Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

INTO-CPS

INTO-CPS Tool Chain User Manual

Deliverable Number: D4.3a

Version: 0.01

Date: December, 2017

Public Document

http://into-cps.au.dk

Contributors:

Victor Bandur, AU
Peter Gorm Larsen, AU
Kenneth Lausdahl, AU
Casper Thule, AU
Anders Franz Terkelsen, AU
Carl Gamble, UNEW
Adrian Pop, LIU
Etienne Brosse, ST
Jörg Brauer, VSI
Florian Lapschies, VSI
Marcel Groothuis, CLP
Christian Kleijn, CLP
Luis Diogo Couto, UTRC

Editors:

Victor Bandur, AU

Reviewers:

TBD

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
0.01	11-01-2017	Victor Bandur	Initial version.

Abstract

This deliverable is the user manual for the INTO-CPS tool chain, an update of deliverable D4.2a [BLL+15]. It is targeted at those wishing to make use of the INTO-CPS technology to design and validate cyber-physical systems. As a user manual, this deliverable is concerned with those aspects of the tool chain relevant to end-users, so it is necessarily high-level. Other deliverables discuss finer details of individual components, including theoretical foundations and software design decisions. Readers interested in this perspective on the tool chain should consult deliverables D4.2b [PBLG16], D4.2c [BQ16], D4.2d [LNH+16], D5.2a [PLM16], D5.2b [BLM16], D5.2c [BHPG16], D5.2d [Gam16], D2.2a [ACM+16], D2.2b [FCC+16], D2.2c [CFTW16] and D2.2d [CW16].

Contents

1 Introduction

This deliverable is the user manual for the INTO-CPS tool chain. The tool chain supports a model-based development and verification approach for Cyber-Physical Systems (CPSs). Development of CPSs with the INTO-CPS technology proceeds with the development of constituent models using established and mature modelling tools. Development also benefits from support for Design Space Exploration (DSE). The analysis phase is primarily based on co-simulation of heterogeneous models compliant with version 2.0 of the Functional-Mockup Interface (FMI) standard for co-simulation [Blo14]. Other verification features supported by the tool chain include hardware- and software-in-the-loop (HiL and SiL) simulation and model-based testing. Presently there is limited support for Linear Temporal Logic model checking of discrete models, with further model checking support being developed.

All INTO-CPS tools can be obtained from

`http://into-cps.github.io`

This is the primary source of information and help for users of the INTO-CPS tool chain. The structure of the website follows the natural flow of CPS development with INTO-CPS, and serves as a natural aid in getting started with the technology. In case access to the individual tools is required, pointers to each are also provided.

Please note: This user manual assumes that the reader has a good understanding of the FMI standard. The reader is therefore strongly encouraged to become familiar with Section 2 of deliverable 4.1d [LLW+15] for background, concepts and terminology related to FMI.

The rest of this manual is structured as follows:

- Section 2 provides an overview of the different features and components of the INTO-CPS tool chain.
- Section 3 explains the relevant parts of the Modelio SysML modelling tool.
- Section 4 explains the different features of the main user interface of the INTO-CPS tool chain, called the INTO-CPS Application.
- Section 5 describes the separate modelling and simulation tools used in elaborating and verifying the different constituent models of a multi-model.

- Design Space Exploration (DSE) for INTO-CPS multi-models is presented in Section 6.
- Section 7 describes model-based test automation and model checking in the INTO-CPS context.
- Section 9 provides a short overview of code generation in the INTO-CPS context.
- The appendices are structured as follows:
  - Appendix A lists the acronyms used throughout this deliverable.
  - Appendix B gives background information on the individual tools making up the INTO-CPS tool chain.
  - Appendix C describes how the individual tools can be obtained.
  - Appendix D gives background information on the various principles underlying the INTO-CPS tool chain.

2 Overview of the INTO-CPS Tool Chain

The INTO-CPS tool chain consists of several special-purpose tools from a number of different providers. Note that it is an open tool chain so it is possible to incorporate other tools that also support the FMI standard for co-simulation and we have already tested this with numerous external tools (both commercial as well as open-source tools). The constituent tools are dedicated to the different phases of co-simulation activities. They are discussed individually through the course of this manual. An overview of the tool chain is shown in Figure 1. The main interface to an INTO-CPS co-simulation activity is the INTO-CPS Application. This is where the user can design co-simulations from scratch, assemble them using existing FMUs and configure how simulations are executed. The result is a co-simulation *multi-model*.

The design of a multi-model is carried out visually using the Modelio SysML tool, in accordance with the SysML/INTO-CPS profile described in D2.2a [ACM+16]. Here one can either design a multi-model from scratch by specifying the characteristics and connection topology of Functional Mockup Units (FMUs) yet to be developed, or import existing FMUs so that the connections between them may be laid out visually. The result is a SysML multi-model of the entire co-simulation, expressed in the SysML/INTO-CPS profile. In the

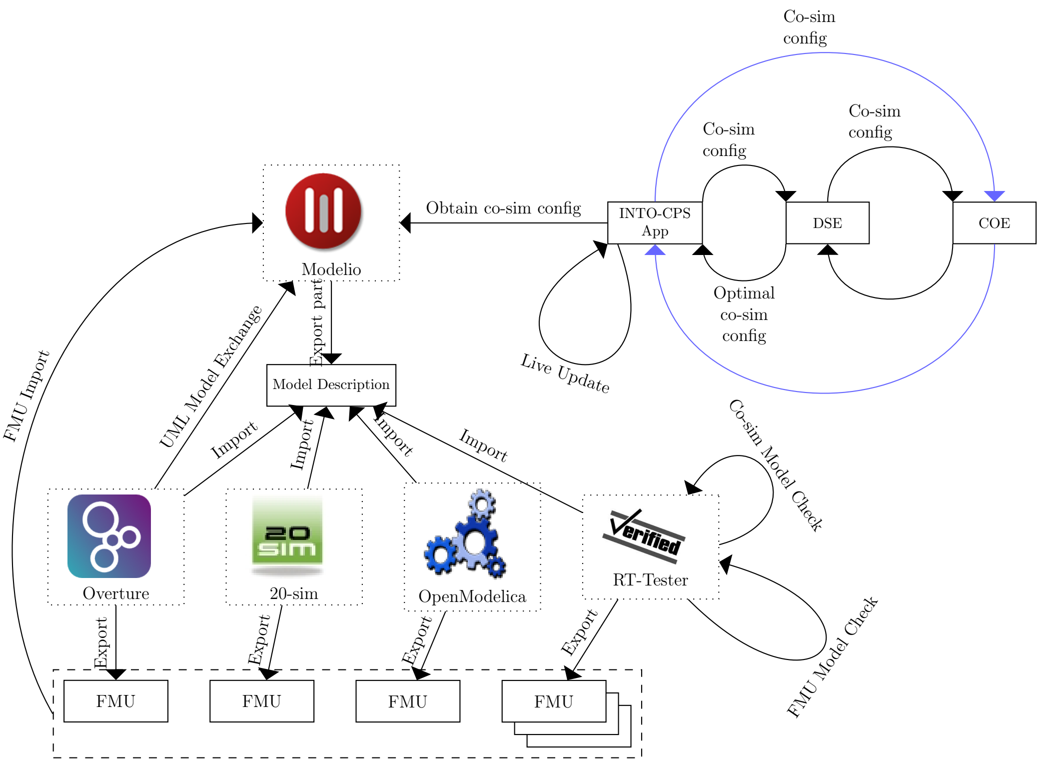


Figure 1: Overview of the structure of the INTO-CPS tool chain.

former case, where no FMUs exist yet, a number of `modelDescription.xml` files are generated from this multi-model which serve as the starting point for constituent model construction inside each of the individual simulation tools, leading to the eventual FMUs.

Once a multi-model has been designed and populated with concrete FMUs, the Co-simulation Orchestration Engine (COE) can be invoked to execute the co-simulation. The COE controls all the individual FMUs in order to carry out the co-simulation. In the case of tool-wrapper FMUs, the model inside each FMU is simulated by its corresponding simulation tool. The tools involved are Overture [LBF⁺10], 20-sim [Con13] and OpenModelica [Lin15]. RT-Tester is not under the direct control of the COE at co-simulation time, as its purpose is to carry out testing and model checking rather than simulation. The user can control a co-simulation, for instance by running it with different simulation parameter values and observing the effect of the different values on the co-simulation outcome.

Alternatively, the user has the option of exploring optimal simulation parameter values by entering a Design Space Exploration phase. In this mode, ranges are defined for various parameters which are explored, in an intelligent way, by a design space exploration engine that searches for optimal parameter values based on defined optimization conditions. This engine interacts directly with the COE and itself controls the conditions under which the co-simulation is executed.

3 Modelio and SysML for INTO-CPS

The INTO-CPS tool chain supports a model-based approach to the development and validation of CPS. The Modelio tool and its SysML/INTO-CPS profile extension provide the diagramming starting point. This section describes the Modelio extension that provides INTO-CPS-specific modelling functionality to the SysML modelling approach.

The INTO-CPS extension module is based on the Modelio SysML extension module, and extends it in order to fulfill INTO-CPS modelling requirements and needs. Figure 2 shows an example of a simple INTO-CPS Architecture Structure Diagram under Modelio. This diagram shows a *System*, named

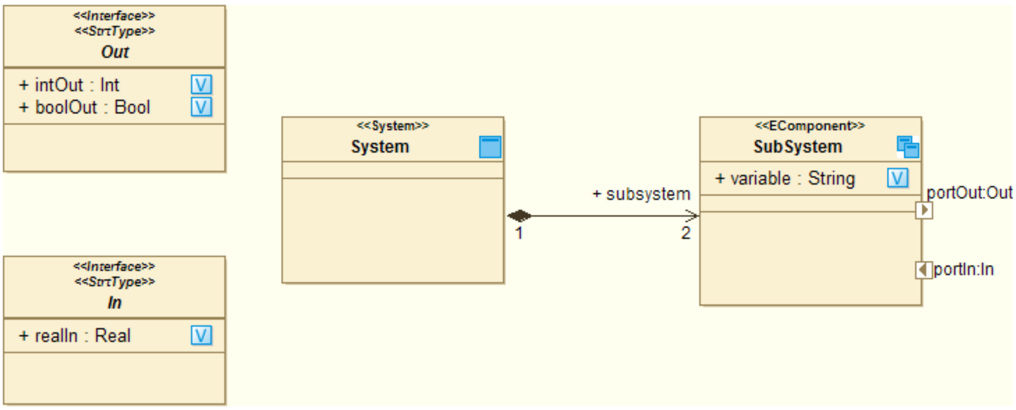


Figure 2: Example INTO-CPS multi-model.

"System"[1], composed of two *EComponents* of kind *Subsystem*, named "SubSystem"[2]. These *Subsystems* have an internal *Variable* called "variable" of type *String* and expose two *FlowPorts* named "portIn" and "portOut". The type of data going through these ports is respectively defined by types *In*

[1]An abstract description of an INTO-CPS multi-model.
[2]Abstract descriptions of INTO-CPS constituent models.

and *Out* of kind *StrtType*. More details on the SysML/INTO-CPS profile can be found in deliverable D2.2a [ACM+16].

Figure 3 illustrates the main graphical interface after Modelio and the INTO-CPS extension have been installed. Of all the panes, the following three are

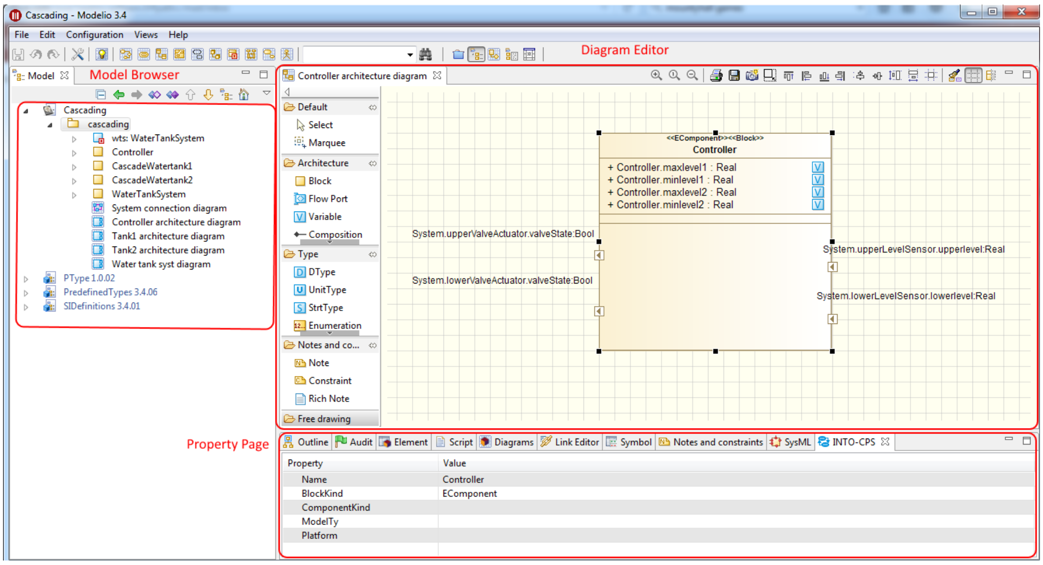


Figure 3: Modelio for INTO-CPS.

most useful in the INTO-CPS context.

1. The Modelio model browser, which lists all the elements of your model in tree form.
2. The diagram editor, which allows you to create INTO-CPS design architectures and connection diagrams.
3. The INTO-CPS property page, in which values for properties of INTO-CPS subsystems are specified.

3.1 Creating a New Project

In the INTO-CPS Modelling workflow described in Deliverable D3.2a [FGPP16], the first step will be to create, as depicted in Figure 4, a Modelio project:

1. Launch Modelio.
2. Click on *File* → *Create a project...*.

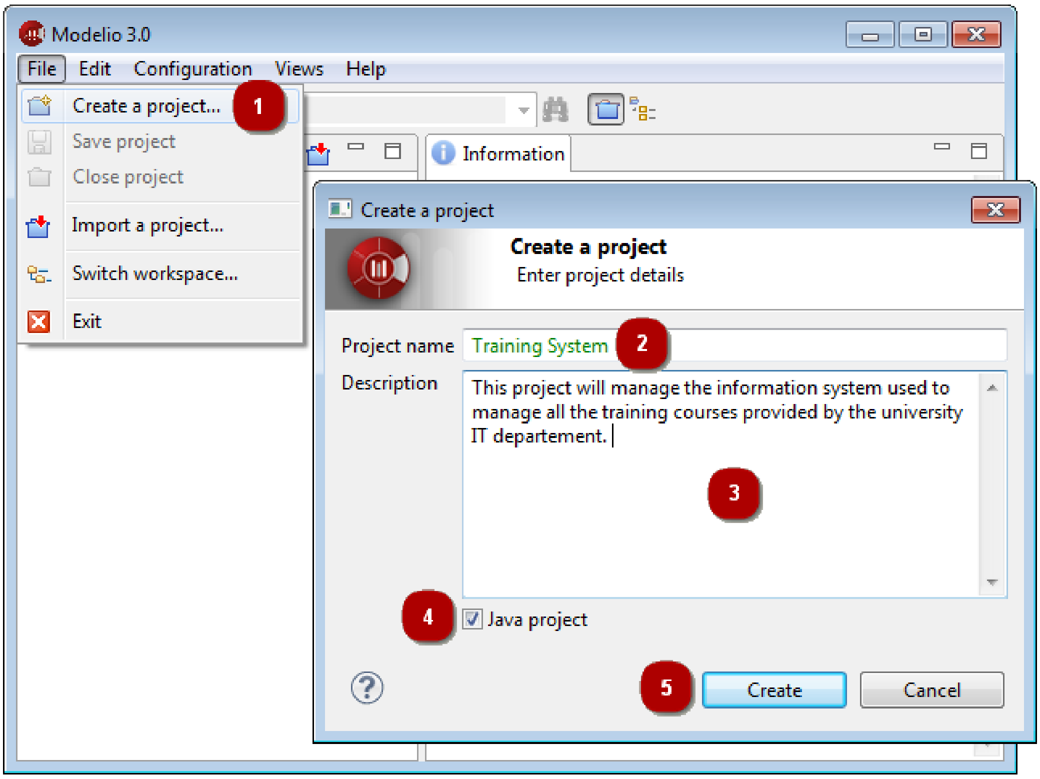


Figure 4: Creating a new Modelio project.

3. Enter the name of the project.

4. Enter the description of the project.

5. If it is envisaged that the project will be connected to a Java development workflow in the future (unrelated to INTO-CPS), you can choose to include the Java Designer module by selecting *Java Project*, otherwise de-select this option.

6. Click on *Create* to create and open the project.

Once you have successfully created a Modelio project, you have to install the Modelio extensions required for INTO-CPS modelling, *i.e.* both Modelio SysML and INTO-CPS extensions, as described at

`http://into-cps.github.io`

If both modules have been correctly installed, you should be able to create, under any package, an INTO-CPS Architecture Structure Diagram in order to model the first subsystem of your multi-model. For that, in the Modelio model browser, right click on a *Package* element then in the *INTO-CPS* entry, choose *Architecture Structure Diagram* as shown in Figure 5. Figure 6 represents an example of an Architecture Structure Diagram. Besides

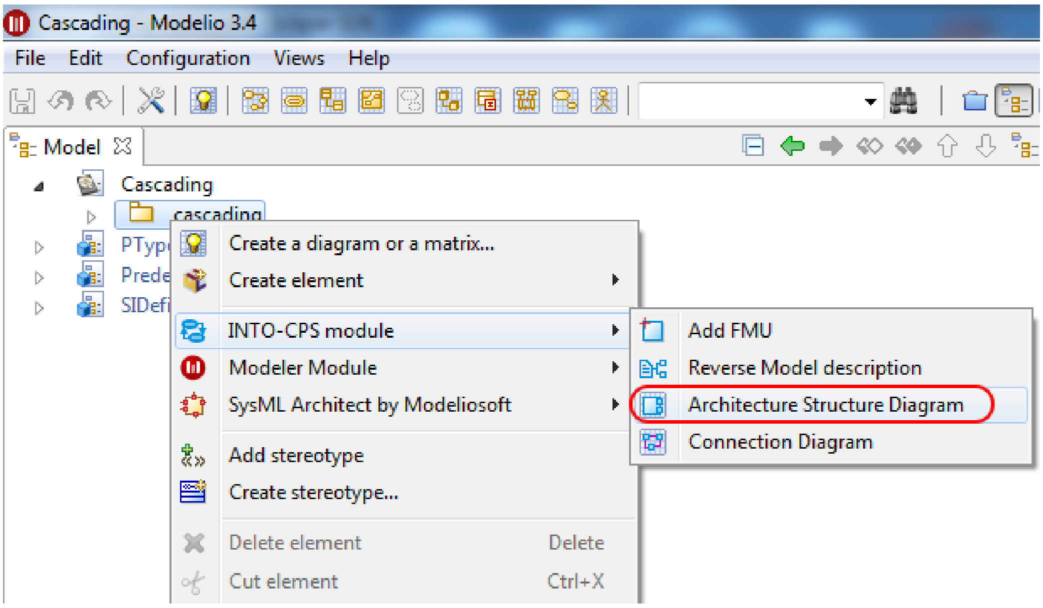


Figure 5: Creating an Architecture Structure diagram.

creating an Architecture Structure Diagram from scratch, the INTO-CPS extension allows the user to create it from an existing `modelDescription.xml` file. A `modelDescription.xml` file is an artifact defined in the FMI standard which specifies, in XML format, the public interface of an FMU. To import a `modelDescription.xml` file,

1. Right click in the Modelio model browser on a *Package* element, then in the *INTO-CPS* entry choose *Import Model description*, as shown in Figure 7.

2. Select the desired `modelDescription.xml` file in your installation and click on *Import* (Figure 8).

This import command creates an Architecture Structure Diagram describing the interface of an INTO-CPS *block* corresponding to the `modelDescription.xml` file imported, *cf.* Figure 9. Once you have created several such blocks, either from scratch or by importing `modelDescription.xml` files, you must eventually connect instances of them in an INTO-CPS Connection Diagram. To create an INTO-CPS Connection diagram, as for an INTO-CPS Architecture Structure Diagram, right click on a *Package* element, then

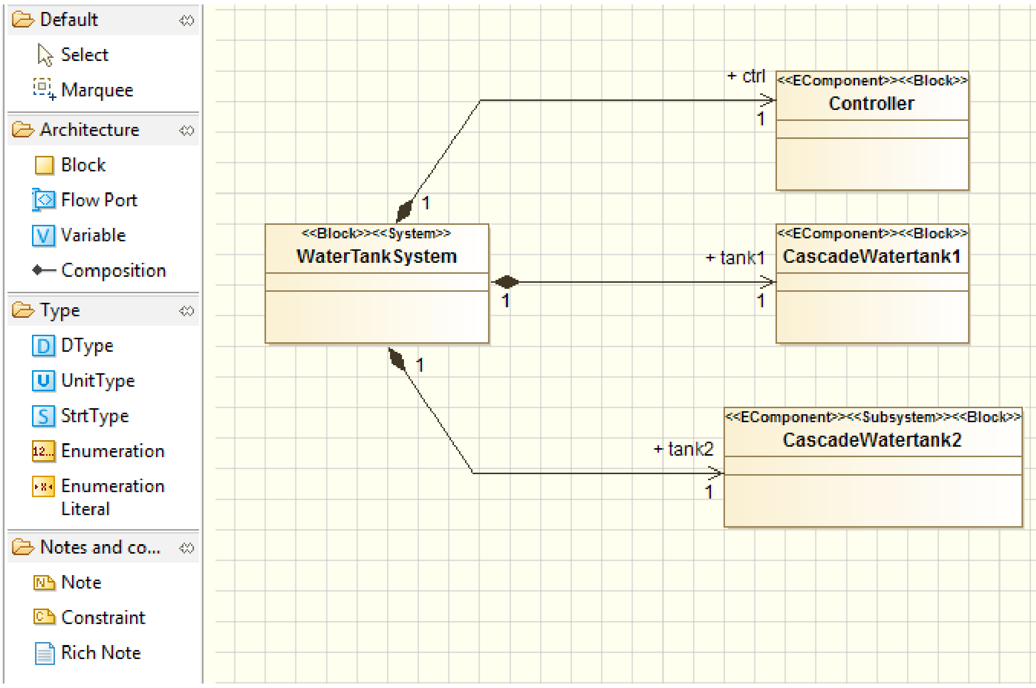


Figure 6: Example Architecture Structure diagram.

in the *INTO-CPS* entry choose *Connection Diagram*, as shown in Figure 10. Figure 11 shows the result of creating such a diagram. Once you have created all desired block instances and their ports by using the dedicated command in the Connection Diagram palette, you will be able to model their connections by using the connector creation command (Figure 12). At this point your blocks have been defined and the connections have been set. The next step is to simulate your multi-model using the app. For that you must first generate a configuration file from your Connection diagram. Select the desired Connection diagram, right click on it and in the *INTO-CPS* entry choose *Generate configuration*, as shown in Figure 13. In the final step, choose a relevant name and click on *Generate*.

3.2 Exporting `modelDescription.xml` Files

The SysML Connection diagram defines the components of the system and their connections. The internals of these block instances are created in the various modeling tools and exported as FMUs. The modeling tools Overture, 20-sim and OpenModelica support importing the interface definition (ports) of the blocks in the Connection diagram by importing a

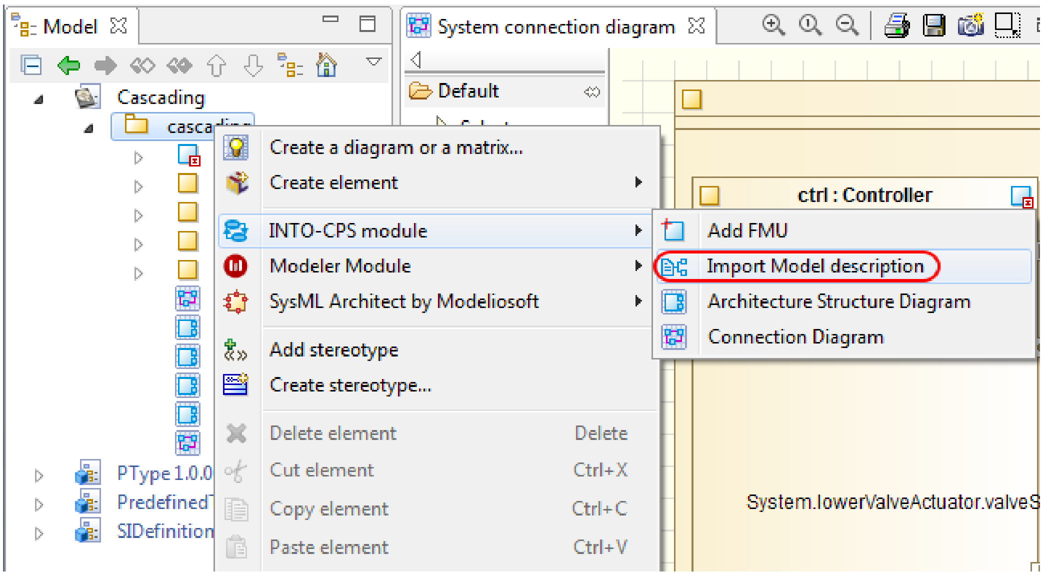


Figure 7: Importing an existing model description.

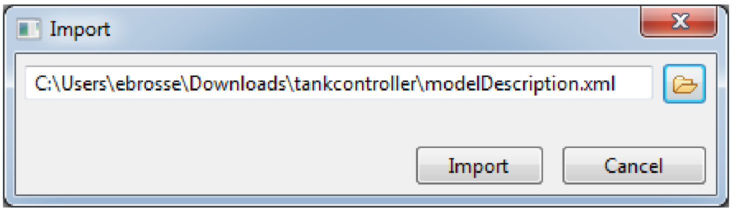


Figure 8: Model description selection.

`modelDescription.xml` file containing the block name and its interface definition.

Follow these steps to export a `modelDescription.xml` file from Modelio:

1. In Modelio, right-click on the model block in the tree.
2. Select *INTO-CPS* → *Generate Model Description* (see Figure 14).
3. Choose a file name containing the text "modelDescription.xml" and click *Export* (see Figure 15).

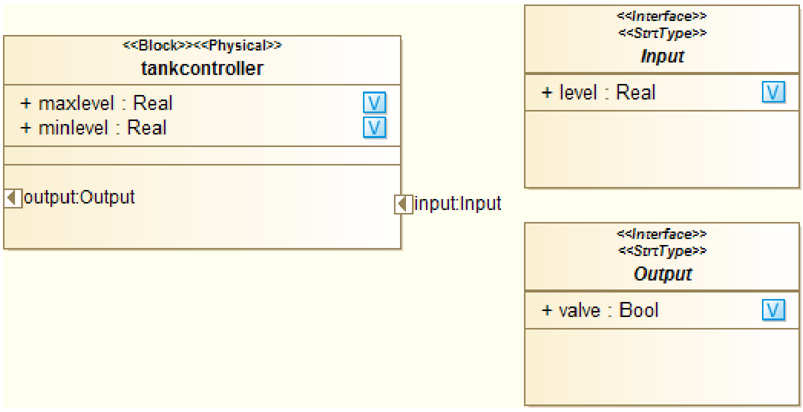


Figure 9: Result of model description import.

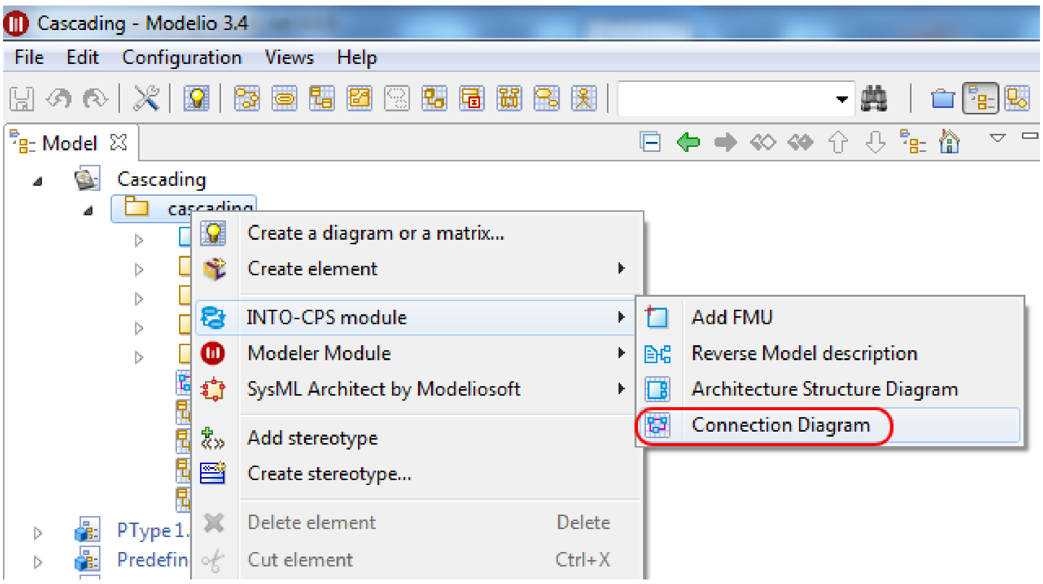


Figure 10: Creating a Connection diagram.

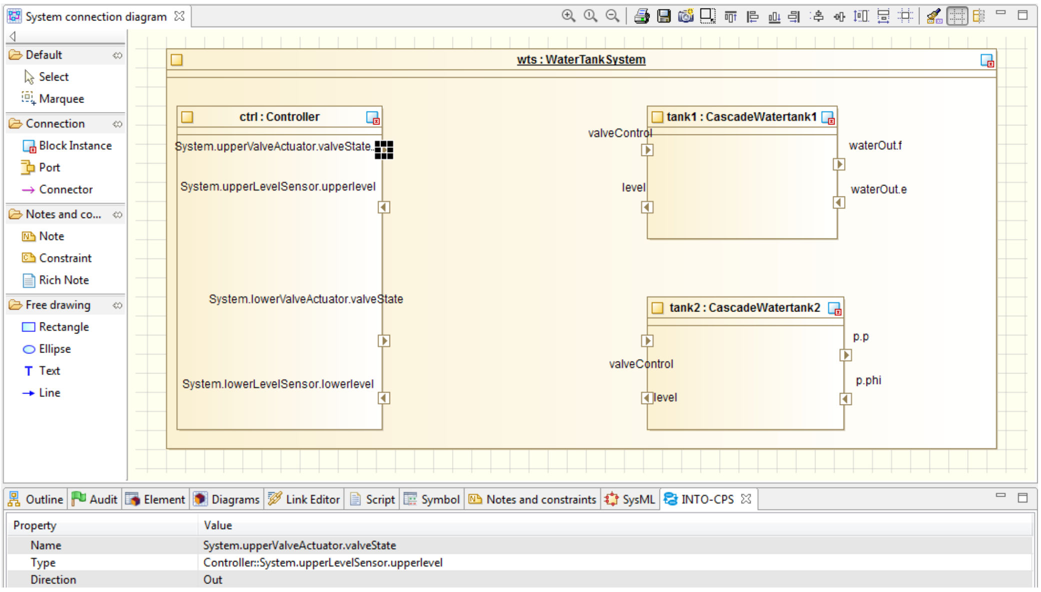


Figure 11: Unpopulated Connection diagram.

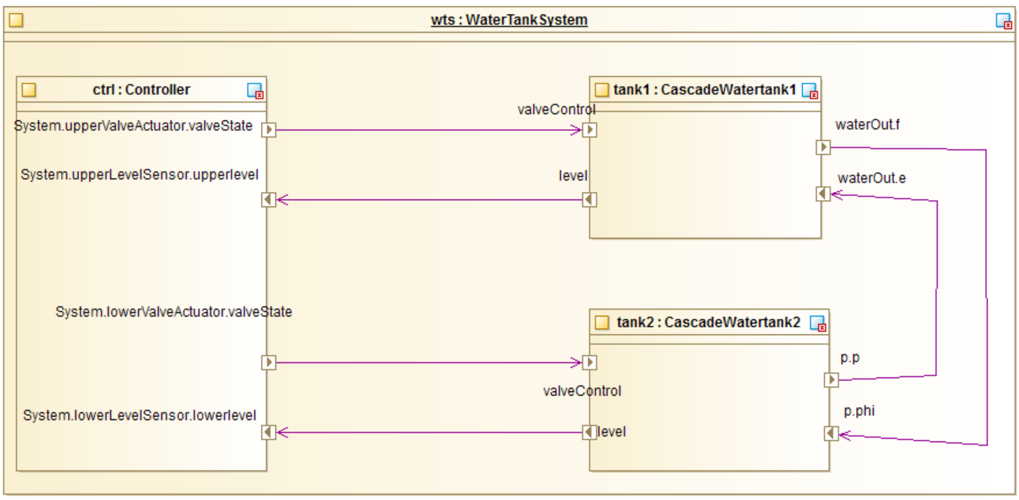


Figure 12: Populated Connection diagram.

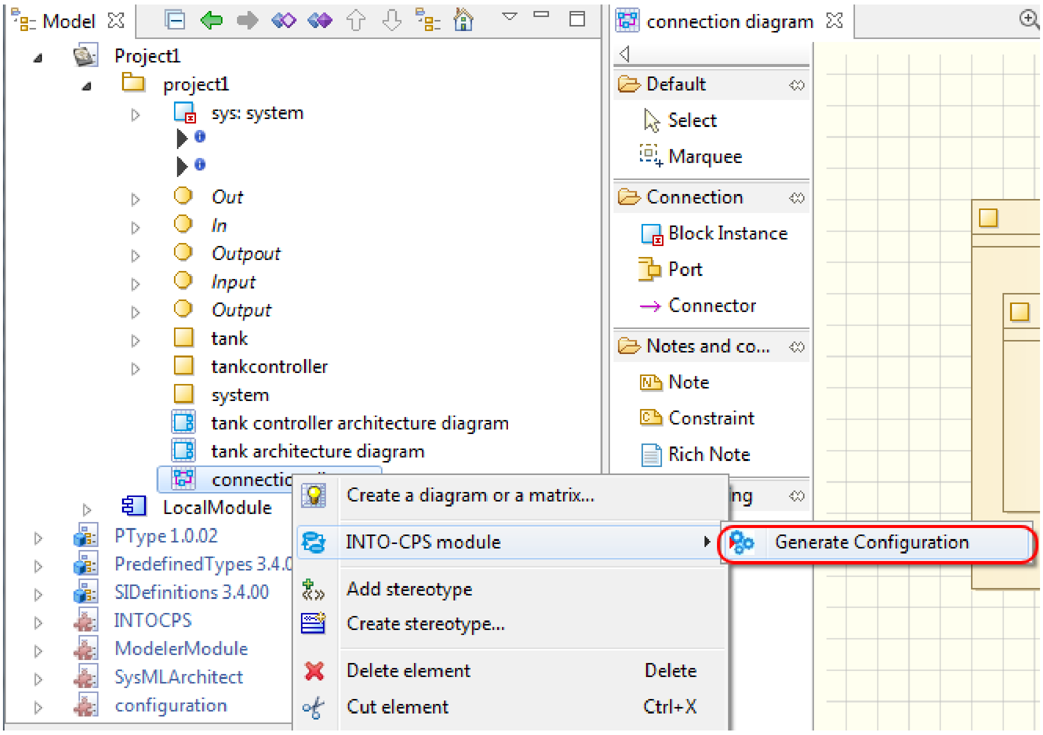


Figure 13: Generating a configuration file.

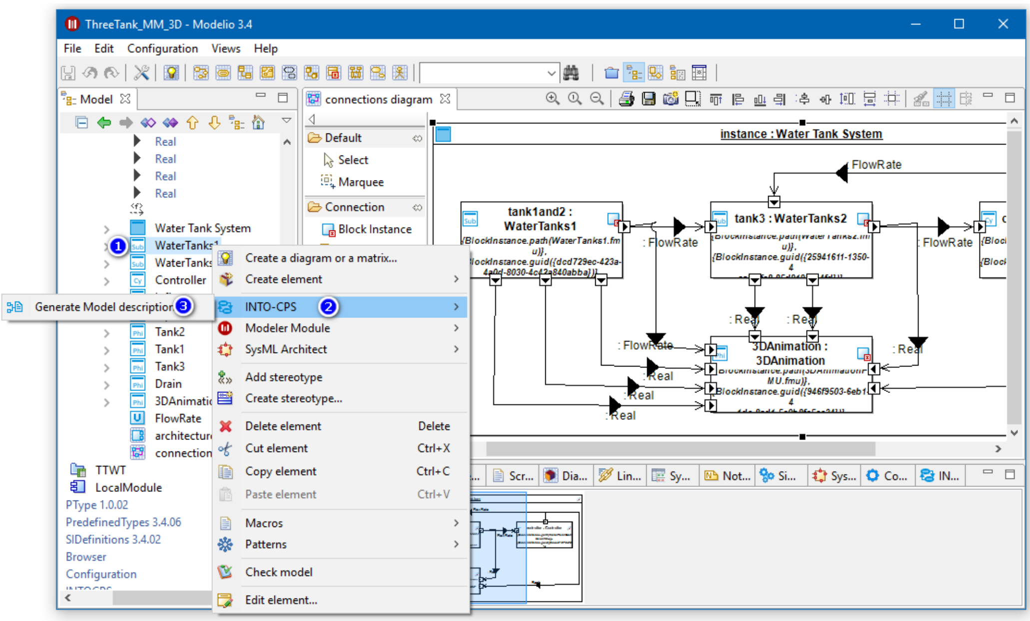


Figure 14: Exporting a `modelDescription.xml` file.

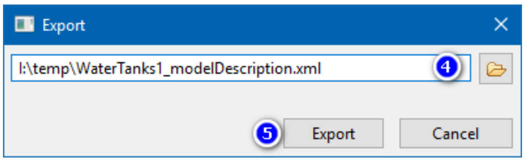


Figure 15: Naming the model description file.

4 The INTO-CPS Application

This section describes the INTO-CPS Application(here referred to as *the app*), the primary gateway to the INTO-CPS tool chain. Section 4.1 gives an introductory overview of the app. Section 4.2 describes how the app can be used to create new INTO-CPS co-simulation projects. Section 4.3 describes how multi-models can be assembled. Section 4.4 describes how co-simulations are configured, executed and visualized. Section 4.5 lists some additional useful features of the app, while Section 4.6 describes how the co-simulation engine itself can be started manually, for specialist use.

4.1 Introduction

The app is the front-end of the entire INTO-CPS tool chain. The app defines a common INTO-CPS project and it is the easiest way to configure and execute co-simulations. Certain features in the tool chain are only accessible through the app. Those features will be explained in their own sections of the user manual. This section introduces the app and its basic features only.

Releases of the app can be downloaded from:

`https://github.com/into-cps/intocps-ui/releases`

Four variants are available:

- `-darwin-x64.zip` – MacOS version
- `-linux-x64.zip` – Linux (64 bit) version
- `-win32-ia32.zip` – Windows (32 bit) version
- `-win32-x64.zip` – Windows (64 bit) version

The app itself has no dependencies and requires no installation. Simply unzip it and run the executable. However, certain app features require Git[3] and Java 8[4] to be already installed.

[3] `https://git-scm.com/`

[4] `http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`

4.2 Projects

An INTO-CPS project contains all the artifacts used and produced by the tool chain. The project artifacts are grouped into folders. You can create as many folders as you want and they will all be displayed in the project browser. The default set of folders for a new project, shown in Figure 16, is:

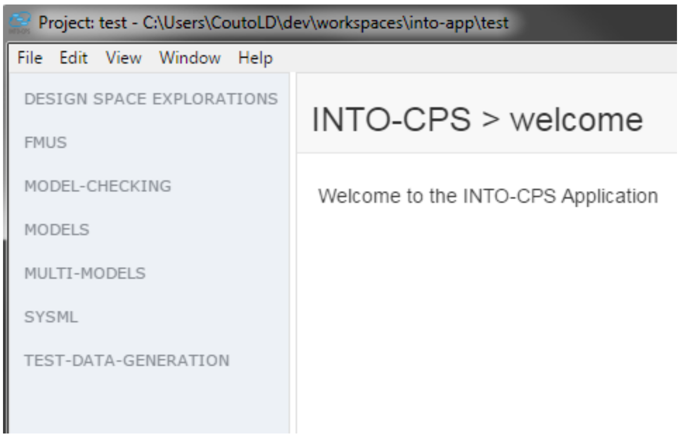


Figure 16: INTO-CPS project shown in the project browser.

Design Space Explorations Scripts and configuration files for performing DSE experiments.

FMUs FMUs for the constituent models of the project.

Model Checking Configuration files for performing Model Checking experiments.

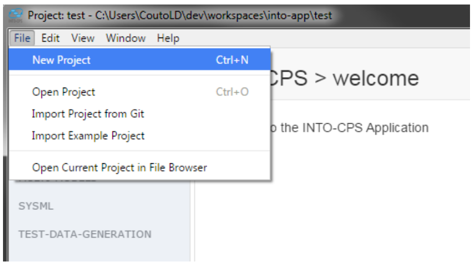
Models Sources for the constituent models of the project.

Multi-Models The multi-models of the project, using the project FMUs. This folder also holds configuration files for performing co-simulations.

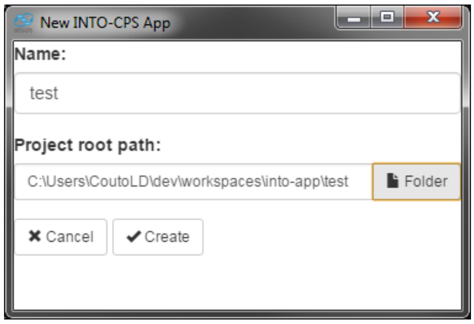
SysML Sources for the SysML model that defines the architecture and connections of the project multi-model.

Test-Data-Generation Configuration files for performing test data generation experiments.

In order to create a new project, select *File* → *New Project*, as shown in Figure 17a. This opens the dialog shown in Figure 17b, where you must choose the project name and location – the chosen location will be the root

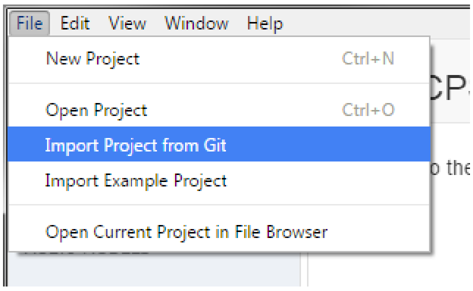


(a) *New Project* menu entry.

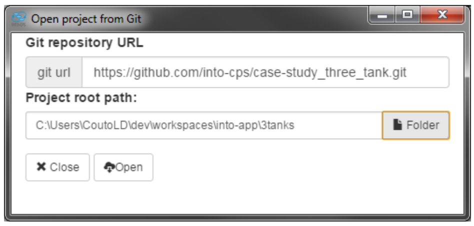


(b) *New Project* dialog.

Figure 17: Creating a new INTO-CPS project.



(a) *Import Git Project* menu entry.

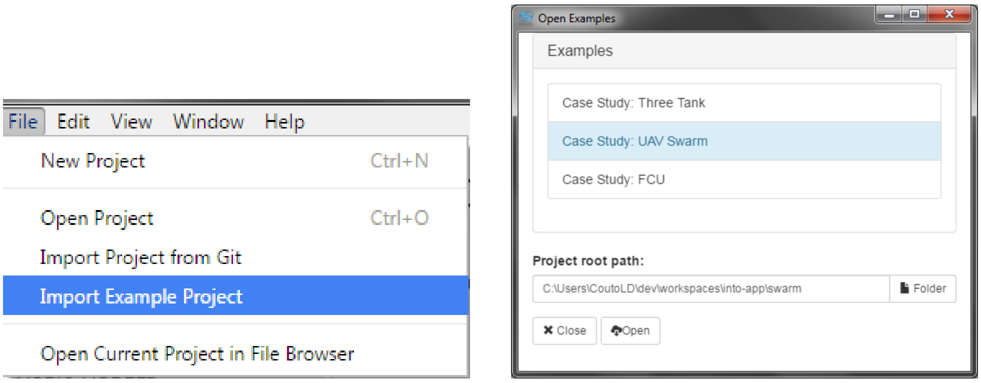


(b) *Import Git Project* dialog.

Figure 18: Importing a Git project.

of the project, so you should manually create a new folder for it. To open an existing project, select *File* → *Open Project*, then navigate to the project's root folder and open it.

To import a project stored in the Git version control system, select *File* → *Import Project from Git*, as shown in Figure 18a. This opens the dialog shown in Figure 18b, where you must choose the project location and also provide the Git URL. The project is checked out using Git, so any valid Git URL will work. You must also have Git available in your PATH environment variable in order for this feature to work. It is possible to import several public example projects that show off the various features of the INTO-CPS tool chain. These examples are described in Deliverable D3.5 [PGP+16]. To import an example, select *File* → *Import Example Project*, as shown in Figure 19a. This opens the dialog box shown in Figure 19b, where you must select which example to import and a project location. The example is checked out via Git, so you must have Git available in your path in order for this feature to work. For both Git projects and examples, once you begin the import



(a) *Import Example Project* menu. (b) *Import Example Project* dialog.

Figure 19: Importing examples.

process, a process dialog is displayed, as shown in Figure 20.

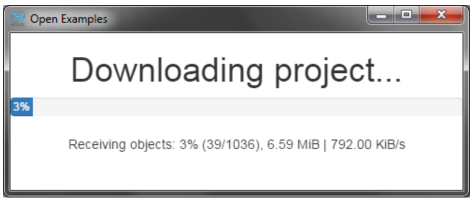


Figure 20: Progress of project imports through Git.

4.3 Multi-Models

For any given project, the app allows you to create and edit multi-models and co-simulation configurations. To create a new multi-model, right click the *Multi-models* node in the project browser and select *New multi-model*, as shown in Figure 21. After creation, the new multi-model is automatically opened for editing. To select an existing multi-model for editing, double-click it. Once a multi-model is open, the multi-model view, shown in Figure 22 is displayed. The top box, *Overview*, displays an overview of the input and output variables in the FMUs, as shown in Figure 23. The bottom box, *Configuration*, enables the user to configure the multi-model. In order to configure a multi-model, it must first be unlocked for editing by clicking the *Edit* button at the bottom of the *Configuration* box. There are four main areas dedicated to configuring various aspects of a multi-model.

The *FMUs* area, shown in Figure 24, allows you to remove or add FMUs

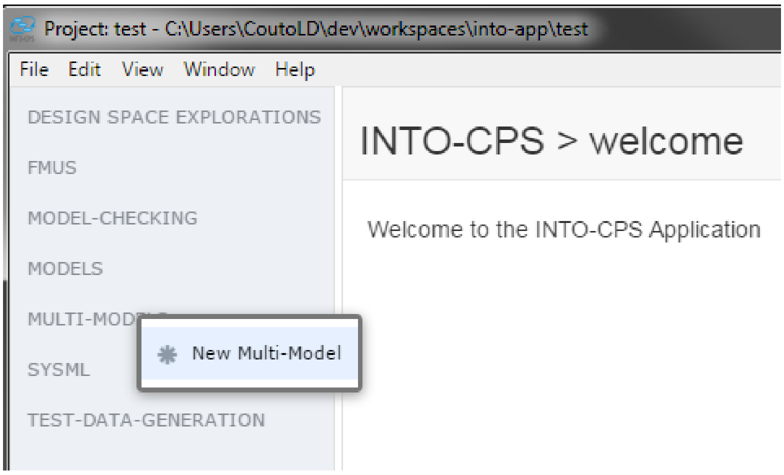


Figure 21: Creating a new multi-model.

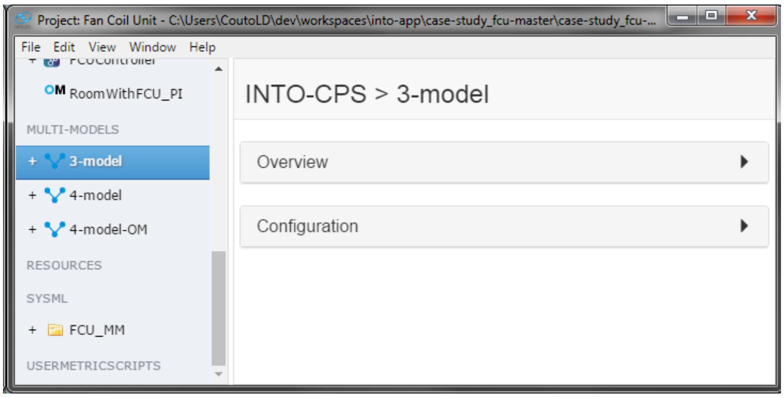


Figure 22: Main multi-model view.

and to associate the FMUs with their files by browsing to, or typing, the path of the FMU file. For each FMU file a marker is displayed indicating whether the FMU is supported by the app and can be used for co-simulation on the current platform. The *FMU instances* area, shown in Figure 25, allows you to create or remove FMU instances and name them. A multi-model consists of one or more interconnected instances of various FMUs. More than one instance may be created for a given FMU. As a convenient workflow shortcut, the *Connections* area, shown in Figure 26, allows you to connect output variables from an FMU instance into input variables of another:

1. Click the desired output FMU instance in the first column. The output variables for the selected FMU appear in the second column.

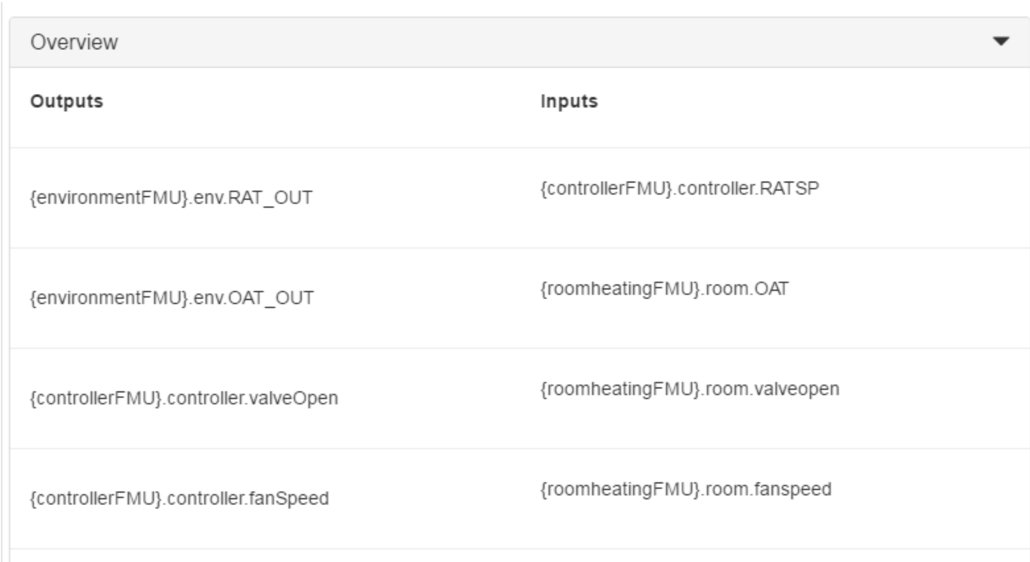


Figure 23: Multi-model overview.

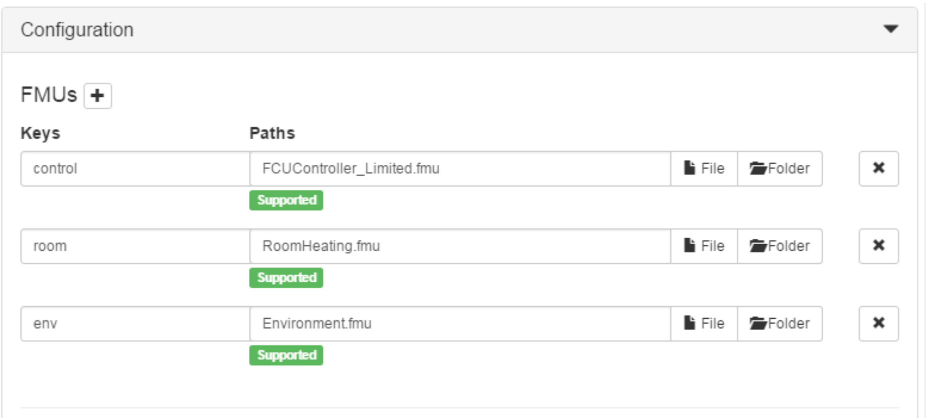


Figure 24: FMUs configuration.

2. Click the desired output variable in the second column. The input instances appear in the third column.

3. Click the desired FMU input instance in the third column. The input variables for the selected FMU appear in the fourth column.

4. Check the box for the desired input variable in the fourth column.

This facility makes it unnecessary to return to Modelio whenever small changes must be made to the connection topology of the multi-model. The *Initial values of parameters* area, shown in Figure 27, allows you to set the

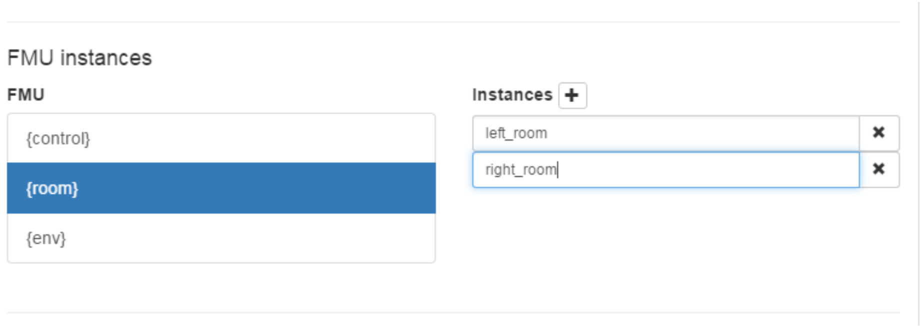


Figure 25: FMU instances configuration.

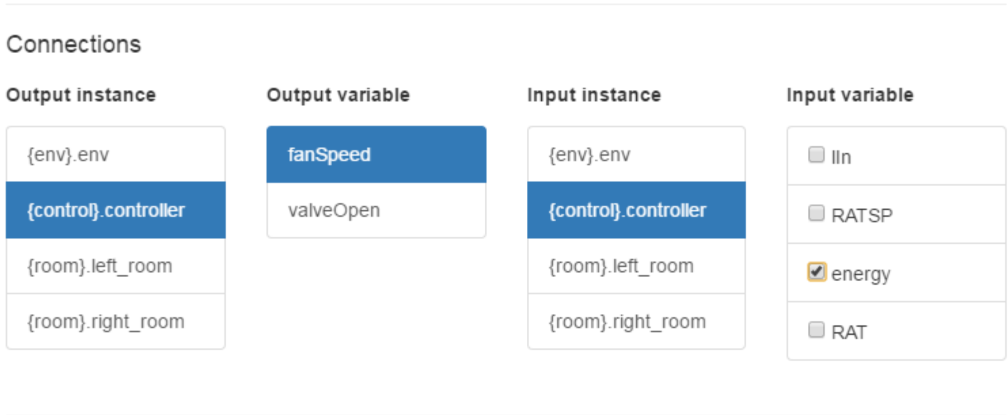
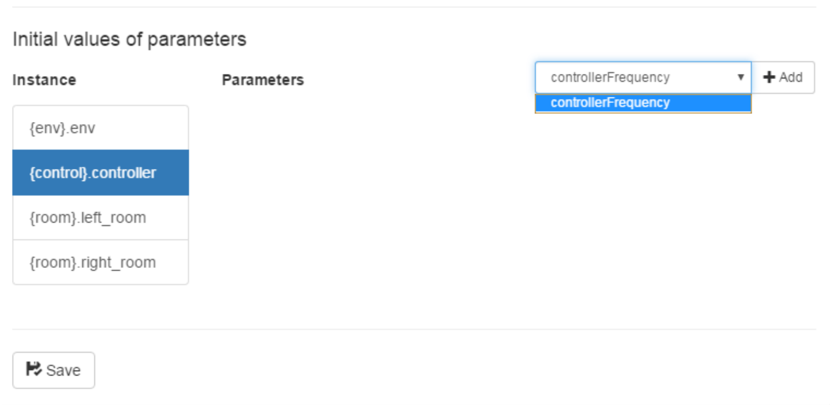


Figure 26: Connections configuration.

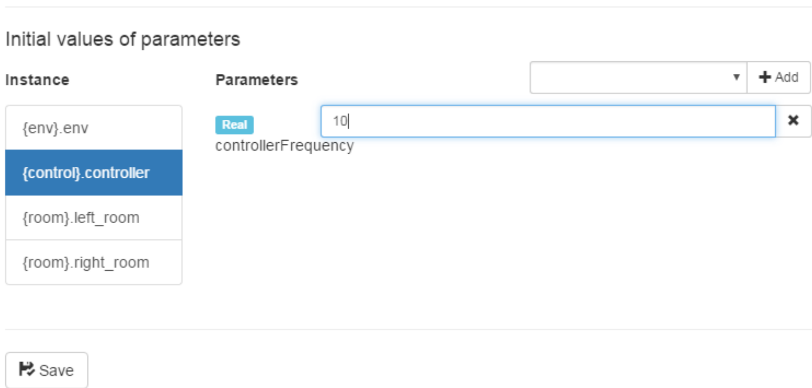
initial values of any parameters defined in the FMUs:

1. Click the desired FMU instance in the *Instance Column*.
2. Select the desired parameter in the *Parameters* dropdown box and click *Add*.
3. Type the parameter value in the box that appears.

Once the multi-model configuration is complete, click the *Save* button at the bottom of the *Configuration* box.



(a) Parameter selection.



(b) Parameter value input.

Figure 27: Initial values of parameters configuration.

4.4 Co-simulations

With the INTO-CPS tool chain it is possible to distribute a co-simulation across several computing nodes such that FMUs need not be co-located with the COE on the same node. This capability caters to situations in which FMUs are restricted to simulation on specific platforms for reasons of legacy technology, licensing *etc.* In the current version of the tool chain this functionality is not fully integrated with the app, and requires the user to start the simulation procedure manually. This is discussed in Section 4.6 below. The remainder of this section discusses standard co-simulations on a single computing node.

To execute co-simulations of a multi-model, a co-simulation configuration is needed. To create a co-simulation configuration, right click the desired multi-model and select *Create Co-Simulation Configuration*, as shown in Figure 28. After creation, the new configuration automatically opens for editing. To select an existing co-simulation configuration, double-click it. Once a

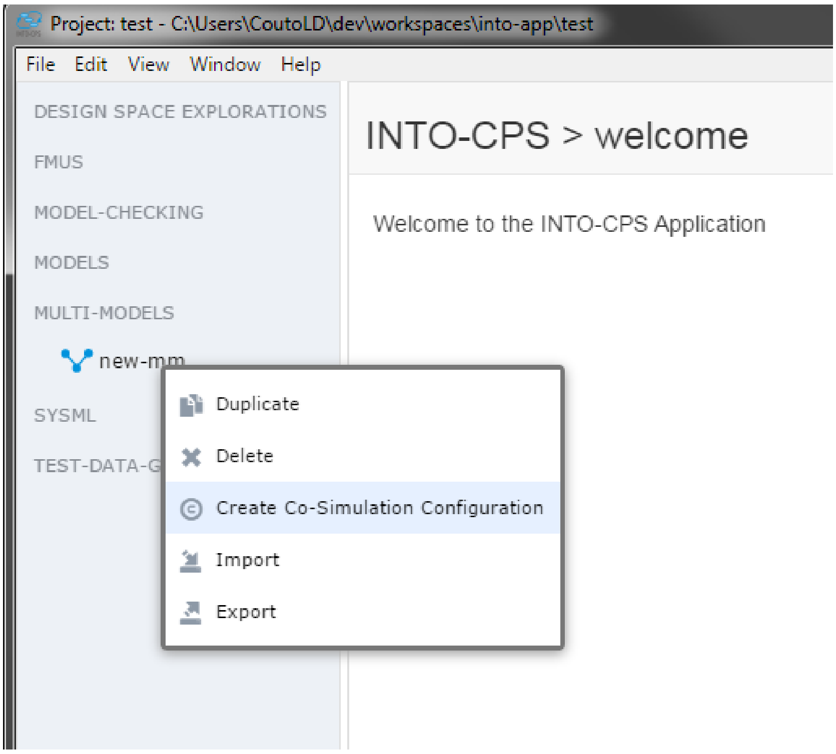


Figure 28: Creating a co-simulation configuration.

configuration is open, the co-simulation configuration, shown in Figure 29, is displayed. The top box, *Configuration*, lets you configure the co-simulation. The bottom box, *Simulation*, lets you execute the co-simulation. In order to

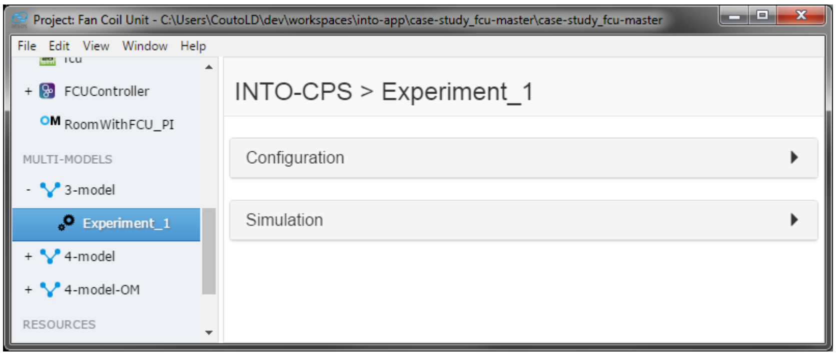


Figure 29: Main co-simulation configuration view.

configure a co-simulation, the configuration must first be unlocked for editing by clicking the *Edit* button at the bottom of the *Configuration* box. There are three things to configure for a co-simulation, discussed next.

The top area, shown in Figure 30, allows you to select the start and end time for the co-simulation as well as the master algorithm to be used. For

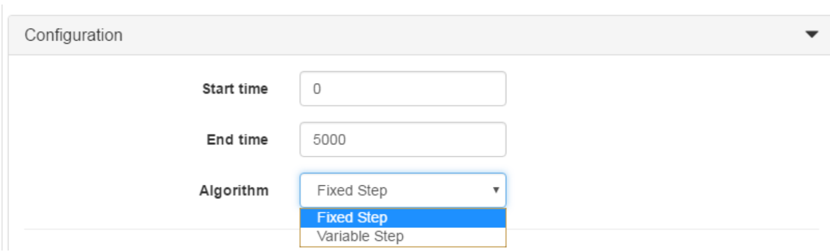
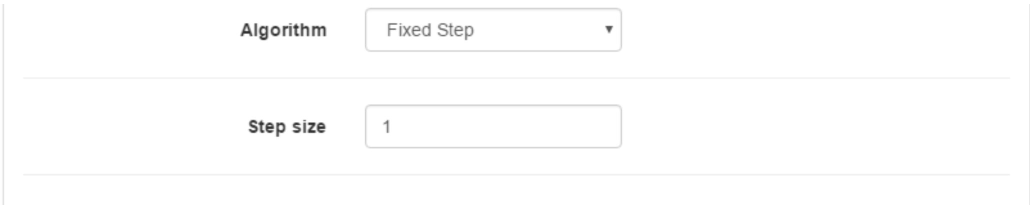


Figure 30: Start/End time and master algorithm configuration.

every algorithm, there are configuration parameters that can be set. These are displayed below the top area, as shown in Figure 31. These parameters differ with the master algorithm chosen. The *Livestream Configuration* area, shown in Figure 32, allows you to select which variables to live stream and plot during the co-simulation. Every instance in the multi-model is displayed and the output variables are shown for each instance. Check the box for each variable that you wish to live stream. Once the co-simulation configuration is complete, click the *Save* button at the bottom of the *Configuration* box.

The *Simulation* box, shown in Figure 33, allows you to launch a co-simulation. To run a co-simulation, the COE must be online. The area at the top of the



(a) Fixed step size.

(b) Variable step size.

Figure 31: Master algorithm configuration.

Simulation box displays the status of the COE. If the COE is offline, you may click the *Launch* button to start it. Once a co-simulation is in progress, any variables chosen for live streaming are plotted in real time in the simulation box, as shown in Figure 34. A progress bar is also displayed. When the simulation is complete, the live stream plot can be explored or exported as a PNG image. In addition, an `outputs.csv` file is created containing the values of every FMU output variable at every point in time in the simulation. This file can be double-clicked and it will open with the default system program for CSV files. It can also be imported into programs such as R, MATLAB or Excel for more complex analysis. Furthermore, it is possible to add a Post-processing script that receives the csv file name and the total simulation time as arguments. It is also possible to configure the amount of logging performed by the Co-Simulation Orchestration Engine.

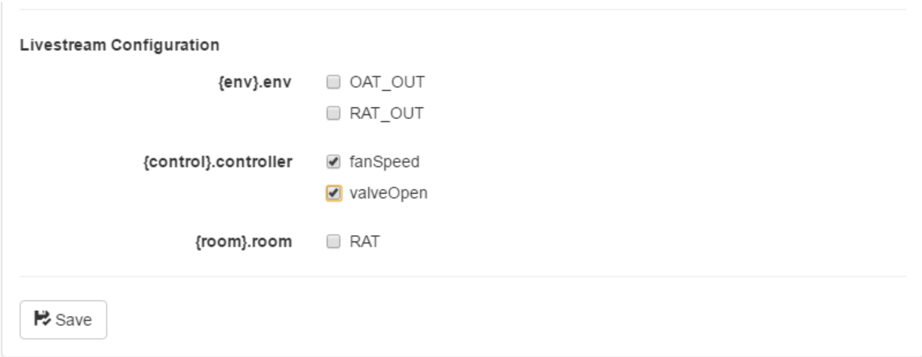
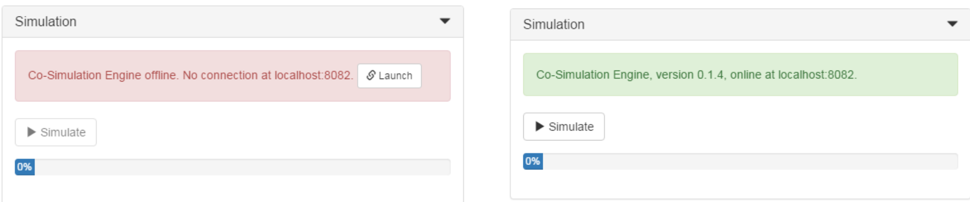


Figure 32: Livestream configuration.



(a) COE offline. (b) COE online.

Figure 33: Launching a co-simulation.

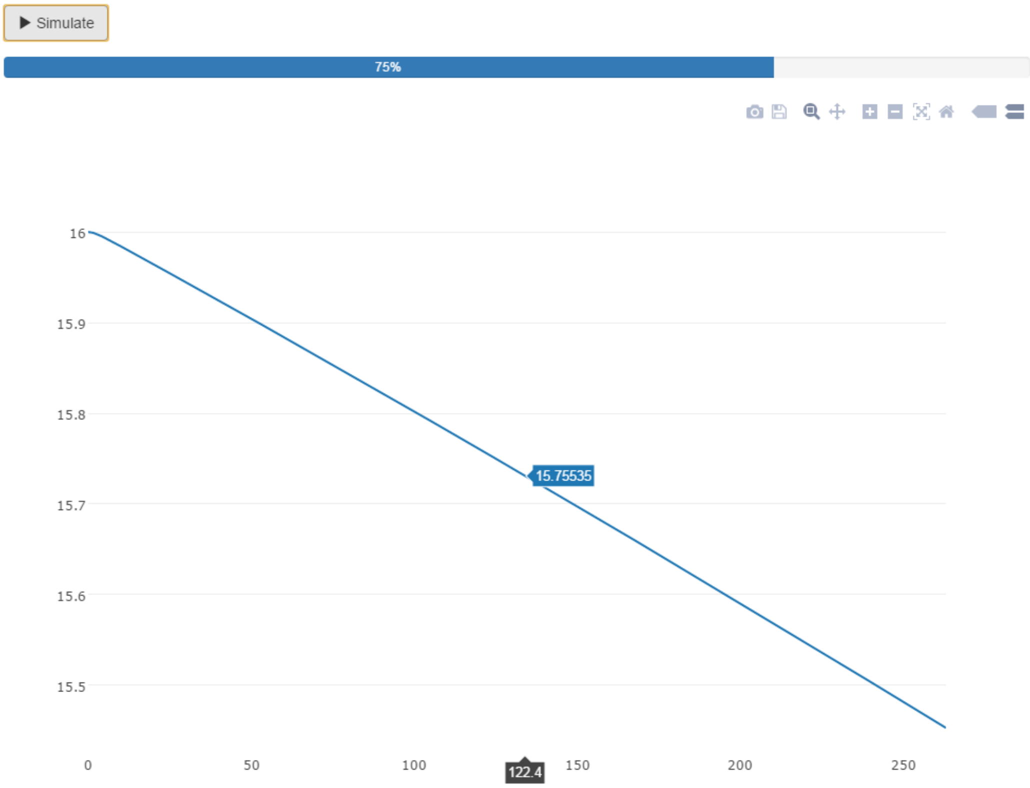


Figure 34: Live stream variable plot.

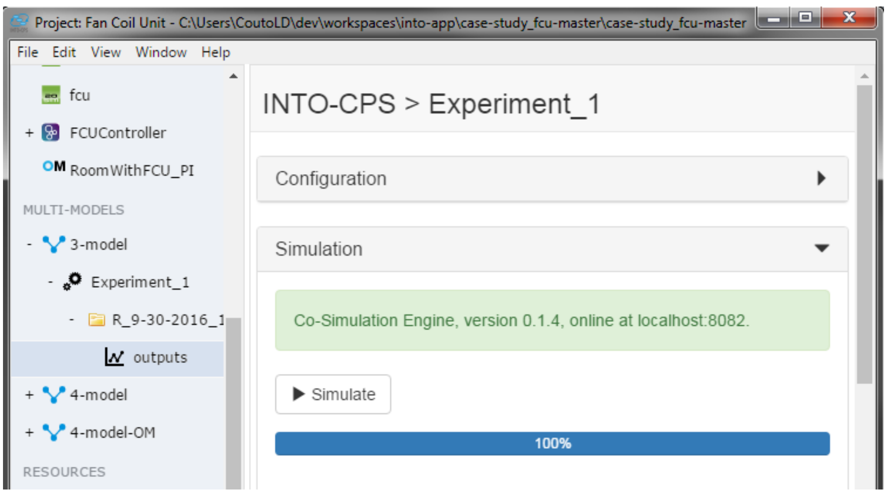


Figure 35: Co-simulation results file.

4.5 Additional Features

The app has several secondary features, most of them accessible through the *Window* menu, as shown in Figure 36. They are briefly explained below.

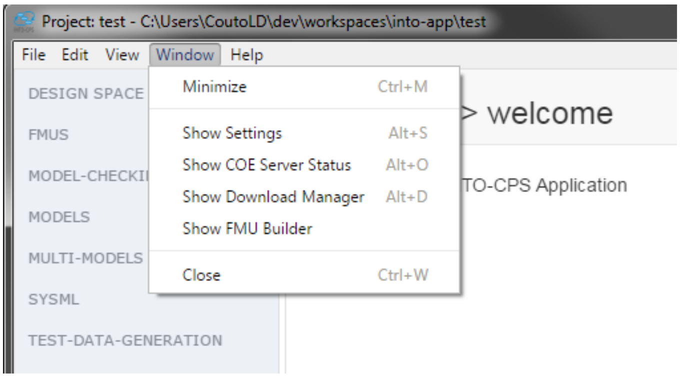


Figure 36: Additional features.

`Show Settings` displays a settings page where various default paths can be set. Development mode can also be enabled from this page, but this feature is primarily meant to be used by app developers for testing.

`Show COE Server Status` displays a page where you can launch and stop the COE as well as observe its log.

`Show Download Manager` displays a page where installers can be downloaded for the various tools of the INTO-CPS tool chain, including the COE.

`Show FMU Builder` displays a page that links to a service where source code FMUs can be uploaded and cross-compiled for various platforms. Note that this is not a secure service and users are discouraged from uploading proprietary FMUs.

4.6 The Co-Simulation Orchestration Engine

The heart of the INTO-CPS Application is the Co-Simulation Orchestration Engine (COE). This is the engine that orchestrates the various simulation tools (described below), carrying out their respective roles in the overall co-simulation. It runs as a stand-alone server hosting the co-simulation API on port 8080. It can be started from the app, but it may be started manually at the command prompt for testing and specialist purposes by executing:

```
java -jar coe.jar 8082
```

TCP port 8082 will be chosen by default if it is omitted in the command above. The COE is entirely hidden from the end user of the INTO-CPS app, but parts of it are transparently configured through the main interface. The design of the COE is documented in deliverable D4.1d [LLW+15].

The COE is controlled using simple HTTP requests. These are documented in the API manual, which can be obtained from the COE's own web page by navigating to `http://localhost:8082`. Port 8082 should be changed to that specified when the COE is started.

Following the protocol detailed in the API document, a co-simulation session can be controlled manually from the command prompt using, for example, the `curl` utility, as demonstrated in the following example.

With the COE running, a session must first be created:

```
curl http://localhost:8082/createSession
```

This command will return a `sessionID` that is used in the following commands.

Next, assuming a COE configuration file called `coeconf.json` has been created as described in the API manual, the session must be initialized:

```
curl -H "Content-Type:  application/json"
--data @coeconf.json
http://localhost:8082/initialize/sessionID
```

Assuming start and end time information has been saved to a file, say `startend.json`, the co-simulation can now be started:

```
curl -H "Content-Type:  application/json"
--data @startend.json
http://localhost:8082/simulate/sessionID
```

Once the co-simulation run ends, the results can be obtained as follows:

```
curl -o results.zip
http://localhost:8082/result/sessionID/zip
```

The session can now be terminated:

```
curl http://localhost:8082/destroy/sessionID
```

The app fundamentally controls the COE in this way.

Distributed co-simulations Presently the app can only control the COE in this way for non-distributed co-simulations. In order to run a distributed co-simulation, a distributed version of the COE, `dcoe`, must be controlled from the command prompt manually, as illustrated above. The distributed COE can be downloaded using the App's *Download Manager*.

In a distributed co-simulation the COE and (some) FMUs execute on physically different compute nodes. The FMUs local to the COE computing node are handled in the same way as in standard co-simulations.

Each FMU on the remote nodes is served externally by a daemon process. This process must be started on the remote node manually as follows:

```
java -jar daemon*-jar-with-dependencies.jar -host
<public-ip> -ip4
```

Here, `<public-ip>` is the IPv4 address of the compute node.

Next, the distributed COE process must be started manually from the command prompt on its own node, with options specific to distributed co-simulation:

```
java -Dcoe.fmu.custom.factory=
org.intocps.orchestration.coe.distribution.
DistributedFmuFactory
-jar dcoe*-jar-with-dependencies.jar
```

The second difference is the way in which the location of the remote FMUs is specified. For a standard co-simulation, the `"fmus"` clause of the co-simulation configuration file (`coeconf.json`, in our example) contains elements of the form

```
"file://fmu-1-path.fmu"
```

These must be modified for each remote FMU to the following URI scheme:

```
"uri://<public-ip>/FMU/#file://local-fmu-path.fmu"
```

The COE configuration file can, of course, be written manually in its entirety, but it is possible to take a faster route, as follows.

This configuration file is only generated when a co-simulation is executed. It is therefore possible to assemble a "dummy" co-simulation that is similar to the desired distributed version, but with a local FMU topology. Since it is likely that the remote FMUs are not supported on the COE platform itself, it is necessary here to construct "dummy" FMUs with the same interface. If this local co-simulation is then executed briefly, a COE configuration file will be emitted that can be easily modified as described above. The app will name this file `config.json` and emit it to the `Multi-models` folder under each co-simulation run. This modified configuration can then be used to execute the distributed co-simulation.

5 Using the Separate Modelling and Simulation Tools

This section provides a tutorial introduction to the FMI-specific functionality of each of the modelling and simulation tools. This functionality is centered on the role of FMUs for each tool. For more general descriptions of each tool, please refer to Appendix B.

5.1 Overture

Overture implements export of both tool-wrapper as well as standalone FMUs. It also has the ability to import a `modelDescription.xml` file in order to facilitate creating an FMI-compliant model from scratch. A typical workflow in creating a new FMI-compliant VDM-RT model starts with the import of a `modelDescription.xml` file created using Modelio. This results in a minimal project that can be exported as an FMU. The desired model is then developed in this context. This section discusses the complete workflow.

5.1.1 Installing the FMI import/export plugin for Overture

In order to use the FMI integration in Overture it is necessary to install a plugin. Below is a guide to install the plugin:

1. Open Overture.
2. Select *Help -> Install New Software*.
3. Click *Add...*
4. In the *Name:* field write *Overture FMU*.

5. In the *Location:* field there are two options:

   INTO-CPS Application: Download the *Overture FMU Import / Exporter - Overture FMI Support* using the Download Manager mentioned in Section 4.5. Locate the file using the *Archive...* button next to the *Location:* field.

   Update site: Enter the following URL in the *Location:* field: *http://overture.au.dk/into-cps/vdm-tool-wrapper/master/latest*.

6. Check the box next to *Overture FMI Integration* as shown in Figure 37.

7. Click *Next* or *Finish* to accept and install.

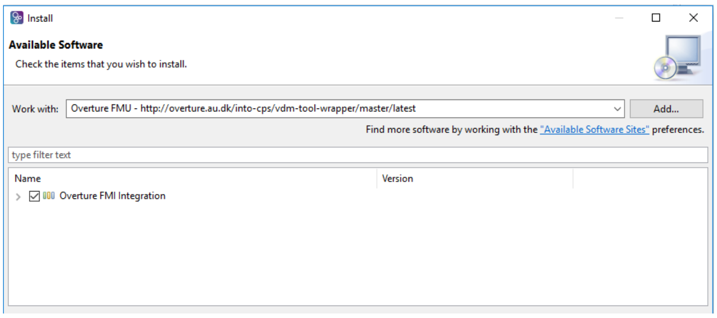


Figure 37: Installing Overture FMI Integration

5.1.2 Import of `modelDescription.xml` File

A `modelDescription.xml` file is easily imported into an existing, typically blank, VDM-RT project from the project explorer context menu as shown in Figure 38. This results in the project being populated with the classes necessary for FMU export:

- A VDM-RT `system` class named "System" containing the system definition. The corresponding "System" class for the water tank controller FMU is shown in Listing 39.
- A standard VDM-RT class named "World". This class is conventional and only provides an entry point into the model. The corresponding "World" class for the water tank controller FMU is shown in Listing 40.
- A standard VDM-RT class named "HardwareInterface". This class contains the definition of the input and output ports of the FMU. Its struc-

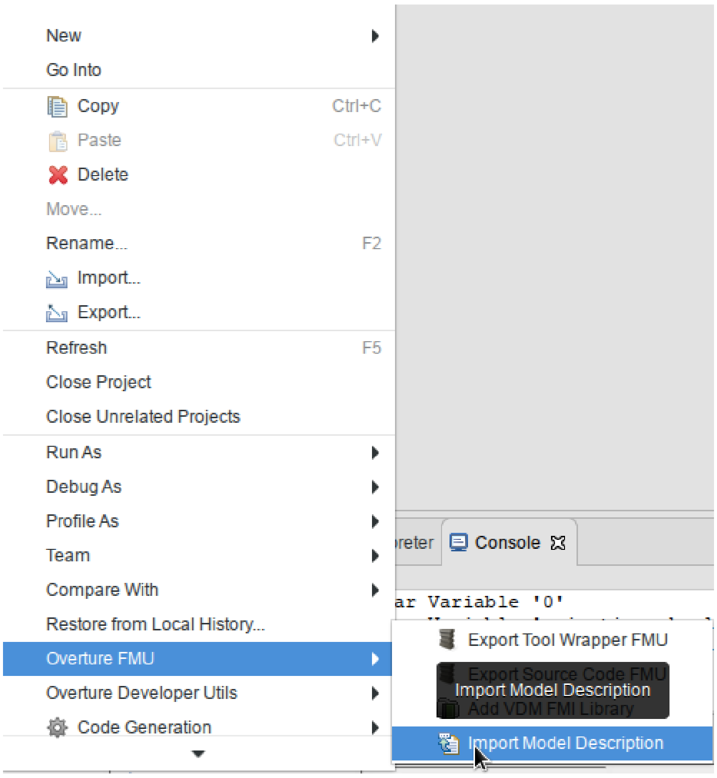


Figure 38: Importing a `modelDescription.xml` file.

ture is enforced, and a self-documenting annotation scheme[5] is used such that the "HardwareInterface" class may be hand-written. The corresponding "HardwareInterface" class for the water tank controller FMU is shown in Listing 41.

- The library file `Fmi.vdmrt` which defines the hardware interface port types used in "HardwareInterface".

[5]The annotation scheme is documented on the INTO-CPS website `into-cps.github.io` under "*Constituent Model Development → Overture → FMU Import/Export*.

```
system System

instance variables

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


instance variables

  public levelSensor : LevelSensor;
  public valveActuator : ValveActuator;
  public static controller : [Controller] := nil;

  cpu1 : CPU := new CPU(<FP>, 20);
operations

public System : () ==> System
System () ==
(
  levelSensor := new LevelSensor(hwi.level);
  valveActuator := new ValveActuator(hwi.valveState);

  controller := new Controller(levelSensor, valveActuator);

  cpu1.deploy(controller,"Controller");
);

end System
```

Figure 39: “System” class for water tank controller.

```
class World

operations

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

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

sync

  per block => false;

end World
```

Figure 40: “World” class for water tank controller.

```
class HardwareInterface

values
  -- @ interface: type = parameter, name="minlevel";
  public minlevel : RealPort = new RealPort(1.0);
  -- @ interface: type = parameter, name="maxlevel";
  public maxlevel : RealPort = new RealPort(2.0);

instance variables
  -- @ interface: type = input, name="level";
  public level : RealPort := new RealPort(0.0);

instance variables
  -- @ interface: type = output, name="valve";
  public valveState : BoolPort := new BoolPort(false);

end HardwareInterface
```

Figure 41: “HardwareInterface” class for water tank controller.

The port structure used in the "HardwareInterface" class is a simple inheritance structure, with a top-level generic "Port", subclassed by ports for specific values: booleans, reals, integers and strings. The hierarchy is shown in Listing 42. When a model is developed without the benefit of an existing `modelDescription.xml` file, this library file can be added to the project from the project context menu, also under the category "Overture FMU".

With all the necessary FMU scaffolding in place, the VDM-RT model can be developed as usual.

5.1.3 Tool-Wrapper FMU Export

Models exported as tool-wrapper FMUs require the Overture tool to simulate. Export is implemented such that the VDM interpreter and its FMI interface are included in the exported FMU. Overture tool-wrapper FMUs currently support Win32, Win64, Linux64, Darwin64 and require Java 1.7 to be installed and available in the PATH environment variable.

A tool-wrapper FMU is easily exported from the project context menu as shown in Figure 43. The FMU will be placed in the `generated` folder.

```
class Port

types
 public String = seq of char;
 public FmiPortType = bool | real | int | String;

operations

 public setValue : FmiPortType ==> ()
 setValue(v) == is subclass responsibility;

 public getValue : () ==> FmiPortType
 getValue() == is subclass responsibility;

end Port

class IntPort is subclass of Port

instance variables
 value: int:=0;

operations
 public IntPort: int ==> IntPort
 IntPort(v)==setValue(v);

 public setValue : int ==> ()
 setValue(v) ==value :=v;

 public getValue : () ==> int
 getValue() == return value;

end IntPort

class BoolPort is subclass of Port

instance variables
 ...
```

Figure 42: Excerpt of "`Fmi.vdmrt`" library file defining FMI interface port hierarchy.

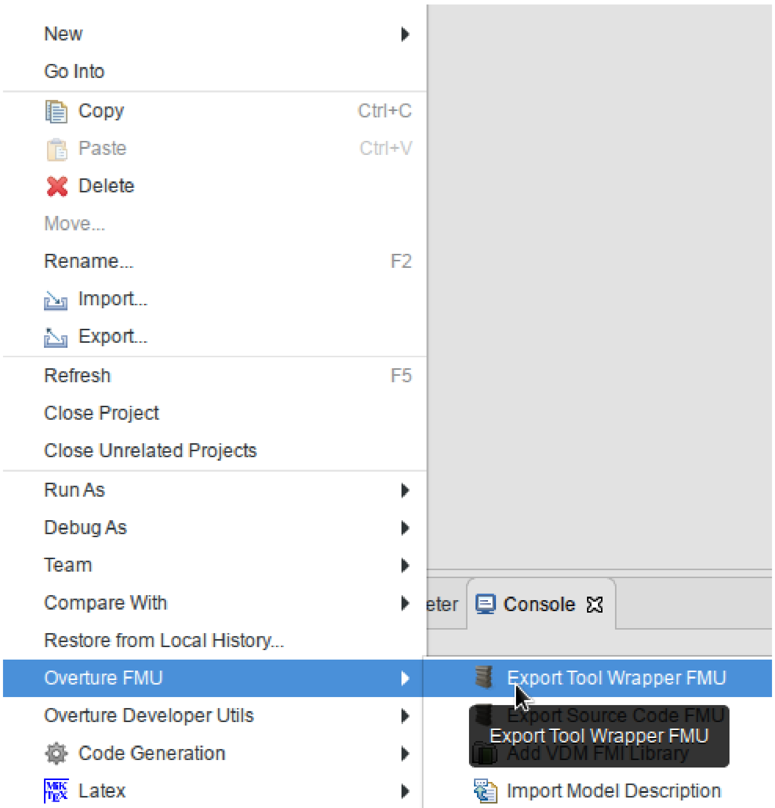


Figure 43: Exporting a tool-wrapper FMU.

5.1.4 Standalone FMU Export

In contrast to tool-wrapper FMUs, models exported as standalone FMUs do not require Overture in order to simulate. Instead, they are first passed through Overture's C code generator such that a standalone implementation of the model is first obtained. Once compiled, this executable model then replaces the combination of VDM interpreter and model, and the FMU executes natively on the co-simulation platform. Currently Mac OS, Windows and Linux are supported, with embedded platform support for SiL and HiL simulation under development.

The export process consists of two steps. First, a source code FMU is obtained from Overture as shown in Figure 44. Second, the INTO-CPS Application must be used to upload the resulting FMU to the FMU compilation server using the built-in facility described in Section 4.5. This is accessed by navigating to *Window* → *Show FMU Builder*.

Please note that only some features of VDM-RT are currently supported by the C code generator. This is discussed in more detail in Section 9.

5.2 20-sim

This section explains the FMI and INTO-CPS related features of 20-sim[6]. We focus on the import of `modelDescription.xml` files, standalone and tool-wrapper FMU export (FMU slave), 3D visualization of FMU operation and an experimental FMU import (FMU master) feature. The complete 20-sim tool documentation can be found in the 20-sim Reference Manual [KGD16].

5.2.1 Import of `modelDescription.xml` File

In Modelio it is possible to export the desired interface for a new FMU from a multi-model as a `modelDescription.xml` file (see Section 3.2. 20-sim can automatically generate an empty 20-sim submodel [7] from this `modelDescription.xml` file with this desired FMU interface. To use

[6] Note that 20-sim is Windows-only. However, it can run fine using Wine [Win16] on other platforms. For details on using 20-sim under Wine, contact Controllab.

[7] Please note that the term "submodel" here should not be confused with the INTO-CPS notion of a "constituent model". A submodel here is a part in a graphical 20-sim model.

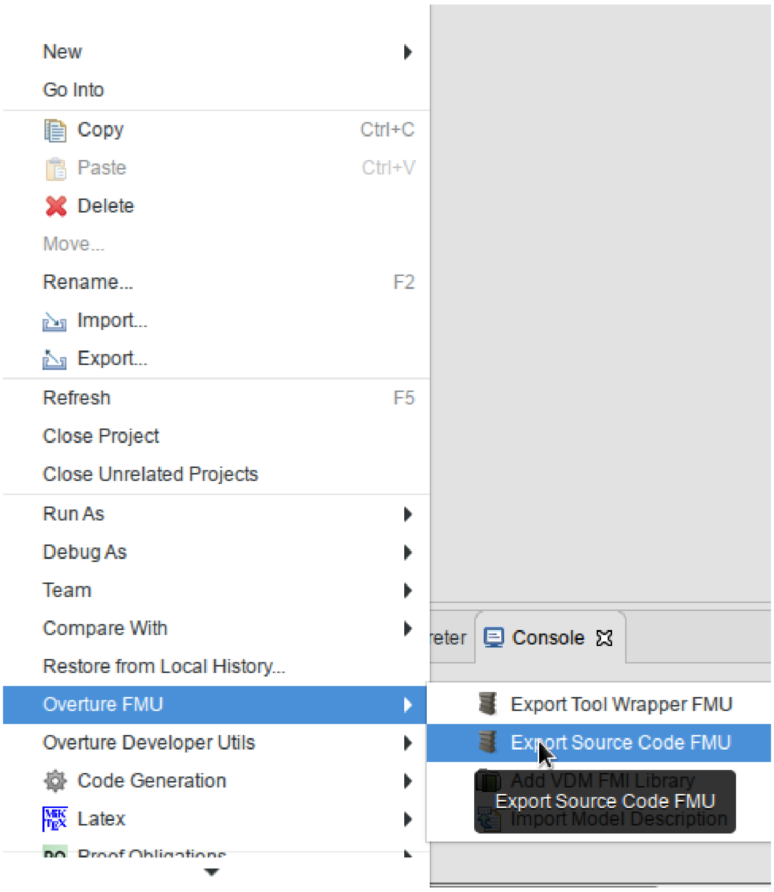


Figure 44: Exporting a standalone FMU.

the `modelDescription.xml` import, you will need to use the "4.6.2-intocps" version of 20-sim[8], since this feature is still under development. A `modelDescription.xml` file can be imported into 20-sim by using Windows Explorer to drag the `modelDescription.xml` file onto your 20-sim model (see Figure 45). This creates a new empty submodel with a blue icon that has the same inputs and outputs as defined in the `modelDescription.xml` file.

5.2.2 Tool-wrapper FMU Export

A tool-wrapper FMU is a communication FMU that opens the original model in the modelling tool and takes care of remotely executing the co-simulation

[8]You can download the INTO-CPS version of 20-sim using the Download Manager in the INTO-CPS Application.

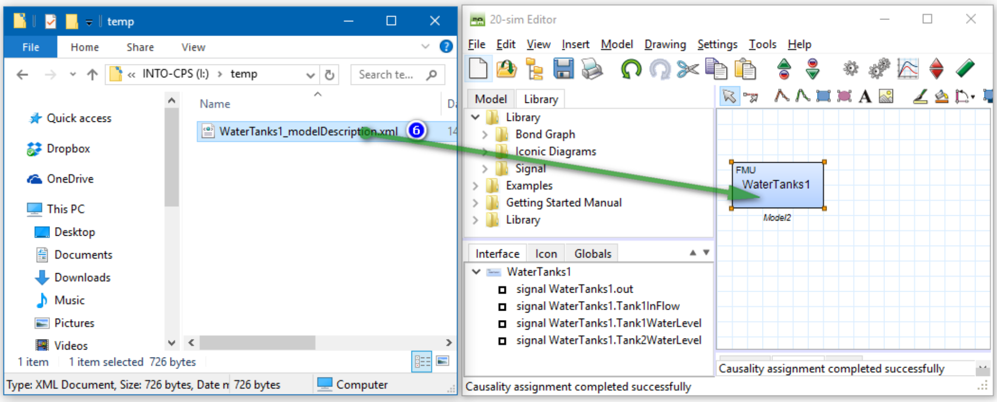


Figure 45: Import a ModelDescription in 20-sim.

steps inside the modelling using some tool-supported communication mechanism. 20-sim supports co-simulation using the XML-RPC-based DESTECS co-simulation interface [LRVG11]. The generation of a tool-wrapper FMU involves two steps that will be explained below:

1. Extend the model with co-simulation inputs, outputs and shared design parameters.
2. Generate a model-specific tool-wrapper FMU.

The tool-wrapper approach involves communication between the co-simulation engine (COE) and the 20-sim model through the tool-wrapper FMU. The 20-sim model should be extended with certain variables that can be set or read by the COE. These variables are the co-simulation inputs and outputs. They can be defined in the model in an equation section called `externals`:

```
externals
        real global export mycosimOutput;
        real global import mycosimInput;
```

To make it possible to set or read a parameter by the co-simulation engine, it should be marked as `'shared'`:

```
parameters
        // shared design parameters
        real mycosimParameter ('shared') = 1.0;
```

The next step is to generate a tool-wrapper FMU for the prepared model.

This requires at least the "4.6.3-intocps" version of 20-sim[9]. This version of 20-sim comes with a Python script that generates a tool-wrapper FMU for the loaded model.

To generate the tool-wrapper FMU:

1. Make sure that the tool-wrapper prepared 20-sim model is saved at a writable location. The tool-wrapper FMU will be generated in the same folder as the model.
2. Open the prepared 20-sim model in 20-sim.
3. Run the BATCH script:
   C:\Program Files (x86)\20-sim 4.6\addons\FMI\ToolwrapperFMUExport\generate.bat
   Note that the `(x86)` is only for 64-bit versions of Windows.
4. You can find the generated tool-wrapper fmu as *<modelname>.fmu* in the same folder as your model.

5.2.3 Standalone FMU Export

Starting with 20-sim version 4.6, the tool has a built-in option to generate standalone co-simulation FMUs for both FMI 1.0 and 2.0 (note that version 2.0 must be used here).

To export a 20-sim submodel as a standalone FMU, make sure that the part of the model that you want to export as an FMU is contained in a submodel and simulate your model to confirm that it behaves as desired.

Next, follow these steps (see also Figure 46):

1. In the Simulator window, choose from the menu: *Tools*.
2. Select *Real Time Toolbox*.
3. Click *C-Code Generation*.
4. Select the *FMU 2.0 export for 20-sim submodel* target.
5. Select the submodel to export as an FMU.
6. Click OK to generate the FMU. This will pop-up a blue window.

[9] You can download the INTO-CPS version of 20-sim using the Download Manager in the INTO-CPS Application.

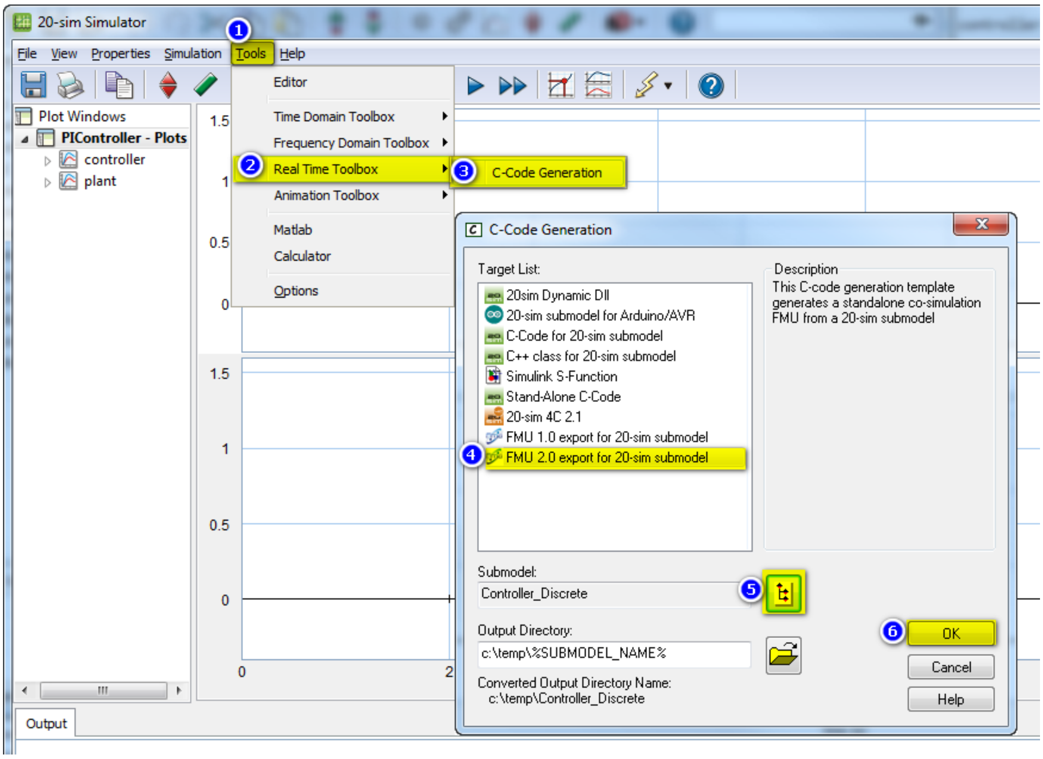


Figure 46: Export an FMU from 20-sim.

Note that to automatically compile the FMU, you will need the Microsoft Visual C++ 2010, 2013 or 2015 compiler installed (normally included with Microsoft Visual Studio, either Express or Community edition). If 20-sim can find one of the supported VC++ compilers, it starts the compilation and reports where you can find the newly generated FMU. The 20-sim FMU export also generates a *Makefile* that allows you to compile the FMU on Windows using Cygwin, MinGW, MinGW64 or on Linux or MacOS X.

20-sim can currently export only a subset of the supported modelling language elements as standalone C-code. Full support for all 20-sim features is only possible through the tool-wrapper FMU approach (described shortly in Section 5.2.2). The original goal for the 20-sim code generator was to export control systems into ANSI-C code to run the control system under a real-time operating system. As a consequence, 20-sim currently only allows code generation for discrete-time submodels or continuous-time submodels using a fixed-step integration method. Support for variable step size integration methods is not yet included by default in the official 20-sim 4.6 release, but it is already included in the 20-sim "4.6.2-intocps" release and on GitHub (see below). Other language features that are not supported, (or are only partly supported) for code generation, are:

- **Hybrid models:** Models that contain both discrete- and continuous-time sections cannot be generated at once. However, it is possible to export the continuous and discrete blocks separate.
- **File I/O:** The 20-sim "Table2D" block is supported; the "datafromfile" block is not yet supported.
- **External code:** Calls to external code are not supported. Examples are: `DLL()`, `DLLDynamic()` and the MATLAB functions.
- **Variable delays:** The `tdelay()` function is not supported due to the requirement for dynamic memory allocation.
- **Event functions:** `timeevent()`, `frequencyevent()` statements are ignored in the generated code.
- **Fixed-step integration methods:** *Euler*, *Runge-Kutta 2* and *Runge-Kutta 4* are supported.
- **Implicit models:** Models that contain unsolved algebraic loops are not supported.
- **Variable-step integration methods:** *Vode-Adams* and *Modified Backward Differential Formula* (MeBDF) are available on GitHub (see below for the link).

The FMU export feature of 20-sim is being improved continuously based on feedback from INTO-CPS members and other customers. To benefit from bug fixes and to try the latest FMU export features like variable step size integration methods (*e.g.* Vode-Adams and MeBDF), you can download the latest version of the 20-sim FMU export template from:

`https://github.com/controllab/fmi-export-20sim`

Detailed instructions for the installation of the GitHub version of the 20-sim FMU export template can be found on this GitHub page. The GitHub FMU export template can be installed alongside the existing built-in FMU export template.

5.2.4 3D Animation FMU

It is possible to visualize a 20-sim simulation as a live 3D animation. This 20-sim 3D animation can be exported as a 3D animation FMU that can be used for visualization purposes in a FMI co-simulation experiment. An example of a 3D animation FMU in action is shown in Figure 47.

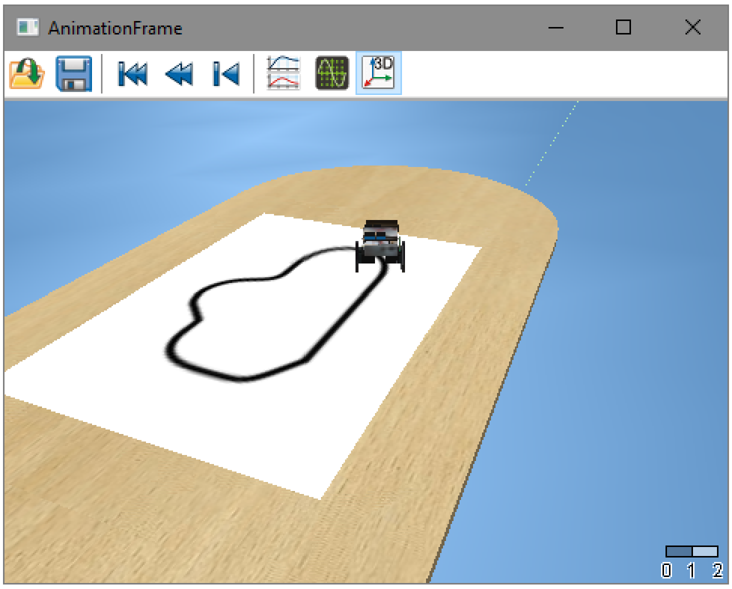


Figure 47: 3D animation FMU

To create a 3D animation FMU, you will need to create a 3D animation in 20-sim that reacts to some signals first (identical to the creation of standard 3D animation in 20-sim):

1. Open your 20-sim model.
2. Open the simulator and add a new 3D animation window using *View* → *New 3D animation window*.
3. Create a new 3D animation scene by following the instructions from the Animation toolbox section in the 20-sim Getting Started manual [KG16].
4. For elements that should move or change color based on external signals, create one equation submodel in 20-sim with all required input signals for the animation.
5. Connect the 3D animation object to the signals from this animation submodel.

The next step is to export the 3D animation as standalone scenery:

1. Go to the 3D animation plot in your 20-sim model.

2. Right-click in the 3D animation plot and select *Plot properties*.
3. Choose *File* → *Save scene*.
4. Select *Yes* to save the whole scenery.
5. Save the scenery under the name `scenery.scn`.

The 3D animation FMU uses the just exported `scenery.scn` file. Since the 3D animation is only a view of the simulation results, the FMU only has a list of inputs. To generate a `modelDescription.xml` file with the right FMU interface, a Python script must be executed which collects the list of external signals referred to by the exported scenery. This Python script and other required resources can be found in the following Controllab GitHub repository:

`https://github.com/controllab/fmi-3D-animation`

To generate the FMU `modelDescription.xml` file, do the following:

1. Copy the generated `scenery.scn` in the `fmu_sources\resources` folder under *3D FMU instructions*.
2. Update `FMU_GUID` in the `scenery_to_fmu.py` Python script with a new GUID for your 3D Animation FMU.
3. Execute the `scenery_to_fmu.py` Python script, *e.g.* using the Python installation that comes with 20-sim 4.6:
   - Start *IPython* found under *20-sim 4.6* in the Windows Start Menu.
   - `cd` *<my 3D FMU instructions path>*
   - `run scenery_to_fmu.py`
     This parses the `scenery.scn` file for objects that point to variables/parameters (references). The variables/parameters are translated to FMU inputs and FMU parameters. The 3D scenery does not contain any information that indicates whether the referred name is a variable or a parameter. As a workaround, all names that start with *parameter.* are marked as as FMU parameters (causality = parameter), while all others are generated as inputs (variability = continuous). This script also generates a `scenery.txt` file with the list of found references. This file is read by the 3D animation DLL to couple the FMU interface to the 3D scenery objects. The output resembles that shown in Figure 48.
4. Create the actual FMU:

- Copy all needed textures to the `fmu_sources\resources` folder.
- Zip the `fmu_sources` folder.
- Rename the Zip file, *e.g.* `3DAnimationFMU.fmu`.

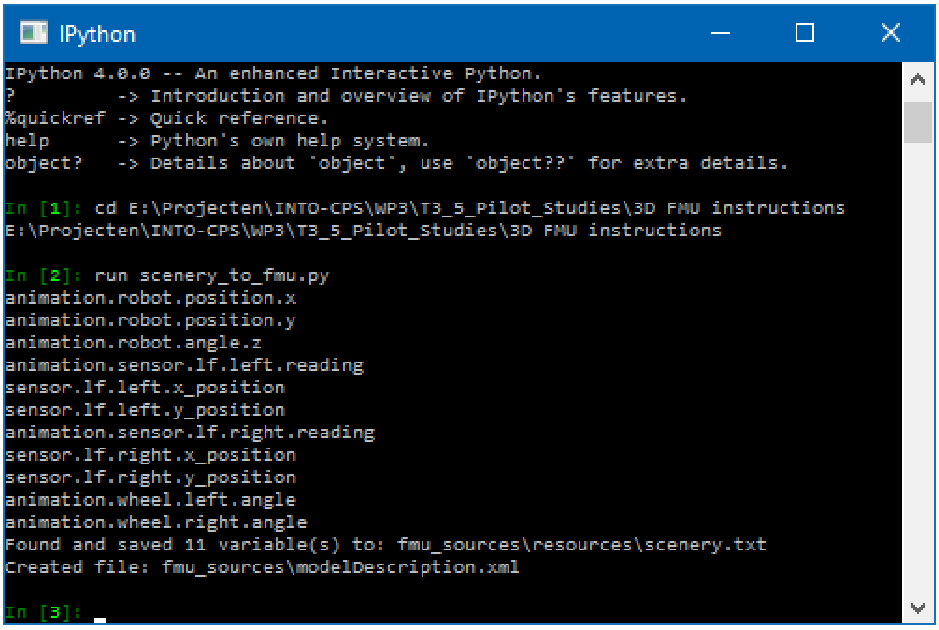


Figure 48: Generating `modelDescription.txt` file from 3D scenery.

5.2.5 FMI 2.0 Import

The "4.6.2-intocps" version of 20-sim has an experimental option to import an FMU directly in 20-sim for co-simulation within 20-sim itself. This is useful for quickly testing exported FMUs without the need to set-up a full co-simulation experiment in the app. Presently only FMI 2.0 co-simulation FMUs can be imported.

The procedure for importing an FMU as 20-sim submodel is similar to importing a `modelDescription.xml` file. Follow these steps to import an FMU in 20-sim:

1. Copy/move the FMU to the same folder as your model. This is not required but recommended to prevent embedding hardcoded paths in your model.
2. Using Windows Explorer, drag the FMU file on your 20-sim model (see Figure 49).

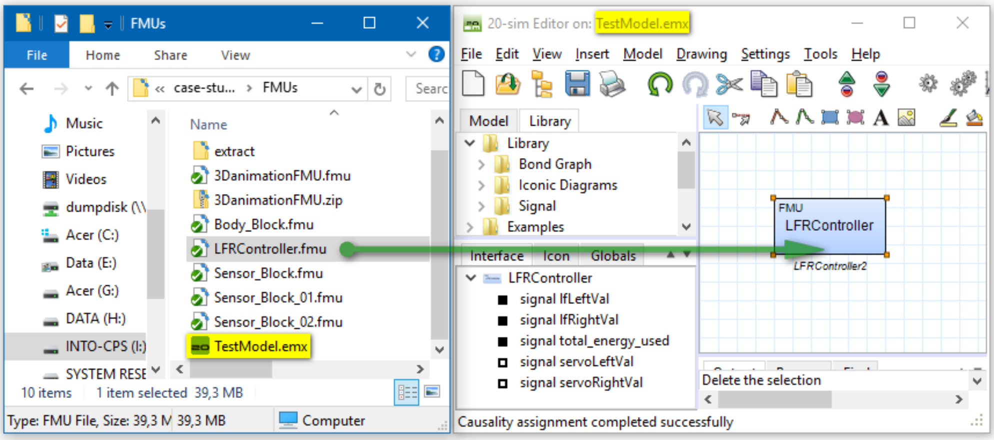


Figure 49: Importing an FMU in 20-sim.

This creates a new submodel with a blue icon that acts as an FMU wrapper. FMU inputs and outputs are translated into 20-sim submodel input and output signals. FMU parameters (scalar variables with causality "parameter") are also available in 20-sim. This means that you can alter the default values of these FMU parameters in 20-sim. The altered FMU parameters are transferred to the FMU during the initialization mode phase of the FMU.

5.3 OpenModelica

This section explains the FMI and INTO-CPS related features of OpenModelica. The focus is on import of `modelDescription.xml` files, and standalone and tool-wrapper FMU export.

5.3.1 Import of `modelDescription.xml` File

OpenModelica can import `modelDescription.xml` interface files created using Modelio and create Modelica models from them. To use the `modelDescription.xml` import feature, you will need to use OpenModelica nightly-builds versions, as this extension is rather new. Nightly builds can be obtained through the main INTO-CPS GitHub site:

    http://into-cps.github.io

To import a `modelDescription.xml` file in OpenModelica one can use:

1. The OpenModelica Connection Editor GUI (OMEdit): *FMI → Import FMI Model Description.*

2. A MOS script, *i.e.* `script.mos`, see below.

```
// start script.mos
// import the FMU modelDescription.xml
importFMUModeldescription("path/to/modelDescription.xml");
    getErrorString();
// end script.mos
```

The MOS script can be executed from command line via:

```
// on Linux and Mac OS
> path/to/omc script.mos
// on Windows
> %OPENMODELICAHOME%\bin\omc script.mos
```

The result is a generated file with a Modelica model containing the inputs and outputs specified in `modelDescription.xml`. For instance:

```
model Modelica_Blocks_Math_Gain_cs_FMU "Output the product
    of a gain value with the input signal"
  Modelica.Blocks.Interfaces.RealInput u "Input signal
      connector" annotation(Placement(transformation(extent
      ={{-120,60},{-100,80}})));
  Modelica.Blocks.Interfaces.RealOutput y "Output signal
      connector" annotation(Placement(transformation(extent
      ={{100,60},{120,80}})));
end Modelica_Blocks_Math_Gain_cs_FMU;"
```

This functionality will ultimately be integrated in the OMEdit (the OpenModelica Connection Editor) graphical user interface.

5.3.2 FMU Export

Currently all FMUs exported from OpenModelica are standalone. There are two ways to export an FMU:

1. From a command prompt.

2. From OMEdit (OpenModelica Connection Editor).

FMU export from a command prompt To export an FMU for co-simulation from a Modelica model a Modelica script file `generateFMU.mos` containing the following calls to the OMC compiler can be used:

```
// load Modelica library
loadModel(Modelica); getErrorString();

// load other libraries if needed
// loadModel(OtherLibrary); getErrorString();

// generate the FMU: PathTo.MyModel.fmu
translateModelFMU(PathTo.MyModel, "2.0", "cs");
    getErrorString();
```

Next, the OMC compiler must be invoked on the `generateFMU.mos` script:

```
// on Linux and Mac OS
> path/to/omc generateFMU.mos
// on Windows
> %OPENMODELICAHOME%\bin\omc generateFMU.mos
```

FMU export from OMEdit One can also use OMEdit (the OpenModelica Connection Editor) to export an FMU as detailed in the figures below.

- Open OMEdit (see Figure 50).
- Load the model in OMEdit (see Figure 51).
- Open the model in OMEdit (see Figure 52).
- Use the menu to export the FMU (see Figure 53).
- The FMU is now generated (see Figure 54).

The generated FMU will be saved to `%TEMP%\OpenModelica\OMEdit`.

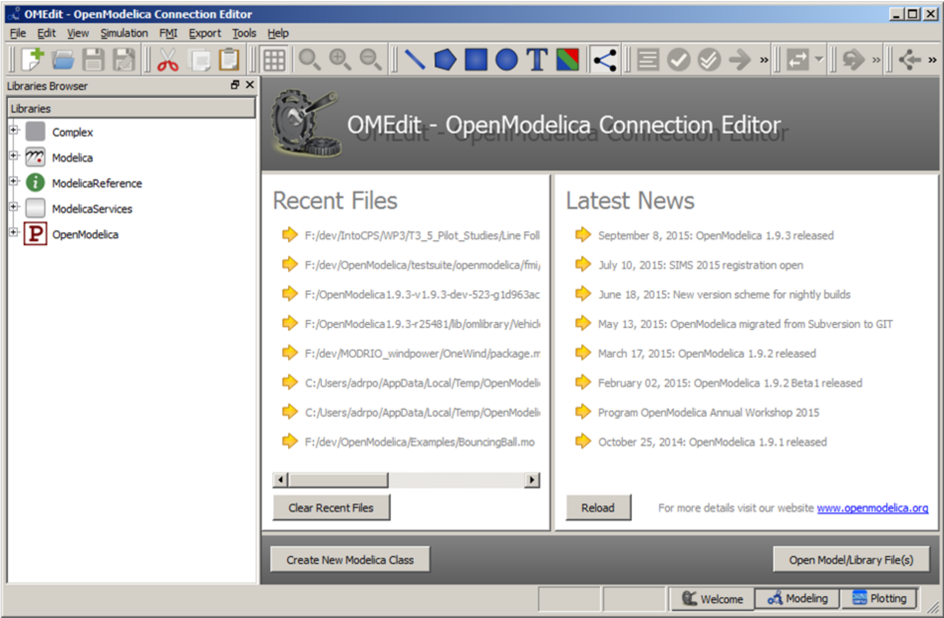


Figure 50: Opening OMEdit.

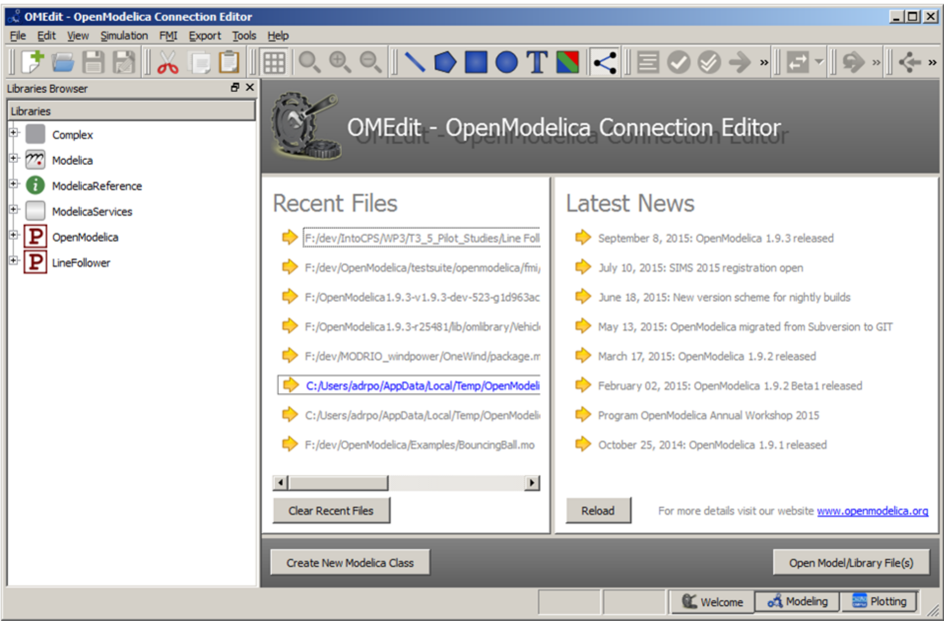


Figure 51: Loading the Modelica model in OMEdit.

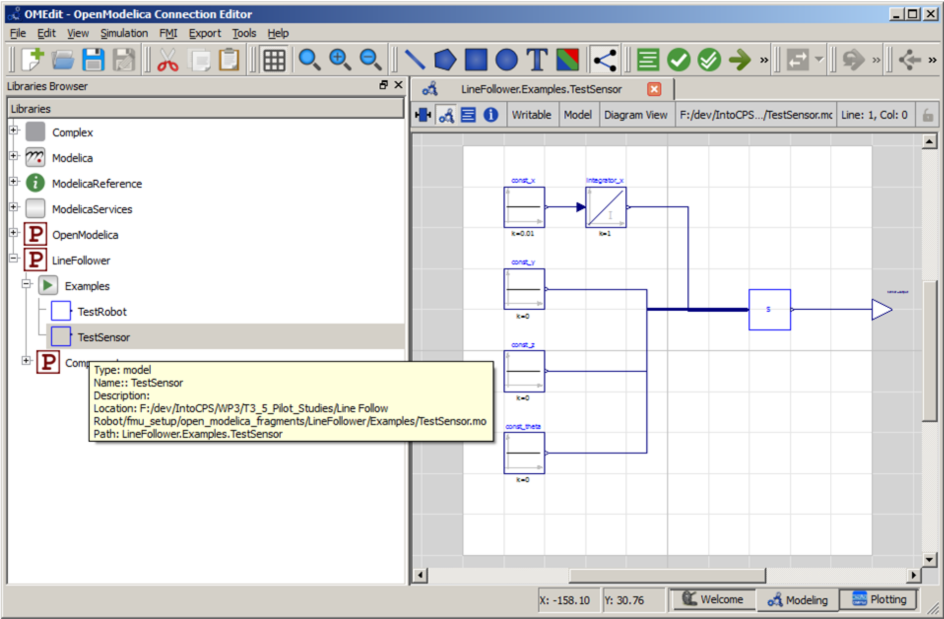


Figure 52: Opening the Modelica model in OMEdit.

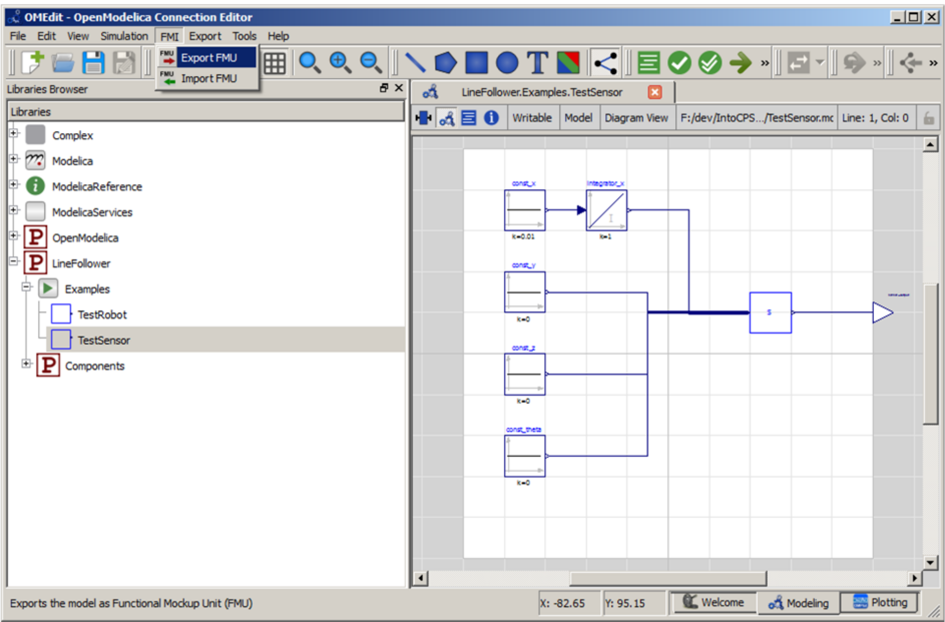


Figure 53: Exporting the FMU.

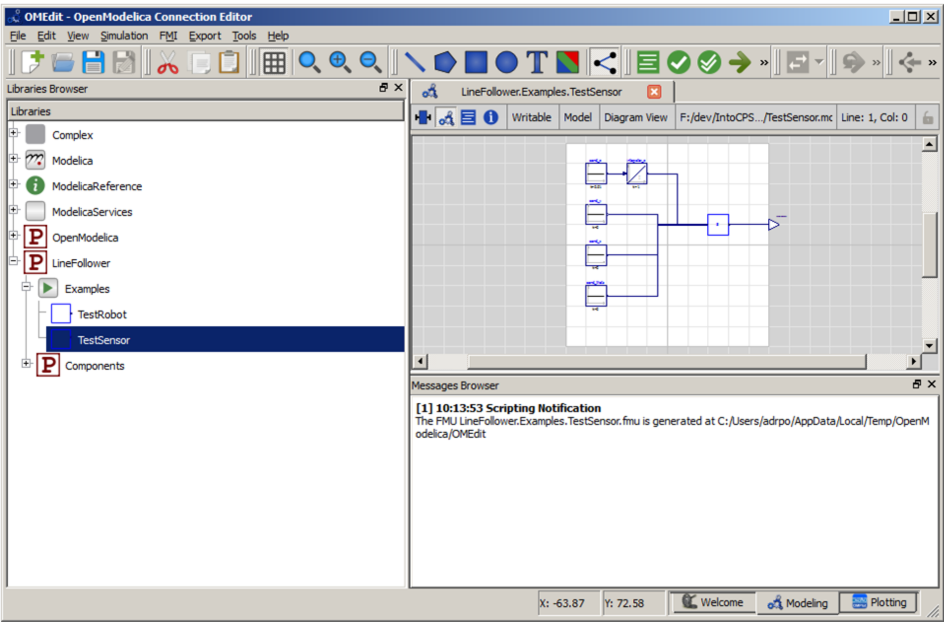


Figure 54: Final step of FMU export.

6 Design Space Exploration for INTO-CPS

This section provides a description of tool support for design space exploration (DSE) developed as part of the INTO-CPS project. Presently the INTO-CPS Application does not provide support for automated creation of the configuration files required to define a DSE experiment. Therefore, this section is split into three parts. Section 6.1 describes how the INTO-CPS Application can be used to launch a DSE using an existing configuration file and Section 6.2 describes how the results from DSE are generated and stored. Section 6.3 describes the structure of the DSE configuration file, giving enough detail for the user to be able to edit one for their purposes.

6.1 How to Launch a DSE

To launch a DSE we need to provide the INTO-CPS Application with the path to two files. The first is the DSE configuration, defining the parameters of the design space, how it should be searched, measured and the results compared. The second is the multi-model configuration, defining the base model that will be used for the search. A DSE configuration is selected by double clicking on one of the configurations listed in the *Design Space Explorations* section of the INTO-CPS Application project explorer; these configurations are identified with the ( ) icon. If the COE is not already running, the DSE page is shown with a red "*Co-simulation engine not running*" status, as shown in Figure 55.

If this is the case, click on the *Launch* button to start the COE. This results in a green co-simulation engine status (see Figure 56). With the DSE configuration selected and the COE running, the next step is to select the multi-model to use. One can be selected from the *Co-simulation Configuration* drop-down box, as shown in Figure 57. Pressing the *Simulate* button starts the DSE background process.

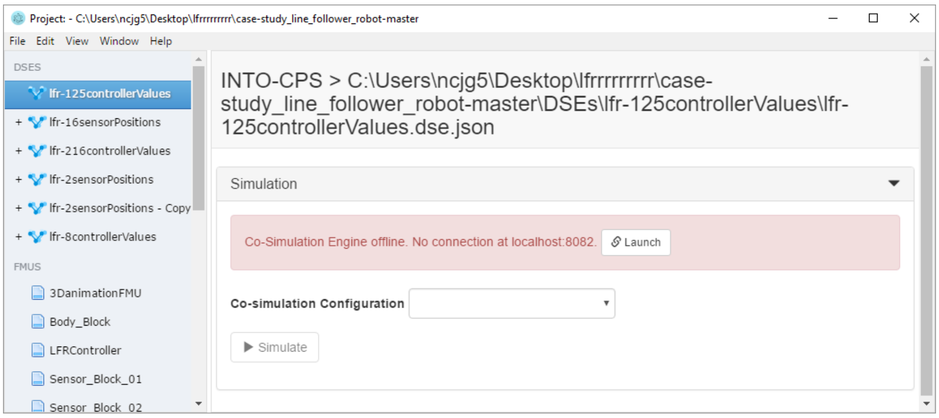


Figure 55: Status when COE is not running.

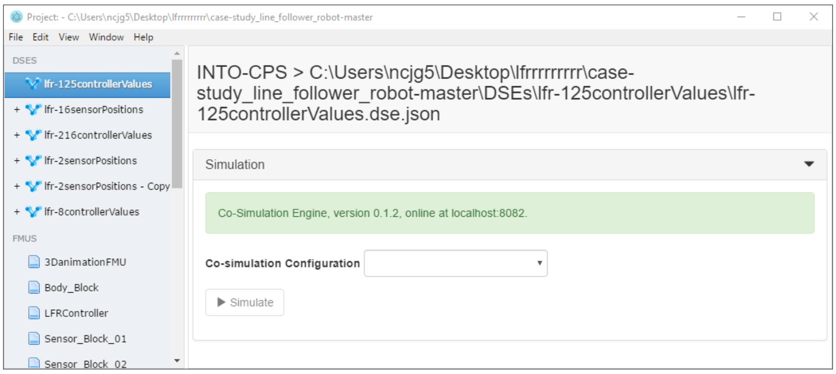


Figure 56: Status when COE is running.

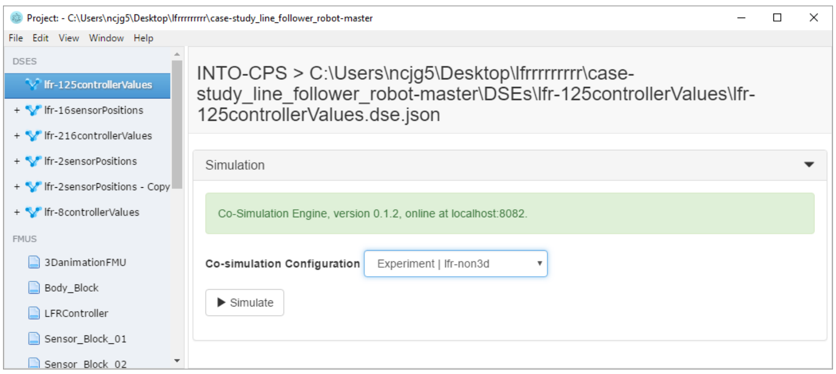


Figure 57: Selecting a multi-model.

6.2 Results of a DSE

The DSE scripts store their results in a folder named for the date and time at which the DSE was started. This folder may be found underneath the name of the DSE script selected, as shown in Figure 58. When the DSE has finished, we can find both a graphs folder and an HTML results page inside the results folder. It may be necessary to refresh the project view to see these new items. The results HTML file is identified by the ( ) icon, and double clicking on it opens the results page in the default browser.

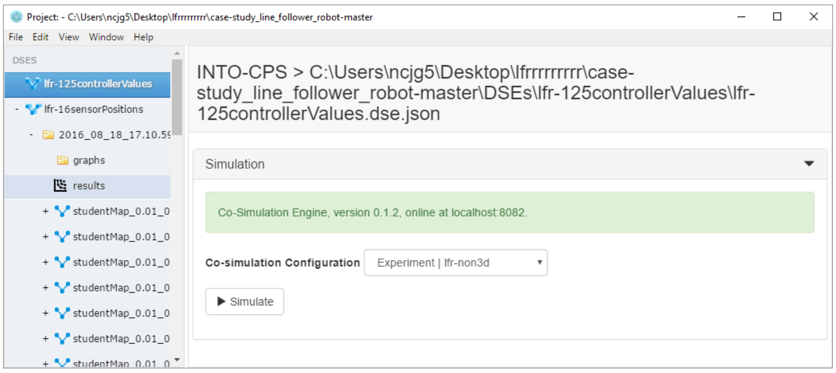
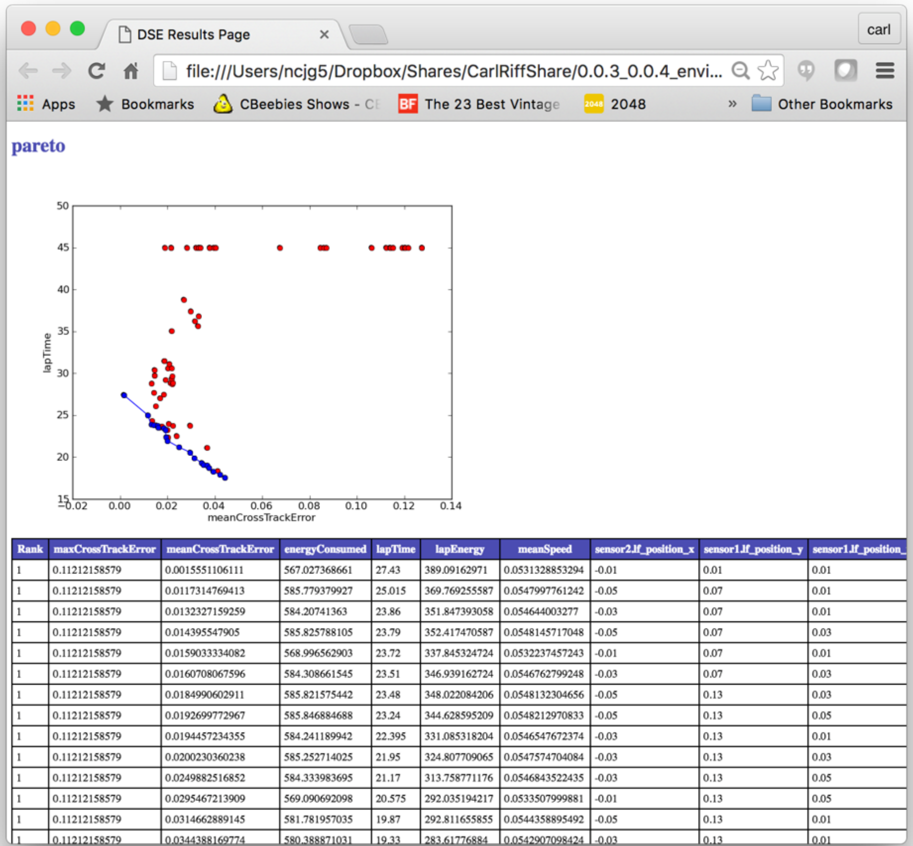


Figure 58: Icon shown when DSE results are ready.

The results, shown in Figure 59, contain two elements. The first element is a Pareto graph showing the results of all simulations on a single plot, with each point on the graph representing a single simulation. The best designs, referred to as the non-dominated set, are shown in blue, with ranks of progressively worse designs coloured alternately red and yellow. The second element is a table of these results, with the rank in the left hand column, followed by the objective values and finally the design parameters that produced the result.

6.3 How to Edit a DSE Configuration

Editing of a DSE configuration is currently a manual process and so guidance regarding each section of the configuration is presented in this section.



Rank	maxCrossTrackError	meanCrossTrackError	energyConsumed	lapTime	lapEnergy	meanSpeed	sensor2.lf_position_x	sensor1.lf_position_y	sensor1.lf_position_
1	0.11212158579	0.0015551106111	567.027368661	27.43	389.09162971	0.0531328853294	-0.01	0.01	0.01
1	0.11212158579	0.0117314769413	585.779379927	25.015	369.769255587	0.0547997761242	-0.05	0.07	0.01
1	0.11212158579	0.0132327159259	584.20741363	23.86	351.847393058	0.054644003277	-0.03	0.07	0.01
1	0.11212158579	0.014395547905	585.825788105	23.79	352.417470587	0.0548145717048	-0.05	0.07	0.03
1	0.11212158579	0.0159033334082	568.996562903	23.72	337.845324724	0.0532237457243	-0.01	0.07	0.01
1	0.11212158579	0.0160708067596	584.308661545	23.51	346.939162724	0.0546762799248	-0.03	0.07	0.03
1	0.11212158579	0.0184990602911	585.821575442	23.48	348.022084206	0.0548132304656	-0.05	0.13	0.03
1	0.11212158579	0.0192699772967	585.846884688	23.24	344.628595209	0.0548212970833	-0.05	0.13	0.05
1	0.11212158579	0.0194457234355	584.241189942	22.395	331.085318204	0.0546547672374	-0.03	0.13	0.01
1	0.11212158579	0.0200230360238	585.252714025	21.95	324.807709065	0.0547574704084	-0.03	0.13	0.03
1	0.11212158579	0.0249882516852	584.333983695	21.17	313.758771176	0.0546843522435	-0.03	0.13	0.05
1	0.11212158579	0.0295467213909	569.090692098	20.575	292.035194217	0.0533507999881	-0.01	0.13	0.05
1	0.11212158579	0.0314662889145	581.781957035	19.87	292.811655855	0.0544358895492	-0.05	0.13	0.01
1	0.11212158579	0.0344388169774	580.388871031	19.33	283.61776884	0.0542907098424	-0.03	0.13	0.01

Figure 59: A page of DSE results.

6.3.1 File Creation

The suggested procedure for creating a new configuration is to make a copy of an existing one and then to edit the required sections. The individual configurations are located in their own folders within the `Design Space Exploration` folder of the INTO-CPS Application project directory, such as the pilot study with the line following robot "LFR-2SensorPositions" configuration shown in Figure 60 (see [PGP+16]). Using your OS's file browser, create a new folder under `DSEs` and then copy in and rename a DSE configuration. The names of the new folder and configuration folder can be chosen at will, but the configuration file must have the extension `.dse.json` .

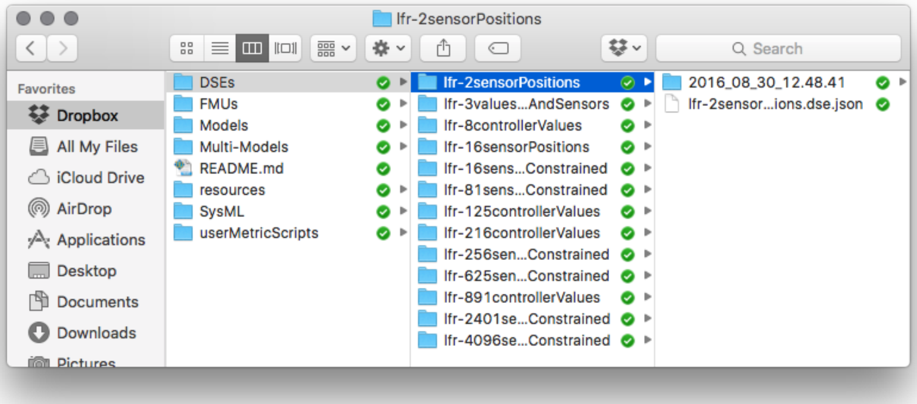


Figure 60: Location of DSE configurations.

6.3.2 Parameters

The parameters section is used to define a list of values for each parameter to be explored. Figure 61 shows the definition of four parameters, each with two values. If a parameter is included in the DSE configuration file, then it must have at least one value defined. The order of the values in the list is not important. If a parameter that is to be explored is not in the list, its ID may be found in the three ways listed below.

1. If the parameter is listed in the multi-model configuration, then copy it from there.

2. If the parameter is not in the multi-model parameters list then its name may be found by examining the model description file in the associated FMU. In this case it will be necessary to prepend the parameter ID with the ID for the FMU and the instance ID of the FMU, for example in "`{sensor1FMU}.sensor1.lf_position_x`".

    - the ID of the FMU is `{sensor1FMU}`.
    - the instance ID of the FMU in the multi-model is `sensor1`.
    - the parameter ID is `lf_position_x`.

3. The IDs for each parameter may also be found on the Architecture Structure Diagram in the SysML models of the system. The full name for use in the multi-model may then be constructed as above.

```
"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
    ]
},
```

Figure 61: Example parameter definitions.

6.3.3 Parameter Constraints

It may be the case that not all combinations of the parameter values defined in the previous section are valid. So, it is necessary to be able to define constraints over the design parameters such that no time is wasted simulating invalid designs. For example, in the line follower robot we define ranges for the x and y co-ordinates of the left and right sensors separately, and running all combinations of these leads to asymmetric designs that do not have the same turning behaviour on left and right turns. To prevent this we can define boolean expressions based upon the design parameters and evaluate these before a simulation is launched. Figure 62 shows two constraints defined for the line follower DSE experiment that ensure only symmetrical designs are allowed. The first constraint ensures the y co-ordinates of both sensors are the same, while the second constraint ensures that the x co-ordinate of the left sensor is the same, but negated as the x co-ordinate of the right sensor. Note that the names used when defining such constraints have the same `FMU_ID.instance_ID.parameter_ID` format as used when defining a parameter range (see Section 6.3.2)

Since the constraints are processed using the Python `eval` function, any boolean expression compatible with it may be used here.

```
"parameterConstraints": [
    "{sensor1FMU}.sensor1.lf_position_y == {sensor2FMU}.sensor2.lf_position_y",
    "{sensor1FMU}.sensor1.lf_position_x == - {sensor2FMU}.sensor2.lf_position_x"
],
```

Figure 62: Example parameter constraints.

6.3.4 Scenario List

The DSE scripts currently have limited support for scenarios referring to a specific set of conditions against which the multi-model is to be tested. In the example of the line following robot, the scenario refers to the map the robot has to follow, along with its starting co-ordinates. For instance, in one scenario the robot would go around a circular track in one direction, predominantly turning left, whereas in a different scenario the same track would be followed in the opposite direction, predominantly turning right. In both scenarios the map of the track is the same.

Changing a scenario may involve changing one or more different parts of the multi-model and its analysis, such as the specific FMUs used, parameters passed to an FMU, the multi-model the DSE is based upon, along with any data files used by the objective scripts (Section 6.3.6) to evaluate performance. This feature is currently under development and so only the objective data file selection is implemented presently.

6.3.5 Objective Definitions: Internal

There are two means for defining the objectives used to assess the performance of a simulated model. The first of these, described here, is using the internal functions included in the DSE scripts. This is a set of simple functions that can be applied to any of the values recorded by the COE during simulation. The current set of internal functions is:

max Returns the maximum value of a variable during a simulation.

min Returns the minimum value of a variable during a simulation.

mean Returns the mean value of a variable during a simulation (*n.b.*, a fixed simulation step size is currently assumed.)

Defining an internal objective requires three pieces of information:

name This is the name that the objective value will be stored under in the objectives file.

type This selects the function to be applied. The key `objectiveType` is used in the DSE configuration file.

variable This defines the variable to which the function is to be applied. The key `columnID` is used to denote this parameter in the DSE configuration file.

```
"energyConsumed": {
    "columnID": "{bodyFMU}.body.total_energy_used",
    "objectiveType": "max"
    }
}
```

Figure 63: Definition of an internal objective.

Figure 63 shows the definition of an objective named `energyConsumed`, which records the maximum value of the variable `{bodyFMU}.body.total_energy_used`. This objective is recorded and may be used later, primarily for the purpose of ranking designs, but it could also be used for any other analysis required.

6.3.6 Objective Definitions: External Scripts

The second form of objective definition makes use of user-defined Python scripts to allow bespoke analysis of simulation results to be launched automatically and results recorded using the common format. The definition has two parts: the construction of the Python script to perform the analysis and the definition of the script's required parameters in the DSE configuration file, these two steps are described below.

Construction of the Script The outline functionality of an analysis script is that, at the appropriate times, a DSE script calls it, passing four or more arguments. The script uses these arguments to locate a raw simulation results file (`results.csv`), processes those results and then writes the objective values into an objectives file (`objectives.json`) for that simulation.

The first three arguments sent to the script are common to all scripts. These are listed below.

argv 1 The absolute path to the folder containing the `results.csv` results file. This is also the path where the script finds the `objectives.json` file.

argv 2 The name of the objective. This is the key against which the script should save its results in the objectives file.

argv 3 The name of the scenario.

With this information the script can find the raw simulation data and also determine where to save its results. The name of the scenario allows the script to locate any data files it needs relating to the scenario. For example, in the case of the script measuring cross track error for the line following robot, the script makes use of a data file that contains a series of coordinates that represent the line to be followed. The name of this data file is `map1px.csv`. It is placed into a folder with the same name as the scenario, which in this case is `studentMap`. That folder is located in the `userMetricScripts` folder, as shown in Figure 64. Using this method, the developer of an external analysis script needs only to define the name of the data file they will need and know that at runtime the script will be passed a path to a folder containing the data file suitable for the scenario under test.

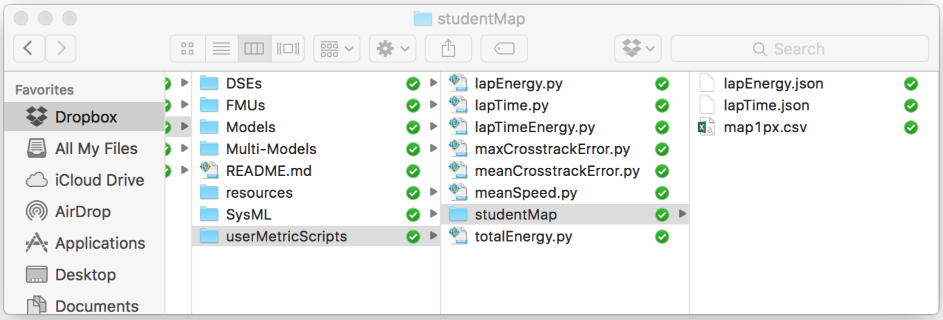


Figure 64: External analysis script data files for the "studentMap" scenario.

Figure 65 shows an example of an external analysis script. In this case it computes the cumulative deviation of the water level from some target level. There are two distinct sections in the file, we shall refer to them as the 'common' and 'script specific' sections.

The common section contains core functions that are common to all external scripts. It reads in the three arguments that are common to all scripts, and contains functions to help the user retrieve the data needed by the analysis script, and to write the computed objective value into the `objectives.json` file. It is recommended that this section be copied to form the basis of any new external analysis scripts.

The second part of the example script shown is specific to the analysis to be performed. The purpose of this section is to actually compute the value of the objective from the results of a simulation. Generally it will have three parts: reading in any analysis specific arguments such as the ID of data in the results that it needs, using the data in `results.csv` to cal-

```
import csv,os, sys, json, io

def getColumnFor(colName, row):
    index = 0
    for thisName in row:
        if thisName.strip() == colName.strip():
            return index
        else:
            index +=1
    return index

def writeObjectiveToOutfile(key, val):
    parsed_json = {}
    if os.path.isfile(objectivesFile):
        json_data = open(objectivesFile)
        parsed_json = json.load(json_data)
    parsed_json[key] = val
    dataString = json.dumps(parsed_json, sort_keys=True,indent=4, separators=(',', ': '))
    with io.open(objectivesFile, 'w', encoding='utf-8') as f:
        f.write(unicode(dataString))

resultsFileName = "results.csv"
resultsFile = sys.argv[1] + os.path.sep + resultsFileName
objectivesFileName = "objectives.json"
objectivesFile = sys.argv[1] + os.path.sep + objectivesFileName
objectiveName = sys.argv[2]
scenarioDataFolder = sys.argv[3]
csvfile = open(resultsFile)
csvdata = csv.reader(csvfile, delimiter=',')
```

Common Section

```
levelColumnID = sys.argv[4]
targetLevel = float(sys.argv[5])

cumulativeDeviation = 0.0
levelColumn = 0
stepSizeColumn = 0
firstRow = True

for row in csvdata:
    if firstRow:
        levelColumn = getColumnFor(levelColumnID, row)
        stepSizeColumn = getColumnFor('step-size', row)
        firstRow = False
    else:
        level = float(row[levelColumn])
        stepSize = float(row[stepSizeColumn])
        cumulativeDeviation += abs ((level - targetLevel)*stepSize)

writeObjectiveToOutfile(objectiveName,cumulativeDeviation)
```

Script Specific Section

Figure 65: External analysis script to calculate cumulative deviation in the water tank example

culate the value of the objective and finally write the objective value into `objectives.json`.

In the 'Script Specific Section' of Figure 65 we see the example of the script calculating the cumulative deviation of the water level from a target level in the water tank model. It starts by reading a further two arguments passed when the script is launched and initializes the variables. The script then iterates through all rows of data in `results.csv` to calculate the cumulative deviation which is then written to the `objectives.json` file in the final line.

```
"externalScripts": {
    "lapTime": {
        "scriptFile": "lapTime.py",
        "scriptParameters": {
            "1": "time",
            "2": "{bodyFMU}.body.robot_x",
            "3": "{bodyFMU}.body.robot_y",
            "4": "studentMap"
            }
        },
    "meanCrossTrackError": {
        "scriptFile": "meanCrosstrackError.py",
        "scriptParameters": {
            "1": "{bodyFMU}.body.robot_x",
            "2": "{bodyFMU}.body.robot_y"
            }
        }

},
```

Figure 66: Definition of the external analysis functions for the line follower robot.

Definition of External Analysis in DSE Configuration With the analysis scripts constructed, the next step is to define their use in the DSE configuration file. The definition essentially contains three parts:, a name for the objective, the file name of the script and a list arguments to pass. The name given to the objective allows it to be referenced in the objectives constraints and ranking sections of the DSE configuration. The file name tells the DSE scripts which script to launch and the arguments define additional data (over the standard three arguments described earlier) that the script needs, such as the names of data it needs or constant values.

In Figure 67 we find the definition of the external analysis used in the three tank water tank example. There are two analysis defined, the first is named 'cumulativeDeviation' and the second is 'vCount'. In each there are two parameters defined, the 'scriptFile' contains the file name of the script file to run in each case, while the 'scriptParameters' parameter contains the list of additional arguments each needs.

```
"objectiveDefinitions": {
    "externalScripts": {
        "cumulativeDeviation": {
            "scriptFile": "cumulativeDeviation.py",
            "scriptParameters": {
                "1": "{tank2}.tank2.level",
                "2": "1.0"
            }
        },
        "vCount": {
            "scriptFile": "valveChanges.py",
            "scriptParameters": {
                "1": "{controller}.controller.wt3_valve"
            }
        }
    },
    "internalFunctions": {}
},
```

Figure 67: Definition of the external analysis functions for the three water tank model.

The purpose of both internal and external analysis functions is to populate the `objectives.json` file with values that characterize the performance of the designs being explored. Figure 68 shows an example objectives file generated during a DSE of the three water tank example. There is an instance of the objectives file created for each simulation in DSE, its primary use being to inform the ranking of designs, but it may be used for any other analysis a user wishes to define.

```
{
    "cumulativeDeviation": 20.47140614676141,
    "vCount": 1
}
```

Figure 68: Contents of `objectives.json` file for a single simulation of the three tank water tank

6.3.7 Ranking

The final part of a DSE configuration file concerns the placing of designs in a partial order according to their performance. The DSE currently supports a Pareto method of ranking, as was shown earlier in Figure 59. The purpose of the ranking section of the configuration is to define the pair of objectives that will be used to rank the designs, and whether to maximize or minimize each. Figure 69 shows an example of a ranking definition from the line following robot example. Here the user has specified that the lap time and mean cross track error objectives will be used to rank. The use of '-' after each indicates that the aim is to minimize both, whereas a '+' indicates the desire to maximize.

```
"ranking": {
    "pareto": {
        "lapTime": "-",
        "meanCrossTrackError": "-"
    }
},
```

Figure 69: Defining parameters and their preferred directions for ranking.

Combining all these sections results in a complete DSE configuration, as shown in Figure 70.

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

        },
        "internalFunctions": {}
    },
    "parameterConstraints": [
        "{sensor1FMU}.sensor1.lf_position_y == {sensor2FMU}.sensor2.lf_position_y",
        "{sensor1FMU}.sensor1.lf_position_x == - {sensor2FMU}.sensor2.lf_position_x"
    ],
    "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 70: A complete DSE configuration for the line follower robot example.

7 Test Automation and Model Checking

Test Automation and Model Checking for INTO-CPS is provided by the RT-Tester RTT-MBT tool. This section first describes installation and configuration of RT-Tester MBT in Section 7.1. It then describes test automation in Section 7.2 and model checking in Section 7.3. Note, that these features are explained in more detail in the deliverables D5.2a [PLM16] and D5.2b [BLM16], respectively.

7.1 Installation of RT-Tester RTT-MBT

In order to use RTT-MBT, a number of software packages must be installed. These software packages have been bundled into two installers:

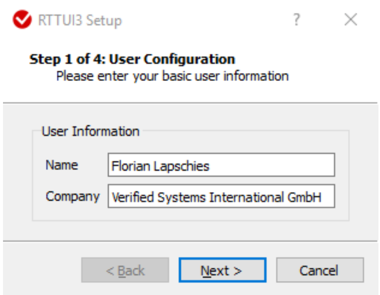
- **VSI tools dependencies bundle:**
  This bundle is required on the Windows platform and installs the following third party software:
  - Python 2.7.
  - GCC 4.9 compiler suite, used to compile FMUs.
- **VSI tools – VSI Test Tool Chain:**
  - RT-Tester 6.0, a stripped version of the RT-Tester core test system that contains the necessary functionality for INTO-CPS.
  - RT-Tester MBT 9.0, the model-based testing extension of RT-Tester.
  - RTTUI 3.9, the RT-Tester graphical user interface.
  - Utility scripts to run RTT-MBT.
  - Examples for trying out RTT-MBT.

These bundles can be downloaded via the download manager of the INTO-CPS Application.

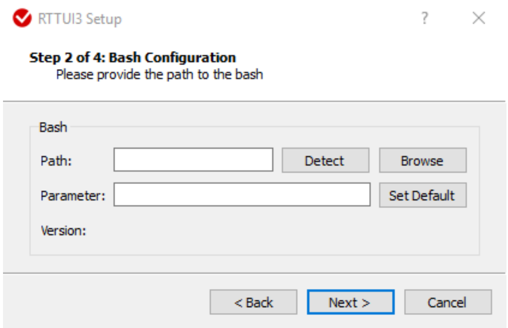
7.1.1 Setup of the RT-Tester User Interface

When the RT-Tester User Interface (RTTUI) is first started, a few configuration settings must be made.

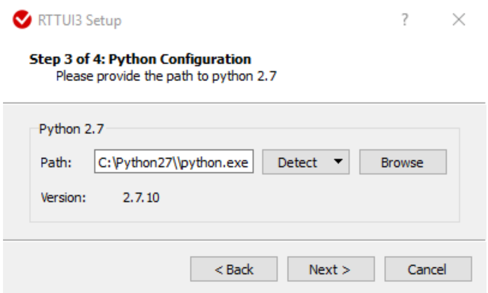
- User name and company name (Figure 71a).
- Location of Bash shell (Figure 71b): You can safely skip this step by clicking *Next*.
- Path to Python 2.7 executable (Figure 71c): Click *Detect* and then *Installation Path* for auto-detection, or *Browse* to select manually.
- Location of RT-Tester (Figure 71d): Click *Browse* to select the directory of your RT-Tester installation. Note that if you did not specify the Bash shell location in step 7.1.1, the version number might not be properly detected.



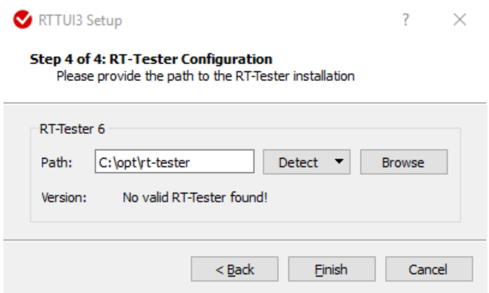
(a) Configuring user.



(b) Configuring Bash.



(c) Configuring Python.



(d) Configuring RT-Tester.

Figure 71: RT-Tester GUI configuration.

7.2 Test Automation

Configuring and using a Test Project involves several activities. These are:

- Creating a test project.
- Defining tests.

- Compiling test driver FMUs.
- Setting up test runs.
- Running tests.
- Evaluating test results.

These activities can be performed either solely using the RT-Tester graphical user interface, or using a combination of the INTO-CPS Application and the RT-Tester GUI. In this section we focus on describing the latter, since it supports the complete set of features necessary for test automation. The INTO-CPS Application currently only exposes a subset of these. A more comprehensive description of the test automation workflow can be found in deliverable D5.2a [PLM16].

In the INTO-CPS Application test automation functionality can be found below the main activity *Test-Data-Generation* in the project browser. Before using most of the test automation utilities, the license management process has to be started. To this, end right-click on *Test-Data-Generation* and select *Start RT-Tester License Dongle* (see Figure 72).

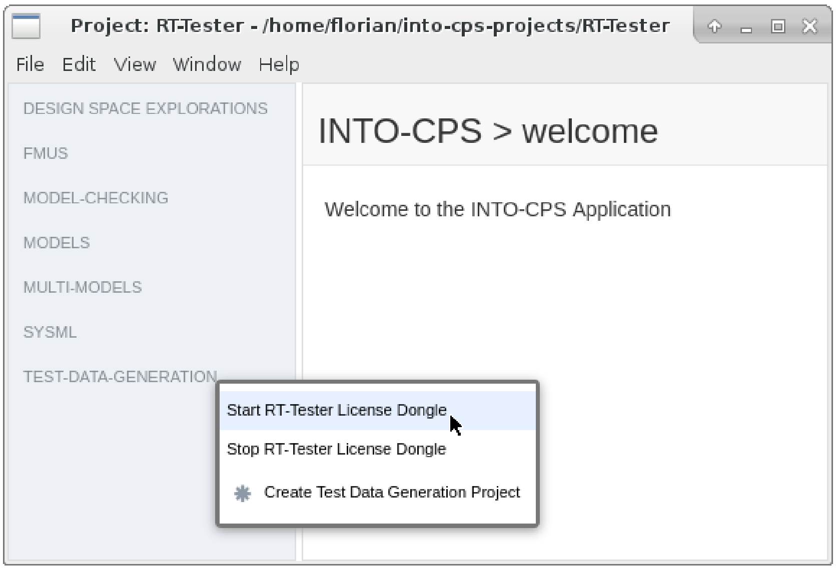


Figure 72: Starting the license management process.

After developing the behavioural model in Modelio and exporting it to an XMI file, test automation projects can be created from the INTO-CPS Application. Such a project is then added as a sub-project within a containing INTO-CPS Application project. To create a project, do the following:

1. Right-click on *Test-Data-Generation* in the project browser and select *Create Test Data Generation Project* (see Figure 73).
2. Specify a name for the project, select the XMI file containing the test model and press *Create*, as shown in Figure 74.

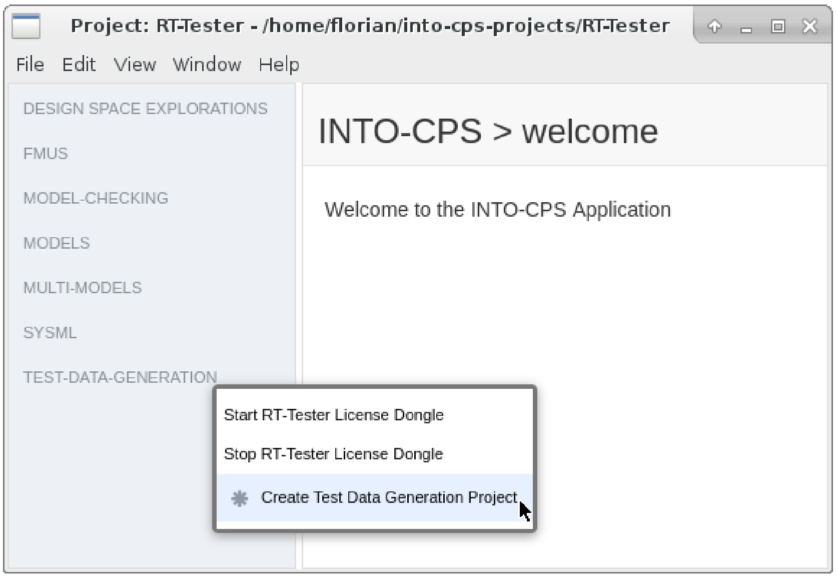


Figure 73: Creating a test automation project.

The newly created sub-project and its directory hierarchy is displayed in the project browser. Some directories and files of the RT-Tester project that are not of great importance to the INTO-CPS workflow are hidden from the browser. The following two folders are of special significance:

- `TestProcedures` contains symbolic test procedures where test objectives are specified in an abstract way, for example by specifying Linear Temporal Logic (LTL) formulas.
- From these symbolic test procedures, concrete executable (RT-Tester 6) test procedures are generated, which then reside in the folder `RTT_TestProcedures`.

The specification of test objectives is done using the RT-Tester GUI. The relevant files can be opened in the RT-Tester GUI directly from the INTO-CPS Application by double-clicking them:

- `conf/generation.mbtconf` allows you to specify the overall test objectives of the test procedure. Test objectives can be specified as LTL formulas, which must then be fulfilled during a test run. Test

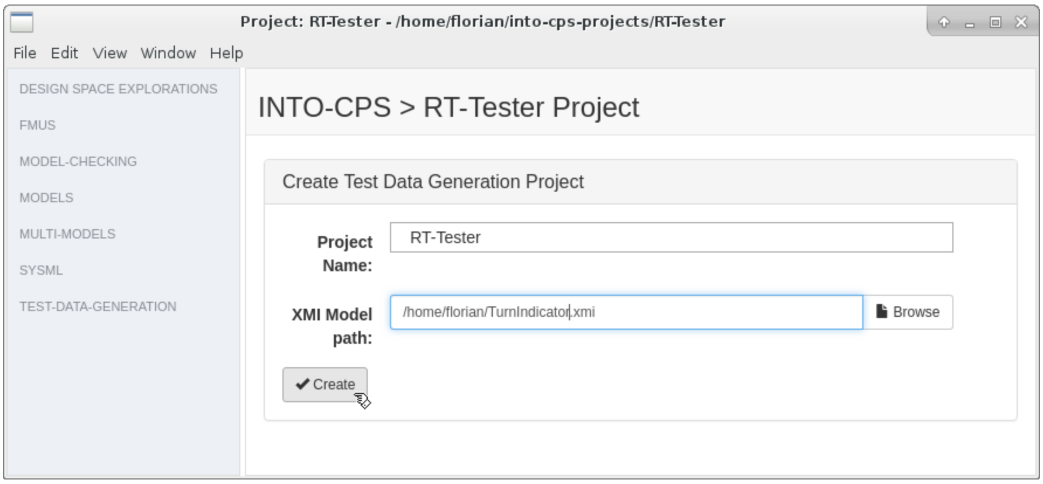


Figure 74: Test automation project specifics.

goals can also be specified by selecting structural elements from a tree representation of the test model and then choosing a coverage metric for that element. For example, the user might select a sub-component of the System Under Test (SUT) and specify that all Basic Control States (BCS) must be reached (see Figure 75), or that all transitions must be exercised (TR) in a test run.

- `conf/signalmap.csv` allows you to configure the input and output signals of the system under test (see Figure 76). This includes defining the admissible signal latencies for checking the SUT's outputs in a test run. This file also allows you to restrict the range of the signals in order to constrain these values during test data generation.

More details on the definition of tests can be found in deliverable D5.2a [PLM16].

After defining the test objectives, a concrete test case can be created by right-clicking on the symbolic test case under *TestProcedures* and then selecting *Solve* (see Figure 77).

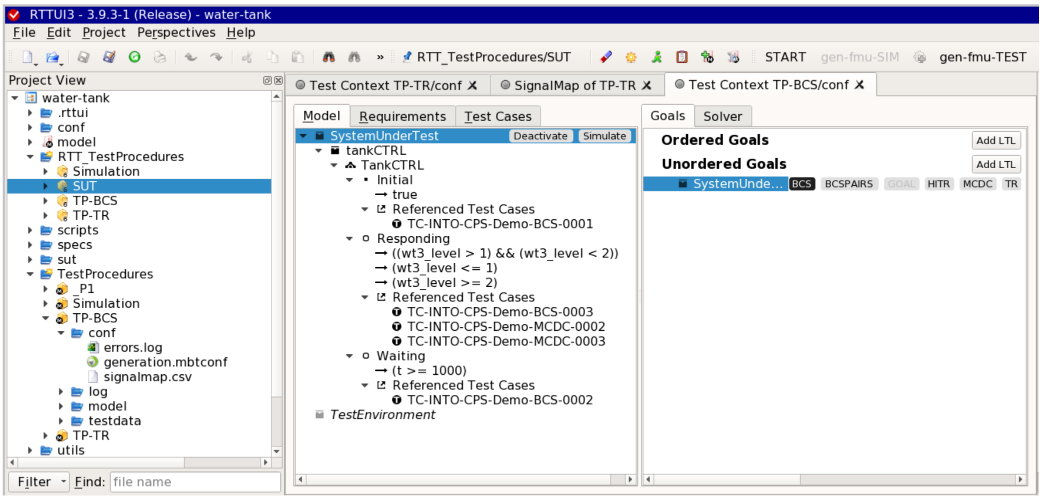


Figure 75: Configuring a test goal.

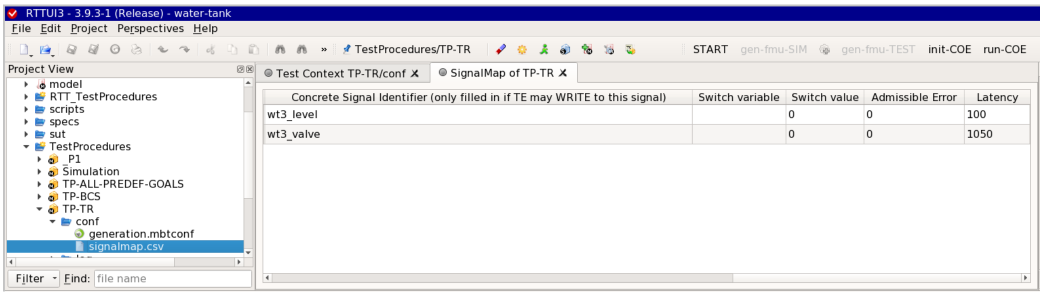


Figure 76: Configuring signals.

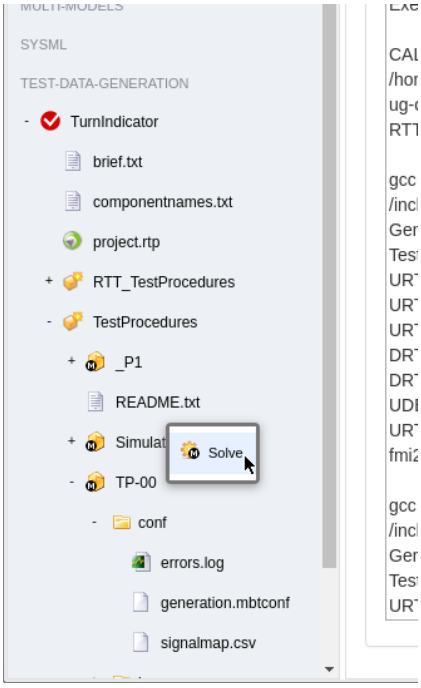


Figure 77: Generating a concrete test procedure.

A solver component then computes the necessary timed inputs to realize the test objectives. A concrete test procedure is generated that feeds a system under test with these inputs and observes its outputs against expected results derived from the test model. This test procedure will be placed in `RTT_TestProcedures` and has the same name as the symbolic test procedure. Figure 78 shows how test generation progresses.

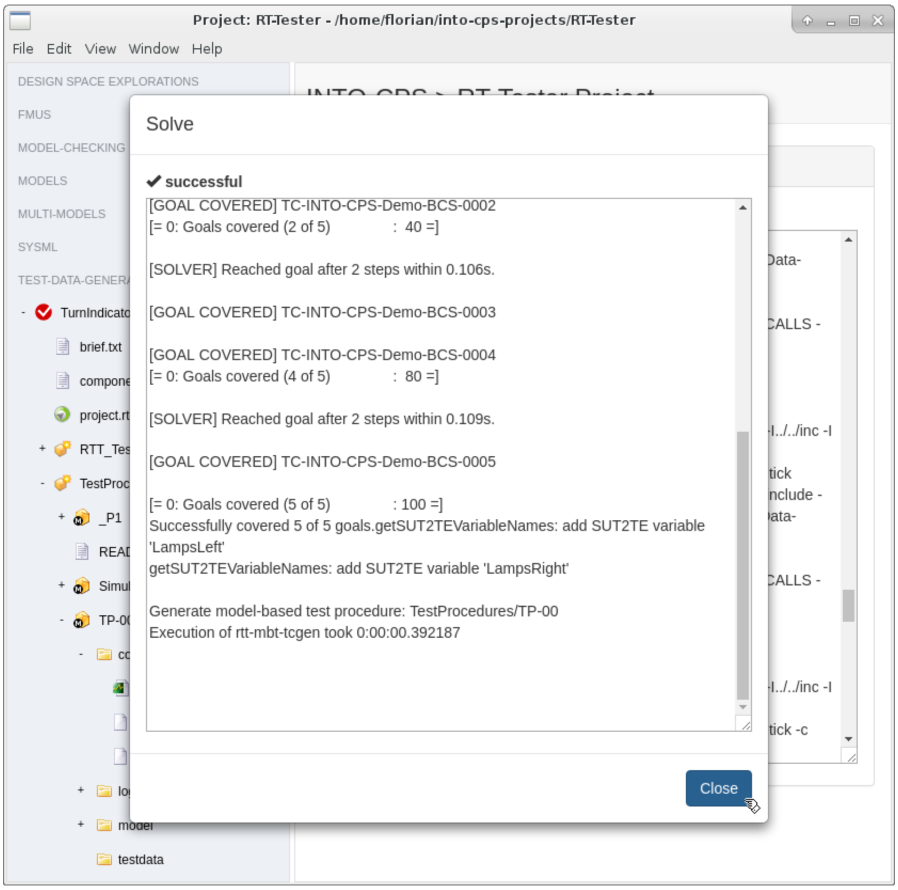


Figure 78: Test data generation progress.

A generated test procedure can be cast into an FMU, which can then be run in a co-simulation against the system under test. To this end, right click on the concrete test procedure and select *Generate Test FMU* (see Figure 79). In cases where a real and perhaps physical system under test is not available, a simulation of the system under test can be generated from the behavioural model. To generate such an FMU, right-click on *Simulation* an select *Generate Simulation FMU* as depicted in Figure 80.

In order to run a test, right-click on the test procedure and select *Run Test*

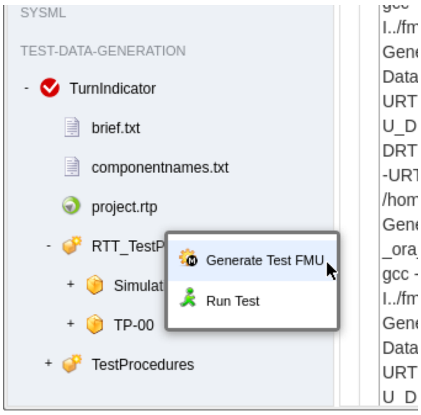


Figure 79: Generating a test FMU.

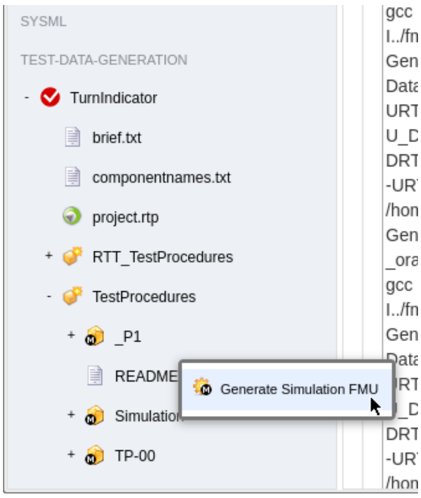


Figure 80: Generating a simulation FMU.

(see Figure 81). Then specify the FMU of the system under test. If the system under test is to be replaced by a simulation, press on the corresponding *Simulation* button. The duration of the test is derived during test data generation and does not need to be manually specified. However, an appropriate step size must be set. Finally, after making sure the COE is running, press *Run* to start the test (see Figure 82).

Every test execution yields as its result an evaluation of test cases, *i.e.*, each is associated with a verdict of PASS, FAIL, or INCONCLUSIVE.[10] The details are found in the test log files below the folder `testdata`. See the RT-Tester

[10]The verdict can also be NOT TESTED. This means a test case has been included in a test procedure, but a run that reaches it is still missing.

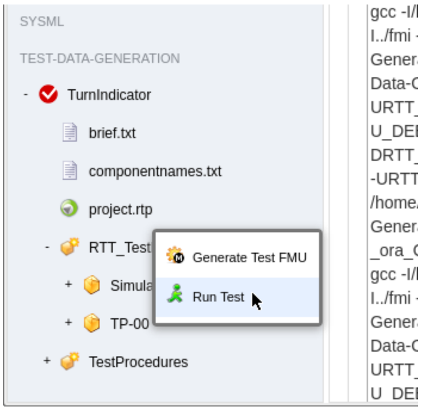


Figure 81: Running a test.

user manual [Ver15a] for details.

The file `testcase_tags.txt` gives a condensed record of test case, verdict, and point in a `*.log` file where a corresponding PASS, FAIL, or—in case of INCONCLUSIVE—test case occurrence without assertion can be found. The project-wide test-case verdict summary as well the requirement verdict summary can be found in the folder `RTT_TestProcedures/verification`. More details on the evaluation of test runs can be found in deliverable D5.2a [PLM16].

7.3 Model Checking

This section describes how to use the INTO-CPS Application as a front-end to the LTL model checker of RT-Tester RTT-MBT. More details on the algorithms used and the syntax of LTL formulas can be found in deliverable D5.2b [BLM16].

Once an INTO-CPS project has been created (see Section 4.2), model checking functionality can be found under the top-level activity *Model Checking* in the project browser. Before getting started, the RT-Tester license management process must be launched. To this end, right-click on *Model Checking* and select *Start RT-Tester License Dongle* (see Figure 83). Model checking projects are presented as sub-projects of INTO-CPS Application projects. In order to add a new project,

1. Right-click on the top-level activity *Model Checking* in the project browser and select *Create Model Checking Project* (see Figure 84).

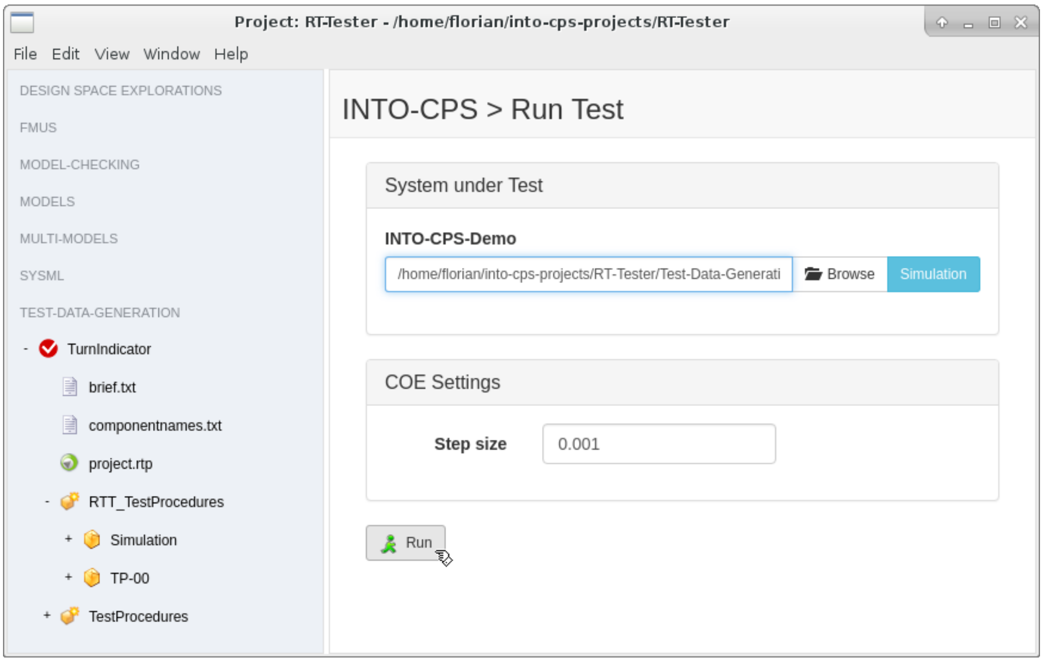


Figure 82: Configuring a test.

2. Provide a project name and the model that has been exported to XMI from Modelio.

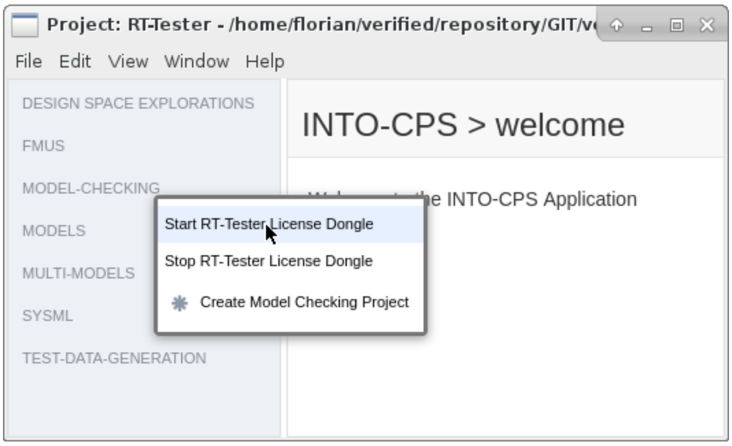


Figure 83: Starting the RT-Tester license dongle.

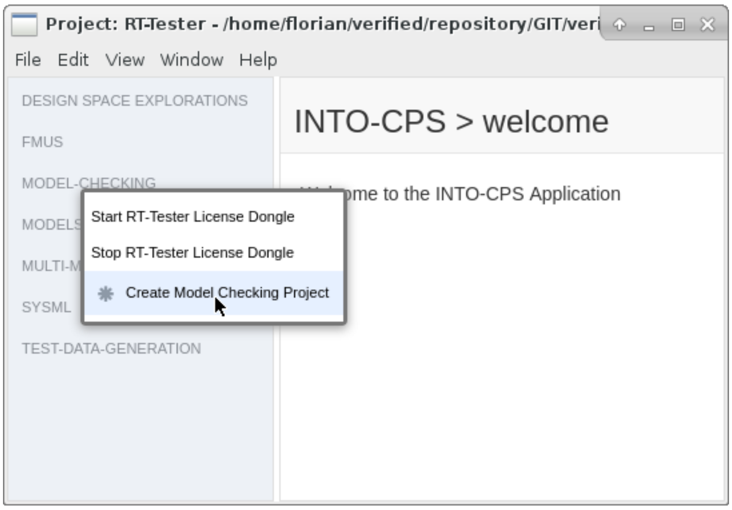


Figure 84: Creating a model checking project.

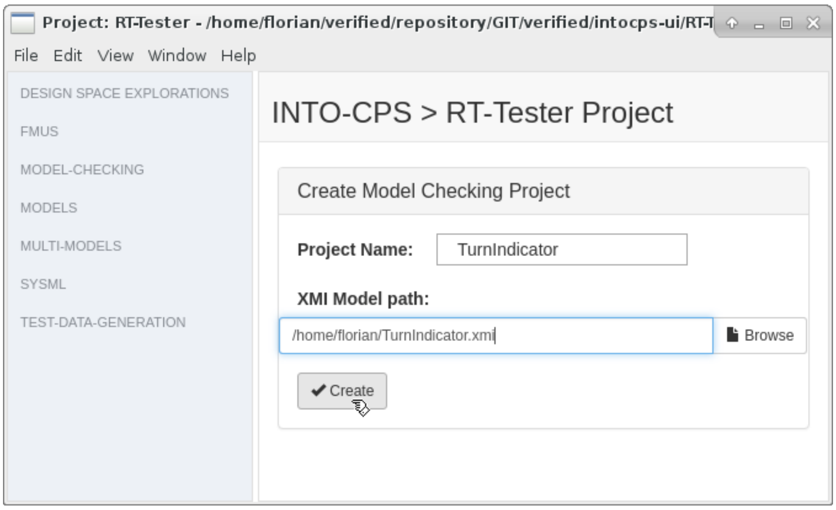


Figure 85: Specifying the model checking project.

After pressing *Create*, a new node representing the model checking project is added to the project browser.

The next step is to add LTL queries to the project:

1. Right click on the project and select *Add LTL Query* (see Figure 86).
2. Enter a name for the new query (see Figure 87).
3. To edit the LTL query, double click on the corresponding node in the project browser (see Figure 88). The LTL formula can then be edited in a text field. Note that the editor supports auto-completion for variable names and LTL operators (see Figure 89).
4. Provide the upper bound for the bounded model checking query.

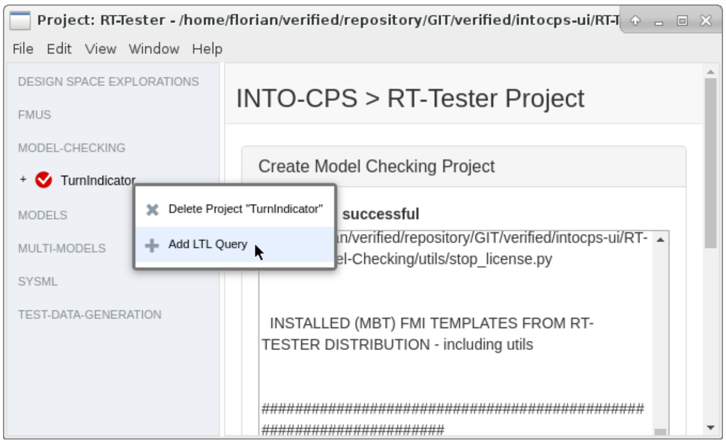


Figure 86: Adding an LTL formula.

Figure 87: Naming the new LTL formula.

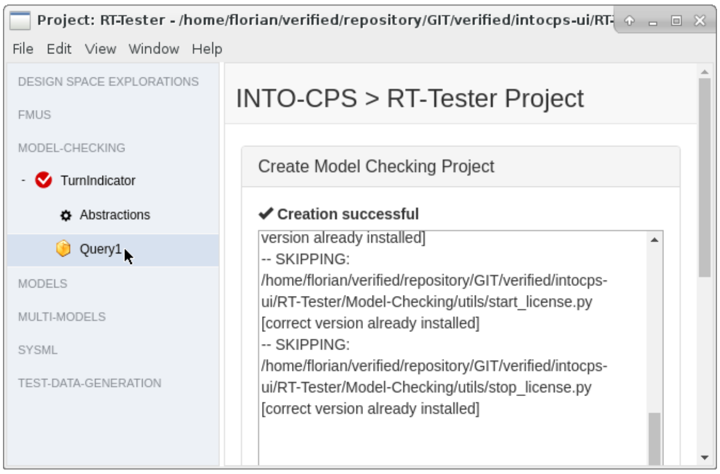


Figure 88: Opening the LTL formula editor.

To check the query, press *Save & Check*. A window opens and is filled with the output of the model checking tool. The tool either reports that the query holds within the specified number of steps — as depicted in Figure 90 — or it prints a counterexample to demonstrate that the property does not hold.

It is possible to configure abstractions[11] for a particular model checking project. To do so, double-click on the corresponding *Abstractions* node below that project in the project browser. It is then possible to choose an abstraction method for each output variable of an environment component along with making the associated setting. In Figure 91 the interval abstraction has been selected for the output variable `voltage`. This abstraction has further been configured to restrict the variable's value within the interval $[10, 12]$. After pressing *Save*, this abstraction is applied to all model checking queries in the current model checking project.

[11] Information on abstractions and their associated configuration items can be found in deliverable D5.2b [BLM16].

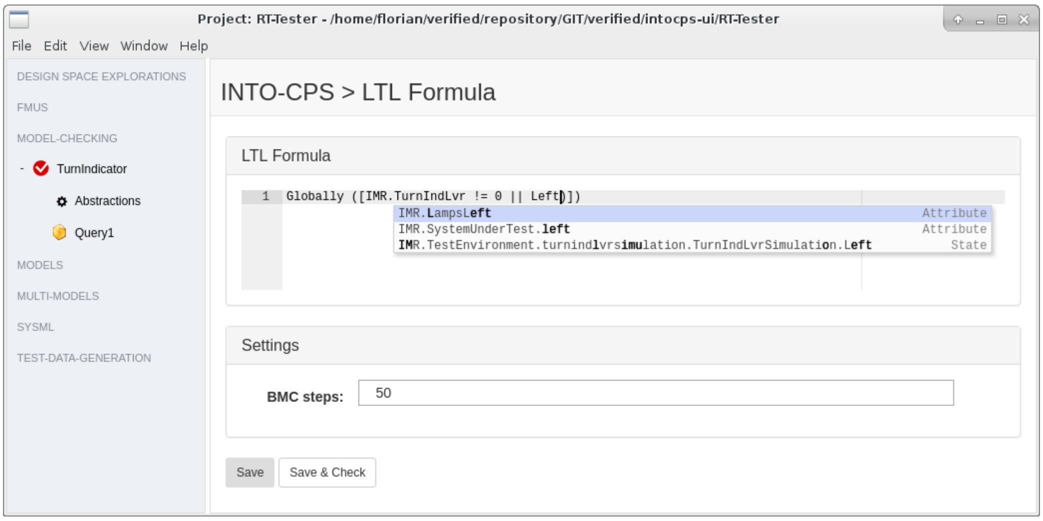


Figure 89: LTL formula editor.

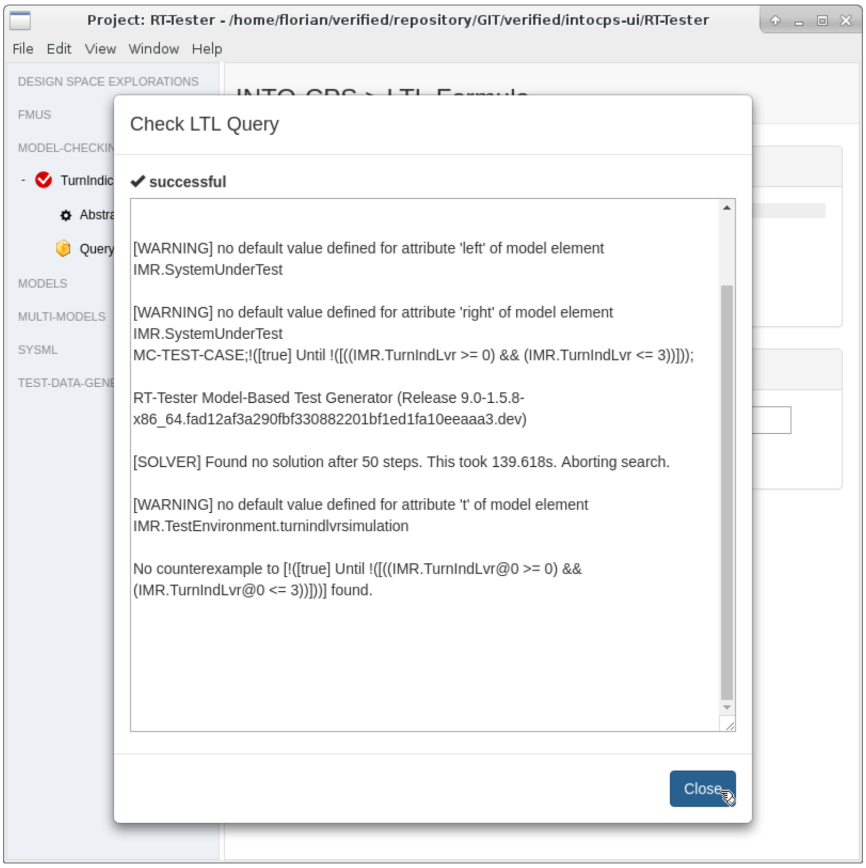


Figure 90: Model checking result.

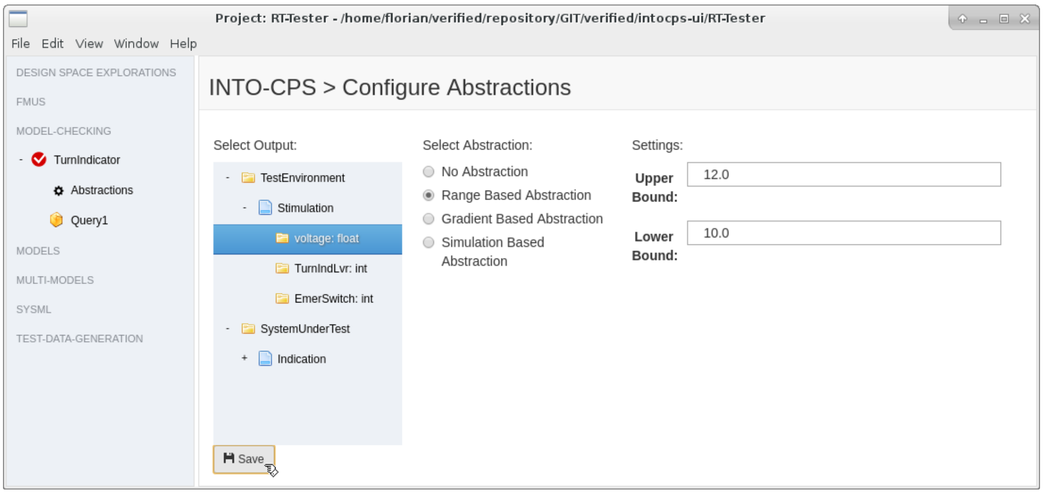


Figure 91: Configuring abstractions.

8 Traceability support for INTO-CPS

This section provides a description of tool support for traceability developed as part of the INTO-CPS project.

8.1 Overview

Traceability support is divided into two steps: sending data from the tools to the traceability database, and retrieving information from the database. Currently, only the first part is available in prototypes in the different tools. This is documented below.

8.2 INTO-CPS application

The traceablility daemon is now (since INTO-CPS App 2.1.19 RC) integrated in the App and it starts with the App. Only Neo4J has to be downloaded. To do so, one can use the download-manager of the INTO-CPS App. When downloaded, Neo4J needs to be extracted by hand into the folder `<user>/into-cps-projects/install` (the archive file is located at `<user>/into-cps-projects/install_downloads` after download). Note that Neo4J is a singleton, so make sure all other instances of Neo4J are down before starting the App.

Treaceability information is captured by the traceability daemon and stored in a Neo4J database. The database is project specific and is deployed on project change within the App. When running, Neo4J is accessible at `http://localhost:7474`. Here one can view the current traceability graph.

Username and password of the databases are always:

username = intoCPSApp
password = KLHJiK8k2378HKsg823jKKLJ89sjklJHBNf8j8JH7FxE

To view the raw data from the database, right-click on the "traceability" entry in the project browser (in the App) and select "view traceability graph" (see figure 92). Select the database symbol, and click in "relationship types" on "Trace". This shows you the graph database. By default, the view is limited to 25 entries. To change this, edit the line `MATCH p=()-[r:Trace]->() RETURN p LIMIT 25` and set the limit to a different value.

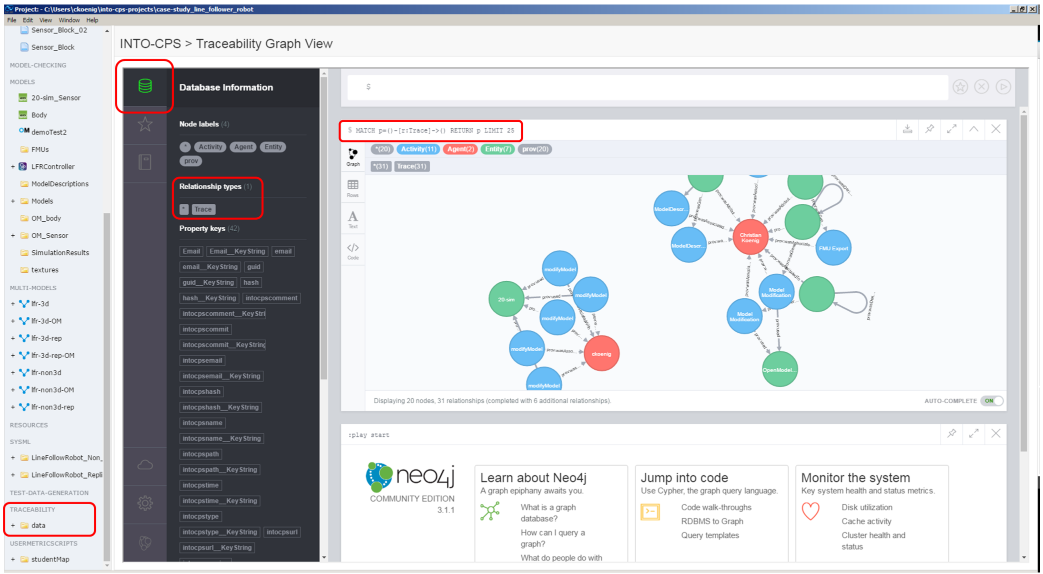


Figure 92: Current view of the traceability in the app

8.3 Modelio

The Modelio module can be downloaded here: `https://www.dropbox.com/s/bad36t9f8x4n0gl/INTOCPS_1.1.03.jmdac?dl=0`. Modelio supports traceability for the following modelling activities:

- Model creation
- Model modification

Steps:
Go to *Configuration* > *Modules...* Select *INTO-CPS* and set the parameters. To commit a change, right click on any element and use the *INTO-CPS* > *Commit* command.

8.4 OpenModelica

The latest nightly builds of OpenModelica support traceability:

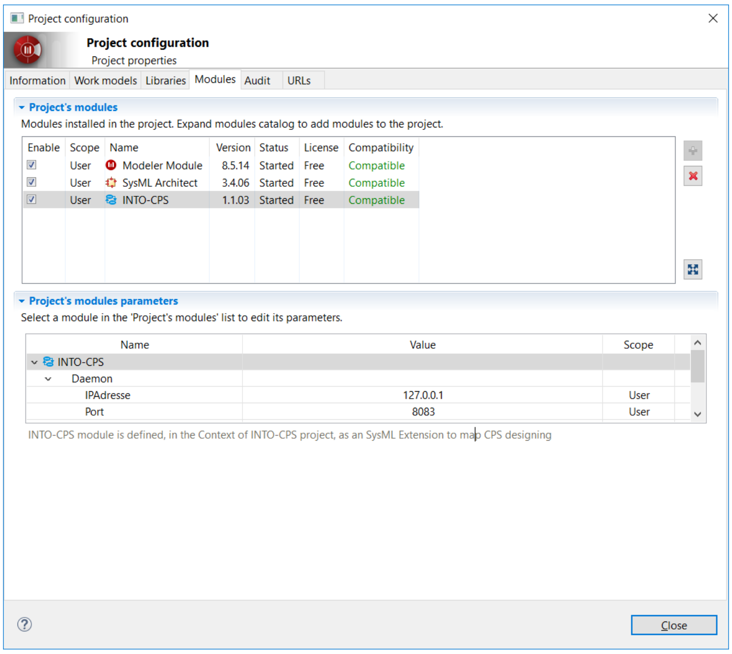


Figure 93: Configuration of traceability features in Modelio

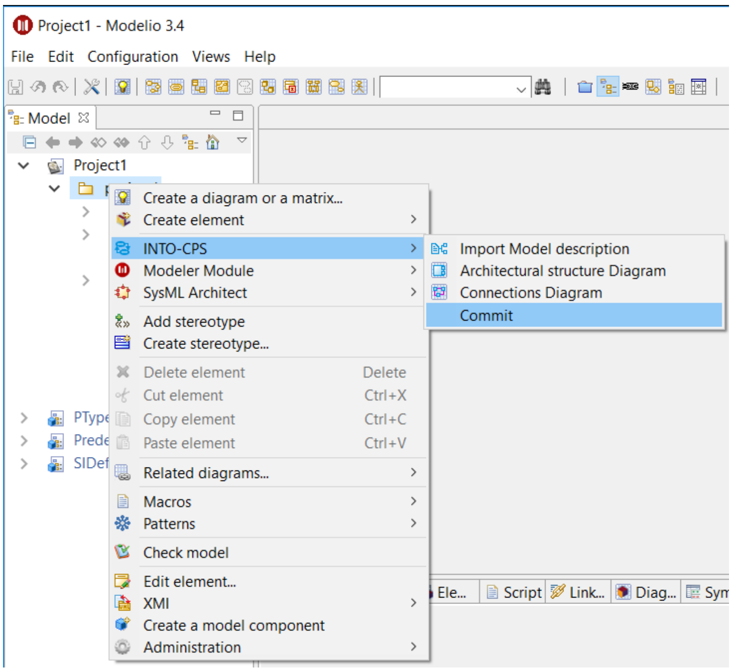


Figure 94: Commit the traceability information in Modelio

Win32: `https://build.openmodelica.org/omc/builds/windows/nightly-builds/32bit/OpenModelica-latest.exe` Win64: `https://build.openmodelica.org/omc/builds/windows/nightly-builds/64bit/OpenModelica-latest.exe`

OpenModelica supports tracing the following modeling activities:

- Model creation
- Model modification
- FMU export
- Model description XML import

As a prerequisite for traceability support, Git should be installed in the system.

To configure the traceability support, go to *Tools > Options > Traceability*, select the traceability checkbox and set all the fields, the traceability daemon IP-Address and Port (see Figure 95). By default, the port is 8083.

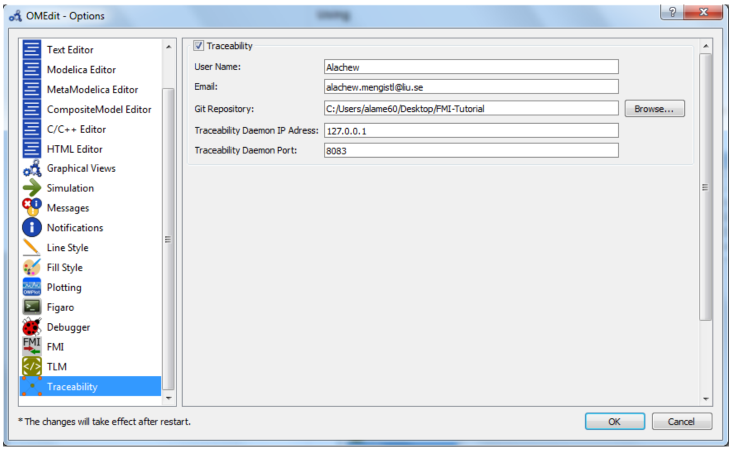


Figure 95: Configure the traceability settings in OpenModelica

Then, go to *Tools > Options > General* and set the working directory to which you would like to export the FMU (see Figure 96).

Create a Modelica model via *File > New Modelica Class* or load a model via *File > Open Model/LibraryFile(s)*, see Figure 97.

After modification of the model/class, click the *File > Save* button, press Ctrl + s or click the Save button from the menu bar shown below in the

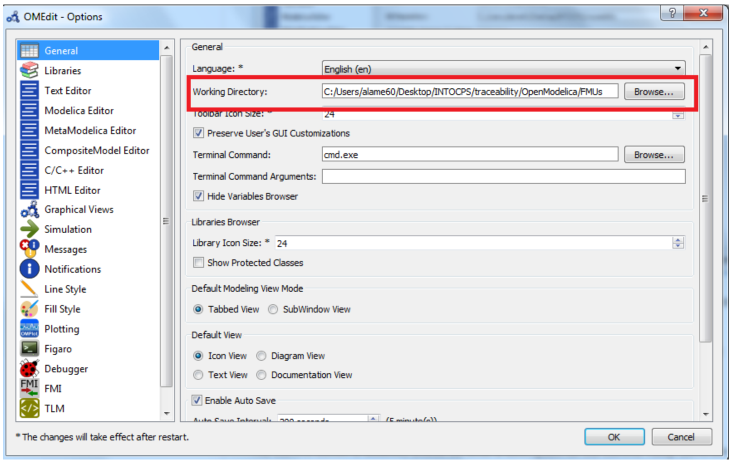


Figure 96: Set the FMU export directory in OpenModelica

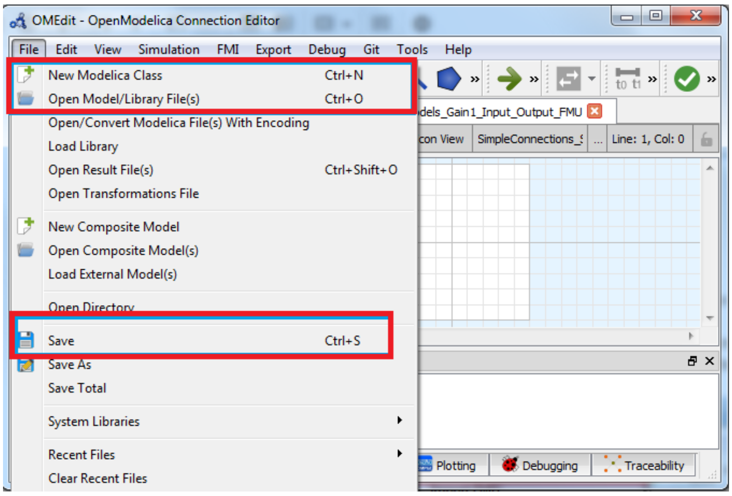


Figure 97: Create or open a Class in OpenModelica

Figure 98. A dialog as shown below in Figure 99 will appear to enter the commit description.

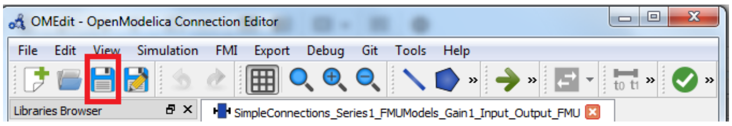


Figure 98: Save a model in OpenModelica

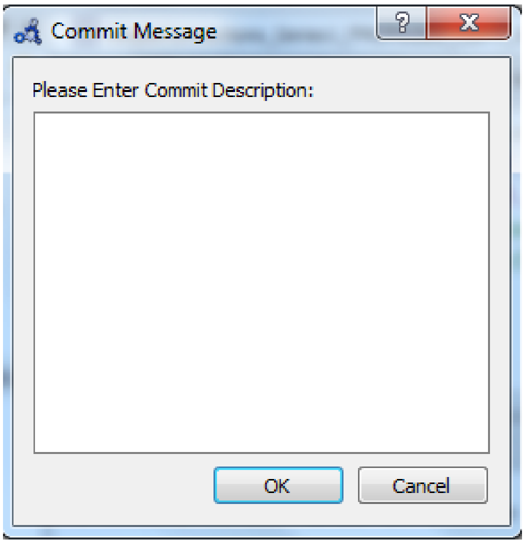


Figure 99: The commit message in OpenModelica

To trace the export of an FMU, load the Modelica model or create a new model, then go to *FMI > Export FMU* (see Figure 100). OpenModela generates the FMU, commits and send the traceability information to daemon automatically.

To import a `modelDescription.xml` file, go to *FMI > Import FMU Model Description*. A dialog as shown in Figure 101 will appear. Select the `modelDescription.xml` file and the output directory then press *OK*. The Modelica model with SysML block inputs and outputs will be generated and automatically loaded (see the left part of Figure 101). To send the traceability information, double click on the model then go to *Git > Traceability > Push Traceability Information*.

To visualize the traceability graph, click on the Traceability perspective button shown below.

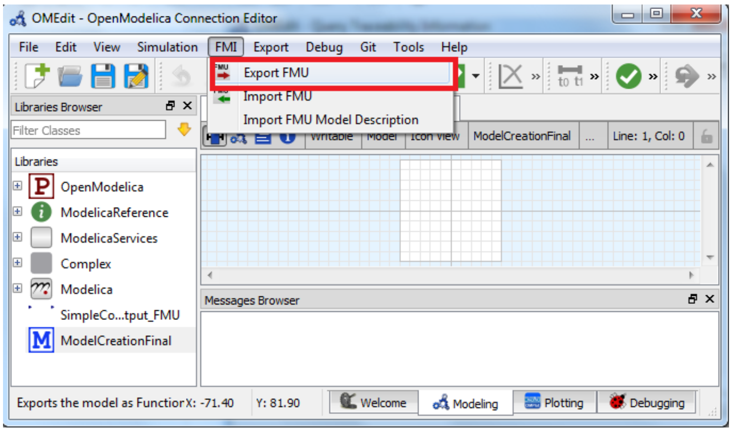


Figure 100: FMU export in OpenModelica

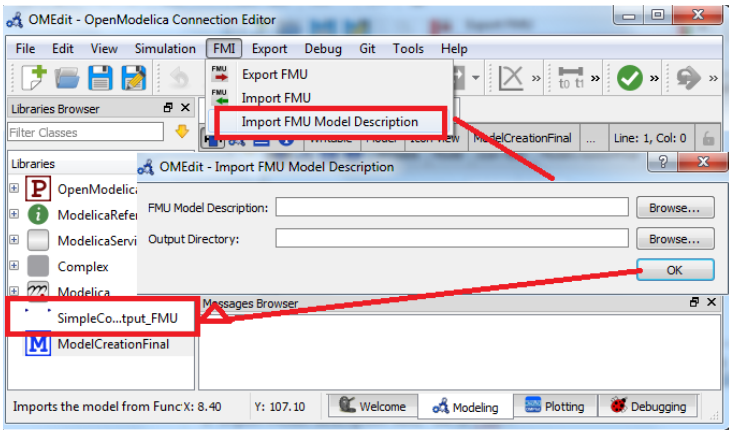


Figure 101: FMI model description import in OpenModelica

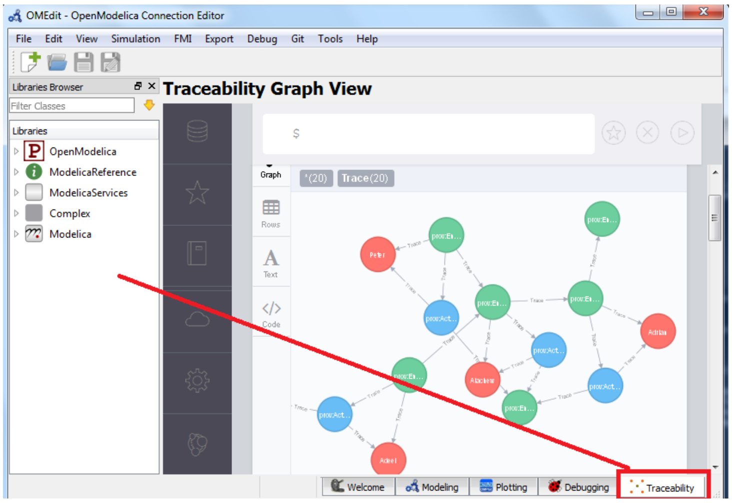


Figure 102: View traceability data in OpenModelica

8.5 20-sim

Use any version of 20-sim 4.6.3-intocps or higher. The one in the download manager for version 2.1.19 RC is not sufficient. The first suitable release bundle is 0.0.12.

The download can be found here:

`https://www.dropbox.com/s/lgr461ddb97kl8h/20-sim-4.6.3.7711-intocps-win32.exe?dl=1.`

During installation, make sure you keep the Python option enabled. This is necessity, even if you already have another Python installation on your PC. This Python version will only overwrite Python versions you installed earlier with 20-sim, it will not install other Python versions.

Currently, the actions "create model" and "modify model" are supported by 20-sim

In 20-sim, go to *Tools* > *Version Control Toolbox* > *Traceability*. First enable "GIT version control" and insert a GIT repository, which can be an existing GIT repository or a folder in the local file system. The model will be committed to this repository on a "save" (modify) or "save as" (create) action. If the model does not reside in the GIT repository, it will also be copied to the GIT repository on a "save" or "save as" action.

You can leave the "Write custom save messages" option unchecked, as it is not currently fully functional.

If you would like to send data to the traceability daemon as well, then you can enable "INTO-CPS Traceability Daemon". Below, you can then enter the IP-address and Port of the daemon. If you run the INTO-CPS application and traceability deamon locally, the IP-address is *localhost* and the port is 8083 by default.

Now, pressing "save" or "save as" in any form, will (copy and) commit your model to the GIT repository, and then send the action you just performed to the traceability daemon.

9 Code Generation for INTO-CPS

Of all the INTO-CPS tools, Overture, OpenModelica and 20-sim have the ability, to varying degrees, to translate models into platform-independent C source code. Overture can moreover translate VDM models written in the executable subset of VDM++ [LLB11] (itself a subset of VDM-RT) to Java, but C is the language of interest for the INTO-CPS technology.

The purpose of translating models into source code is twofold. First, the source code can be compiled and wrapped as standalone FMUs for co-simulation, such that the source tool is not required. Second, with the aid of existing C compilers, the automatically generated source code can be compiled for specific hardware targets.

The INTO-CPS approach is to use 20-sim 4C to compile and deploy the code to hardware targets, since the tool incorporates the requisite knowledge regarding compilers, target configuration *etc.* This is usually done for control software modelled in one of the high-level modelling notations, after validation through the INTO-CPS tool chain. Deployment to target hardware is also used for SiL and HiL validation and prototyping.

For each of the modelling and simulation tools of the INTO-CPS tool chain, code generation is a standalone activity. As such, the reader should refer to the tool-specific documentation referenced in Appendix B for guidance on code generation. Deliverable D5.1d [HLG+15] contains the details of how each tool approaches code generation.

The remainder of this section lists information about the code generation capabilities of each tool. It describes what the user can expect currently from each tool's code generator, in the hopes that this will be helpful in eliminating stumbling blocks for new users trying to quickly get started with the INTO-CPS tool chain. Extensive guidance on how to tailor models for problem-free translation to code can be found in the tools' individual user manuals, as referenced in Appendix B.

9.1 Overture

A complete description of Overture's C code generator can be found in the Overture User Manual, accessible through Overture's Help system. As a quick-start guide, this section only provides an introduction to invoking the C code generator, and an overview of the features of VDM-RT that are currently considered stable from a code generation point of view. Please note that exporting a source code FMU with Overture (Section 5.1) automatically invokes the code generator and packages the result as an FMU.

The C code generator is invoked from the context menu in the Project Explorer as shown in Figure 103. The code generator currently supports the

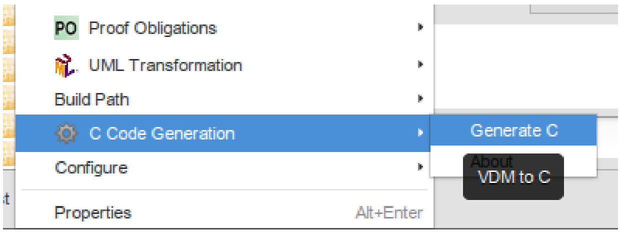


Figure 103: Invoking the code generator.

following VDM-RT language constructs:

- Basic data types and operations: integers, reals, booleans, *etc.*
- The `is_` type test for basic types.
- Quote types.
- `let` expressions.
- Pattern matching.
- For and while loops.
- `case` expressions.
- Record types.
- Products.
- Aggregate types and operations: sets, sequences, maps (to a limited extent).
- Object-oriented features: classes and class field access, inheritance, method overloading and overriding, the `self` keyword, subclass responsibility, `is not yet specified`, multiple constructors, and constructor calls within constructors.
- The `time` expression.

The following language features are not yet supported:

- Lambda expressions.

- Pre-conditions, post-conditions and invariants.
- Quantifiers.
- Type queries on class instances.
- File I/O via the I/O library.

Most importantly, the development of Overture's C code generator is now being geared toward resource-constrained embedded platforms. Improvements are currently being made to enable deployment of the generated code on PIC and ATmega microcontrollers.

A key feature of this development is the use of a garbage collector for memory management. Generating a VDM-RT model to C code via the context menu results in a `main.c` file containing a skeletal `main()` function. This function contains calls to `vdm_gc_init()` and `vdm_gc_shutdown()`, the garbage collector initialization and shutdown functions. The collector proper can not be invoked automatically, so calls to the essential function `vdm_gc()` must be inserted manually in the main code, for instance after each repetition of a cyclic task. The source code FMU exporter, on the other hand, can handle automatic invocation of the garbage collector, so no manual intervention is required. Please note that it is generally unsafe to insert calls to `vdm_gc()` in the generated code.

9.2 20-sim

20-sim supports ANSI-C and C++ code generation through the usage of external and user-modifiable code-generation templates. Currently only a subset of the supported 20-sim modelling language elements can be exported as ANSI-C or C++ code code. The exact supported features depend on the chosen template and its purpose and are discussed in Section 5.2.

The main purpose of the 20-sim code generator is to export control systems. Therefore the focus in on running code on bare-bone targets (*e.g.* Arduino) or as a real-time task on a real-time operating system.

The code generated by 20-sim does not contain any target-related or operating system specific code. The exported code is generated such that it can be embedded in an external software project. For running 20-sim generated code on a target, you can use 20-sim 4C. This is a tool that extends the 20-sim generated code with target code based on target templates [Con16].

9.3 OpenModelica

OpenModelica supports code generation from Modelica to source-code targeting both ANSI-C and C++. From the generated source code, co-simulation and model-exchange FMUs can be built. Currently, the only supported solver in the generated co-simulation FMUs is forward Euler. Work to support additional solvers is underway. The ability to deploy the generated code to specific hardware targets will be supported via 20-sim 4C.

9.4 RT-Tester/RTT-MBT

When generating test FMUs from SysML discrete-event state-chart specifications using RTTester/RTT-MBT, the user should be aware of the following sources of errors:

- Livelock resulting from a transition cycle in the state-chart specification in which all transition guards are true simultaneously. This can be checked separately using a livelock checker.
- Race conditions arising from parallel state-charts assigning different values to the same variable. Model execution in this case will deadlock.
- State-charts specifying a replacement SUT must be deterministic.

10 Issue handling

Should you experience an issue while using one or more of the INTO-CPS tools, please take the time to report the issue to the INTO-CPS project team, so we can help you resolve it as soon as possible.

The following three small sub-sections will guide you through the three simple steps of issue handling and reporting.

10.1 Are you using the newest INTO-CPS release?

Before you go any further with your current issue, please check that the INTO-CPS version you are using is the newest. The version number is part of the file name of the ZIP-bundle of the release. To find the list of released

INTO-CPS bundle versions, and to see what the current version of INTO-CPS is, please visit

    https://github.com/into-cps/intocps-ui/releases/

10.2 Has the issue already been reported?

To make it easy for you to check whether the issue you are experiencing is an already known one, we have created a list of all currently known issues across all the INTO-CPS tools, with links directly to the online issue report page of the relevant tool supplier. Have a quick look at the list, and if your issue is already known, we recommend you follow the link and read more about the specifics of the issue. Perhaps someone has found a work-around or perhaps you have new information to add that might help the developers solve the issue faster.

For the list of currently known issues, please visit

    http://into-cps.github.io/weekly-issue/index.html

Note that some of the issue tracker sites might require you to register before you can view or submit issues. Registration is free.

10.3 Reporting a new issue

If you have followed the steps in the two previous sections and are now certain that you have spotted a new issue relating to a specific INTO-CPS tool, please visit the issue tracker site for that tool and report it. To ease this process we have listed direct links for each tool to their relevant online issue reporting page. To see the list of issue tracker links please visit

    http://into-cps.github.io/report-an-issue.html

11 Conclusions

This deliverable is the user manual for the INTO-CPS tool chain after the second year of the project. The tool chain supports model-based design and validation of CPSs, with an emphasis on multi-model co-simulation.

Several independent simulation tools are orchestrated by a custom co-simulation orchestration engine, which implements both fixed and variable step size co-simulation semantics. A multi-model thus co-simulated can be further verified through automated model-based testing and bounded model checking.

The tool chain benefits from a cohesive management interface, the INTO-CPS Application, the main gateway to modelling and validation with the INTO-CPS technology. Following the manual should give a new user of the INTO-CPS tool chain an understanding of all the elements of the INTO-CPS vision for co-simulation. This manual is accompanied by tutorial material and guidance on the main INTO-CPS tool chain website,

`http://into-cps.github.io`

Features that have not yet been fully developed or integrated with the INTO-CPS Application are currently being addressed and are targeted for the final year of the INTO-CPS project.

References

[ACM+16] 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.

[BHJ+06] Armin Biere, Keijo Heljanko, Tommi A. Juntilla, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.

[BHPG16] Victor Bandur, Miran Hasanagic, Adrian Pop, and Marcel Groothuis. FMI-Compliant Code Generation in the INTO-CPS Tool Chain. Technical report, INTO-CPS Deliverable, D5.2c, December 2016.

[BLL+15] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Sune Wolff, Carl Gamble, Adrian Pop, Etienne Brosse, Jörg Brauer, Florian Lapschies, Marcel Groothuis, and Christian Kleijn. User Manual for the INTO-CPS Tool Chain. Technical report, INTO-CPS Deliverable, D4.1a, December 2015.

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

[Blo14] Torsten Blochwitz. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads`, July 2014.

[BQ16] Etienne Brosse and Imran Quadri. SysML and FMI in INTO-CPS. Technical report, INTO-CPS Deliverable, D4.2c, December 2016.

[Bro97] Jan F. Broenink. Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD*, 38(3):22–25, 1997.

[CFTW16] Ana Cavalcanti, Simon Foster, Bernhard Thiele, and Jim Woodcock. Initial semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.2c, December 2016.

[Con13] Controllab Products B.V. http://www.20sim.com/, January 2013. 20-sim official website.

[Con16] Controllab Products B.V. http://www.20sim4C.com/, October 2016. 20-sim 4Cofficial website.

[CW16] Ana Cavalcanti and Jim Woodcock. Foundations for FMI comodelling. Technical report, INTO-CPS Deliverable, D2.2d, December 2016.

[Fav05] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, March 2005.

[FCC+16] Simon Foster, Ana Cavalcanti, Samuel Canham, Ken Pierce, and Jim Woodcock. Final Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.2b, December 2016.

[FE98] 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.

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

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, January 2004.

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

[GFR+12] Anand Ganeson, Peter Fritzson, Olena Rogovchenko, Adeel Asghar, Martin Sjölund, and Andreas Pfeiffer. An OpenModelica Python interface and its use in pysimulator. In Martin Otter and Dirk Zimmer, editors, *Proceedings of the 9th International Modelica Conference*. Linköping University Electronic Press, September 2012.

[HLG+15] Miran Hasanagić, Peter Gorm Larsen, Marcel Groothuis, Despina Davoudani, Adrian Pop, Kenneth Lausdahl, and Victor Bandur. Design Principles for Code Generators. Technical report, INTO-CPS Deliverable, D5.1d, December 2015.

[KG16] C. Kleijn and M.A. Groothuis. *Getting Started with 20-sim 4.5*. Controllab Products B.V., 2016.

[KGD16] C. Kleijn, M.A. Groothuis, and H.G. Differ. *20-sim 4.6 Reference Manual*. Controllab Products B.V., 2016.

[KR68] D.C. Karnopp and R.C. Rosenberg. *Analysis and Simulation of Multiport Systems: the bond graph approach to physical system dynamic*. MIT Press, Cambridge, MA, USA, 1968.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[LBF$^+$10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[Lin15] Linköping University. http://www.openmodelica.org/, August 2015. OpenModelica official website.

[LLB11] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. A Deterministic Interpreter Simulating A Distributed real time system using VDM. In Shengchao Qin and Zongyan Qiu, editors, *Proceedings of the 13th international conference on Formal methods and software engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, Heidelberg, October 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

[LLJ$^+$13] Peter Gorm Larsen, Kenneth Lausdahl, Peter Jørgensen, Joey Coleman, Sune Wolff, and Nick Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, April 2013.

[LLW$^+$15] Kenneth Lausdahl, Peter Gorm Larsen, Sune Wolf, Victor Bandur, Anders Terkelsen, Miran Hasanagić, Casper Thule Hansen, Ken Pierce, Oliver Kotte, Adrian Pop, Etienne Brosse, Jörg Brauer, and Oliver Möller. Design of the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D4.1d, December 2015.

[LNH$^+$16] 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.

[LRVG11] Kenneth G. Lausdahl, Augusto Ribeiro, Peter Visser, and Frank Groen. D3.2b co-simulation. DESTECS Deliverable D3.2b, The DESTECS Project (INFSO-ICT-248134), January 2011.

[Ope] Open Source Modelica Consortium. OpenModelica User's Guide.

[PBLG15] Adrian Pop, Victor Bandur, Kenneth Lausdahl, and Frank Groen. Integration of Simulators using FMI. Technical report, INTO-CPS Deliverable, D4.1b, December 2015.

[PBLG16] Adrian Pop, Victor Bandur, Kenneth Lausdahl, and Frank Groen. Updated Integration of Simulators in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D4.2b, December 2016.

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

[PLM16] Adrian Pop, Florian Lapschies, and Oliver Möller. Test automation module in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D5.2a, December 2016.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *18th Symposium on the Foundations of Computer Science*, pages 46–57. ACM, November 1977.

[Ver13] Verified Systems International GmbH. RTT-MBT Model-Based Test Generator - RTT-MBT Version 9.0-1.0.0 User Manual. Technical Report Verified-INT-003-2012, Verified Systems International GmbH, 2013. Available on request from Verified System International GmbH.

[Ver15a] Verified Systems International GmbH, Bremen, Germany. *RT-Tester 6.0: User Manual*, 2015. `https://www.verified.de/products/rt-tester/`, Doc. Id. Verified-INT-014-2003.

[Ver15b] Verified Systems International GmbH, Bremen, Germany. *RT-Tester Model-Based Test Case and Test Data Generator – RTT-MBT: User Manual*, 2015. `https://www.verified.de/products/model-based-testing/`, Doc. Id. Verified-INT-003-2012.

[Win16] Wine community. https://www.winehq.org/, November 2016. Wine website.

A List of Acronyms

20-sim	Software package for modelling and simulation of dynamic systems
API	Application Programming Interface
AST	Abstract Syntax Tree
AU	Aarhus University
BCS	Basic Control States
CLE	ClearSy
CLP	Controllab Products B.V.
COE	Co-simulation Orchestration Engine
CORBA	Common Object Request Broker Architecture
CPS	Cyber-Physical Systems
CT	Continuous-Time
DE	Discrete Event
DESTECS	Design Support and Tooling for Embedded Control Software
DSE	Design Space Exploration
FMI	Functional Mockup Interface
FMI-Co	Functional Mockup Interface – for Co-simulation
FMI-ME	Functional Mockup Interface – Model Exchange
FMU	Functional Mockup Unit
HiL	Hardware-in-the-Loop
HMI	Human Machine Interface
HW	Hardware
ICT	Information Communication Technology
IDE	Integrated Design Environment
LTL	Linear Temporal Logic
M&S	Modelling and Simulation
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MBD	Model-based Design
MBT	Model-based Testing
MC/DC	Modified Decision/Condition Coverage
MDE	Model Driven Engineering
MiL	Model-in-the-Loop
MIWG	Model Interchange Working Group
OMG	Object Management Group
OS	Operating System
PID	Proportional Integral Derivative
PROV-N	The Provenance Notation
RPC	Remote Procedure Call
RTT	Real-Time Tester

SiL	Software-in-the Loop
SMT	Satisfiability Modulo Theories
ST	Softeam
SUT	System Under Test
SVN	Subversion
SysML	Systems Modelling Language
TA	Test Automation
TE	Test Environment
TR	TRansitions
TRL	Technology Readiness Level
TWT	TWT GmbH Science & Innovation
UML	Unified Modelling Language
UNEW	University of Newcastle upon Tyne
UTP	Unifying Theories of Programming
UTRC	United Technologies Research Center
UY	University of York
VDM	Vienna Development Method
VSI	Verified Systems International
WP	Work Package
XML	Extensible Markup Language

B Background on the Individual Tools

This appendix provides background information on each of the independent tools of the INTO-CPS tool chain.

B.1 Modelio

Modelio is a comprehensive MDE [Fav05] workbench tool which supports the UML2.x standard. Modelio adds modern Eclipse-based graphical environment to the solid modelling and generation know-how obtained with the earlier Softeam MDE workbench, Objecteering, which has been on the market since 1991. Modelio provides a central repository for the local model, which allows various languages (UML profiles) to be combined in the same model, abstraction layers to be managed and traceability between different model elements to be established. Modelio makes use of extension modules, enabling the customization of this MDE environment for different purposes and stakeholders. The XMI module allows models to be exchanged between different UML modelling tools. Modelio supports the most popular XMI UML2 flavors, namely EMF UML2 and OMG UML 2.3. Modelio is one of the leaders in the OMG Model Interchange Working Group (MIWG), due to continuous work on XMI exchange improvements.

Among the extension modules, some are dedicated to IT system architects. For system engineering, SysML or MARTE modules can be used. They provide dedicated modelling support for dealing with general, software and hardware aspects of embedded or cyber physical systems. In addition, several utility modules are available, such as the Document Publisher which provides comprehensive support for the generation of different types of document.

Modelio is highly extendable and can be used as a platform for building new MDE features. The tool enables users to build UML2 Profiles, and to combine them with a rich graphical interface for dedicated diagrams, model element property editors and action command controls. Users can use several extension mechanisms: light Python scripts or a rich Java API, both of which provide access to Modelio‘s model repository and graphical interface.

B.2 Overture

The Overture platform [LBF+10] is an Eclipse-based integrated development environment (IDE) for the development and validation of system specifications in three dialects of the specification language of the Vienna Development Method. Overture is distributed with a suite of examples and step-by-step tutorials which demonstrate the features of the three dialects. A user manual for the platform itself is also provided [LLJ+13], which is accessible through Overture's help system. Although certain features of Overture are relevant only to the development of software systems, VDM itself can be used for the specification and validation of any system with distinct states, known as *discrete-event systems*, such as physical plants, protocols, controllers (both mechanical and software) *etc.*, and Overture can be used to aid in validation activities in each case.

Overture supports the following activities:

- The definition and elaboration of syntactically correct specifications in any of the three dialects, via automatic syntax and type validation.
- The inspection and assay of automatically generated proof obligations which ensure correctness in those aspects of specification validation which can not be automated.
- Direct interaction with a specification via an execution engine which can be used on those elements of the specification written in an executable subset of the language.
- Automated testing of specifications via a custom test suite definition language and execution engine.
- Visualization of test coverage information gathered from automated testing.
- Visualization of timing behaviours for specifications incorporating timing information.
- Translation to/from UML system representations.
- For specifications written in the special executable subset of the language, obtaining Java implementations of the specified system automatically.

For more information and tutorials, please refer to the documentation distributed with Overture.

The following is a brief introduction to the features of the three dialects of the VDM specification language.

VDM-SL This is the foundation of the other two dialects. It supports the development of monolithic state-based specifications with state transition operations. Central to a VDM-SL specification is a definition of the state of the system under development. The meaning of the system and how it operates is conveyed by means of changes to the state. The nature of the changes is captured by state-modifying operations. These may make use of auxiliary functions which do not modify state. The language has the usual provisions for arithmetic, new dependent types, invariants, pre- and post-conditions *etc*. Examples can be found in the VDM-SL tutorials distributed with Overture.

VDM++ The VDM++ dialect supports a specification style inspired by object-oriented programming. In this specification paradigm, a system is understood as being composed of entities which encapsulate both state and behaviour, and which interact with each other. Entities are defined via templates known as *classes*. A complete system is defined by specifying *instances* of the various classes. The instances are independent of each other, and they may or may not interact with other instances. As in object-oriented programming, the ability of one component to act directly on any other is specified in the corresponding class as a state element. Interaction is naturally carried out via precisely defined interfaces. Usually a single class is defined which represents the entire system, and it has one instance, but this is only a convention. This class may have additional state elements of its own. Whereas a system in VDM-SL has a central state which is modified throughout the lifetime of the system, the state of a VDM++ system is distributed among all of its components. Examples can be found in the VDM++ tutorials distributed with Overture.

VDM-RT VDM-RT is a small extension to VDM++ which adds two primary features:

- The ability to define how the specified system is envisioned to be allocated on a distributed execution platform, together with the communication topology.
- The ability to specify the timing behaviours of individual components, as well as whether certain behaviours are meant to be cyclical.

Finer details can be specified, such as execution synchronization and mutual exclusion on shared resources. A VDM-RT specification has the same structure as a VDM++ specification, only the conventional system class of VDM++ is mandatory in VDM-RT. Examples can be found in the VDM-RT tutorials distributed with Overture.

B.3 20-sim

20-sim [Con13, Bro97] is a commercial modelling and simulation software package for mechatronic systems. With 20-sim, models can be created graphically, similar to drawing an engineering scheme. With these models, the behaviour of dynamic systems can be analyzed and control systems can be designed. 20-sim models can be exported as C-code to be run on hardware for rapid prototyping and HiL-simulation. 20-sim includes tools that allow an engineer to create models quickly and intuitively. Models can be created using equations, block diagrams, physical components and bond graphs [KR68]. Various tools give support during the model building and simulation. Other toolboxes help to analyze models, build control systems and improve system performance. Figure 104 shows 20-sim with a model of a controlled

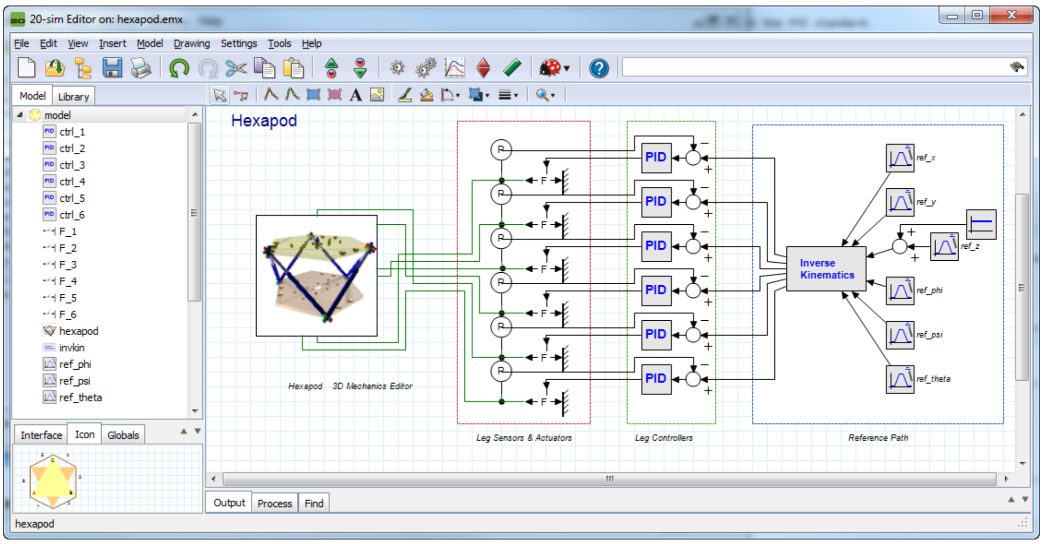


Figure 104: Example of a hexapod model in 20-sim.

hexapod. The mechanism is generated with the 3D Mechanics Toolbox and connected with standard actuator and sensor models from the mechanics library. The hexapod is controlled by PID controllers which are tuned in the frequency domain. Everything that is required to build and simulate this model and generate the controller code for the real system is included inside the package.

The 20-sim Getting Started manual [KG16] contains examples and step-by-step tutorials that demonstrate the features of 20-sim. More information on 20-sim can be found at `http://www.20sim.com` and in the user manual at `http://www.20sim.com/webhelp` [KGD16]. The integration of 20-sim into the INTO-CPS tool-chain is realized via the FMI standard.

B.4 OpenModelica

OpenModelica [Fri04] is an open-source Modelica-based modelling and simulation environment. Modelica [FE98] is an object-oriented, equation based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. The Modelica language (and OpenModelica) supports continuous, discrete and hybrid time simulations. OpenModelica already compiles Modelica models into FMU, C or C++ code for simulation. Several integration solvers, both fixed and variable step size, are available in OpenModelica: euler, rungekutta, dassl (default), radau5, radau3, radau1.

OpenModelica can be interfaced to other tools in several ways as described in the OpenModelica user's manual [Ope]:

- via command line invocation of the omc compiler
- via C API calls to the omc compiler dynamic library
- via the CORBA interface
- via OMPython interface [GFR+12]

OpenModelica has its own scripting language, Modelica script (mos files), which can be used to perform actions via the compiler API, such as loading, compilation, simulation of models or plotting of results. OpenModelica supports Windows, Linux and Mac Os X.

The integration of OpenModelica into the INTO-CPS tool chain is realized via compliance with the FMI standard, and is described in deliverable D4.1b [PBLG15].

B.5 RT-Tester

The RT-Tester [Ver15a] is a test automation tool for automatic test generation, test execution and real-time test evaluation. Key features include a strong C/C++-based test script language, high performance multi-threading, and hard real-time capability. The tool has been successfully applied in avionics, rail automation, and automotive test projects. In the INTO-CPS tool chain, RT-Tester is responsible for model-based testing, as well as for model checking. This section gives some background information on the tool from these two perspectives.

B.5.1 Model-based Testing

The RT-Tester Model Based Test Case and Test Data Generator (RTT-MBT) [Ver15b] supports model-based testing (MBT), that is, automated generation of test cases, test data, and test procedures from UML/SysML models. A number of common modelling tools can be used as front-ends for this. The most important technical challenge in model-based test automation is the extraction of test cases from test models. RTT-MBT combines an SMT solver with a technique akin to bounded model checking so as to extract finite paths through the test model according to some predefined criterion. This criterion can, for instance, be MC/DC coverage, or it can be requirements coverage (if the requirements are specified as temporal logic formulae within the model). A further aspect is that the environment can be modelled within the test model. For example, the test model may contain a constraint such that a certain input to the system-under-test remains in a predefined range. This aspect becomes important once test automation is lifted from single test models to multi-model cyber-physical systems. The derived test procedures use the RT-Tester Core as a back-end, allowing the system under test to be provided on real hardware, software only, or even just simulation to aid test model development.

Further, RTT-MBT includes requirement tracing from test models down to test executions and allows for powerful status reporting in large scale testing projects.

B.5.2 Model Checking of Timed State Charts

RTT-MBT applies model checking to behavioural models that are specified as timed state charts in UML and SysML, respectively. From these models, a transition relation is extracted and represented as an SMT formula in bit-vector theory [KS08], which is then checked against LTL formulae [Pnu77] using the algorithm of Biere *et al.* [BHJ+06]. The standard setting of RTT-MBT is to apply model checking to a single test model, which consists of the system specification and an environment.

- A component called *TestModel* that is annotated with stereotype *TE*.
- A component called *SystemUnderTest* that is annotated with stereotype *SUT*.

RTT-MBT uses the stereotypes to infer the role of each component. The interaction between these two parts is implemented via input and output interfaces that specify the accessibility of variables using UML stereotypes.

- A variable that is annotated with stereotype *SUT2TE* is written by the system model and readable by the environment.
- A variable that is annotated with stereotype *TE2SUT* is written by the environment and read by the system model as an input.

A simple example is depicted in Figure 105, which shows a simple composite structure diagram in Modelio for a turn indication system. The purpose of the system is to control the lamps of a turn indication system in a car. Further details are given in [Ver13]. The test model consists of the two aforementioned components and two interfaces:

- **Interface1** is annotated with stereotype *TE2SUT* and contains three variables `voltage`, `TurnIndLvr` and `EmerSwitch`. These variables are controlled by the environment and fed to the system under test as inputs.
- **Interface2** is annotated with stereotype *SUT2TE* and contains two variables `LampsLeft` and `LampsRight`. These variables are controlled by the system under test and can be read by the environment.

Observe that the two variables `LampsLeft` and `LampsRight` have type `int`, but should only hold values `0` or `1` to indicate states *on* or *off*. A straightforward system property that could be verified would thus be that `LampsLeft` and `LampsRight` indeed are only assigned `0` or `1`, which could be expressed by the following LTL specification:

$$\mathbf{G}(0 \leq \texttt{LampsLeft} \leq 1 \wedge 0 \leq \texttt{LampsRight} \leq 1)$$

A thorough introduction with more details is given in the RTT-MBT user manual [Ver13].

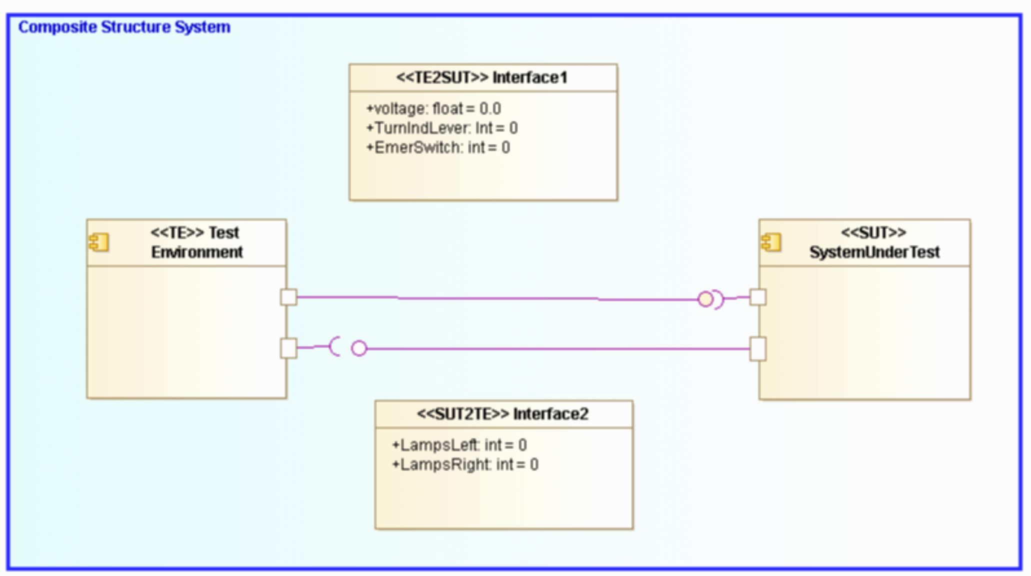


Figure 105: Simple model that highlights interfaces between the environment and the system-under-test.

C Underlying Principles

The INTO-CPS tool chain facilitates the design and validation of CPSs through its implementation of results from a number of underlying principles. These principles are co-simulation, design space exploration, model-based test automation and code generation. This appendix provides an introduction to these concepts.

C.1 Co-simulation

Co-simulation refers to the simultaneous simulation of individual models which together make up a larger system of interest, for the purpose of obtaining a simulation of the larger system. A co-simulation is performed by a co-simulation orchestration engine. This engine is responsible for initializing the individual simulations as needed; for selecting correct time step sizes such that each constituent model can be simulated successfully for that duration, thus preventing time drift between the constituent simulations; for asking each individual simulation to perform a simulation step; and for synchronizing information between models as needed after each step. The result of one such round of simulations is a single simulation step for the complete multi-model of the system of interest.

As an example, consider a very abstract model of a nuclear power plant. This consists of a nuclear reactor core, a controller for the reactor, a water and steam distribution system, a steam-driven turbine and a standard electrical generator. All these individual components can be modelled separately and simulated, but when composed into a model of a nuclear power plant, the outputs of some become the inputs of others. In a co-simulation, outputs are matched to inputs and each component is simulated one step at a time in such a way that when each model has performed its simulation step, the overall result is a simulation step of the complete power plant model. Once the correct information is exchanged between the constituent models, the process repeats.

C.2 Design Space Exploration

During the process of developing a CPS, either starting from a completely blank canvas or constructing a new system from models of existing components, the architects will encounter many design decisions that shape the final product. The activity of investigating and gathering data about the merits of the different choices available is termed Design Space Exploration. Some of the choices the designer will face could be described as being the selection of parameters for specific components of the design, such as the exact position of a sensor, the diameter of wheels or the parameters affecting a control algorithm. Such parameters are variable to some degree and the selection of their value will affect the values of objectives by which a design will be measured. In these cases it is desirable to explore the different values each parameter may take and also different combinations of these parameter values if there are more than one parameter, to find a set of designs that best meets its objectives. However, since the size of the design space is the product of the number of parameters and the number of values each may adopt, it is often impractical to consider performing simulations of all parameter combinations or to manually assess each design.

The purpose of an automated DSE tool is to help manage the exploration of the design space, and it separates this problem into three distinct parts: the search algorithm, obtaining objective values and ranking the designs according to those objectives. The simplest of all search algorithms is the exhaustive search, and this algorithm will methodically move through each design, performing a simulation using each and every one. This is termed an open loop method, as the simulation results are not considered by the algorithm at all. Other algorithms, such as a genetic search, where an initial set of randomly generated individuals are bred to produce increasingly good results, are closed loop methods. This means that the choice of next design to be simulated is driven by the results of previous simulations.

Once a simulation has been performed, there are two steps required to close the loop. The first is to analyze the raw results output by the simulation to determine the value for each of the objectives by which the simulations are to be judged. Such objective values could simply be the maximum power consumed by a component or the total distance traveled by an object, but they could also be more complex measures, such as the proportion of time a device was operating in the correct mode given some conditions. As well as numerical objectives, there can also be constraints on the system that are either passed or failed. Such constraints could be numeric, such as the maximum power that a substation must never exceed, or they could be based on temporal logic to check that undesirable events do not occur, such as all the lights at a road junction not being green at the same time.

The final step in a closed loop is to rank the designs according to how well each performs. The ranking may be trivial, such as in a search for a design that minimizes the total amount of energy used, or it may be more complex if there are multiple objectives to optimize and trade off. Such ranking functions can take the form of an equation that returns a score for each design, where the designs with the highest/lowest scores are considered the best. Alternatively, if the relationship between the desired objectives is not well understood, then a Pareto approach can be taken to ranking, where designs are allocated to ranks of designs that are indistinguishable from each other, in that each represents an optimum, but there exist different tradeoffs between the objective values.

C.3 Model-Based Test Automation

The core fragment of test automation activities is a model of the desired system behaviour, which can be expressed in SysML. This test model induces a transition relation, which describes a collection of execution paths through the system, where a path is considered a sequence of timed data vectors (containing internal data, inputs and outputs). The purpose of a test automation tool is to extract a subset of these paths from the test model and turn these paths into test cases, respectively test procedures. The test procedures then compare the behaviour of the actual system-under-test to the path, and produce warnings once discrepancies are observed.

C.4 Code Generation

Code generation refers to the translation of a modelling language to a common programming language. Code generation is commonly employed in control engineering, where a controller is modelled and validated using a tool such as 20-sim, and finally translated into source code to be compiled for some embedded execution platform, which is its final destination.

The relationship that must be maintained between the source model and translated program must be one of refinement, in the sense that the translated program must not do anything that is not captured by the original model. This must be considered when translating models written in high-level specification languages, such as VDM. The purpose of such languages is to allow the specification of several equivalent implementations. When a model written in such a language is translated to code, one such implementation is essentially chosen. In the process, any non-determinism in the specification, the specification technique that allows a choice of implementations, must be resolved. Usually this choice is made very simple by restricting the modelling language to an executable subset, such that no such non-determinism is allowed in the model. This restricts the choice of implementations to very few, often one, which is the one into which the model is translated via code generation.