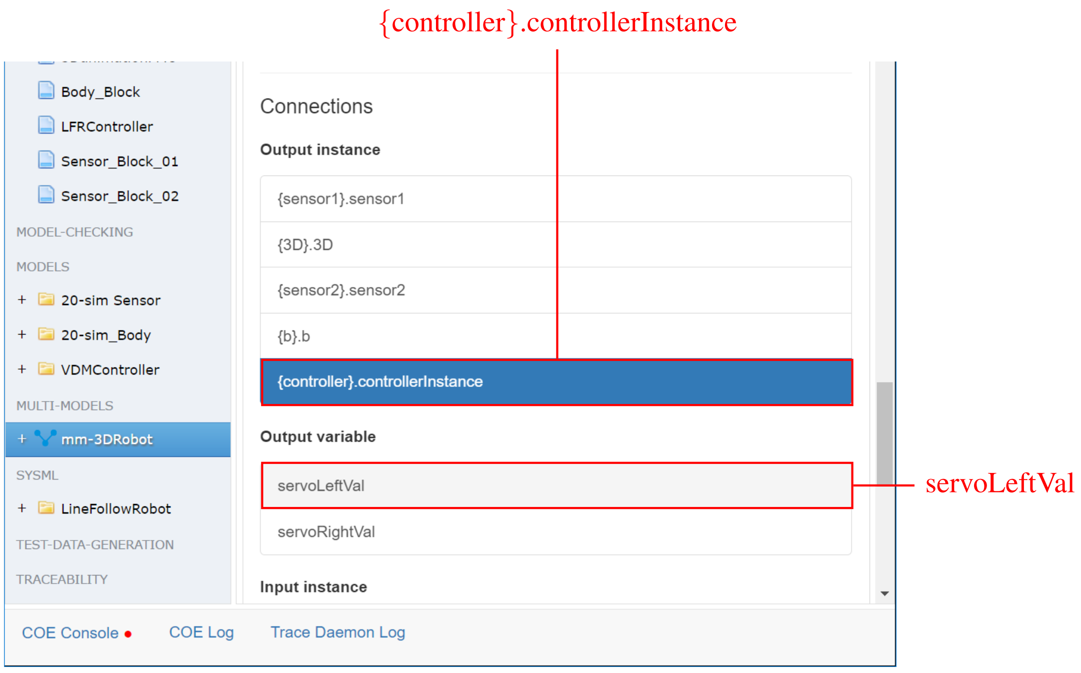
Step 13. **MacOS X / Linux Only:** As in the first tutorial, the 3D visualisation FMU from this multi-model is only supported on Windows. Therefore delete it using the *X* button.



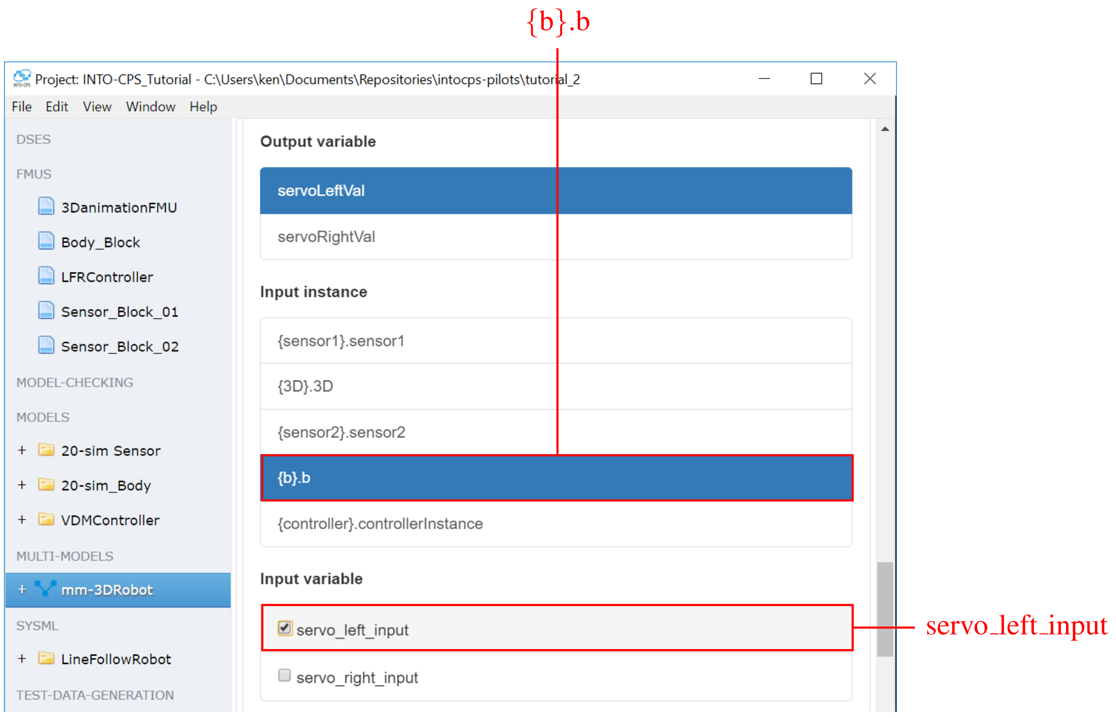
Step 14. Next we must add an instance of the controller. Under *FMU Instances* click {*controller*} then click the + icon.



Step 15. Next the controller outputs must be connected to the body inputs (to control the motors). Scroll down to *Connections*. Under **Output instance** click on {*controller*}.*controllerInstance* and under **Output variable** select *servoLeftVal*.

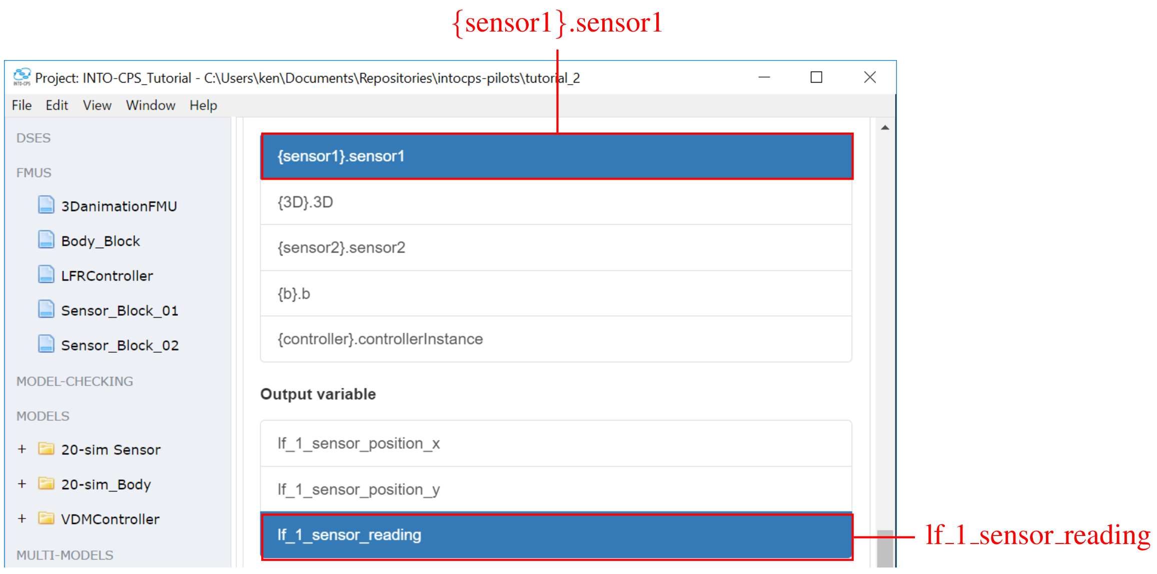


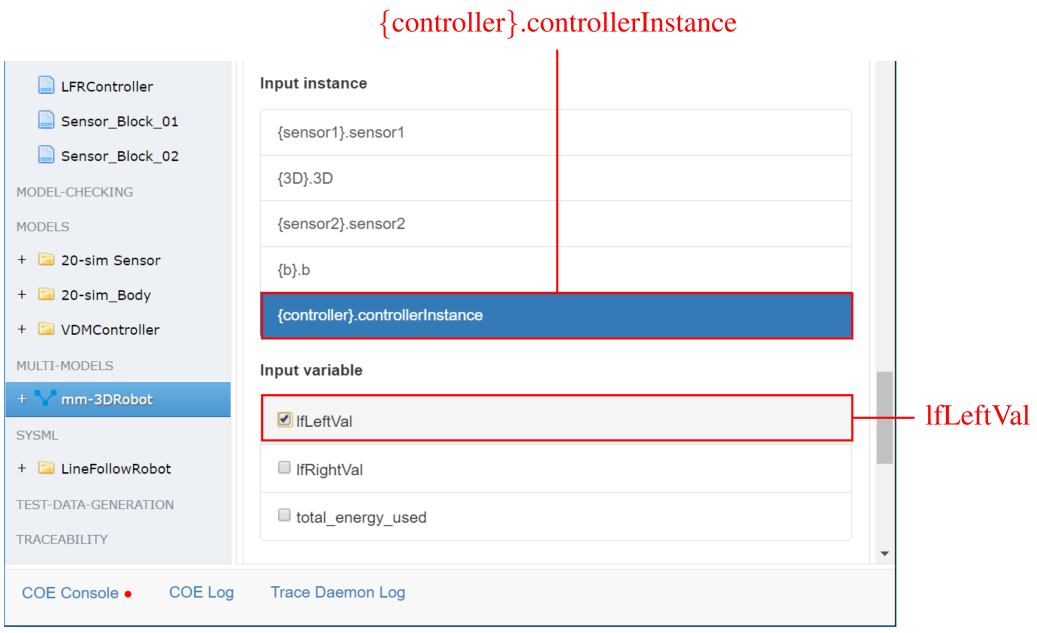
Step 16. Then under **Input instance** select $\{b\}.b$ and under **Input variable** check *servo_left_input*.



Step 17. Repeat the previous step to connect $\{controller\}.controllerInstance$ / *servoRightVal* to $\{b\}.b$ / *servo_right_input*.

Step 18. Then repeat to connect $\{sensor1\}.sensor1$ / *lf_1__sensor_reading* to $\{controller\}.controllerInstance$ / *lfLeftVal*.





LFRController
Sensor_Block_01
Sensor_Block_02
MODEL-CHECKING
MODELS
+ 20-sim Sensor
+ 20-sim_Body
+ VDMController
MULTI-MODELS
+ mm-3DRobot
SYSML
+ LineFollowRobot
TEST-DATA-GENERATION
TRACEABILITY

Input instance

{sensor1}.sensor1
{3D}.3D
{sensor2}.sensor2
{b}.b
{controller}.controllerInstance

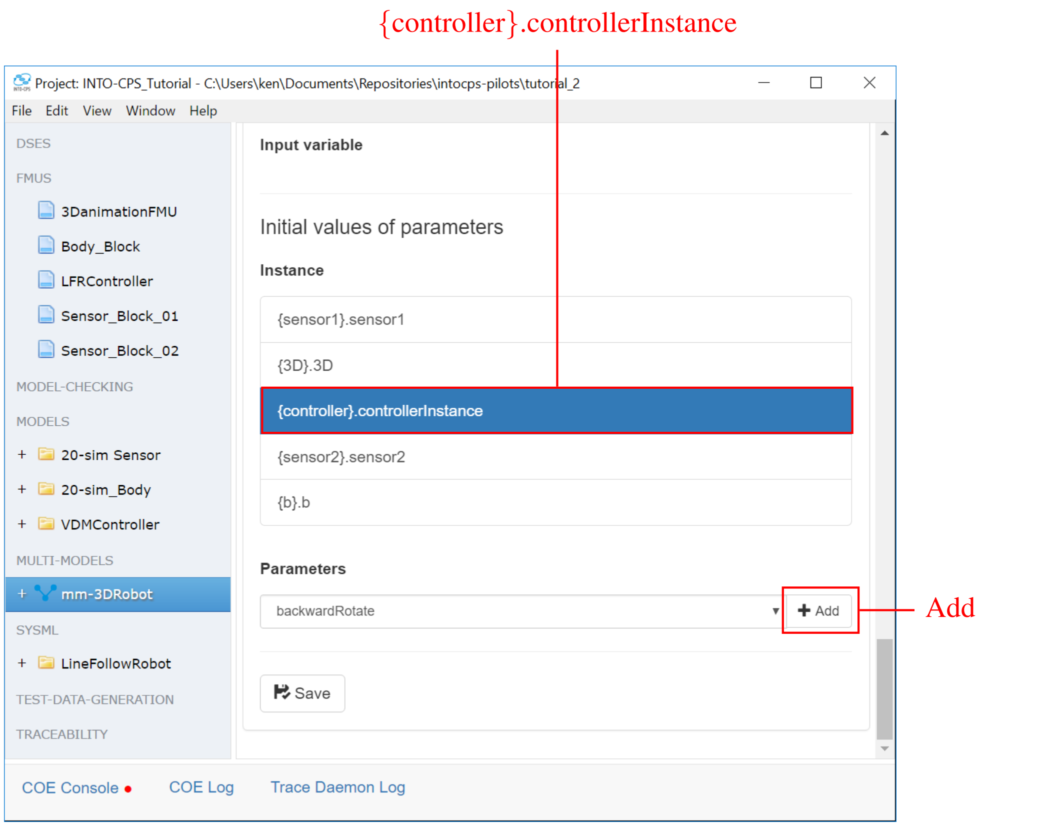
Input variable

☑ lfLeftVal
☐ lfRightVal
☐ total_energy_used

COE Console ● COE Log Trace Daemon Log

Step 19. Finally, repeat to connect {*sensor2*}.*sensor2* / *lf_1_sensor_reading* to {*controller*}.*controllerInstance* / *lfRightVal*.

Step 20. Next we must set the parameters of the controller, which determines how it responds to sensor input. Scroll down to *Initial values of parameters* and select {*controller*}.*controllerInstance*. Then click + *Add* for *backwardRotate*.



Project: INTO-CPS_Tutorial - C:\Users\ken\Documents\Repositories\intocps-pilots\tutorial_2
File Edit View Window Help

DSES
FMUS
3DanimationFMU
Body_Block
LFRController
Sensor_Block_01
Sensor_Block_02
MODEL-CHECKING
MODELS
+ 20-sim Sensor
+ 20-sim_Body
+ VDMController
MULTI-MODELS
+ mm-3DRobot
SYSML
+ LineFollowRobot
TEST-DATA-GENERATION
TRACEABILITY

Input variable

Initial values of parameters

Instance

{sensor1}.sensor1
{3D}.3D
{controller}.controllerInstance
{sensor2}.sensor2
{b}.b

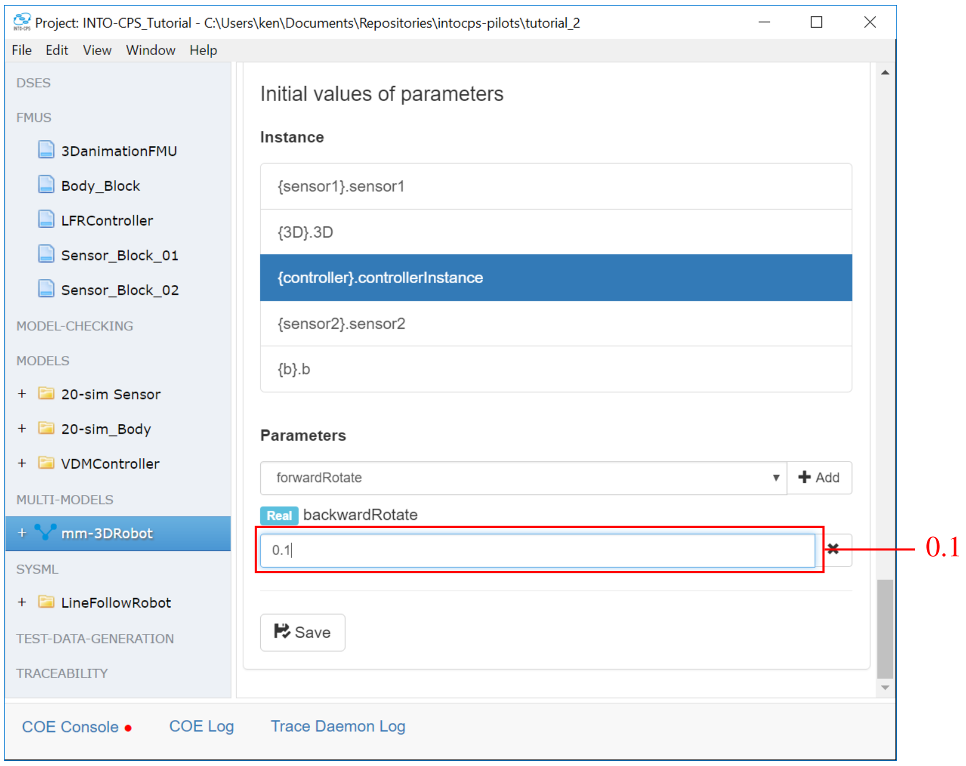
Parameters

backwardRotate + Add

Save

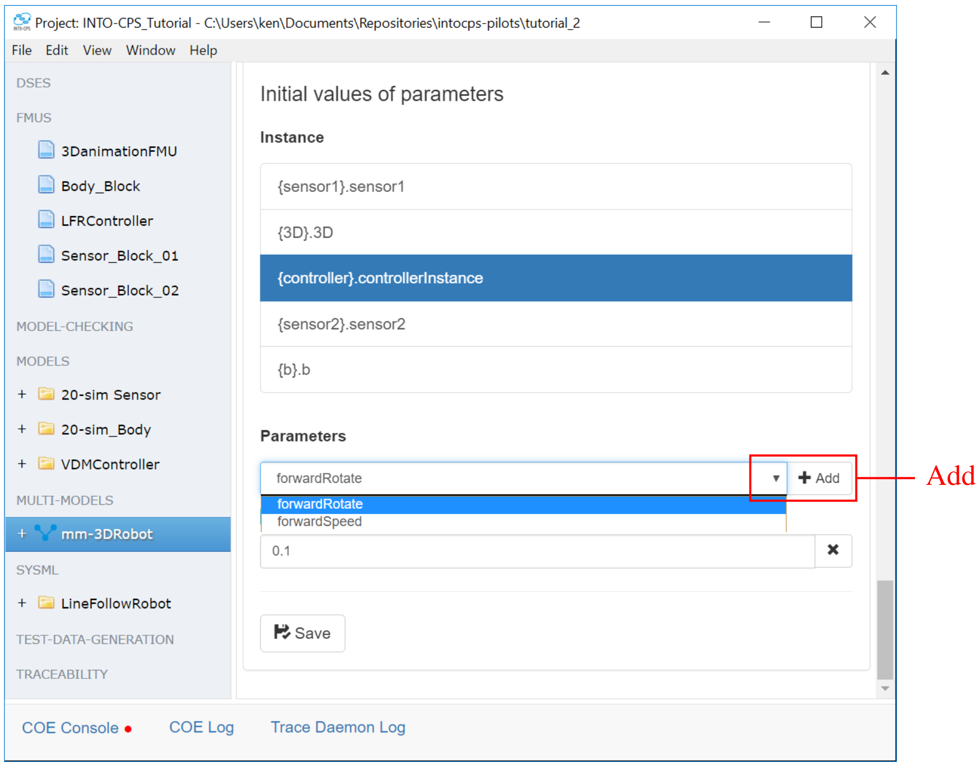
COE Console ● COE Log Trace Daemon Log

Step 21. Set *backwardRotate* to *0.1*.

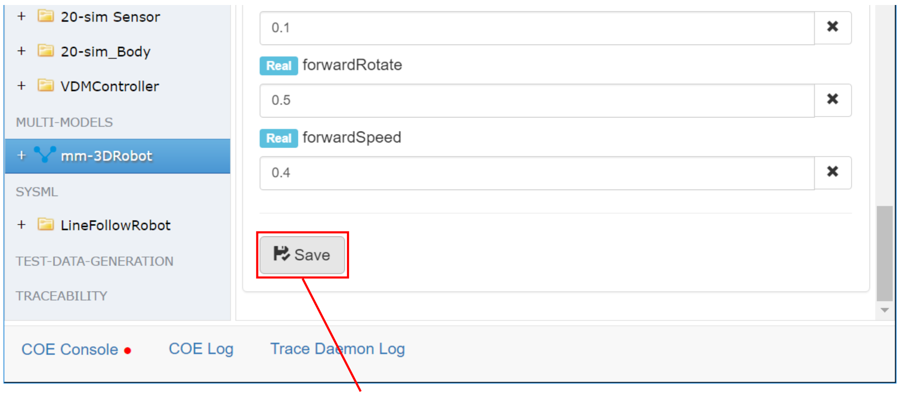


Step 22. Use the drop-down box and + *Add* button to add:

- *forwardRotate* with a value of *0.5*.
- *forwardSpeed* with a value of *0.4*.

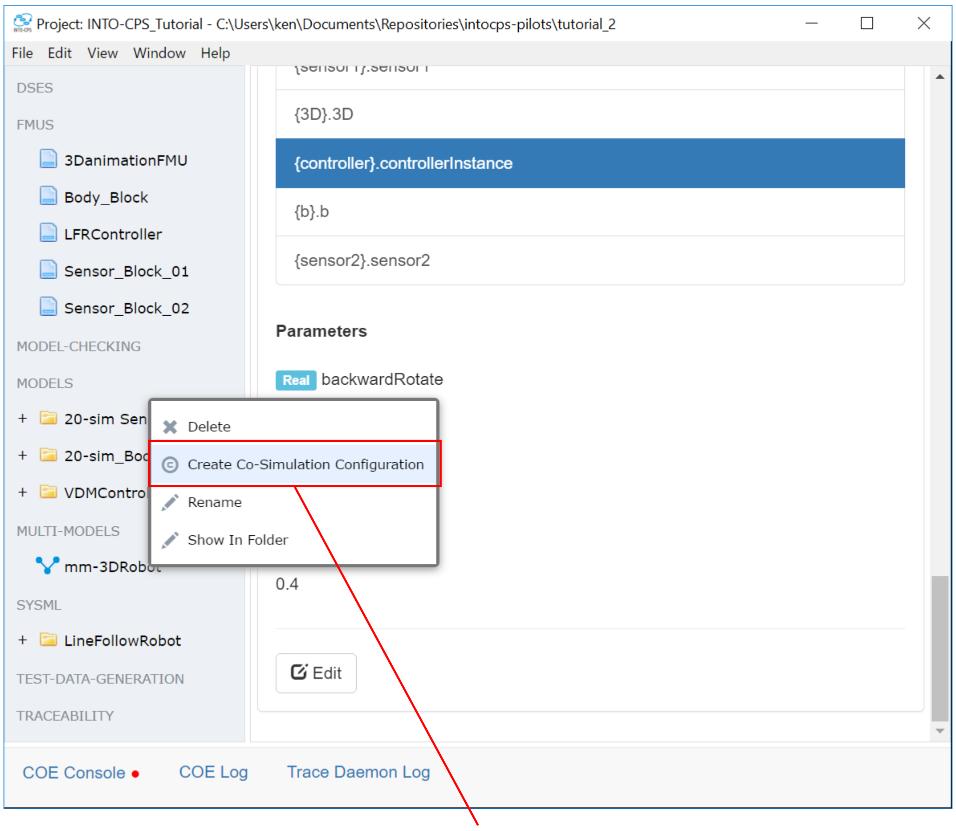


Step 23. *Save* the *Configuration*.



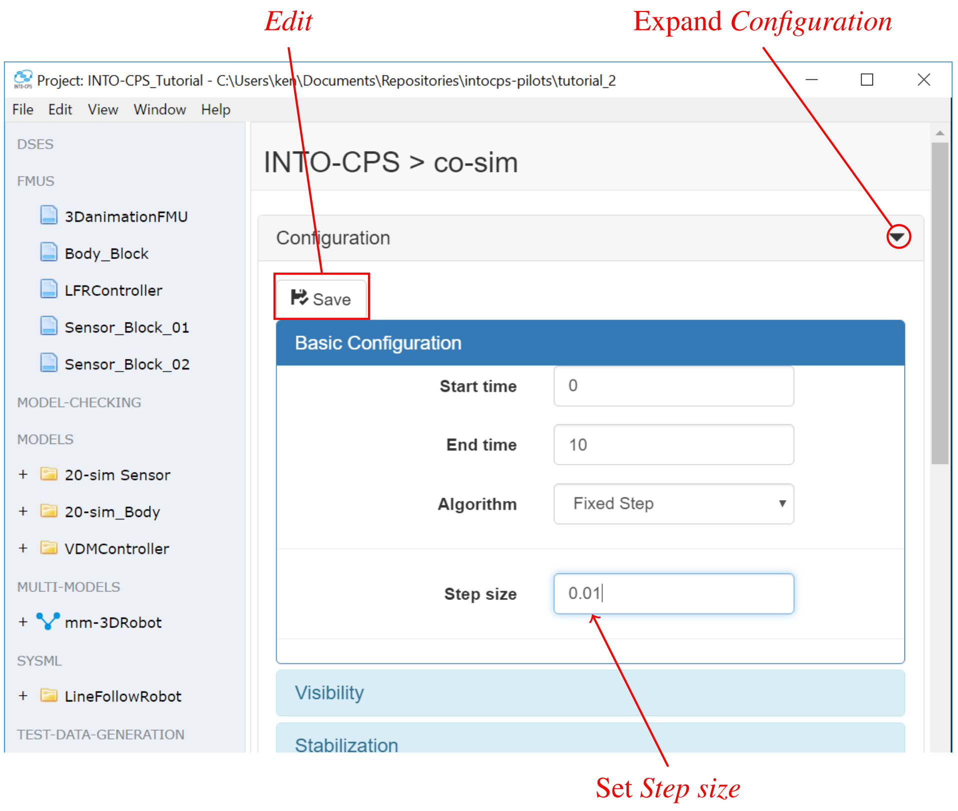
Add

Step 24. The multi-model configuration is complete. Right-click on the multi-model configuration and select *Create Co-simulation Configuration*. You can use the default name, *co-sim*, or choose your own name.



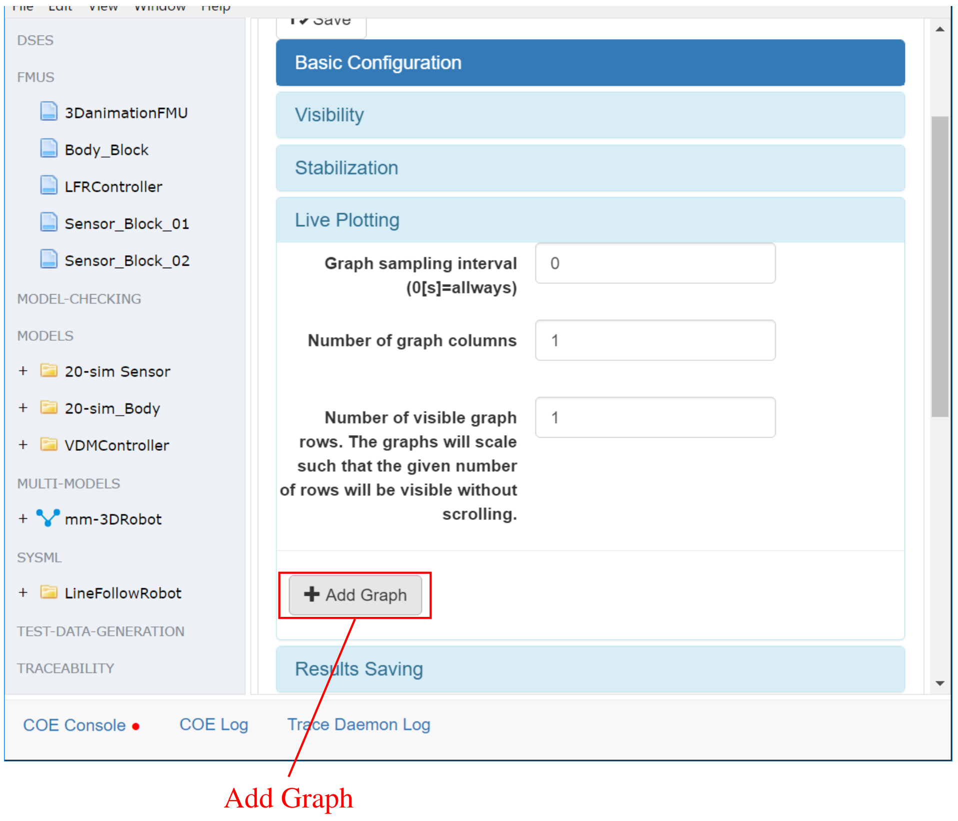
Create Co-Simulation Configuration

Step 25. Under *Basic Configuration* set the the *Step size* to 0.01. Don't forget to press *Edit*.



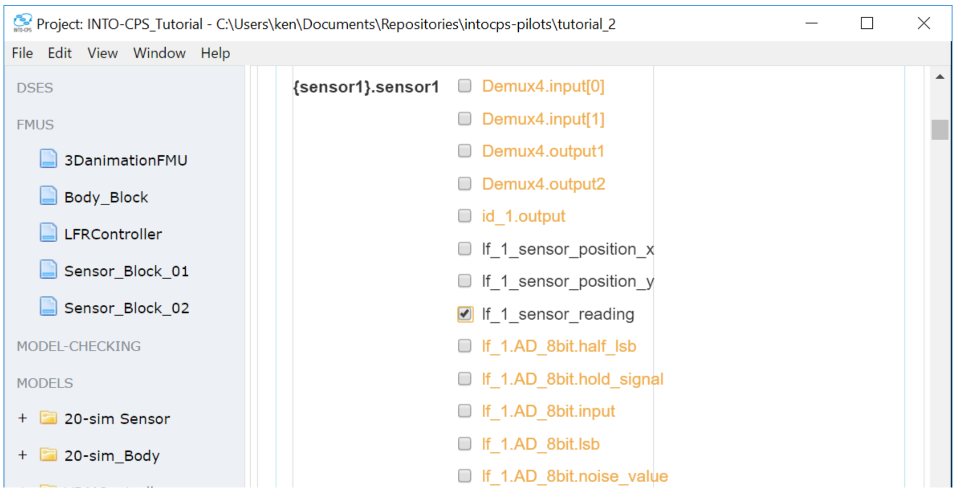
Start time: 0
End time: 10
Algorithm: Fixed Step
Step size: 0.01

Step 26. Under *Live Plotting* click *Add Graph*.



Graph sampling interval (0[s]=allways): 0
Number of graph columns: 1
Number of visible graph rows. The graphs will scale such that the given number of rows will be visible without scrolling.: 1

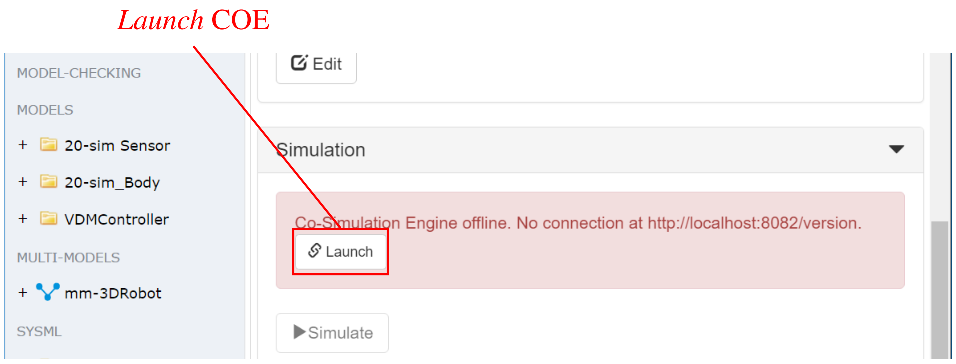
Step 27. Check *lf_1_sensor_reading* from {*sensor1*}.*sensor1* and {*sensor2*}.*sensor2* to see the sensor values appear in the *Livestream Configuration*.



{sensor1}.sensor1
- ☐ Demux4.input[0]
- ☐ Demux4.input[1]
- ☐ Demux4.output1
- ☐ Demux4.output2
- ☐ id_1.output
- ☐ lf_1_sensor_position_x
- ☐ lf_1_sensor_position_y
- ☑ lf_1_sensor_reading
- ☐ lf_1.AD_8bit.half_lsb
- ☐ lf_1.AD_8bit.hold_signal
- ☐ lf_1.AD_8bit.input
- ☐ lf_1.AD_8bit.lsb
- ☐ lf_1.AD_8bit.noise_value

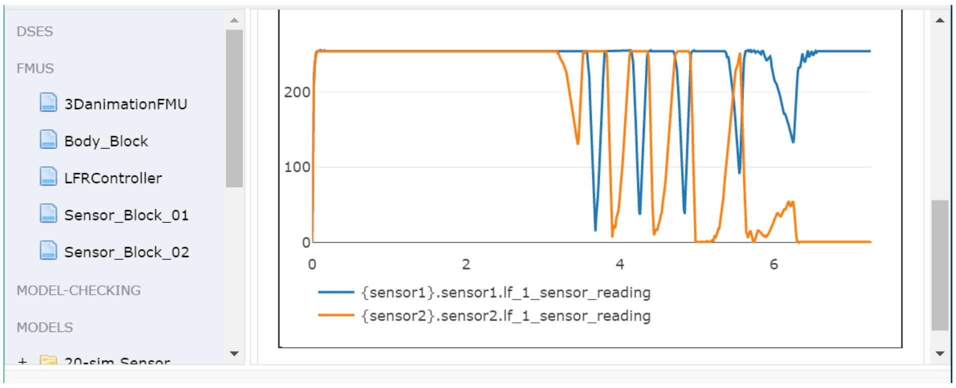
3 Running a Co-simulation

Step 28. Launch the COE if necessary (see *Tutorial 1 — First Co-simulation* for a reminder if needed).

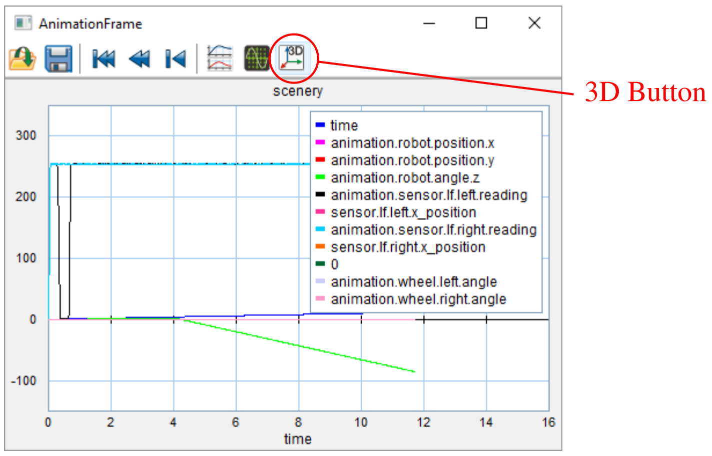


Co-Simulation Engine offline. No connection at http://localhost:8082/version.

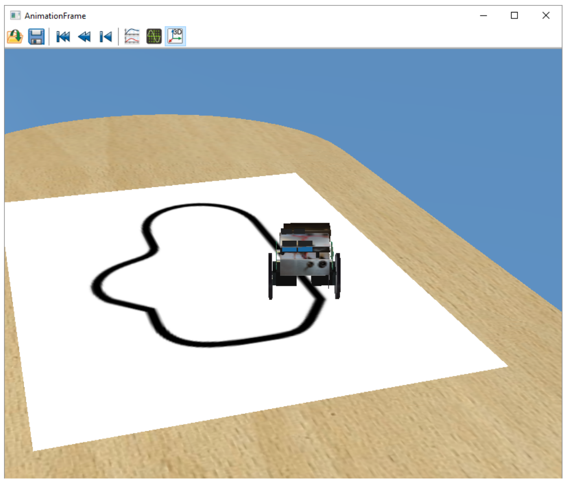
Step 29. When the COE is running (see *Tutorial 1* for more details if you need a reminder), click the *Simulate* button. The graph should show blue and orange lines, and if these should move up and down, indicating that the robot is sweeping left and right, following the line.



Step 30. **Windows only:** During co-simulation, a Java window called *Animation Frame* will appear like the one below. It shows a plot of variables from the co-simulation. You can click the *3D* button to see the 3D visualisation of the robot.


AnimationFrame
3D Button
scenery
time
animation.robot.position.x
animation.robot.position.y
animation.robot.angle.z
animation.sensor.lf.left.reading
sensor.lf.left.x_position
animation.sensor.lf.right.reading
sensor.lf.right.x_position
0
animation.wheel.left.angle
animation.wheel.right.angle
300
200
100
0
-100
0
2
4
6
8
10
12
14
16
time

Step 31. **Windows only:** A 3D model of the line following robot will appear. This view may be changed by clicking and dragging the mouse (note this is currently quite sensitive, so dont make quick movements). When the simulation has finished, this window will close. If everything went well, the robot should follow the line.



4 Additional Exercises

When this tutorial is complete, either move onto Tutorial 3, or try the following additional exercises:

1. Experiment with different sensor positions. Repeat steps 13 and 14 to change the position of the left and right sensors relative the the robot body. How do the different values effect the simulation?

2. Experiment with different robot speeds. Repeat steps 13 and 14 to change the different speed values for the *Controller*. How do the different values effect the simulation?

Tutorial 3 — Using SysML

Overview

This second tutorial will show you how to:

1. Add a new FMU in a SysML model
2. Generate a new multi-model configuration
3. Associate an FMU with a multi-model configuration
4. Execute a co-simulation using the new multi-model configuration

Requirements

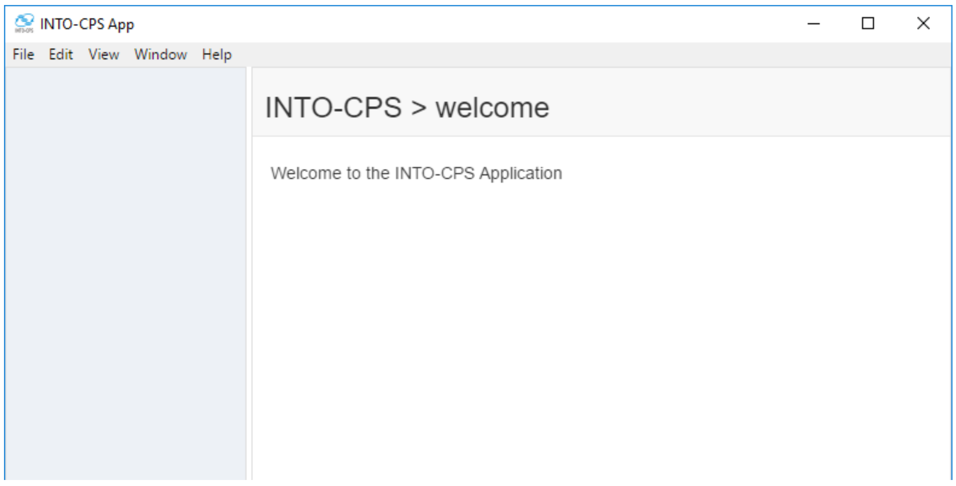
This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- INTO-CPS Application
- COE (Co-simulation Orchestration Engine) accessible to the Application
- Modelio v3.4.1

You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window > Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

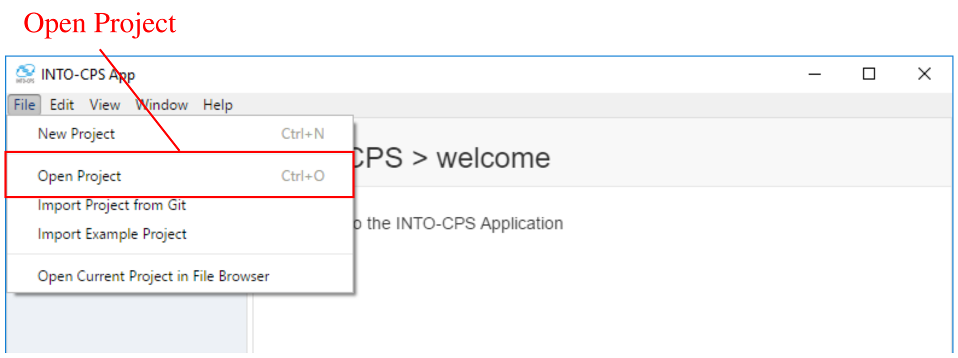
1 Opening a Project

Step 1. Launch the *INTO-CPS Application*. On first loading, it will look like the screenshot below. If you have opened a project previously, that project will be opened automatically.

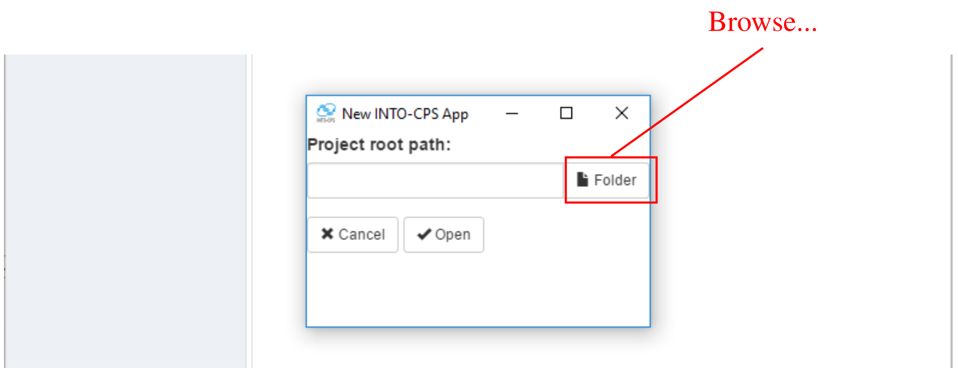


Welcome to the INTO-CPS Application

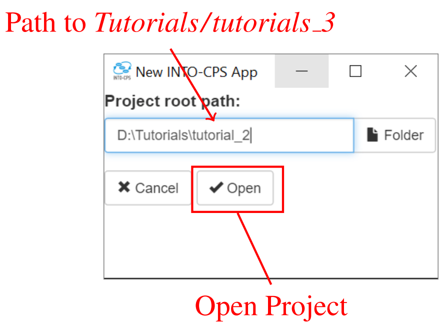
Step 2. To open a project, select *File* > *Open Project*.



Step 3. Set the *Project root path* to the location of *Tutorials/tutorials_3*. You can browse using the *Folder* button.



Step 4. Click *Open Project*.



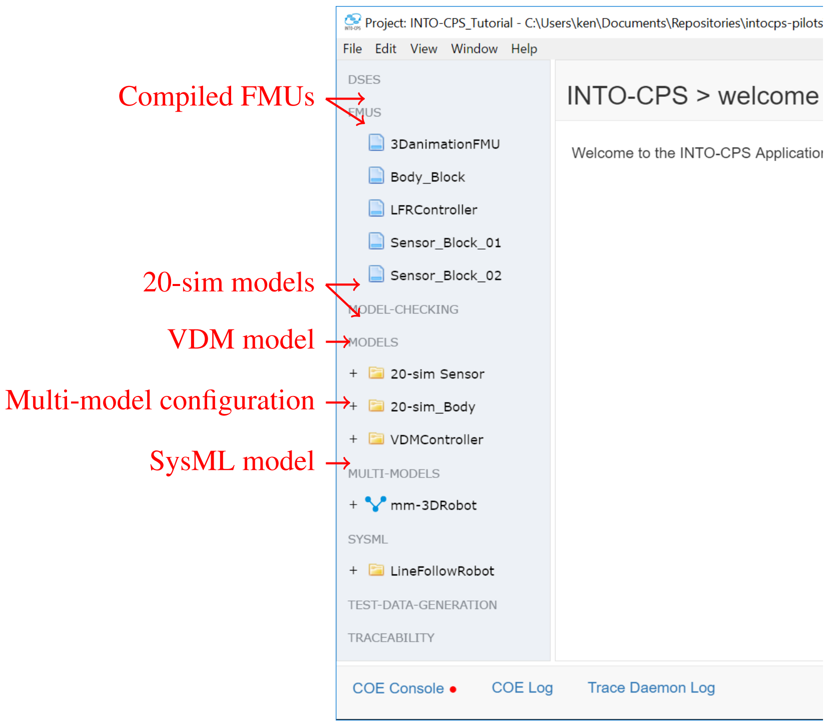
Step 5. Once the project is opened, you will see that project browser on the left of the INTO-CPS Application window is now populated. The entries in the project browser correspond to folders and files in the *Tutorials/tutorials_3* folder. These are:

FMUs Compiled FMUs (with file extension .fmu) that are used in co-simulation.

Models Source models used to generate the FMUs. The icon of each entry shows which tool created the model. In this case Overture and 20-sim.

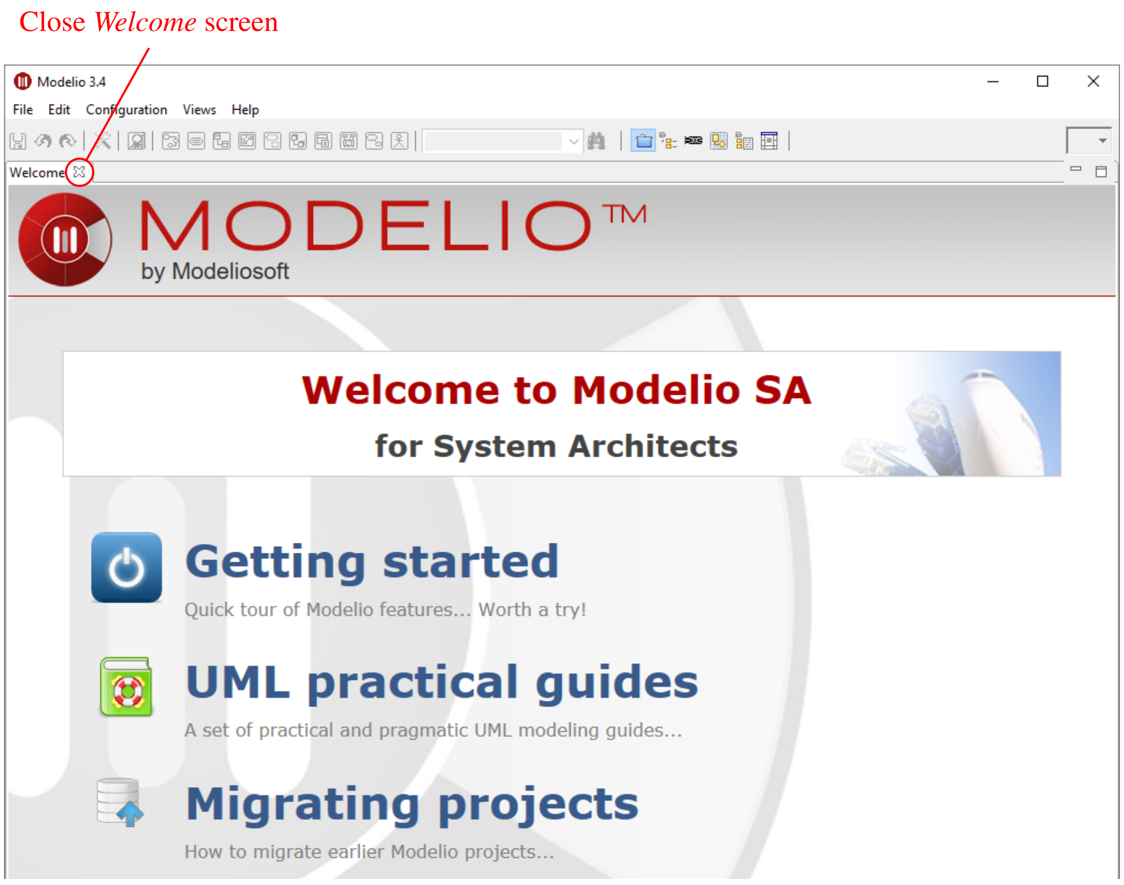
Multi-models Used to configure co-simulations, including which FMUs are used and other co-simulation settings.

SysML Architectural models that are used to create model and multi-model descriptions.

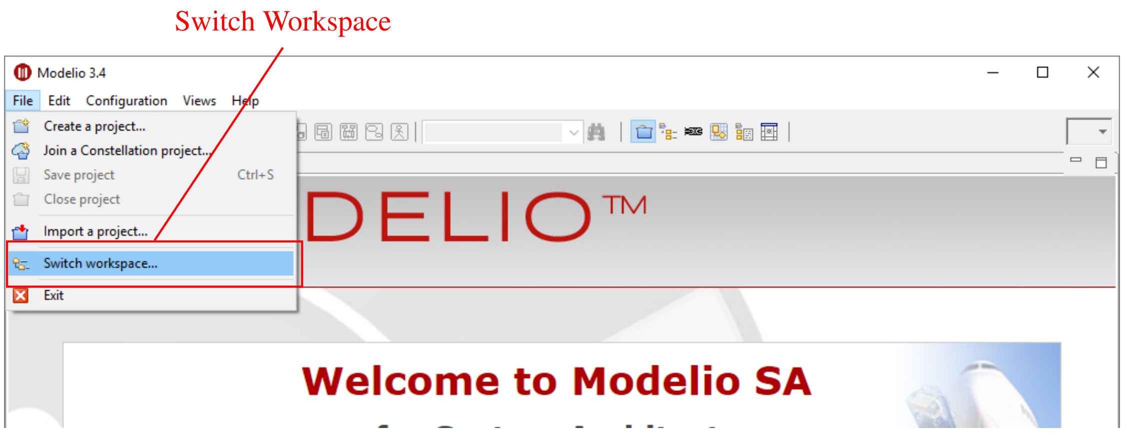


2 Edit Architecture

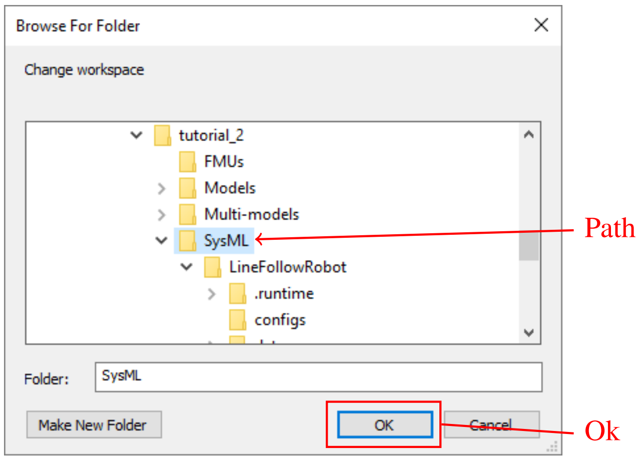
Step 6. Launch *Modelio*. On first loading, you may have to close the *Welcome* screen (you can bring it back with *Help* > *Welcome* if you need)


Close *Welcome* screen
Modelio 3.4
File Edit Configuration Views Help
Welcome
MODELIO™
by Modeliosoft
Welcome to Modelio SA
for System Architects
Getting started
Quick tour of Modelio features... Worth a try!
UML practical guides
A set of practical and pragmatic UML modeling guides...
Migrating projects
How to migrate earlier Modelio projects...

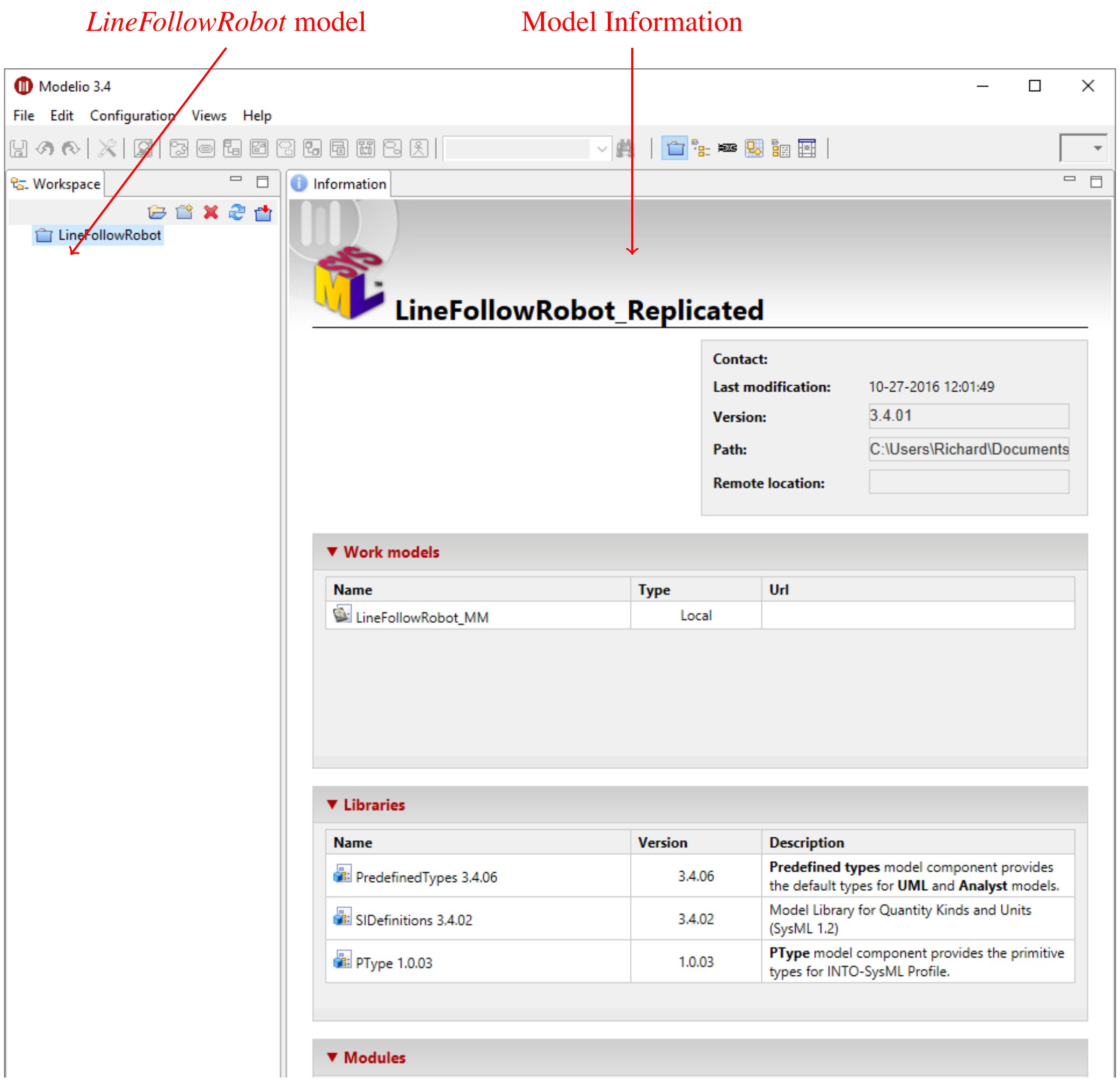
Step 7. A workspace must be chosen, select *File* > *Switch Workspace*.



Step 8. Set the *Workspace* to the location of *Tutorials/tutorials_3/SysML* and click *Ok*.



Step 9. Left-click on the *LineFollowRobot* model once on the left to see details of the model. Double-click the *LineFollowRobot* model to open the model.



Contact:
Last modification: 10-27-2016 12:01:49
Version: 3.4.01
Path: C:\Users\Richard\Documents
Remote location:

Work models

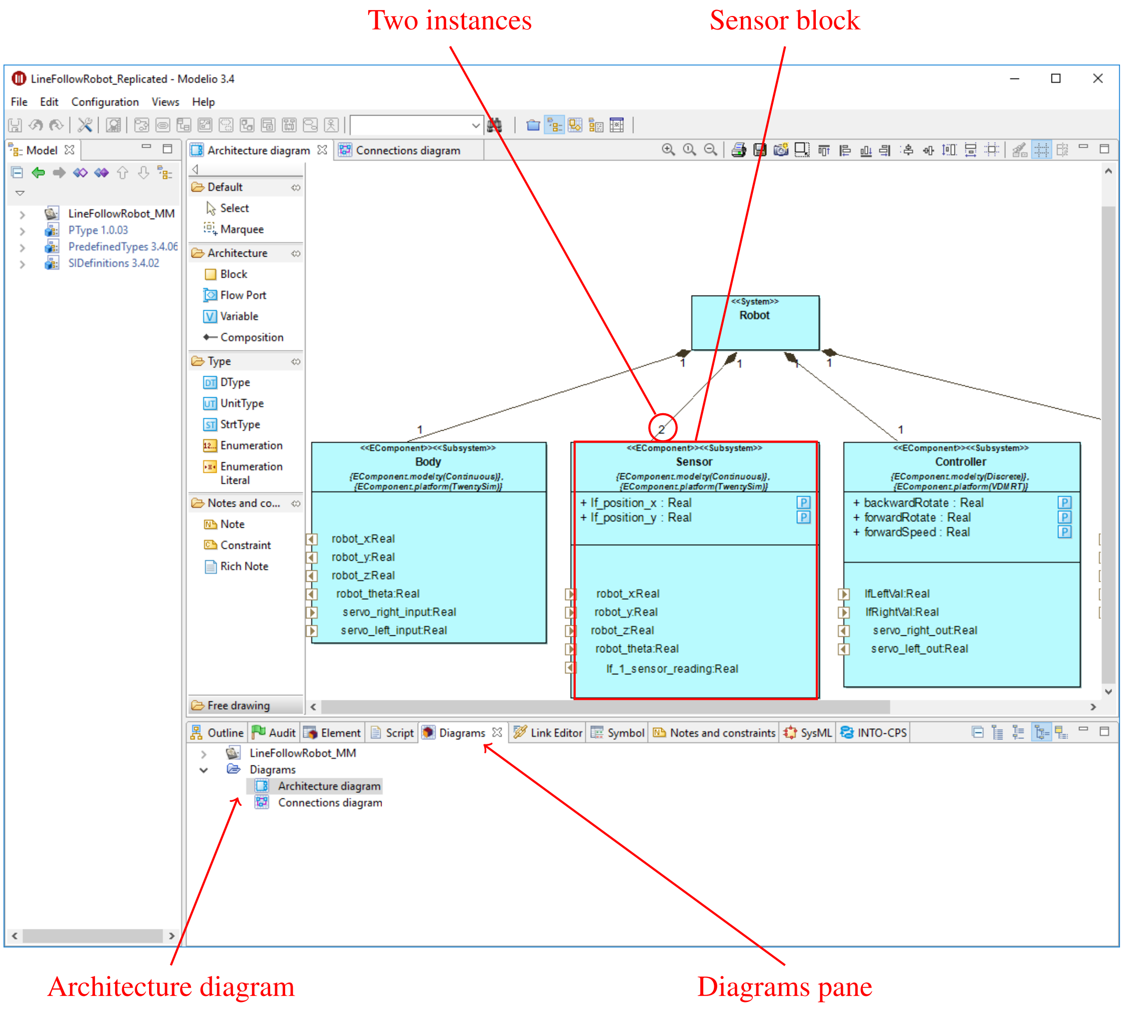
Name	Type	Url
LineFollowRobot_MM	Local	

Libraries

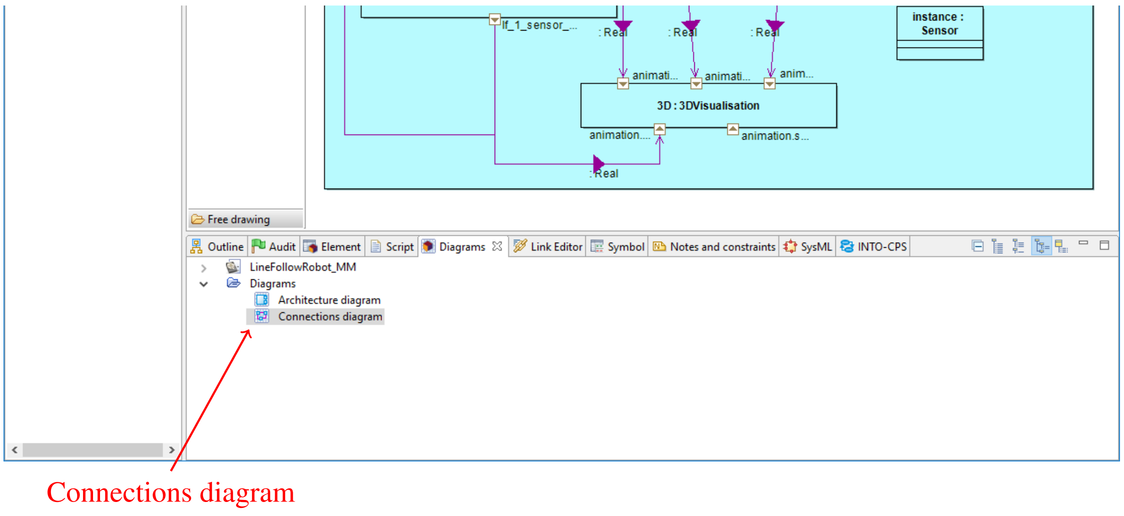
Name	Version	Description
PredefinedTypes 3.4.06	3.4.06	**Predefined types** model component provides the default types for **UML** and **Analyst** models.
SIDefinitions 3.4.02	3.4.02	Model Library for Quantity Kinds and Units (SysML 1.2)
PType 1.0.03	1.0.03	**PType** model component provides the primitive types for INTO-SysML Profile.

Modules

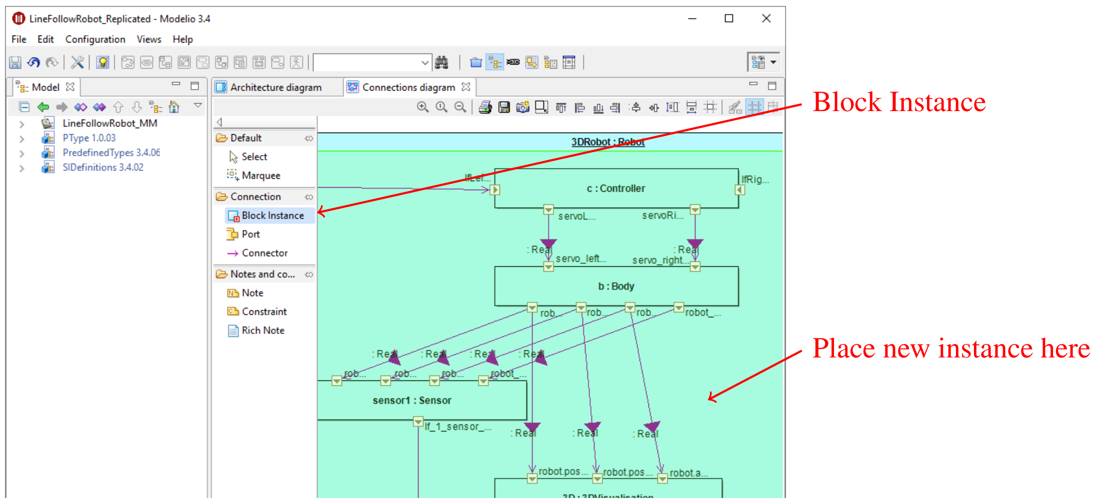
Step 10. In the *Diagrams* pane, expand the *Diagrams* folder and double click the *Architecture Diagram*. The diagram below will open. Notice there are two instances of the Sensor model.



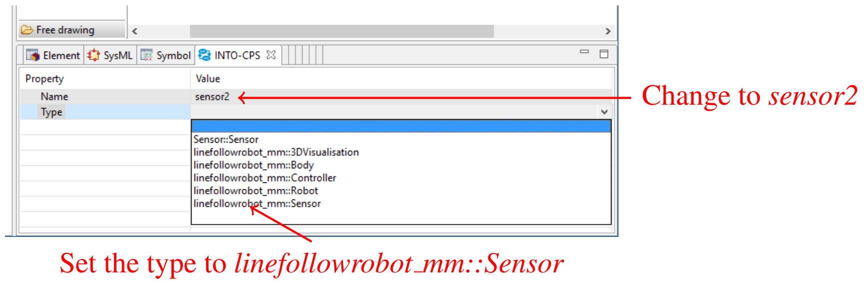
Step 11. Double click the *Connections Diagram*



Step 12. To add a new Sensor, select *Block Instance* from the palette menu and add the new instance to the 3DRobot – simply click inside the 3DRobot, as indicated below.



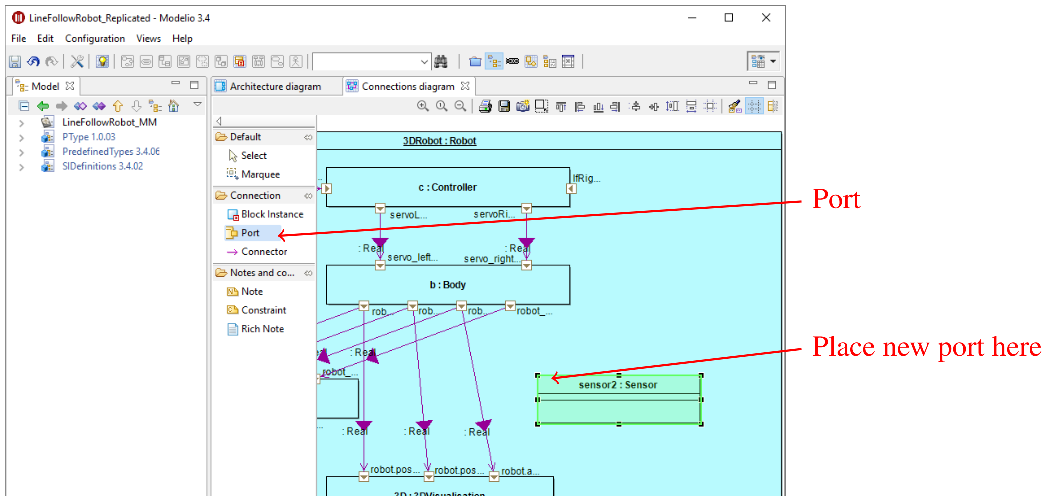
Step 13. In the *INTO-CPS* panel, change the name of the new instance to ‘sensor2’ and set the type to be ‘linefollowrobot_mm::Sensor’.



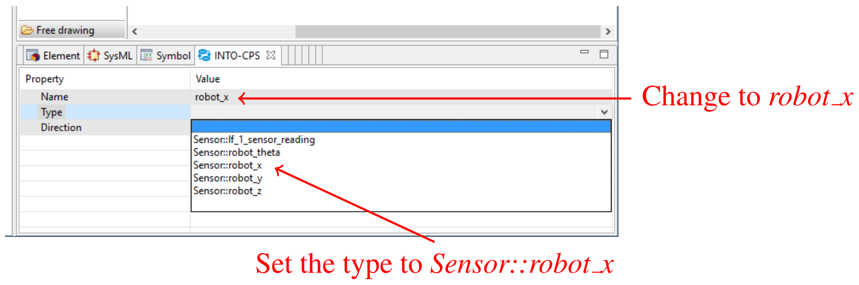
Property	Value
Name	sensor2
Type	

```
Sensor::Sensor
linefollowrobot_mm::3DVisualisation
linefollowrobot_mm::Body
linefollowrobot_mm::Controller
linefollowrobot_mm::Robot
linefollowrobot_mm::Sensor
```

Step 14. The next step is to add ports to the sensor instance. Select *Port* from the palette menu and add the new port to the sensor2.



Step 15. Select the new port and in the *INTO-CPS* panel change the name to 'robot_x' and type to be 'Sensor::robot_x'.



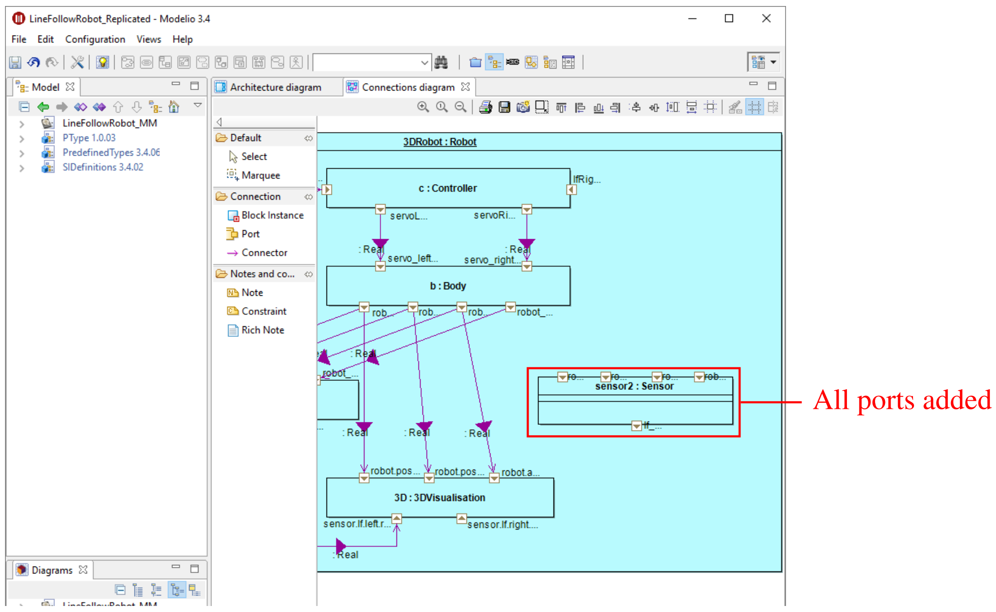
Property	Value
Name	robot_x
Type	
Direction	

Sensor::lf_1_sensor_reading
Sensor::robot_theta
Sensor::robot_x
Sensor::robot_y
Sensor::robot_z

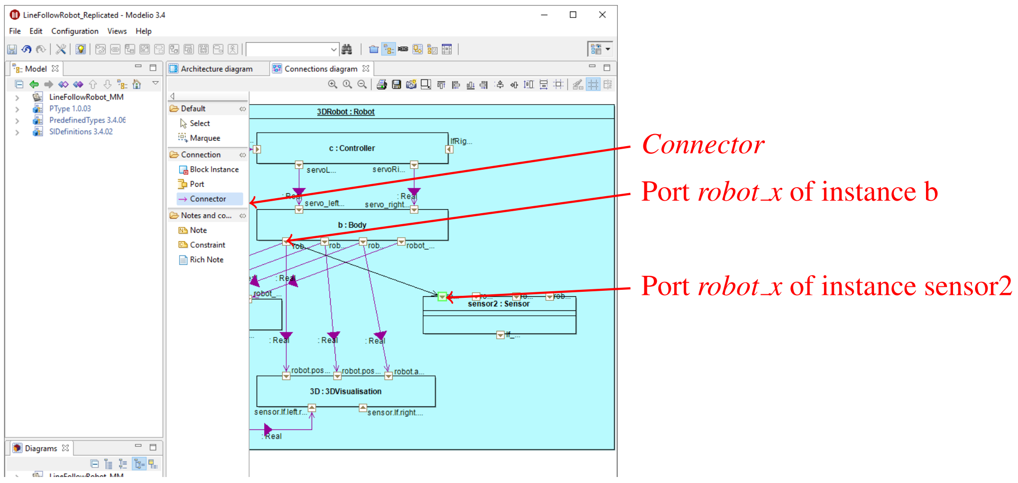
Step 16. Repeat steps 14 and 15 to add four more ports:

- Name: 'robot_y'; Type: 'Sensor::robot_y'.
- Name: 'robot_z'; Type: 'Sensor::robot_z'.
- Name: 'robot_theta'; Type: 'Sensor::robot_theta'.
- Name: 'lf_1_sensor_reading'; Type: 'Sensor::lf_1_sensor_reading'.

The connections diagram should look like that below:



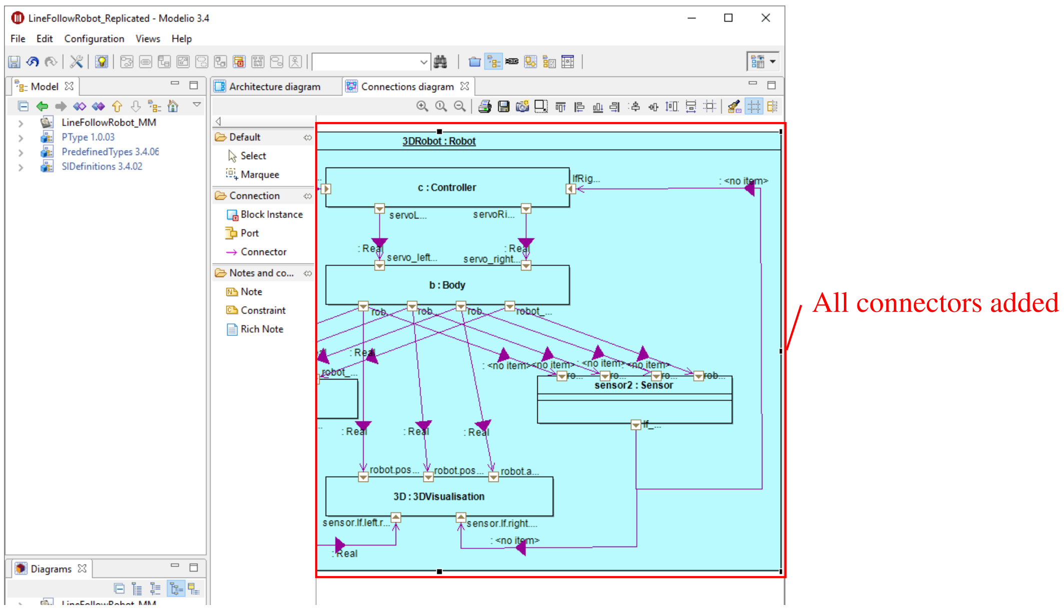
Step 17. The next step is to add connections between the different models of the robot. Select *Connector* from the palette menu and connect the *robot_x* port of the *body* component to the *robot_x* port of the new *sensor2* component



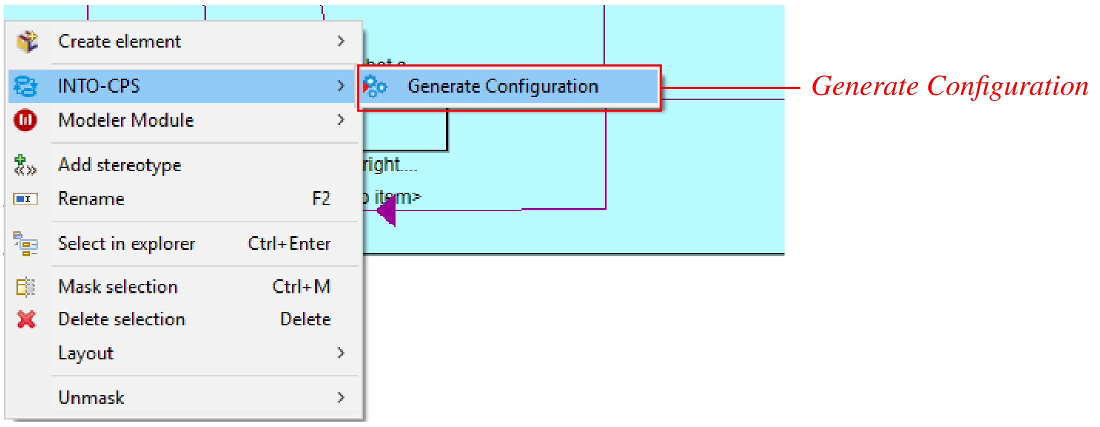
Step 18. Repeat step 17 to add five more connectors:

- 'body.robot_y' to 'sensor2.robot_y'.
- 'body.robot_z' to 'sensor2.robot_z'.
- 'body.robot_theta' to 'sensor2.robot_theta'.
- 'sensor2.lf_1_sensor_reading' to 'controller.lfRightVal'.
- 'sensor2.lf_1_sensor_reading' to '3D.animation.sensor.lf.right'.

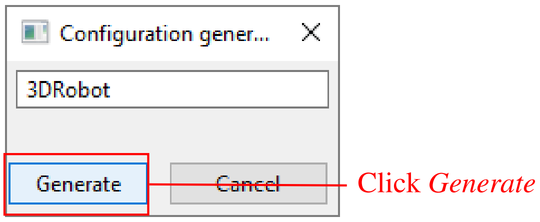
The connections diagram should look like that below:



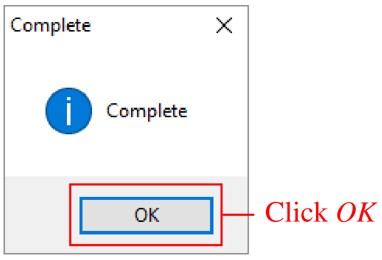
Step 19. To export this new configuration, right click on the 3DRobot instance and select *INTO-CPS > Generate Configuration. If nothing happens, closing and re-opening Modelio often helps.*



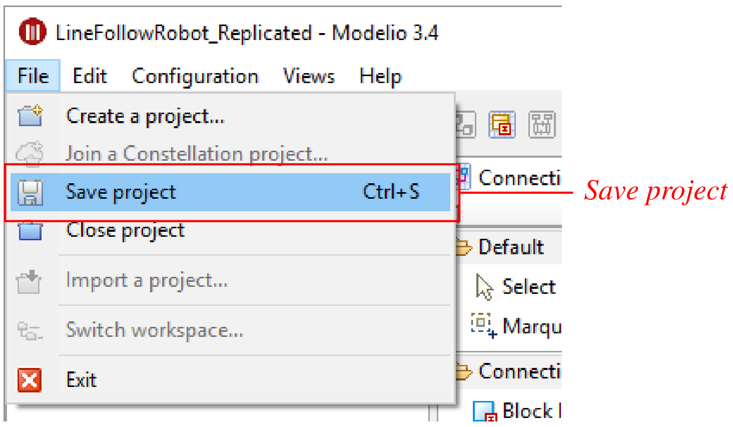
Step 20. Click *Generate*. If this seems to be unresponsive, then click *Cancel*, save the model, close and reopen *Modelio* and try again.



Step 21. Click *OK*.

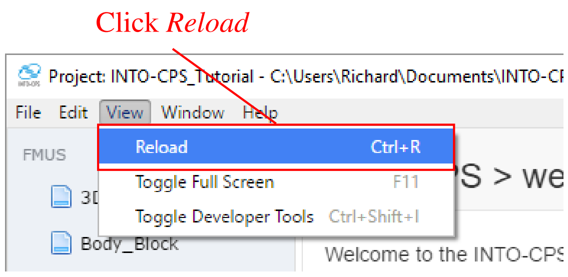


Step 22. Finally, save the SysML model.

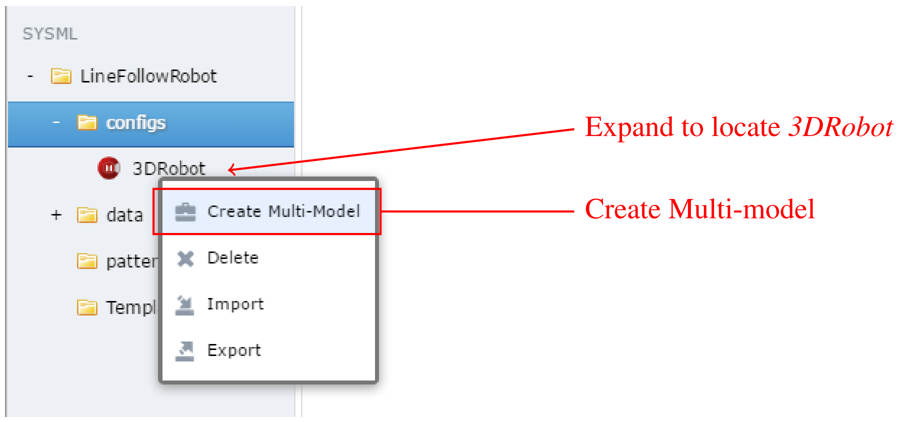


3 Configuring a Multi-model

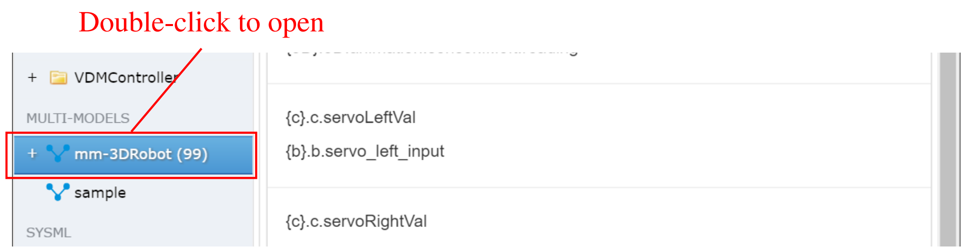
Step 23. Return to the *INTO-CPS Application* and reload the view by selecting *View* > *Reload*.


Click *Reload*
Project: INTO-CPS_Tutorial - C:\Users\Richard\Documents\INTO-CI
File Edit View Window Help
FMUS
Reload Ctrl+R
Toggle Full Screen F11
Toggle Developer Tools Ctrl+Shift+I
Body_Block
S > we
Welcome to the INTO-CPS

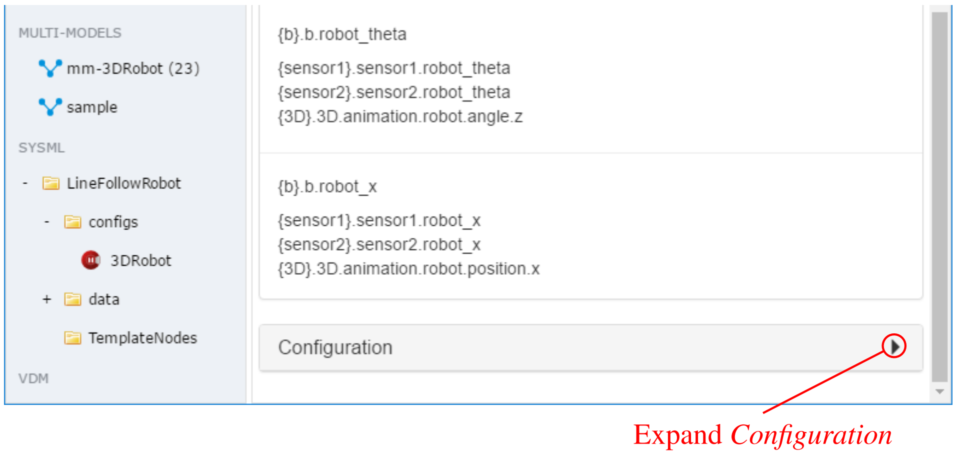
Step 24. In the SysML entry of the project browser, expand the *LineFollowRobot* and then *config* folders. There should be a *3DRobot* icon (as in the Figure below). Right click on *3DRobot*, select *Create Multi-Model*. You can just accept the default name in the prompt that appears.



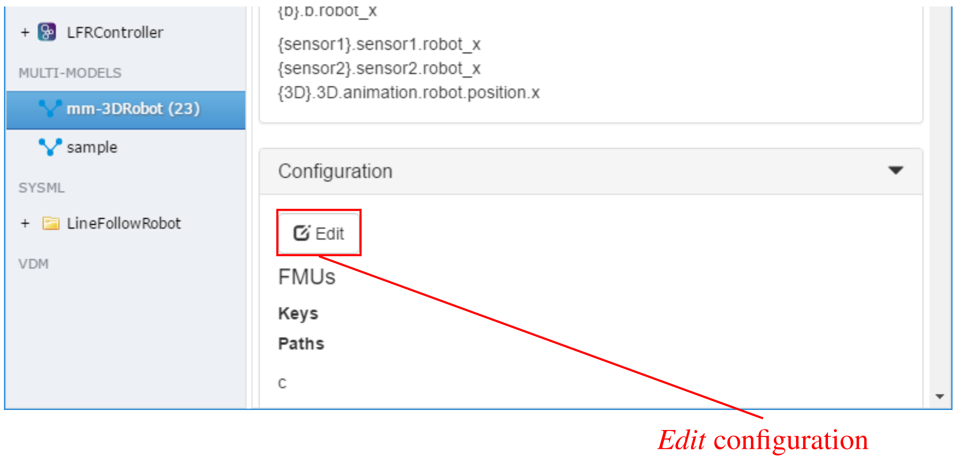
Step 25. A new multi-model configuration has been created and is shown in the multi-model entry of the project browser. Double-click on the new multi-model to open it.



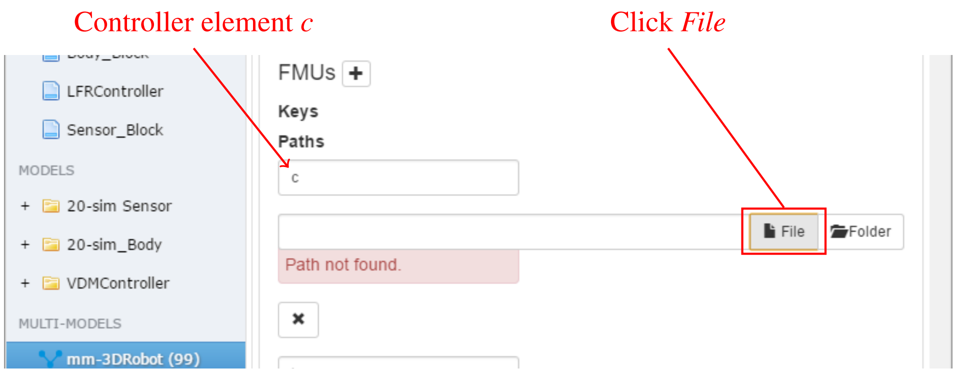
Step 26. We need to associate FMUs with this multi-model and set its parameters. Expand the *Configuration* section of the multi-model by clicking on the triangle.


MULTI-MODELS
mm-3DRobot (23)
sample
SYSML
LineFollowRobot
configs
3DRobot
data
TemplateNodes
VDM
{b}.b.robot_theta
{sensor1}.sensor1.robot_theta
{sensor2}.sensor2.robot_theta
{3D}.3D.animation.robot.angle.z
{b}.b.robot_x
{sensor1}.sensor1.robot_x
{sensor2}.sensor2.robot_x
{3D}.3D.animation.robot.position.x
Configuration
Expand *Configuration*

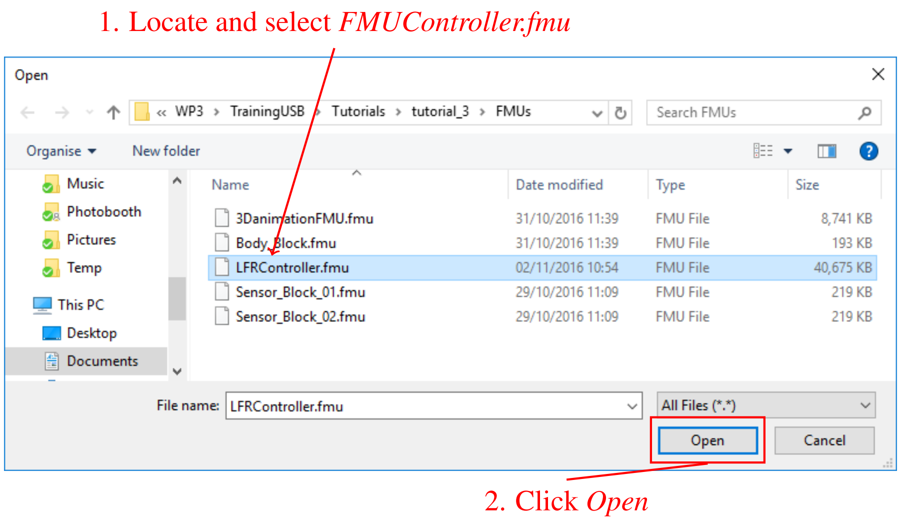
Step 27. Scroll down and click *Edit*.



Step 28. In the *FMUs* section, next to the *Controller* element *c*, click the *File* button.



Step 29. A file browser window will open and show five FMUs (if the file browser does not show the FMUs, navigate to *tutorials_3/FMUs*). Select *FMUController.fmu* and click *Open*.

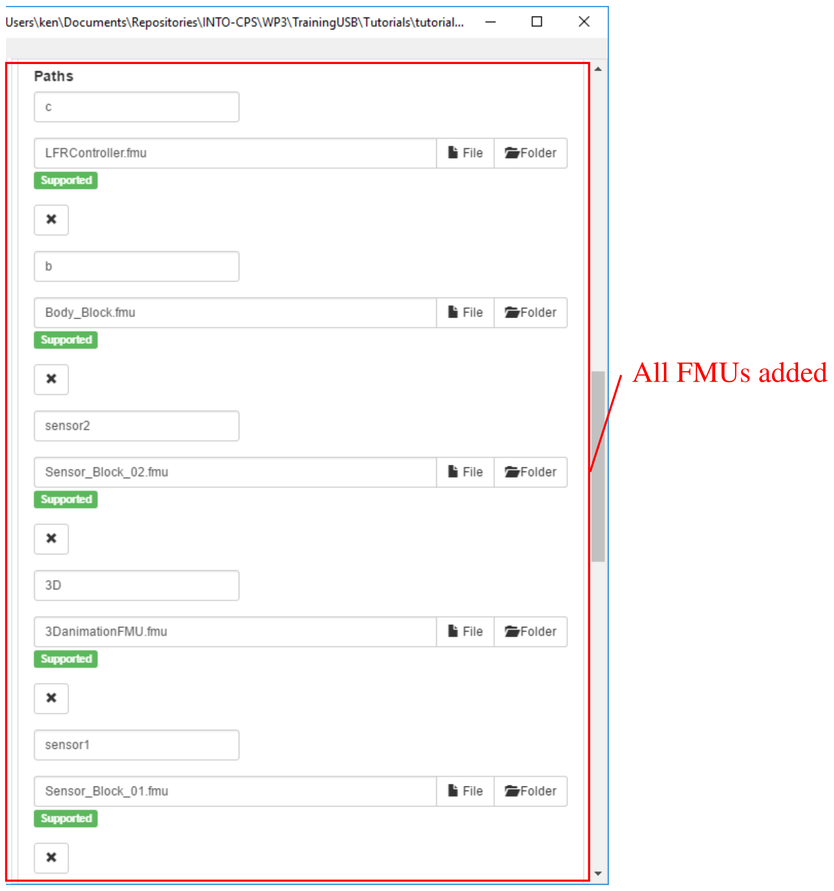


Name	Date modified	Type	Size
3DanimationFMU.fmu	31/10/2016 11:39	FMU File	8,741 KB
Body_Block.fmu	31/10/2016 11:39	FMU File	193 KB
LFRController.fmu	02/11/2016 10:54	FMU File	40,675 KB
Sensor_Block_01.fmu	29/10/2016 11:09	FMU File	219 KB
Sensor_Block_02.fmu	29/10/2016 11:09	FMU File	219 KB

File name: LFRController.fmu

Step 30. The LFRController has been added. Repeat this for the remaining elements:

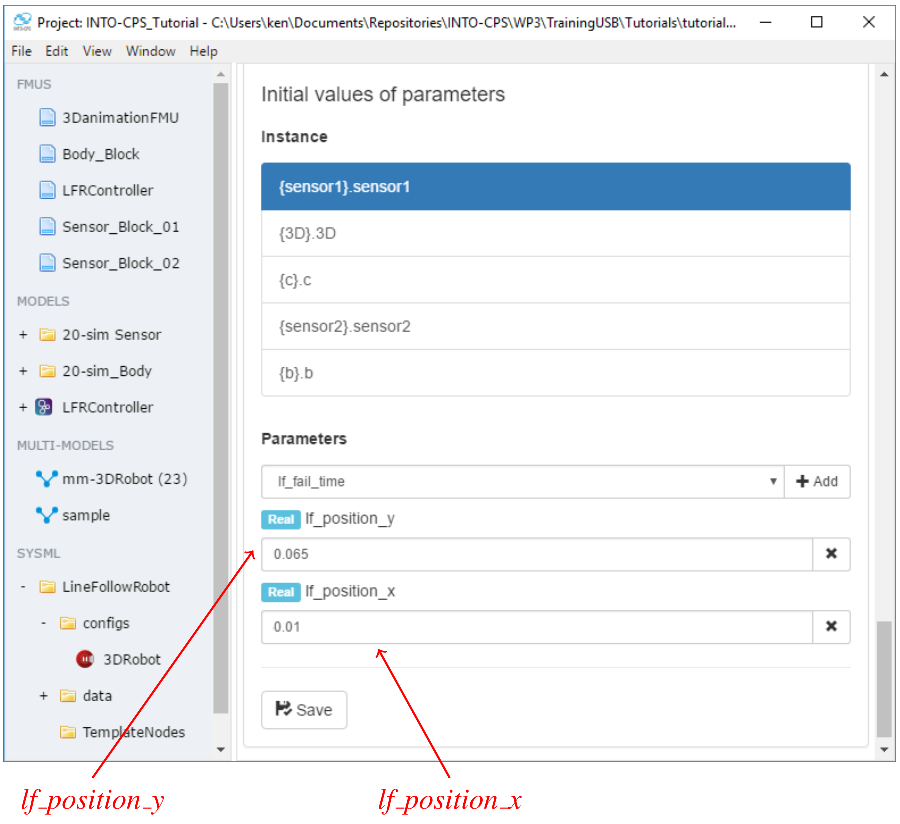
- *b* : *Body_Block.fmu*
- *3D* : *3DanimationFMU.fmu*
- *sensor1* : *Sensor_Block_01.fmu*
- *sensor2* : *Sensor_Block_02.fmu*


Users\ken\Documents\Repositories\INTO-CPS\WP3\TrainingUSB\Tutorials\tutorial...
Paths
File
Folder
Supported
All FMUs added

c	LFRController.fmu
b	Body_Block.fmu
sensor2	Sensor_Block_02.fmu
3D	3DanimationFMU.fmu
sensor1	Sensor_Block_01.fmu

Step 31. Next the sensor positions must be defined. Scroll down to the *Initial values of parameters section*, and click {*sensor1*}.*sensor1*. In the *Parameters* section, enter the following values:

- *lf_position_y* = 0.065
- *lf_position_x* = 0.01



Initial values of parameters

Instance

{sensor1}.sensor1
{3D}.3D
{c}.c
{sensor2}.sensor2
{b}.b

Parameters

lf_fail_time

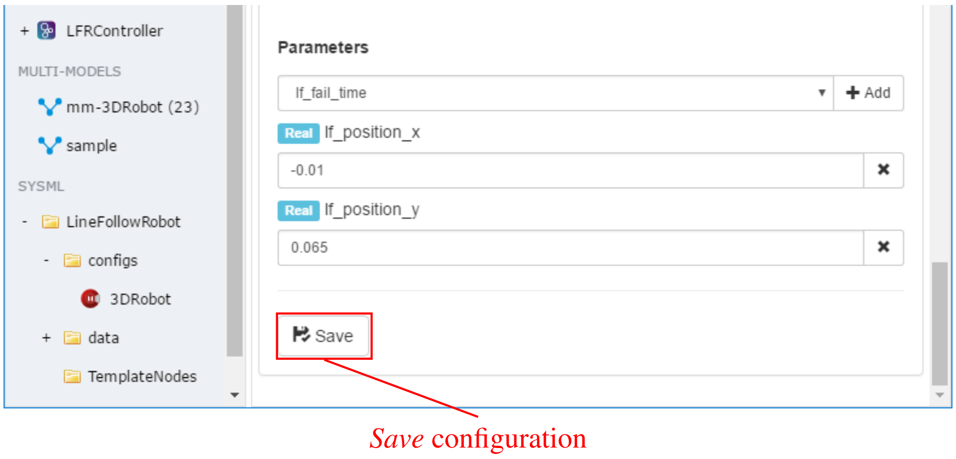
Real lf_position_y: 0.065

Real lf_position_x: 0.01

Step 32. Repeat the previous step for the second sensor – {*sensor2*}.*sensor2* with the following values:

- *lf_position_x* = -0.01
- *lf_position_y* = 0.065

Step 33. *Save* the *Configuration*.



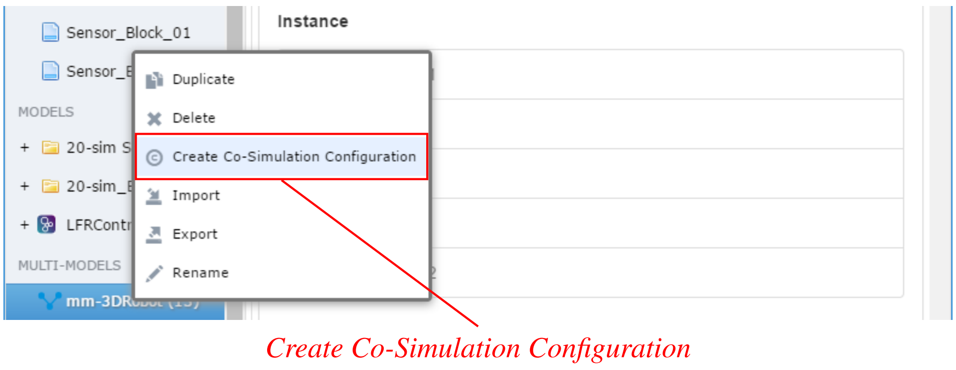
Parameters

lf_fail_time

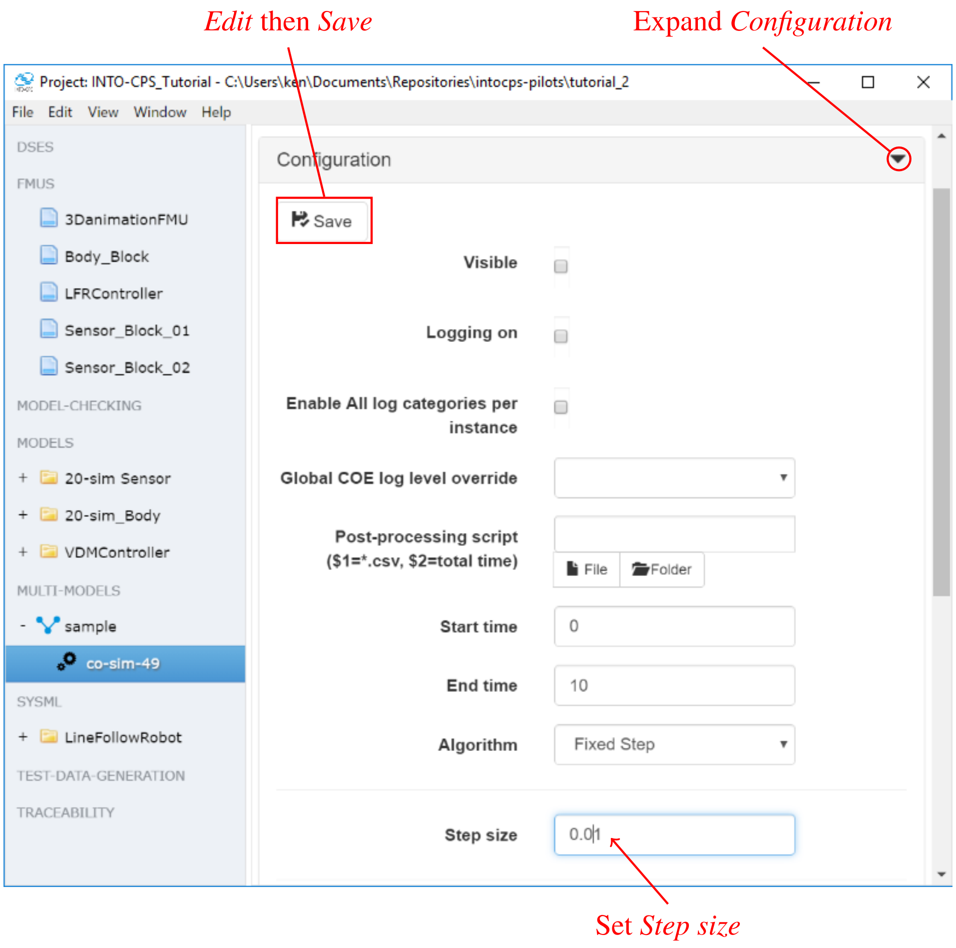
Real lf_position_x: -0.01

Real lf_position_y: 0.065

Step 34. The multi-model configuration is complete. Right-click on the multi-model configuration and select *Create Co-simulation Configuration*.



Step 35. Set the *Step size* to 0.01. Don't forget to press *Edit* then *Save*.



Visible ☐

Logging on ☐

Enable All log categories per instance ☐

Global COE log level override

Post-processing script ($1=*.csv, $2=total time)

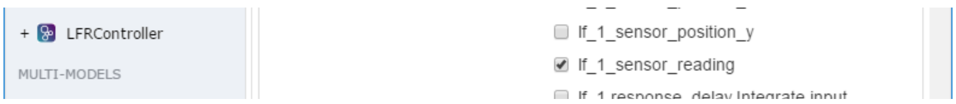
Start time 0

End time 10

Algorithm Fixed Step

Step size 0.01

Step 36. Check *lf_1_sensor_reading* from {*sensor1*}.*sensor1* and {*sensor2*}.*sensor2* to see the sensor values appear in the *Livestream Configuration*.



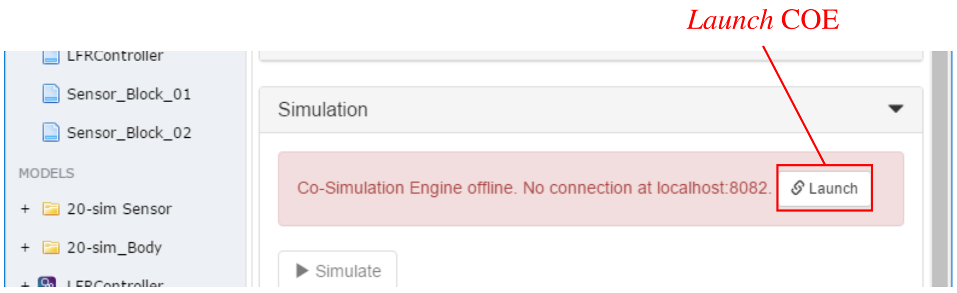
☐ lf_1_sensor_position_y

☑ lf_1_sensor_reading

☐ lf_1_response_delay.Integrate.input

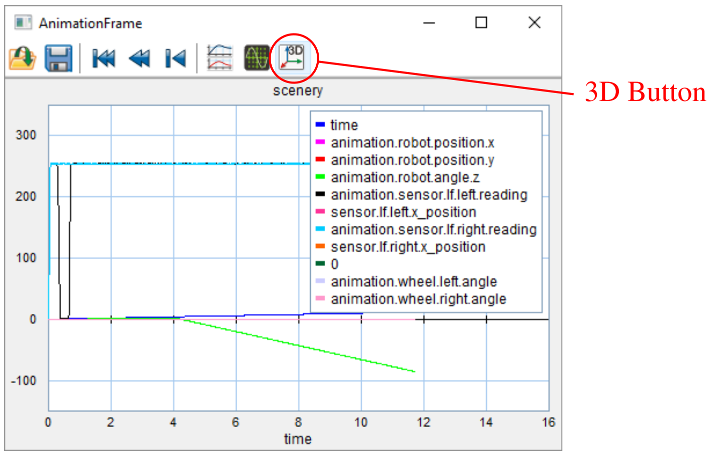
4 Running a Co-simulation

Step 37. Launch the COE if necessary (see *Tutorial 1 — First Co-simulation* for a reminder if needed).

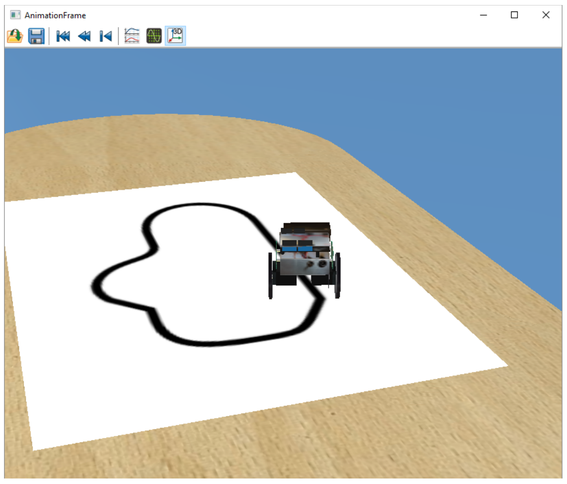


Co-Simulation Engine offline. No connection at localhost:8082.

Step 38. When the COE is running (see *Tutorial 1* for more details if you need a reminder), click the *Simulate* button. After a few seconds, a Java window called *Animation Frame* will appear like the one below. It shows a plot of variables from the co-simulation. You can click the *3D* button to see the 3D visualisation of the robot.


AnimationFrame
3D Button
scenery
time
animation.robot.position.x
animation.robot.position.y
animation.robot.angle.z
animation.sensor.lf.left.reading
sensor.lf.left.x_position
animation.sensor.lf.right.reading
sensor.lf.right.x_position
0
animation.wheel.left.angle
animation.wheel.right.angle
300
200
100
0
-100
0
2
4
6
8
10
12
14
16
time

Step 39. A 3D model of the line following robot will appear. This view may be changed by clicking and dragging the mouse (note this is currently quite sensitive, so dont make quick movements). When the simulation has finished, this window will close. If everything went well, the robot should follow the line.



5 Additional Exercises

When this tutorial is complete, either move onto Tutorial 3, or try the following additional exercises:

1. Experiment with different sensor positions. Repeat steps 13 and 14 to change the position of the left and right sensors relative the the robot body. How do the different values effect the simulation?
2. Experiment with different robot speeds. Repeat steps 13 and 14 to change the different speed values for the *Controller*. How do the different values effect the simulation?

Tutorial 4 — FMU Export (Overture)

Overview

This third INTO-CPS tutorial will show you how to:

1. Generate a new controller FMU in Overture
   (a) Import a model description into Overture
   (b) Complete the skeleton model to produce a working controller
   (c) Export the controller FMU
2. Associate the new controller FMU with a multi-model configuration
3. Execute a co-simulation using the new controller

Requirements

This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

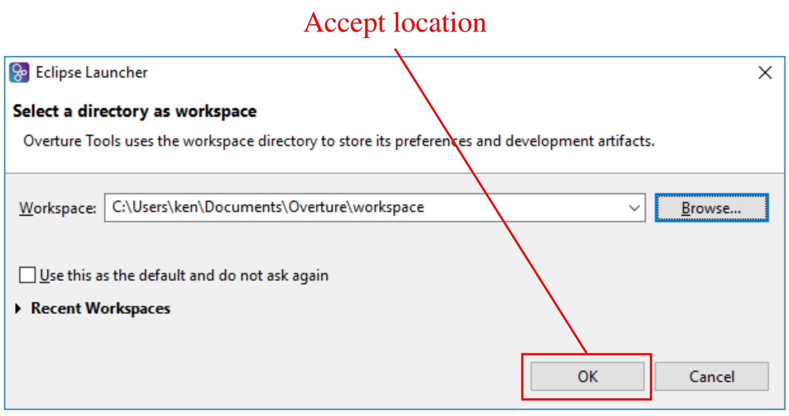
- INTO-CPS Application
- COE (Co-simulation Orchestration Engine) accessible to the Application
- Overture and the FMU plug-in extension

You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window > Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

1 Creating a Project in Overture

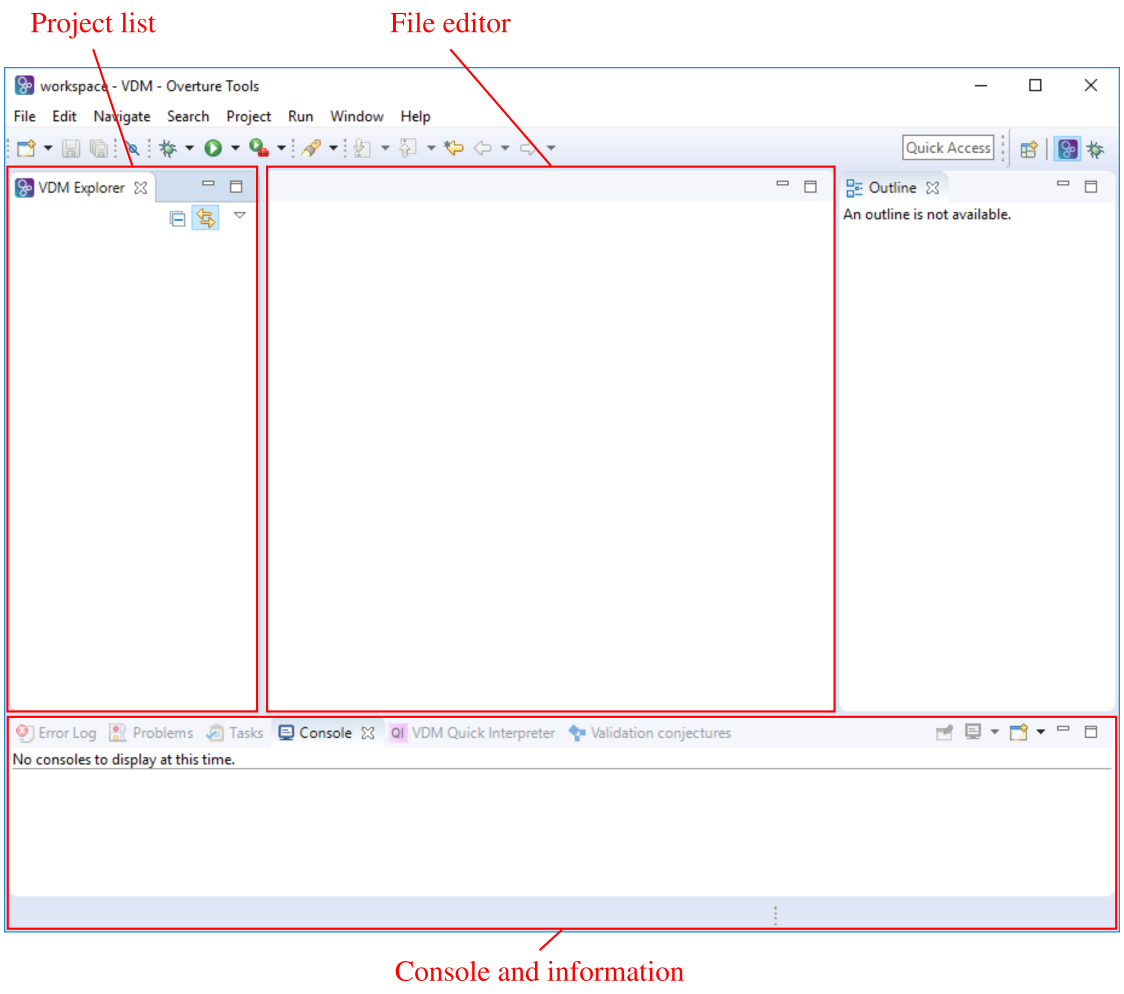
The example in this tutorial is a small line-following robot with two infrared sensors. We will generate a controller FMU that reads these sensors and controls the wheels to follow the line. First we will create a project.

Step 1. Open *Overture*. It will prompt you to select a location for its workspace. You may accept the default location by pressing *OK*, or press *Browse...* to select a different location. If you do not want to be prompted in future, check *Use this as the default and do not ask again*.

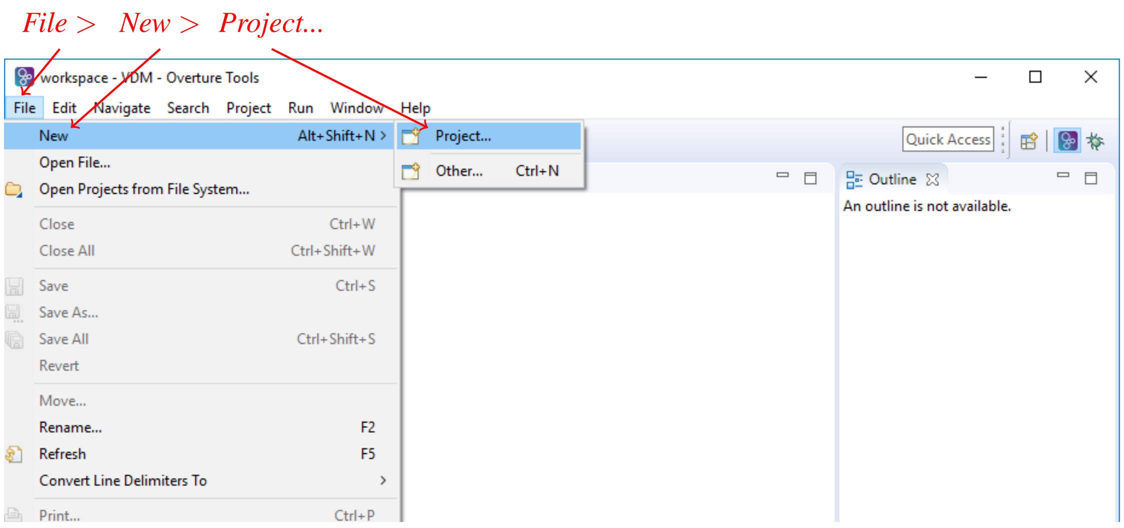


Select a directory as workspace
Overture Tools uses the workspace directory to store its preferences and development artifacts.
Workspace: C:\Users\ken\Documents\Overture\workspace
☐ Use this as the default and do not ask again
Recent Workspaces

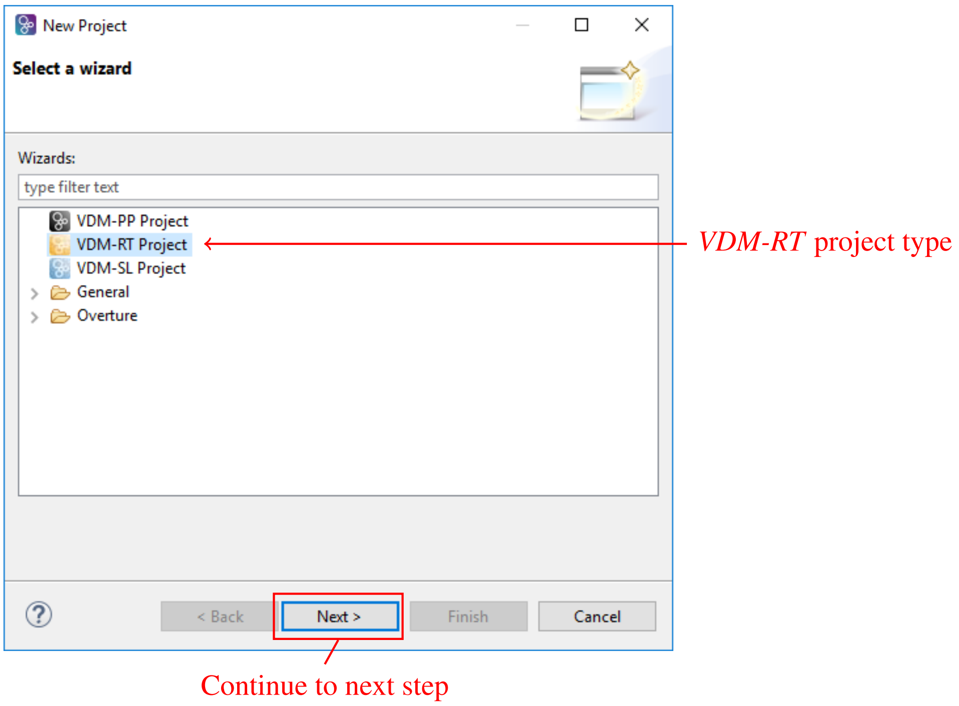
This is the *Overture* window, which includes a project list, file editor and a console.


Project list
File editor
Console and information
workspace - VDM - Overture Tools
File Edit Navigate Search Project Run Window Help
VDM Explorer
Outline
An outline is not available.
Error Log Problems Tasks Console VDM Quick Interpreter Validation conjectures
No consoles to display at this time.
Quick Access

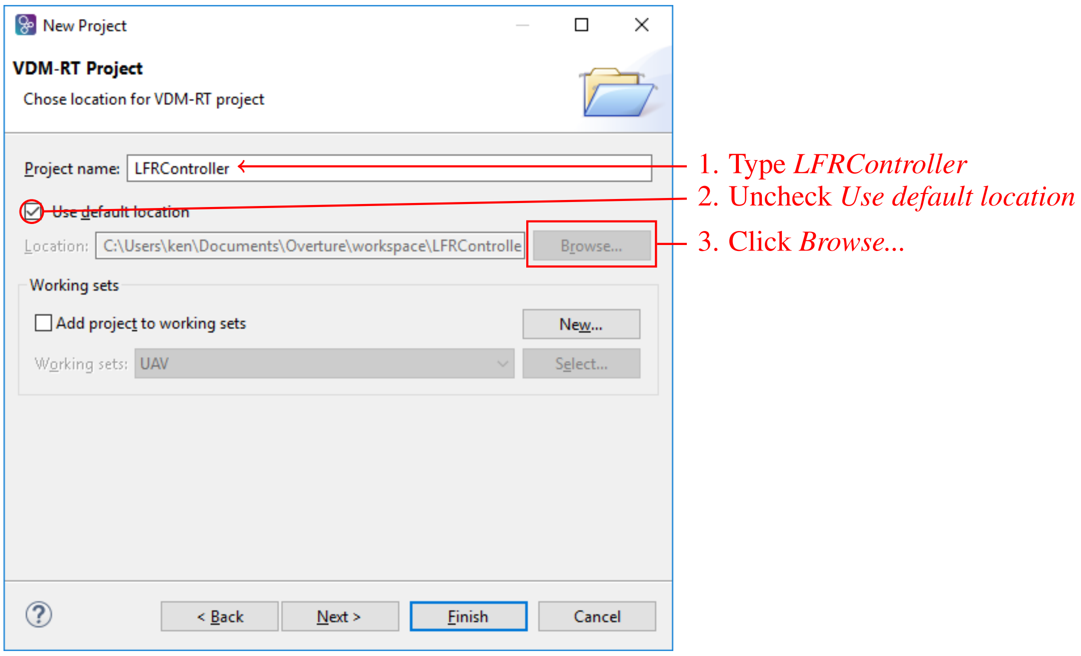
Step 2. First create a project that will hold the controller model. Select *File* > *New* > *Project...*.



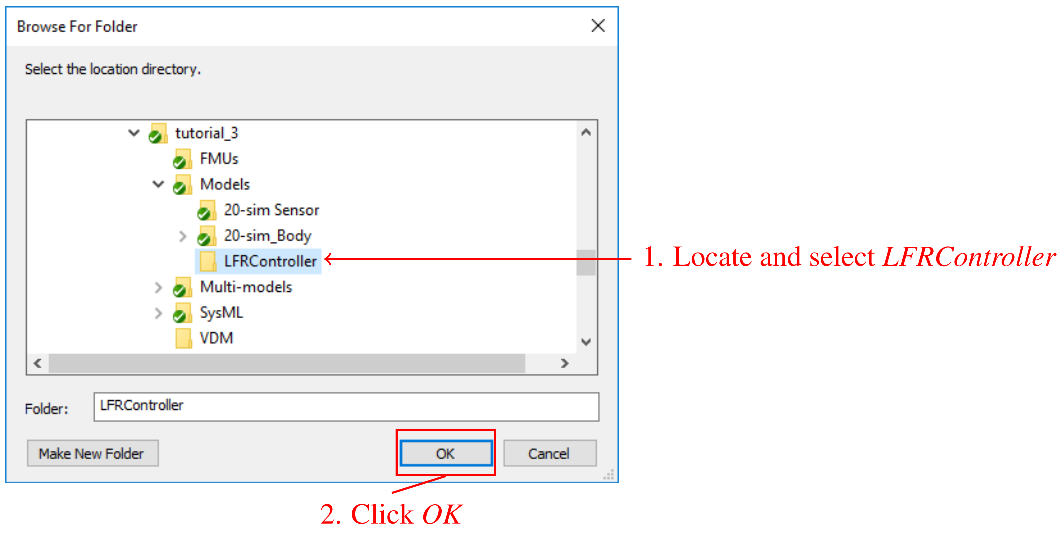
Step 3. In the *New Project* window, select *VDM-RT Project* and click *Next >* to go to the next step.



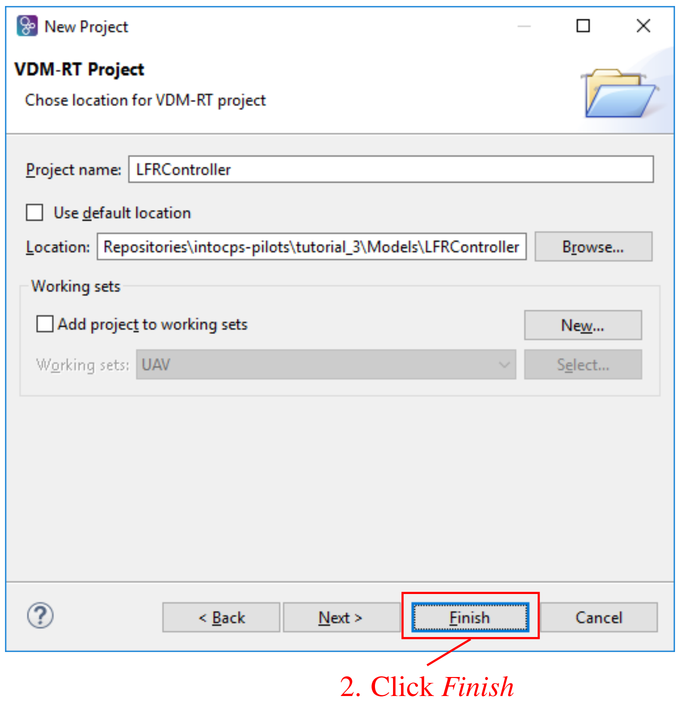
Step 4. The next screen asks for a name for the project. Call it *LFRController*. We will place the project in the *tutorial_3/Models* folder, so uncheck *Use default location* and click *Browse...*


New Project
VDM-RT Project
Chose location for VDM-RT project
Project name: LFRController
Use default location
Location: C:\Users\ken\Documents\Overture\workspace\LFRControlle
Browse...
Working sets
Add project to working sets
New...
Working sets: UAV
Select...
< Back
Next >
Finish
Cancel
1. Type *LFRController*
2. Uncheck *Use default location*
3. Click *Browse...*

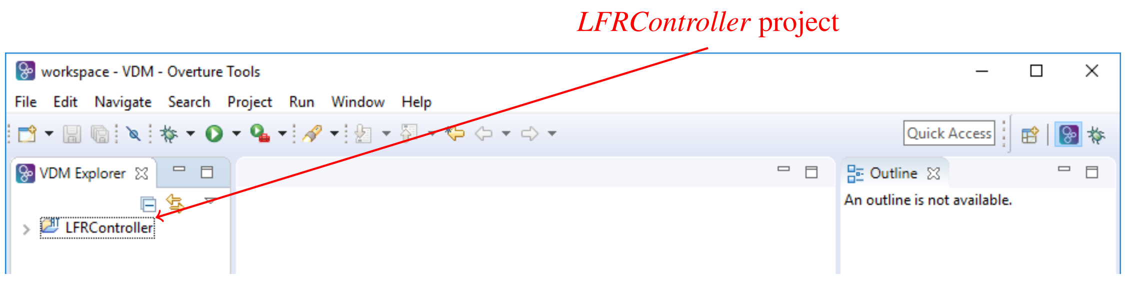
Step 5. Locate and select the folder *tutorial_3/Models/LFRController* and click *OK*.



Step 6. Finally click *Finish* to create the project.


New Project
VDM-RT Project
Chose location for VDM-RT project
Project name: LFRController
Use default location
Location: Repositories\intocps-pilots\tutorial_3\Models\LFRController
Browse...
Working sets
Add project to working sets
New...
Working sets: UAV
Select...
< Back
Next >
Finish
Cancel
2. Click *Finish*

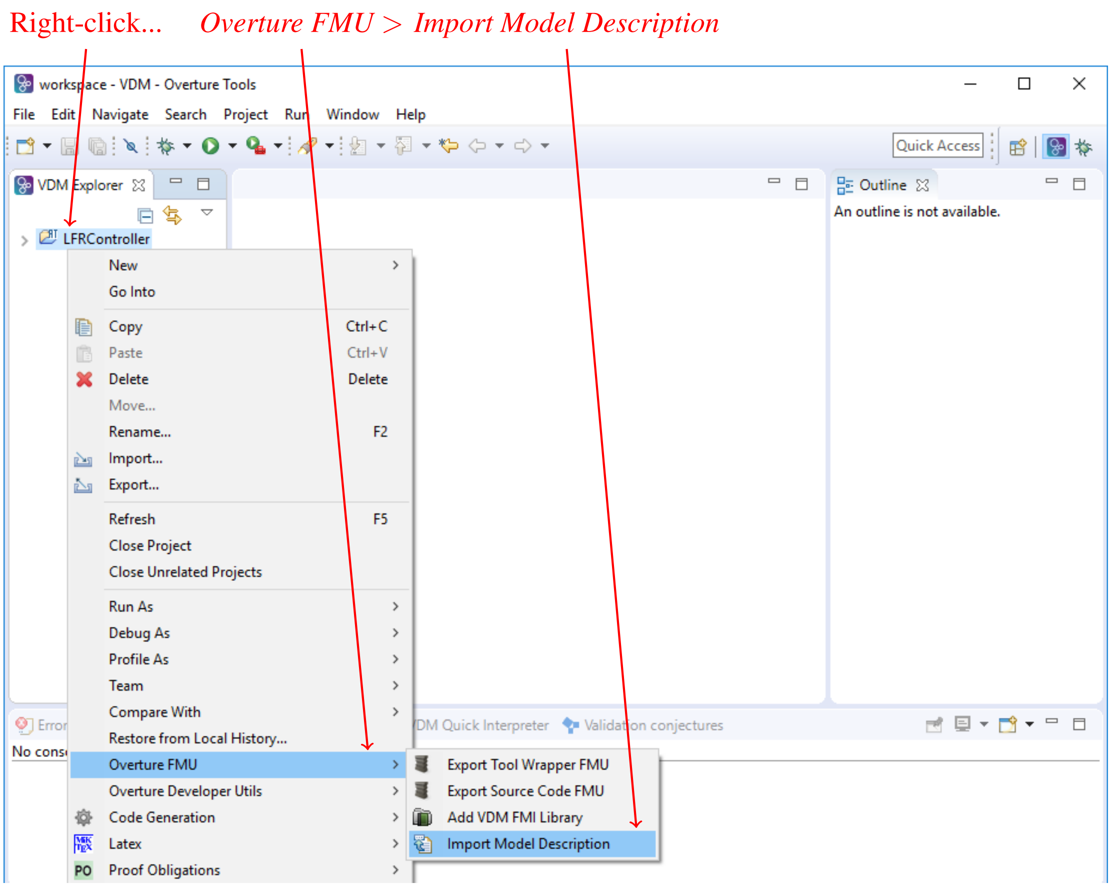
You should see the new project in the project list.


LFRController project
workspace - VDM - Overture Tools
File Edit Navigate Search Project Run Window Help
Quick Access
VDM Explorer
LFRController
Outline
An outline is not available.

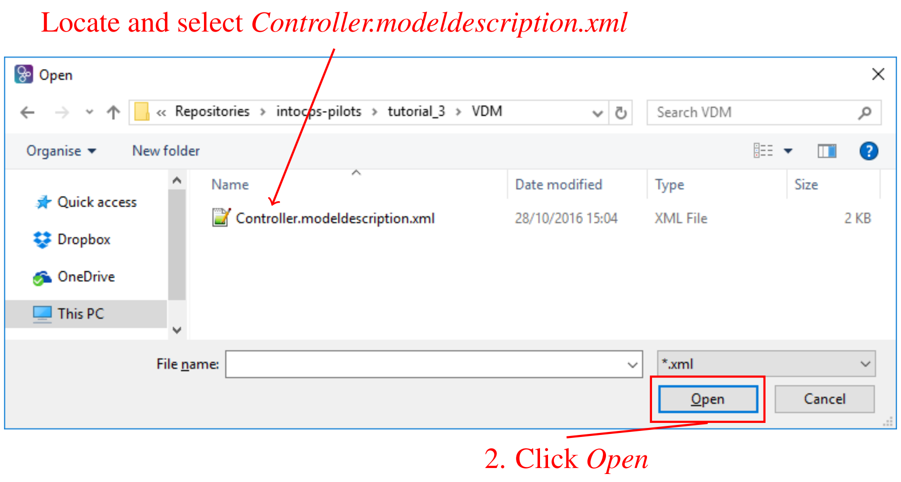
2 Importing a Model Description into Overture

Overture can import model description files to create a skeleton project with the correct input, output and parameter ports, as well as standard boilerplate elements needed in a VDM-RT model.

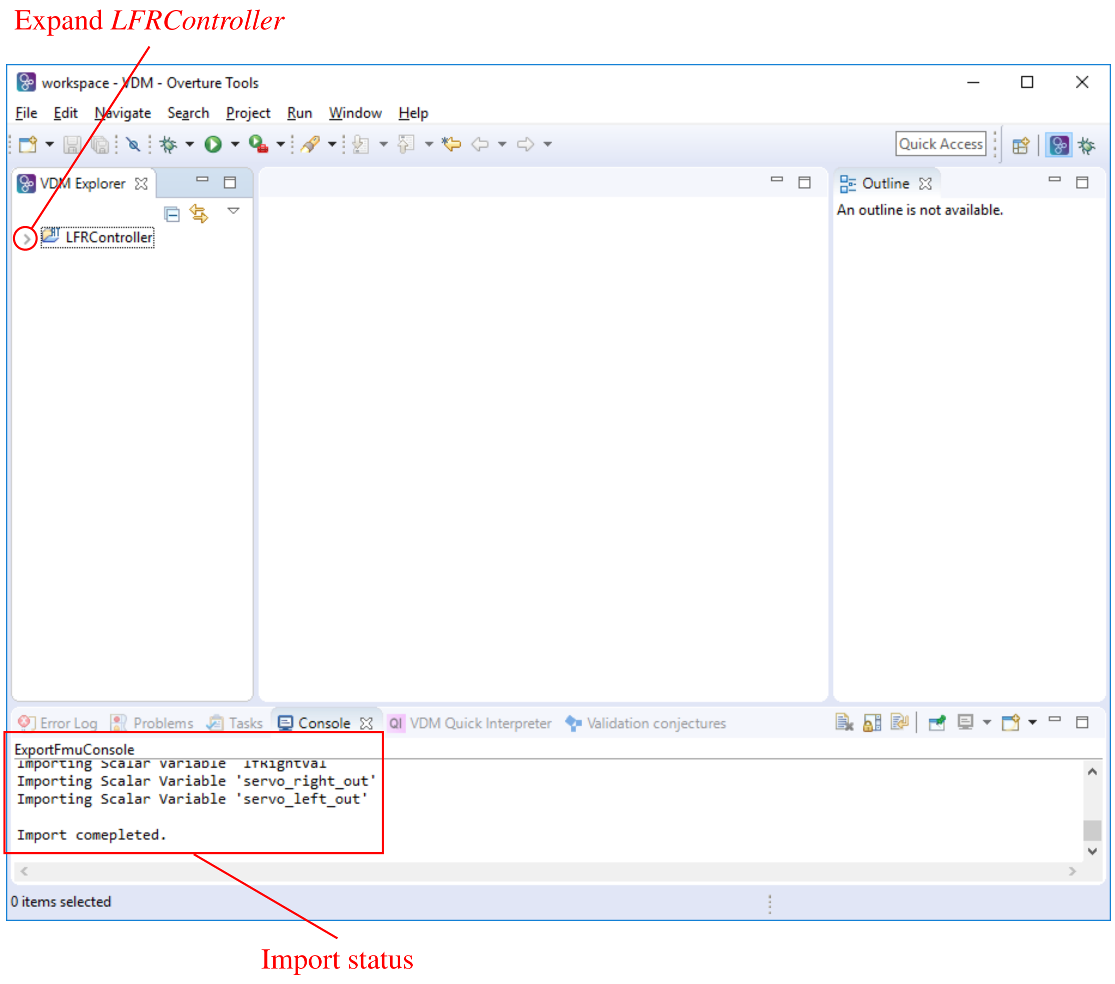
Step 7. To import a model description, right-click on the *LFRController* project and select *Overture FMU > Import Model Description*.



Step 8. Locate the file *tutorial_3/VDM/Controller.modeldescription.xml* that is included in the project and click *Open*.



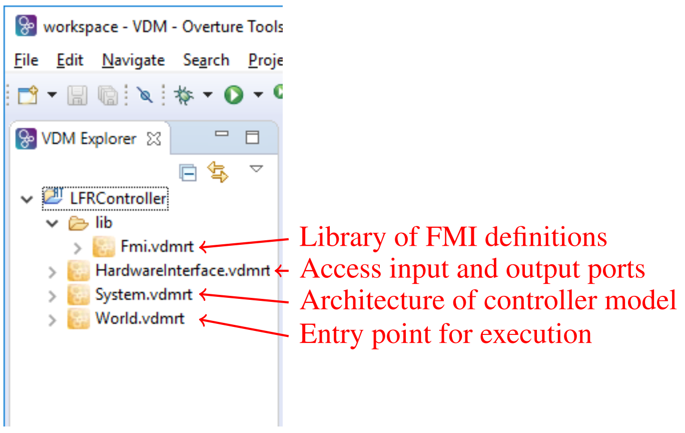
Step 9. Overture will parse the file and populated the project. You can see status messages from the import in the *Console*. Expand the *LFRController* project to see what was imported.



Expand *LFRController*
workspace - VDM - Overture Tools
File Edit Navigate Search Project Run Window Help
Quick Access
VDM Explorer
LFRController
Outline
An outline is not available.
Error Log Problems Tasks Console VDM Quick Interpreter Validation conjectures
0 items selected
Import status

```
ExportFmuConsole
Importing Scalar Variable 'lfRightVal'
Importing Scalar Variable 'servo_right_out'
Importing Scalar Variable 'servo_left_out'

Import compeleted.
```

You should see the following structure:

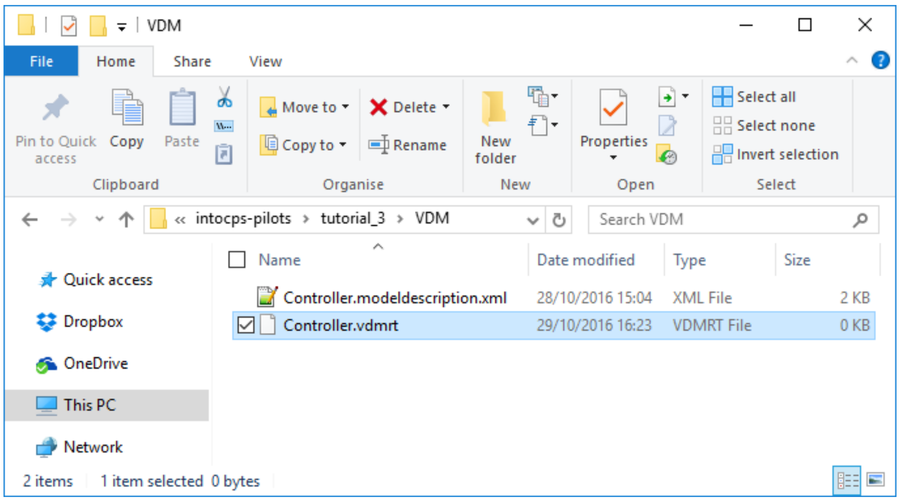


workspace - VDM - Overture Tools
File Edit Navigate Search Proje
VDM Explorer
LFRController
lib
Fmi.vdmrt
HardwareInterface.vdmrt
System.vdmrt
World.vdmrt
Library of FMI definitions
Access input and output ports
Architecture of controller model
Entry point for execution

3 Adding a Controller Class

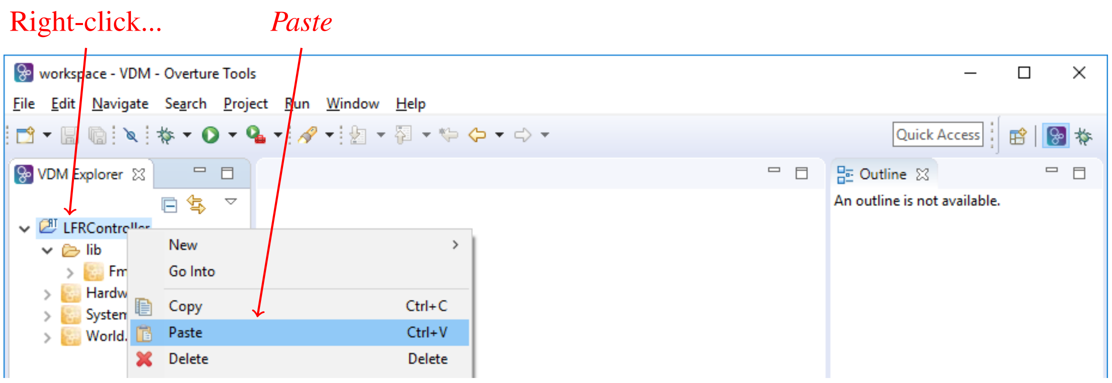
To make a functional controller, we will add a *Controller* class and instantiate it as an object in the *System* class, and set the *World* to start the controller thread. A basic controller class is included in the *tutorial_3* project.

Step 10. Locate the file *tutorial_3/VDM/Controller.vdmrt* on on your file system and copy it.

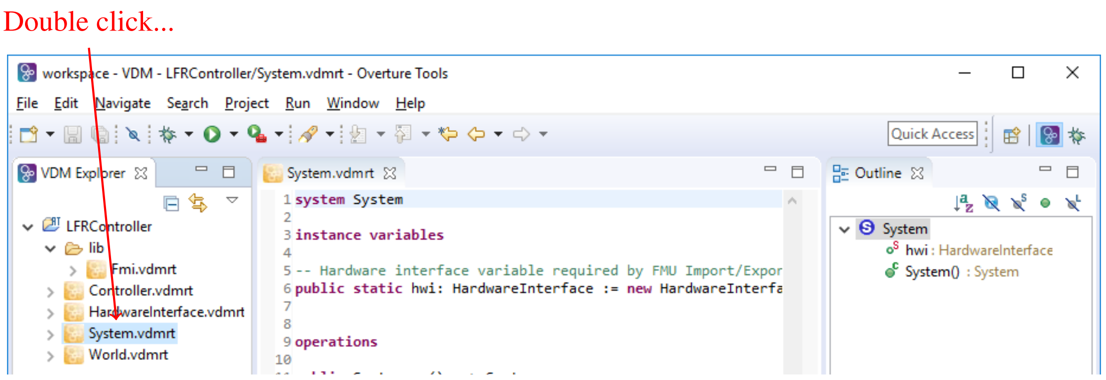


Name	Date modified	Type	Size
Controller.modeldescription.xml	28/10/2016 15:04	XML File	2 KB
Controller.vdmrt	29/10/2016 16:23	VDMRT File	0 KB

Step 11. Right-click on right-click on the *LFRController* project and select *Paste*.



Step 12. Double-click *System.vdmrt* to open the `System` class.



```
system System

instance variables

-- Hardware interface variable required by FMU Import/Expor
public static hwi: HardwareInterface := new HardwareInterfa


operations
```

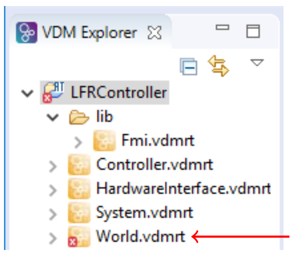
Step 13. Add the highlighted lines to *System.vdmrt*. This will define a `controller` object of the `Controller` class and instantiate it.

```
system System

instance variables

-- Hardware interface variable required by FMU Import/Export
public static hwi: HardwareInterface := new HardwareInterface();

public static controller: Controller := new Controller(
  hwi.servoLeftVal, hwi.servoRightVal, hwi.lfRightVal, hwi.lfLeftVal);

cpu : CPU := new CPU(<FP>, 1E6);

operations

public System : () ==> System
System () ==
(
  cpu.deploy(controller);
);

end System
```

Step 14. Double-click *World.vdmrt* to open the `World` class. Uncomment the highlighted line to tell the controller thread to start at the beginning of co-simulation.

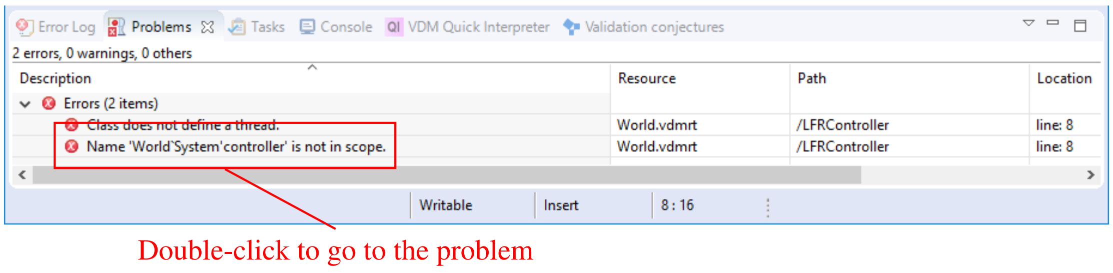
```
class World

operations

public run : () ==> ()
run() ==
 (
  start(System`controller);
  block();
 );

private block : () ==>()
block() ==
  skip;

sync

  per block => false;

end World
```

Step 15. Ensure that your model has no errors. If it does, a red cross will appear next to the file icon in the project browser. (You might have to refresh the project by right-clicking and selecting *Refresh* to see these.)



Class with syntax or type error

Check that you have correctly replicated the listings from Steps 13 and 14. Look at the *Problems* tab at the bottom for information, and double-click items to take you to the problem in the file editor.



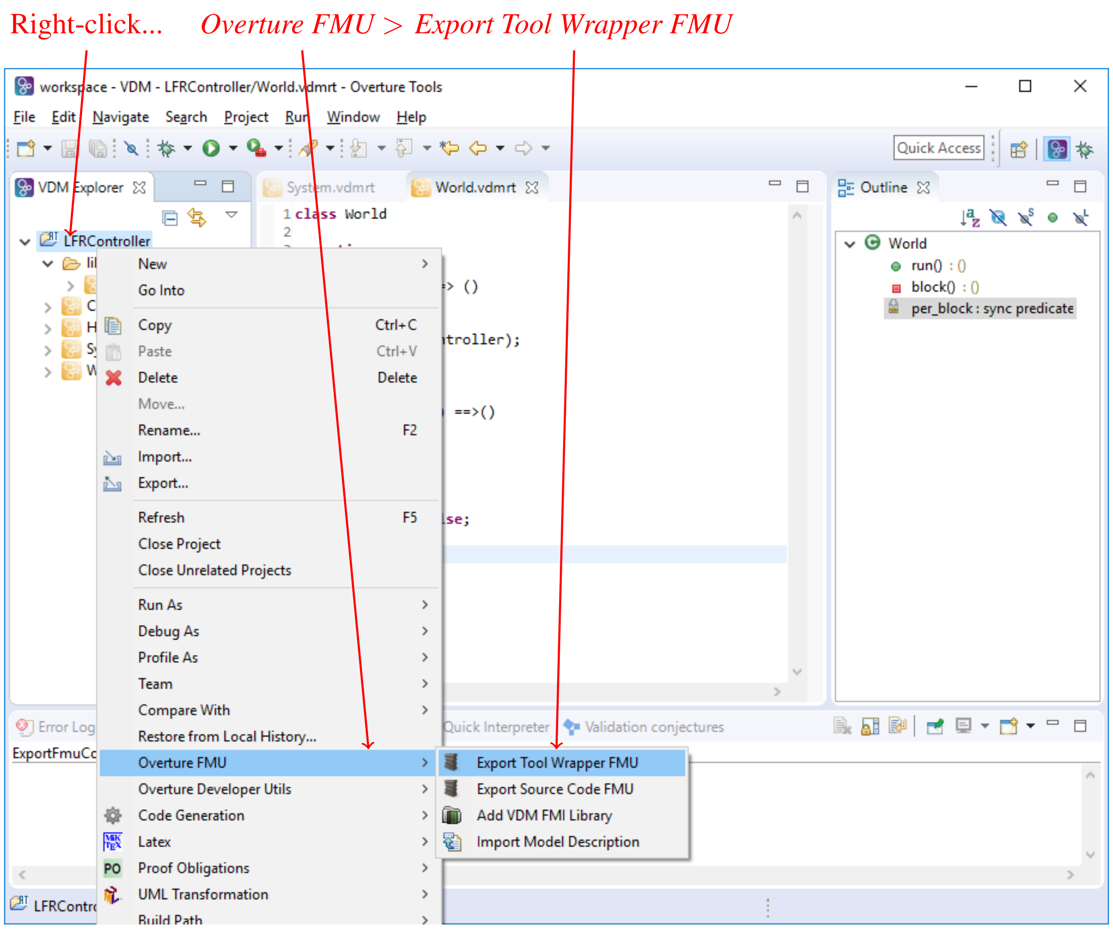
2 errors, 0 warnings, 0 others

Description	Resource	Path	Location
Errors (2 items)			
~~Class does not define a thread.~~	World.vdmrt	/LFRController	line: 8
Name 'World'System'controller' is not in scope.	World.vdmrt	/LFRController	line: 8

4 Exporting an FMU and Adding it to a Multi-model

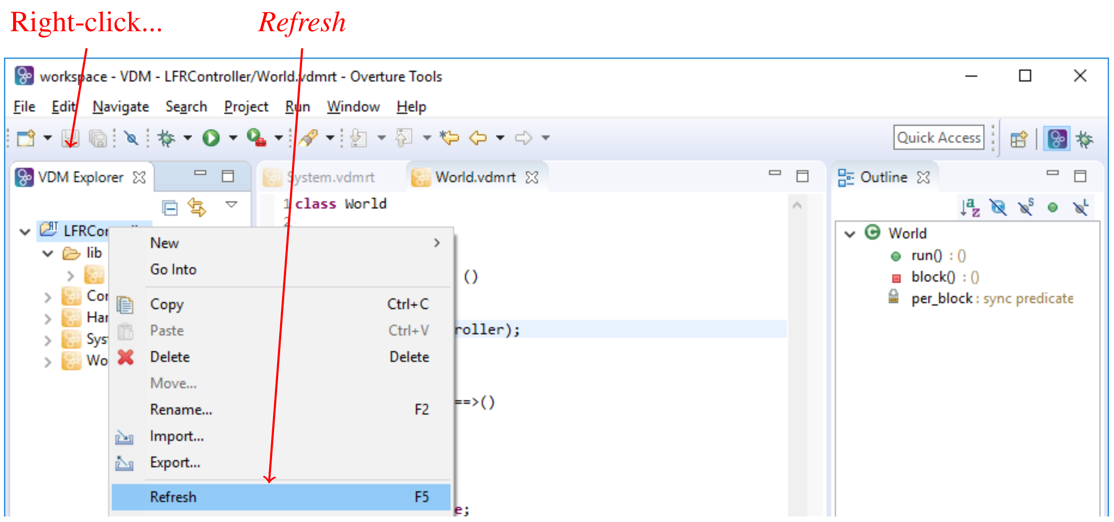
Now that the controller model is complete, we can export an FMU and place it in the *tutorial_3* where the INTO-CPS Application can see it.

Step 16. To export an FMU, right-click on the *LFRController* project and select *Overture FMU* > *Export Tool Wrapper FMU*.

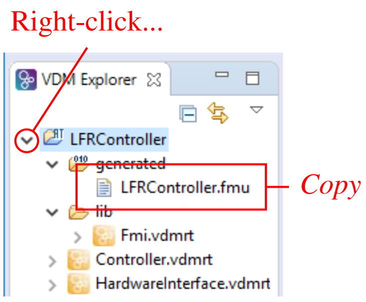


```
class World
```

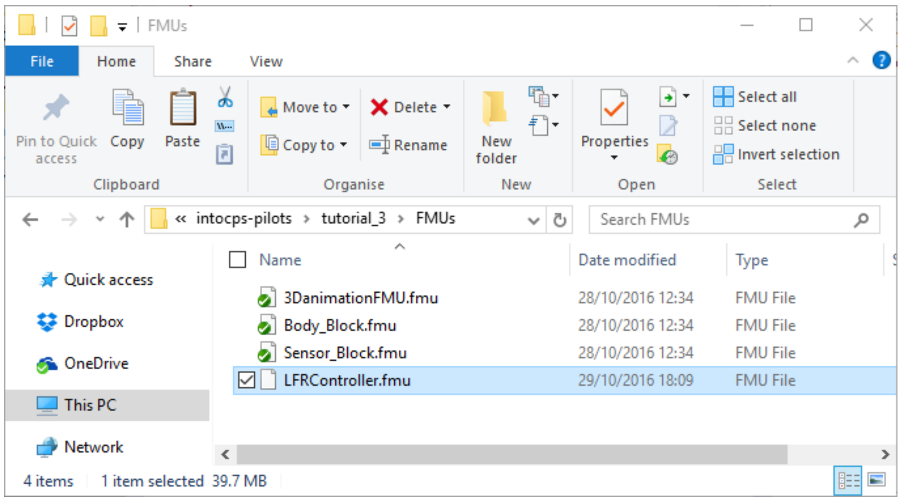
Step 17. Refresh the project so that the generated FMU appears. To do this, right-click on the project and select *Refresh*.



Step 18. A new folder called *generated* will appear. Expand this to see *LFRController.fmu*. Select *LFRController.fmu* and copy it using *Ctrl+C* or right-clicking and selecting *Copy*.



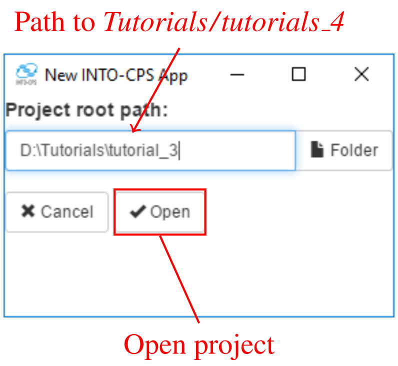
Step 19. Paste *LFRController.fmu* into the *tutorial_3/FMUs* folder on your file system.



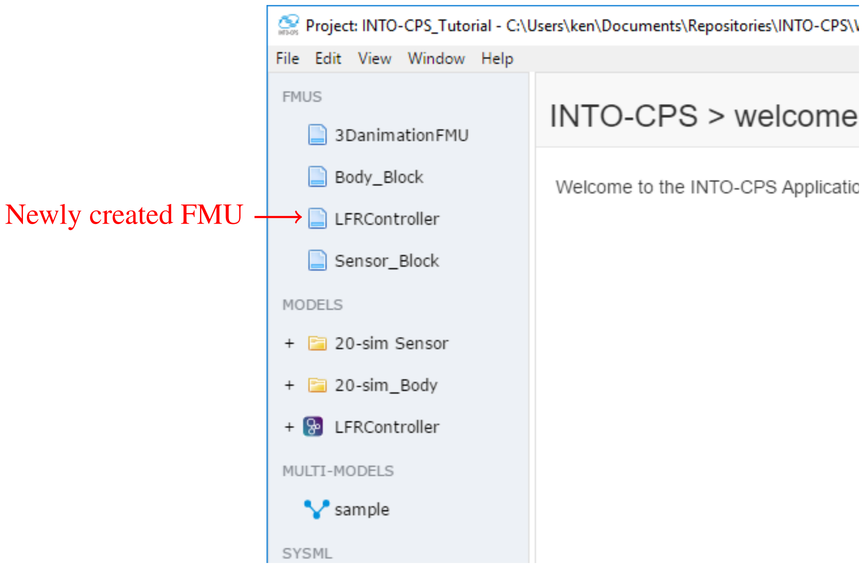
Name	Date modified	Type
3DanimationFMU.fmu	28/10/2016 12:34	FMU File
Body_Block.fmu	28/10/2016 12:34	FMU File
Sensor_Block.fmu	28/10/2016 12:34	FMU File
LFRController.fmu	29/10/2016 18:09	FMU File

5 Co-simulating with the New Controller

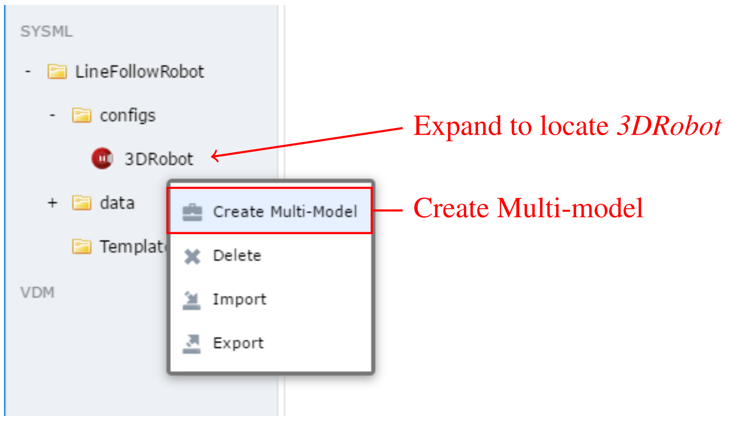
Step 20. Launch the *INTO-CPS Application* and select *File > Open Project*. Set the *Project root path* to the location of *Tutorials/tutorials_4* and click *Open*. You can browse using the *Folder* button.



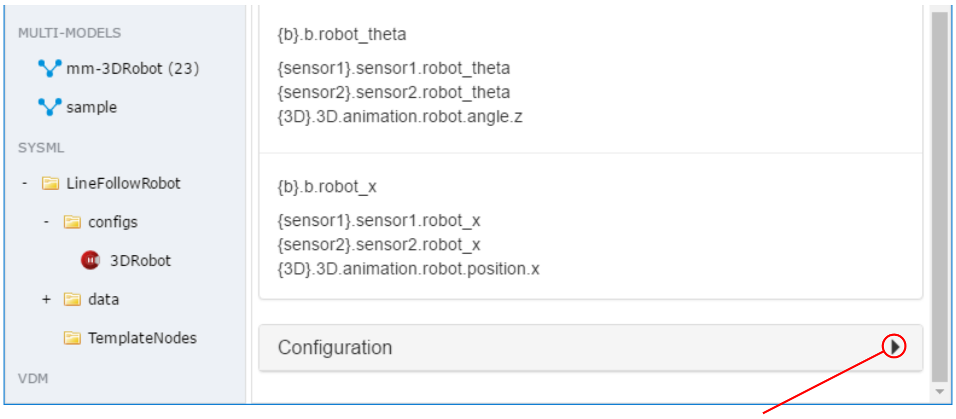
You should see the newly export *LFRController* FMU in the list.



Step 21. In the SysML entry of the project browser, expand the *LineFollowRobot* folder, then *config* folders. Right-click on *3DRobot* and select *Create Multi-Model*.

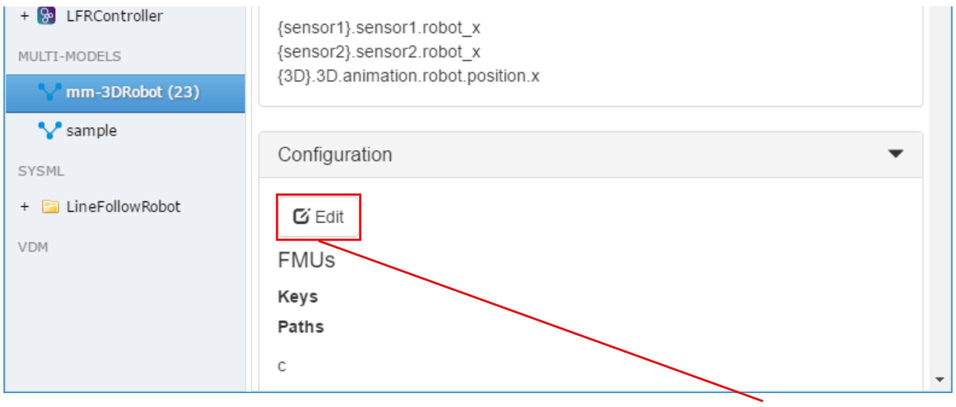


Step 22. We now need to associate FMUs to the multi-model as we did in *Tutorial 2*. Scroll down to find the *Configuration* panel and expand it by clicking the arrow.


MULTI-MODELS
mm-3DRobot (23)
sample
SYSML
LineFollowRobot
configs
3DRobot
data
TemplateNodes
VDM
{b}.b.robot_theta
{sensor1}.sensor1.robot_theta
{sensor2}.sensor2.robot_theta
{3D}.3D.animation.robot.angle.z
{b}.b.robot_x
{sensor1}.sensor1.robot_x
{sensor2}.sensor2.robot_x
{3D}.3D.animation.robot.position.x
Configuration

Expand *Configuration*

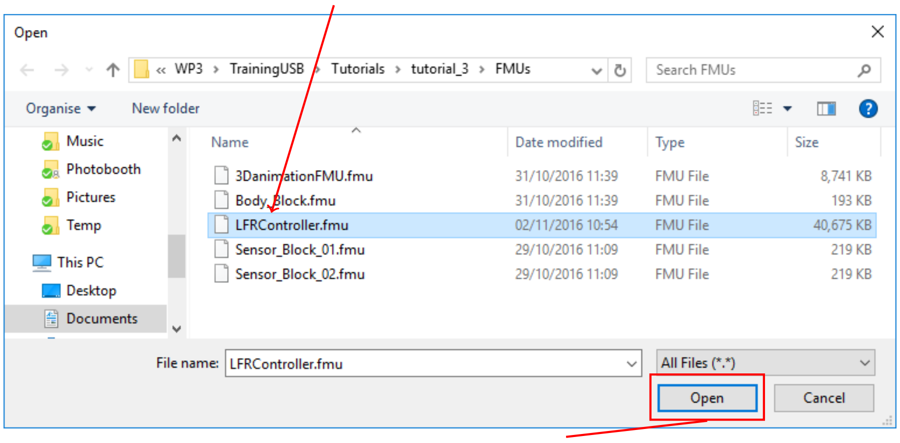
Step 23. Scroll down and click *Edit*.



Edit configuration

Step 24. As in *Tutorial 2*, in the FMUs section press *File* next to the Controller element, *c*. A file browser window will open and show five FMUs (if the file browser does not show the FMUs, navigate to *tutorials_4/FMUs*). Select *FMUController.fmu* and click *Open*.

1. Locate and select *FMUController.fmu*



Name	Date modified	Type	Size
3DanimationFMU.fmu	31/10/2016 11:39	FMU File	8,741 KB
Body_Block.fmu	31/10/2016 11:39	FMU File	193 KB
LFRController.fmu	02/11/2016 10:54	FMU File	40,675 KB
Sensor_Block_01.fmu	29/10/2016 11:09	FMU File	219 KB
Sensor_Block_02.fmu	29/10/2016 11:09	FMU File	219 KB

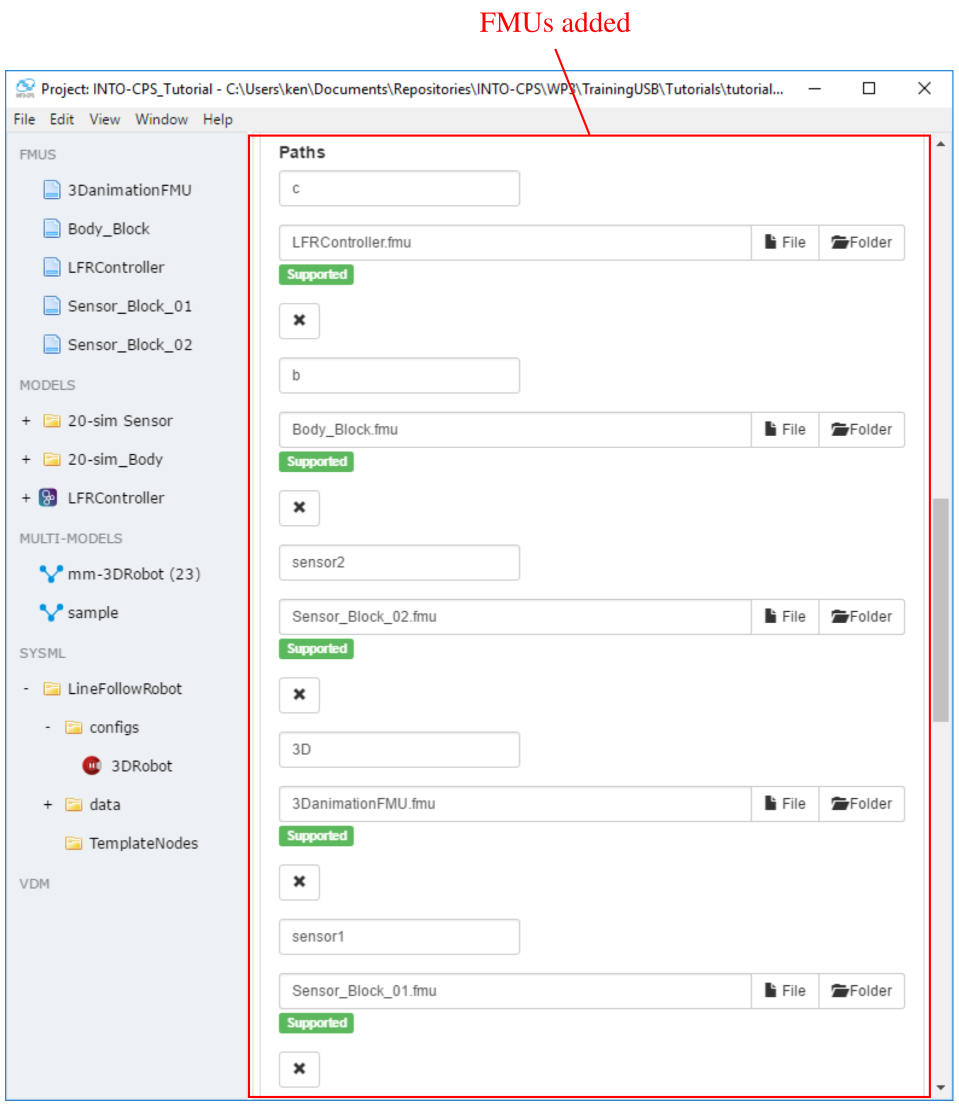
File name: LFRController.fmu

2. Click *Open*

Step 25. Repeat this for the remaining elements:

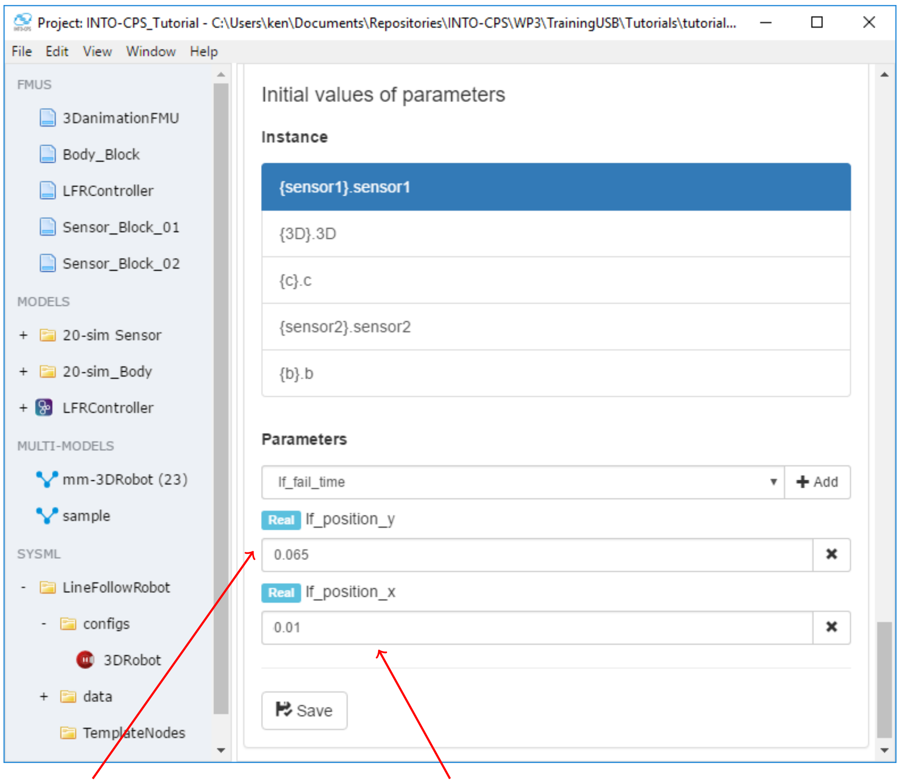
- *b* : *Body_Block.fmu*
- *3D* : *3DanimationFMU.fmu*
- *sensor1* : *Sensor_Block_01.fmu*
- *sensor2* : *Sensor_Block_02.fmu*

The complete set of FMUs will look like this:



Step 26. Scroll down to the *Initial values of parameters section*, and click {*sensor1*}.*sensor1*. In the *Parameters* section, enter the following values:

- *lf_position_y* = 0.065
- *lf_position_x* = 0.01

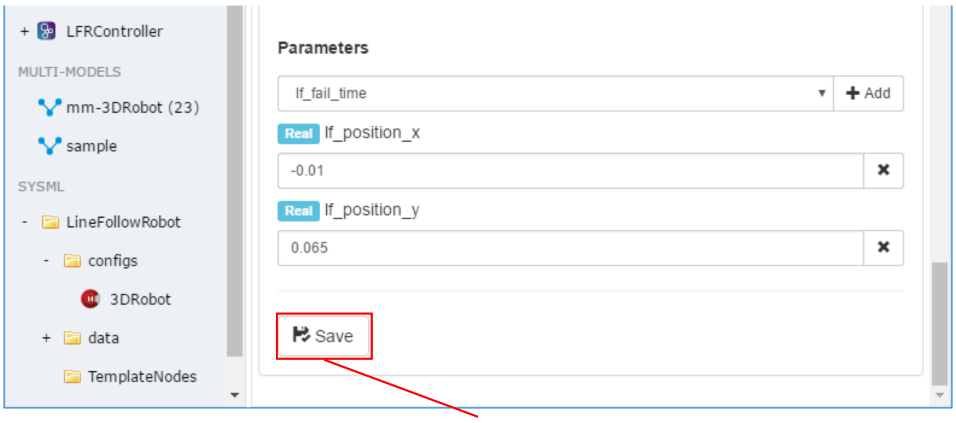


lf_position_y *lf_position_x*

Step 27. Repeat the previous step for the second sensor, {*sensor2*}.*sensor2*, with the following values:

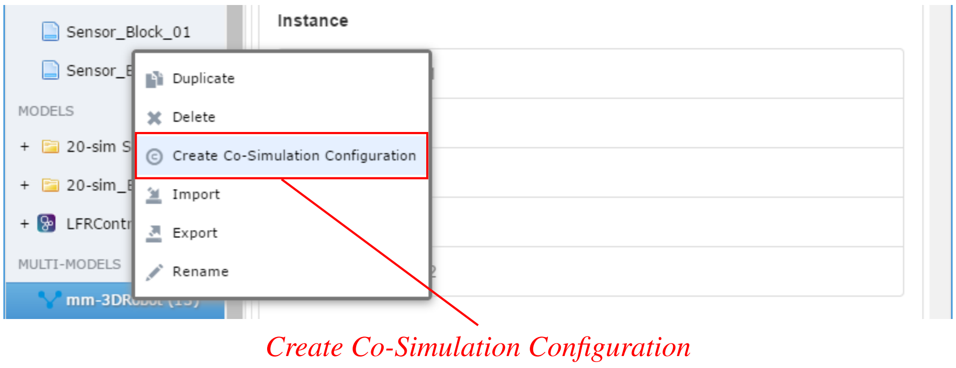
- *lf_position_x* = -0.01
- *lf_position_y* = 0.065

Step 28. *Save* the *Configuration*.

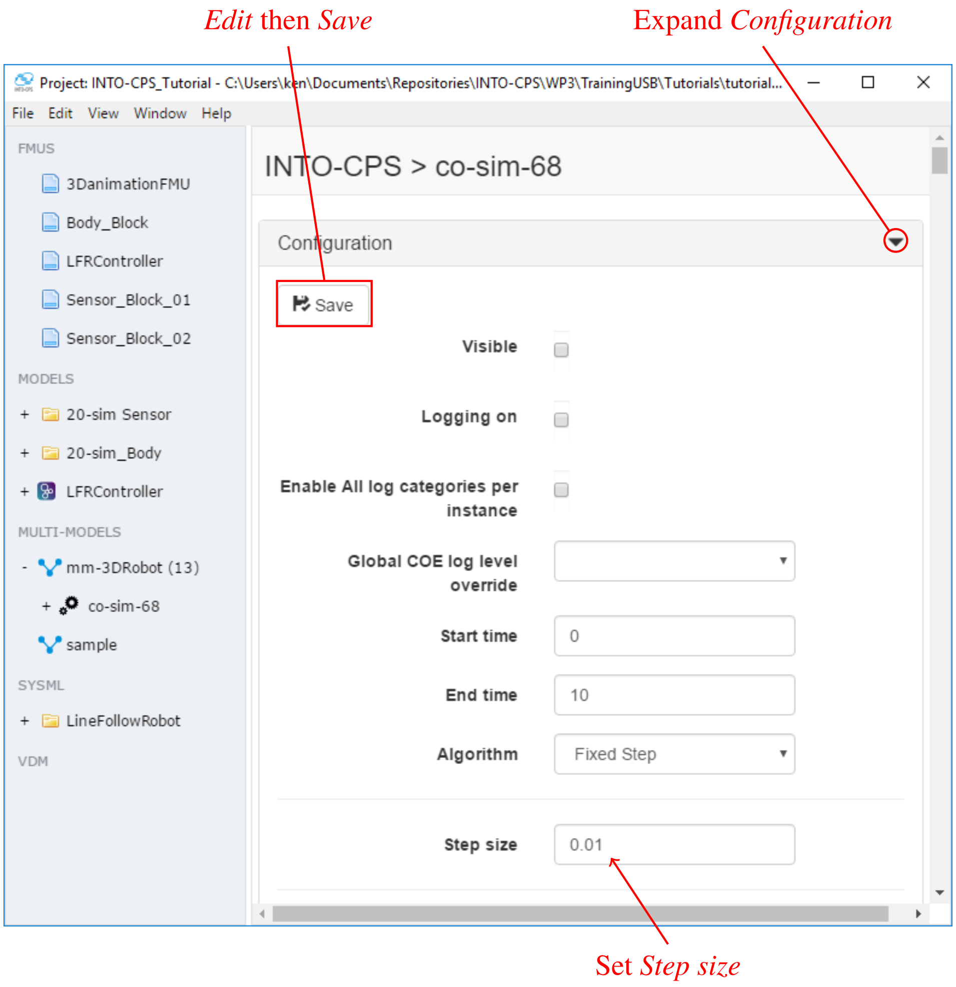


Save configuration

Step 29. Right-click on the new multi-model configuration and select *Create Co-simulation Configuration*.

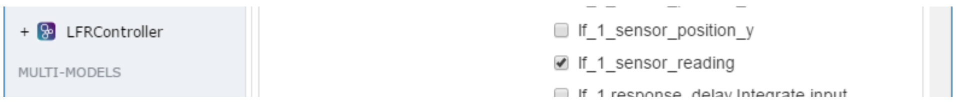


Step 30. Set the *Step size* to 0.01. Don't forget to press *Edit* then *Save*.



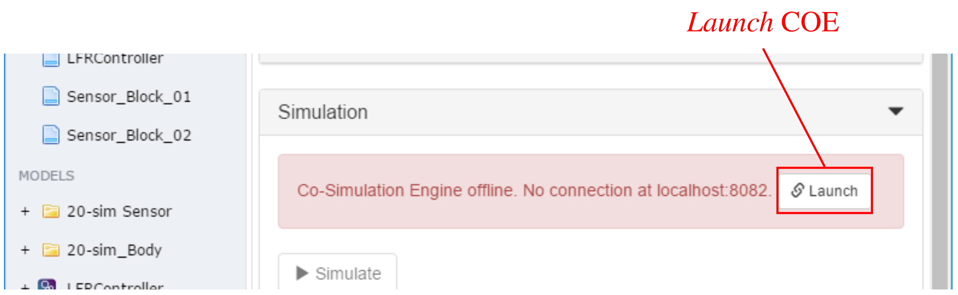
Visible ☐
Logging on ☐
Enable All log categories per instance ☐
Global COE log level override
Start time 0
End time 10
Algorithm Fixed Step
Step size 0.01

Step 31. Check *lf_1_sensor_reading* from {*sensor1*}.*sensor1* and {*sensor2*}.*sensor2* to see the sensor values appear in the *Livestream Configuration*.



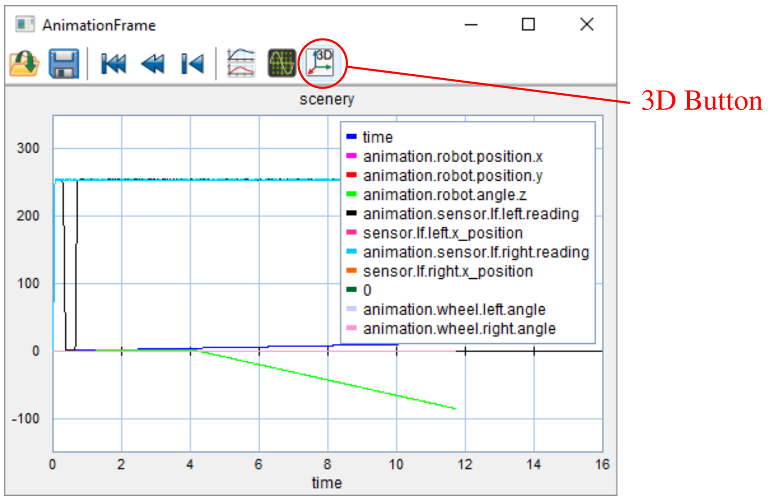
☐ lf_1_sensor_position_y
☑ lf_1_sensor_reading
☐ lf_1_response_delay.Integrate.input

Step 32. Launch the COE if necessary (see *Tutorial 1 — First Co-simulation* for a reminder if needed).

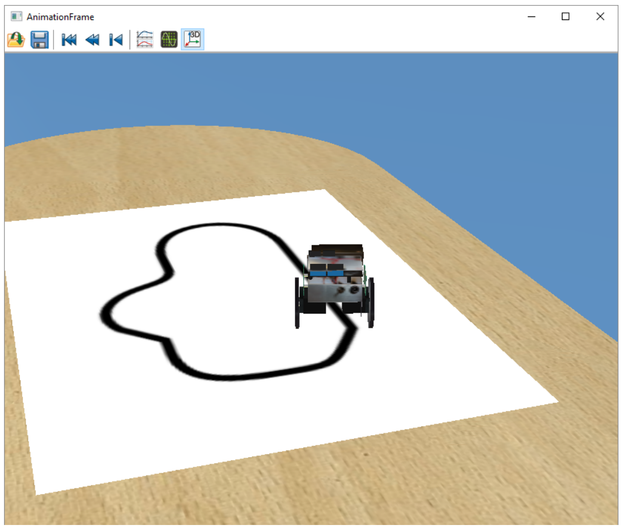


Co-Simulation Engine offline. No connection at localhost:8082.

Step 33. When the COE is running, click the *Simulate* button. The *Animation Frame* should appear. You can click the *3D* button to see the 3D visualisation of the robot.


AnimationFrame
3D Button
scenery
time
animation.robot.position.x
animation.robot.position.y
animation.robot.angle.z
animation.sensor.lf.left.reading
sensor.lf.left.x_position
animation.sensor.lf.right.reading
sensor.lf.right.x_position
0
animation.wheel.left.angle
animation.wheel.right.angle
300
200
100
0
-100
0
2
4
6
8
10
12
14
16
time

Step 34. If everything went well, the robot should follow the line as in *Tutorial 2 — Adding FMUs*.



You can go back to *Overture* and look at the logic in `Controller.vdmrt`, and try to make some changes. Just repeat Step 16. to Step 19. to regenerate and copy the FMU, then press *Simulate*.

Tutorial 5 — Building Controllers in VDM

Overview

This fourth INTO-CPS tutorial will help you to:

1. Generate a more complete controller in VDM using Overture
2. Add behaviours to deal with realistic behaviours (noise and ambient light)
3. Add modes for degraded behaviours after sensor failure

Requirements

This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- INTO-CPS Application
- COE (Co-simulation Orchestration Engine) accessible to the Application
- Overture including FMU plug-in

Tools can be downloaded through the Application (*Window > Show Download Manager*) or may have been provided on the USB drive at your training session. Please ask if you are unsure.

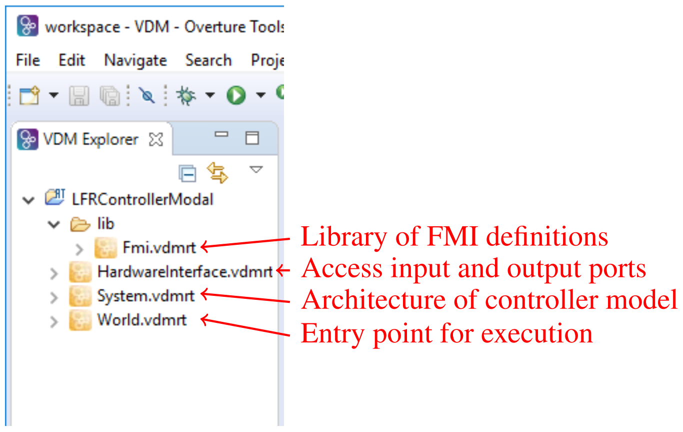
1 Make an Overture Project

We will begin by creating a project in Overture and importing a model description file as in Tutorial 4.

Step 1. Create a project in Overture called *LFRControllerModal*. If you are unsure, follow Steps 1–6 of Tutorial 4. You can accept the default location (your Overture workspace) in Step 4, or make a folder in your *tutorial_5/Models* directory and place the project there.

Step 2. Import *tutorial_5/Models/Controller.modeldescription.xml* using the *Overture FMU > Import Model Description* context menu, as in Steps 7–9 of Tutorial 4.

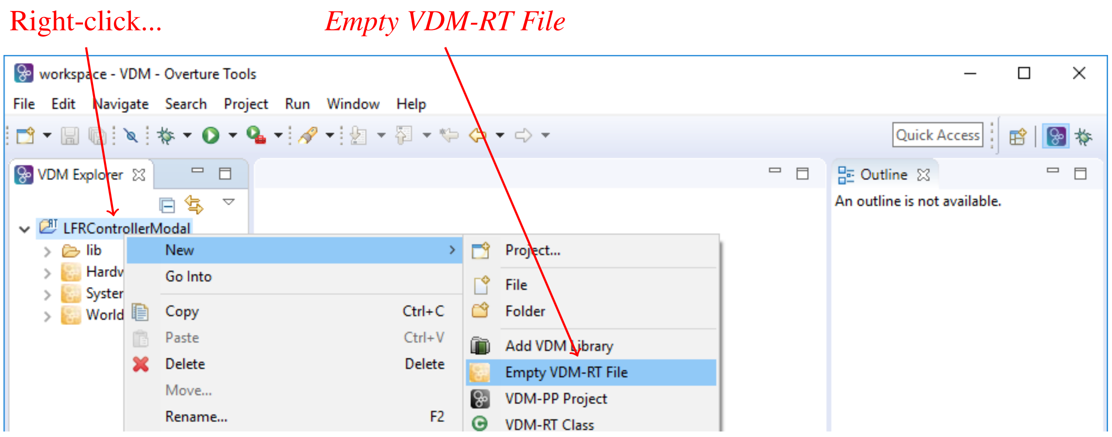
You should see the following structure:


workspace - VDM - Overture Tools
File Edit Navigate Search Proje
VDM Explorer
LFRControllerModal
lib
Fmi.vdmrt
HardwareInterface.vdmrt
System.vdmrt
World.vdmrt
Library of FMI definitions
Access input and output ports
Architecture of controller model
Entry point for execution

2 Adding Structure and a Skeleton Controller

To make a functional controller, we will we will add a *Controller* class to contain the control logic. To provide structure to help with more complicated logic later, we will add classes to represent the *sensors* and *actuators* that manage the interface to the COE (and the physical simulation made in 20-sim). The *Controller* class will use objects of these classes to read the environment and control the robot.

Step 3. Right-click on the *LFRControllerModal* project and select *New* > *Empty VDM-RT File*. Call is *Controller* and click *Finish*.



Step 4. Paste in the following listing and click *Save*.

```
class Controller

instance variables

leftSensor: IRSensor;
rightSensor: IRSensor;

leftServo: Servo;
rightServo: Servo;

operations

public Controller: IRSensor * IRSensor * Servo * Servo ==> Controller
Controller(lfl, lfr, ls, rs) == (
  leftSensor := lfl;
  rightSensor := lfr;
  leftServo := ls;
  rightServo := rs
);

Step: () ==> ()
Step() == cycles(20) (
    -- debug information
    IO`printf("Left sensor: %s (%s), right sensor: %s (%s)\n",
      [leftSensor.getReading(),leftSensor.hasFailed(),
       rightSensor.getReading(),rightSensor.hasFailed()]);
);

thread

periodic(10E6, 0, 0, 0)(Step)

end Controller
```

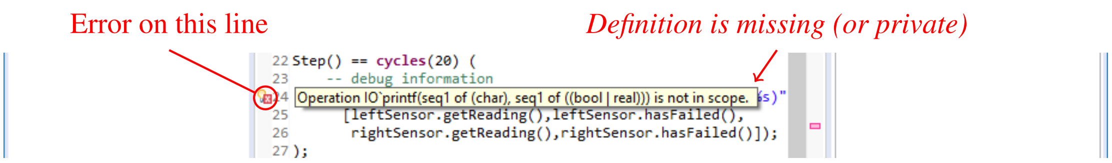
Step 5. Repeat the above step to make a file called *IRSensor* and populate it with the listing below. This class provides read access to two FMI ports, one for the sensor reading (`getReading`) and one to say if the sensor has failed (`hasFailed`).

```
class IRSensor

instance variables

-- access to ports from co-simulation
port : RealPort;
failed : BoolPort

operations

-- constructor for IRSensor
public IRSensor: RealPort * BoolPort ==> IRSensor
IRSensor(p,f) == (
  port := p;
  failed := f
);

public getReading: () ==> real
getReading() == (
  return port.getValue()
);

public hasFailed: () ==> bool
hasFailed() == (
  failed.getValue()
)

end IRSensor
```

Step 6. Next, create a file called *Servo* with the listing below. This class provides write access to a port to move the wheels of the robot (`setSpeed`). The range is -1 to 1 for full forwards or backwards, so a pre-condition is included to protect the operation. Note that since one servo on the robot is flipped over, a reverse flag can be set in the constructor to that setting both servos to 1 makes the robot go forwards at full speed.

```
class Servo

instance variables

-- access to ports from co-simulation
port: RealPort;
reversed: bool

operations

-- constructor for Servo
public Servo: RealPort * bool ==> Servo
Servo(p,r) == (
  port := p;
  reversed := r
);

public setSpeed: real ==> ()
setSpeed(value) == (
  if reversed
  then port.setValue(-value)
  else port.setValue(value)
)
pre -1 <= value and value <= 1

end Servo
```

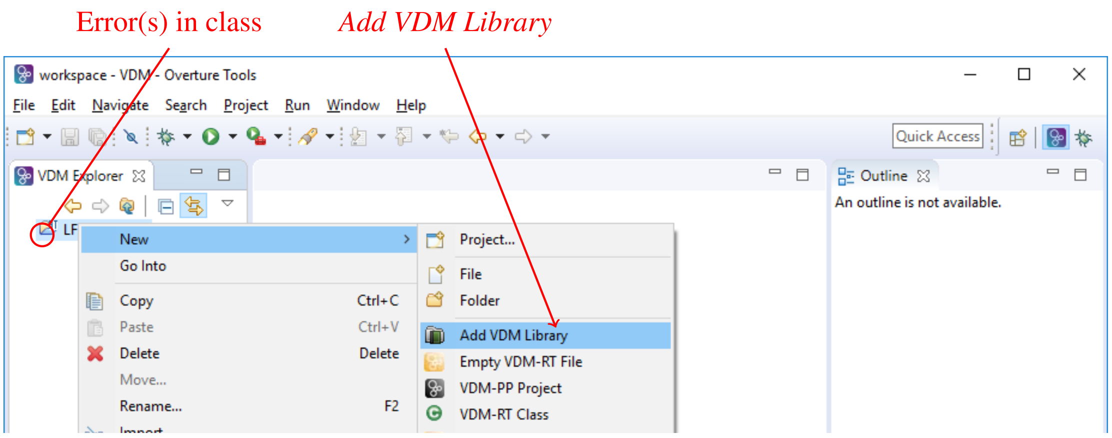
Step 7. You should notice in the *VDM Explorer* that `Controller.vdmrt` has a small red cross next to its icon. This means that there is one or more errors in the definition. In this case, the `IO` standard library is missing, which can be seen by hovering the mouse pointer over the error in the class. Note that "out of scope" either means that the definition is private, or missing entirely.



```
Step() == cycles(20) (
    -- debug information
                                                      %s)"
      [leftSensor.getReading(),leftSensor.hasFailed(),
       rightSensor.getReading(),rightSensor.hasFailed()]);
);
```

Operation IO`printf(seq1 of (char), seq1 of ((bool | real))) is not in scope.

Add the library by right-clicking the project and selecting *New* > *Add VDM Library*. Then check the *IO* box and click *Finish*. This will add `IO.vdmrt` to the `/lib` folder and the error should go away.



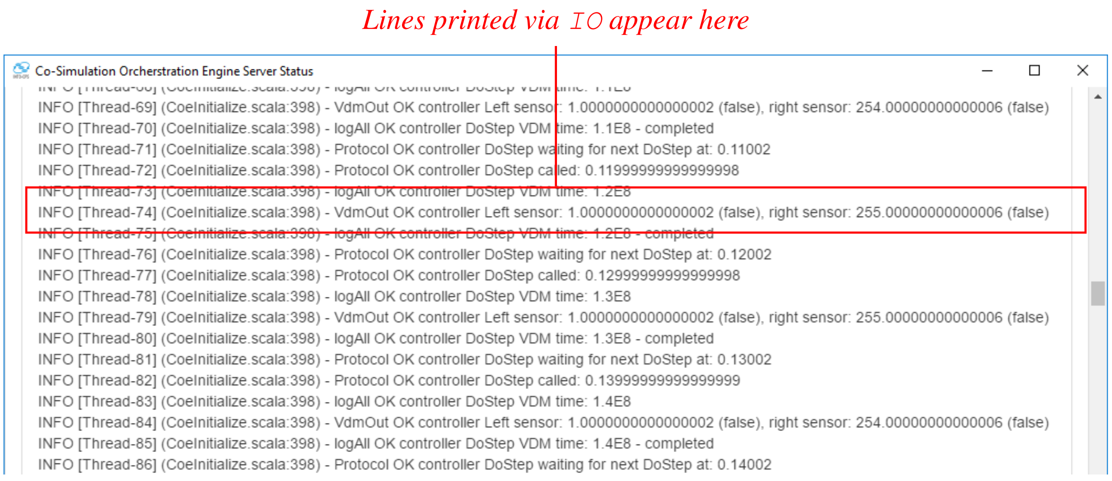
Step 8. In order to complete this basic controller and make an FMU, we must update the `System` class to instantiate sensor and actuator objects, then instantiate a controller object with these. Add the following lines to `System.vdmrt`:

```
system System

instance variables

-- Hardware interface variable required by FMU Import/Export
public static hwi: HardwareInterface := new HardwareInterface();

public static controller: [Controller] := nil;

private leftSensor: IRSensor;
private rightSensor: IRSensor;

private leftServo: Servo;
private rightServo: Servo;

private cpu : CPU := new CPU(<FP>, 1E6);

operations

public System : () ==> System
System () ==
(
  -- create sensor and actuator objects
  leftSensor := new IRSensor(hwi.lfLeftVal, hwi.lfLeftFailFlag);
  rightSensor := new IRSensor(hwi.lfRightVal, hwi.lfRightFailFlag);
  leftServo := new Servo(hwi.servoLeftVal, false);
  rightServo := new Servo(hwi.servoRightVal, true);
  -- create controller object
  controller := new Controller(leftSensor, rightSensor, leftServo, rightServo);

  -- deploy objects
  cpu.deploy(controller,  "Controller");
  cpu.deploy(leftSensor,  "Left sensor");
  cpu.deploy(rightSensor, "Right sensor");
  cpu.deploy(leftServo,   "Left servo");
  cpu.deploy(rightServo,  "Right servo");
);

end System
```

Step 9. Finally, uncomment line 8 of `World.vdmrt` to ensure that the controller logic will be started at the beginning of co-simulation:

```
class World

operations

public run : () ==> ()
run() ==
 (
  start(System`controller);
  block();
 );

private block : () ==>()
block() ==
  skip;

sync

  per block => false;

end World
```

Step 10. You can now test the project in INTO-CPS by exporting an FMU and pasting it into the *tutorial_5/FMUs* (follow Steps 16–19 of Tutorial 4). Then you can then open *tutorial_5* in the *INTO-CPS Application* and co-simulate with the *lfr-3d* or *lfr-non3d* multi-models. You should see the output from the VDM controller in the COE output:


Lines printed via IO appear here
Co-Simulation Orcherstration Engine Server Status

```
INFO [Thread-68] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.1E8
INFO [Thread-69] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 254.00000000000006 (false)
INFO [Thread-70] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.1E8 - completed
INFO [Thread-71] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.11002
INFO [Thread-72] (CoeInitialize.scala:398) - Protocol OK controller DoStep called: 0.11999999999999998
INFO [Thread-73] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.2E8
INFO [Thread-74] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 255.00000000000006 (false)
INFO [Thread-75] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.2E8 - completed
INFO [Thread-76] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.12002
INFO [Thread-77] (CoeInitialize.scala:398) - Protocol OK controller DoStep called: 0.12999999999999998
INFO [Thread-78] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.3E8
INFO [Thread-79] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 255.00000000000006 (false)
INFO [Thread-80] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.3E8 - completed
INFO [Thread-81] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.13002
INFO [Thread-82] (CoeInitialize.scala:398) - Protocol OK controller DoStep called: 0.13999999999999999
INFO [Thread-83] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.4E8
INFO [Thread-84] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 254.00000000000006 (false)
INFO [Thread-85] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.4E8 - completed
INFO [Thread-86] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.14002
```

As you can see from the live graph (and 3D visualisation), the left sensor is over black (low) and the right sensor is over white (high). The range is (0,255).

3 A Basic Controller

We will now add some basic line following logic. A so-called "bang-bang" controller turns left if the line is to the left, and right if the line is to the right. This creates a characteristic zig-zag motion.

Step 11. The control logic in the `Controller` class is in the `Step` operation. This is called periodically. Add an `if` statement to the `Step` operation to turn the robot to the left if the left sensor is over black and the right sensor is over white. You can assume that a sensor reading over 150 (halfway) is white and below 150 is black. You can drive the robot left and forward using the following calls:

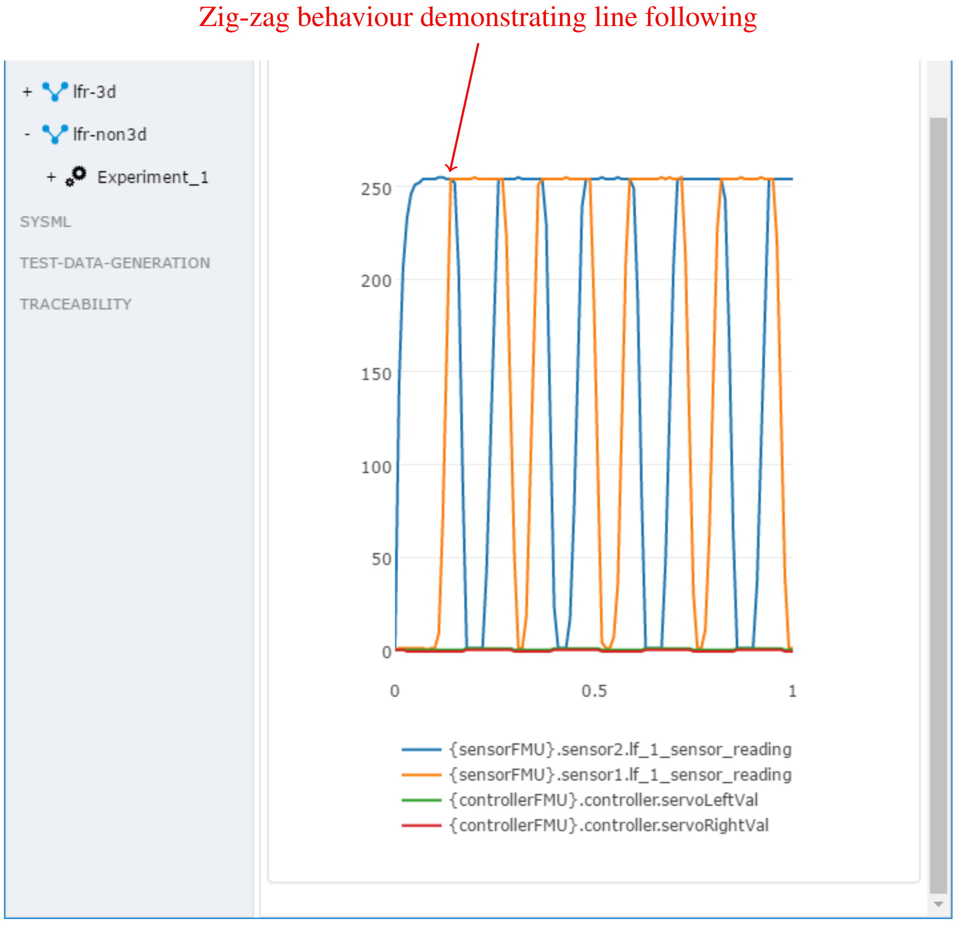
```
leftServo.setSpeed(0);
rightServo.setSpeed(0.8)
```

Changing the values will make the robot turn more or less. If both values are the same, the robot will move forwards or backwards in a straight line. If both values are exactly opposite (e.g. -1 and 1), the robot will turn on the spot.

Step 12. Add an `else if` clause to this statement to turn right if the left sensor is over white and the right sensor is over black.

Step 13. Add an `else if` clause to go forwards if both sensors are over black.

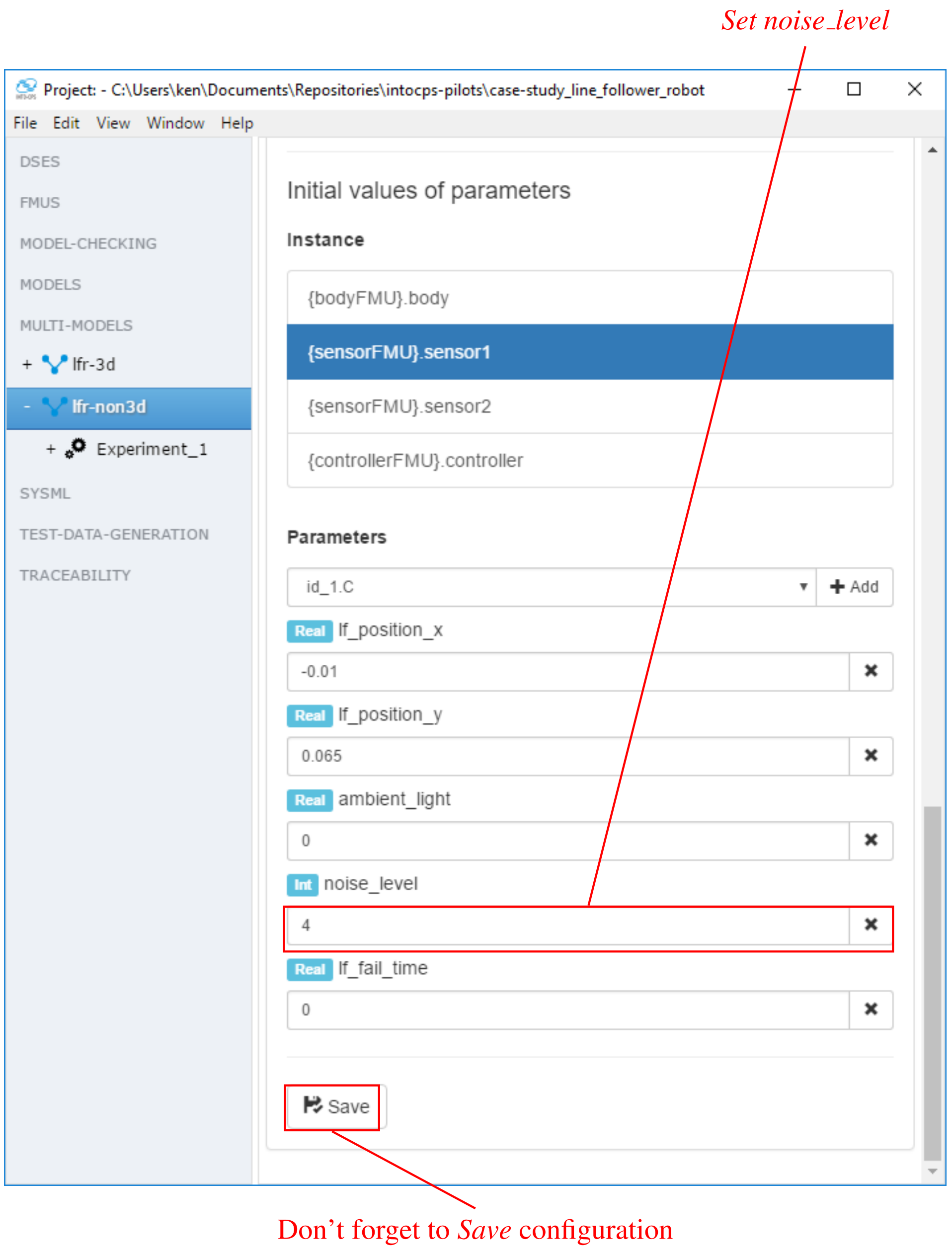
Step 14. Re-generate your FMU and check that the robot follows the line. If not, you can add `print` statements to your if-statement to check what conditions are being triggered.


Zig-zag behaviour demonstrating line following
+ lfr-3d
- lfr-non3d
+ Experiment_1
SYSML
TEST-DATA-GENERATION
TRACEABILITY
250
200
150
100
50
0
0
0.5
1
{sensorFMU}.sensor2.lf_1_sensor_reading
{sensorFMU}.sensor1.lf_1_sensor_reading
{controllerFMU}.controller.servoLeftVal
{controllerFMU}.controller.servoRightVal

4 Dealing with Noisy Data

The sensor model contains some *realistic* and *faulty* behaviours, which can be turned on or off from the INTO-CPS Application in the multi-model configuration. The first realistic behaviour is sensor noise. This occurs when converting analogue readings to a digital values, and results in readings that bounce up and down.

Step 15. Edit the *Initial values of parameters* for {*sensorFMU*}.*sensor1* and {*sensorFMU*}.*sensor2* and set the *noise_level* to *4*, where the range is (0,8).


Set noise_level
Project: - C:\Users\ken\Documents\Repositories\intocps-pilots\case-study_line_follower_robot
File Edit View Window Help
DSES
FMUS
MODEL-CHECKING
MODELS
MULTI-MODELS
+ lfr-3d
- lfr-non3d
+ Experiment_1
SYSML
TEST-DATA-GENERATION
TRACEABILITY
+ Add
Save
Don't forget to *Save* configuration

Initial values of parameters

Instance

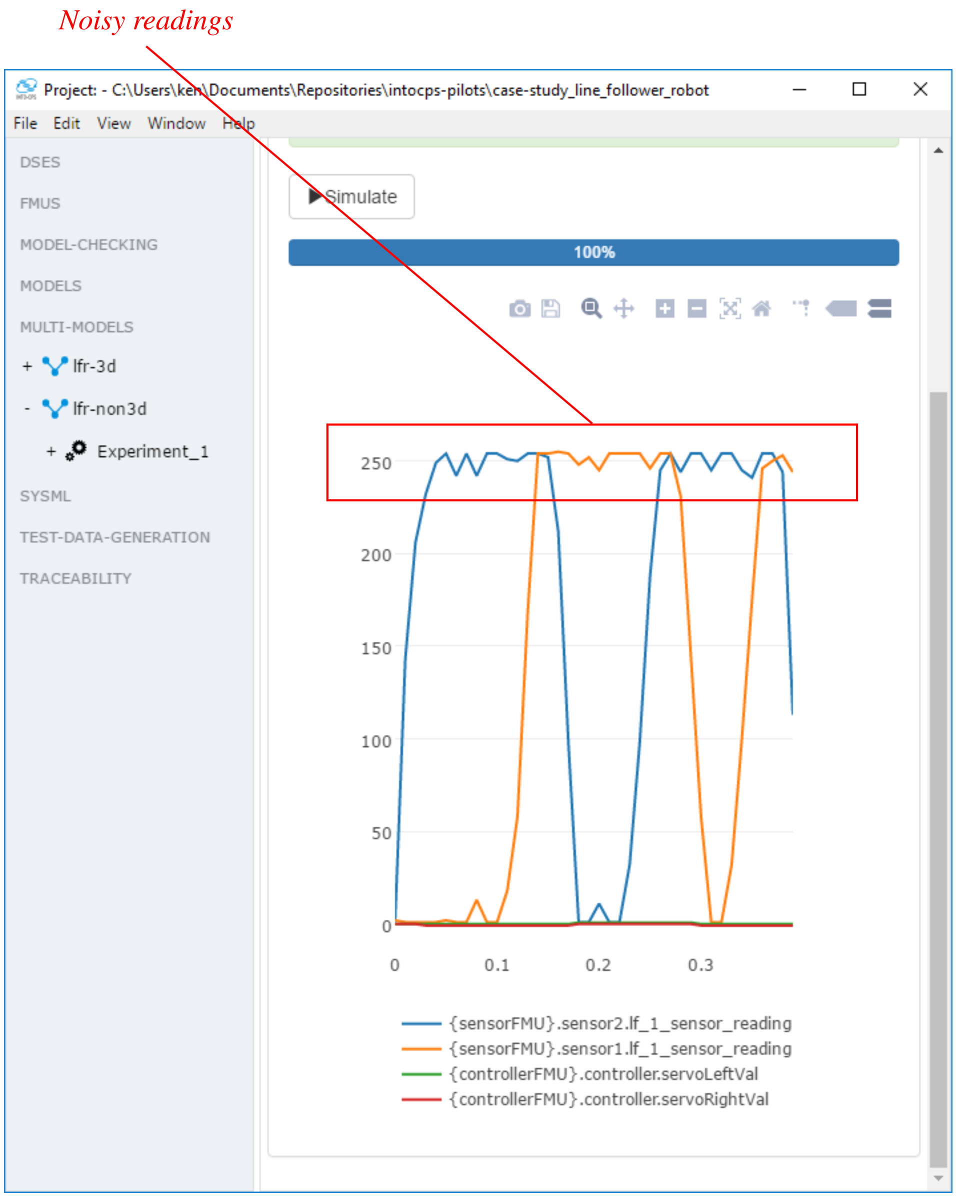
- {bodyFMU}.body
- {sensorFMU}.sensor1
- {sensorFMU}.sensor2
- {controllerFMU}.controller

Parameters

id_1.C

Real lf_position_x	-0.01
Real lf_position_y	0.065
Real ambient_light	0
Int noise_level	4
Real lf_fail_time	0

Step 16. Run the co-simulation and observe the curve of the sensor readings now shows a noisy signal.


Noisy readings
Project: - C:\Users\ken\Documents\Repositories\intocps-pilots\case-study_line_follower_robot
File Edit View Window Help
DSES
FMUS
MODEL-CHECKING
MODELS
MULTI-MODELS
lfr-3d
lfr-non3d
Experiment_1
SYSML
TEST-DATA-GENERATION
TRACEABILITY
Simulate
100%
250
200
150
100
50
0
0
0.1
0.2
0.3
{sensorFMU}.sensor2.lf_1_sensor_reading
{sensorFMU}.sensor1.lf_1_sensor_reading
{controllerFMU}.controller.servoLeftVal
{controllerFMU}.controller.servoRightVal

Step 17. To cope with this noise we will add a filter that provides a floating average of the last five readings. Create a file called `FilteredIRSensor.vdmrt` and populate it from the listing below. This class is defined as a *subclass* of `IRSensor` so it can be passed seamlessly to the `Controller` class. It encapsulates an `IRSensor` object, so it can intercept the readings and provide a filtered value:

```
class FilteredIRSensor is subclass of IRSensor

instance variables

-- sensor to be filtered
private sensor: IRSensor;

-- sequence of previous readings
private samples: seq of real
```

```
operations

-- constructor for FilteredIRSensor
public FilteredIRSensor: IRSensor ==> FilteredIRSensor
FilteredIRSensor(s) == (
  sensor := s;
  samples := []
);

public getReading: () ==> real
getReading() == (
  dcl reading: real := port.getValue()
  dcl average: real := 0;

  IO`printf("Average: %s of %s", [average, samples]);
  return average
);

public hasFailed: () ==> bool
hasFailed() ==
  return sensor.hasFailed();

end FilteredIRSensor
```

Step 18. As defined above, the `getReading` operation simply passes on a value of 0. Extend this operation (at the highlighted line) to store `reading` in the `samples` sequence and to calculate the *average* value of the sequence. The samples should store only the 5 newest values. *Hint: the ^ operator concatenates lists, **hd** yields the first item in a list, and **tl** yields the remainder of a list once the head is removed.*

Step 19. We have to modify the `System` class create `FilteredIRSensor` objects and pass them to the controller. Modify `System` as follows, then run your co-simulation again. You can check your filtered value with the information printed in the COE status window.

```
private leftSensor: IRSensor;
private rightSensor: IRSensor;

private leftSensorFiltered: FilteredIRSensor;
private rightSensorFiltered: FilteredIRSensor;

private leftServo: Servo;
private rightServo: Servo;
```

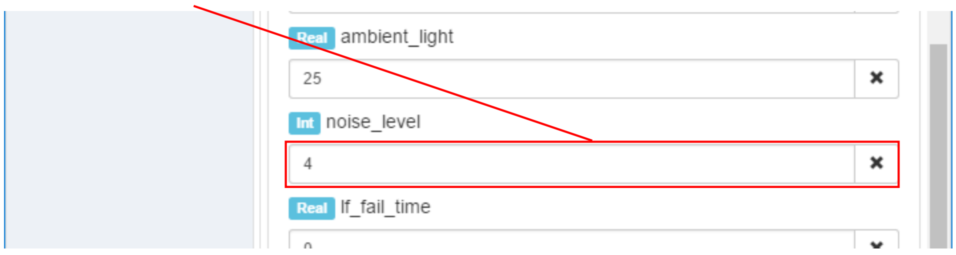
```
public System : () ==> System
System () ==
(
  -- create sensor and actuator objects
  leftSensor := new IRSensor(hwi.lfLeftVal, hwi.lfLeftFailFlag);
  rightSensor := new IRSensor(hwi.lfRightVal, hwi.lfRightFailFlag);
  leftFilter := new FilteredIRSensor(leftSensor);
  rightFilter := new FilteredIRSensor(rightSensor);
  leftServo := new Servo(hwi.servoLeftVal, false);
  rightServo := new Servo(hwi.servoRightVal, true);

  -- create controller object
  controller := new Controller(leftFilter, rightFilter, leftServo, rightServo);

  -- deploy objects
  cpu.deploy(controller,  "Controller");
  cpu.deploy(leftFilter,  "Left sensor");
  cpu.deploy(rightFilter, "Right sensor");
  cpu.deploy(leftServo,   "Left servo");
  cpu.deploy(rightServo,  "Right servo");
);
```

5 Dealing with Ambient Light

The second realistic behaviour is ambient light. The infrared sensor works by shining a beam of infrared light out and looking for a reflection, however the environment can contain a lot of infrared light, e.g. if it's a sunny day. This can make it difficult for the sensor to see black.

Step 20. Edit the *Initial values of parameters* for {*sensorFMU*}.*sensor1* and {*sensorFMU*}.*sensor2* and set the *ambient_light* to *25* (W), where the range is (0, 40).

Set noise_level



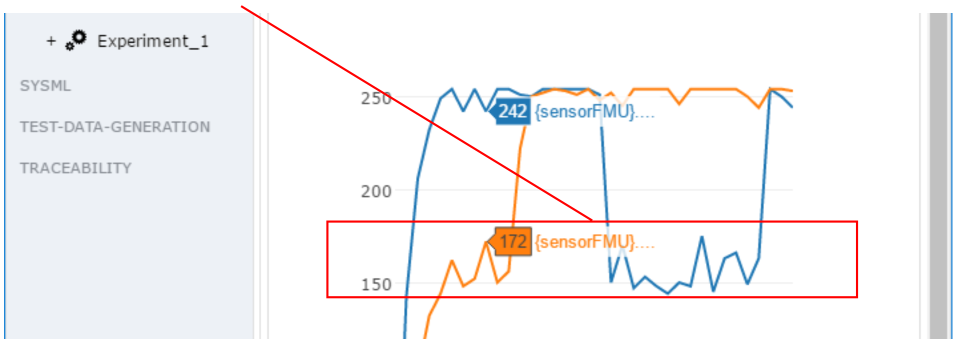
Real ambient_light: 25

Int noise_level: 4

Real lf_fail_time: 0

Step 21. Run the co-simulation and observe the increased black level.

Increased black level above halfway



Step 22. This can be overcome by adding modal behaviour to the control. Since we know that the left sensor begins over black, we can add a *calibration* mode that takes some readings to determine what value black is and uses this to determine the threshold. The controller can then switch to the existing logic in a *following*. Because the filtering delays the response of the sensor, we should also *wait* briefly before taking the calibration readings.

Step 23. Add the following type to the `Controller` class:

```
types
Mode = <WAIT> | <CALIBRATE> | <FOLLOW>
```

Step 24. Add the following instance variables

```
mode: Mode := <WAIT>;
samples: seq of real := [];
THRESHOLD: real := 150
```

Step 25. Modify the `Step` operation to include the modal behaviour described above. A simple way is to add a top-level `if` statement such as:

```
if mode = <WAIT> then ...
elseif mode = <CALIBRATE> then ...
elseif mode = <FOLLOW> then ...
```

The `<WAIT>` should do nothing until the simulation time is at 0.5 seconds (the current simulation in seconds time is given `time/1e9`), then change `mode` to `<CALIBRATE>`. Calibrate mode should add five readings from the `leftSensor` to the `samples` list, compute `threshold` as the average, then change mode to `<FOLLOW>`. The follow mode should contain your existing logic, but use `threshold` to determine if a sensor is seeing black and white.

Step 26. Add some `IO`printf` statements to your controller to indicate when it changes mode, then run the co-simulation and convince yourself the controller is working.

6 Dealing with Sensor Failure

The faulty behaviour in the sensor model is a complete failure, which will always produce a value of zero. It is possible to follow the line using a single sensor if this occurs. The parameter sets the time, in seconds, when the failure will occur (0 means never).

Step 27. Edit the *Initial values of parameters* for *{sensorFMU}.sensor1* and set the *lf_fail_time* to *2* (s). You can set it later but you have to simulate longer until it triggers. Run the co-simulation to see how the robot behaves after the failure.

Step 28. Extend your controller to add a new mode called `<SINGLE_FOLLOW>`. Your controller should switch to this mode if one of the sensors fails, then continue following the line using the remaining working sensor. If both sensors fail the robot should stop. The robot will need to move slower, which is a *degraded behaviour*: where a service is still offered but with lower performance. *Hint: you can follow the line with a single sensor by detecting the edge of the line – a change from black to white, or vice versa. Also slow means 0.3 power maximum.*

Step 29. Run the co-simulation again to check that the controller switches mode at the right time, and can now follow the line despite the failed sensor.

7 Additional Exercises

The suggested layout for the controller logic is not necessarily easily maintainable for larger controllers. Try refactoring the controller to make it more maintainable. Suggestions include:

- Moving the logic for each mode, and for mode changes, to auxiliary functions.
- Make an object-oriented version using the *State pattern*. Each mode is represented by an object that contains the logic. You can create an abstract mode class that provides access to the sensor and actuators, and empty `Enter`, `Step`, and `Exit` operations. Each mode is then defined as a subclass of this mode, overriding the operations as required. You can permit internal mode changes by allowing modes to return a new mode from their `Step` operation.

Tutorial — SysML for DSE

Overview

This tutorial will show you how to:

1. Declare the objective functions that will be used to assess the system under test
2. Define the parameters that will be varied during DSE along with their allowed values
3. Define which objectives will be used for ranking along with the preference for higher or lower values
4. Export the configuration for use during DSE

Requirements

This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- Modelio v3.4.1

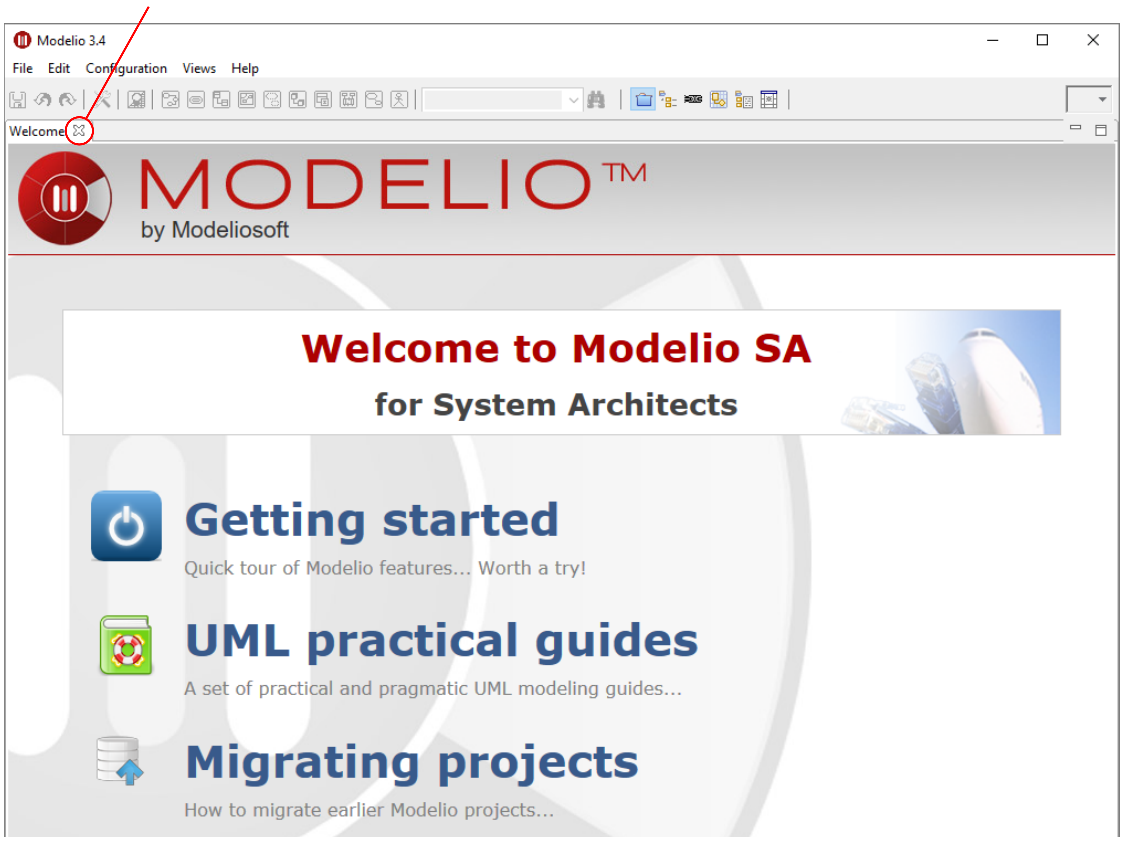
You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window > Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

You may need to update the INTO-CPS extension for Modelio to utilise DSE components. The extension can be downloaded from `http://forge.modelio.org/projects/intocps-modelio34/files`. Version 1.2.05 or later is required for DSE modelling.

1 Opening a Project

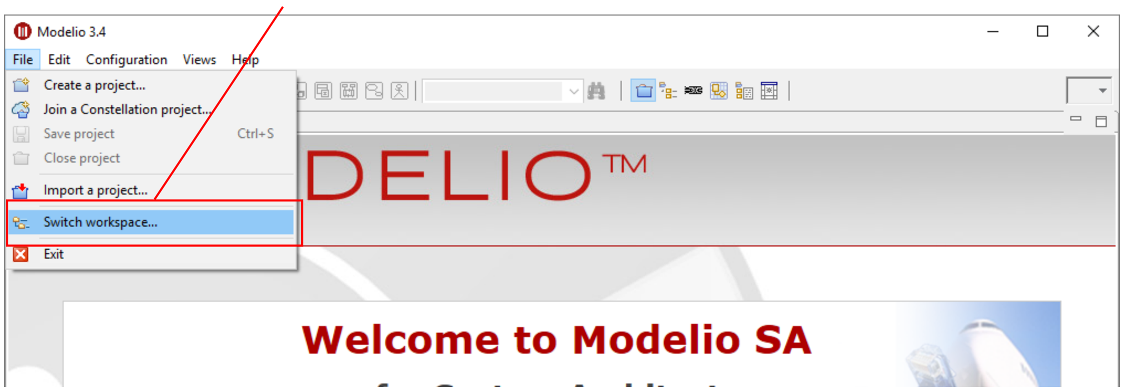
Step 1. Launch *Modelio*. On first loading, you may have to close the *Welcome* screen (you can bring it back with 'Help > Welcome' if you need)

Close *Welcome* screen

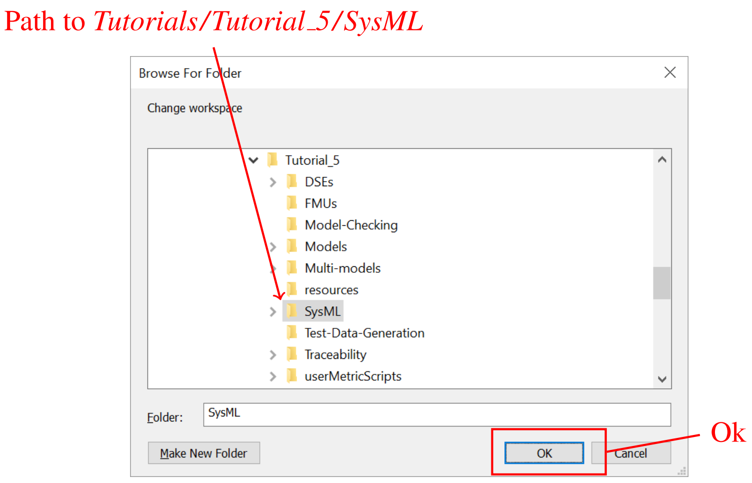

Modelio 3.4
File Edit Configuration Views Help
Welcome
MODELIO™
by Modeliosoft
Welcome to Modelio SA
for System Architects
Getting started
Quick tour of Modelio features... Worth a try!
UML practical guides
A set of practical and pragmatic UML modeling guides...
Migrating projects
How to migrate earlier Modelio projects...

Step 2. A workspace must be chosen, select 'File > Switch Workspace.

Switch Workspace

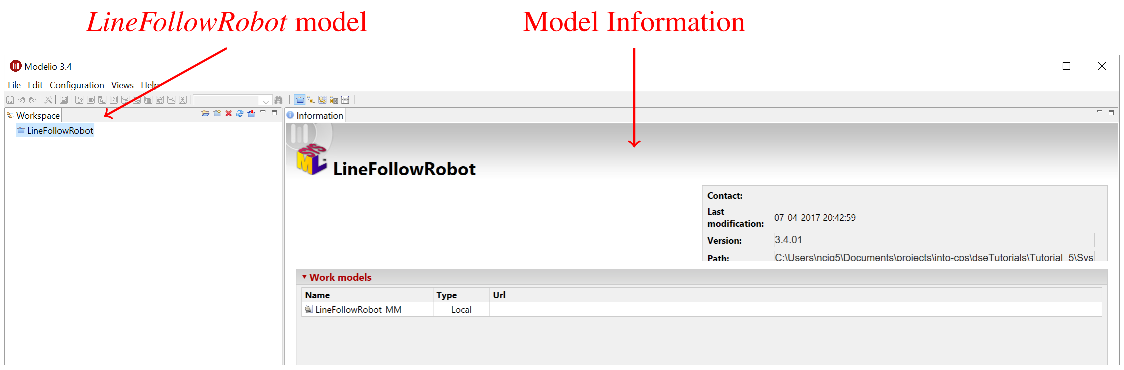


Step 3. Set the *Workspace* to the location of *Tutorials/Tutorial_5/SysML* and click *Ok*.



Folder: SysML

Step 4. Left-click on the *LineFollowRobot* model once on the left to see details of the model. Double-click the *LineFollowRobot* model to open the model.



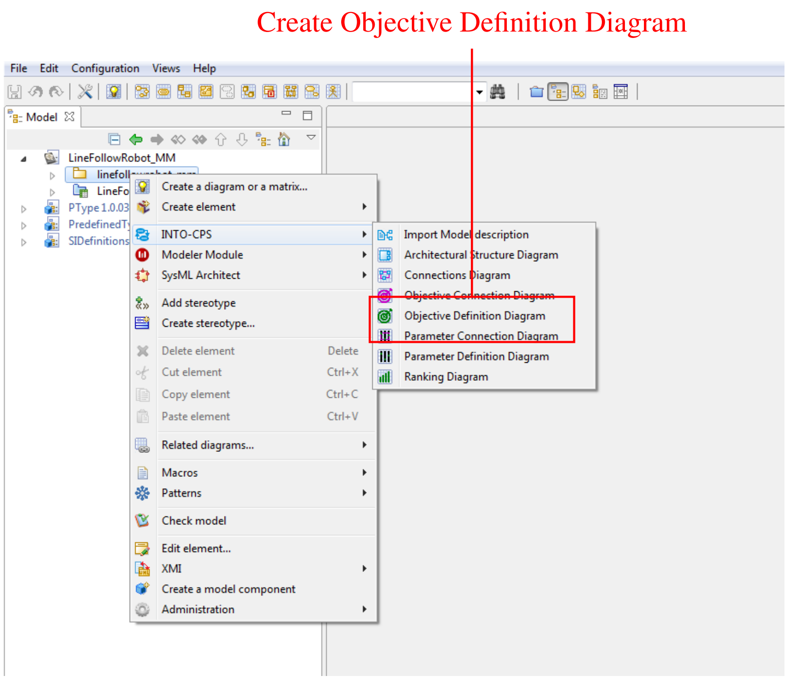
Contact:
Last modification: 07-04-2017 20:42:59
Version: 3.4.01
Path: C:\Users\ncig5\Documents\projects\into-cps\dseTutorials\Tutorial_5\Sys

Name	Type	Url
LineFollowRobot_MM	Local	

2 Defining Objectives

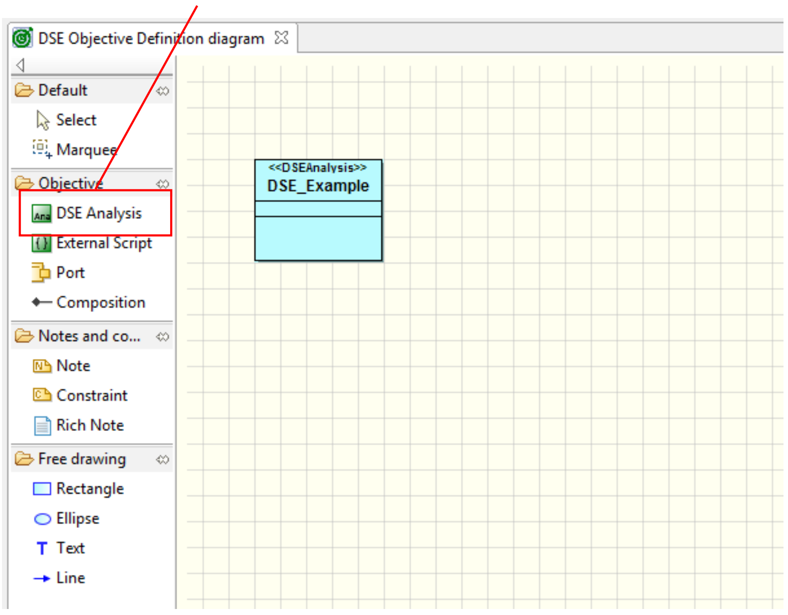
This section will describe the definition of objectives for use by DSE. Objectives are characterising measures of performance that may be used to determine the relative benefits of competing designs. Examples include, as we shall see, the time a robot takes to complete a circuit or the accuracy with which the robot is able to follow a path.
When defining the objectives we first describe them in terms of their name, the script file that will be used to compute them and the ports that will provide the data they require. Instances of these definitions are then created and the ports either filled with a static value or connected to data exchanged in the multi-model.

Step 5. Right click '$linefollowrobot\_mm$' in model outline. Select 'INTO-CPS > Objective Definition Diagram'.



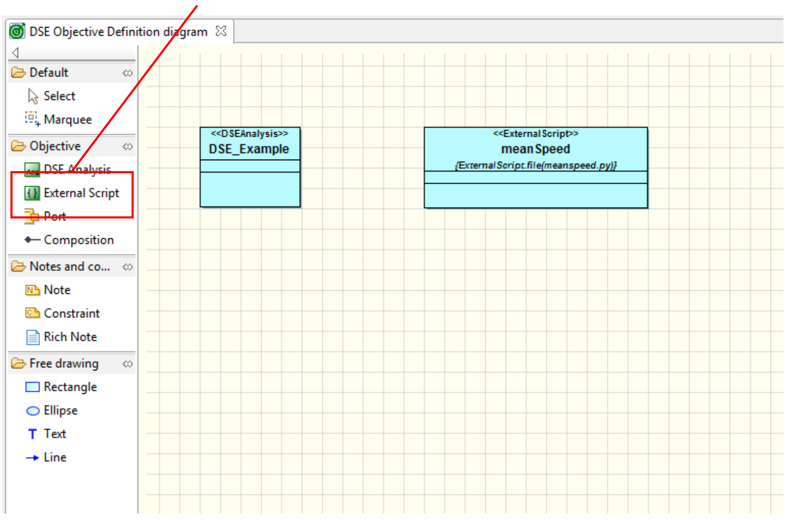
Step 6. From the *Toolbox*, select *DSE Analysis* and click on the empty diagram to create the DSE block. Double click the block and rename it '$DSE\_Example$'.

Create DSE Analysis Block

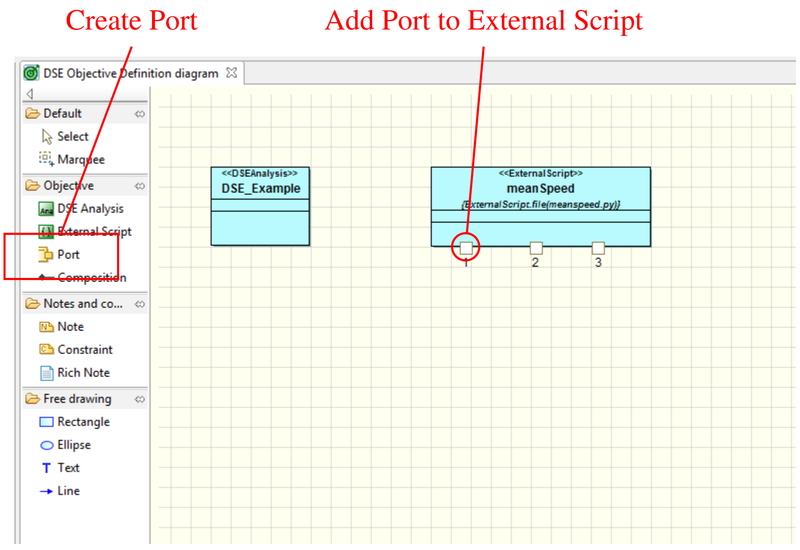


Step 7. From the *Toolbox*, select *External Script* and add it to the diagram. Double click the block that this creates and change its name to '$meanSpeed$'. On the left hand side of the editor window, select *ExternalScript* to show INTO-CPS DSE properties. Change {}file to '$meanspeed.py$', and close the editor window.

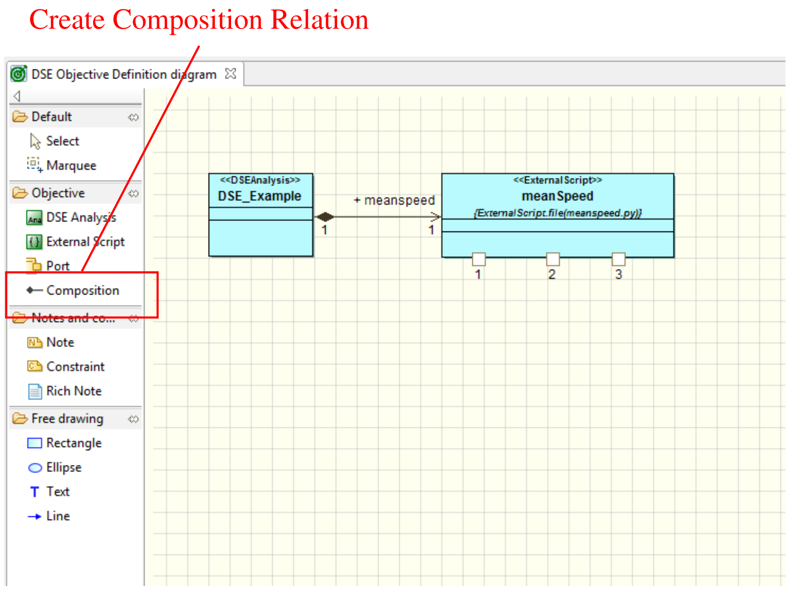
Create External Script Block



Step 8. So that objective scripts can later be connected, select the *Port* tool from the toolbox, and click on the External Script block to add a port to it. Click on the new port and change its name to '1' in the *Element* editor. Repeat this to add two more ports to the block, named '2' and '3'.



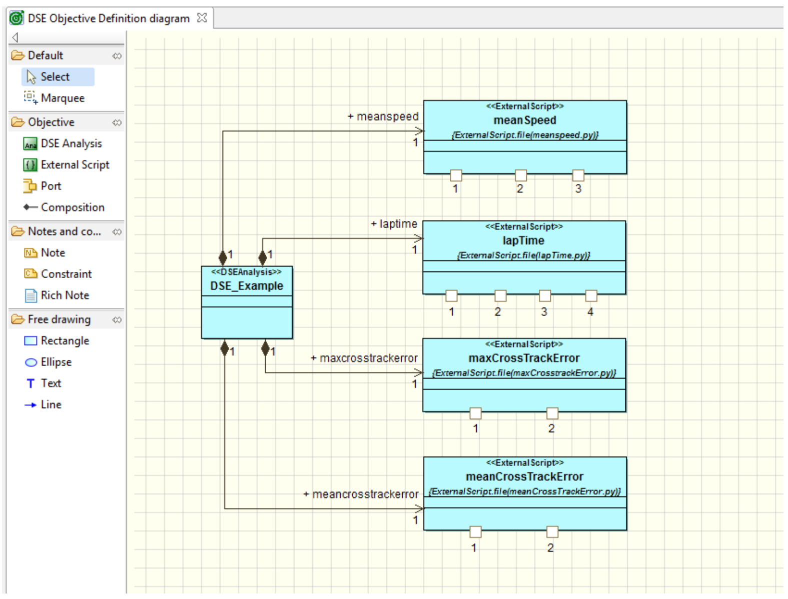
Step 9. To associate the external script with the DSE, select the *Composition* relation from the *Toolbox*, and click first on the '$DSE\_Exampleblock$', and then on the new '$meanSpeed$' block.



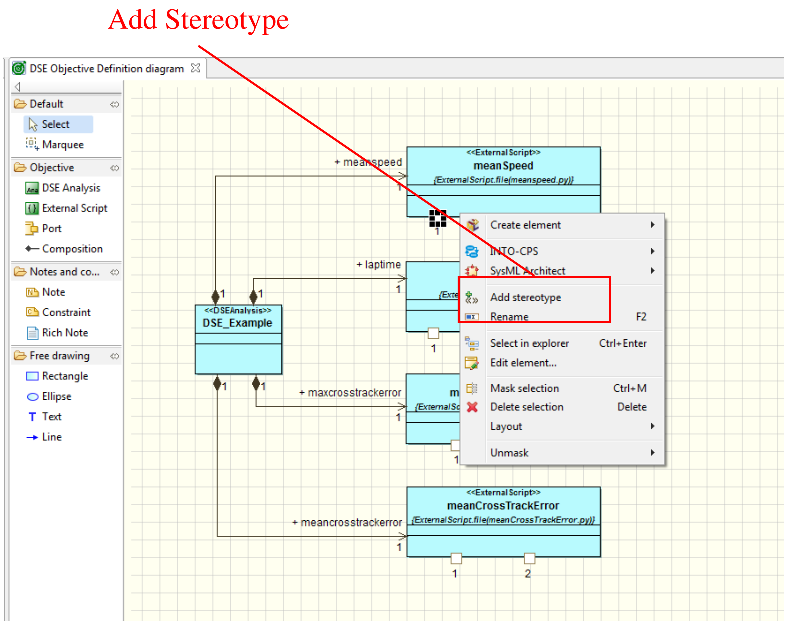
Step 10. Repeat Step 7. – Step 9. to associate the following scripts with the DSE:

- Name: ‘$lapTime$’, Script: ‘$lapTime.py$’, Ports: ‘$1, 2, 3, 4$’;
- Name: ‘$maxCrossTrackError$’, Script: ‘$maxCrosstrackError.py$’, Ports: ‘$1, 2$’;
- Name: ‘$meanCrossTrackError$’, Script: ‘$meanCrosstrackError.py$’, Ports: ‘$1, 2$’;

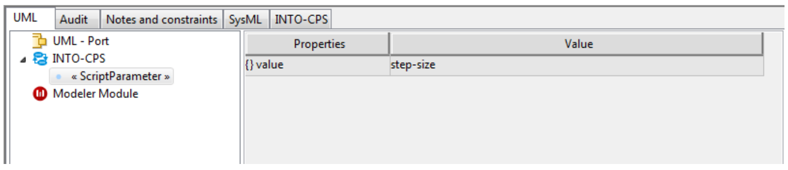
The model should now look like this:



Step 11. Some objective ports are associated with named values instead of model parts. An example of this is Port '1' of '$meanSpeed$'. Right click on the port and select *Add Stereotype*. Double click 'ScriptParameter' nested under 'INTO-CPS'.



Step 12. Double click the port and select 'INTO-CPS > ScriptParameter'. Change {}value to 'step-size' and close the editor window.



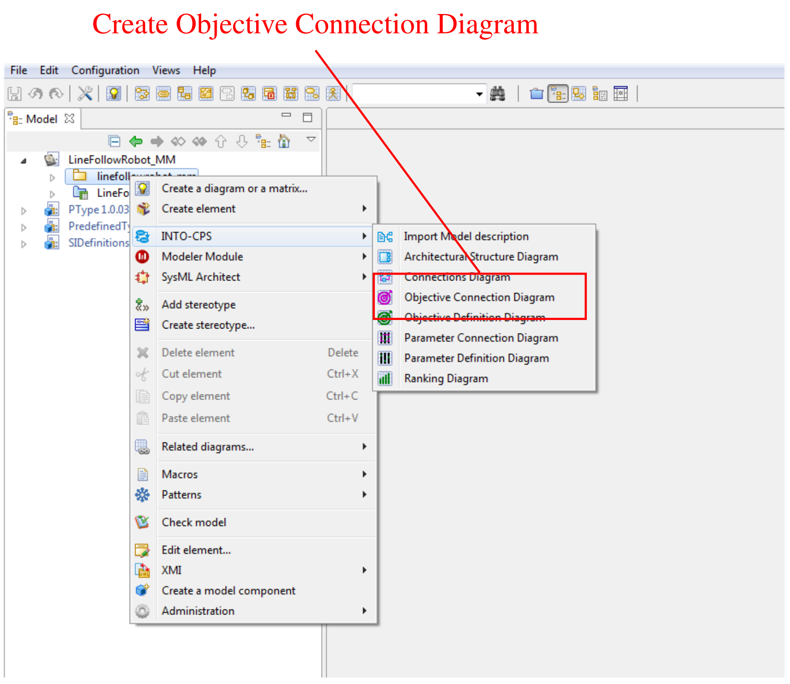
Properties	Value
{} value	step-size

Step 13. Repeat Step 11. & Step 12. to assign the following script parameters:

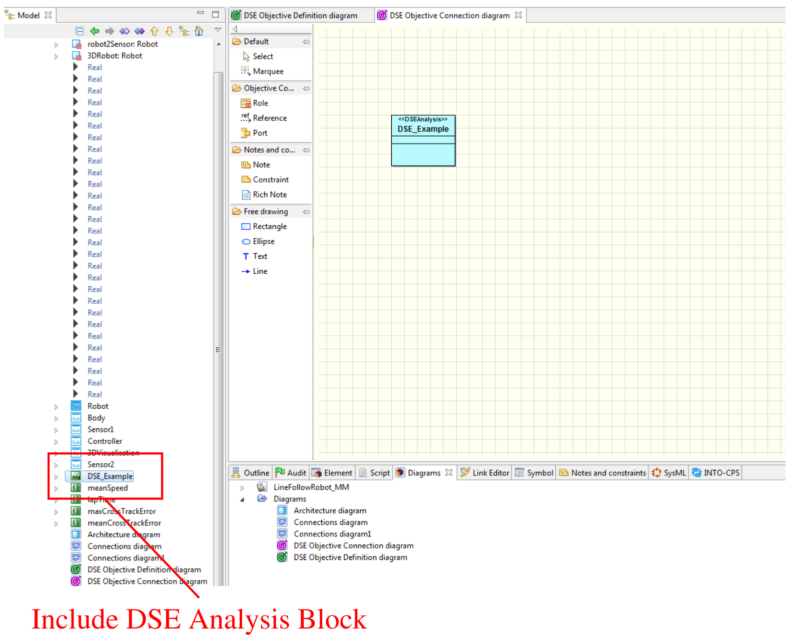
- Port '1' of '$lapTime$': Value '$time$';
- Port '4' of '$lapTime$': Value '$studentMap$';

3 Connecting Objectives

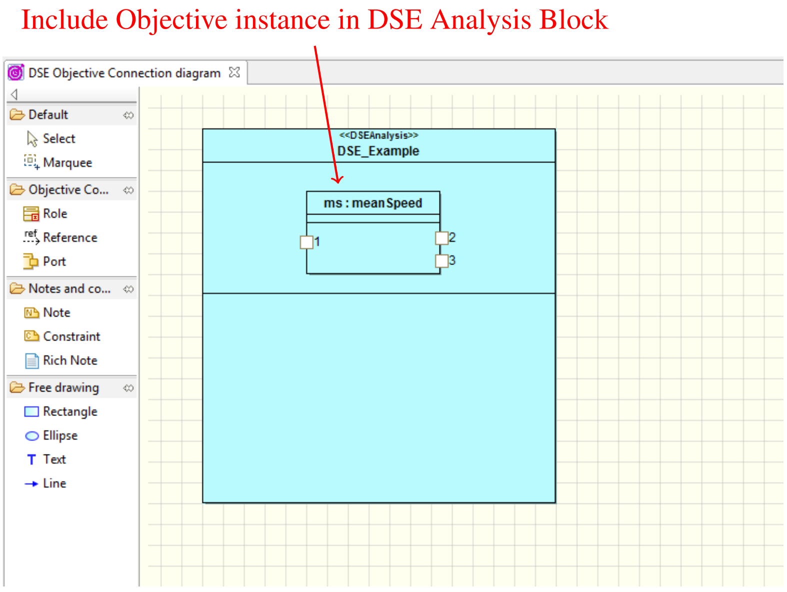
Step 14. Right click ‘$linefollowrobot\_mm$’ in model outline. Select ‘INTO-CPS > Objective Connection Diagram’.



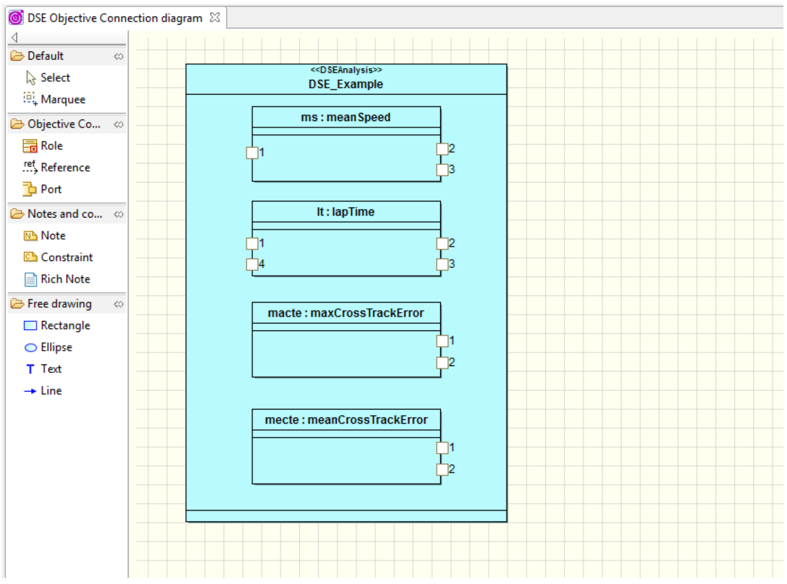
Step 15. Find the ‘$DSE\_Example$’ block in the model outline and drag it onto the empty diagram to add the DSE block created in Part 2.



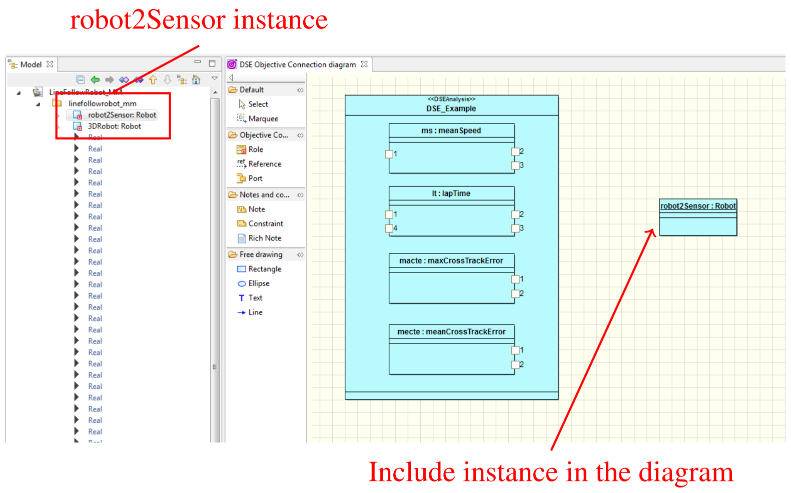
Step 16. Find the '$meanSpeed$' block in the model outline and drag it onto the DSE example block now visible on the diagram. Double click the block that this creates and name the instance 'ms'.



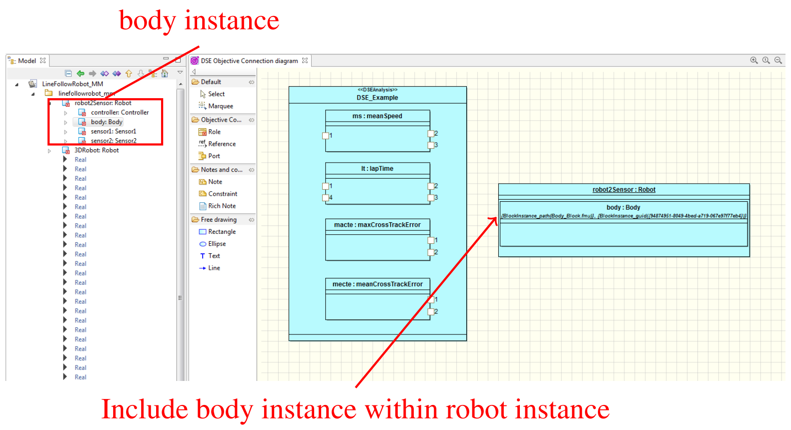
Step 17. Repeat Step 16. for '$lapTime$', '$maxCrossTrackError$', and '$meanCrossTrackError$', using the instance labels 'lt', '$mecte$', and '$macte$' respectively.



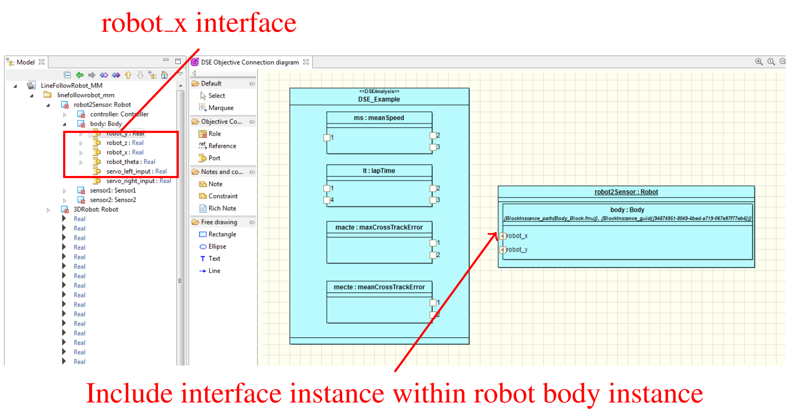
Step 18. To link Objectives to the multi-model, find the '$robot2Sensor$' instance in the model overview and drag it onto the diagram to include it in the DSE.
NB drag the block onto empty space, not onto the $DSE\_Example$ block.



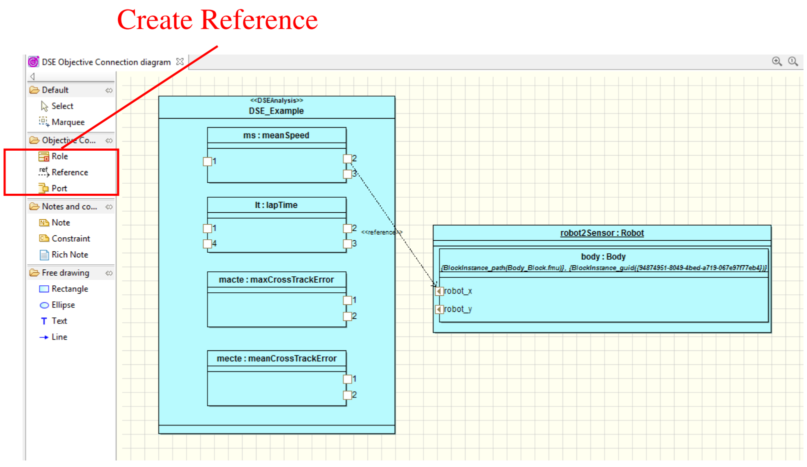
Step 19. Expand '$robot2Sensor$' in the model overview and drag the nested instance of '$body$' onto the '$robot2sensor$' instance just created.



Step 20. Expand the instance of ‘$body$’ (nested under ‘$robot2sensor$’) in the model overview and drag the ‘$robot\_x$’ port onto the body instance just created. Repeat this for the ‘$robot\_y$’ port.



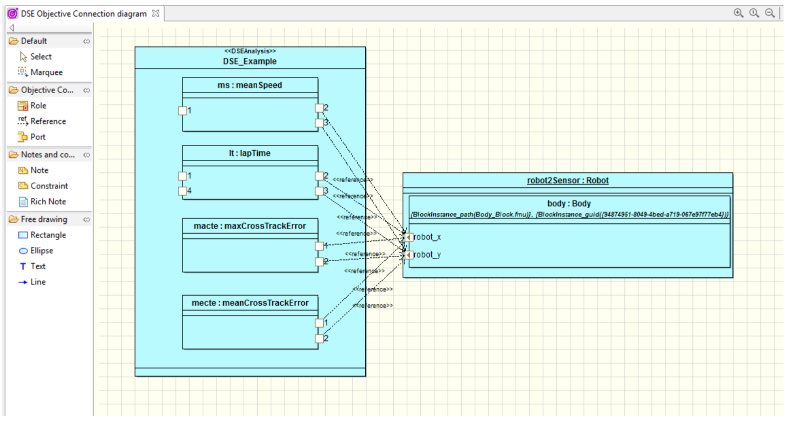
Step 21. Finally, select the *Reference* tool from the *Toolbox* and link port ‘2’ from ‘ms’ to port ‘$robot\_x$’ from ‘$body$’.



Step 22. Repeat Step 21. to link:

- '$ms > 3$' to '$body > robot\_y$';
- '$lt > 2$' to '$body > robot\_x$';
- '$lt > 3$' to '$body > robot\_y$';
- '$mecte > 1$' to '$body > robot\_x$';
- '$mecte > 2$' to '$body > robot\_y$';
- '$macte > 1$' to '$body > robot\_x$';
- '$macte > 2$' to '$body > robot\_y$';

The model should now look like this:



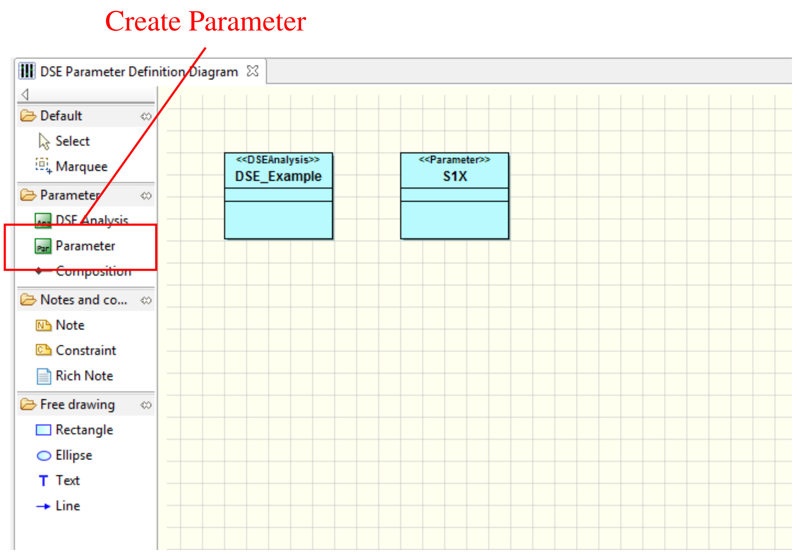
4 Defining Parameters

This section will present how to define the parameters that will be varied during a DSE. In the first part we define the parameters in terms of their name and a set of values that we wish to test. The second step here is to connect the defined parameters to the parameters of the FMUs in the multi-model that we actually want to vary.
If we multiply the cardinality of the set of values for each parameter then we can find the size of the design space. It is important to keep the design space size in mind since this, along with the time require to run each simulation, may be used to determine if a DSE will complete in a reasonable time.

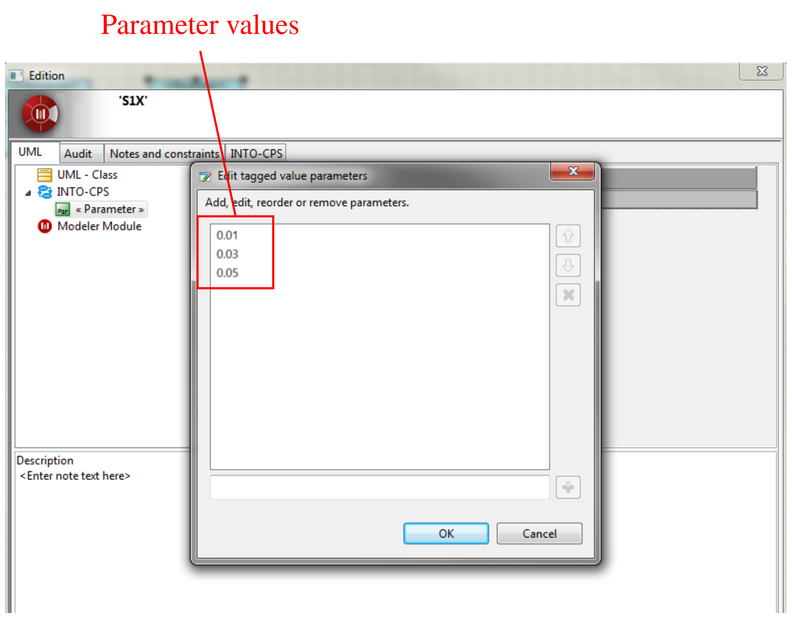
Step 23. Right click '$linefollowrobot\_mm$' in model outline. Select 'INTO-CPS > Parameter Definition Diagram'.

Step 24. Find the '$DSE\_Example$' block in the model outline and drag it onto the empty diagram to add the DSE block created in Part 2.

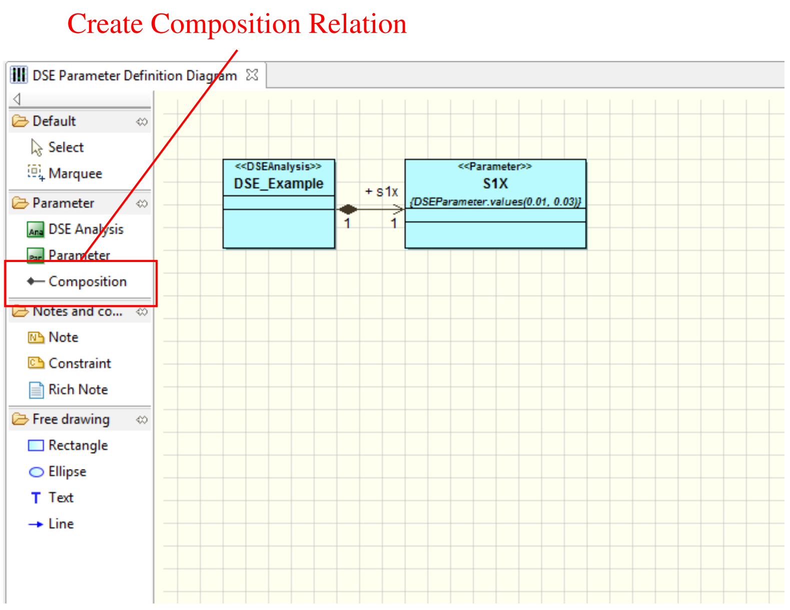
Step 25. From the *Toolbox*, select *Parameter* and add it to the diagram. Rename the new Parameter block '$S1X$'.



Step 26. Double click the Parameter block and select 'INTO-CPS > Parameter'. Click next to {}values to add parameter values. Enter the value '0.01' and click '+' to submit the value. Repeat this to add the values '0.03' and '0.05', then click *OK*. Click *Close* to return to the diagram.



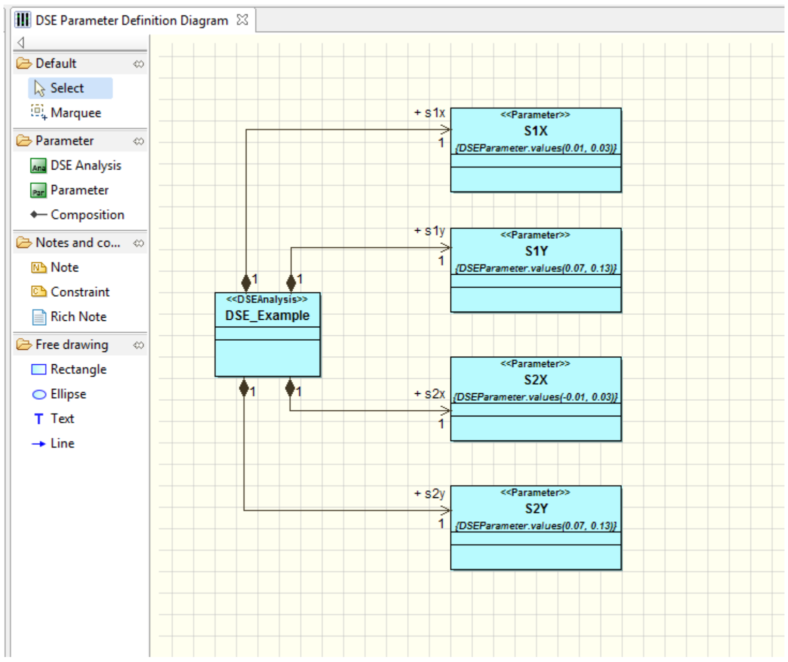
Step 27. To associate the parameter with the DSE, select the *Composition* tool from the *Toolbox*, click first on the '$DSE\_Exampleblock$', and then on the new '$S1X$' block.



Step 28. Repeat Step 25. – Step 27. to associate the following parameters with the DSE:

- Name: '$S1Y$', Values: '$0.01, 0.07, 0.13$';
- Name: '$S2X$', Values: '$-0.01, -0.03, -0.05$';
- Name: '$S2Y$', Values: '$0.01, 0.07, 0.13$';

The model should now look like this:

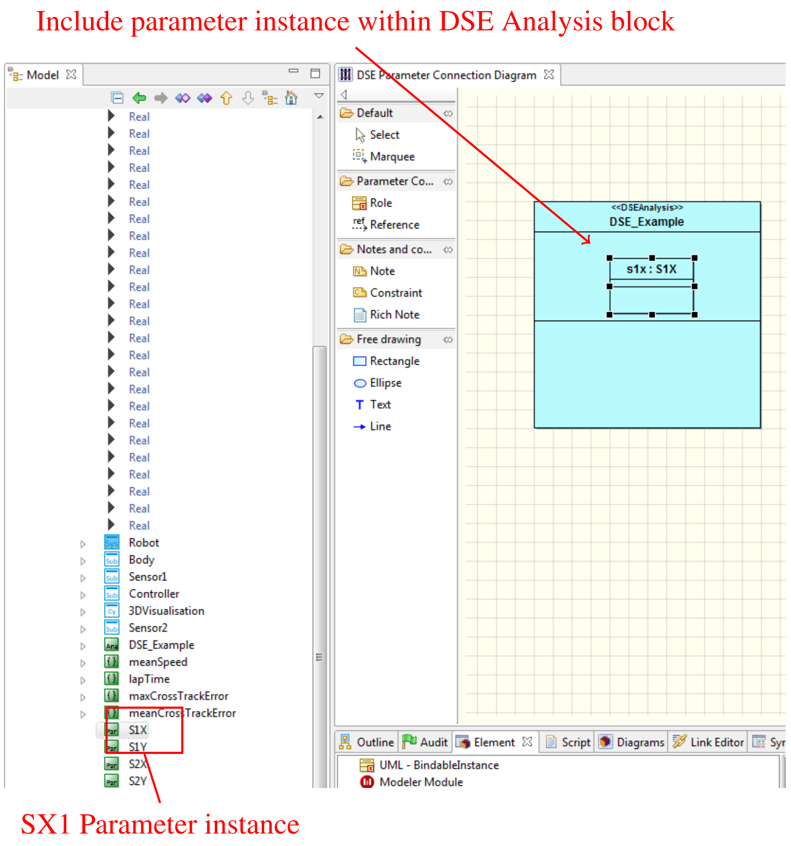


5 Connecting Parameters

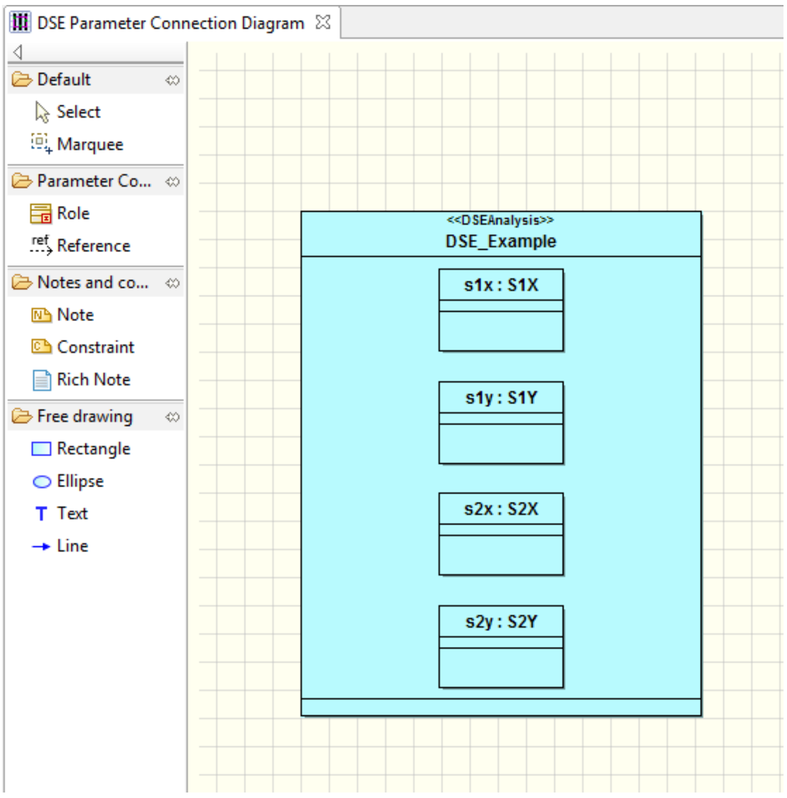
Step 29. Right click ‘$linefollowrobot\_mm$’ in model outline. Select ‘INTO-CPS > Parameter Connection Diagram’.

Step 30. Find the ‘$DSE\_Example$’ block in the model outline and drag it onto the empty diagram to add the DSE block created in Part 2.

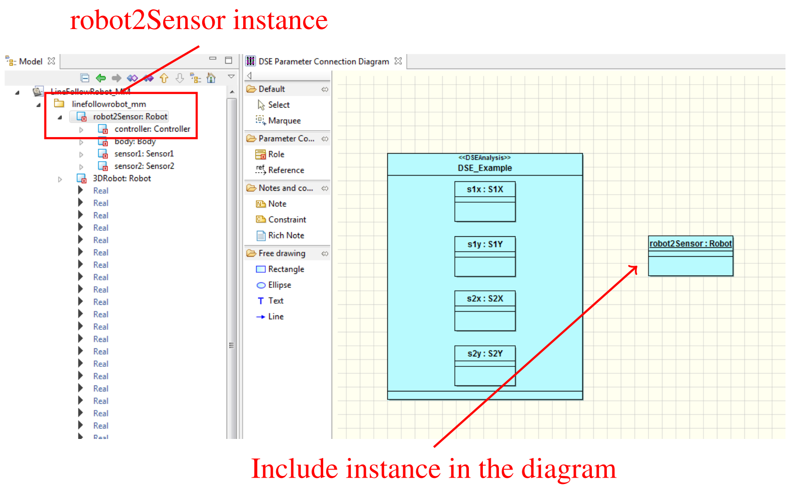
Step 31. Find the ‘$S1X$’ block in the model outline and drag it onto the DSE example block now visible on the diagram. Double click the block that this creates and name the instance ‘$s1x$’.



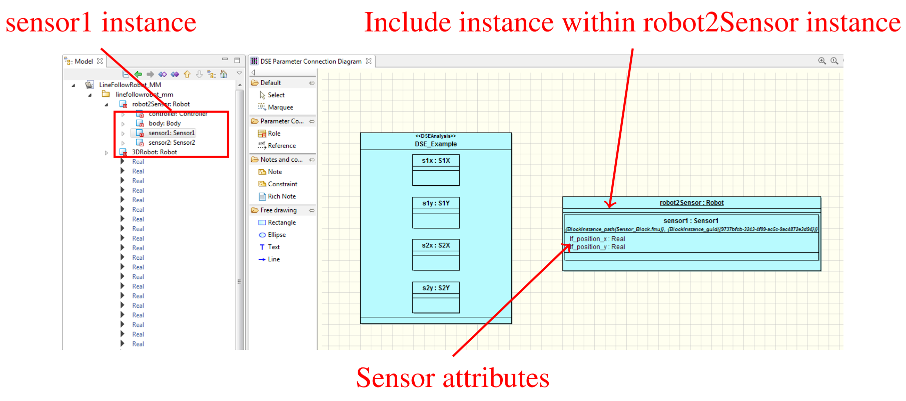
Step 32. Repeat Step 31. for '$S1Y$', '$S2X$', and '$S2Y$', using the instance labels '$s1y$', '$s2x$', and '$s2y$' respectively.



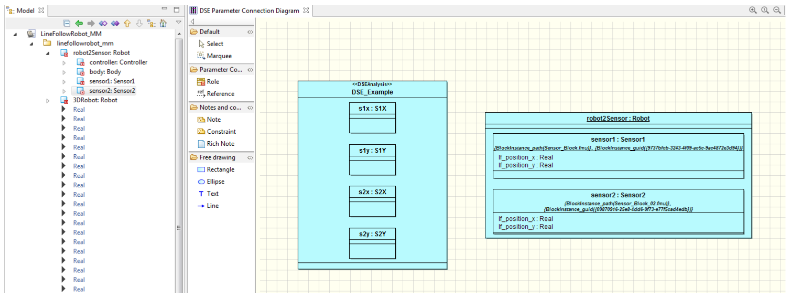
Step 33. To link Parameters to the multi-model, find the '$robot2Sensor$' instance in the model overview and drag it onto the diagram to include it in the DSE
(NB drag the block onto empty space, not onto the DSE_Example block).



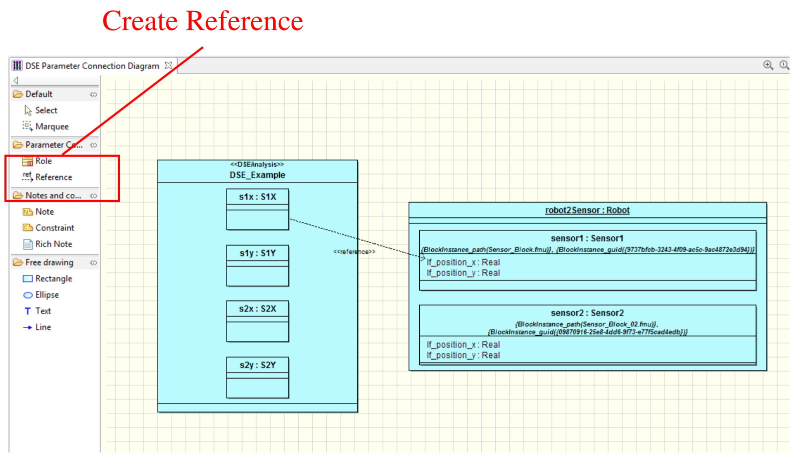
Step 34. Expand '$robot2Sensor$' in the model overview and drag the nested instance of '$sensor1$' onto the '$robot2sensor$' instance just created. This block should include the attributes '$lf\_position\_x$' and '$lf\_position\_y$'.



Step 35. Repeat Step 34. to include '$sensor2$' in the '$robot2sensor$' block.



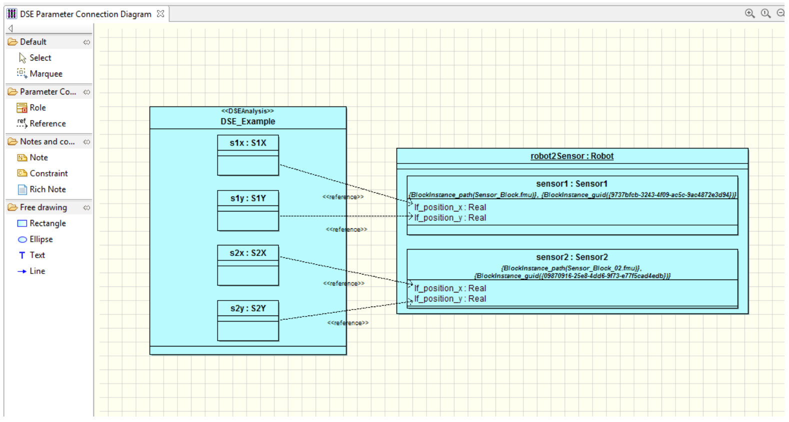
Step 36. Finally, select the *Reference* tool from the *Toolbox* and link parameter '$s1x$' to attribute '$lf\_position\_x$' from '$sensor1$'.



Step 37. Repeat Step 36. to link:

- '$s1y$' to '$sensor1 > lf\_position\_y$';
- '$s2x$' to '$sensor2 > lf\_position\_x$';
- '$s2y$' to '$sensor2 > lf\_position\_y$';

The model should now look like this:



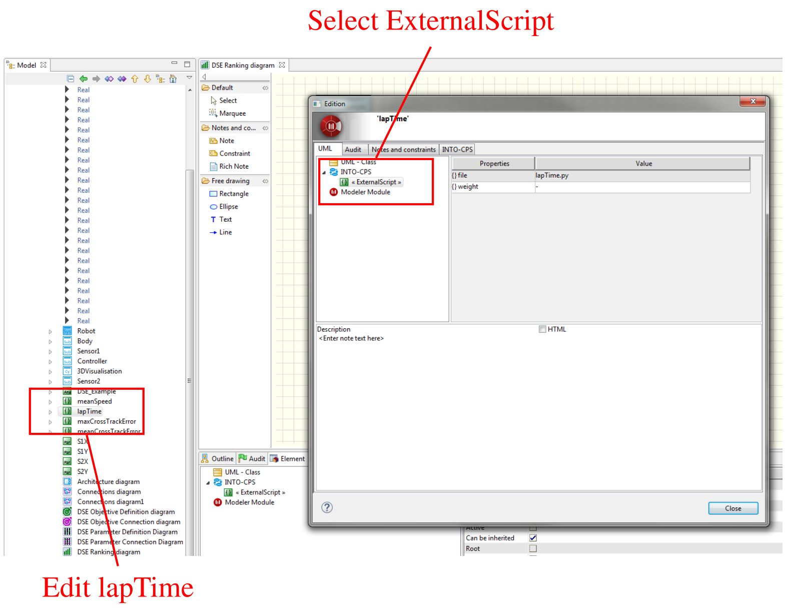
6 Ranking Results

In DSE all the designs are essentially competing to be considered the best and in this section will present how to define the way in which competing are compared. Here we are going to declare which of the objectives should be used to compare competing designs and whether there is a preference for lower or higher values for each of the objectives.

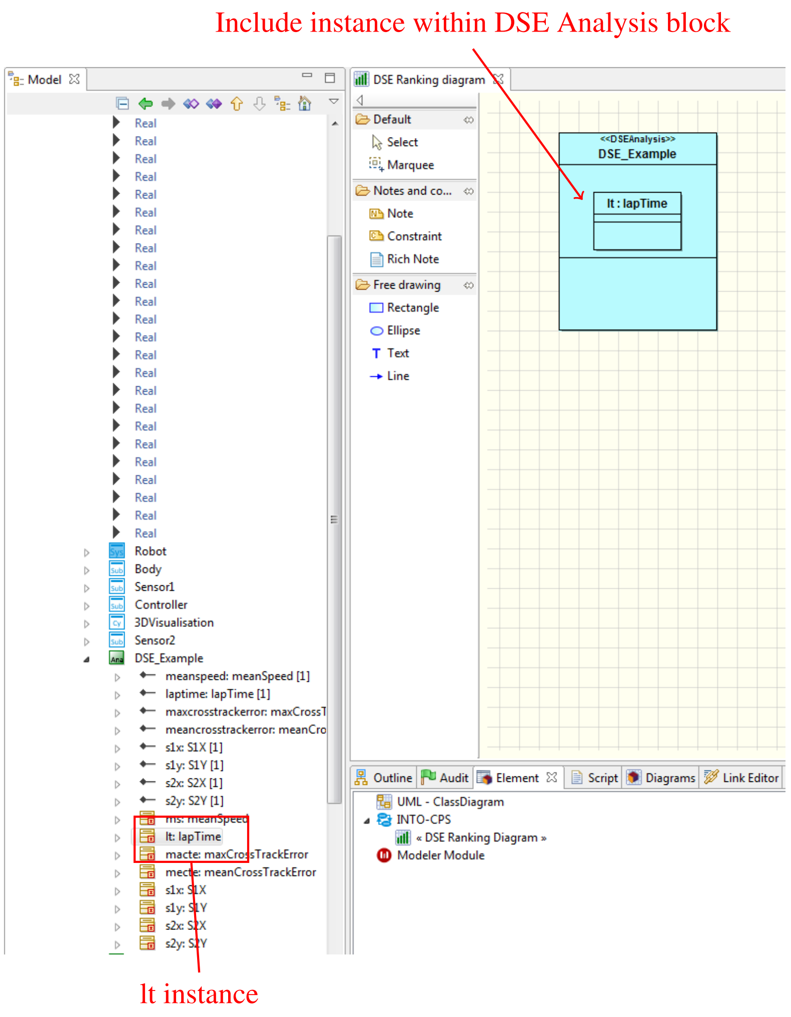
Step 38. Right click ‘$linefollowrobot\_mm$’ in model outline. Select ‘INTO-CPS > Ranking Diagram’.

Step 39. Find the ‘$DSE\_Example$’ block in the model outline and drag it onto the empty diagram to add the DSE block created in Part 2.

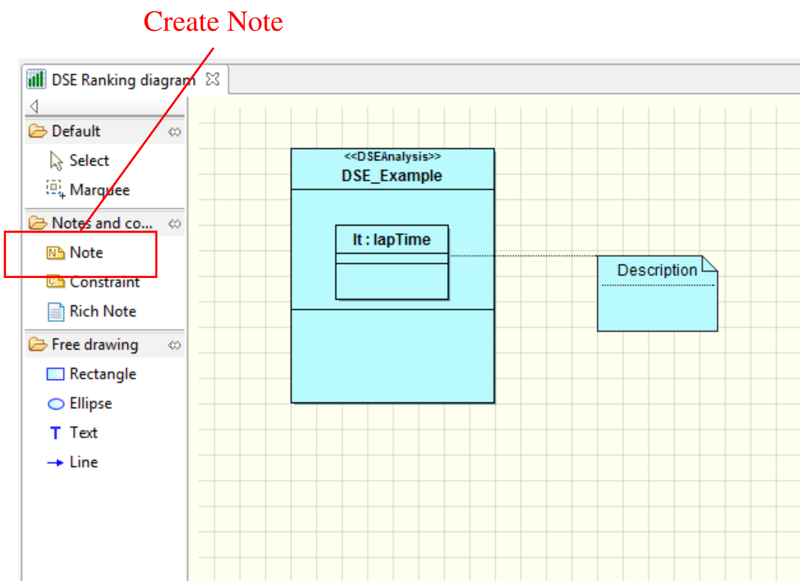
Step 40. Double click on the ‘$lapTime$’ objective in the model outline. Click ‘ExternalScript’ nested under ‘INTO-CPS’. Assign the value ‘$-$’ to {}weight and close the editor window.



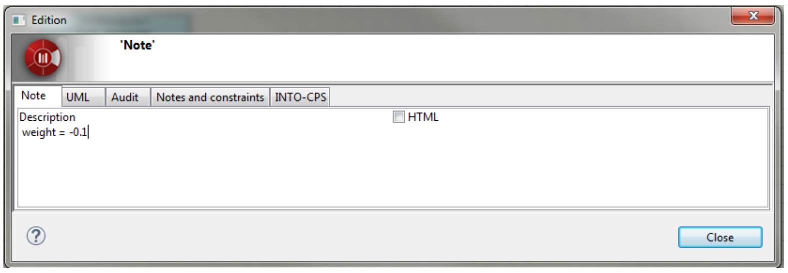
Step 41. Expand the ‘$DSE\_Example$’ block in the model outline and drag the nested instance ‘lt’ onto the DSE example block now visible on the diagram.



Step 42. Select the Note tool from the toolbox and click on the 'lt' block just created. Click on a clear space in the diagram to create the description block.



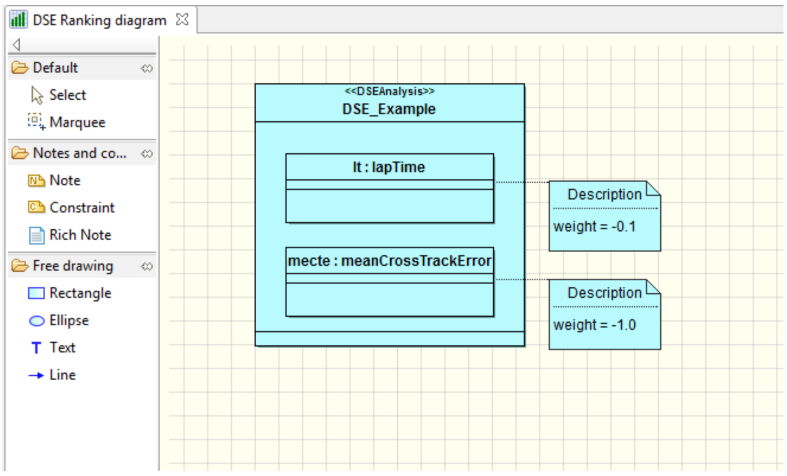
Step 43. Double click the description block to edit the note text. Deselect the 'HTML' option and paste the text '$weight = -1.0$'. Close the description window to update the diagram.



Description
weight = -0.1

Step 44. Repeat Step 40. – Step 43. to set the {}weight property of '$meanCrossTrackError$' and add the '$mecte$' objective instance with the description '$weight = -1.0$'.

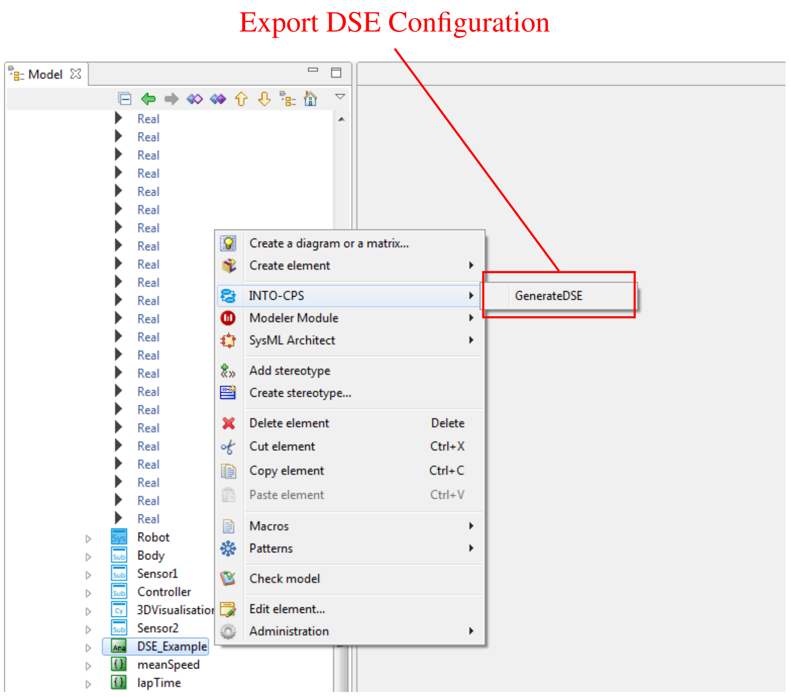
The model should now look like this:



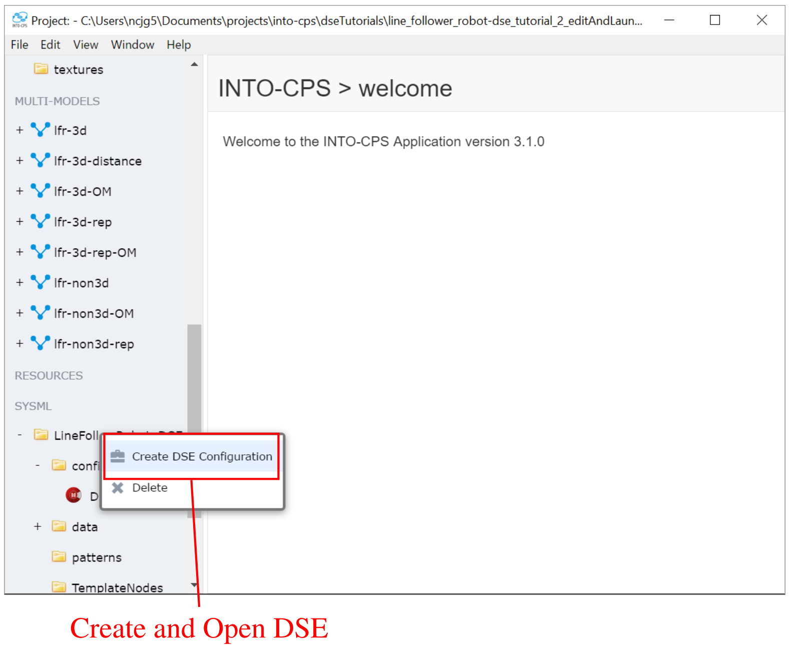
7 Exporting DSE Configuraton

This section will outline the steps necessary to generate a DSE configuration in Modelio and include the file to the INTO-CPS app.

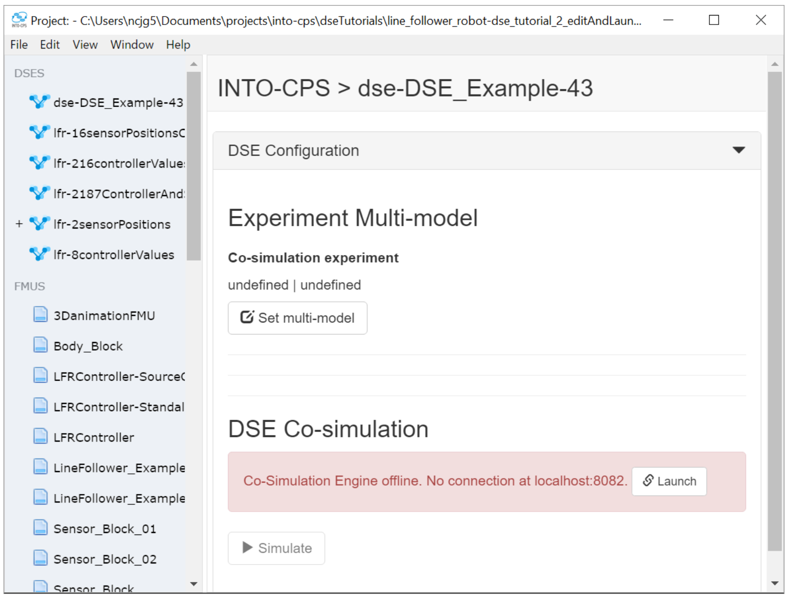
Step 45. Find the '*DSE_Example*' block in the model outline and right click. Select 'INTO-CPS > GenerateDSE'. Click *Export* then *OK*.



In the INTO-CPS app, this configuration file can be found in 'SysML > LineFollowerRobot > config. Right click the configuration file and select 'Create DSE Configuration.



The configuration will be moved to the DSE directory and opened. The DSE can now be viewed, edited, or started. This will be covered in Tutorial 6.


Project: - C:\Users\ncjg5\Documents\projects\into-cps\dseTutorials\line_follower_robot-dse_tutorial_2_editAndLaun...

INTO-CPS > dse-DSE_Example-43

DSE Configuration

Experiment Multi-model

Co-simulation experiment

undefined | undefined

DSE Co-simulation

Co-Simulation Engine offline. No connection at localhost:8082.

Tutorial — Editing and Launching a DSE in the App

Overview

This tutorial will show you how to:

1. Open a DSE configuration in the INTO-CPS app.
2. Edit the configuration in the app
3. Launch a DSE from the app
4. Read the results and find the details for each simulation

Requirements

This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- INTO-CPS app 3.10 or above
- Python 2.7 with numpy and matplotlib
- DSE scripts 0.2.0 or above

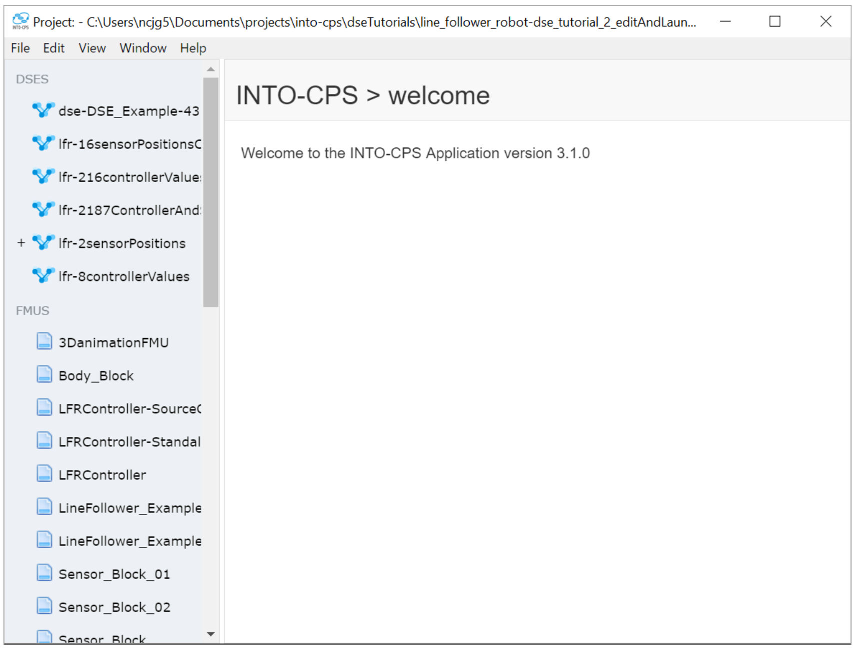
You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window > Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

1 Preparing the Workspace

This tutorial requires the contents of Tutorial6.zip to be unzipped and placed in your desired location. Please note that the DSE scripts currently do not handle path names including spaces, so please ensure that the path to where Tutorial6.zip extract does not include spaces.

2 Opening a DSE Configuration

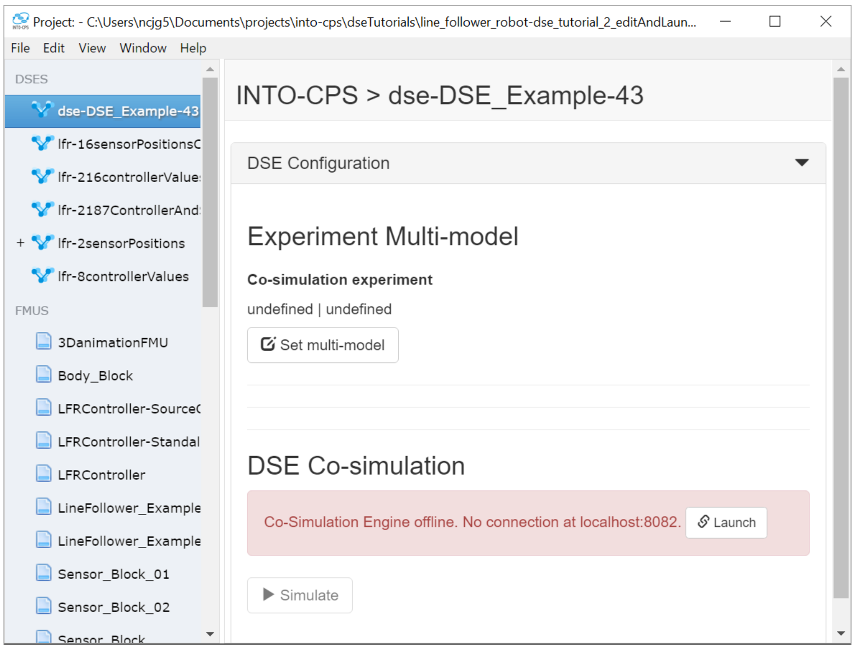
After opening the app, you will need to move to the Tutorial_6 workspace that you unzipped in the previous step. In the app select *File > Switch Workspace* and navigate to the your *Tutorial_6* workspace. With the workspace open you should be presented with the app welcome screen as below.


Project: - C:\Users\ncjg5\Documents\projects\into-cps\dseTutorials\line_follower_robot-dse_tutorial_2_editAndLaun...

INTO-CPS > welcome

Welcome to the INTO-CPS Application version 3.1.0

The configuration we will used in the tutorial is called dse-DSE_Example-43, which was the name given to it when the app imports the DSE configuration that was exported from Modelio in Tutorial 5. To open the configuration, double click on its name in the DSEs section. Please note that the dse name ’dse-DSE_Example-43’ is the one that will be found in the work space distributed with this tutorial, if you have exported a DSE configuration from the app yourself then the folder name will likely be different.


Project: - C:\Users\ncjg5\Documents\projects\into-cps\dseTutorials\line_follower_robot-dse_tutorial_2_editAndLaun...

INTO-CPS > dse-DSE_Example-43

DSE Configuration

Experiment Multi-model

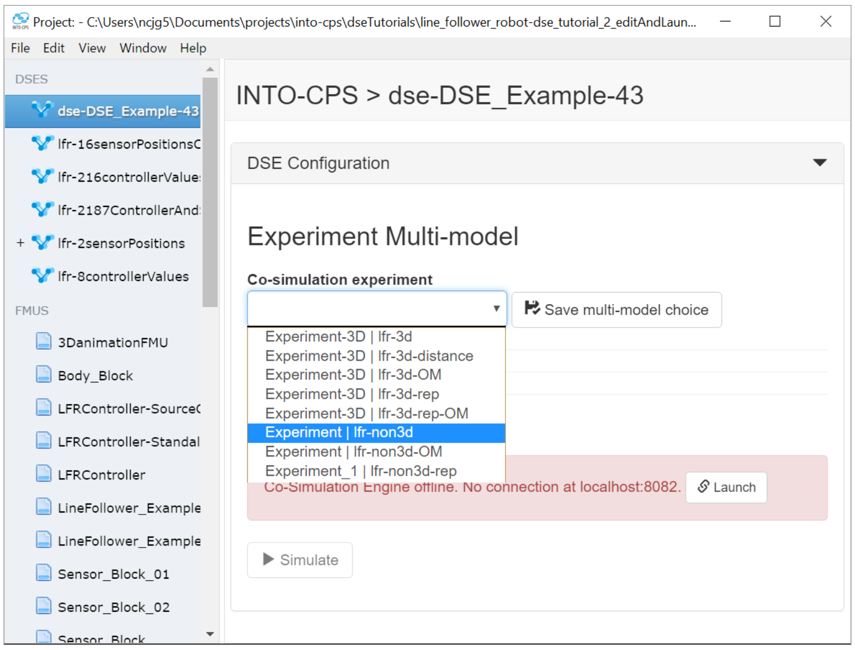
Co-simulation experiment

undefined | undefined

DSE Co-simulation

Co-Simulation Engine offline. No connection at localhost:8082.

A DSE configuration can only work in concert with an existing multi-model since it only describes the changes that should be made to the model, it does not describe the connections or FMUs. So the first thing we need to do is select the multi-model the DSE scripts will use. As below, click on Set multimodal and then scroll down to select *Experiment/lfr-non3d*. Then click on *Save multi-model choice*.


Project: - C:\Users\ncjg5\Documents\projects\into-cps\dseTutorials\line_follower_robot-dse_tutorial_2_editAndLaun...

File Edit View Window Help

DSES
dse-DSE_Example-43
lfr-16sensorPositionsC
lfr-216controllerValue
lfr-2187ControllerAnd
+ lfr-2sensorPositions
lfr-8controllerValues

FMUS
3DanimationFMU
Body_Block
LFRController-SourceC
LFRController-Standal
LFRController
LineFollower_Example
LineFollower_Example
Sensor_Block_01
Sensor_Block_02
Sensor_Block

INTO-CPS > dse-DSE_Example-43

DSE Configuration

Experiment Multi-model

Co-simulation experiment

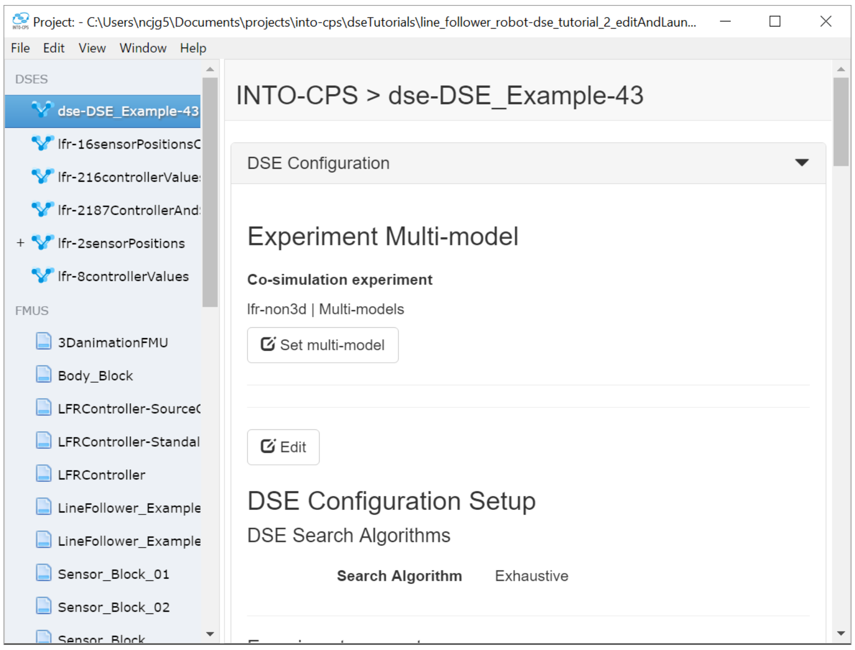
Save multi-model choice

Experiment-3D | lfr-3d
Experiment-3D | lfr-3d-distance
Experiment-3D | lfr-3d-OM
Experiment-3D | lfr-3d-rep
Experiment-3D | lfr-3d-rep-OM
Experiment | lfr-non3d
Experiment | lfr-non3d-OM
Experiment_1 | lfr-non3d-rep

Co-Simulation Engine offline. No connection at localhost:8082. Launch

Simulate

With the multi-model set, the app will now load and parse the DSE configuration and you should be able to see the top of the new panel below, with the algorithm choice *Exhaustive* showing.


Project: - C:\Users\ncjg5\Documents\projects\into-cps\dseTutorials\line_follower_robot-dse_tutorial_2_editAndLaun...

File Edit View Window Help

DSES
dse-DSE_Example-43
lfr-16sensorPositionsC
lfr-216controllerValue
lfr-2187ControllerAnd
+ lfr-2sensorPositions
lfr-8controllerValues

FMUS
3DanimationFMU
Body_Block
LFRController-SourceC
LFRController-Standal
LFRController
LineFollower_Example
LineFollower_Example
Sensor_Block_01
Sensor_Block_02
Sensor_Block

INTO-CPS > dse-DSE_Example-43

DSE Configuration

Experiment Multi-model

Co-simulation experiment

lfr-non3d | Multi-models

Set multi-model

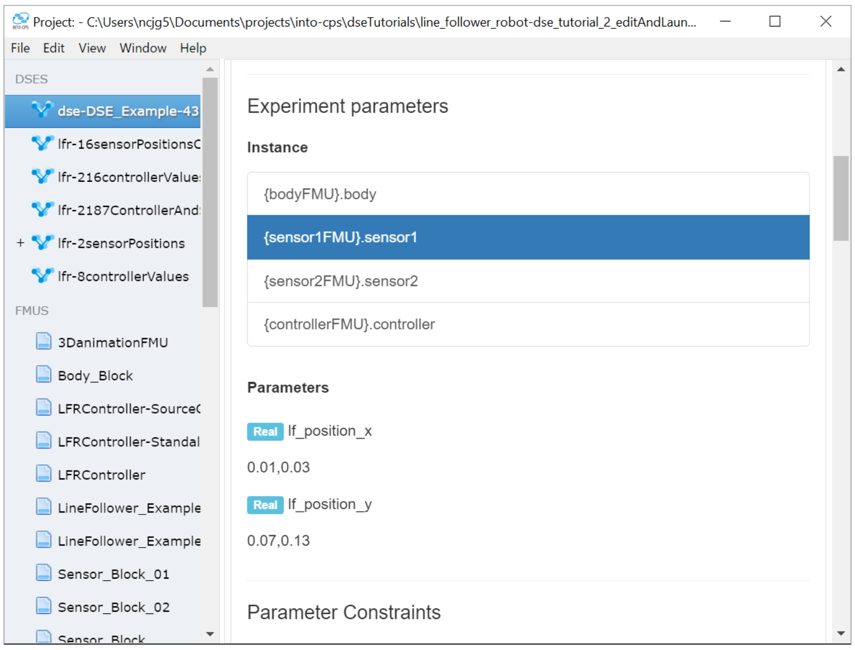
Edit

DSE Configuration Setup

DSE Search Algorithms

Search Algorithm	Exhaustive

At this point, the app is just showing the contents of the configuration, so you may scroll down and view the contents. You may for example scroll down to the parameters section and see the values each parameter of each FMU will take by clicking on the name of the FMU instance. In the example below, the '$sensor1$ FMU is selected and the values for its '$lf\_position\_x$ and '$lf\_position\_y$ parameters are shown.



Experiment parameters

Instance

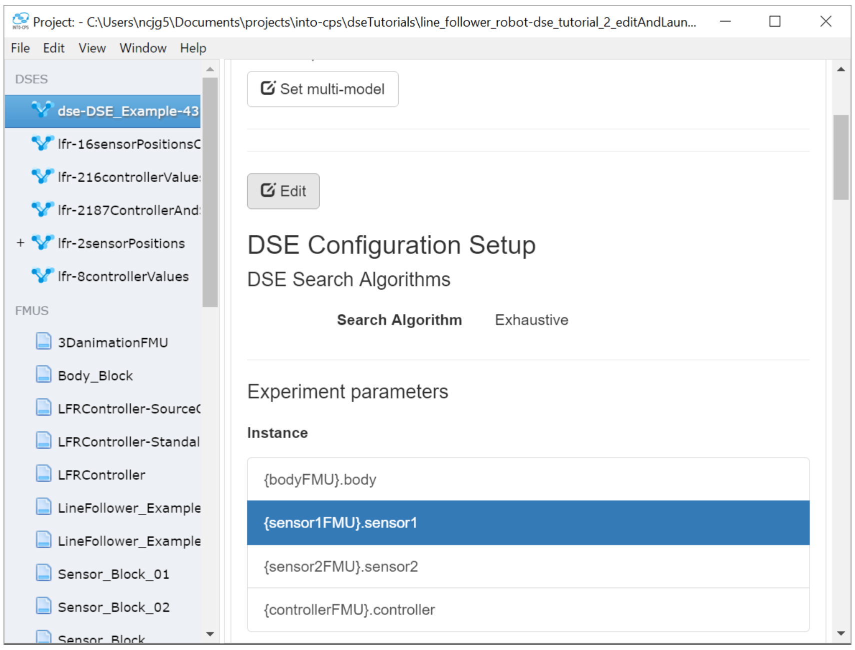
{bodyFMU}.body
{sensor1FMU}.sensor1
{sensor2FMU}.sensor2
{controllerFMU}.controller

Parameters

Real lf_position_x
0.01,0.03
Real lf_position_y
0.07,0.13

Parameter Constraints

To switch the app into editing mode, scroll back to the top of the DSE configuration section and click on the *Edit* button. This will update all the fields in the DSE configuration to be editable and the button will change to say *Save*, ready for when you are finished editing.



DSE Configuration Setup

DSE Search Algorithms

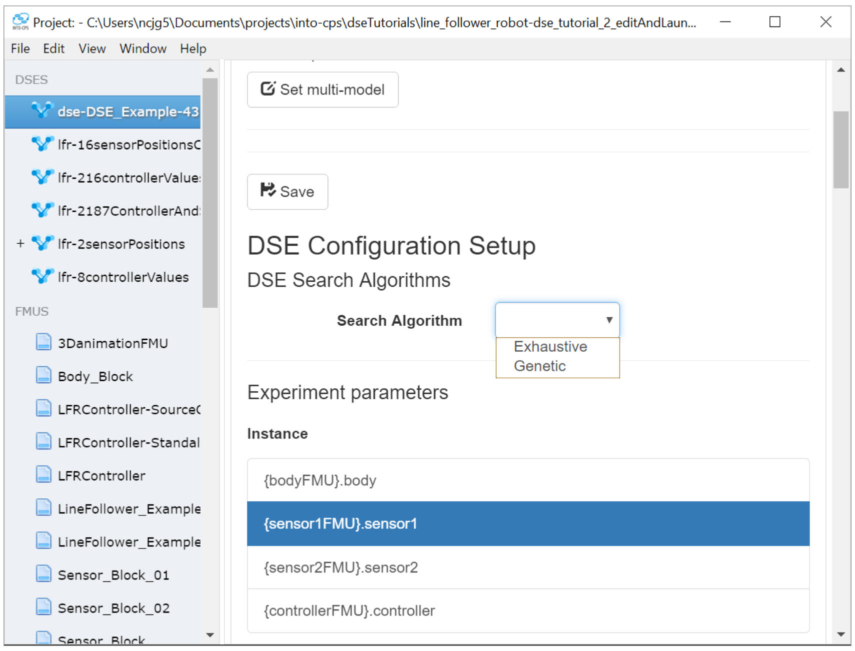
Search Algorithm Exhaustive

Experiment parameters

Instance

{bodyFMU}.body
{sensor1FMU}.sensor1
{sensor2FMU}.sensor2
{controllerFMU}.controller

The view changes to give editable options. Starting with the algorithm section, clicking on the *Search algorithm* dropdown will reveal the two options which are *Genetic* and *Exhaustive*.



DSE Configuration Setup

DSE Search Algorithms

Search Algorithm: Exhaustive / Genetic

Experiment parameters

Instance

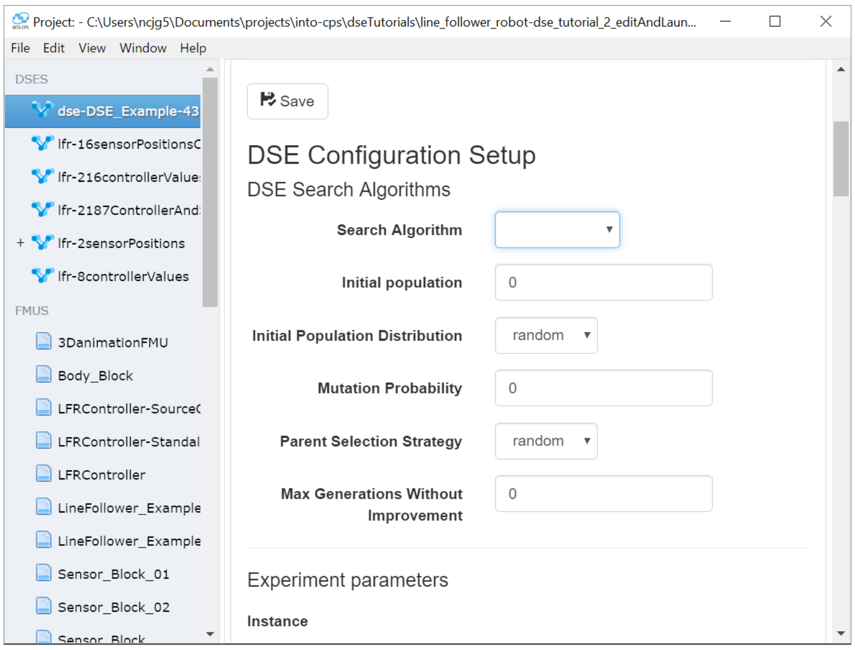
{bodyFMU}.body

{sensor1FMU}.sensor1

{sensor2FMU}.sensor2

{controllerFMU}.controller

If you click on *Genetic*, this will reveal the options available for a genetic search. The initial population parameter accepts an integer value > 1, which defines the number of designs that will be simulated at the start of the algorithm. The population distribution option currently only has one option random, meaning the initial population will be randomly distributed throughout the design space. Other options are currently in development and will be added in the future. The mutation probability accepts integer values ($0 < value < 100$) which describes the probability that a parameter will spontaneously adopt another valid value when new designs are generated by breeding two parents. The parent selection strategy affects how parents are selected during the breeding phase of the genetic algorithm. Currently the only option is random, meaning that the only factor affecting parent selection is their rank, where the designs on rank 1 (best) have a higher probability of being selected than those on rank 2, and designs on rank 2 have a higher chance than those on rank 3 and so on. Other options will be added here in the future. A more detailed description of how the genetic algorithm works and its parameters may be found in the INTO-CPS deliverable D5.2D 'DSE in the INTO-CPS Platform, which may be found on the INTO-CPS website. We will not be using the genetic algorithm in this tutorial, so select the *Exhaustive* algorithm in the *Search algorithm* dropdown. This will remove the genetic options and put the configuration back the way we need it for the tutorial.



DSE Configuration Setup

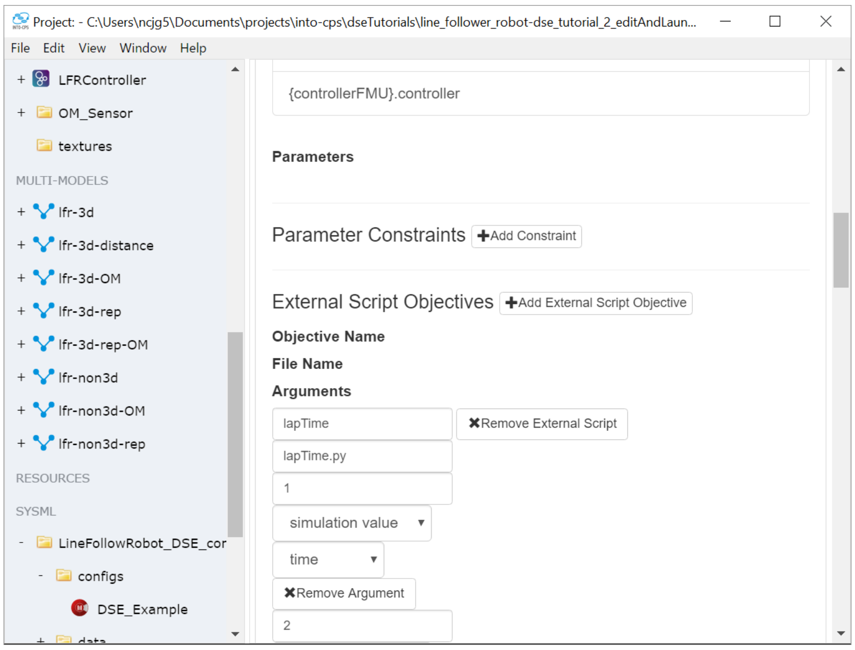
DSE Search Algorithms

Search Algorithm	
Initial population	0
Initial Population Distribution	random
Mutation Probability	0
Parent Selection Strategy	random
Max Generations Without Improvement	0

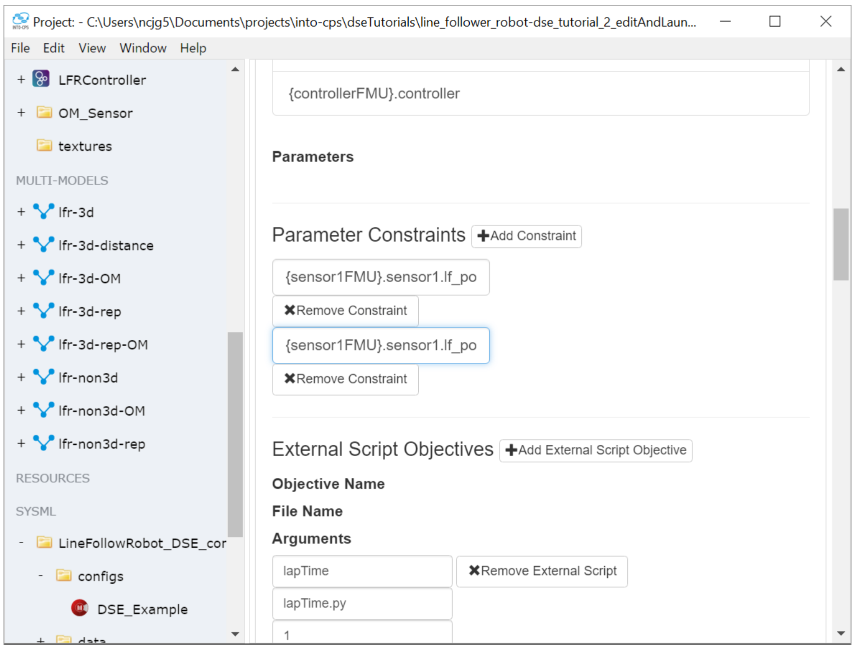
Experiment parameters

Instance

Scrolling down a little further we arrive at the *Parameters* section, and specifically the *Parameter Constraints* section, which is currently empty. Without adding constraints, the exhaustive algorithm will explore all combinations of the parameter values we have defined. Now since the x and y coordinate parameters of the left and right line follower sensors are independent of each other, if we do not add constraints we leave ourselves open to trying the robot with asymmetrical sensor placement, such as the left sensor close to the robot with the right hand sensor way out in front. In this case we dont want this so we are going add constraints to make the design symmetrical. Constraints are written as boolean expressions using Python syntax and referencing the full names of the parameters the DSE configuration knows about. In this case, to make the designs symmetrical, we want to make the y coordinates of the left and right sensors the same while making the x coordinate of one sensor the negation of the other sensor.



To add the constraints, click on the 'add constraint' button, this will add a text box and enter a single constraint. Repeat this for each new constraint you wish to add. For the purpose of the example you will need to add the following two constraints, the first ensures the y coordinates of the two sensors are the same and the second mirrors the x coordinate.
$\{sensor1FMU\}.sensor1.lf\_position_y == \{sensor2FMU\}.sensor2.lf\_position_y$
$\{sensor1FMU\}.sensor1.lf\_position_x == -\{sensor2FMU\}.sensor2.lf\_position_x$



{controllerFMU}.controller

Parameters

Parameter Constraints ✚Add Constraint

{sensor1FMU}.sensor1.lf_po

✖Remove Constraint

{sensor1FMU}.sensor1.lf_po

✖Remove Constraint

External Script Objectives ✚Add External Script Objective

Objective Name

File Name

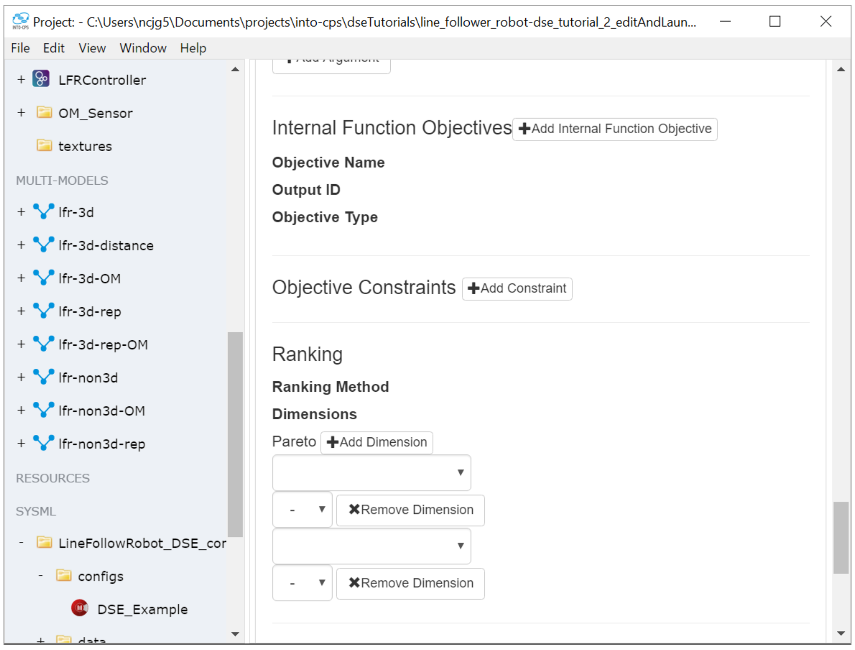
Arguments

lapTime ✖Remove External Script

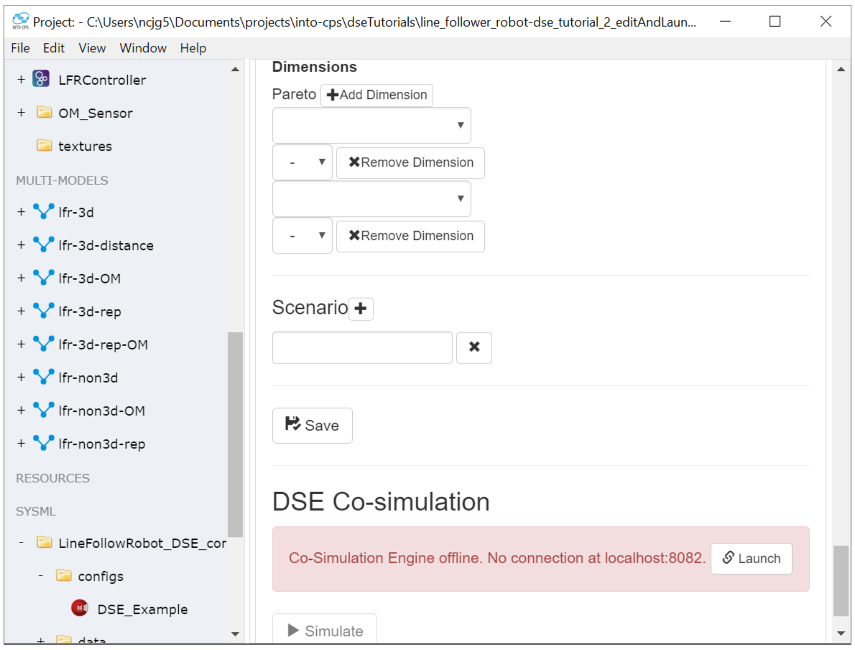
lapTime.py

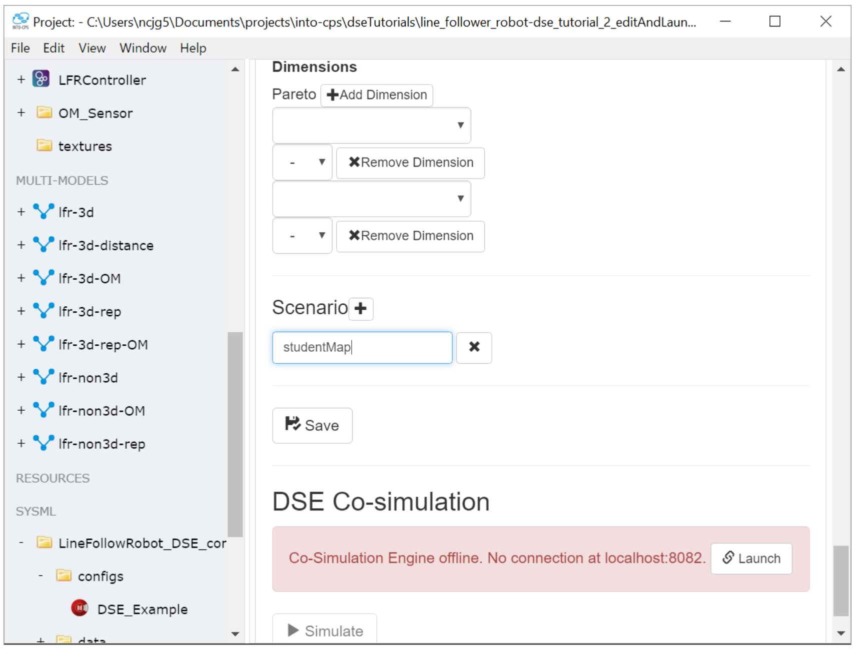
1

The next image shows a section named *Objective Constraints*. This is not yet implemented, but when it is you will be able to add boolean expressions referencing the Objective names. These constraints will control which designs are presented to the engineer in the final results, and, in the genetic algorithm, will only let acceptable designs enter the breeding phase of the algorithm.
Next is the Ranking section. Recall from the previous tutorial (Tutorial 5), that we set the Pareto ranking to use the mean cross track error and lap time objectives. This is still true here, though in editing mode the currently selected objectives are obscured. If you want change the objectives used for ranking, you could select their names from the two drop down boxes.
Hint: dont do this now or you will get different results later in the tutorial!.

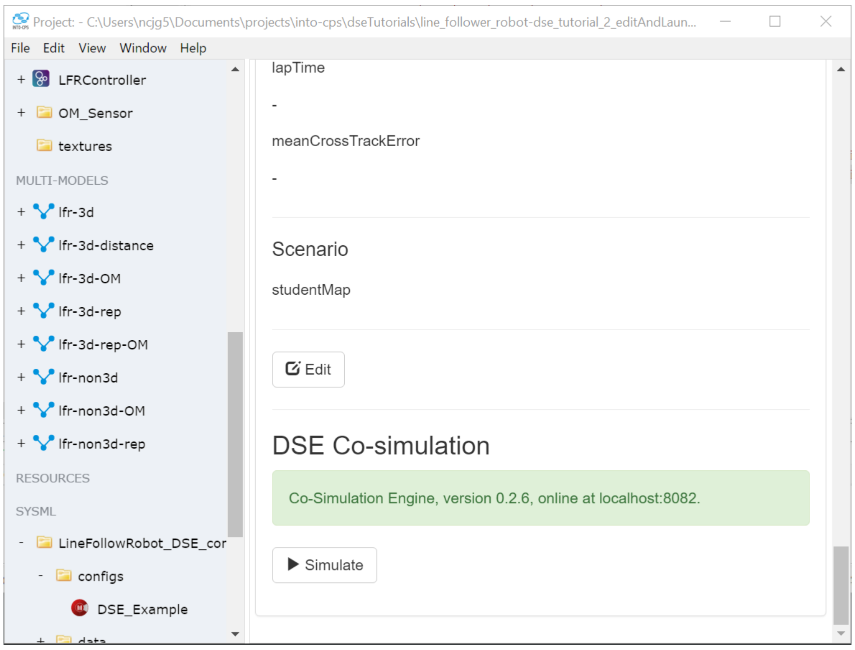


In the final part of the configuration, we need to edit the Scenario name. Scenarios are a way of letting the objective scripts know what test environment the CPS was faced with, which in the case of the line follower robot, is the name of the map it is using. This details of what the objective scripts do with this name are discussed in Tutorial 7 on objective scripts. For now we simply need to enter the name of the map which is $studentMap$ in the scenarios text box.





With the scenario name entered, you may now save the configuration and launch the COE ready to run the DSE. So click on the *Save* button and then on the *Launch* button for the COE.



lapTime

-

meanCrossTrackError

-

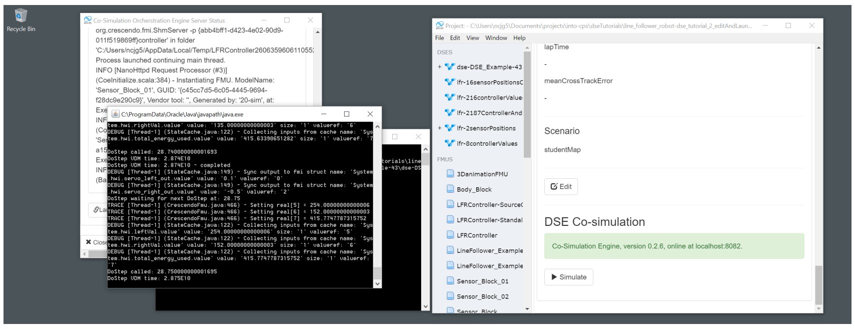
Scenario

studentMap

DSE Co-simulation

Co-Simulation Engine, version 0.2.6, online at localhost:8082.

When the COE status panel is green, you may click on the *Simulate* button to start the DSE.

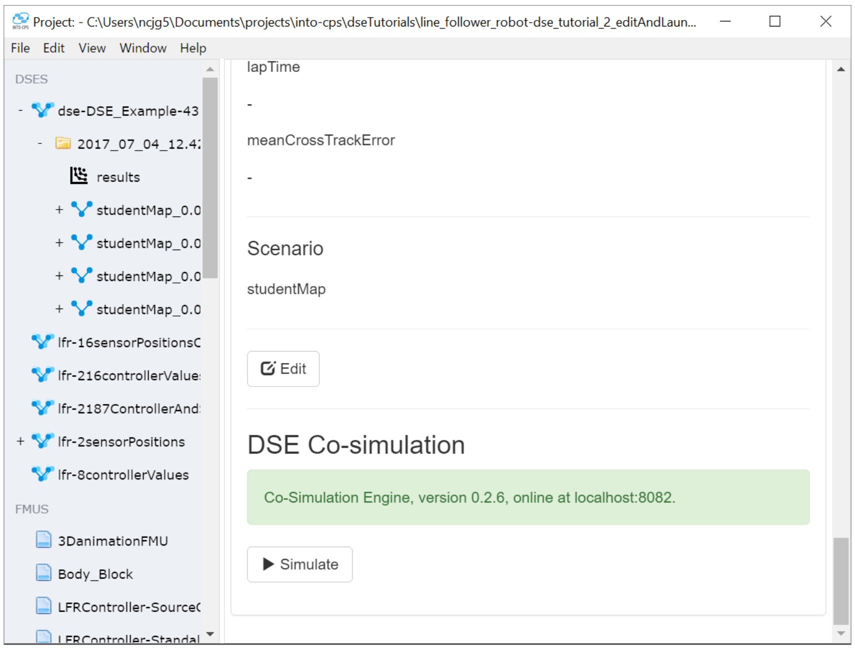


```
org.crescendo.fmi.ShmServer -p {abb4bff1-d423-4e02-90d9-
011f519869ff}controller' in folder
'C:/Users/ncjg5/AppData/Local/Temp/LFRController260635960611055
Process launched continuing main thread.
INFO [NanoHttpd Request Processor (#3)]
(CoeInitialize.scala:384) - Instantiating FMU. ModelName:
'Sensor_Block_01', GUID: '{c45cc7d5-6c05-4445-9694-
f28dc9e290c9}', Vendor tool: '', Generated by: '20-sim', at:
```

```
tem.hwi.rightVal.value' value: '135.00000000000003' size: '1' valueref: '6'
DEBUG [Thread-1] (StateCache.java:122) - Collecting inputs from cache name: 'Sys
tem.hwi.total_energy_used.value' value: '415.63390651282' size: '1' valueref: '7
'
DoStep called: 28.740000000001693
DoStep UDM time: 2.874E10
DoStep UDM time: 2.874E10 - completed
DEBUG [Thread-1] (StateCache.java:149) - Sync output to fmi struct name: 'System
.hwi.servo_left_out.value' value: '0.1' valueref: '0'
DEBUG [Thread-1] (StateCache.java:149) - Sync output to fmi struct name: 'System
.hwi.servo_right_out.value' value: '-0.5' valueref: '2'
DoStep waiting for next DoStep at: 28.75
TRACE [Thread-1] (CrescendoFmu.java:466) - Setting real[5] = 254.00000000000006
TRACE [Thread-1] (CrescendoFmu.java:466) - Setting real[6] = 152.00000000000003
TRACE [Thread-1] (CrescendoFmu.java:466) - Setting real[7] = 415.7747787315752
DEBUG [Thread-1] (StateCache.java:122) - Collecting inputs from cache name: 'Sys
tem.hwi.leftVal.value' value: '254.00000000000006' size: '1' valueref: '5'
DEBUG [Thread-1] (StateCache.java:122) - Collecting inputs from cache name: 'Sys
tem.hwi.rightVal.value' value: '152.00000000000003' size: '1' valueref: '6'
DEBUG [Thread-1] (StateCache.java:122) - Collecting inputs from cache name: 'Sys
tem.hwi.total_energy_used.value' value: '415.7747787315752' size: '1' valueref:
'7'
DoStep called: 28.750000000001695
DoStep UDM time: 2.875E10
```

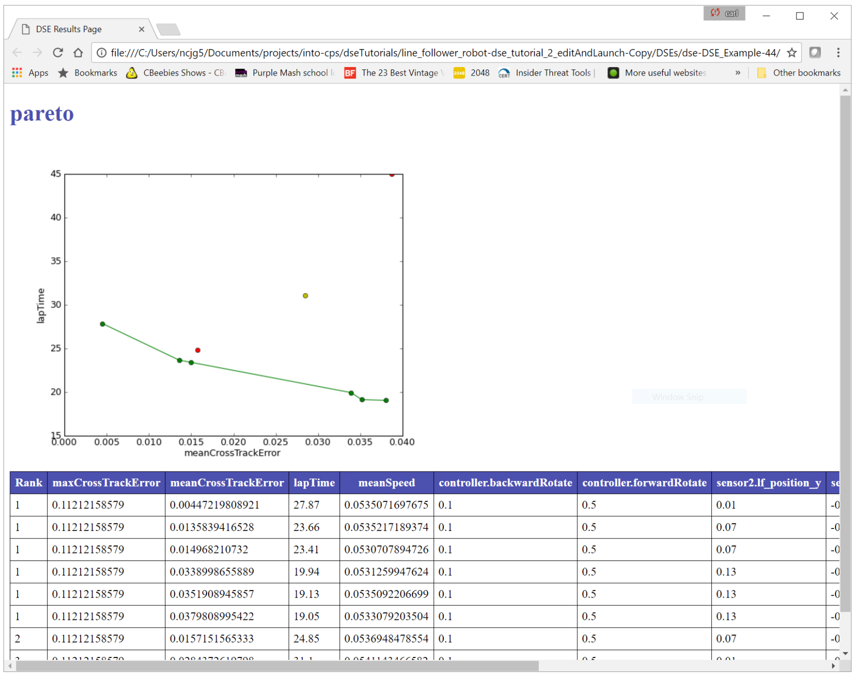
Co-Simulation Engine, version 0.2.6, online at localhost:8082.

This DSE will run 9 simulations and takes a little under 9 minutes on a 3Ghz laptop; It may take more or less time on different hardware. If you are on Windows you will see terminal windows appearing as the app launches the DSE scripts. This confirms that the DSE is running, but there may be less obvious signs of activity on other operating systems. Currently, the main indicator that a DSE is complete may be found in the app itself. In the DSEs section, expanding the structure under the name of the DSE configuration reveals a folder named for the date and time that the DSE commenced. Expanding this directory reveals subdirectories containing results from each individual simulation. When the DSE is complete there will be another file in this directory named '$Results$, as below.



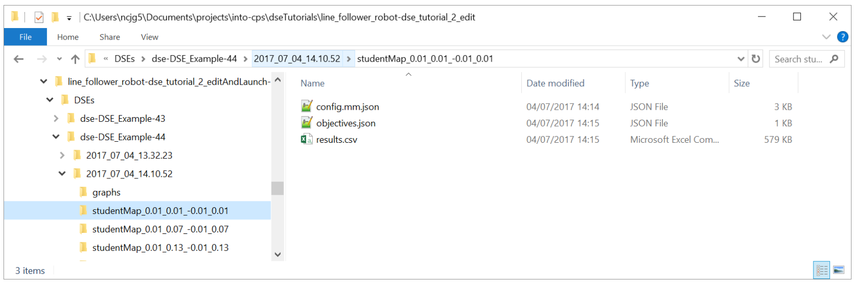
Co-Simulation Engine, version 0.2.6, online at localhost:8082.

Double clicking on the Results icon will open the results HTML file in your default internet browser. The results consist of two parts: The first part shows a graph of results in the form of series of points representing each individual simulation. The non-dominated set (best results) are shown in green with a green line connecting them, with the remaining points shown in red and yellow. Below the graph are the details of the results, ordered by rank. Each row represents one design and includes details of the calculated objective values, along with the parameters that resulted in that particular outcome.
Results in rank 1 (green) are suitable for further investigation or testing on a physical prototype.


DSE Results Page
file:///C:/Users/ncjg5/Documents/projects/into-cps/dseTutorials/line_follower_robot-dse_tutorial_2_editAndLaunch-Copy/DSEs/dse-DSE_Example-44/
pareto
lapTime
meanCrossTrackError

Rank	maxCrossTrackError	meanCrossTrackError	lapTime	meanSpeed	controller.backwardRotate	controller.forwardRotate	sensor2.lf_position_y	se
1	0.11212158579	0.00447219808921	27.87	0.0535071697675	0.1	0.5	0.01	-0
1	0.11212158579	0.0135839416528	23.66	0.0535217189374	0.1	0.5	0.07	-0
1	0.11212158579	0.014968210732	23.41	0.0530707894726	0.1	0.5	0.07	-0
1	0.11212158579	0.0338998655889	19.94	0.0531259947624	0.1	0.5	0.13	-0
1	0.11212158579	0.0351908945857	19.13	0.0535092206699	0.1	0.5	0.13	-0
1	0.11212158579	0.0379808995422	19.05	0.0533079203504	0.1	0.5	0.13	-0
2	0.11212158579	0.0157151565333	24.85	0.0536948478554	0.1	0.5	0.07	-0

The results may also be explored by opening a file browser and navigating to the *INTO-CPS* project folder, then opening the DSEs directory, then the subdirectory containing the DSE config. The results may be found in the directory labelled with the date and time the DSE was completed. In there you will find subdirectories named with the values of the parameters used in that simulation. In each of these directories you will find *results.csv* containing the raw simulation results and two other files.


C:\Users\ncjg5\Documents\projects\into-cps\dseTutorials\line_follower_robot-dse_tutorial_2_edit
File
Home
Share
View
DSEs > dse-DSE_Example-44 > 2017_07_04_14.10.52 > studentMap_0.01_0.01_-0.01_0.01
Search stu...
line_follower_robot-dse_tutorial_2_editAndLaunch-
DSEs
dse-DSE_Example-43
dse-DSE_Example-44
2017_07_04_13.32.23
2017_07_04_14.10.52
graphs
studentMap_0.01_0.01_-0.01_0.01
studentMap_0.01_0.07_-0.01_0.07
studentMap_0.01_0.13_-0.01_0.13
3 items

Name	Date modified	Type	Size
config.mm.json	04/07/2017 14:14	JSON File	3 KB
objectives.json	04/07/2017 14:15	JSON File	1 KB
results.csv	04/07/2017 14:15	Microsoft Excel Com...	579 KB

The *objectives.json* file contains the objective values calculated from the raw simulation results.

```
{
    "lapTime": 23.410000000001048,
    "maxCrossTrackError": 0.11212158578971308,
    "meanCrossTrackError": 0.014968210731969004,
    "meanSpeed": 0.05307078947255971
}
```

The *config.mm.json* file contains the complete multi-model configuration sent to the coe when launching the simulation. In this you may find all parameters used to produce the simulation result, including those not altered by the DSE configurations. The only detail not included in this file is the length of the simulation. This file can provide the details required to replicate a simulation, perhaps with another multi-model including 3D output for visualisation.

```
"parameters": {
    "{controllerFMU}.controller.backwardRotate": 0.1,
    "{controllerFMU}.controller.forwardRotate": 0.5,
    "{controllerFMU}.controller.forwardSpeed": 0.4,
    "{sensor1FMU}.sensor1.lf_position_x": 0.01,
    "{sensor1FMU}.sensor1.lf_position_y": 0.07,
    "{sensor2FMU}.sensor2.lf_position_x": -0.01,
    "{sensor2FMU}.sensor2.lf_position_y": 0.07
}
```

Tutorial - SysML for Co-Simulation

Overview

This set of exercises will show you how to:

1. Create a simple CPS Design
2. Generate a Co-Simulation
3. Export a FMI Model Description
4. Import a FMI Model Description

Requirements

This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- Modelio v3.4.1

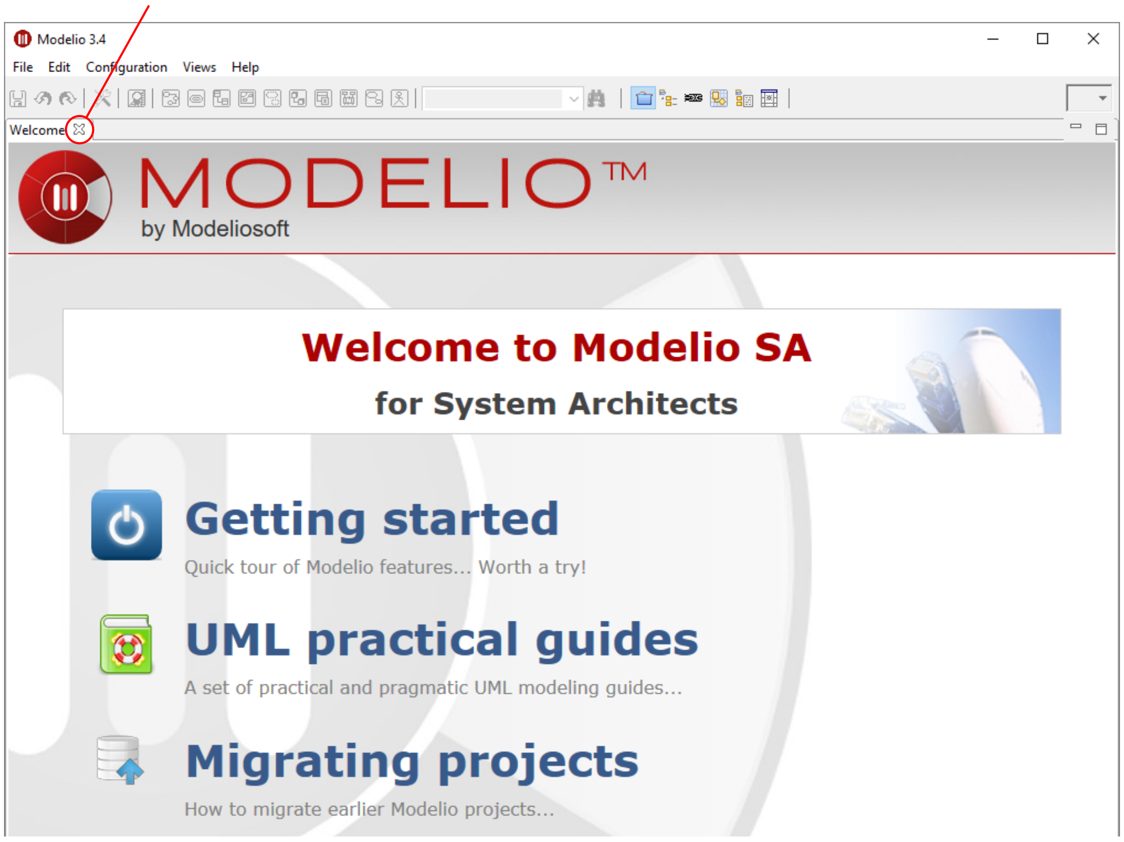
You may have been provided with tools on a USB drive at your training session. Otherwise the INTO-CPS Application can be downloaded from `https://into-cps.github.io/download/` and tools can be downloaded from there through *Window > Show Download Manager* to your *into-cps-projects* install downloads directory. Please ask if you are unsure.

You may need to update the INTO-CPS extension for Modelio to utilise DSE components. The extension can be downloaded from `http://forge.modelio.org/projects/intocps-modelio34/files`. Version 1.2.05 or later is required for DSE modelling.

1 Opening a Project

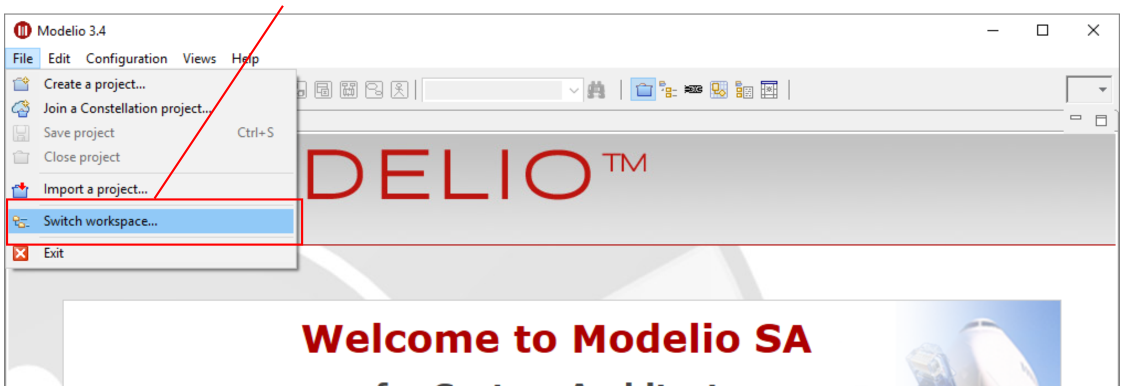
Step 1. Launch *Modelio*. On first loading, you may have to close the *Welcome* screen (you can bring it back with 'Help > Welcome' if you need)

Close *Welcome* screen

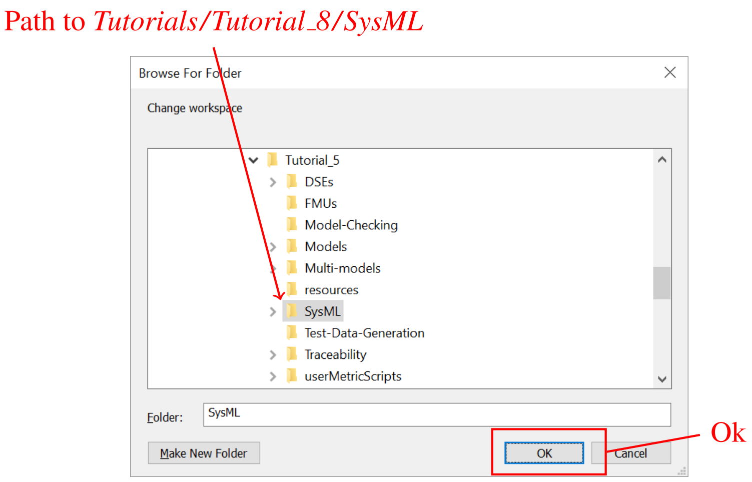

Modelio 3.4
File Edit Configuration Views Help
Welcome
MODELIO™
by Modeliosoft
Welcome to Modelio SA
for System Architects
Getting started
Quick tour of Modelio features... Worth a try!
UML practical guides
A set of practical and pragmatic UML modeling guides...
Migrating projects
How to migrate earlier Modelio projects...

Step 2. A workspace must be chosen, select 'File > Switch Workspace.

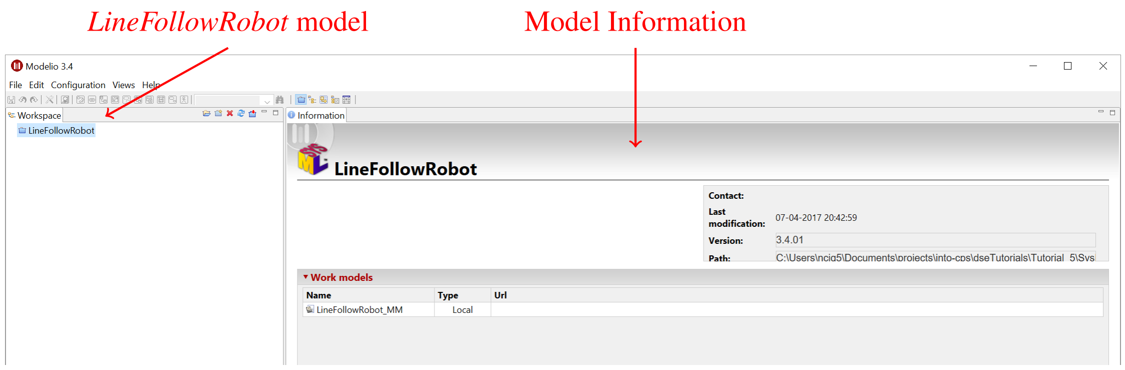
Switch Workspace



Step 3. Set the *Workspace* to the location of *Tutorials/Tutorial_8/SysML* and click *Ok*.



Step 4. Left-click on the *LineFollowRobot* model once on the left to see details of the model. Double-click the *LineFollowRobot* model to open the model.



Contact:
Last modification: 07-04-2017 20:42:59
Version: 3.4.01
Path: C:\Users\ncig5\Documents\projects\into-cps\dseTutorials\Tutorial_5\Sys

Work models

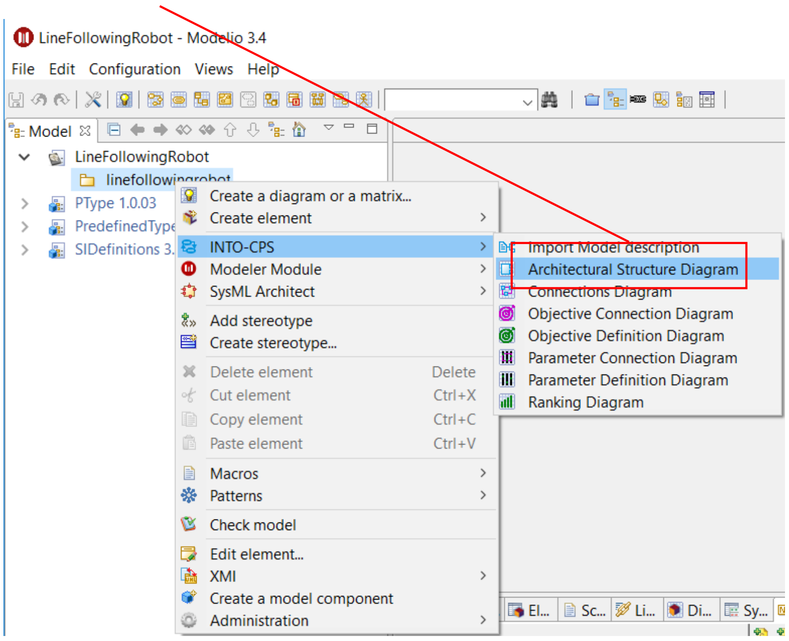
Name	Type	Url
LineFollowRobot_MM	Local	

2 Define an Architecture

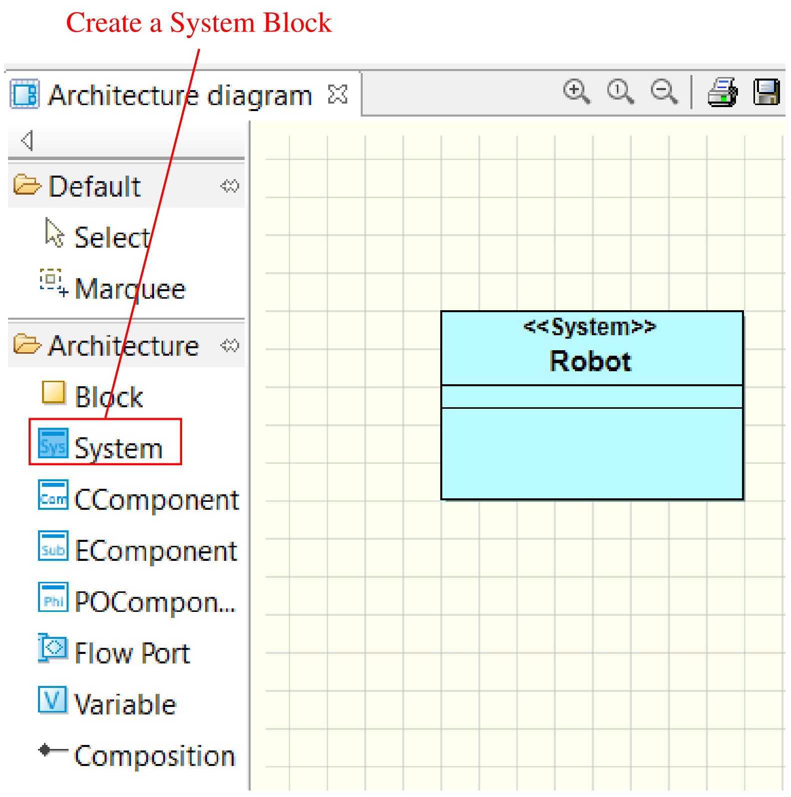
This section describes how to the design of the architecture of your system architecture. CPS Architecture is mainly composed of an unique System Block and a set of CComponent.

Step 5. Right click '$linefollowrobot\_mm$' in model outline. Select 'INTO-CPS > Architectural Structure Diagram'.

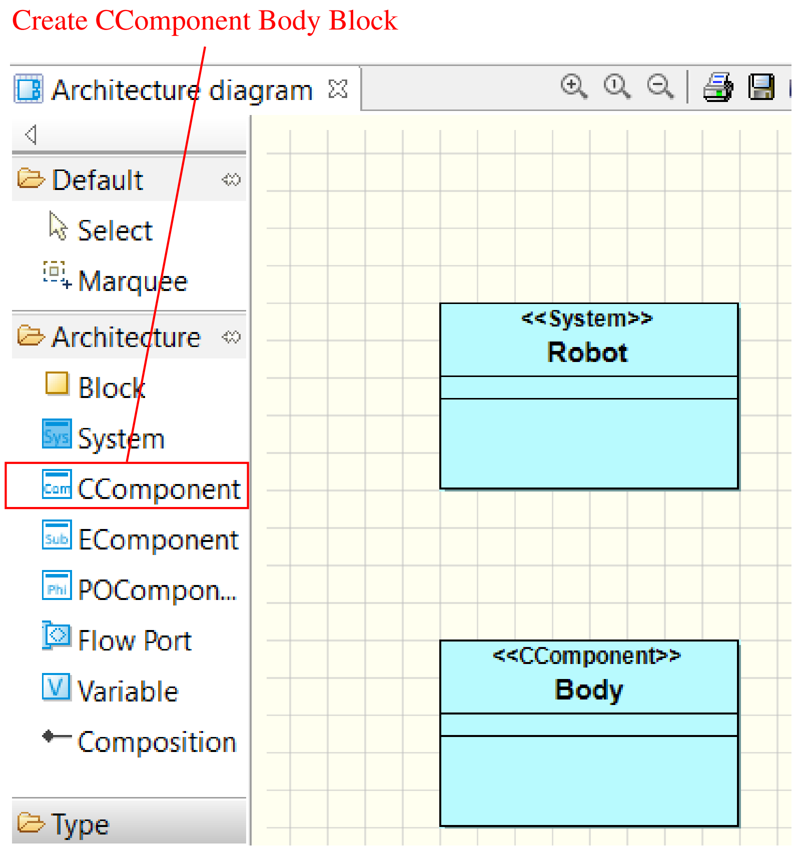
Create an Architectural Structure Diagram



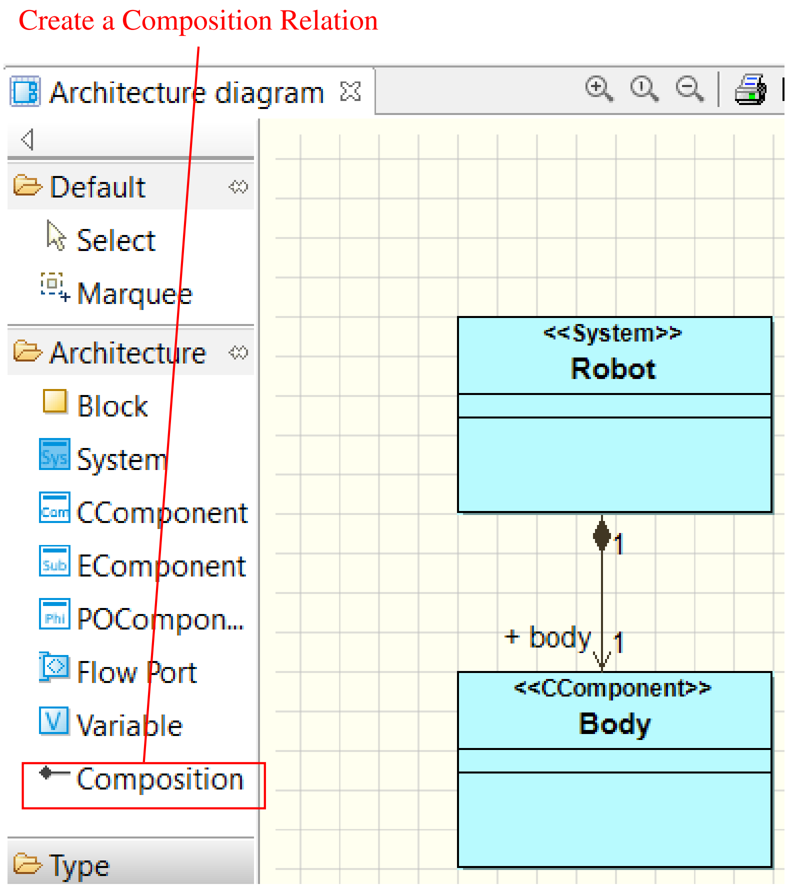
Step 6. From the *Toolbox*, select *System* and click on the empty diagram to create the System block. Double click on the created block and change its name to '*Robot*'.



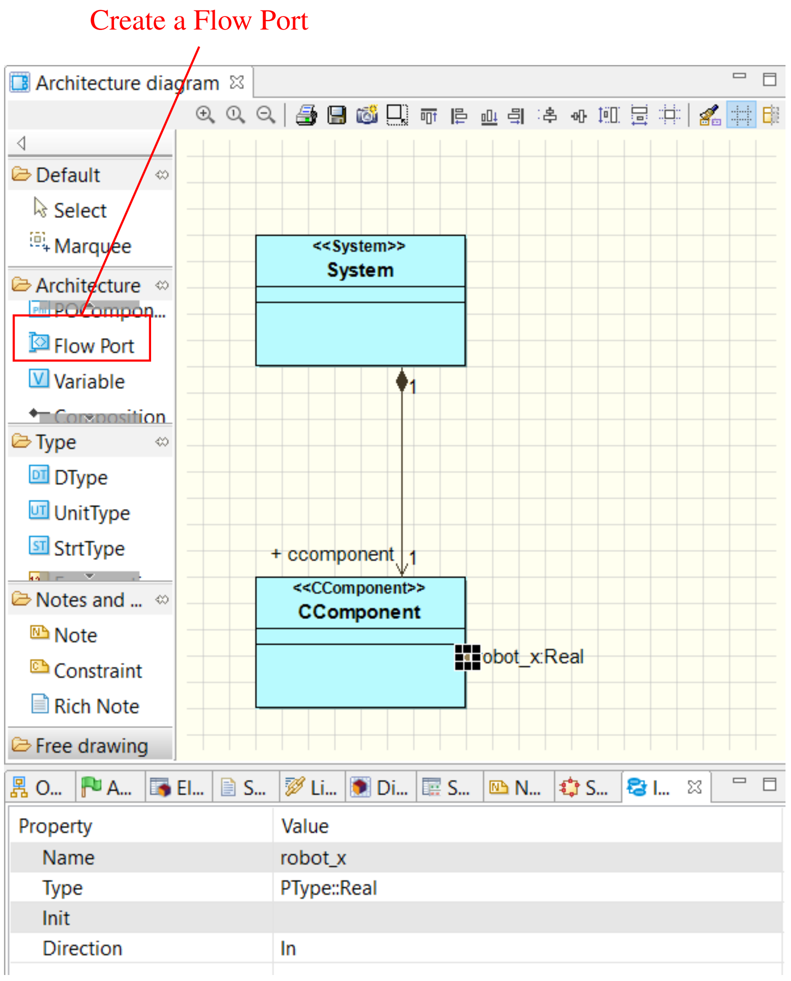
Step 7. From the *Toolbox*, select *CComponent* and add it to the diagram. Double click on the created block and change its name to '*Body*'.



Step 8. To associate the '*Body*' block with the Robot, select the *Composition* relation from the *Toolbox*, and click first on the '*Robot*' and then on the '*Body*' block.



Step 9. From the *Toolbox*, select *Flow Port* and add it to the '*Body*' CComponent. Select the block that this creates and F2 to change its name to '*robot_x*'. On the *INTO-CPS* property view , change *Type* to '*PType* :: *Real*', and *Direction* to '*Out*'.



Property	Value
Name	robot_x
Type	PType::Real
Init	
Direction	In

Step 10. Repeat Step 9. to create the following flow ports to the '*Body*' CComponent:

- Name: '*robot_y*', Type: '*PType* :: *Real*', Direction: '*Out*';
- Name: '*robot_z*', Type: '*PType* :: *Real*', Direction: '*Out*';
- Name: '*robot_theta*', Type: '*PType* :: *Real*', Direction: '*Out*';
- Name: '*servo_right_input*', Type: '*PType* :: *Real*', Direction: '*In*';
- Name: '*servo_left_input*', Type: '*PType* :: *Real*', Direction: '*In*';

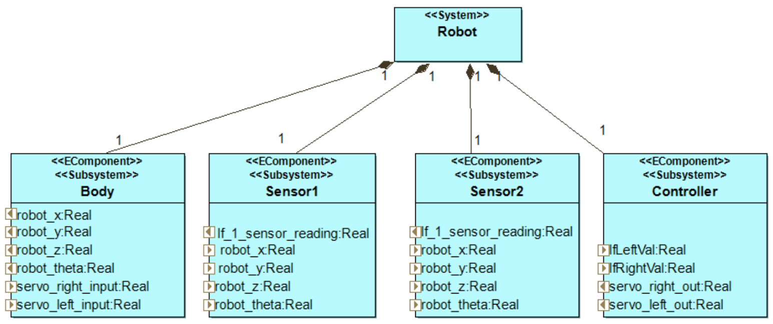
Step 11. Repeat from Step 7. to Step 9. to create the following ccomponent and their corresponding ports:

- Name: '*Sensor*1';
  - Name: '*robot_x*', Type: '*PType* :: *Real*', Direction: '*In*';
  - Name: '*robot_y*', Type: '*PType* :: *Real*', Direction: '*In*';
  - Name: '*robot_z*', Type: '*PType* :: *Real*', Direction: '*In*';
  - Name: '*robot_theta*', Type: '*PType* :: *Real*', Direction: '*In*';
  - Name: '*lf_1_sensor_reading*', Type: '*PType* :: *Real*', Direction: '*Out*';
- Name: '*Sensor*2';
  - Name: '*robot_x*', Type: '*PType* :: *Real*', Direction: '*In*';

  - Name: '$robot\_y$', Type: '$PType :: Real$', Direction: 'In';
  - Name: '$robot\_z$', Type: '$PType :: Real$', Direction: 'In';
  - Name: '$robot\_theta$', Type: '$PType :: Real$', Direction: 'In';
  - Name: '$lf\_1\_sensor\_reading$', Type: '$PType :: Real$', Direction: 'Out';

- Name: '$Controller$';
  - Name: '$ILeftVal$', Type: '$PType :: Real$', Direction: 'In';
  - Name: '$IRightVal$', Type: '$PType :: Real$', Direction: 'In';
  - Name: '$servo\_right\_out$', Type: '$PType :: Real$', Direction: 'Out';
  - Name: '$servo\_left\_out$', Type: '$PType :: Real$', Direction: 'Out';

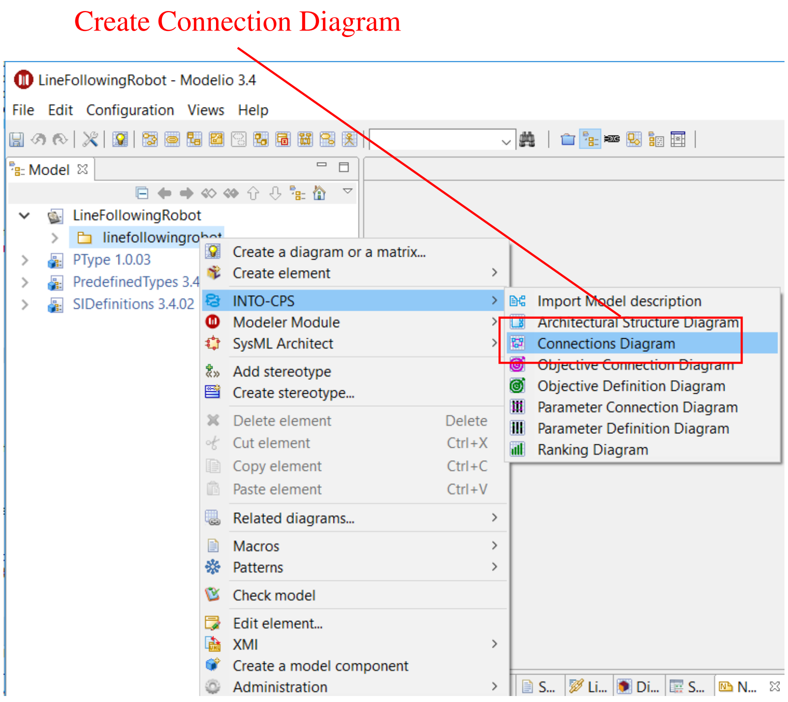
The model should now look like this:



3 Design a Connection Diagram

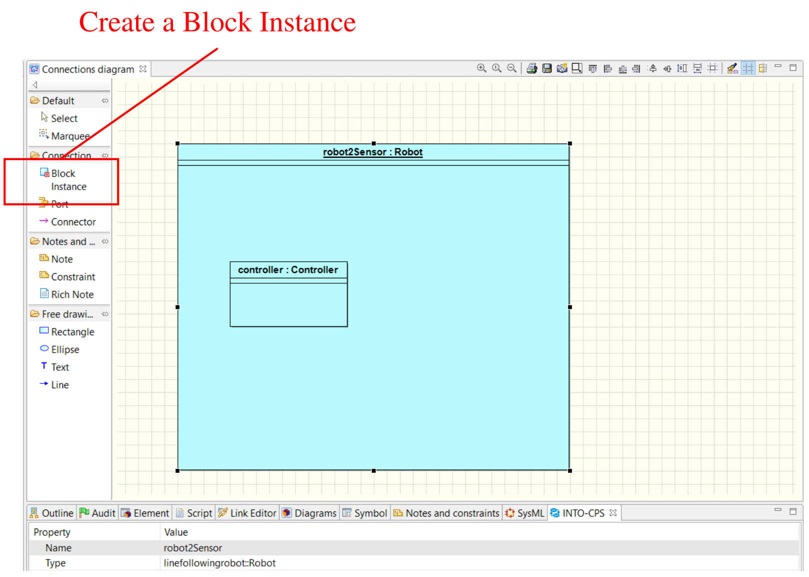
This section will outline the steps necessary to create a Connection Diagram under Modelio.

Step 12. Right click '$linefollowrobot\_mm$' in model outline. Select 'INTO-CPS > Connection Diagram'.



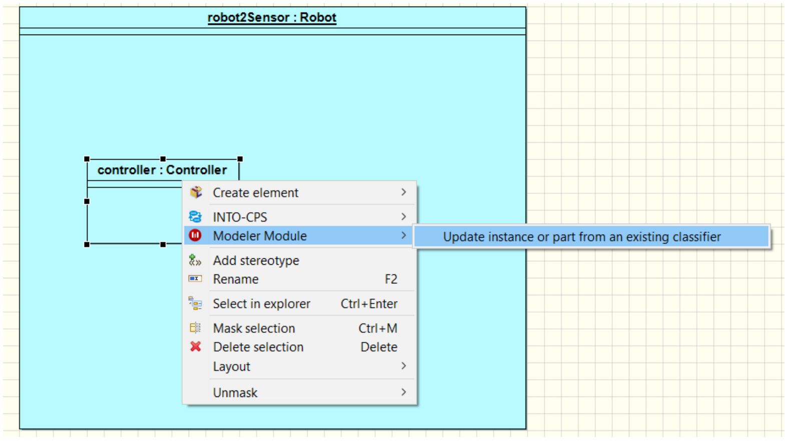
Step 13. Select the Block Instance automatically created with the Connection Diagram. On the *INTO-CPS* property view , change *Name* to '$robot2Sensor$', and its *Type* to '$Robot$'.

Step 14. From the *Toolbox*, select *BlockInstance* and click on the instance element available on the created diagram to create the a block instance. Select the Block Instance created and on the *INTO-CPS* property view , change its *Name* to '$controller$', and its *Type* to '$Controller$'

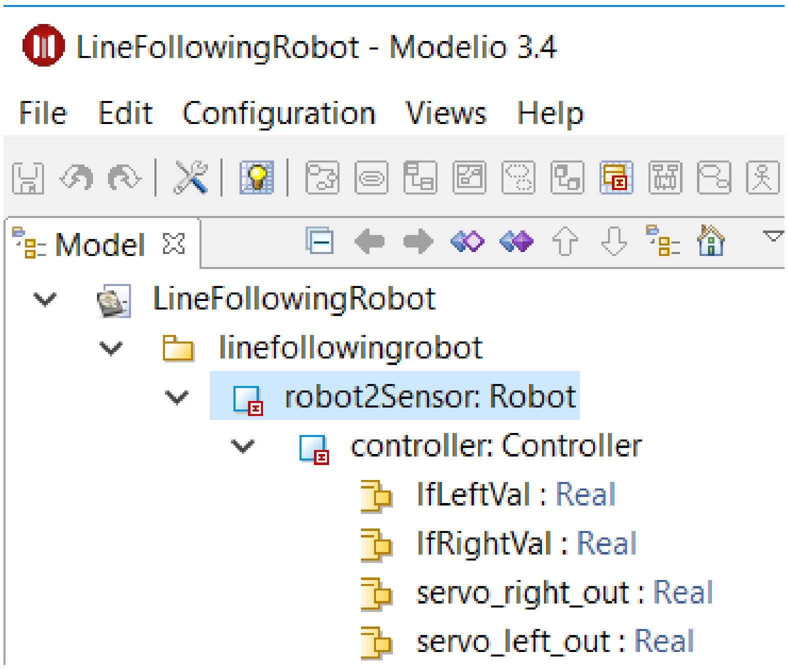


Property	Value
Name	robot2Sensor
Type	linefollowingrobot::Robot

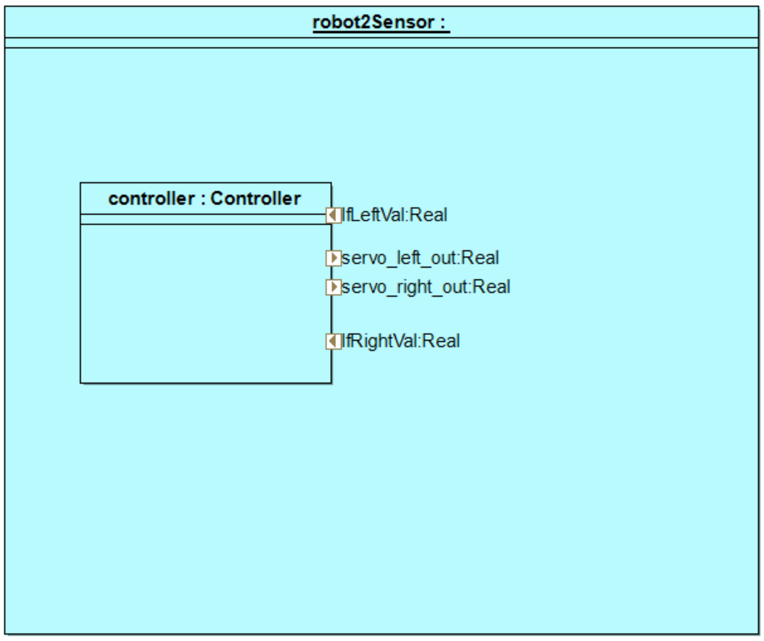
Step 15. Select '*controller*' block instance in the model outline and right click. Select 'Modeller Module > Update instance or part from an existing classifier'. Click *Update* then *OK*.



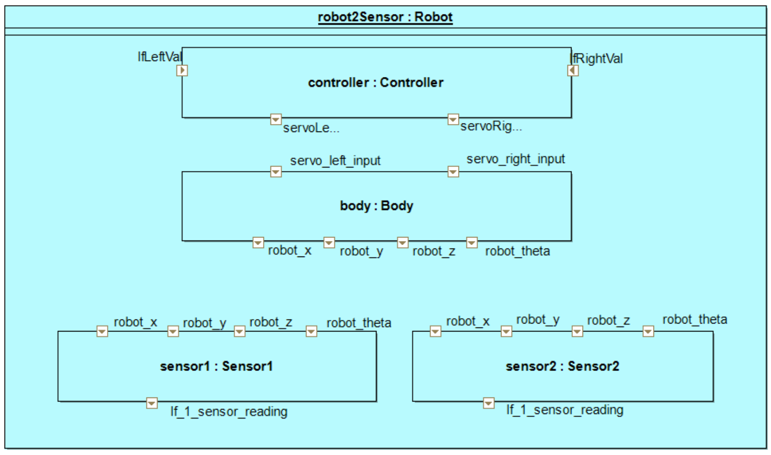
Step 16. Check the created ports inside the model explorer.



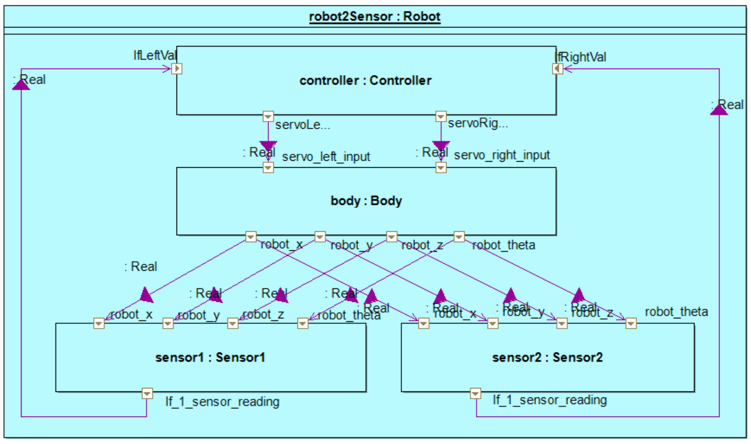
Step 17. Drag and Drop the ports from the model explore inside the diagram to unmask them.



Step 18. Repeat from Step 14. to Step 17. to create block instance of each CComponent.



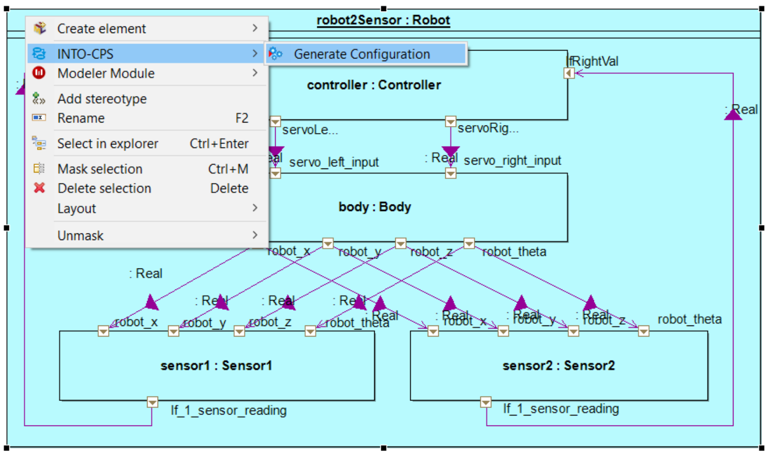
Step 19. From the *Toolbox*, select *Connector* and draw link from Output port to Input to design all connections.



4 Generating Co-Simulation Configuraton

This section will outline the steps necessary to generate a Co-simulation configuration in Modelio and include the file to the INTO-CPS app.

Step 20. Find any '$CComponent$' block in the model outline and right click. Select 'INTO-CPS > Generate Model Description'. Click *Export* then *OK*.

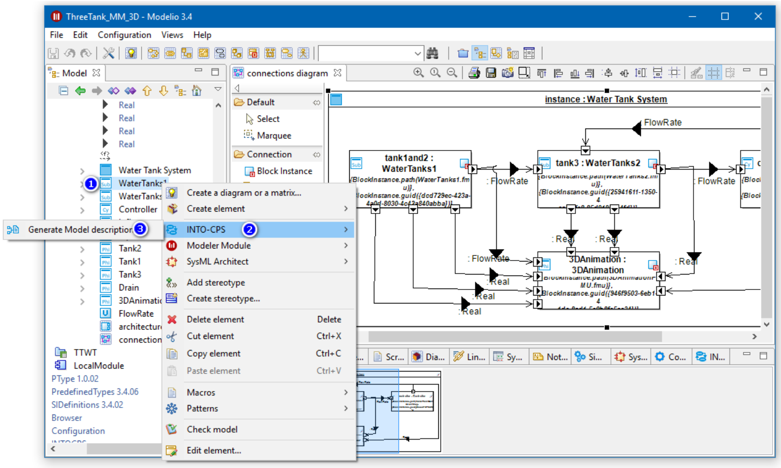


In the INTO-CPS app, this configuration file can be found in 'SysML > LineFollowerRobot > config. Right click the configuration file and select 'Create Multi model.

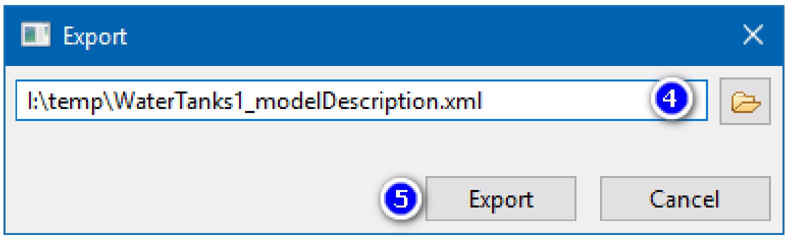
5 Exporting Model Description

This section will outline the steps necessary to export a CComponent as a FMI Model Description.

Step 21. Select any CComponent block like the '$Controller$' block in the model outline and right click. Select 'INTO-CPS > Generate Model Description'. Click *Export* then *OK*.



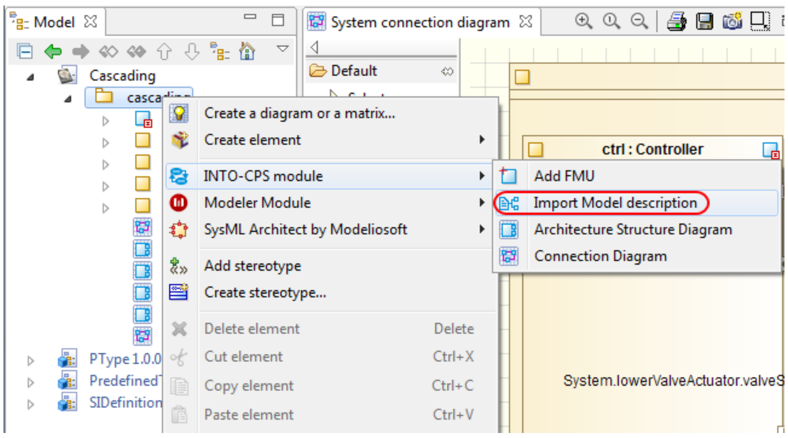
Step 22. Choose a file name destination.



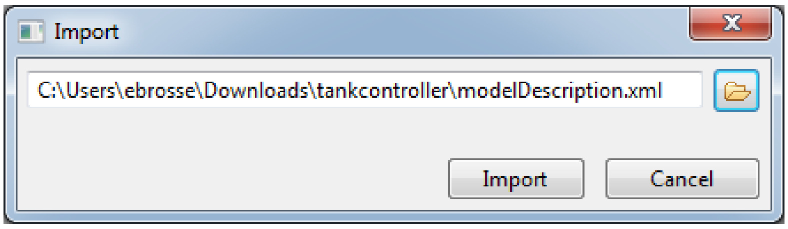
6 Importing Model Description

This section will outline the steps necessary to import an existing Model Description in Modelio.

Step 23. Right click in the Modelio model browser on any *Package* element, then select 'INTO-CPS > Import Model Description'.

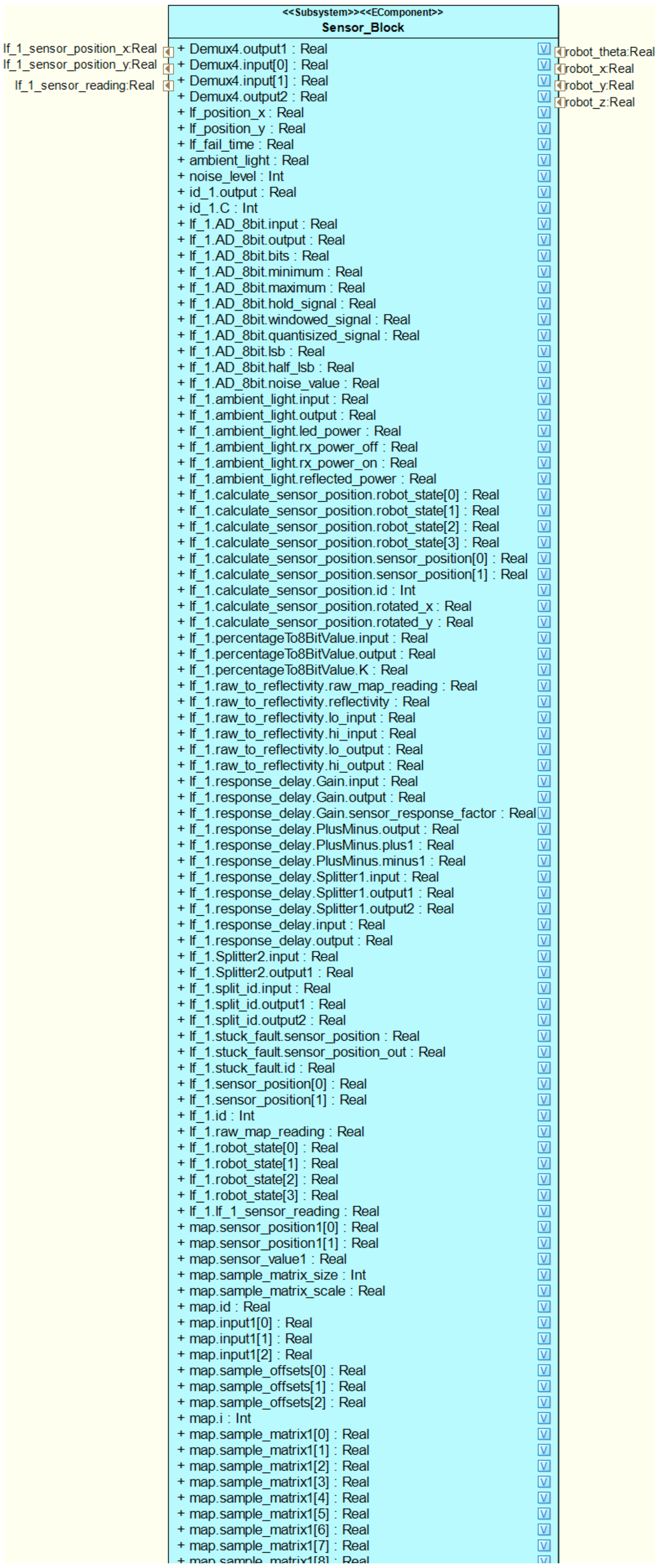


Step 24. Select the desired target file in your computer and click on *Import*.



C:\Users\ebrosse\Downloads\tankcontroller\modelDescription.xml

This import command creates an Architecture Structure Diagram describing the interface of an INTO-CPS *CComponent* corresponding to the `modelDescription.xml` file imported.



<<Subsystem>><<EComponent>>
Sensor_Block
lf_1_sensor_position_x:Real
lf_1_sensor_position_y:Real
lf_1_sensor_reading:Real
robot_theta:Real
robot_x:Real
robot_y:Real
robot_z:Real
+ Demux4.output1 : Real
+ Demux4.input[0] : Real
+ Demux4.input[1] : Real
+ Demux4.output2 : Real
+ lf_position_x : Real
+ lf_position_y : Real
+ lf_fail_time : Real
+ ambient_light : Real
+ noise_level : Int
+ id_1.output : Real
+ id_1.C : Int
+ lf_1.AD_8bit.input : Real
+ lf_1.AD_8bit.output : Real
+ lf_1.AD_8bit.bits : Real
+ lf_1.AD_8bit.minimum : Real
+ lf_1.AD_8bit.maximum : Real
+ lf_1.AD_8bit.hold_signal : Real
+ lf_1.AD_8bit.windowed_signal : Real
+ lf_1.AD_8bit.quantisized_signal : Real
+ lf_1.AD_8bit.lsb : Real
+ lf_1.AD_8bit.half_lsb : Real
+ lf_1.AD_8bit.noise_value : Real
+ lf_1.ambient_light.input : Real
+ lf_1.ambient_light.output : Real
+ lf_1.ambient_light.led_power : Real
+ lf_1.ambient_light.rx_power_off : Real
+ lf_1.ambient_light.rx_power_on : Real
+ lf_1.ambient_light.reflected_power : Real
+ lf_1.calculate_sensor_position.robot_state[0] : Real
+ lf_1.calculate_sensor_position.robot_state[1] : Real
+ lf_1.calculate_sensor_position.robot_state[2] : Real
+ lf_1.calculate_sensor_position.robot_state[3] : Real
+ lf_1.calculate_sensor_position.sensor_position[0] : Real
+ lf_1.calculate_sensor_position.sensor_position[1] : Real
+ lf_1.calculate_sensor_position.id : Int
+ lf_1.calculate_sensor_position.rotated_x : Real
+ lf_1.calculate_sensor_position.rotated_y : Real
+ lf_1.percentageTo8BitValue.input : Real
+ lf_1.percentageTo8BitValue.output : Real
+ lf_1.percentageTo8BitValue.K : Real
+ lf_1.raw_to_reflectivity.raw_map_reading : Real
+ lf_1.raw_to_reflectivity.reflectivity : Real
+ lf_1.raw_to_reflectivity.lo_input : Real
+ lf_1.raw_to_reflectivity.hi_input : Real
+ lf_1.raw_to_reflectivity.lo_output : Real
+ lf_1.raw_to_reflectivity.hi_output : Real
+ lf_1.response_delay.Gain.input : Real
+ lf_1.response_delay.Gain.output : Real
+ lf_1.response_delay.Gain.sensor_response_factor : Real
+ lf_1.response_delay.PlusMinus.output : Real
+ lf_1.response_delay.PlusMinus.plus1 : Real
+ lf_1.response_delay.PlusMinus.minus1 : Real
+ lf_1.response_delay.Splitter1.input : Real
+ lf_1.response_delay.Splitter1.output1 : Real
+ lf_1.response_delay.Splitter1.output2 : Real
+ lf_1.response_delay.input : Real
+ lf_1.response_delay.output : Real
+ lf_1.Splitter2.input : Real
+ lf_1.Splitter2.output1 : Real
+ lf_1.split_id.input : Real
+ lf_1.split_id.output1 : Real
+ lf_1.split_id.output2 : Real
+ lf_1.stuck_fault.sensor_position : Real
+ lf_1.stuck_fault.sensor_position_out : Real
+ lf_1.stuck_fault.id : Real
+ lf_1.sensor_position[0] : Real
+ lf_1.sensor_position[1] : Real
+ lf_1.id : Int
+ lf_1.raw_map_reading : Real
+ lf_1.robot_state[0] : Real
+ lf_1.robot_state[1] : Real
+ lf_1.robot_state[2] : Real
+ lf_1.robot_state[3] : Real
+ lf_1.lf_1_sensor_reading : Real
+ map.sensor_position1[0] : Real
+ map.sensor_position1[1] : Real
+ map.sensor_value1 : Real
+ map.sample_matrix_size : Int
+ map.sample_matrix_scale : Real
+ map.id : Real
+ map.input1[0] : Real
+ map.input1[1] : Real
+ map.input1[2] : Real
+ map.sample_offsets[0] : Real
+ map.sample_offsets[1] : Real
+ map.sample_offsets[2] : Real
+ map.i : Int
+ map.sample_matrix1[0] : Real
+ map.sample_matrix1[1] : Real
+ map.sample_matrix1[2] : Real
+ map.sample_matrix1[3] : Real
+ map.sample_matrix1[4] : Real
+ map.sample_matrix1[5] : Real
+ map.sample_matrix1[6] : Real
+ map.sample_matrix1[7] : Real
+ map.sample_matrix1[8] : Real

Part IV

Appendices

Appendix A

Glossary

20-sim The 20-sim tool can represent continuous time models in a number of ways. The core concept is that of connected *blocks*.

Abstraction Models may be abstract "in the sense that aspects of the product not relevant to the analysis in hand are not included" [13]. CPS models may reasonably contain multiple levels of abstraction, for representing views of individual constituent systems and for the view of the CPS level. Adapted from [11].

Architecture The term architecture has many different definitions, and range in scope depending upon the scale of the product being 'architected'. In the INTO-CPS project, we use the simple definition from [16]: "an architecture defines the major elements of a system, identifies the relationships and interactions between the elements and takes into account process. An architecture involves both a definition of structure and behaviour. Importantly, architectures are not static but must evolve over time to reflect the change in a system as it evolves to meet changes to its requirements."

Architecture Diagram In the INTO-CPS project, a diagram refers to the symbolic representation of information contained in a model.

Architectural Framework "A defined set of viewpoints and an ontology" and "is used to structure an architecture from the point of view of a specific industry, stakeholder role set, or organisation. [11]. [11].

Architecture Structure Diagram (ASD) The INTO-CPS SysML profile ASDs specialise SysML block definition diagrams to support the specification of a system architecture described in terms of a system's components.

Architecture View "work product expressing the architecture of a system from the perspective of specific system concerns" [16].

Bond graph Bond graphs offer a domain-independent description of a physical system's dynamics, realised as a directed graph. The vertices of these graphs are idealised descriptions of physical phenomena, with their edges (*bonds*) describing energy exchange between vertices.

Co-model "The term *co-model* is used to denote a model comprising a DE model, a CT model and a contract" [8]. A related term ***multi-model*** is a model comprising any combination of constituent DE and CT models.

Code generation Transformation of a model into generated code suitable for compilation into one or more target languages (e.g. C or Java).

Collaborative simulation (co-simulation) The simultaneous, collaborative, execution of models and allowing information to be shared between them. The models may be CT-only, DE-only or a combination of both.

Co-simulation Configuration The configuration that the COE needs to initialise a co-simulation. It contains paths to all FMUs, their inter connection, parameters and step size configuration. When this is combined with a start and end time, a co-simulation can be performed.

Co-simulation Orchestration Engine (COE) The Co-simulation Orchestration Engine combines existing co-simulation solutions (FMUs) and scales them to the CPS level, allowing CPS co-models to be evaluated through co-simulation. The COE will also allow real software and physical elements to participate in co-simulation alongside models, enabling both Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation.

Component The constituent elements of a system.

Connections Diagram (CD) The INTO-CPS SysML profile CDs specialise SysML internal block diagrams to convey the internal configuration of the system's components and the way they are connected.

Constituent Model A constituent model comprising a multi-model.

Continuous Time (CT) model A model with state that can be changed and observed *continuously* [12], and are described using either explicit continuous functions of time either implicitly as a solution of differential equations.

Context In requirements engineering, a ***context*** is the point of view of some system component or domain, or interested stakeholder.

Cyber Physical System (CPS) Cyber-Physical Systems "refer to ICT systems (sensing, actuating, computing, communication, etc.) embedded in physical objects, interconnected (including through the Internet) and providing citizens and businesses with a wide range of innovative applications and services" [9, 10].

Discrete Event (DE) model A model with state that can be changed and observed only at fixed, *discrete*, time intervals [12].

Denotational Semantics Where an operational semantics defines how a program is executed, a denotational approach defines a language in terms of denotations, in the form of abstract mathematical objects, which represent the semantic function that maps over the inputs and outputs of a program [37].

Design Alternatives Where two or more models represent different possible solutions to the same problem. Each choice involves making a selection from alternatives on the basis of criteria that are important to the developer, such as cost or performance. The alternative selected at each point constrains the range of design alternatives that may be viable next steps forward from the current position.

Design Architecture The design architectural model of the system is effectively a multi-model. The INTO-CPS SysML profile [17] is designed to enable the specification of CPS design architectures, which emphasises a decomposition of a system into ***subsystems***, where each subsystem is modelled separately in isolation using a special notation and tool designed for the domain of the subsystem.

Design Parameter A *design parameter* is a property of a model that can be used to affect the model's behaviour, but that remains constant during a given simulation [8].

Design Space "The *design space* is the set of possible solutions for a given design problem" [8].

Design-Space Exploration (DSE) "an activity undertaken by one or more engineers in which they build and evaluate co-models in order to reach a design from a set of requirements" [8].

Effort and Flow The energy exchanged in 20-sim is the product of *effort* and *flow*, which map to different concepts in different domains, for example voltage and current in the electrical domain.

Environment A system's *environment* is everything outside of the system. The behaviour exhibited by the environment is beyond the direct control of the developer [8].

Evolution This refers to the ability of a system to benefit from a varying number of alternative system components and relations, as well as its ability to gain from the adjustments of the individual components' capabilities over time (Adjusted from SoS [18]).

Foundations Developer An individual who uses the developed foundations and associated tool support (see Section 2.6) to reason about the development of tools.

Functional Mockup Interface (FMI) The Functional Mock-up Interface (FMI) is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code [19].

Functional Mockup Unit (FMU) Component that implements FMI is a Functional Mockup Unit (FMU) [19].

Hardware-in-the-Loop (HiL) Testing In ***HiL*** there is (target) hardware involved, thus the FMU representing the hardware in a co-simulation is mainly a wrapper that interacts (timed) with this hardware; it is perceivable that realisation heavily depends on hardware interfaces and timing properties.

Holistic Architecture The aim of a holistic architecture is to identify the main units of functionality of the system reflecting the *terminology and structure of the domain of application*. It describes a conceptual model that highlights the main units of the system architecture and the way these units are connected with each other, taking a holistic view of the overall system.

Hybrid-CSP This is a continuous version of CSP defined originally by He Jifeng [40]. It will be used as a basis to inform the design of INTO-CSP.

Hybrid Model A model which contains both DE and CT elements.

Interface "Defines the boundary across which two entities meet and communicate with each other" [11]. Interfaces may describe both digital and physical interactions: digital interfaces contain descriptions of operations and attributes that are *provided* and *required* by components. Physical interfaces describe the flow of physical matter (for example fluid and electrical power) between components.

INTO-CPS Application The INTO-CPS Application is a front-end to the INTO-CPS tool chain. The application allows the specification of the co-simulation configuration to be orchestrated by the COE, and the co-simulation execution itself. The application also provides access to features of the tool chain without an existing user interface (such as design space exploration and model checking).

INTO-CPS tool chain The INTO-CPS tool chain is a collection of software tools, based centrally around FMI-compatible co-simulation, that supports the collaborative development of CPSs.

INTO-CSP A version of CSP, which will be used to provide a model for the SysML-FMI profile, FMI, VDM-RT and Modelica semantics. It is a front end for a UTP theory of reactive concurrent continuous systems customised for the needs of INTO-CPS.

Master Algorithm A Master Algorithm (MA) controls the data exchange between FMUs and the synchronisation of all simulation solvers [19].

Model A potentially partial and abstract description of a system, limited to those components and properties of the system that are pertinent to the current goal [11]. "A model is a simplified description of a system, just complex enough to describe or study the phenomena that are relevant for our problem context" [12]. A model "may contain representations of the system, environment and stimuli" [14]

Model Checking (MC) An analysis technique that exhaustively checks whether the model of the system meets its specification [25], which is typically expressed in some temporal logic such as ***Linear Time Logic (LTL)*** [26] or ***Computation Tree Logic (CTL)*** [27].

Model Description The model description file is an XML file that supplies a description of all properties of a model (for example input/output variables) [19].

Model-in-the-Loop (MiL) Testing in ***MiL*** the test object of the test execution is a (design) model, represented by one or more FMUs. This is similar to the SiL (if e.g., the SUT is generated from the design model), but MiL can also imply that running the SUT-FMU has a representation on model level; e.g., a playback functionality in the modelling tool could some day be used to visualise a test run.

Modelling "The activity of creating models" [14]. See also **co-modelling** and **multi-modelling**.

Modelica Modelica is an "object-oriented language for modelling of large, complex, and heterogeneous physical systems" [33]. Modelica models are described by ***schematics***, also called ***object diagrams***, which consist of connected components. Components are connected by ports and are defined by sub components or a textual description in the Modelica language.

Multi-model "A model comprising *multiple* constituent DE and CT models".

Non-dominated Set The Non-dominated set (NDS) is the current set of best results according to a Pareto analysis. For a result exist in the NDS it must be true that it is not possible to find another result in the set of all results that improves on one property of the result without degrading another property of the result.

Non-functional Property Non-functional properties (NFPs) pertain to characteristics other than functional correctness. For example, reliability, availability, safety and performance of specific functions or services are NFPs that are quantifiable. Other NFPs may be more difficult to measure [15].

Objective Criteria or constraints that are important to the developer, such as cost or performance

Port 20-sim blocks may have input and output *ports* that allow data to be passed between them. In SysML, blocks own ports — the points of interaction between blocks.

Proof The process of showing how the validity of one statement is derived from others by applying justified rules of inference [43].

Provenance "Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness." [21].

Refinement Refinement is a verification and formal development technique pioneered by [41] and [42]. It is based on a behaviour preserving relation that allows the transformation of an abstract specification into more and more concrete models, potentially leading to an implementation.

Requirement A requirement is a statement of need and may impose restrictions, define system capabilities or identify qualities of a system and should indicate some value or use for the different stockholders of a CPS.

Requirements Engineering (RE) The process of the specification and documentation of requirements placed upon a CPS.

Semantics Describes the meaning of a (grammatically correct) language [35].

Software-in-the-Loop (SiL) Testing In ***SiL*** testing the object of the test execution is an FMU that contains a software implementation of (parts of) the system. It can be compiled and run on the same machine that the COE runs on and has no (defined) interaction other than the FMU-interface.

SoS-ACRE System of Systems Approach to Context-based Requirements Engineering [56], an approach adapted from standard systems engineering, tailored for systems of systems (SoSs).

Structural Operational Semantics (SOS) Describes how the individual steps of a program are executed on an abstract machine [36]. An SOS definition is akin to an interpreter in that it provides the meaning of the language in terms of relations between beginning and end states. The relations are defined on a per-construct basis. Accompanying the relations are a collection of semantic rules which describe how the end states are achieved.

SysML The systems modelling language (SysML) [34] extends a subset of the Unified Modelling language (UML) to support modelling of heterogeneous systems.

System "A combination of interacting elements organized to achieve one or more stated purposes" [7].

System boundary The *system boundary* is the common frontier between the system and its environment. System boundary definition is application-specific [8].

System of Systems (SoS) “A System of Systems (SoS) is a collection of constituent systems that pool their resources and capabilities together to create a new, more complex system which offers more functionality and performance than simply the sum of the constituent systems” [11]. CPSs may exhibit the characteristics of SoSs.

System Under Test “The system currently being tested for correct behaviour. An alias for system of interest, from the point of view of the tester. The same concept can be extended from systems engineering to SoS engineering, changing the focus from a single system of interest to an SoS under test.
The system of systems currently being tested for correct behaviour” [11].

Test Automation Test Automation (TA) is defined as the machine assisted automation of system tests. In INTO-CPS we concentrate on various forms of *model-based testing*, centering on testing system models against their requirements.

Test Case A finite structure of input and expected output [22].

Test model Specifies the expected behaviour of a system under test. Note that a test model can be different from a design model. It might only describe a part of a system under test that is to be tested and it can describe the system on a different level of abstraction [23].

Test procedures Detailed instructions for the set-up and execution of a set of test cases, and instructions for the evaluation of results of executing the test cases [24, 23].

Test suite A collection of test procedures.

Tool Chain User An individual who uses the INTO-CPS Tool Chain and its various analysis features.

Traceability The association of one model element (e.g. requirements, design artefacts, activities, software code or hardware) to another. ***Requirements traceability*** “refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction” [20].

Unifying Theories of Programming (UTP) The Unifying Theories of Programming (UTP) [38] is a technique to for describing language semantics in a unified framework. A theory of a language is composed of an ***alphabet***, a ***signature*** and a collection of ***healthiness conditions***.

Variable A *variable* is feature of a model that may change during a given simulation [8].

VDM-RT VDM-RT is based upon the ***object-oriented*** paradigm where a model is comprised of one or more ***objects***. An object is an instance of a ***class*** where a class gives a definition of zero or more ***instance variables*** and ***operations*** an object will contain. Instance variables define the identifiers and types of the data stored within an object, while operations define the behaviours of the object.

Workflow A sequence of **activities** performed to aid in modelling. A workflow has a defined purpose, and may cover a subset of the CPS engineering development lifecycle.