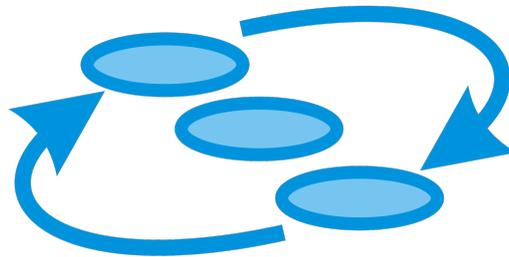




Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

INTO-CPS Tool Chain User Manual

Deliverable Number: D4.1a

Version: 1.00

Date: 2015

Public Document

<http://into-cps.au.dk>

Contributors:

Victor Bandur, AU
Peter Gorm Larsen, AU
Kenneth Lausdahl, AU
Sune Wolf, AU
Carl Gamble, UNEW
Adrian Pop, LIU
Etienne Brosse, ST
Jörg Brauer, VSI
Florian Lapschies, VSI
Marcel Groothuis, CLP
Christian Kleijn, CLP

Editors:

Victor Bandur, AU

Reviewers:

Ken Pierce, UNEW
Nuno Amálio, UY
Hassan Ridouane, UTRC

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	21-05-2015	Peter Gorm Larsen	Full structure of the deliverables and responsibilities.
0.02	01-10-2015	Adrian Pop	Add OpenModelica related text.
0.03	09-10-2015	Carl Gamble	Added introduction to DSE.
0.04	15-10-2015	Victor Bandur	Restructured so that discussion of COE is now a subsection of “The INTO-CPS Platform”.
0.05	18-10-2015	Victor Bandur	Completed section on obtaining the individual components.
0.06	23-10-2015	Jörg Brauer	Wrote introduction to test automation.
0.07	25-10-2015	Victor Bandur	Completed section on the Overture component.
0.08	26-10-2015	Jörg Brauer	Further contributions to sections on test automation.
0.09	27-10-2015	Etienne Brosse	Draft of section on using Modelio.
0.10	28-10-2015	Victor Bandur	Proof-reading.
0.11	28-10-2015	Victor Bandur	Draft of section on code generation.
0.12	03-11-2015	Victor Bandur	Reconstructed document history from Subversion log information.
0.13	03-11-2015	Marcel Groothuis	Add 20-sim related text.
0.14	06-11-2015	Carl Gamble	DSE v1.0 manual text and images added.
0.15	10-11-2015	Adrian Pop	Section on FMU export for OpenModelica
0.16	10-11-2015	Etienne Brosse	Section on the INTO-CPS Application and Modelio
0.17	10-11-2015	Victor Bandur	Final version for internal review.
0.18	7-12-2015	Carl Gamble	Internal review comments on DSE addressed.
1.00	17-12-2015	Victor Bandur	Final corrections.

Abstract

This deliverable is the user manual for the INTO-CPS tool chain. It is targeted at those who want to make use of these tools 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 D4.1b [PBLG15], D4.1c [BQS15], D4.1d [LLW⁺15], D5.1a [GHJ⁺15], D5.1b [MPB15], D5.1c [BF15] and D5.1d [HLG⁺15].

Contents

1	Introduction	6
2	Overview of the INTO-CPS Tool Chain	7
3	Modelio and SysML for INTO-CPS	8
4	The INTO-CPS Application	20
4.1	The Co-Simulation Orchestration Engine	23
5	Using the Separate Modelling and Simulation Tools	24
5.1	Overture	25
5.2	20-sim	29
5.3	OpenModelica	31
5.4	RT-Tester / RTT-MBT	35
6	Design Space Exploration for INTO-CPS	39
7	Test Automation for INTO-CPS	43
8	Code Generation for INTO-CPS	44
9	Conclusions	45
A	List of Acronyms	49
B	Background on the Individual Tools	51
B.1	Modelio	51
B.2	Overture	52
B.3	20-sim	54
B.4	OpenModelica	55
B.5	RT-Tester	56
C	Obtaining the Individual Tools	59
D	Underlying Principles	63
D.1	Co-simulation	63
D.2	Design Space Exploration	63
D.3	Model-Based Test Automation	65
D.4	Code Generation	65

1 Introduction

This deliverable is the user manual for the INTO-CPS tool chain. This tool chain is meant to support a model-based development approach for Cyber-Physical Systems (CPSs). The analysis is primarily based on simulation of heterogeneous models making use of the Functional-Mockup Interface (FMI) standard [Blo14] using co-simulation. Other verification features supported by the tool chain include hardware- and software-in-the-loop simulation and model-based testing. Verification by model checking is planned for the second year of the project.

The release process of the complete tool chain is managed at

`http://overture.au.dk/into-cps/site`

under the category “Download”. In case access to the individual tools is required, pointers are also provided there and in Appendix C.

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 technical content of the 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 submodels 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 in the INTO-CPS context.
- Section 8 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 document.
- 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. The constituent tools are dedicated to the different phases of a co-simulation activity. They are discussed individually through the course of this manual. An overview of the tool chain is shown in Figure 1.

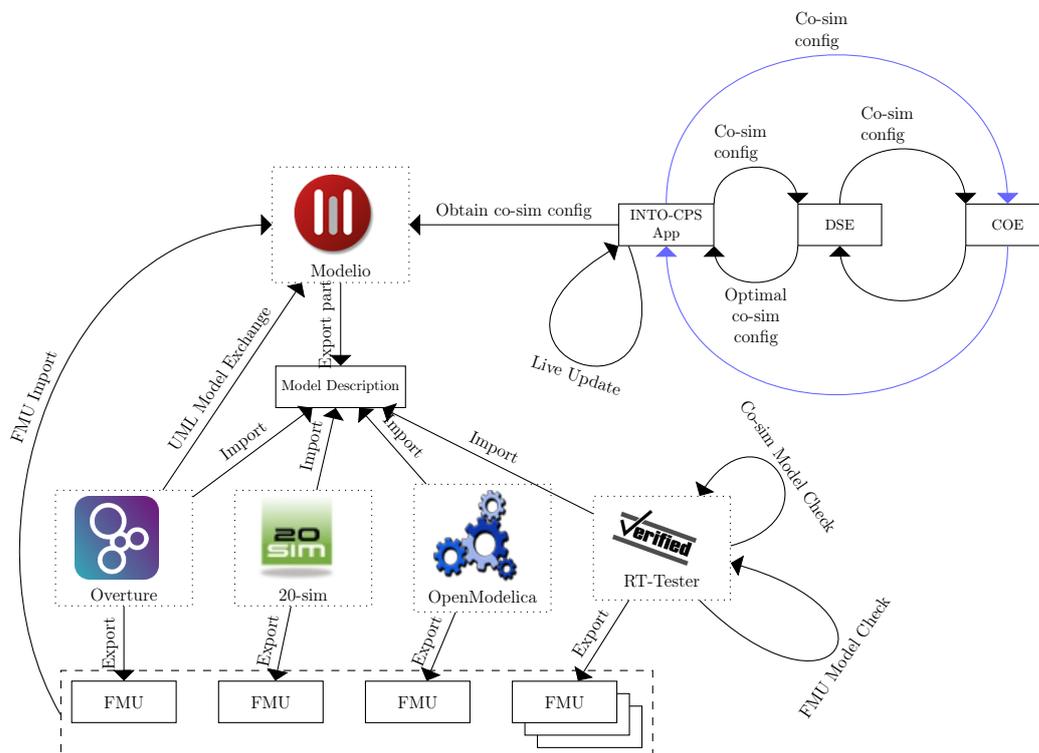


Figure 1: Overview of the structure of the INTO-CPS tool chain.

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 [APCB15]. Here one can either design a multi-model from scratch by specifying the characteristics and connection topology of FMUs yet to be developed, or import existing FMUs so that the connections between them can 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 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 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, 20-sim and OpenModelica. 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 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 which 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 which 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 fulfil 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 System, composed of two *EComponent* of kind *Subsystem*, named SubSystem. This Subsystem have an internal *Variable* called variable of type *String* and exposes two *FlowPorts* named portIn and portOut. The type of data going through these ports are respectively defined by In and Out *StrtType* type. More details on the SysML/INTO-CPS profile can be found in deliv-

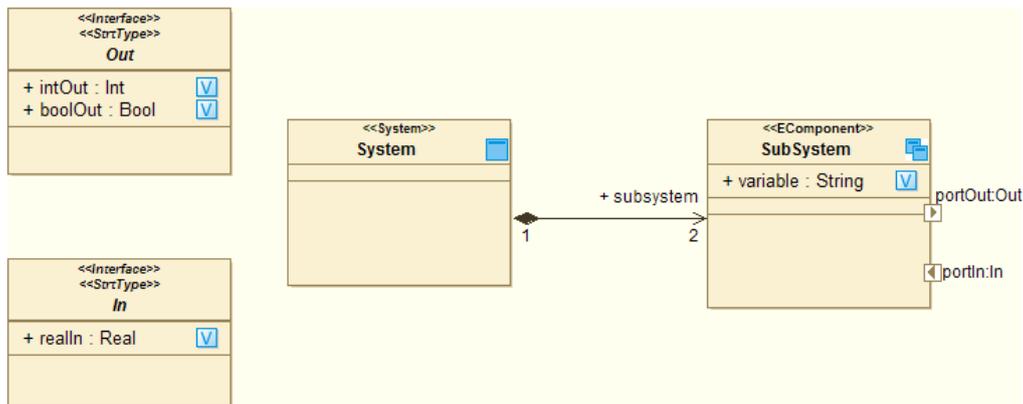


Figure 2: Example INTO-CPS multi-model.

erable D2.1a [APCB15].

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 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.

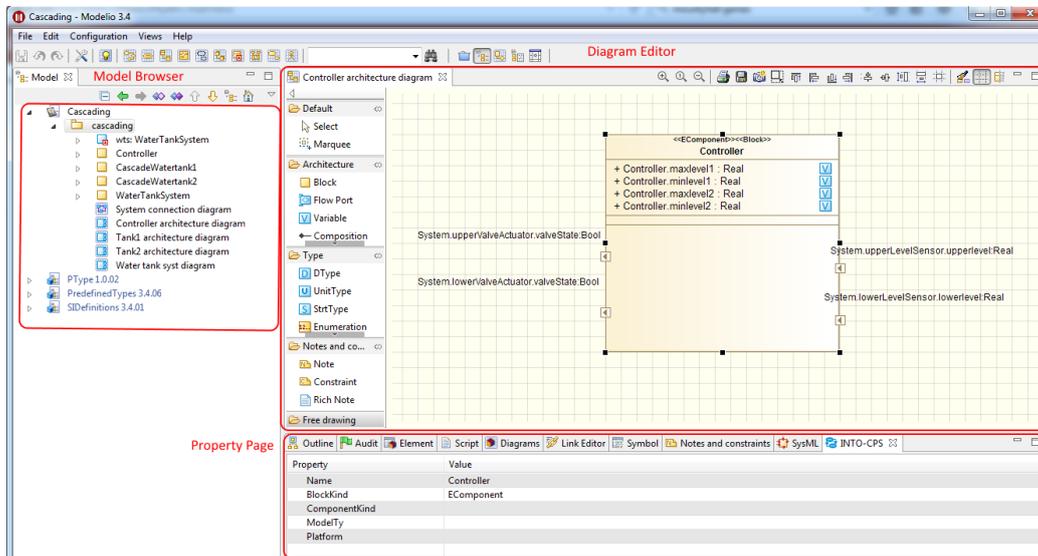


Figure 3: Modelio for INTO-CPS.

In the INTO-CPS Modelling workflow [FGPP15], the first step will be to create, as depicted diagram in Figure 4, a Modelio project for this:

1. Launch Modelio.
2. Click on *File* → *Create a 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 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 in Appendix C. 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 empty *Architecture Structure Diagram*.

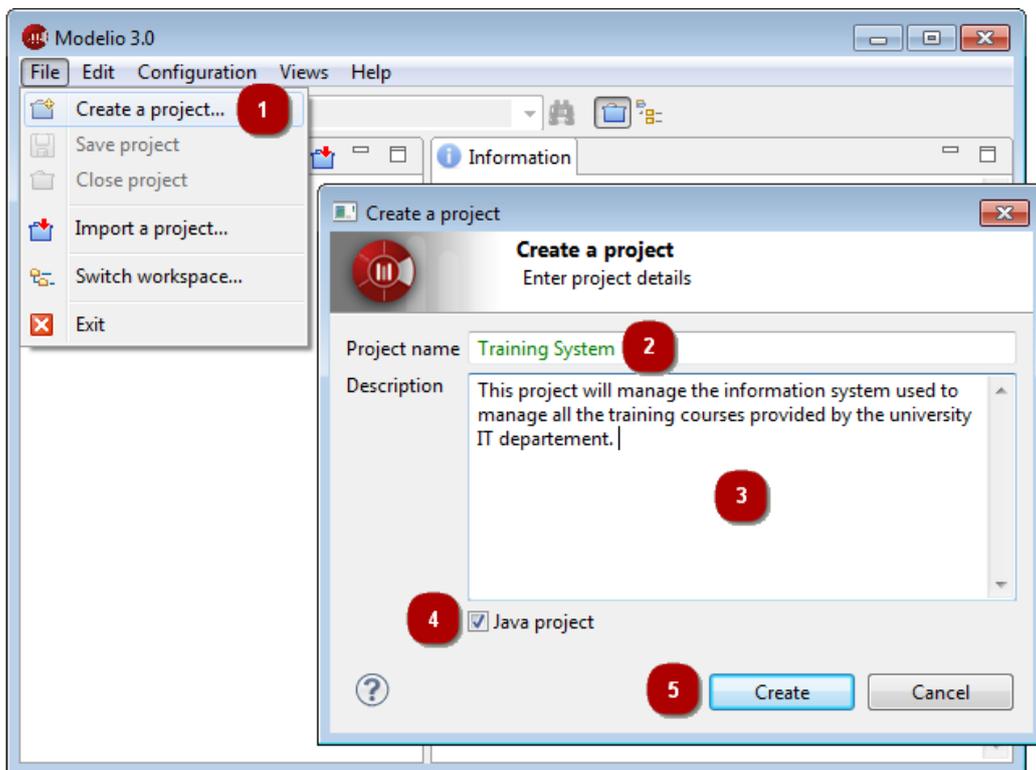


Figure 4: Creating a new Modelio project.

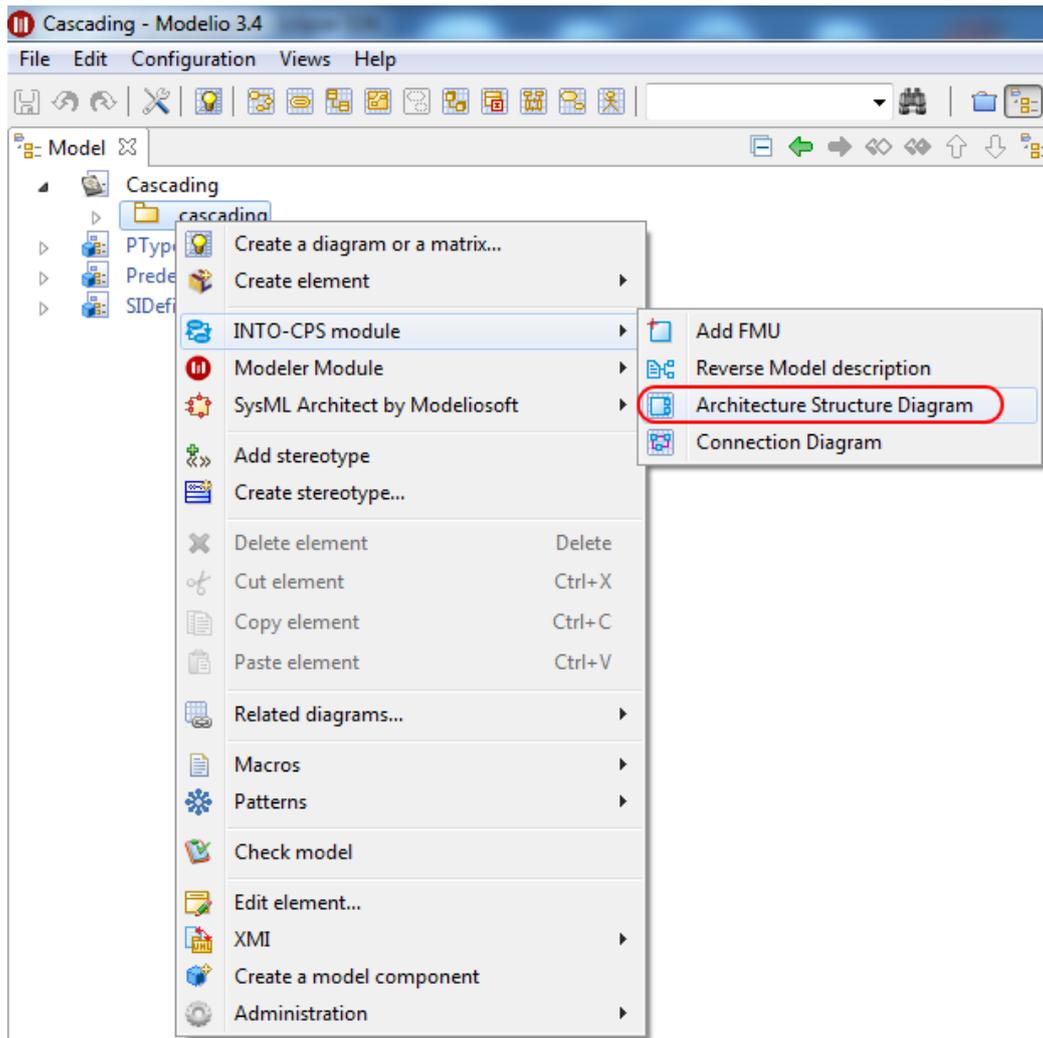


Figure 5: Creating an Architecture Structure diagram.

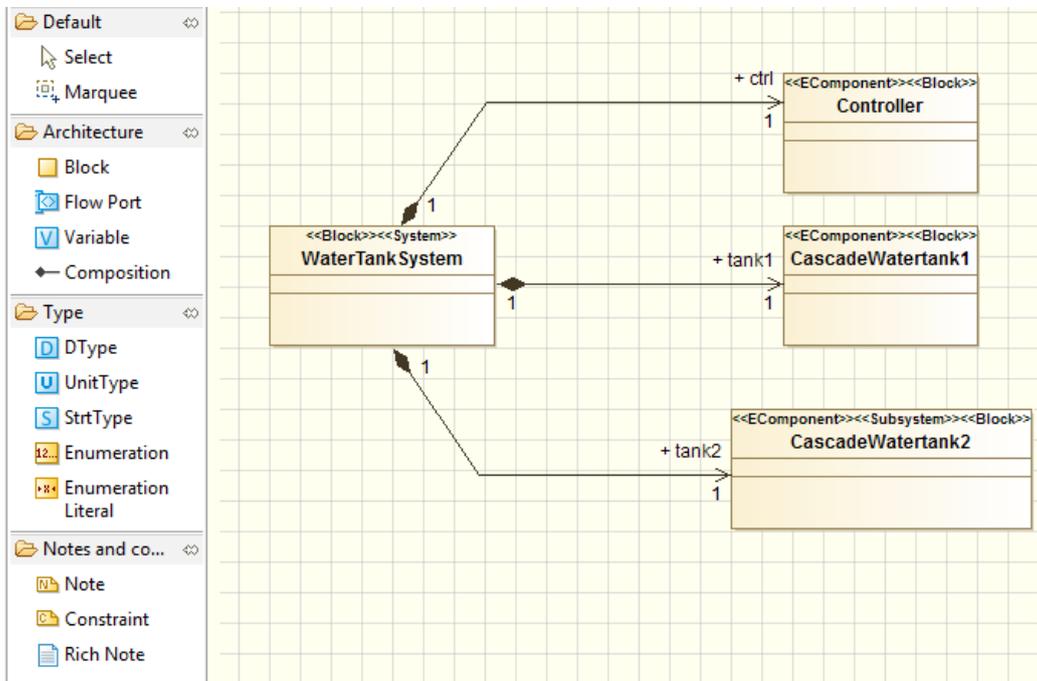


Figure 6: Architecture Structure diagram.

Instead of creating an Architecture 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, 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.

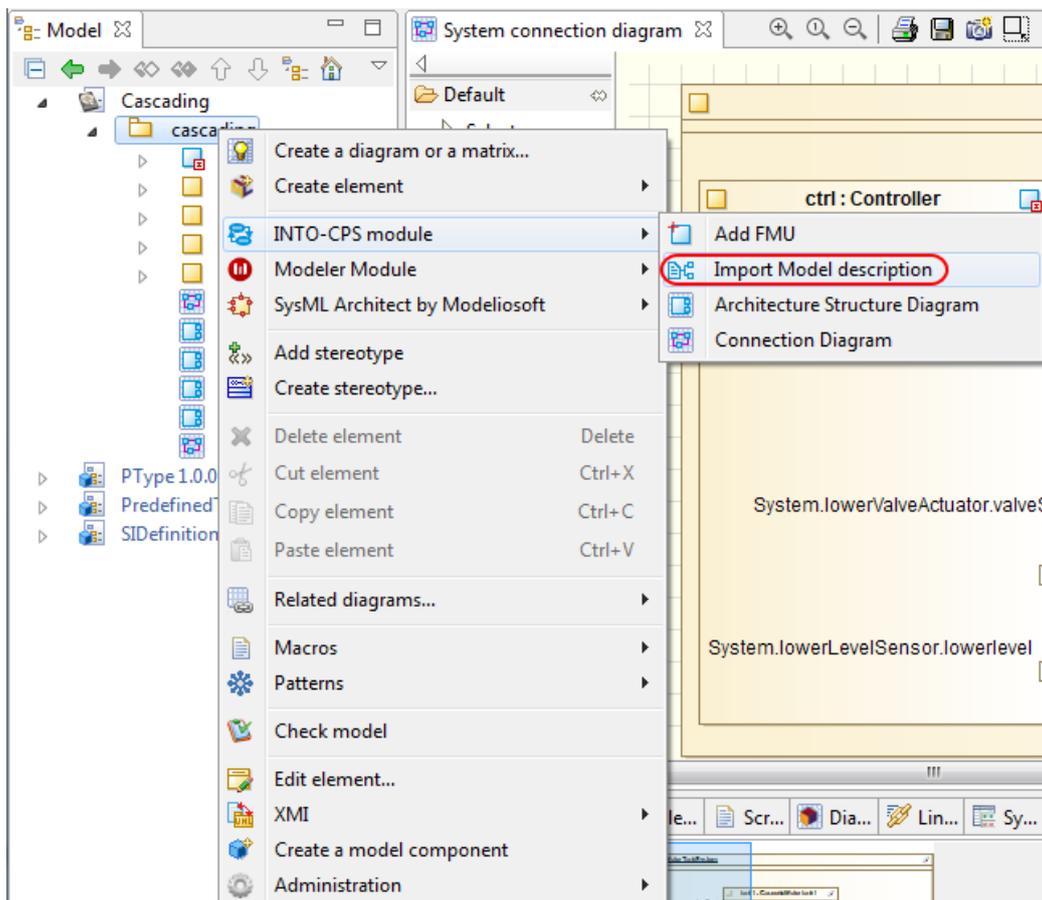


Figure 7: Importing an existing model description.

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



Figure 8: Model description selection.

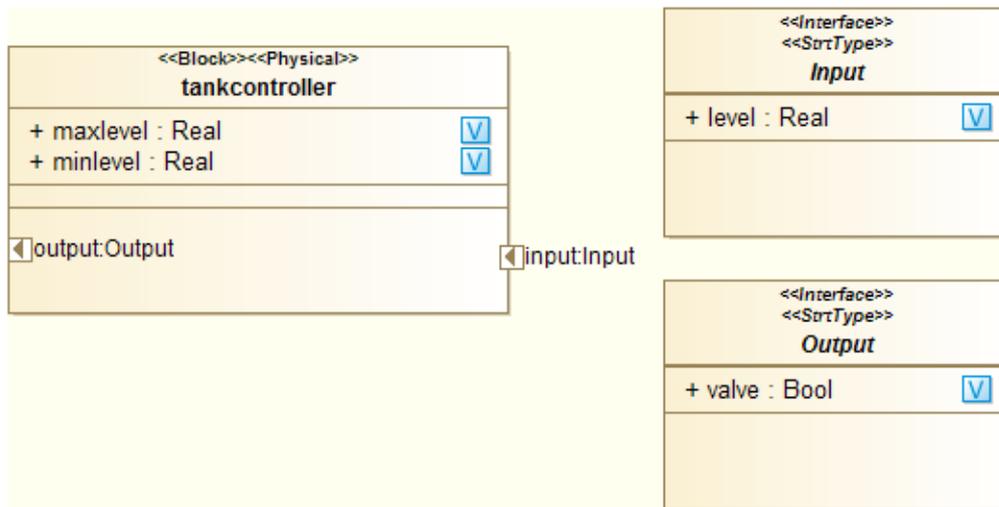


Figure 9: Result of Model description import.

of them in an *INTO-CPS Connection diagram*. To create an *INTO-CPS Connection diagram*, as for an *INTO-CPS Architecture diagram*, right click on a *Package* element, then in the *INTO-CPS* entry choose *Connection Diagram*, as shown in Figure 10. Figure 11 shows the result of creating such a diagram.

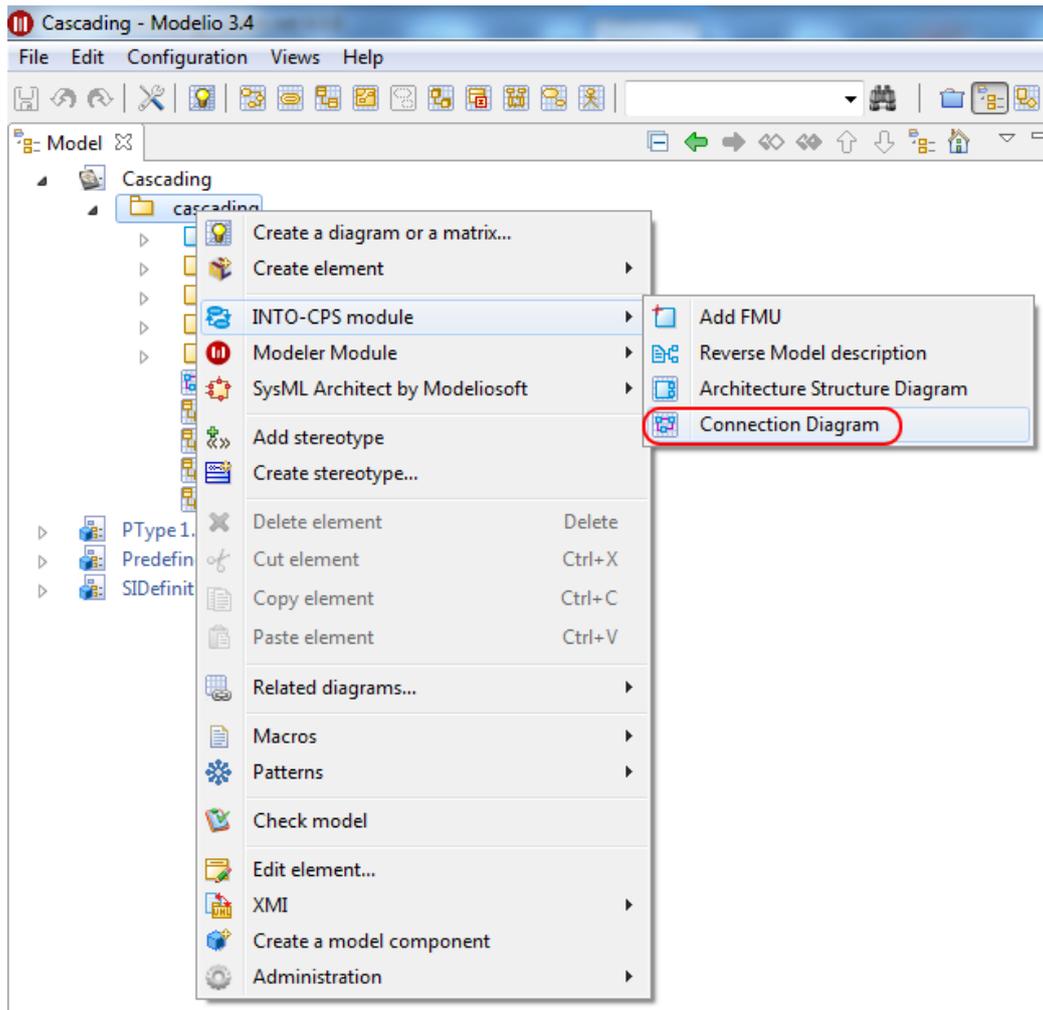


Figure 10: Creating a Connection diagram.

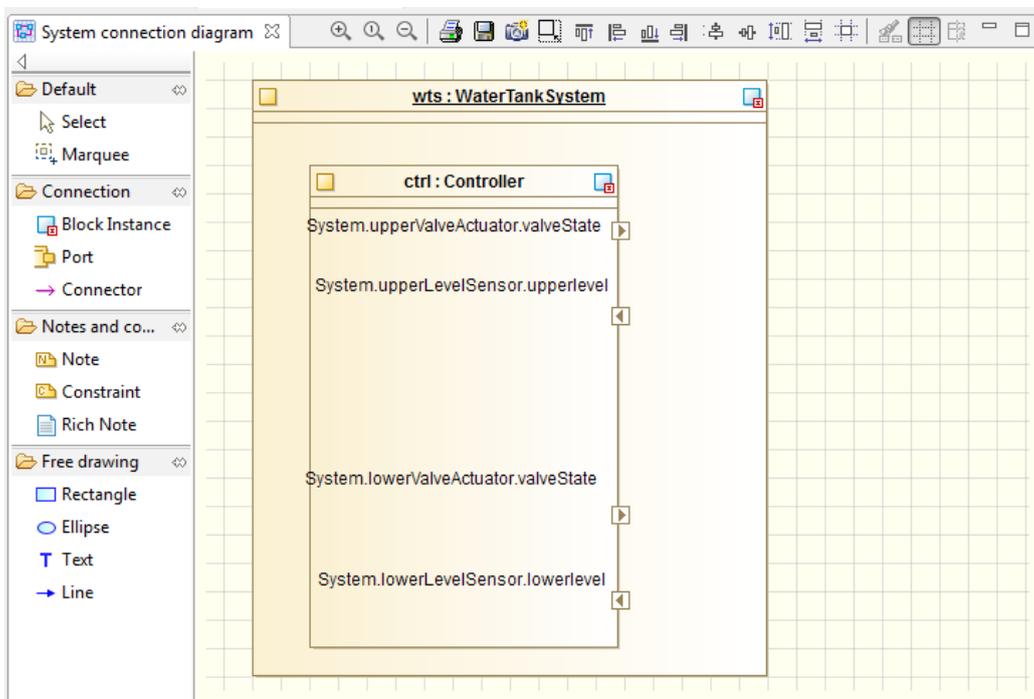


Figure 11: Example of connection diagram.

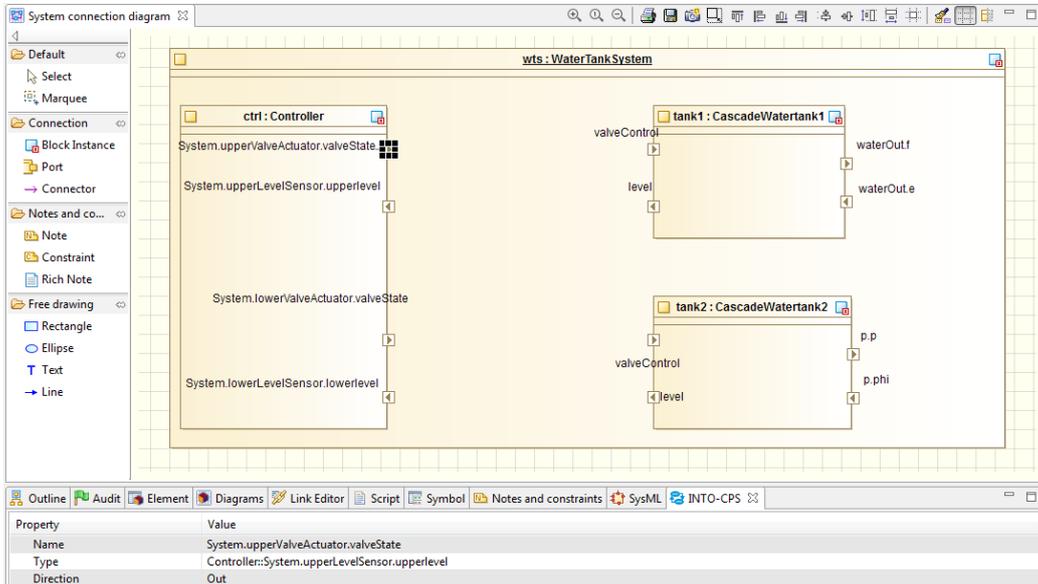


Figure 12: Connection diagram example.

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 (Figure 13).

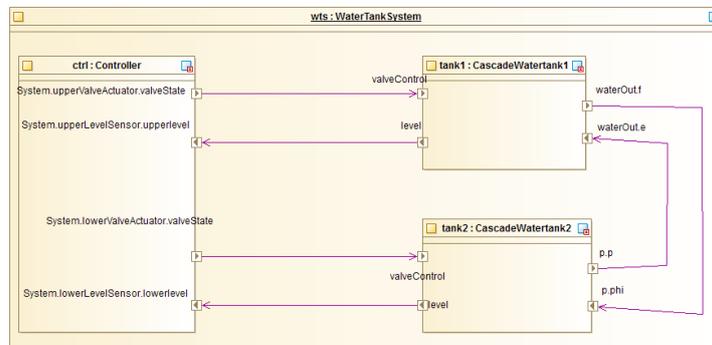


Figure 13: Connection diagram with connection.

Before simulating your model, you have to *associate* these block instances with dedicated FMUs. This is done in two steps:

1. Add an FMU to the project.
2. Associate each block instance to one of these FMUs.

To add an FMU, once again right click on a *Package* element, then in the *INTO-CPS* entry choose *Add FMU*, as shown in Figure 14. Select your FMU

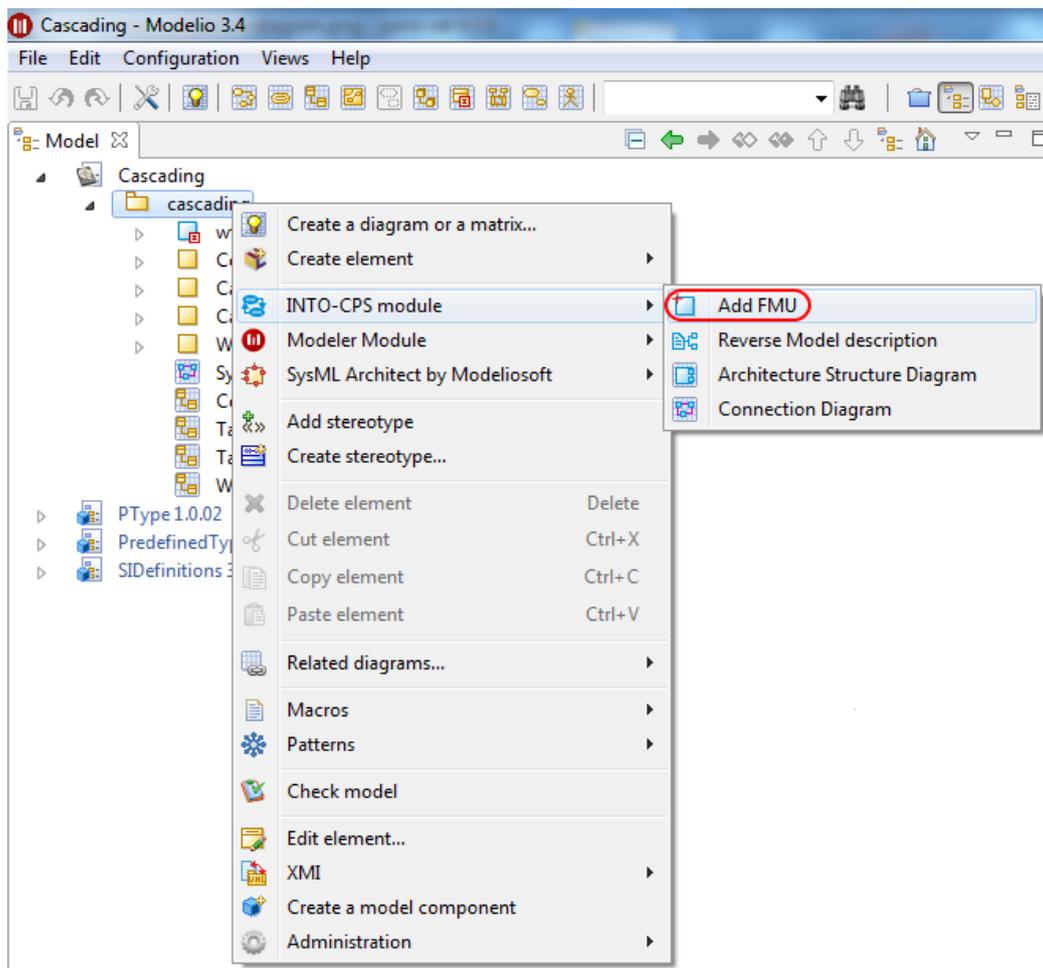


Figure 14: Add a FMU to the project.

and click on *Add*, cf. Figure 15. Then, in the INTO-CPS property page of each block Instance, Figure 16, you will be able to choose which FMU is related to that instance.

At this point your blocks have been defined, the connections have been set

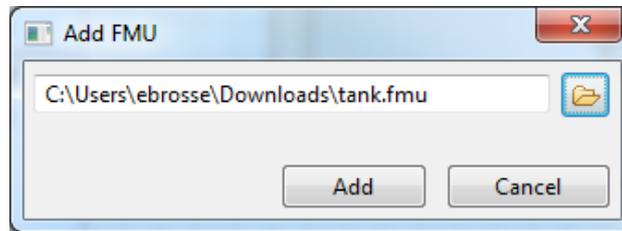


Figure 15: Select the FMU.

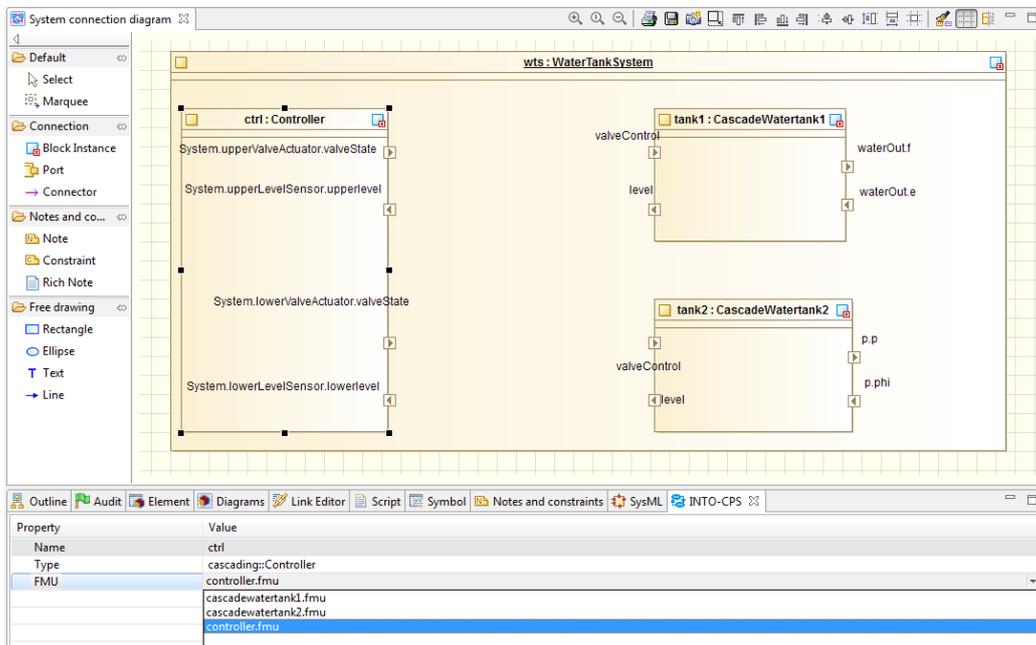


Figure 16: Associate Model and FMU.

and FMUs have been associated. The next step would be to simulate your model. 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 17. Choose a relevant name (Figure 18) and click on *Generate*.

4 The INTO-CPS Application

Built on top of the Modelio tool (Section 3), the INTO-CPS Application is the front-end of the INTO-CPS tool chain. Automatically deployed with the INTO-CPS modelling environment, Figure 19 shows how it looks. Taking

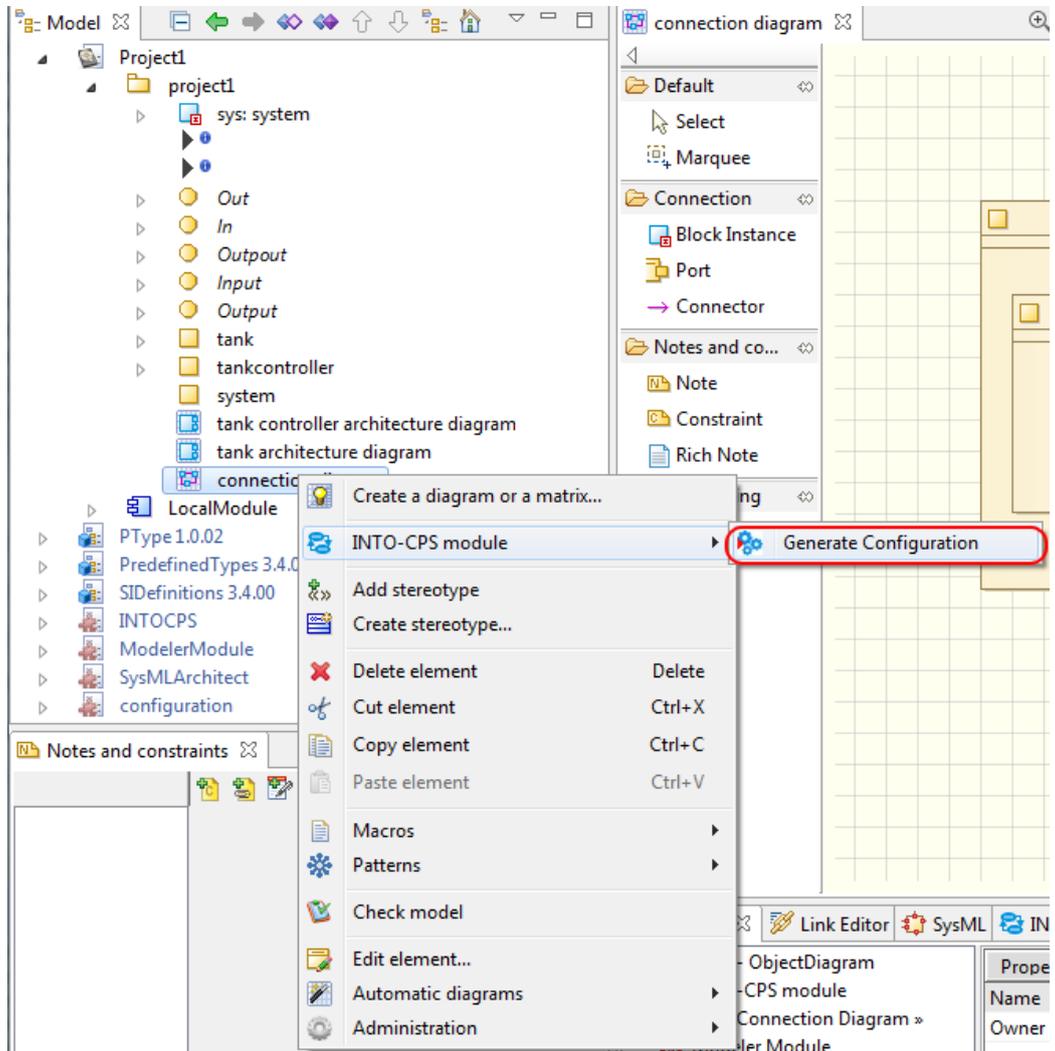


Figure 17: Example INTO-CPS multi-model.

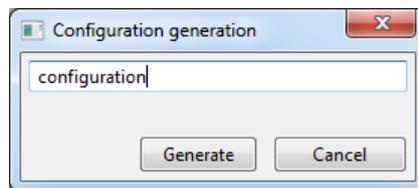


Figure 18: Example INTO-CPS multi-model.

as input a configuration generated from an INTO-CPS Connection diagram such as that in Figure 11, the INTO-CPS Application allows you to complete, thanks to a specific editor, the specification of the co-simulation orchestrated

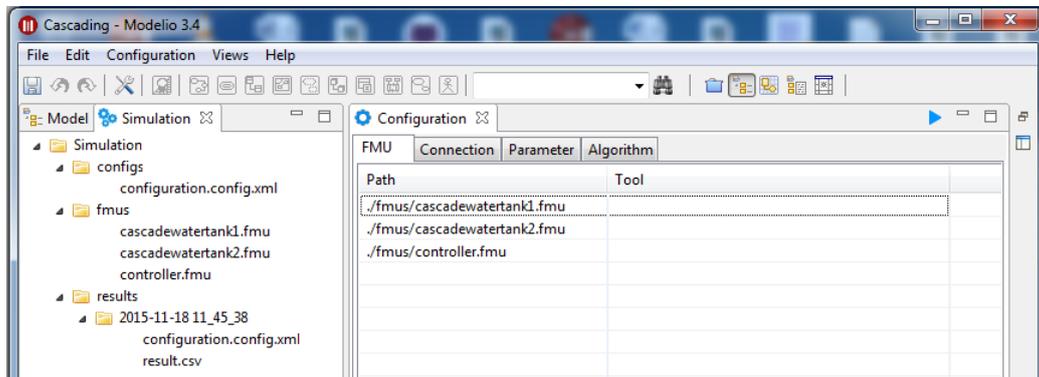


Figure 19: The INTO-CPS Application.

by the COE (Section 4.1). This editor is composed of four tabs:

1. FMU Tab, listing the FMUs used for the simulation.
2. Connection Tab, showing the specified connections.
3. Parameter Tab, where parameter values are initialized.
4. Algorithm Tab, for specifying the algorithm type and its different values.

Note that the FMU and Connection tabs (Figure 20 and Figure 21, respectively) can not be modified from this editor. The values specified here come from an INTO-CPS multi-model created as discussed above. The Param-

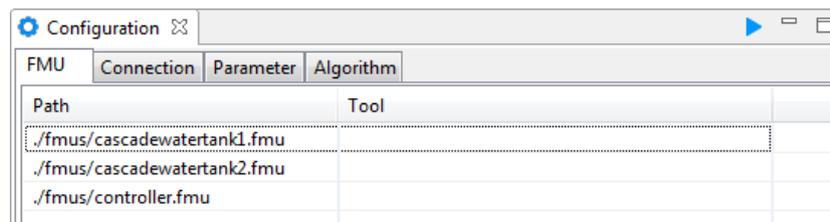
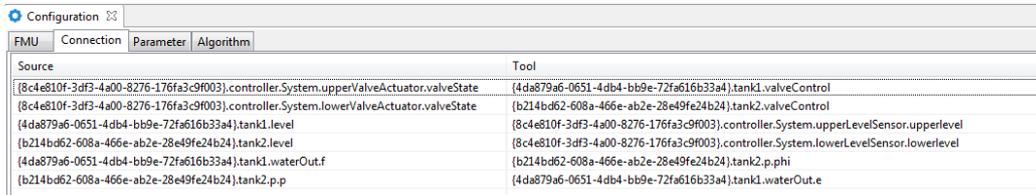


Figure 20: FMUs configuration tab.

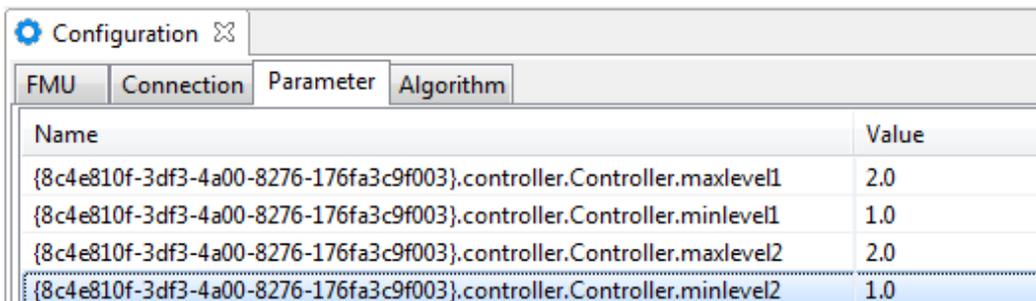
eters tab (Figure 22) lists all the parameters coming from the model, making it possible to specify a *Value* for each. Finally, the Algorithm tab (Figure 23) lists the parameters used for the simulation, including the Algorithm type (*Fixed* or *Variable*), the time step, the starting time and the ending time.

Once the co-simulation configuration has been fully specified, it is possible to run the co-simulation. At the end of the simulation activity, a folder will be



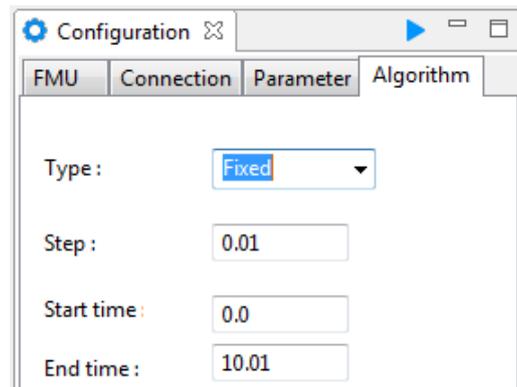
Source	Tool
{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.System.upperValveActuator.valveState	{4da879a6-0651-4db4-bb9e-72fa616b33a4}.tank1.valveControl
{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.System.lowerValveActuator.valveState	{b214bd62-608a-466e-ab2e-28e49fe24b24}.tank2.valveControl
{4da879a6-0651-4db4-bb9e-72fa616b33a4}.tank1.level	{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.System.upperLevelSensor.upperlevel
{b214bd62-608a-466e-ab2e-28e49fe24b24}.tank2.level	{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.System.lowerLevelSensor.lowerlevel
{4da879a6-0651-4db4-bb9e-72fa616b33a4}.tank1.waterOut.f	{b214bd62-608a-466e-ab2e-28e49fe24b24}.tank2.p.phi
{b214bd62-608a-466e-ab2e-28e49fe24b24}.tank2.p.p	{4da879a6-0651-4db4-bb9e-72fa616b33a4}.tank1.waterOut.e

Figure 21: Connections configuration tab.



Name	Value
{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.Controller.maxlevel1	2.0
{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.Controller.minlevel1	1.0
{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.Controller.maxlevel2	2.0
{8c4e810f-3df3-4a00-8276-176fa3c9f003}.controller.Controller.minlevel2	1.0

Figure 22: Parameters initialisation tab.



Configuration 

FMU Connection Parameter Algorithm

Type:

Step:

Start time:

End time:

Figure 23: Algorithm selection tab.

created inside the *results* container (Figure 24), containing the co-simulation configuration and the results, in CSV format.

4.1 The Co-Simulation Orchestration Engine

The heart of the INTO-CPS Application is the Co-Simulation Orchestration Engine (COE). This is the engine which performs the *orchestration* of the various simulation tools (described below), carrying out their respective roles in the overall co-simulation. It is written in a combination of Java and Scala,

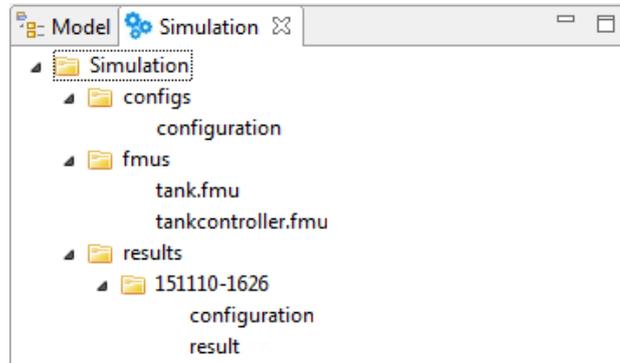


Figure 24: Simulation contents browser.

and runs as a stand-alone server hosting the co-simulation API on port 8080. It is started automatically by the INTO-CPS Application. It may be started manually for testing purposes by executing:

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

The COE responds to simple HTTP requests, which are documented in the API manual, which is also hosted by the COE. With the COE running, the API manual can be obtained by executing:

```
curl -o api.pdf http://localhost:8082/api/pdf
```

The COE is entirely hidden from the end user of the INTO-CPS Application, but parts of it are transparently configured through the main interface. The design of the COE is documented in deliverable D4.1d [LLW⁺15].

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 a high-level description of each tool, please refer to Appendix B.

5.1 Overture

Overture FMU support is implemented as a tool wrapper, meaning that models exported from Overture require the Overture tool to simulate. This is implemented such that the VDM Interpreter and its FMI interface are included in the exported FMU. The 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.

FMU export from Overture is currently done manually and supports the FMI 2.0 standard. This means that the user will have to download the tool wrapper and assemble the FMU manually.

FMU Layout The FMU has the layout shown in Listing 1.

```
binaries
  linux64
    <FMU name>.so
  darwin64
    <FMU name>.dylib
  win32
    <FMU name>.dll
  win64
    <FMU name>.dll

sources
  *.vdmrt
resources
  config.txt
  crescendo-fmi-2.0.7-SNAPSHOT-jar-with-dependencies.jar
  modelDescription.xml
```

Listing 1: FMU Layout.

The VDM Tool wrapper includes the folders `binaries` and `resources` and a `modelDescription.xml` file that can be used as a template.

Export Procedure The export procedure currently consists of the following tasks that the user must perform:

1. Download the VDM tool wrapper (link shown above).
2. Extract it and create the directory structure as shown above.
3. Rename the binaries (`*.so/*.dll/*.dylib`) such that they match the FMU name e.g: `watertank.so`.

4. Copy the VDM source files (`*.vdmrt`) into the `sources` directory.
5. Update the model description (example shown below).
6. Compress the directories `binaries`, `sources`, `resources` and the `modelDescription.xml` file into a zip archive named `<FMU name>.fmu` using the compression method *deflate*.

The VDM language dialect which supports FMU export is object-oriented and thus does not have blocks with defined inputs and outputs. Therefore, a custom mapping of the internal system state to the scalar variables of FMI must be created manually. To illustrate how such a mapping can be made, the Watertank example from the Crescendo tool [IPG⁺12] is used. The Watertank is a simple model of a controller that adjusts the level in a tank with a constant in-flow. It keeps the water level between a low and high mark by opening and closing a valve. The system model consists of a level sensor Listing 3, valve actuator Listing 4, the controller Listing 5 and a system configuration Listing 2.

```

system System

instance variables

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

```

Listing 2: The Watertank system.

```

class LevelSensor

instance variables
  level : real := 0.0;
  ...
end LevelSensor

```

Listing 3: Level Sensor.

```

class ValveActuator

instance variables
  valveState : real := 0.0;
  ..
end ValveActuator

```

Listing 4: Valve Actuator.

```

class Controller

values

```

```
public minLevel : real = 0.0; -- {m}
public maxLevel : real = 0.0; -- {m}
...

end Controller
```

Listing 5: The Watertank controller.

The mapping between VDM values (constants), instance variables, and FMI scalar variables, is shown in Table 1. The mapping includes the VDM value `maxLevel` and the instance variables `level` and `valveState`. This illustration only shows how basic, unstructured VDM values and instance variables can be mapped to FMI. Not all types of instance variable can be exported, since it must be possible to express a path to the variable from the system class using dots to separate fields, and the final element in the path may not be more than just the name of a variable. This means that instance variables and values may not come in the form of arrays, lists, sets *etc.*, as accessing leaf values stored in such variables would require the use of the corresponding accessor mechanism (array index for arrays, head element retrieval for lists *etc.*)¹. The name mapping for Table 1 is given in Listing 6.

The scalar variables must be mapped to the internal state of the VDM model as described above. This can be done using the model description element `Overture` illustrated in Table 1, where the scalar variable value reference is linked to the corresponding path inside the `System` class of the VDM model.

```
<Overture>
  <link valueReference="0" name="Controller.maxLevel"/>
  <link valueReference="3" name="System.levelSensor.level"/>
  <link valueReference="4" name="System.valveActuator.
    valveState"/>
</Overture>
```

Listing 6: Name mapping for the variables in Table 1.

¹The restriction on accessor mechanisms is due to a limitation of the current implementation and not the language.

Table 1: VDM to FMI mapping.

VDM	FMI 2
<pre> class Controller -- Parameter values maxLevel: real = 5; end Controller </pre>	<pre> <ScalarVariable name="maxLevel" valueReference="0" causality="parameter" variability="fixed" initial="exact"> <Real start="5"/> </ScalarVariable> </pre>
<pre> class LevelSensor -- input instance variables level : real := 0.0; end LevelSensor </pre>	<pre> <ScalarVariable name="level" valueReference="3" causality="input" variability="continuous"> <Real start="0"/> </ScalarVariable> </pre>
<pre> class ValveActuator -- output instance variables valveState : real := 0.0; end ValveActuator </pre>	<pre> <ScalarVariable name="valveState" valueReference="4" causality="output" variability="discrete" initial="calculated"> <Real/> </ScalarVariable> </pre>

5.2 20-sim

The current 20-sim version (version 4.5) does not yet include the necessary FMU export feature for INTO-CPS by default². It can be downloaded separately from <https://github.com/controllab/fmi-export-20sim>. Note that to automatically compile the FMU, you will need the Microsoft VC++ 2010, 2013 or 2015 compiler installed (Express or Community edition is fine). To install the FMU export extension:

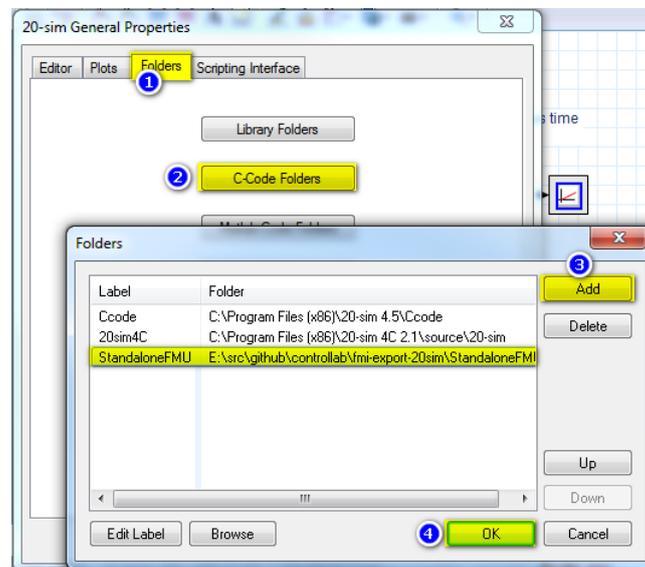


Figure 25: Add the FMU export template.

1. Download the code generation template using the *Download ZIP* button on the above mentioned Github website.
2. Extract the zip file.
3. Copy the just extracted folder `StandaloneFMU` to a location on your PC with 20-sim.
4. Open 20-sim.
5. From the main menu, choose *Tools* → *Options*.
6. Choose *Folders*.
7. Choose *C-Code Folders*.

²Note that 20-sim is only supported on the Windows platform.

8. Add the folder you copied in step 3 (see also Figure 25).

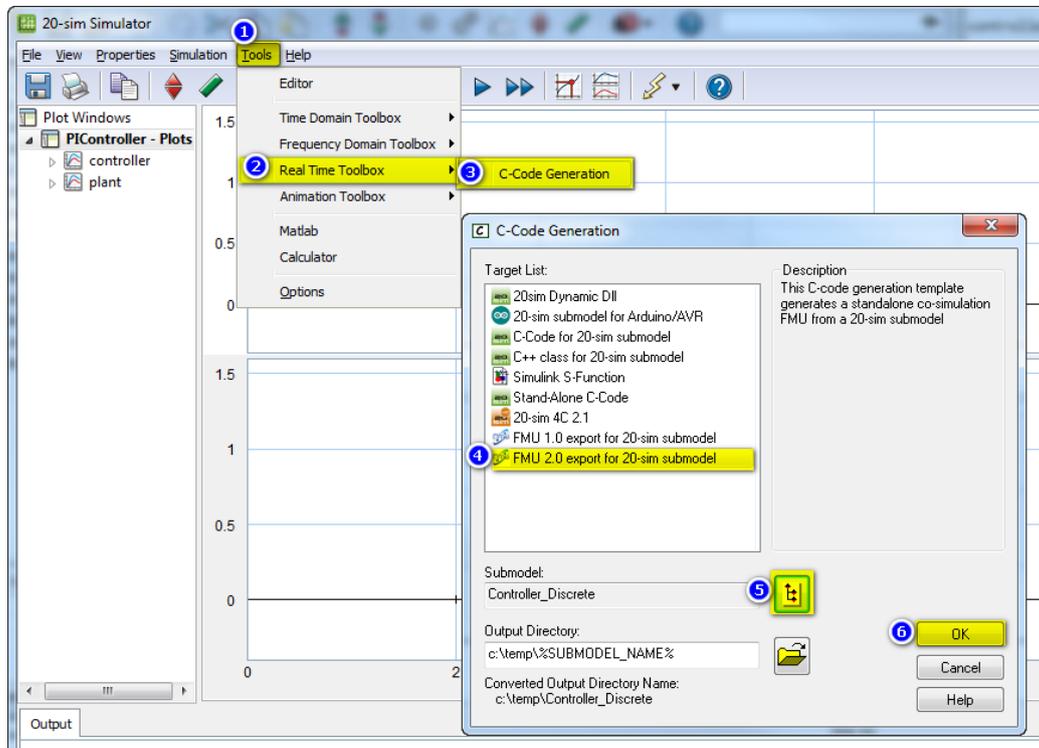


Figure 26: Export an FMU from 20-sim.

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 wanted.

Next, follow these steps (see also Figure 26):

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.

If 20-sim can find one of the supported VC++ compilers, it will start the compilation and report where you can find the freshly generated FMU.

Please note that currently only a subset of the supported 20-sim modeling language elements can be exported as code. 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. Other language features that are not, or are only partly supported for code generation, are:

- File I/O.
- Calls to external code (DLLs, Matlab *etc.*)
- Variable delay blocks.
- Event functions.

Full support for all 20-sim features is only possible through the toolwrapper FMU approach. Support for this is planned for the second year of the project.

5.3 OpenModelica

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

- From a command line.
- From OMCedit (OpenModelica Connection Editor).

FMU export from a command line To export an FMU for co-simulation or model exchange from a Modelica model in OpenModelica, you can use a Modelica script file `generateFMU.mos` containing the following calls to the OMC compiler:

```
// --- start file generateFMU.mos
// 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
();
// --- end file generateFMU.mos
```

Then, 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 27.
- Load the Modelica Model in OMEdit: see Figure 28.
- Open the Modelica model in OMEdit: see Figure 29.
- Export the FMU via the menu: see Figure 30.
- The FMU is now generated: see Figure 31.

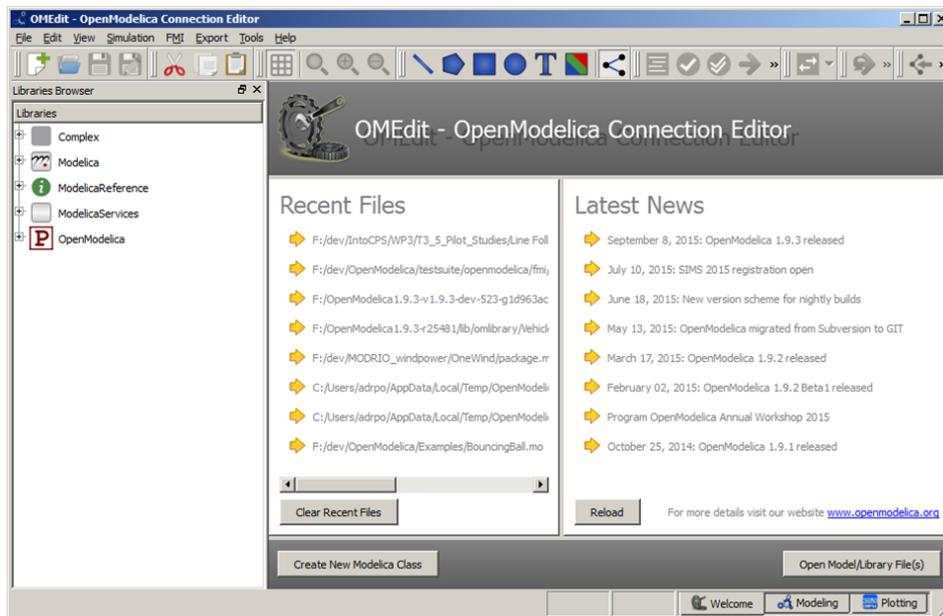


Figure 27: Open OMEdit.

At the end the FMU will be present in: `%TEMP%\OpenModelica\OMEdit.`

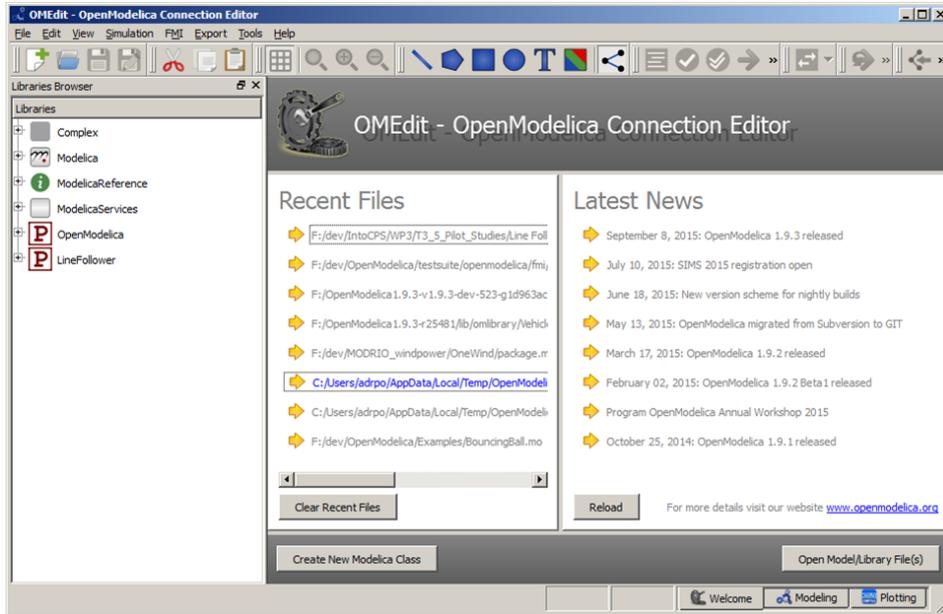


Figure 28: Load the Modelica model in OMEdit.

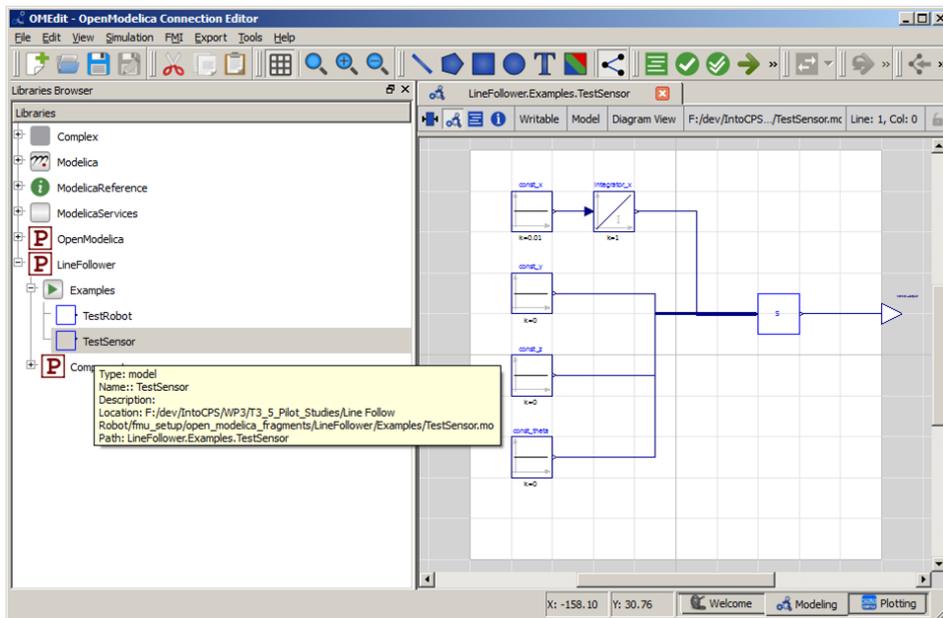


Figure 29: Open the Modelica model in OMEdit.

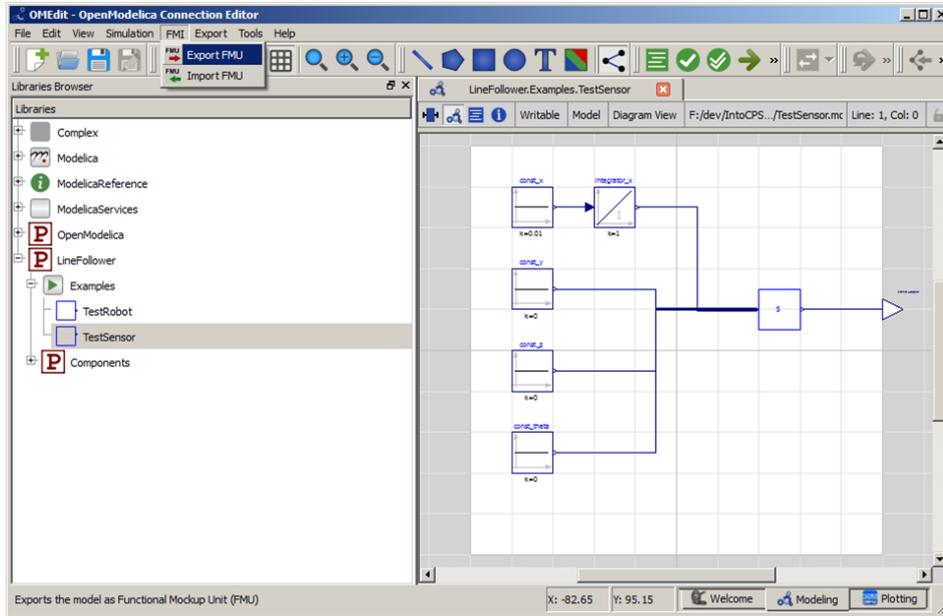


Figure 30: Export the FMU via the menu.

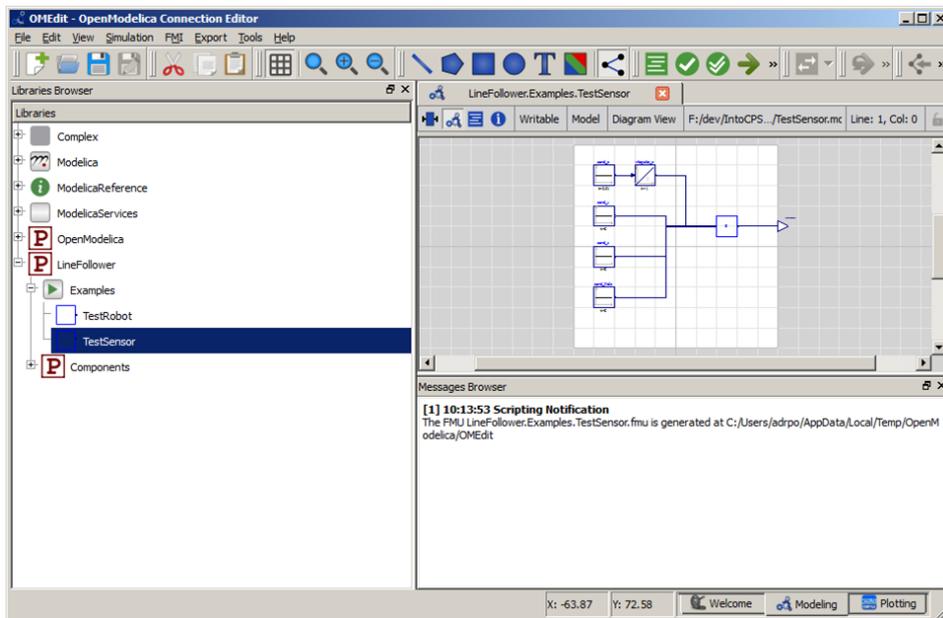


Figure 31: The FMU is generated

5.4 RT-Tester / RTT-MBT

The RTT-MBT component caters to FMI/FMU through a specialised feature release. This feature makes it possible to treat model exports from the INTO-CPS tool chain as behavioural entities that come attached with annotations that reflect functional and time-related requirements - also known as *test models*.

Some of these annotations are implicit (like states and transitions) and can always be used to compose a *test goal*, *i.e.*, a number of situations that shall be reached in an execution, *cf.* Figure 32. Often used strategies—like state

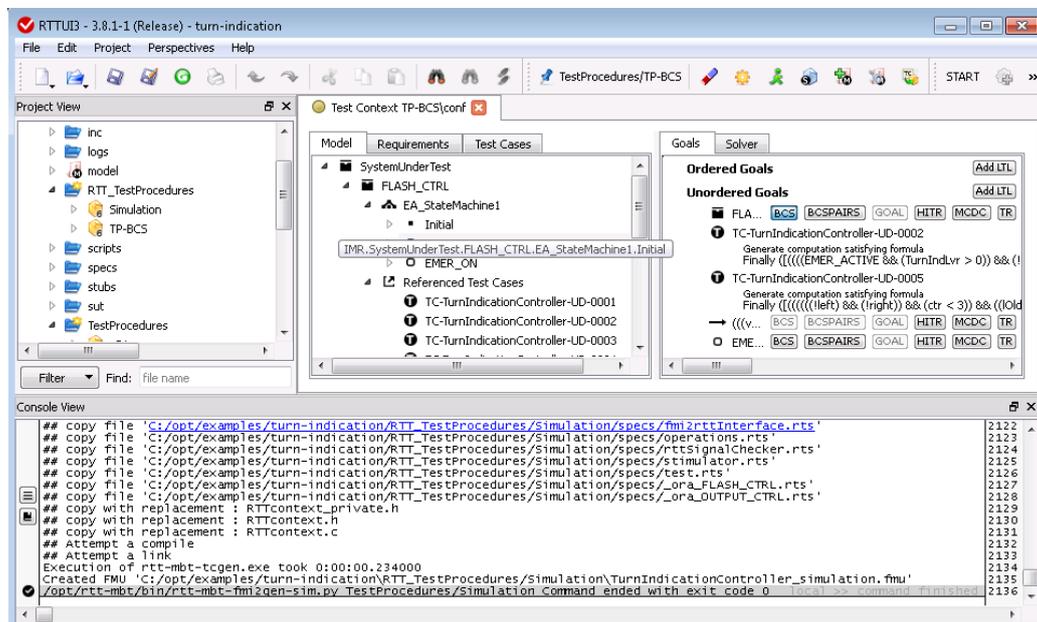


Figure 32: Example of a RTT-MBT configuration.

coverage—are activated by clicking on them. Specific reachability goals can be added via drag-and-drop of model elements, and more complex goals can be textually supplemented by an LTL formula.

Based on a test goal, a sequence of stimulations (*i.e.*, timed inputs) is then generated by a SAT solving technique. The RT-Tester testing back-end serves as the execution engine for this sequence by means of a *test procedure*, which is then automatically cast into an FMI-compliant FMU. That “test” FMU can then be run against one or more system components (also provided as FMUs) and use the reporting mechanisms of the test tool to record and trace the results.

5.4.1 Setup RT-Tester User Interface

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

1. User name and company name (Figure 33a).
2. Location of Bash shell (Figure 33b): You can safely skip this step by clicking *Next*.
3. Path to Python 2.7 executable (Figure 33c): Click *Detect* and then *Installation Path* for auto-detection, or *Browse* to select manually.
4. Location of RT-Tester (Figure 33d): Click *Browse* to select the directory of your RT-Tester installation. Note that if you did not specify the Bash shell location in step 2, the version number might not be properly detected.

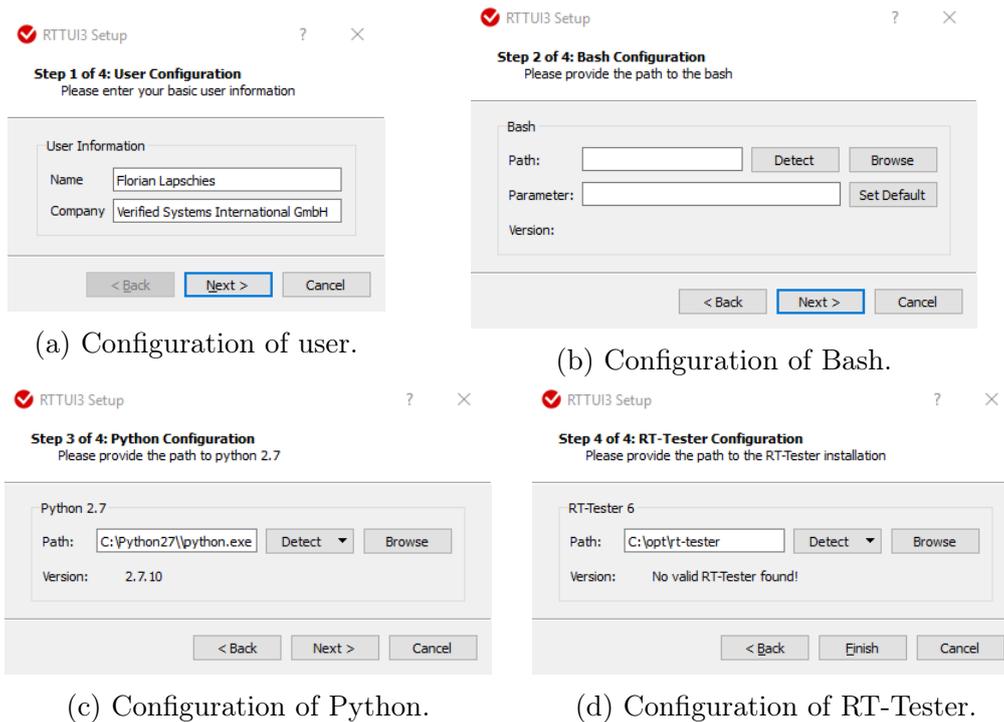


Figure 33: RT-Tester: GUI Configuration.

5.4.2 First Steps Using the Example Project

This section provides instructions for taking the first model based testing steps with RT-Tester using a small toy project. In this toy project a simplified turn indicator model is controlling the flashing lights of a car. We will learn how to generate FMUs for existing test procedures and how to create FMUs for simulation of a system under test. More details on how to specify additional test procedures can be obtained from the RT-Tester Model-Based Test Case and Test Data Generator Manual [Ver15b].

To open the project click *File* → *Open* → *Project* → *Local RT-Tester Project* and select the directory containing the turn indicator project (see Figure 34).

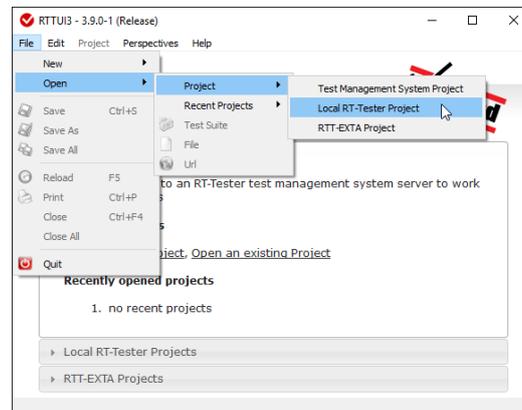


Figure 34: RT-Tester: Open Project.

We now have to setup the License Management for that specific project. Click *Project* → *Project Settings*, select *Local Environment* and change the Variable *USER* to your Windows user name (see Figure 35). The user name can be obtained by typing `echo %username%` into the Windows Command Prompt. Then click on the *Project Action* called *START* (see Figure 36).

Next, import the model by clicking *Project* → *Model-based Testing* → *Import Model* → *Import from File*. Select the `xmi` file in the `model` directory of the test project.

After importing the project and the model, the Project View can be found on the left side (see Figure 36). Of special importance is a sub-folder called *TestProcedures* containing the so called *Test Procedure Generation Context*. Here the test-engineer specifies the desired test procedures in an ab-

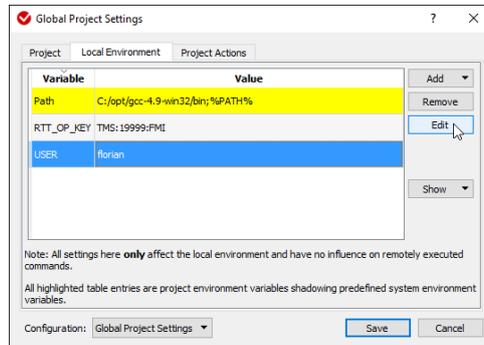


Figure 35: RT-Tester: Set user name.

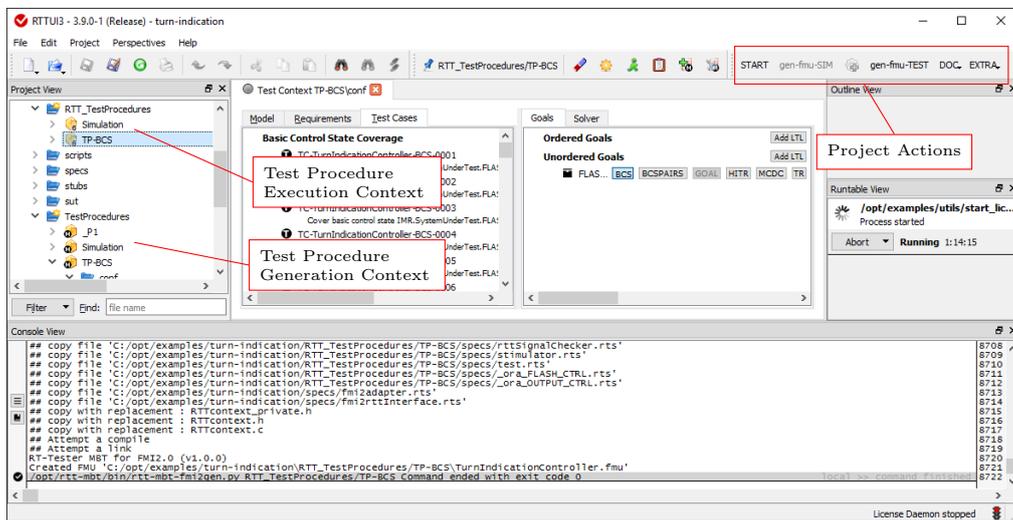


Figure 36: RT-Tester: Overview.

stract way. When commanded to generate the concrete procedure meeting the configured test objective, RT-Tester places the selected concrete test procedure in a separate folder named `RTT_TestProcedures`. This folder is the so called *Test Procedure Execution Context* containing test procedures that can be compiled to an FMU which drives the system under test through the specified test scenario.

The abstract test procedure *TP-BCS* has already been preconfigured with the test obligation to reach all basic control states in the test model. In order to have RT-Tester generate a concrete test procedure that provides concrete timed inputs that will steer the system under test to the desired states, select the test procedure and click on the *SOLVE* Project Action and then on *OK*. To view any newly generated folders hit the *F5*-key. The folder *RTT_TestProcedures* should now contain a folder *TP-BCS* with the generated test procedure. In order to create an FMU for it, select the test procedure in the Project View and execute the Project Action *gen-FMU-TEST*. This should generate an FMU named `TurnIndicationController.fmu`.

RT-Tester is also able to generate a simulation from the test model which can serve as a replacement for the system under test. To create such an FMU, select the abstract test procedure *Simulation* and then execute the Project Action *gen-fmu-SIM*. This will generate the simulation FMU named `TurnIndicationController_simulation.fmu`.

6 Design Space Exploration for INTO-CPS

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

The DSE module [GHJ⁺15] at this time exists as a pair command line Python scripts, and this section describes how the current version is used and what it does. It is important to note that these scripts have been built with Python version 2.7 in mind and may not be compatible with other versions. The DSE scripts also only support open-loop exhaustive search at this point³.

The assumptions of the scripts are that they are placed in a folder along with the `config.json` file which describes the multi-model the DSE will be based upon, as shown in Figure 37. The DSE scripts themselves are found in the archive `DSE.zip` that comes as part of the INTO-CPS release bundle.

³There are updates planned by first quarter of 2016 to implement closed loop DSE. This is described in more detail in deliverable D5.1a [GHJ⁺15]

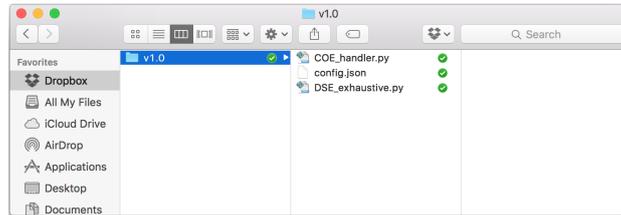
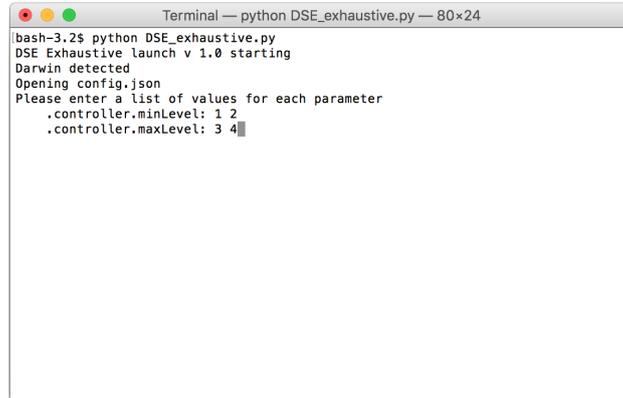


Figure 37: The contents of a folder at the start of a DSE.

Create a folder in which the DSE results are to be stored and extract the scripts from `DSE.zip` into it. Please note that the name of the folder is not important. Next, it is necessary to copy the simulation configuration file (`config.json`) that describes the model of the system into this folder. The scripts also assume that the COE is running and that the required FMUs are in the location defined in the `config.json` file. The script is launched in the same way in both Mac OSX and Windows using the command `python DSE_exhaustive.py`, as on the top line of Figure 38 (OSX) and Figure 39 (Windows), and pressing `return` (the following examples are all taken from OSX, but their output in Windows is identical). A final comment is that the script is not yet tolerant of user input mistakes and does not allow the user to return to alter information entered earlier, so if a mistake is made or if the user wishes to change an earlier value, it is suggested that the script be escaped by using the `<ctrl>-<c>` key sequence.

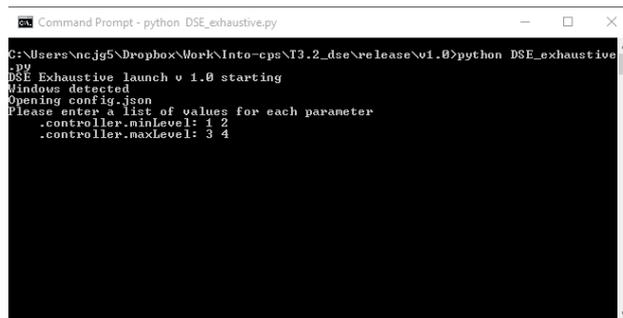
Once launched, the first action of the script is to extract the parameters defined in the simulation configuration. Each of these parameters defines the value of some part of the model and it is these parameter values that the script will vary during the DSE. In Figure 38, the script has found two parameters, `.controller.minLevel` and `.controller.maxLevel` and the user has been asked to enter a set of values for each. For each parameter the script expects the user to enter a sequence of integers (e.g. `1 5 500`) or decimal numbers (e.g. `0.1 15.7 200.0`) or a combination of these (e.g. `1 2.7 100`) separated by spaces. In this case the user entered `1` and `2` for `.controller.minLevel`, then pressed `return` and entered `3` and `4` for `.controller.maxLevel`. As we will see later, in this exhaustive model of DSE the simulation will be run with all four combinations of these parameters. There is no upper limit on the maximum number of values one may enter for each parameter, but each parameter must have at least one.

Pressing `return` after entering the parameter values moves the script to prompt the user to enter the start and end times for the simulations, Figure 40. Here the user should enter a value for the start time, press `return`



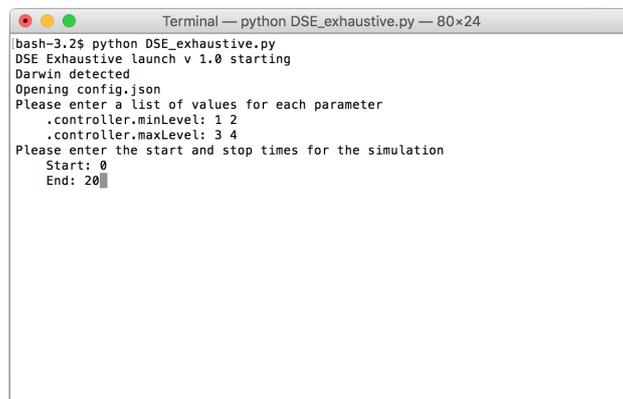
```
Terminal — python DSE_exhaustive.py — 80x24
bash-3.2$ python DSE_exhaustive.py
DSE Exhaustive launch v 1.0 starting
Darwin detected
Opening config.json
Please enter a list of values for each parameter
.controller.minLevel: 1 2
.controller.maxLevel: 3 4
```

Figure 38: Simulation parameter values entered (OSX).



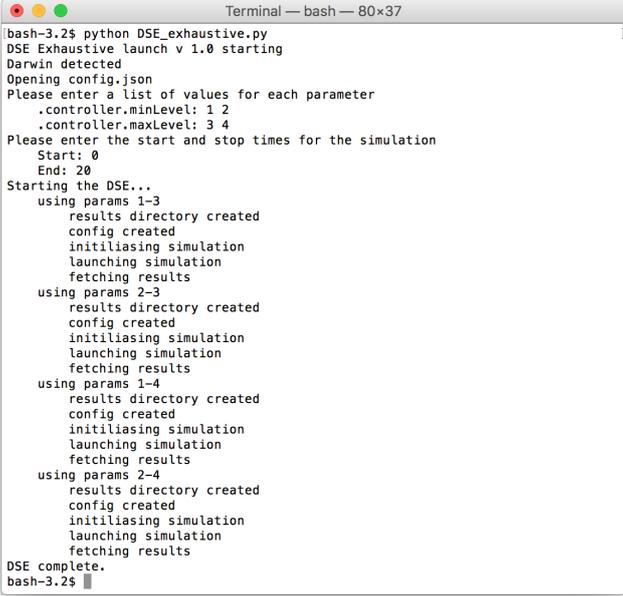
```
Command Prompt - python DSE_exhaustive.py
C:\Users\ncjg5\Dropbox\Work\Into-cps\T3.2_dse\release\v1.0>python DSE_exhaustive
.py
DSE Exhaustive launch v 1.0 starting
Windows detected
Opening config.json
Please enter a list of values for each parameter
.controller.minLevel: 1 2
.controller.maxLevel: 3 4
```

Figure 39: Simulation parameter values entered (Windows).



```
Terminal — python DSE_exhaustive.py — 80x24
bash-3.2$ python DSE_exhaustive.py
DSE Exhaustive launch v 1.0 starting
Darwin detected
Opening config.json
Please enter a list of values for each parameter
.controller.minLevel: 1 2
.controller.maxLevel: 3 4
Please enter the start and stop times for the simulation
Start: 0
End: 20
```

Figure 40: Simulation start and end time entered.

A terminal window titled "Terminal — bash — 80x37" showing the execution of a Python script named "DSE_exhaustive.py". The script outputs the following text:

```
bash-3.2$ python DSE_exhaustive.py
DSE Exhaustive launch v 1.0 starting
darwin detected
Opening config.json
Please enter a list of values for each parameter
.controller.minLevel: 1 2
.controller.maxLevel: 3 4
Please enter the start and stop times for the simulation
Start: 0
End: 20
Starting the DSE...
using params 1-3
  results directory created
  config created
  initiliasing simulation
  launching simulation
  fetching results
using params 2-3
  results directory created
  config created
  initiliasing simulation
  launching simulation
  fetching results
using params 1-4
  results directory created
  config created
  initiliasing simulation
  launching simulation
  fetching results
using params 2-4
  results directory created
  config created
  initiliasing simulation
  launching simulation
  fetching results
DSE complete.
bash-3.2$
```

Figure 41: The complete terminal output from the DSE script for a small DSE run.

and then enter a value for the end time, where the values may be integers, decimals or a mixture, representing the time in seconds. These times will be used for all simulations in the DSE. Pressing `enter` once both values are entered will start the simulation phase of the DSE. After the script has reported that it is starting the DSE, it provides some simple information about the progress of the DSE as shown in Figure 41. Here we see that DSE begins by using parameter values 1 and 3, and with these parameters it first creates a suitably named folder to store the launch configuration and simulation results. The script then proceeds to interact with the COE following the standard pattern of first initialising the simulation, then launching the simulation. Once the simulation is complete, it fetches the results and places them in a file called `results.csv` in the folder for that simulation. This process is repeated until all combinations of the simulation parameters have been exhausted, at which point the script reports that DSE is complete and terminates.

After the DSE script terminates, the original folder will now contain one new subfolder for each simulation run during the DSE, where each folder is named with the values of the parameters used in the simulation it represents, Figure 42. Each subfolder contains two files, a `config.json` file containing the configuration used for that simulation, including its specific parameter

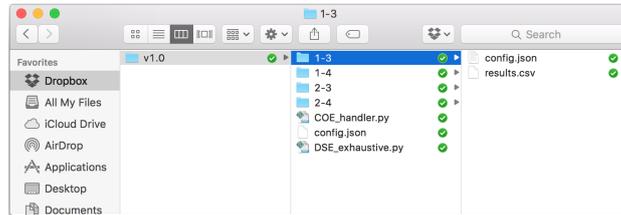


Figure 42: The contents of the DSE folder when the script and simulations have completed.

values, and a `results.csv` file containing all the values logged by the COE during the simulation.

This is where the DSE script support ends. In the current version, the process of analysing each `results.csv` to obtain the objective values for that simulation and the subsequent processing of the objective values for all simulations to rank the results to reveal the best performers is not yet implemented.

talk about this functionality being included in the into app, as in d7.3, and the time scales it will occur on

These scripts do not represent the final vision for DSE support within INTO-CPS and features to support closed loop DSE and the presentation of the findings are currently under development. An outline plan for the development of these features, which are driven by the case studies and user requirements, may be found in D5.1a [GHJ⁺15].

7 Test Automation for INTO-CPS

This section discusses how test automation can be used in the context of INTO-CPS. The key idea of test automation is to express the desired behaviour of an INTO-CPS system multi-model by means of a separate testing model. This model, which ideally reflects the specification of the system behaviour, is then used to automatically generate test stimulations and/or test oracles. The stimulations serve as inputs to the tests, whereas the oracles examine whether the observed system behaviour coincides with the desired behaviour. It is important to stress that these two functionalities, if generated using the RT-Tester Model-Based Test Case Generator (RTT-MBT),

are entirely independent⁴. It is thus possible, for example, to combine hand-written test inputs with auto-generated test oracles.

By default, RTT-MBT takes as input one single test model and treats this model as the system specification. The setting in INTO-CPS is different because the system is composed of a number of connected components, some of which may have no corresponding model. As an example, a component may actually be a real hardware controller. However, a commonality among all components is that they are integrated as FMUs and thus exhibit input and output interfaces, which form the basis for the connections between the components. The FMUs define which variables become visible to the outside, and the ranges of these variables. It is thus possible to treat FMUs that do not come with a model as black boxes, the internal behaviour of which is unknown. Such black boxes too induce a transition relation, though with some form of nondeterminism, that can be used by RTT-MBT for test generation.

The RTT-MBT approach to model-based testing of simple systems can thus be lifted to a co-simulation environment for cyber-physical systems by including the FMU specification in the transition relation used for test generation. These FMUs are treated as part of the environment and impose constraints on input and output variables. Further details are given in [MPB15].

8 Code Generation for INTO-CPS

Each of the tools described in Section 5 has the ability to translate models into platform-independent C source code. Currently, Overture can translate VDM models written in the executable subset of VDM++ [LLB11] to Java code, while translation to C is under development. 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 here is to use 20-sim 4C to compile and deploy the code to hardware targets, since the tool incorporates the requisite knowledge

⁴RT-Tester is a test system that is based on tests written in a dedicated C-like programming language called *Real-Time Test Language* (RTTL). Such tests can be compiled, executed and documented using RT-Tester. RTT-MBT is an upgrade for RT-Tester, which adds model-based testing functionality to the RT-Tester system. The tests are thus generated as RTTL source code and can be turned into executable tests using RT-Tester.

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 MiL 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.

9 Conclusions

This deliverable is the user manual for the INTO-CPS tool chain after the first 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. 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.

There are still gaps in the tool chain where fully automated connectivity is not yet achieved. For instance, model checking of multi-models is planned for the second year of the project. However, significant progress has been made and it is felt that the foundations and the procedures are in place to achieve a fully connected chain of tools later in the INTO-CPS project.

References

- [APCB15] Nuno Amalio, Richard Payne, Ana Cavalcanti, and Etienne Brosse. Foundations of the SysML profile for CPS modelling. Technical report, INTO-CPS Deliverable, D2.1a, December 2015.
- [BF15] Jörg Brauer and Simon Foster. Abstraction Techniques from CT to DE. Technical report, INTO-CPS Deliverable, D5.1c, December 2015.
- [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.
- [Blo14] Torsten Blochwitz. Functional mock-up interface for model exchange and co-simulation. <https://www.fmi-standard.org/downloads>, July 2014. Torsten Blochwitz Editor.
- [BQS15] Etienne Brosse, Imran Quadri, and Andrey Sadovykh. COE Contracts from SysML. Technical report, INTO-CPS Deliverable, D4.1c, December 2015.
- [Bro97] Jan F. Broenink. Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD*, 38(3):22–25, 1997.
- [CKD15] M.A. Groothuis C. Kleijn and H.G. Differ. *20-sim 4.5 Reference Manual*. Controllab Products B.V., 2015.
- [Con13] Controllab Products B.V. <http://www.20sim.com/>, January 2013. 20-sim official website.
- [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.
- [FE98] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *EC-COP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.
- [FGPP15] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Method Guidelines 1. Technical report, INTO-CPS Deliverable, D3.1a, December 2015.

- [Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, January 2004.
- [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.
- [GHJ⁺15] Carl Gamble, Francois Hantry, Claes Dühring Jæger, Christian König, Alie El din Madie, and Richard Payne. Design Space Exploration in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D5.1a, December 2015.
- [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.
- [IPG⁺12] Claire Ingram, Ken Pierce, Carl Gamble, Sune Wolff, Martin Peter Christensen, and Peter Gorm Larsen. Examples compendium. Technical report, The DESTTECS Project (INFSO-ICT-248134), October 2012.
- [KG15] C. Kleijn and M.A. Groothuis. *Getting Started with 20-sim 4.5*. Controllab Products B.V., 2015.
- [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.
- [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, Anders Terkelsen, Miran Hasanagić, Casper Thule Hansen, Carl Gamble, Oliver Kotte, Adrian Pop, and Etienne Brosse. Design of the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D4.1d, December 2015.
- [MPB15] Oliver Möller, Adrian Pop, and Jörg Brauer. Distributed Testing and Simulation Network. Technical report, INTO-CPS Deliverable, D5.1b, December 2015.
- [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.
- [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.

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
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
DESTTECS	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	Softteam
SUT	System Under Test
SVN	Subversion
SysML	Systems Modelling Language
TA	Test Automation
TE	Test Environment
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, Objectteering, 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]. 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 analysed 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 analyse models, build control systems and improve system performance. Figure 43 shows 20-sim with a model of a controlled

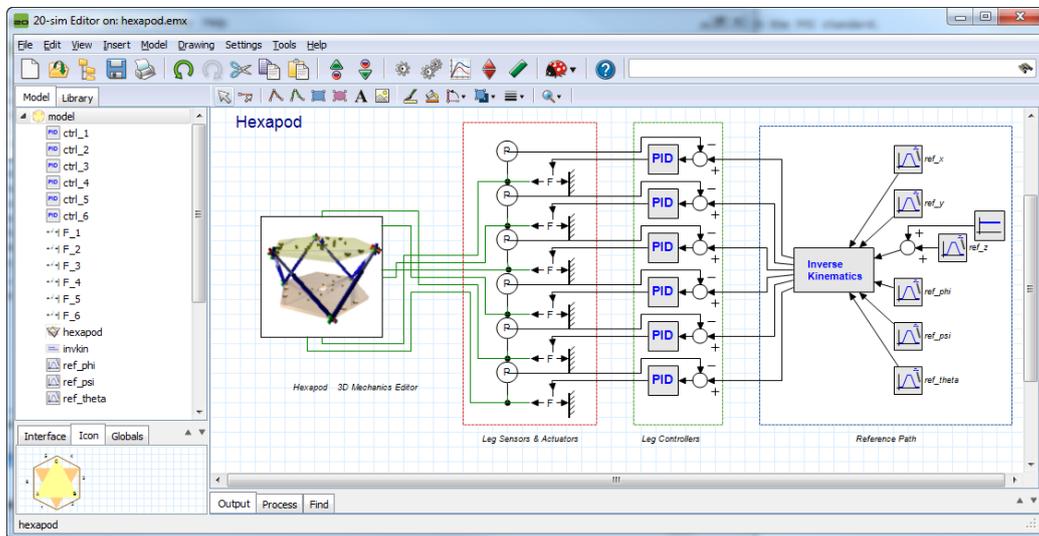


Figure 43: Example of a hexapod model in 20-sim.

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 [KG15] 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> [CKD15]. 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 step 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

On top of this, 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 44, 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:

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

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

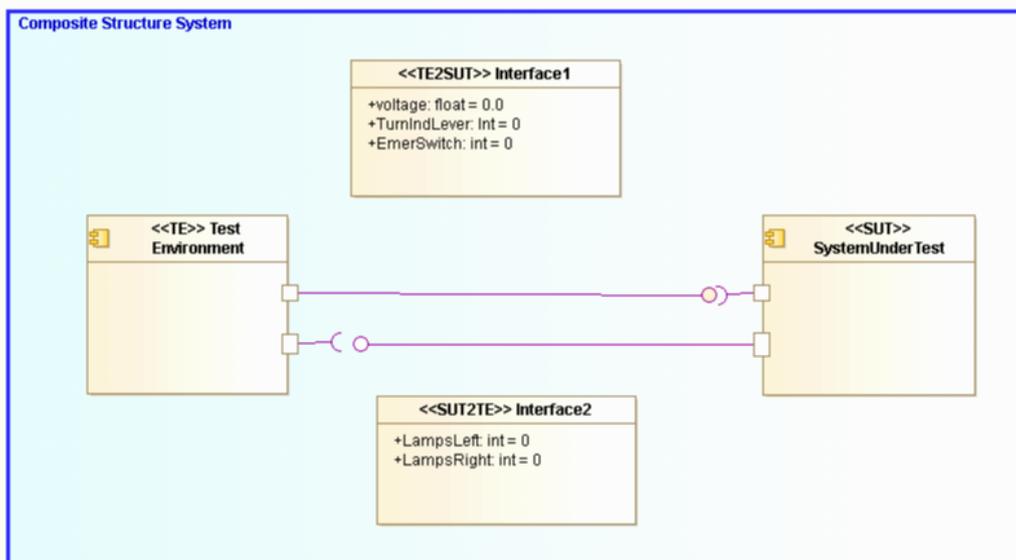


Figure 44: Simple model that highlights interfaces between the environment and the system-under-test.

C Obtaining the Individual Tools

If required, the individual components of the INTO-CPS tool chain can be obtained as follows.

Modelio: The Modelio modelling tool can be obtained from

```
https://www.modelio.org/downloads/  
download-modelio.html
```

INTO-CPS Application: The Modelio SysML extension can be obtained as an extension to Modelio from

```
http://forge.modelio.org/projects/sysml-modelio34/  
files
```

INTO-CPS Application: The INTO-CPS Application can be obtained as an extension to Modelio from

```
http://forge.modelio.org/projects/  
intocps-modelio34/files
```

To be able to use the INTO-CPS/SysML and INTO-CPS Application extensions in Modelio, you have to first add the SysML and INTO-CPS Modelio modules to the Modelio module catalog, as shown in Figure 45.

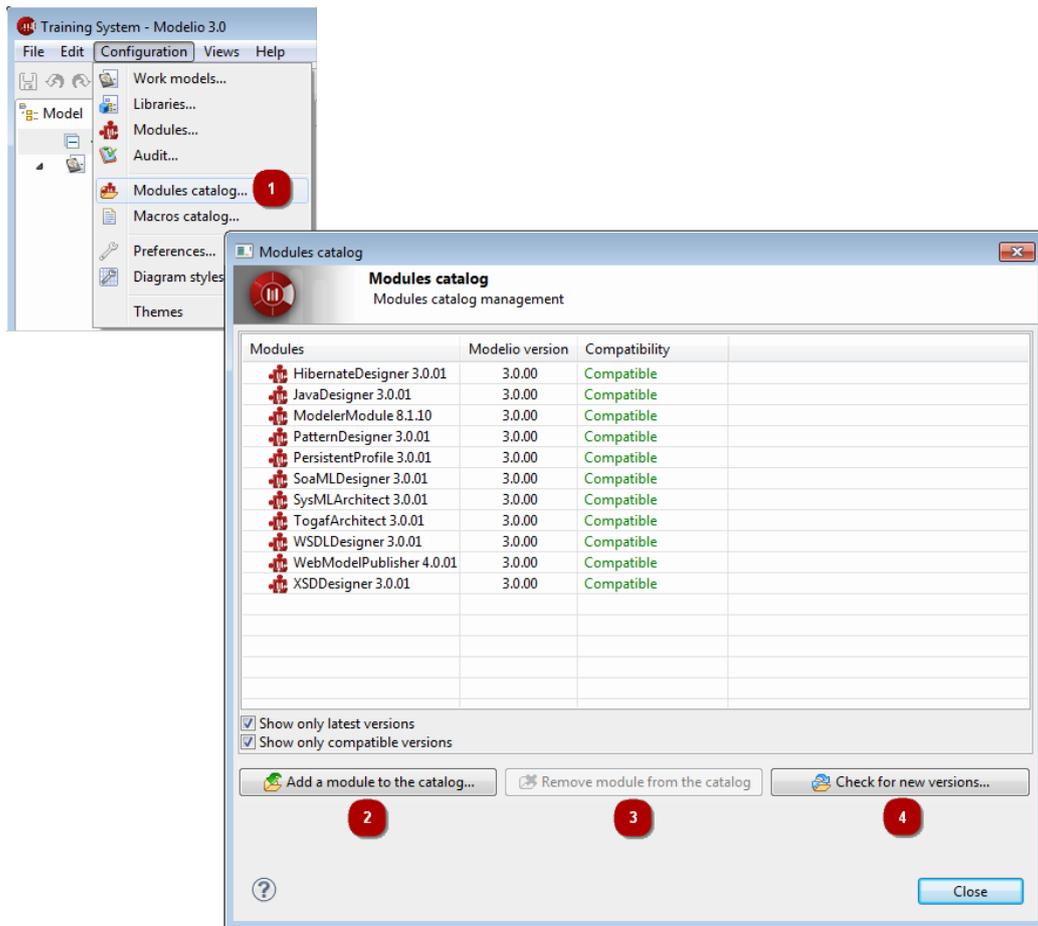


Figure 45: Modelio module catalog.

Follow these steps:

1. Run the *Configuration* →  *Modules catalog...* command.
2. To add the extension module, click on *Add a module to the catalog...* and use the file browser to select the INTO-CPS module file INTOCPS_X.X.X. jmdac from its download location.
3. To check whether a new version of the module exists, and to install it in the catalog, click on *Check for new versions...* .

Next, install the INTO-CPS extension module into each INTO-CPS project, as shown in Figure 46. Follow these steps.

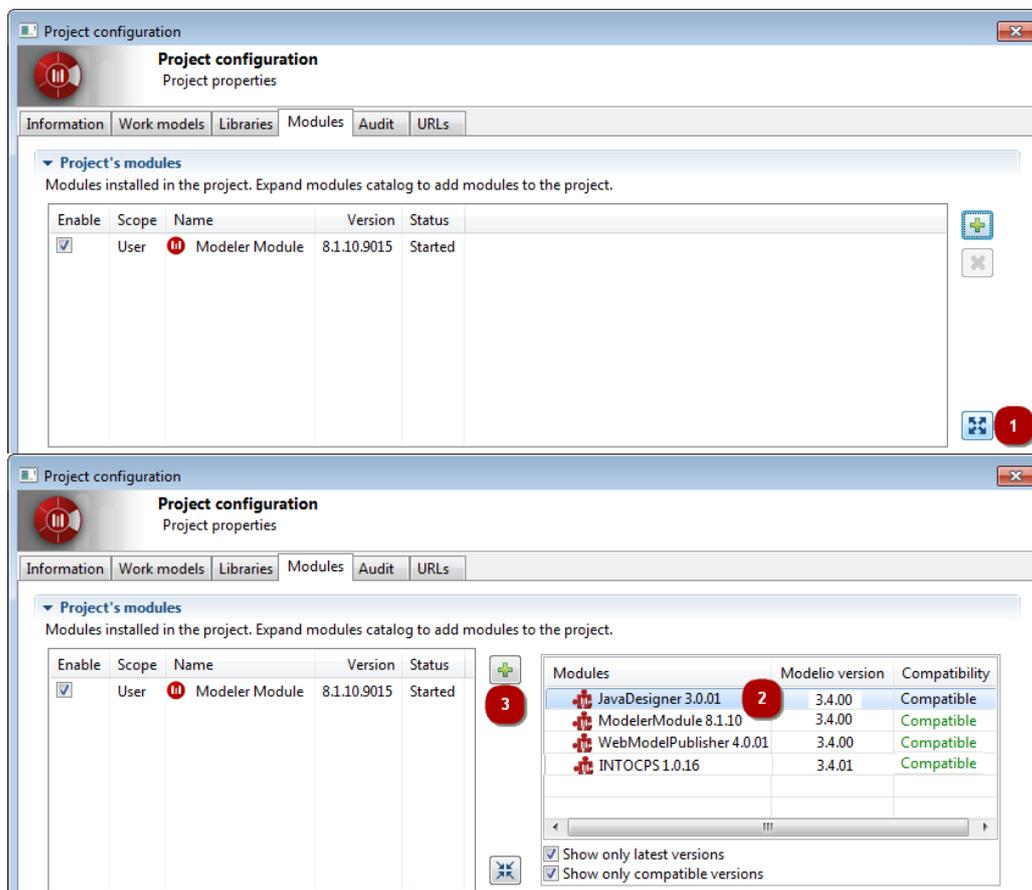


Figure 46: Modelio module installation.

1. Open the project configuration.
2. Click on  to expand the Modules catalog.

3. In the Modules catalog, select the INTO-CPS module.
4. Click on  to install the module in the project.

The INTO-CPS Application is now ready for use inside Modelio.

OpenModelica: Distributions of OpenModelica in several flavours, including a virtual machine image, can be downloaded from

`http://www.openmodelica.org`

20-sim: The 20-sim modelling and simulation tool can be obtained for Windows from:

`http://www.20sim.com/download.html`

Overture: The Overture platform can be obtained for various platforms from:

`http://overturetool.org`

RT-Tester: RT-Tester can only be obtained as part of the INTO-CPS complete bundle from:

`http://overture.au.dk/into-cps/site/download`

D 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.

D.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 drift between the constituent simulations; for asking each individual simulation to perform a simulation step; and for passing 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 assembled 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 sub-models, the process repeats.

D.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 analyse 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 travelled 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 minimises the total amount of energy used, or it may be more complex if there are multiple objectives to optimise 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.

D.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.

D.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 allowed 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 one, which is the one into which the model is translated via code generation.