

**Crescendo Technical Report Series
No. TR-001**

November 2013

Crescendo Tool Support: User Manual

Version 2.0.0

by

Peter Gorm Larsen, Kenneth Lausdahl, Joey Coleman and Sune Wolff
Aarhus University, Department of Engineering
Finlandsgade 22, DK-8200 Aarhus N, Denmark

Christian Kleijn and Frank Groen
Controllab Products B.V.
Hengelosestraat 500, 7521 AN Enschede, The Netherlands





Document history

Month	Year	Version	Version of Crescendo.exe
December	2012	1	1.1.8 (then called DESTecs)
November	2013	2	2.0.0

Contents

1	Introduction	1
1.1	What is the Crescendo Tool?	1
1.2	What was the DEST ECS Project?	1
1.3	What is the Vienna Development Method?	1
1.4	What are Bond Graphs?	2
1.5	Related Tools	2
1.5.1	Overture	3
1.5.2	Crescendo	3
1.5.3	Symphony	3
1.6	Structure of this User Manual	3
2	Basic Crescendo Concepts	5
2.1	Models	5
2.2	Simulation	5
2.3	Co-Simulation	6
2.4	Contract	6
3	Getting Hold of the Software	7
3.1	Requirements	7
3.2	Installation	7
3.2.1	Combined Installer	7
3.2.2	License	8
3.2.3	Manuals	8
3.3	20-sim Standalone	8
4	Quick Start with Crescendo	11
4.1	Opening Crescendo	11
4.2	Opening a Project	12
4.3	Running a Project	12
5	Editors and Management of Projects	17
5.1	The Crescendo Workbench	17
5.1.1	Explorer View	18



5.1.2	Editor View	18
5.1.3	Outline View	19
5.1.4	Simulation Engine View	19
5.1.5	Console View	20
5.2	Handling Projects	21
5.2.1	Creating new Projects	21
5.2.2	Importing Projects	21
5.2.3	Exporting Projects	23
5.3	Managing Contracts	24
5.3.1	Creating a new Contract File	24
5.3.2	Contents of a Contract	25
5.3.3	Error Detection in the Contract/Link File	27
5.3.4	Managing the Link Files	28
5.3.5	Contract Overview	30
6	Co-Simulation Possibilities	31
6.1	Debug Configuration	31
6.1.1	Creating a New Debug Configuration	31
6.1.2	Main Tab	31
6.1.3	Shared Design Parameters Tab	32
6.1.4	DE Simulator Tab	33
6.1.5	CT Simulator Tab	34
6.2	Post-Processing Tab	34
6.2.1	Advanced Tab	35
6.2.2	Common Tab	35
6.3	Scenarios	36
6.3.1	Creating a New Scenario File	36
6.3.2	CSL Syntax	37
6.3.3	CSL Examples	38
6.4	Logfiles	39
6.4.1	DE Variables	39
7	Design Space Exploration Possibilities	41
7.1	ACA Workflow	41
7.2	Using the ACA Features	42
7.2.1	The Main Tab	42
7.2.2	The Architecture Tab - Deployment Architectures	43
7.2.3	Shared Design Parameters Tab	43
7.2.4	Scenario Tab	45
7.2.5	CT Settings Tab	46
7.2.6	Common Tab	46
7.3	Repeating a Single Launch Part of an ACA	47



7.4	Folder Launch Configuration	47
7.5	Control Library	47
7.5.1	Accessing the Control Library	48
7.5.2	Using the Control Library	48
7.5.3	Advanced Use	50
7.5.4	Constructors	50
7.6	DE Architecture	52
7.7	Events	52
7.7.1	Simulation setup	53
7.7.2	Events in CT	53
7.7.3	Events in DE	54
8	Post-Analysis Possibilities	55
8.1	Octave	55
8.1.1	Octave Version	55
8.1.2	Octave use in Crescendo	55
8.1.3	Show Plot Automatically when Script is Run	55
8.1.4	Invoking Octave from Crescendo	56
8.1.5	Setting Octave path	57
A	Glossary	61



ABSTRACT

This document is the user manual for the Crescendo Integrated Development Environment (IDE) version 2.0.0 enabling collaborative analysis of models written in the Discrete-Event (DE) formalism VDM and the Continuous-Time (CT) formalism bond graphs. The specific dialect of VDM is called VDM Real Time (VDM-RT) and it is supported by the Overture tool whereas the bond graph formalism is supported by the tool called 20-sim. Both the Crescendo as well as the Overture tool is built on top of the Eclipse platform.

Chapter 1

Introduction

1.1 What is the Crescendo Tool?

The Crescendo tool was originally called the DESTTECS tool since it was produced in the European research project called DESTTECS (see Section 1.2 below). It supports collaborative modelling and simulation called *co-simulation*.

1.2 What was the DESTTECS Project?

The DESTTECS (Design Support and Tooling for Embedded Control Software)¹ project had a consortium of research groups and companies working on the challenge of developing fault-tolerant embedded systems [BLV⁺10]. The consortium focussed on developing design methods and tools that bridge the gap between the disciplines involved in designing an embedded system: systems, control, mechanical and software engineering, for example. DESTTECS aimed to develop methods and tools that combine Continuous-Time (CT) models with Discrete-Event (DE) controller models through co-simulation to allow multi-disciplinary modelling, including modelling of faults and fault tolerance mechanisms. The analysis of these effects at every stage in a design process will help to build more dependable real-time embedded systems. The DE modelling is carried out using the Vienna Development Method and its support tool Overture (see Section 1.3 below). The CT modelling is carried out using bond graphs and its support tool 20-sim (see Section 1.4 below).

1.3 What is the Vienna Development Method?

The Vienna Development Method (VDM) is one of the longest established model-oriented formal methods for the development of computer-based systems and software [BJ78, Jon90, FLV08]. It consists of a group of mathematically well-founded languages for expressing system models during early design stages, before expensive implementation commitments are made. VDM has a

¹See www.destecs.org.



strong record of industrial application, in many cases has been used by practitioners who were not specialists in the underlying formalism or logic [LH96, CCFJ99, KN09]. Experience with the method suggests that the effort expended on formal modelling and analysis can be recovered in reduced rework costs arising from design errors.

VDM models are expressed in a specification language (VDM-SL) which supports the description of data and functionality [ISO96, FL98, FL09]. Data are defined by means of types built using constructors that define structured data and collections such as sets, sequences and mappings from basic values such as Booleans and natural numbers. These types are very abstract, allowing you to add any relevant constraints using data type invariants. Functionality is defined in terms of operations over these data types. Operations can be defined implicitly by preconditions and postconditions that characterize their behavior, or explicitly by means of specific algorithms. An extension of VDM-SL, called VDM++, supports object-oriented structuring of models and permits direct modelling of concurrency [FLM⁺05]. A further extension to VDM++, called VDM Real Time (VDM-RT²), includes support for discrete time models [MBD⁺00, VLH06]. All three VDM dialects are supported by Overture.

1.4 What are Bond Graphs?

Bond graphs are directed graphs in which the vertices are submodels and the edges, called *bonds*, denote the ideal (or idealised) exchange of energy. Entry points of submodels are called *ports*. The exchange of energy through a port (p) is always described by two implicit variables, effort ($p.e$) and flow ($p.f$). The product of these variables is the amount of energy that passes through the port. The meaning of these two variables depends on the physical domain (examples include voltage and current, and force and velocity).

The 20-sim tool supports the creation and simulation of models that can be represented in a variety of forms, including basic bond graphs; collections of differential equations describing the behaviour of nodes; and iconic diagram. Although the tool is commercial, all the model libraries are open source. The package supports mixed-mode integration techniques to allow the modelling and simulation of computer controlled *physical systems* that contain Continuous-Time as well as Discrete-Event elements. The level of complexity of many modern controllers means that discrete-event elements are better modelled using a rich formalism such as VDM. The 20-sim package supports the connection of external software both for model construction and simulation (Discrete-Event, Continuous-Time or hybrid), and this connection is exploited in providing support for co-simulation.

1.5 Related Tools

The Crescendo tool is one in a family of tools with common shared code.

²Formerly called VDM In a Constrained Environment (VICE).



1.5.1 Overture

The **Overture** tool represents the *opening* of these tools. This tool is build on top of the Eclipse platform and it support all the VDM dialects: VDM-SL, VDM++ and VDM Real-Time (VDM-RT). Many different features are included but the emphasis is on validation of VDM models by interpretation of executable subsets. This also includes support for DE notation VDM-RT used inside the Crescendo tool. Users who are only interested in Discrete Event (DE) modelling using one or more of the VDM dialects should use this Overture tool.

1.5.2 Crescendo

The **Crescendo** tool is a *gradual increase* of the Overture tool in the sense that it has all the support present inside Overture and in addition it supports collaborative modelling and simulation with a combination of the DE notation VDM-RT and the Continuous Time (CT) simulation by the 20-sim tool³. This is particular useful for those who are interested in modelling and validation of embedded control systems including modelling of the physical plants to be controlled. Users who are interested in both DE and CT modelling in a collaborative fashion should use this.

1.5.3 Symphony

The **Symphony** tool is an ,extended musical composition carried out by a large band (i.e. a System of Systems). This tool is developed in the COMPASS project⁴ and it supports the COMPASS Modelling Language (CML). CML is a combination of VDM and CSP/Circus. This tool provides validation using execution of an executable subset and test automation using the RT Tester tool but also formal verification using model checking (partially supported by the FORMULA model checker) and theorem proving (partially supported by the Isabelle theorem prover). Users who are interested in modelling and analysis of Systems of Systems should use this.

1.6 Structure of this User Manual

This user manual explains how to use the Crescendo IDE for developing collaborating models (co-models) and analyse them. In essence it is structured in 2 parts: The first part provides the basics for getting started using the Crescendo tool and the second part act as a reference manual for the Crescendo tool. The first part contains Chapter 2 introducing the basic Crescendo concepts⁵; Chapter 3 explaining how to get hold of the software; and finally Chapter 4 with information about how to quickly get started using the Crescendo tool with existing models that can be imported directly. Afterwards the second part also contains four chapters explaining the main possibilities

³This tool was originally developed in the DEST ECS (Design Support and Tooling for Embedded Control Software) project. This was partially supported by EU as project number 248134 under the embedded system design area.

⁴This is an acronym for “Comprehensive Modelling for Advanced Systems of Systems” project number 287829 under the EU FP7 programme.

⁵Appendix A provides a complete list of the Crescendo common concepts.



in the Crescendo tool. Chapter 5 explains the different views in the Crescendo tool, its editors and its way of handling projects. Afterwards, Chapter 6 explains the co-simulation possibilities. This is followed by Chapter 7 which provides information about how co-simulation can be extended with exploring the candidate design space. Finally this part is completed in Chapter 8 explaining the post-analysis possibilities in particular relevant in an exploration setting.

Chapter 2

Basic Crescendo Concepts

The Crescendo tool allows you to define and run a co-simulation. To get a basic understanding of the tool we first need to define some concepts. We will use a popular description of these concepts that might not be completely correct but will, hopefully, enhance the understanding of beginning Crescendo users.

2.1 Models

It starts with models. Models are a more or less abstract representation of a system or component of interest. In Crescendo we use *Continuous-Time models (CT models)* and *Discrete-Event models (DE models)*. CT models are models that describe real physical systems. These models describe the behaviour of physical systems at any desired time. DE models typically describe computer systems that run at predetermined time steps. Between these time steps nothing happens.

2.2 Simulation

Continuous-Time models can be created and simulated in 20-sim. This tool will simulate CT models with as many small time steps as required to get accurate results. Sometime the accuracy is violated. The tool will then step back and use smaller time steps until the required accuracy is met. This is called a *Continuous-Time simulation*. A Continuous-Time simulation is therefore always characterized by the accuracy of the simulation and the time steps taken. Discrete-Event models can be created and simulated in Overture/VDM. This tool will simulate Discrete-Event models with predetermined discrete time steps. This is called a *Discrete-Event simulation*. There is no accuracy issues involved and therefore no backstepping is required.

The properties of a model that affects its behaviour, but which remains constant during a simulation are called *parameters*. Examples of parameters are the `height` of a watertank with varying `waterlevel` or the `mass` of a car with varying `speed`. A *variable* is a property of a model that may change during a given simulation. Examples of variables are the varying `waterlevel` of a watertank or the varying `speed` of a car.



2.3 Co-Simulation

A *co-simulation* is a combined simulation of a Continuous-Time model and a Discrete-Event model in separate tools. The Crescendo tool allows you to run Discrete-Event models in VDM and Continuous-Time models in 20-sim and exchange information between VDM and 20-sim during run time. Because the notion of a model in a co-simulation may lead to misinterpretations, we will use the following definitions:

constituent model: the CT submodel or the DE submodel of a co-simulation.

co-model: a model comprising two constituent models (a DE submodel and a CT submodel).

2.4 Contract

The description of the communication between the constituent models of a co-model is called the *contract*. A contract typically describes the variables that are shared between the Continuous-Time model and the Discrete-Event model. An example of a *shared variable* is the `waterlevel` that is calculated in the Continuous-Time model and sent to the Discrete-Event model where it is used to calculate the response of a water level controller.

In most cases a Continuous-Time model and a Discrete-Event model will use similar parameters. For the watertank example such a parameter may be the maximum water level. In the Continuous-Time model this parameter indicates the height at which a sensor is placed and in the Discrete-Event model this parameter may indicate a property of the water level controller. To prevent different values to be used in the Continuous-Time model and Discrete-Event model, we may share this parameter in the contract. This is called a *shared design parameter*.

Chapter 3

Getting Hold of the Software

3.1 Requirements

The Crescendo tool suite can be downloaded as a single installation package from the Crescendo website. The package contains a full installation for Overture/VDM, 20-sim and the Crescendo tools. Overture/VDM and the Crescendo tools are open source tools and will run on any computer. However, 20-sim is a commercial tool that will run as a viewer on any computer. If you want to build your own models in 20-sim and store them, you will need a license (see Section 3.2). In order to install the package of the Crescendo package you need to have:

- Windows platform (XP / Vista / 7 / 8)
- At least 256 MB memory
- At least 200 MB free disk space
- An x86 compatible CPU

3.2 Installation

3.2.1 Combined Installer

First-time users are advised to use the combined installer that will install the Crescendo tool and that will also include the Overture/VDM features and the 20-sim tool on your computer. You can download the installer from the Crescendo website:

`http://www.crescendotool.org`

During installation the main installer will pause. A second installer will then guide you through the installation of 20-sim. Once 20-sim is installed, the Crescendo installer will continue. Both the Overture/VDM tool as well as the Crescendo tool are released as open source under the GNU General Public License v3.0.



3.2.2 License

Both VDM and the Crescendo tools are open source and do not require an additional license. 20-sim is a commercial tool that will run in Viewer mode on any computer. This means that you can only run and edit models! If you want to save model changes, you will need a license. You can send an email to Controllab <mailto://info@controllab.nl> to get a trial license.

3.2.3 Manuals

VDM: To help you work with VDM the VDM language manual [LLB⁺13] and the Overture user guide [LLJ⁺13]¹.

20-sim: To help you work with 20-sim, you can visit the website [Con13] or look at the reference manual [Kle09]².

3.3 20-sim Standalone

The Crescendo installer will install the 20-sim Viewer on your computer. The 20-sim Viewer will allow you to run and edit all the Crescendo example models, but does not allow you to save them. When 20-sim is opened, the license dialog will be opened.



Figure 3.1: License for 20-sim.

¹Both of these and more manuals can be found at <http://overturetool.org/?q=Documentation>.

²See also <http://www.20sim.com/support/movies.html> or <http://www.20sim.com/downloads/files/20sim43GettingStartedManual.pdf>.



- You have to click the *Close* button to continue.

Unless you have purchased a license during model editing and simulation, the 20-sim Viewer will annoy you with a message:



Figure 3.2: Message if you do not have a license for 20-sim.

- You have to *click on top* of this message to continue.

If you feel the Viewer is too annoying or want to store models, you have to ask Controllab for trial license.



Chapter 4

Quick Start with Crescendo

To help you get started with Crescendo this chapter gives you step by step instructions how to configure the software, get a basic `WaterTankPeriodic` example running and create your own simple project.

4.1 Opening Crescendo

- **Open Crescendo** from the **start menu**. You will see a splash screen when the program opens and a dialog prompting you to give a location for the workspace as in Figure 4.1.

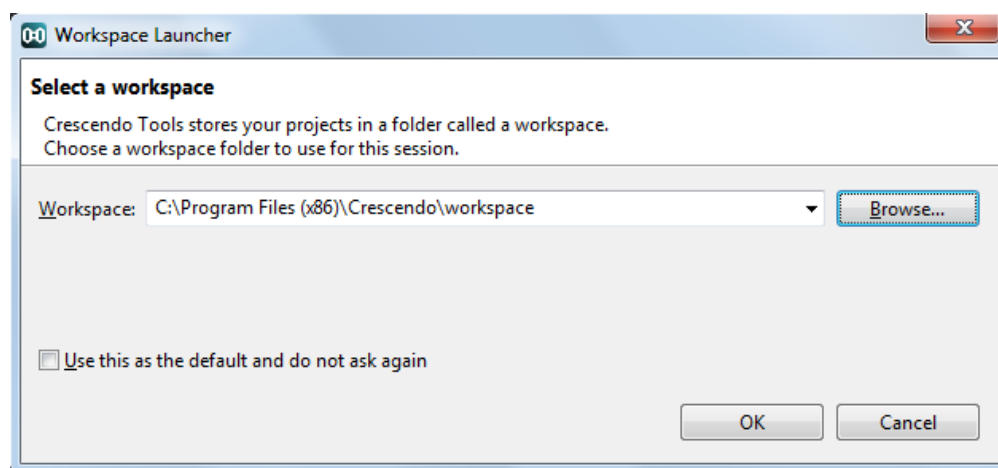


Figure 4.1: The Crescendo Workspace Location Screen.

- **Enter a location** where you have both read and write access. The program should respond by opening with a welcome screen as shown in Figure 4.2.

When the welcome screen is closed the standard Crescendo perspective as shown in Figure 5.1 with different views explained further in Chapter 5.

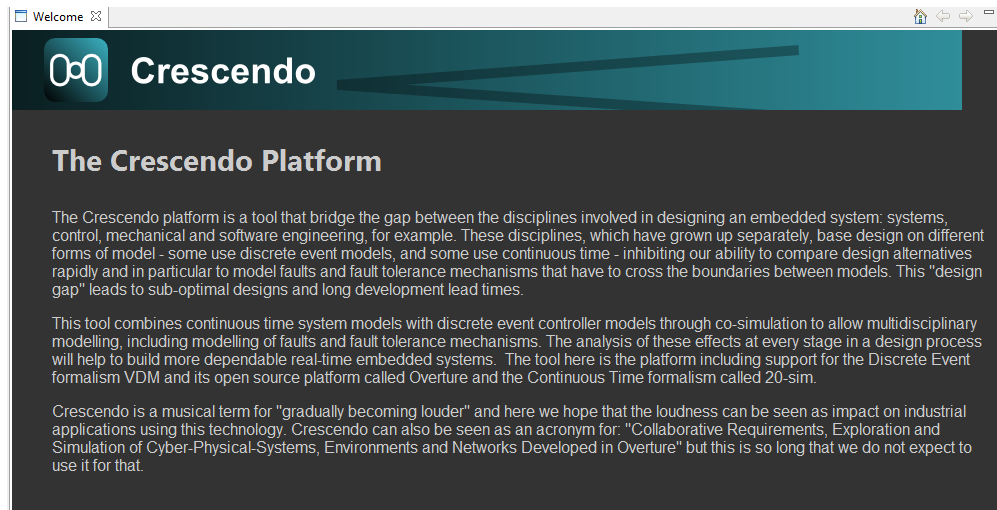


Figure 4.2: The Crescendo Welcome Screen.

4.2 Opening a Project

- From the *File* menu choose “*File and then Import*”.
- Select “*General and Existing Projects into Workspace*” and click *Next* and get a window similar to Figure 4.3.
- *Crescendo Examples* and select at least the `WatertankPeriodic` example.
- Click *Finish* to import the selected project(s).

4.3 Running a Project

Now (at least) the `WaterTankPeriodic` project should be visible.

- **Click on the `WaterTankPeriodic` project entry to select it.**
- Press the *Debug button* (if you have multiple projects loaded, you have to select the `WaterTankPeriodic` project first, by clicking the black triangle at the right of the Debug button)

Now a co-simulation will start. The 20-sim editor (showing the Continuous-Time model) will be opened as shown in Figure 4.4, the 20-sim Simulator (showing the plot of the Continuous-Time part of the simulation) will be opened and the 3D animator (showing an animation of the watertank) will be opened as shown in Figure 4.5. In addition graphs for selected variables during the simulation is shown as in Figure 4.6.

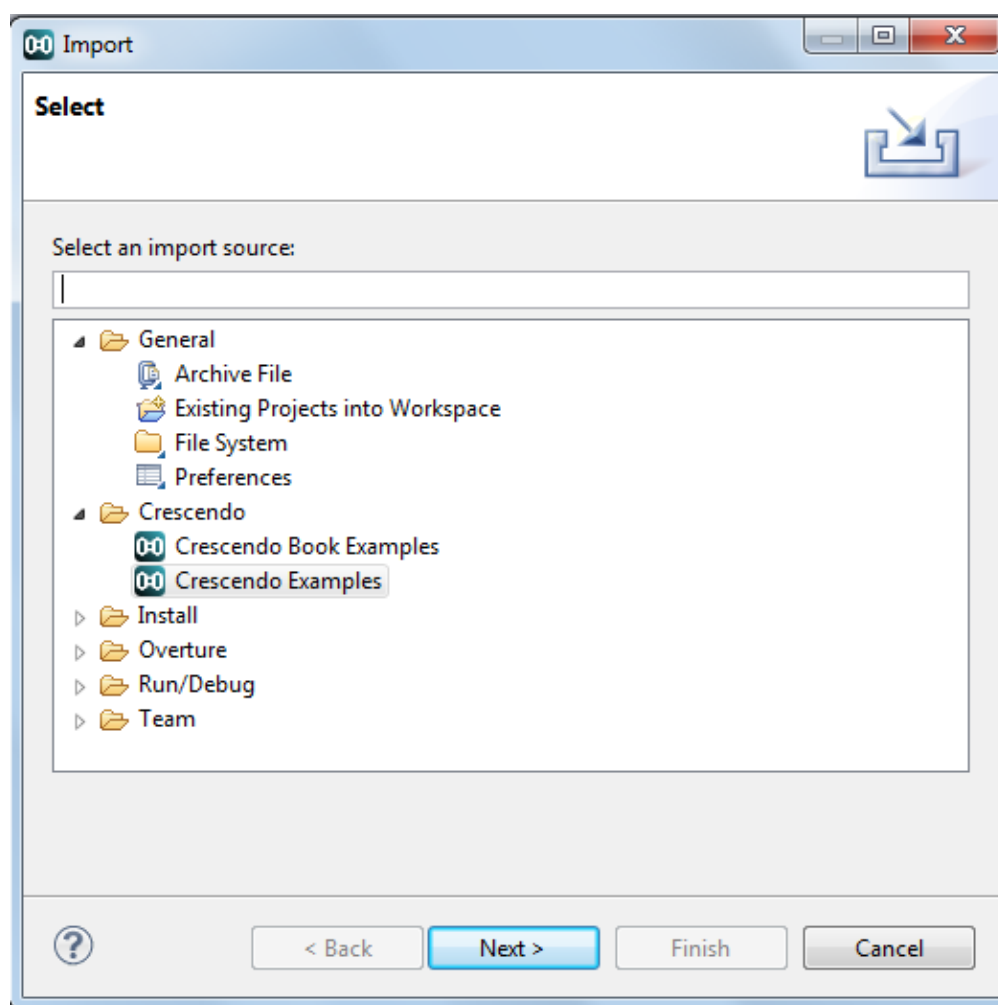


Figure 4.3: Dialog for Importing Crescendo Examples

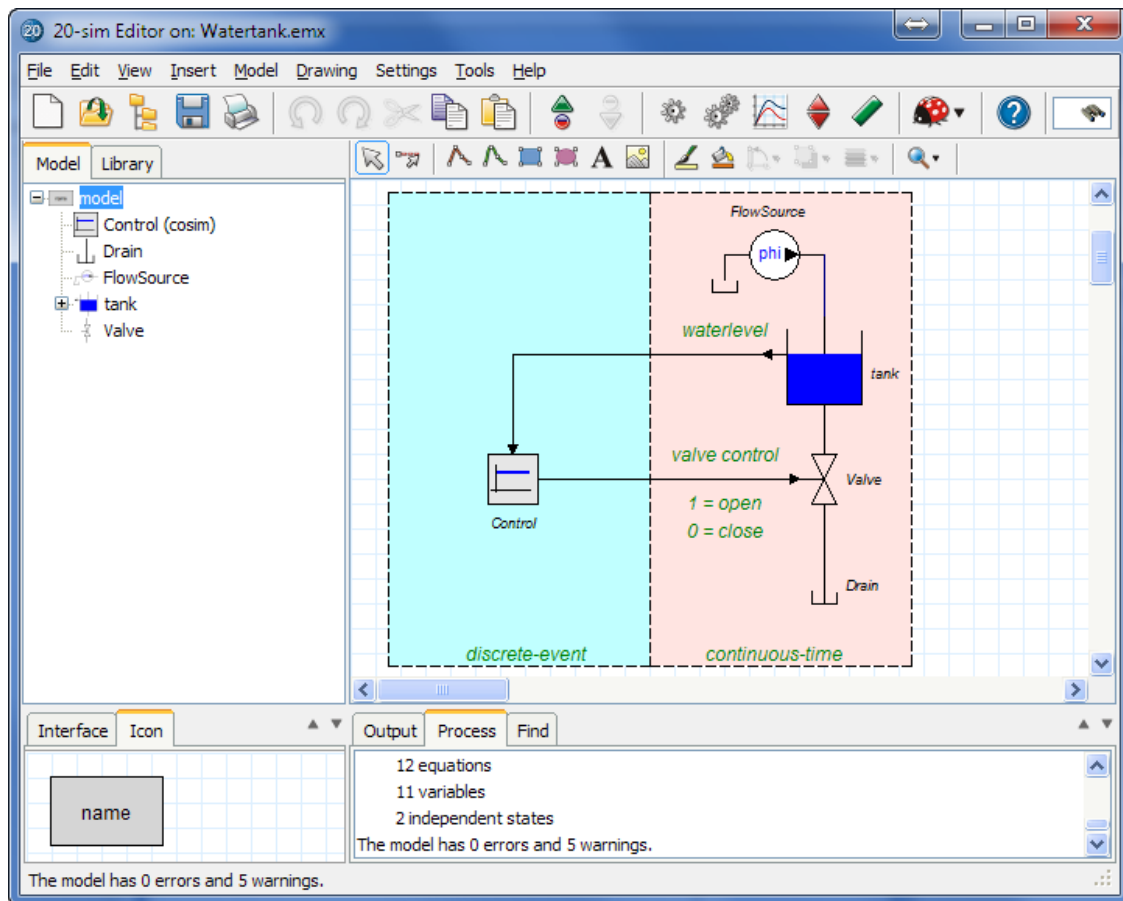


Figure 4.4: The 20-sim Editor contains the continuous-time WaterTankPeriodic model.

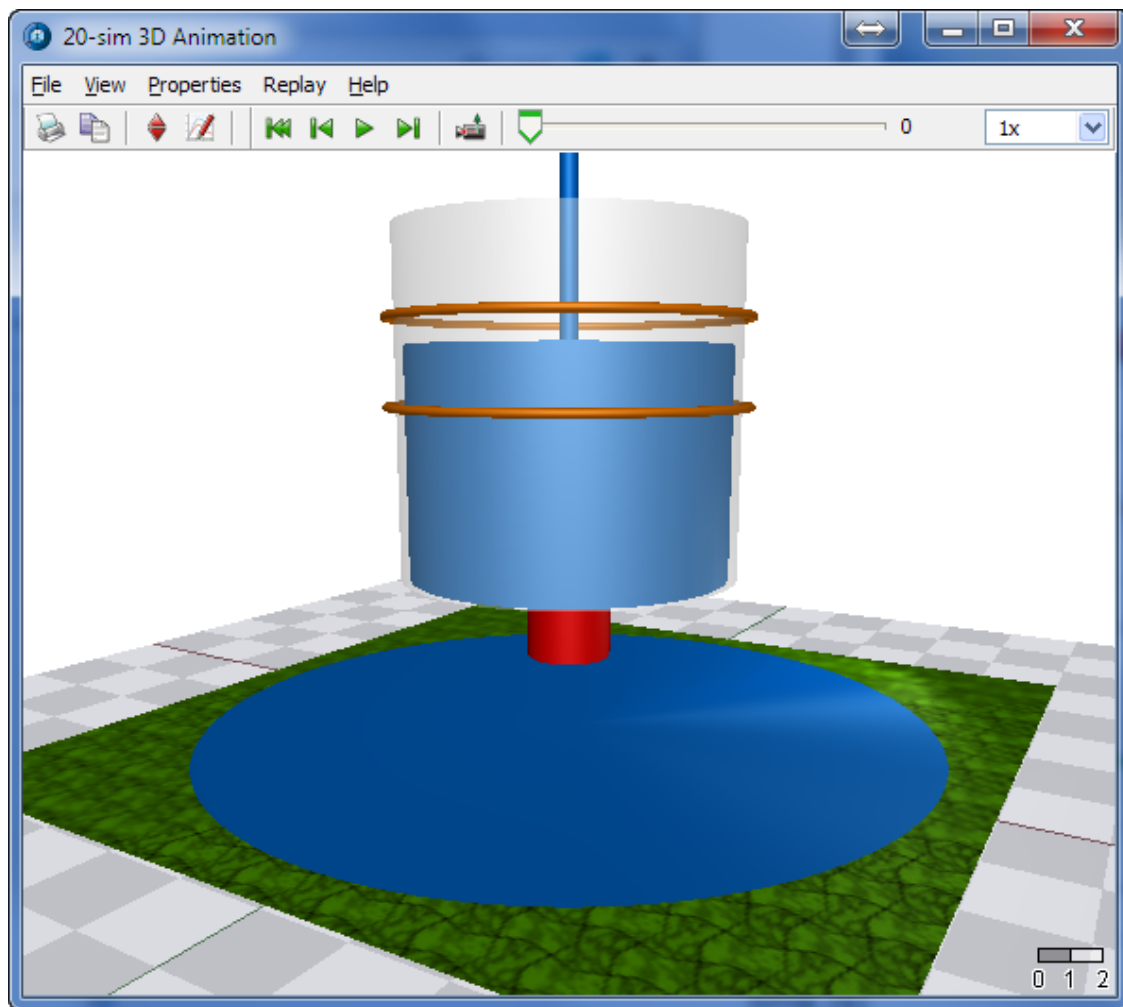


Figure 4.5: 20-sim can also show simulation results in a 3D animation.

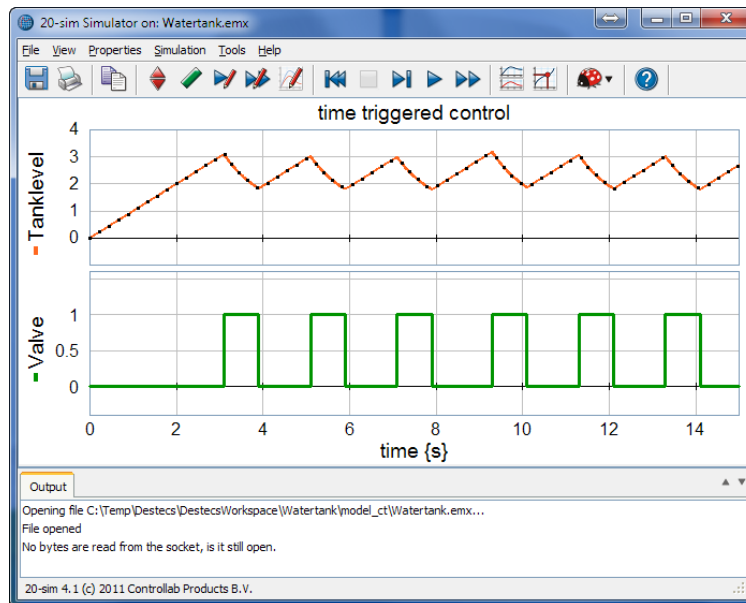


Figure 4.6: The 20-sim Simulator shows the co-simulated plots.

Chapter 5

Editors and Management of Projects

5.1 The Crescendo Workbench

Eclipse is an open source platform based around a workbench that provides a common look and feel to a large collection of extension products. Thus, if a user is familiar with one Eclipse product, it will generally be easy to start using a different product on the same workbench. The Eclipse workbench consists of several panels known as views as shown in Figure 5.1. A collection of panels is called a perspective. The figure below shows the standard Crescendo perspective. The Crescendo perspective consists of a set of views for managing Crescendo projects and viewing and editing files in a project. Different perspectives are available in Crescendo based on the task that you are doing. In the subsections below the different views of the standard Crescendo perspective will be presented.

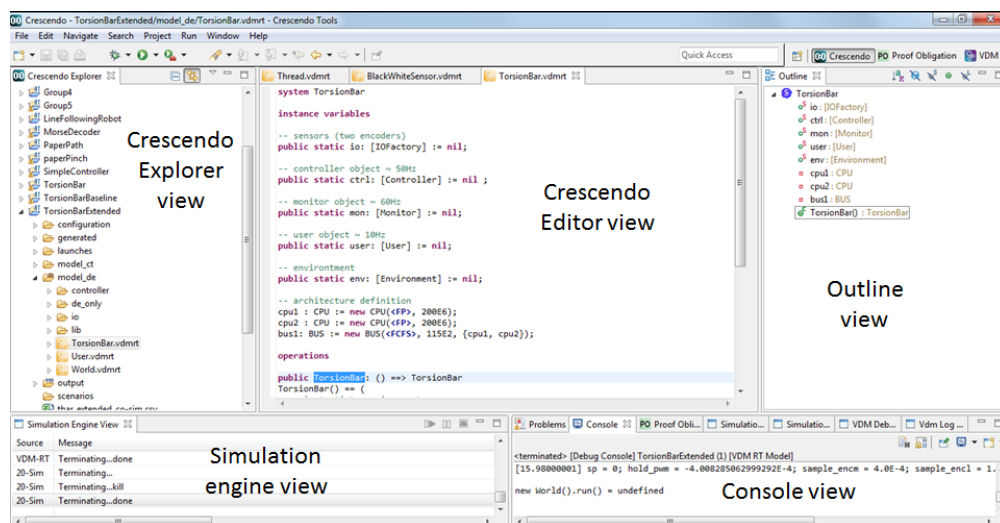


Figure 5.1: Standard Crescendo Perspective.



5.1.1 Explorer View

The Crescendo *Explorer view* lets you create, select, import, export and delete Crescendo projects and navigate between the files in these projects. It is also from this view that deleting existing files and adding new files to existing projects is enabled. In Figure 5.2 the kind of contents inside one project is illustrated.

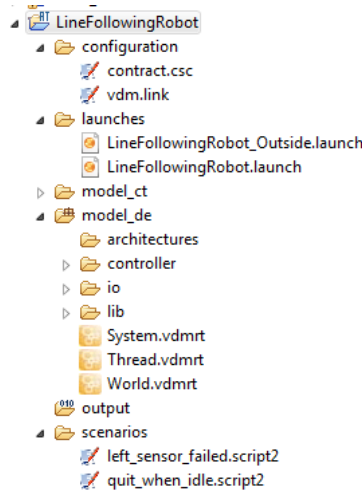


Figure 5.2: The Crescendo explorer view.

5.1.2 Editor View

The Crescendo *Editor View* allows you to edit VDM files, Contracts and Scenarios and it highlights the different keywords. Figure 5.3 shows how a Crescendo contract looks in the editor (and in a different pane a VDM-RT file).

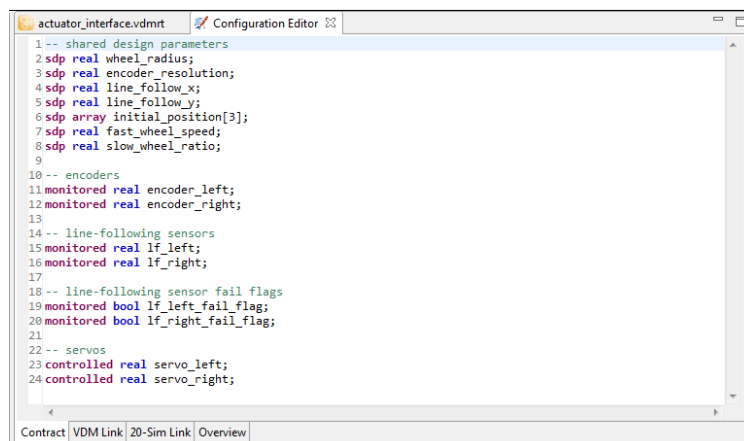


Figure 5.3: The Editor View.



5.1.3 Outline View

The Crescendo *Outline view*, on the right hand side of Figure 5.1, presents an outline of the file selected in the editor. This view displays any declared VDM definitions such as their state components, values, types, functions and operations. The type of the definitions are also shown in the outline view. The Outline view is at the moment only available for the VDM models of the system. In the case another type of file is selected, the message: “An outline is not available” will be displayed. Figure 5.4 shows an extract of the outline of a class called `System`.

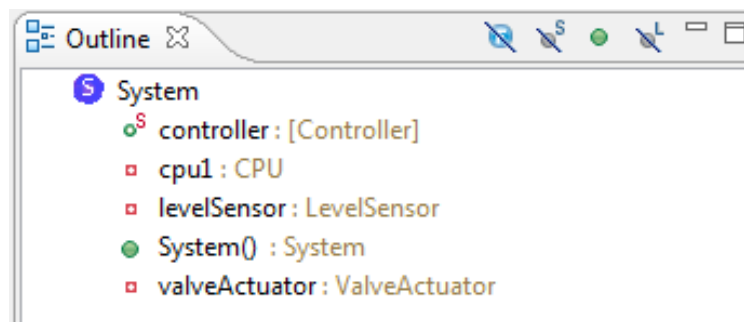


Figure 5.4: The outline view showing the composition of the `System` VDM-RT class.

The type of each definition is also shown in the view and the colour of the icons in front of the names indicates the accessibility of each definition. Red is used for private definitions, yellow for protected definitions and green for public definitions. Triangles are used for type definitions, small squares are used for values, state components and instance variables, functions and operations are represented by larger circles and squares, permission predicates are shown with small lock symbols and traces are shown with a “T”. Functions have a small “F” superscript over the icons and static definitions have a small “S” superscript. Record types have a small arrow in front of the icon, and if that is clicked the fields of the record can be seen. Figure 5.5 illustrates the different outline icons. At the top of the view there are buttons to filter what is displayed, for instance it is possible to hide non-public members.

Clicking on the name of a definition in the outline will navigate to the definition and highlight the name in the Editor view (see Section 5.1.2).

5.1.4 Simulation Engine View

The Crescendo *Simulation Engine View*, located in the lower left part of the environment is showing the evolution of a co-simulation. This is done by monitoring the interaction between the VDM-RT Discrete-Event simulation, the 20-sim Continuous-Time simulation and the engine. This view has two columns, the first one is specifying the source of the message and the second one the progress of a co-simulation in seconds and percentages of completion. An extract of this is shown in Figure 5.6.

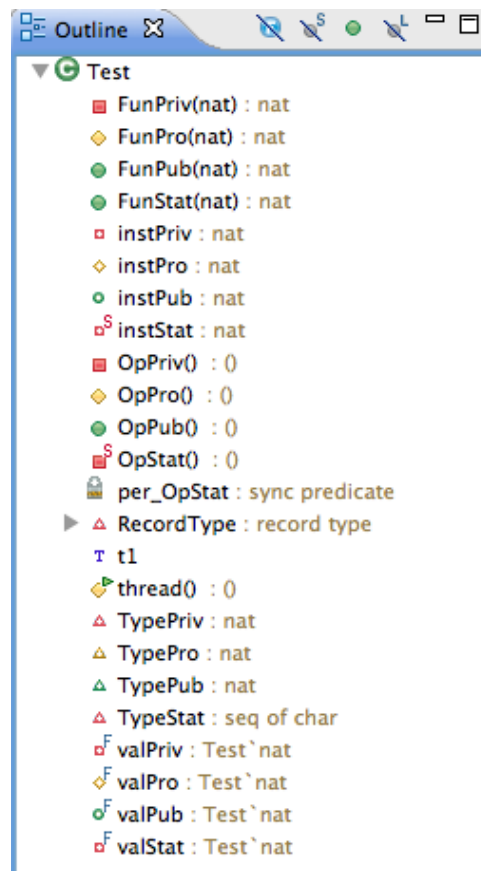


Figure 5.5: Icons in the Outline View

Simulation Engine View	
Source	Message
All	Simulation time: 16,200000 seconds / Completed: 81 %
All	Simulation time: 16,400000 seconds / Completed: 82 %
All	Simulation time: 16,600000 seconds / Completed: 83 %
All	Simulation time: 16,800000 seconds / Completed: 84 %
All	Simulation time: 17,000000 seconds / Completed: 85 %
All	Simulation time: 17,200000 seconds / Completed: 86 %
All	Simulation time: 17,400000 seconds / Completed: 87 %

Figure 5.6: The Crescendo Simulation Engine view.

5.1.5 Console View

The Crescendo *Colsole View*, located in the lower right part of the environment is acting as a console showing any output from the DE-part of the co-simulation. An extract of this is shown in



Figure 5.7.

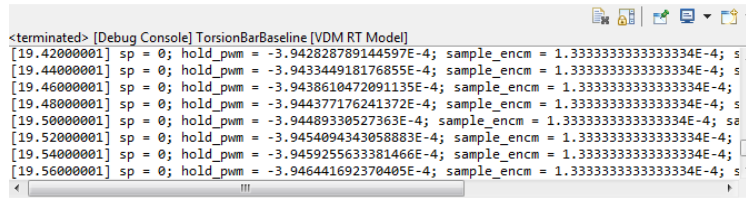


Figure 5.7: The Crescendo Console view.

5.2 Handling Projects

All data that is necessary for a co-simulation (e.g., models, contracts etc.) is stored in a Crescendo project. This section explains how to use the Crescendo tool to manage projects. Step by step instructions for importing, exporting and creating projects will be given.

5.2.1 Creating new Projects

Follow these steps in order to create a new Crescendo project:

- Create a new project by choosing *File* and *New* and *Project* and *Crescendo project*.
- Type in a project name.
- Click the button “*Finish*” (see Figure 5.8).

You can create projects in the Crescendo tool. The highlighted project is the project that is currently selected.

5.2.2 Importing Projects

Follow these steps in order to import an already existing Crescendo project.

- Right-click the *Explorer View* (see Section 5.1.1 above) and select *Import*.
- Select either the standard *Crescendo* or the *General - Existing Projects into Workspace* part. Using the *Crescendo* one enables you to import the standard Crescendo examples or the examples used in the book about this [FLV13]. Using the *General* entry you can import anything else that has been produced by someone and exported for your use.
- Click *Next* to proceed.

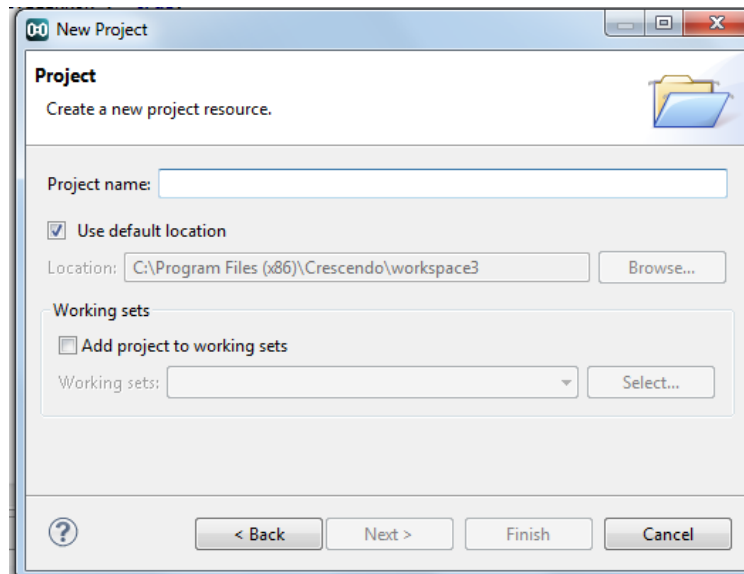


Figure 5.8: Create project dialog.

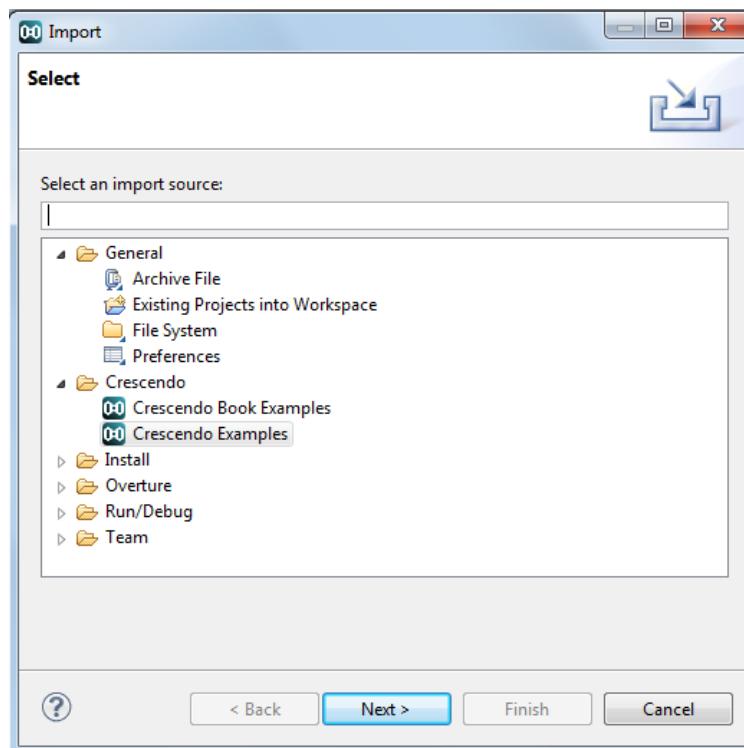


Figure 5.9: Import project dialog.



- If one of the *Crescendo* options is taken a list of the potential projects to import will appear. Otherwise if the *General* one was selected you shall select the the radio button “*Select root directory*” if the project is uncompressed. Otherwise select the the radio button “*Select archive file*” if the project is contained in a compressed archive file. Afterwards, use the *Browse* button to locate the project.
- A compressed archive file may contain multiple projects. Select the projects that you want to import as shown in Figure 5.10.
- Click the *Finish* button. The imported project will appear on the Crescendo explorer view.

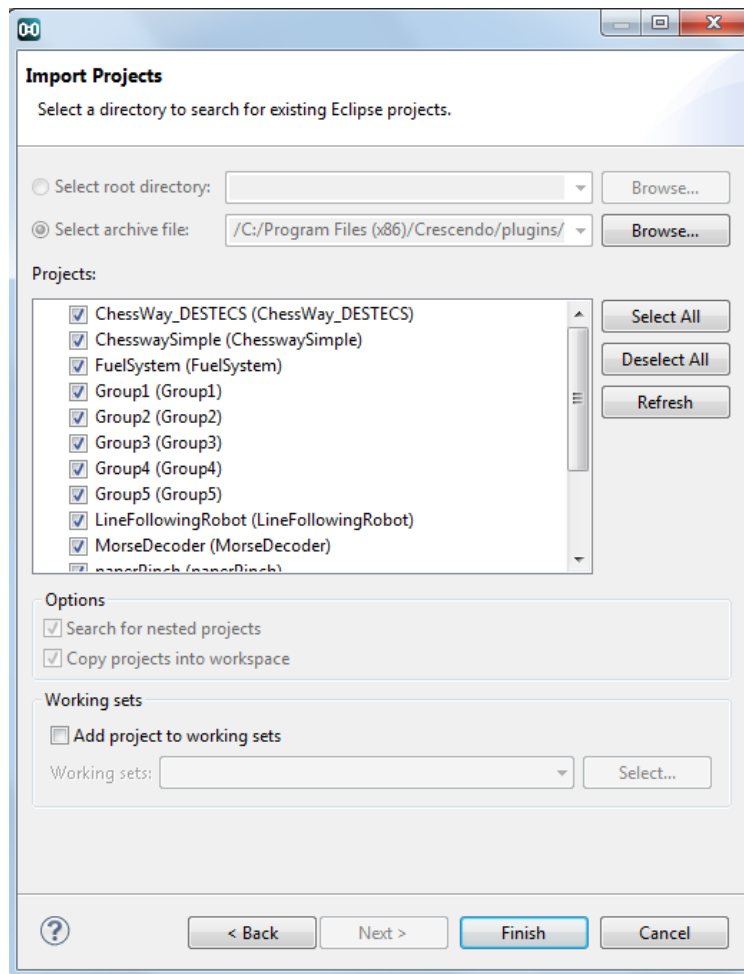


Figure 5.10: Import projects.

5.2.3 Exporting Projects

Follow these steps in order to export a Crescendo project:

- Right click on the target project and select *Export*, followed by *General* and *Archive File*. See Figure 5.11 for more details.
- Click *Next* to proceed.

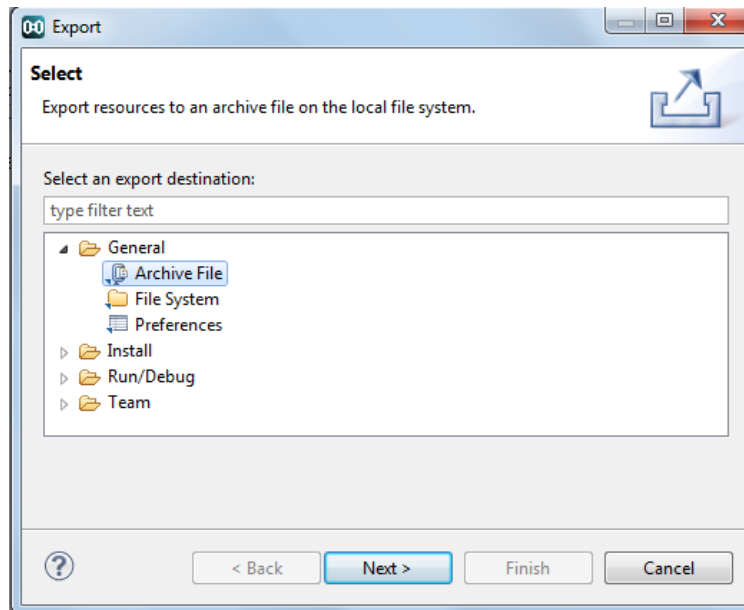


Figure 5.11: Select an output format for the exporting process.

5.3 Managing Contracts

To connect the Continuous-Time model and Discrete-Event model we have to define a *contract*. The contract also needs to be linked into DE and CT elements respectively. In order to show the co-simulation tool how to link the elements from the contract to the DE and CT models respectively a link-file must be present for each co-model. This is stored in a *vdm.link* part.

5.3.1 Creating a new Contract File

Right click on the project that is going to contain the contract file. Select *New* and *Crescendo New Contract* as shown in Figure 5.12.

- A new window will pop up. Choose a `contract` name and click on the *Finish* button to end the process.

After following these steps a new file named `contract.csc` will appear under the `configuration` folder contained in the project tree. The contract can be viewed in the editor as shown in Figure 5.13.

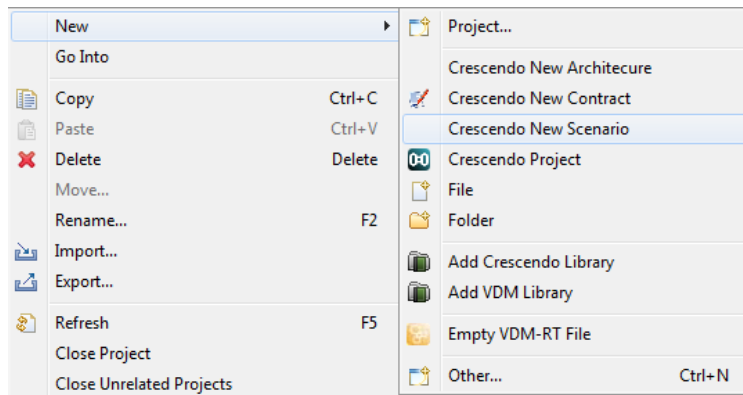


Figure 5.12: Choosing a new Crescendo contract.

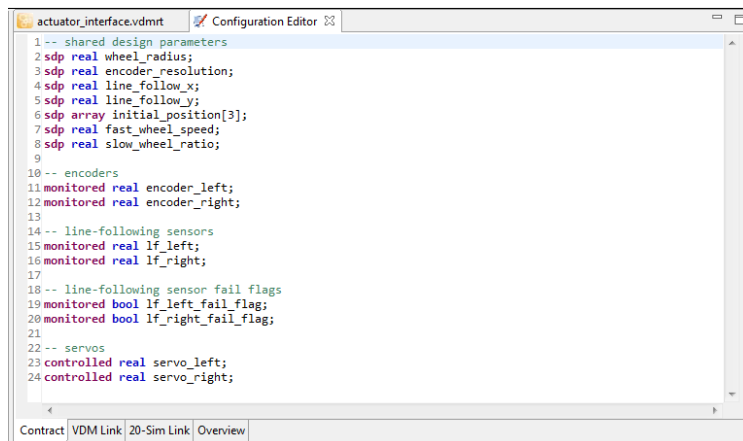


Figure 5.13: The Editor with a new contract.

5.3.2 Contents of a Contract

A contract between a CT and a DE model can contain the following kind of information:

Design parameters: These are typically values which indicate the properties of components (e.g. size, weight, temperature). A designer would like to explore different values of these parameters in order to find an optimal solution to the challenge he is working on. The actual values for the shared design parameters are set outside the contract in a separate file.

Variables: The variables are the active interface between the CT and DE models so these indicate the variables that change during one simulation. Variables typically represent sensor readings and signals to actuators.

Events: Events can be triggered in the CT world. They will stop the simulation before the allowed time slice is completed. The co-simulation engine will then allow the DE simulator to take action but only until the point where the event has been raised. The events are used in the contract in order to support event-based triggering and not just time-triggered scheduling.



The syntax for contracts follow the following rules:

$\langle contract \rangle ::= parameters \mid variables \mid events;$

$\langle parameters \rangle ::= 'shared_design_parameter' \text{ type identifier } ';' \mid 'sdp' \text{ type identifier } ';' ;$

$\langle variables \rangle ::= kind \text{ type identifier } ';' \mid kind 'matrix' \text{ identifier shape } ';' ;$

$\langle shape \rangle ::= '[' 'integer' ',' 'integer' ') * '[' ;$

$\langle events \rangle ::= 'event', identifier, ';' ;$

$\langle type \rangle ::= 'real' \mid 'bool' ;$

$\langle identifier \rangle ::= initial_letter (following \text{ letter})* ;$

$\langle kind \rangle ::= 'monitored' \mid 'controlled' ;$

$\langle value \rangle ::= float \mid boolean_literal ;$

$\langle boolean \text{ literal} \rangle ::= 'true' \mid 'false' ;$

In the following listing, an extract from the contract file provided with the WaterTank-Periodic example is shown.

```
-- Shared design parameters
sdp real maxlevel;
sdp real minlevel;

-- Monitored variables (seen from the DE controller)
monitored real level;

-- Controlled variables (seen from the DE controller)
controlled bool valve;
```

Matrices

It is also possible to exchange matrices between DE and CT models. To be able to do this, a matrix needs to be declared in the contract. The adopted syntax is similar to 20-sim, where the shape of the matrix is indicated by a sequence of integers $[m_1, \dots, m_n]$. For example, to declare a 2x2 matrix named *M* which is **monitored** the following must be added to the contract:

```
monitored matrix M[2, 2];
```



In VDM matrices of “n” dimensions ($m_1 * \dots * m_n$) are represented as (**seq of ... seq of real**). So a 2x2 matrix is represented as a (**seq of seq of real**).

The contract matrix variables are linked in the same manner as any other variable but the target variable needs to be of the correct type, in our case **seq of seq of real**. At the VDM level this would typically be declared as:

```
instance variables
```

```
M: seq of seq of real := [[0.0,0.0],[0.0,0.0]];
```

Arrays

Arrays which are limited to one dimension can be declared in the same style:

```
monitored array position[3];
```

5.3.3 Error Detection in the Contract/Link File

A static error check is performed every time the contract or the link file are saved. This is a cross-file consistency check which resolves if all the variables/events declared in a contract are also present in the link and vice-versa. Crescendo will prevent the launch of projects with consistency errors between the contract and link files but there is the possibility to turn this protection off by un-checking the referent preference (accessible in the menu “*Windows* followed by *Preferences*”). This is illustrated in Figure 5.14.

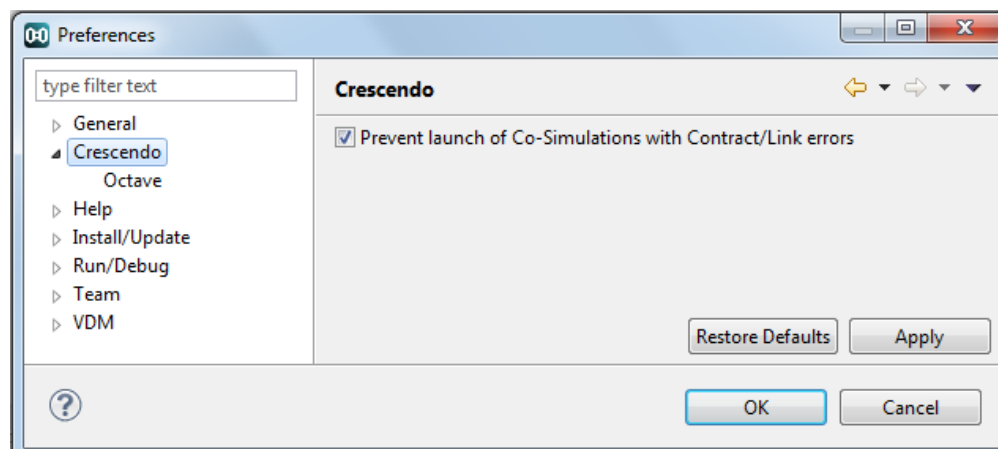
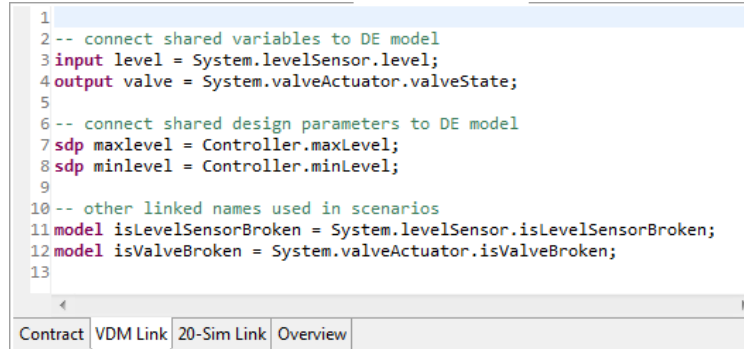


Figure 5.14: Launching with or without contract or link errors.

5.3.4 Managing the Link Files

The link file is automatically created when you start a new project. You can edit the link file by selecting the configuration folder in the project tree.

Expand the project and configuration folder and *select the file* `vdm.link`. Figure 5.15 shows the editor with the contents of the link file.



```

1
2 -- connect shared variables to DE model
3 input level = System.levelSensor.level;
4 output valve = System.valveActuator.valveState;
5
6 -- connect shared design parameters to DE model
7 sdp maxlevel = Controller.maxLevel;
8 sdp minlevel = Controller.minLevel;
9
10 -- other linked names used in scenarios
11 model isLevelSensorBroken = System.levelSensor.isLevelSensorBroken;
12 model isValveBroken = System.valveActuator.isValveBroken;
13

```

Figure 5.15: Expand the configuration folder to see the link file.

Contents of a link file

The syntax of a link file is a sequence of link definitions (each definition is formed by an interface type, a qualified name, “=” sign and a qualified name) separated by line breaks. Here all the design parameters, the variables and the events from the contract must be present on the left-hand-side of each of these definitions. It is important to note that the link file may contain more links than required by the contract, this allows a DE model to be reused in different simulations where different contracts are used. Additionally, links can be made to variables that exist within the model in order to be able to reference them from a script (the keyword “**model**” is used for this purpose). The right-hand-side of all the “=” signs provide the names seen from the DE co-model side, e.g., the instance variables inside a system class in the VDM-RT model.

The syntax of these definitions are:

$\langle \text{vdmlink-file} \rangle ::= \{ \langle \text{interface} \rangle, \langle \text{qualified-name} \rangle, '=', \langle \text{qualified-name} \rangle, ';' \}$

$\langle \text{interface} \rangle ::= \text{'output'} \mid \text{'input'} \mid \text{'sdp'} \mid \text{'event'} \mid \text{'model'};$

$\langle \text{qualified-name} \rangle ::= \langle \text{identifier} \rangle, [\text{'.'}, \langle \text{identifier} \rangle];$

$\langle \text{identifier} \rangle ::= \langle \text{initial-letter} \rangle, \{ \langle \text{following-letter} \rangle \};$

Link File Parts

- **input** links one monitored variable in the contract with a instance variable in the DE model. The qualified name must start with the system class name.



- **output** links one controlled variable in the contract with an instance variable in the DE model. The qualified name must start with the system class name.
- **sdp** links a shared design parameter in the contract with a value in the DE model. The qualified name can start by any class name and the referenced value must be a “value” in VDM.
- **model** links a “name” and a variable in the VDM model. The name can then be used to reference the variable in scripts. The qualified name must start with the system class name.

The `vdm.link` file for the `WaterTankPeriodic` example looks as:

```
-- connect shared variables to DE model
input level = System.levelSensor.level;
output valve = System.valveActuator.valveState;

-- connect shared design parameters to DE model
sdp maxlevel = Controller.maxLevel;
sdp minlevel = Controller.minLevel;

-- other linked names used in scenarios
model isLevelSensorBroken =
    System.levelSensor.isLevelSensorBroken;
model isValveBroken =
    System.valveActuator.isValveBroken;
```

CT Model

On the 20-sim side, a link file is not used, but still, the variables/parameters need to be declared in a certain way inside the 20-sim model in order to carry out the co-simulation.

Variables used in the co-simulation, need to be in the **externals** field and marked as **global**. Depending if they are used as input or output they need to be marked **import** or **export** respectively. Example:

```
externals
  real global export level;
  real global import valve;
```

The parameters to be shared across the two models need to be marked with the keyword **shared**. Example:

```
parameters
  real aParam ('shared') = 5;
```



Events need to be marked using the **event** keyword this marks the variable that it used as return value of the event function to be an event variable. The keywords **eventdown** and **eventup** are used as in standalone 20-sim models. Example:

```
variables
  boolean minLevelReached ('event');
equations
  maxLevelReached = eventup(levelIn-maxlevel);
```

5.3.5 Contract Overview

An overview of the contract can be seen on the last tab of multi-editor as shown in Figure 5.16.

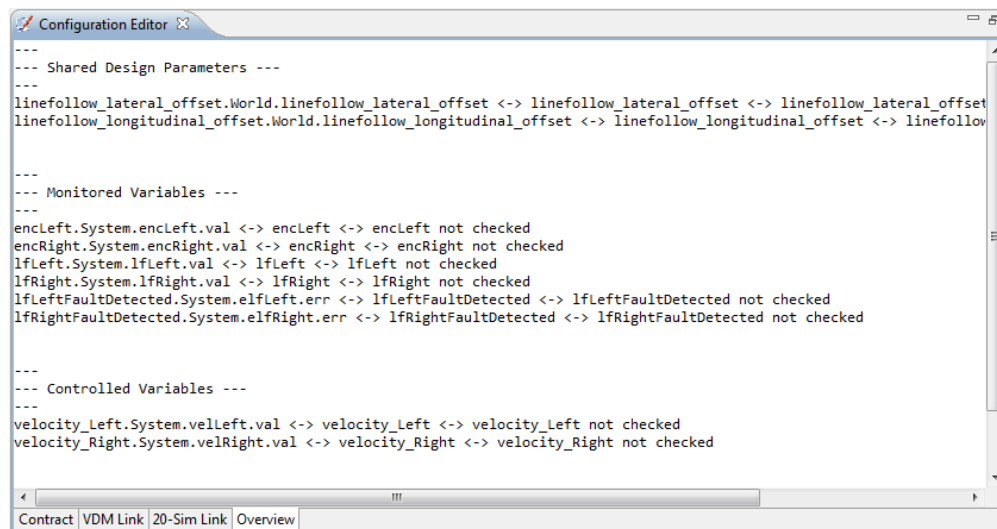


Figure 5.16: Overview of contract information.

In this view it is possible to see which variable from the DE side is connected to which contract variable and transitively to which CT variable. The form they are presented is:

VDM variable <-> Contract Variable <-> 20-sim variable

The “*not checked*” appears next to the 20-sim variables because at this moment is not possible to static check if the variables exist in the 20-sim model.


Chapter 6

Co-Simulation Possibilities

6.1 Debug Configuration

Before starting a co-simulation, a debug configuration must be created if a launch configuration is not already available. The purpose of this is to define where the Continuous Time and Discrete Event models are located, as well as the scenario file and the simulation time. In this section we will go through each pane in the debug configuration.

6.1.1 Creating a New Debug Configuration

- *Select the project* for which you want to create a Debug Configuration.
- Press the **small arrow** next to the debug icon  at the top of the the Crescendo Tool.
- A drop-down menu will appear, in which the option *Debug configuration* has to be selected, and as a consequence a new window as shown in Figure 6.1.
- Select the option *Co-Sim Launch and New Configuration*

Now a window will show up as Figure 6.2 where you can enter the settings of the debug configuration. We will describe all the tabs that can be configured before running a co-simulation.

6.1.2 Main Tab

The Main Tab is where the project to co-simulate is selected. This can be done by pressing the “Browse...” button. After selecting the wanted project, the DE model path is automatically filled since it is only possible to have one DE model in the `model_de` folder. Though the CT model path needs to be selected using the *Browse...* button. If a scenario should be used (see Section 6.3 for more information on scenarios), it is possible to select which one in the Simulation Configuration section. The total simulation time should be a number greater than zero to be able to run the co-simulation.

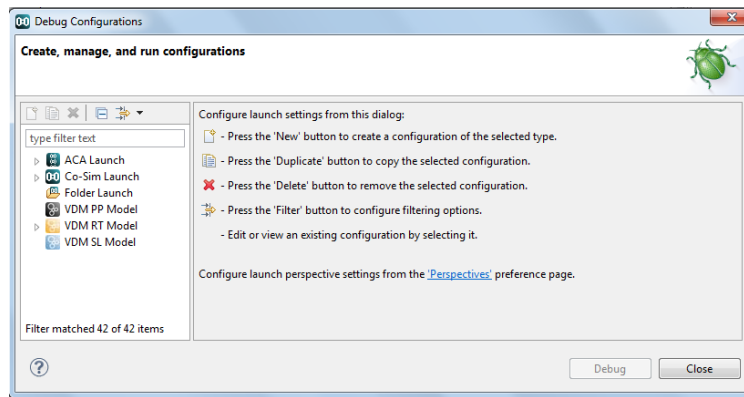


Figure 6.1: Select a new debug configuration.

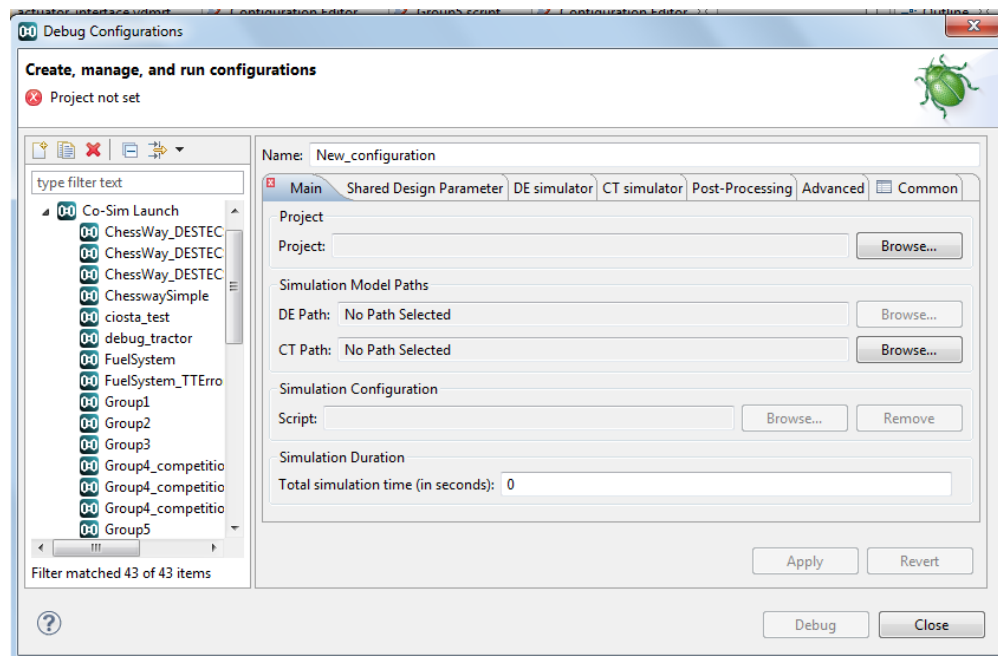


Figure 6.2: The Main tab of the Debug Configuration.

6.1.3 Shared Design Parameters Tab

An important feature of the debug configuration is the possibility to view and modify the *shared design parameters* of the co-simulation. This is configured in Figure 6.3.

In the *Shared Design Parameters* tab, a list of the parameters used in the simulation can be viewed. For the variables to appear for the first time the button “*Synchronize with contract*” needs to be pressed. Every time the shared design parameters are changed in the contract, the button must be pressed again in order to synchronize the view with the contract.

For the variables present in the table it is possible to decide which values they will have when

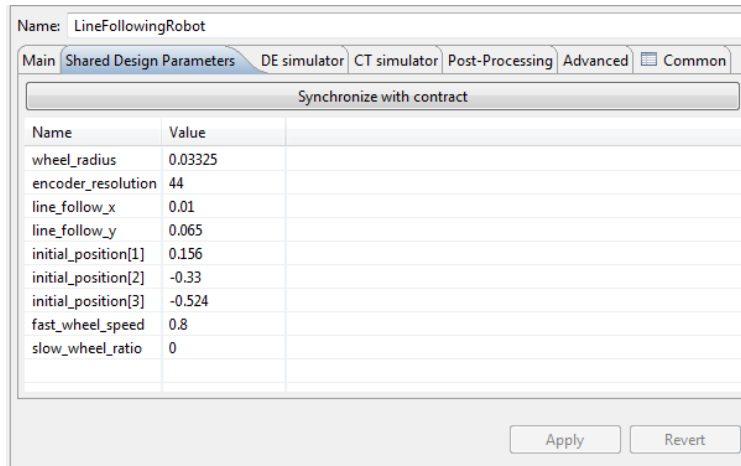


Figure 6.3: The Shared Design Parameters tab of the Debug Configuration.

the co-simulation starts. Figure 6.3 shows the Shared Design Parameters tab for a project that has an array[3] as a shared design parameter.

6.1.4 DE Simulator Tab

The *DE Simulator* tab is the tab where runtime options for the DE part of the model can be activated/deactivated. It is divided in 4 options groups:

Interpreting: These are options related with the interpretation of the DE models. Certain checks and also the generation of reports such as coverage or the real-time events can be turned on/off. Usage of these options are further explained in the Overture/VDM user manual [LLJ⁺13].

Log: In this group it is possible to select variables from the DE model that should be logged during the simulation. To find more details about this feature, see Section 6.4.

Faults: In this group it is possible to chose a class A to replace a class B before the co-simulation start. The intention is to experiment with faulty modules that can be substituted by the non-faulty model. To make sure there will be no run-time exceptions, class B should be subclass of A. To indicate that class A should be substituted by class B, the following should be inserted in the text box *DE Replace Pattern (A/B)*. It is possible to make several substitutions by separating the substitutions with a comma A/B, C/D, . . .

Architecture: In this area it is possible to select an architecture file that defines the architecture of the deployment of the DE controller. More information on the architecture file can be seen in Section 7.6.

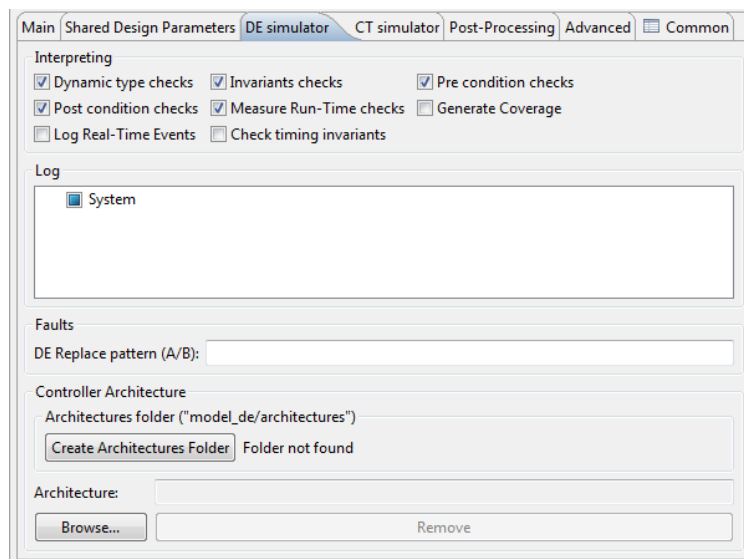


Figure 6.4: The DE simulator tab of the Debug Configuration.

6.1.5 CT Simulator Tab

The 20-sim Options tab contains options related with the execution of the CT model. At first, both tables (Log and Settings) contain only the previously saved settings, if no settings were previously selected then the tables will be empty; the tables can be populated by pressing the "Populate..." button. The "Populate..." button launches the model selected in 20-sim model and dynamically extracts the settings and the variables present in the model. As shown in Figure 6.5 there is two areas present in the 20-sim options tab:

Log: In this area it is possible to select which CT variables should be logged during the co-simulation execution.

Settings: The settings are presented in a tree view. In this tree there is two types of nodes, option nodes and "virtual" nodes which are only there to give the tree structure. If an option node is selected, the different possibilities will be presented on the right side ("Options" group).

6.2 Post-Processing Tab

The post processing tab shows the options available for the post-processing phase (see Figure 6.6).

"Show plot automatically when the script runs" with this option enabled, the Octave script that is generated after each run will contain the commands to show the plot automatically, i.e., simply running the script will show the plots. For more information on the use of Octave, please see Section 8.1.

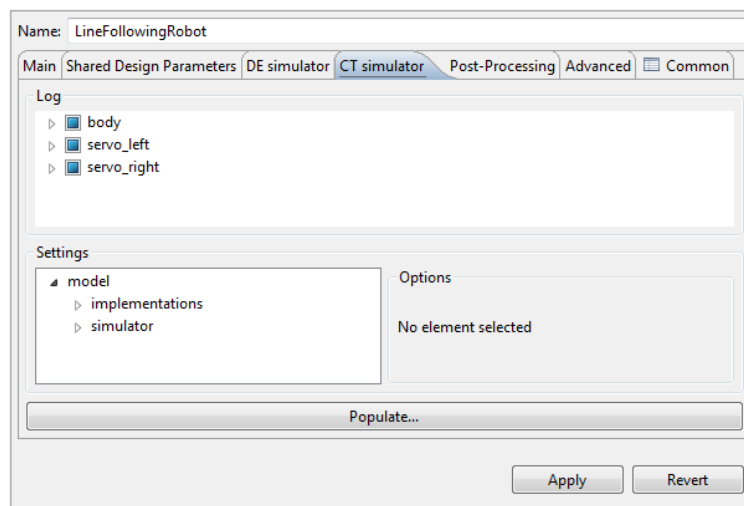


Figure 6.5: The CT simulator tab of the Debug Configuration.

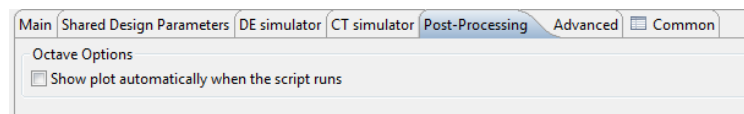


Figure 6.6: Post-processing tab for debug configuration.

6.2.1 Advanced Tab

The advanced tab is reserved for developers, extra debug information can be turned on or off in this tab (see Figure 6.7).

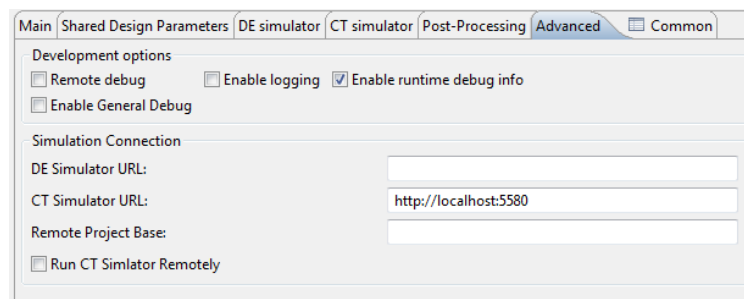


Figure 6.7: Advanced options tab.

6.2.2 Common Tab

The Common tab is a standard Eclipse tab which, for example, allows users to save the debug configurations into files so that they can be shared with others. Figure 6.8 shows how to produce a launch file that can be shared with others.

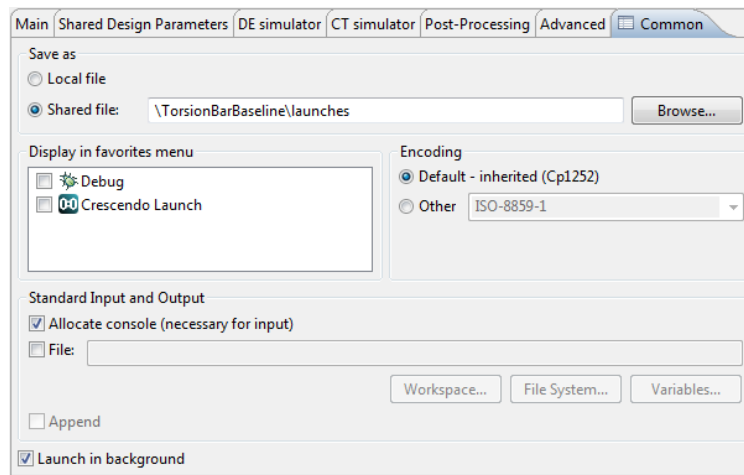


Figure 6.8: This is where launch files can be created.

6.3 Scenarios

Scripts allow users to define condition-action pairs (using a statement called **when**), which perform an action *when* the condition becomes true during a co-simulation. This script allows these conditions to reference the current co-simulation time and the state of the co-model, and to combine them with logical operators. Actions can make assignments to selected parts of the co-model and also provide information back to the user, as well as terminating the simulation.

This section describes how to create scenario files and introduces a command language for Crescendo scripts called CSL (Crescendo Scripting Language). The main purpose of CSL is to allow engineers to simulate user input and activate latent non-normative behaviours during a co-simulation. The language is designed to be sufficiently rich as to allow engineers to influence a co-model during co-simulation, without being overly complex. For example, it does not allow local variables to be defined.

6.3.1 Creating a New Scenario File

Follow these steps in order to create a new scenario file:

- Right-click on the project that is going to contain the contract file. Select “New” and *Crescendo New Scenario*.
- A new window will pop up, named *Scenario Wizard*. Select the current project by clicking on the “Browse” button. Click on the *Finish* button to end the process.

After following these steps a new file named *Scenario.script2* will be placed under the scenarios folder.



6.3.2 CSL Syntax

Here we give the syntax of the CSL using standard notation. Note that the definitions of $\langle \text{real-literal} \rangle$, $\langle \text{name} \rangle$, and $\langle \text{string} \rangle$ are not given.

$\langle \text{script} \rangle ::= \langle \text{trigger} \rangle \langle \text{script} \rangle \mid \langle \text{trigger} \rangle$

$\langle \text{trigger} \rangle ::= \langle \text{trigger-type} \rangle \langle \text{expression} \rangle \langle \text{duration} \rangle \text{'do'} \langle \text{body} \rangle \langle \text{after} \rangle$

$\langle \text{trigger-type} \rangle ::= \text{'when'} \mid \text{'once'}$

$\langle \text{expression} \rangle ::= \text{'time'} \mid \langle \text{literal} \rangle \mid \langle \text{identifier} \rangle \mid \langle \text{unary-expression} \rangle \mid \langle \text{binary-expression} \rangle$

$\langle \text{literal} \rangle ::= \langle \text{boolean-literal} \rangle \mid \langle \text{real-literal} \rangle$

$\langle \text{boolean-literal} \rangle ::= \text{'true'} \mid \text{'false'}$

$\langle \text{identifier} \rangle ::= \langle \text{simulator} \rangle \langle \text{type} \rangle \langle \text{name} \rangle$

$\langle \text{simulator} \rangle ::= \text{'de'} \mid \text{'ct'}$

$\langle \text{type} \rangle ::= \text{'boolean'} \mid \text{'real'}$

$\langle \text{unary-expression} \rangle ::= \langle \text{unary-operator} \rangle \langle \text{expression} \rangle$

$\langle \text{unary-operator} \rangle ::= \text{'not'} \mid \text{'+'} \mid \text{'-'} \mid \text{'abs'} \mid \text{'floor'} \mid \text{'ceil'}$

$\langle \text{binary-expression} \rangle ::= \langle \text{expression} \rangle \langle \text{binary-operator} \rangle \langle \text{expression} \rangle$

$\langle \text{binary-operator} \rangle ::= \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{'**'} \mid \text{'div'} \mid \text{'mod'} \mid \text{'<'} \mid \text{'<='} \mid \text{'='} \mid \text{'>='} \mid \text{'>'} \mid \text{'<>'} \mid \text{'and'} \mid \text{'or'} \mid \text{'=>'} \mid \text{'<=>'}$

$\langle \text{duration} \rangle ::= \langle \text{empty} \rangle \mid \text{'for'} \langle \text{real-literal} \rangle \text{'{' } \langle \text{time-unit} \rangle \text{'}'}$

$\langle \text{time-unit} \rangle ::= \text{'us'} \mid \text{'ms'} \mid \text{'s'} \mid \text{'m'} \mid \text{'h'}$

$\langle \text{body} \rangle ::= \langle \text{block} \rangle \mid \langle \text{statement} \rangle$

$\langle \text{block} \rangle ::= \text{'(' } \langle \text{statement-list} \rangle \text{'}'}$

$\langle \text{statement-list} \rangle ::= \langle \text{statement} \rangle \mid \langle \text{statement} \rangle \text{';' } \langle \text{statement-list} \rangle$

$\langle \text{statement} \rangle ::= \langle \text{identifier} \rangle \text{' := ' } \langle \text{expression} \rangle$

$\mid \text{'print'} \langle \text{string} \rangle$
 $\mid \text{'warn'} \langle \text{string} \rangle$
 $\mid \text{'error'} \langle \text{string} \rangle$
 $\mid \text{'quit'}$



$\langle \text{after} \rangle ::= \text{'after' } \langle \text{revert-list} \rangle$

$\langle \text{revert-list} \rangle ::= \text{'revert' } \langle \text{identifier} \rangle \mid \text{'revert' } \langle \text{identifier} \rangle \text{' ;' } \langle \text{revert-list} \rangle$

6.3.3 CSL Examples

The following introduces a series of simple examples that demonstrate the features of this script language.

```
when time = 5 do
  (de real x := 10;);
```

The **time** keyword yields the current co-simulation time. The **de** keyword indicates that *x* resides (at the top level) in the DE model. Naturally, the **ct** keyword is used to indicate the CT model. Comments may also be included:

```
when time = 5 do
  // comment
  (ct real y := true;);
```

Statements can also be grouped in blocks (surrounded by parentheses and separated by semi-colons. Expressions of time can optionally include a unit (e.g. milliseconds) given in curly braces. Units are assumed to be in seconds if no unit is given. The engineer may output messages to the tool (or to a log in batch mode) with the **print** statement:

```
when time = 900 {ms} do
(
  de real x := 10;
  ct real y := true;
  print "Co-simulation time reached 900 ms.";
);
```

Logical operators can be used in expressions. When the condition becomes true, the statement(s) in the do clause will execute.

```
when time >= 10 and time < 15 do
  (print "Co-simulation time reached 10 seconds.");)
```

If the condition becomes false again, the optional **after** clause will execute once. Note that block statements do not permit local variables to be defined. Since this script language does not allow local variables to be defined, a special statement, **revert**, may be used in an **after** clause to change a value back to what it was when the do clause executed.



```

when time >= 10 and time < 15 do
  (// assume x = 5
   de real x := 10;
  )
after (revert de real x;);

```

The engineer can reference co-model state in conditions and assignment and revert statements. The state that can be referred is either for VDM specified with the **model** keyword in the link file or for 20-sim marked as global (note 20-sim access is not yet implemented). Additionally all shared variables can be accessed with the contract name and used in conditions, assignments or revert statements.

It is also possible to have some statements executed exactly once, on the first time a condition is detected. This is achieved using the **once** keyword instead of **when**.

```

once de real x >= 500 do
(
  // set some flag
  de bool flag = true;
  print "First time x exceeds 500";
)

```

6.4 Logfiles

When starting a simulation, it is possible to select a set of variables that are logged throughout the co-simulation. At the moment of writing this manual there is only the possibility of logging DE variables; support to log CT variables will be added later. The result of this logging is a CSV file (comma separated values).

6.4.1 DE Variables

The variables of the DE model to log can be selected in the tab VDM Options presented in Figure 6.9. If a model does not contain type errors, this tab will display all instance variables that are accessible from the VDM **system** class.

Checking the box next to a variable enables the logging of that variable. Currently it is only possible to log variables with basic types (all types except objects).

For the `WatertankPeriodic` example, if we use the configuration shown above, a file with the contents as follows is generated:

```

time_, levelSensor.fault, levelSensor.level, valveActuator.valveState
0.0, 0, 0, 0

```

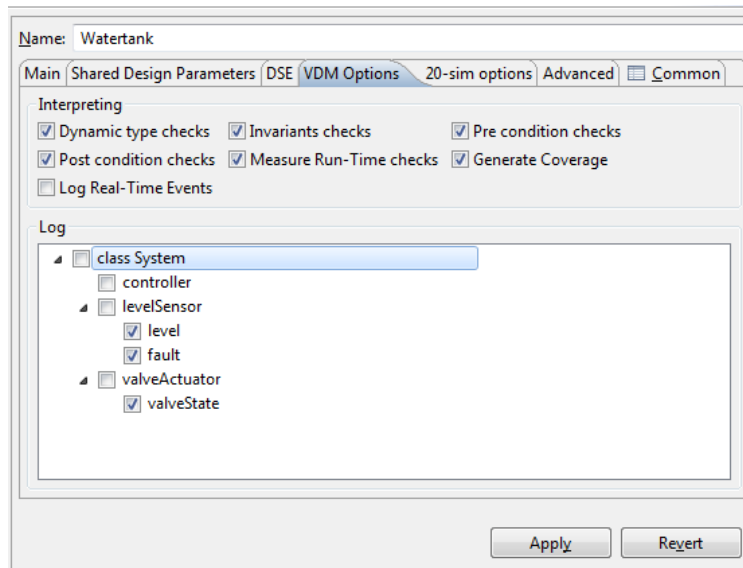


Figure 6.9: VDM Options tab permits the selection of variables to log.

```
0.01,0,0.01,0
0.01,0,0.02,0
0.02,0,0.03,0
0.03,0,0.03,0
... ..
```

The first column is the time and the following ones are the value of the variable at the given moment. A CSV file can be better visualized, for example, in *20-sim*, *Excel* or other software capable of opening this format.

Chapter 7

Design Space Exploration Possibilities

In order to support Design Space Exploration (DSE), the Automated Co-model Analysis (ACA) feature enables automatically running many different co-simulations with minimal user intervention. The ACA feature enables the user to select different configurations for each individual parts of the co-model and then runs the co-simulation combining all possible configurations that were selected by the user.

7.1 ACA Workflow

Figure 7.1 illustrates the steps in the process of the ACA work flow. First the user provides configurations for different parts of the co-simulation, then the tool generates different complete configurations by combining the different configurations parts that were provided by the user.



Figure 7.1: Illustration of the ACA process.

These complete configurations are used to execute co-simulations. Currently, it is only possible for the user to select different configurations for different parts of the co-simulation, more specifically, chose different architectures for deployment of the controller (DE side), and select different starting values for the shared design parameters.

From these partial configurations it is possible to construct complete configurations by combining each of the different partial configurations. Figure 7.2 together with the following description helps illustrating the concept. The result of generating complete configurations from the partial configuration would be 4 different complete configurations: A1-B1-C1; A1-B1-C2; A1-B2-C1; and A1-B2-C2. The user can easily get many more configurations by adding more parameters or adding more values to existing parameters, for example, simply adding a A2 value would result in 4 more different configurations.

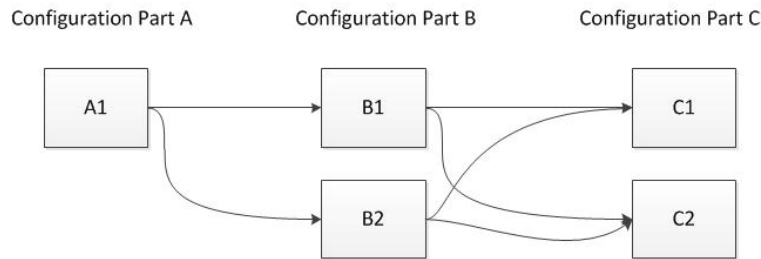


Figure 7.2: Illustration of the ACA process.

7.2 Using the ACA Features

Launching an ACA is done through the Debug Configuration menu. Creating a new Debug Configuration of an ACA Launch type will bring up the menu to configure the ACA. The different tabs in Figure 7.3, will be explained in the following subsections.

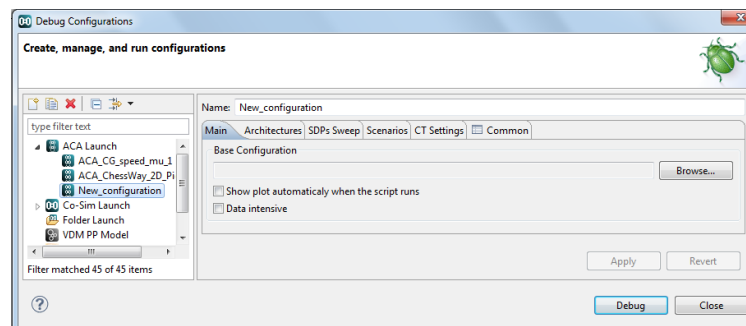


Figure 7.3:

To start an ACA launch, a base configuration needs to be selected. This configuration is a normal Crescendo launch which will be used as base for the ACA settings. This means that launch options that are not overwritten in the ACA will use as default the ones present in the base launch.

7.2.1 The Main Tab

The Main tab is the place where general settings for the ACA launch are set. Here the Base Configuration first of all needs to be selected. The Base Configuration is the co-simulation configuration that forms the base for the ACA to work. In addition there are two choices that can be made as shown in Figure 7.4.

- The first option allows the model designer to use the usual plots shown at the 20-sim side for each single co-simulation or to not spending time on that.
- The second option can be ticked if a huge amount of data is expected to be produced. If this is ticked the data generated are not included in the directory used by the Crescendo tool (in general Eclipse does not like to have a very high number of files).

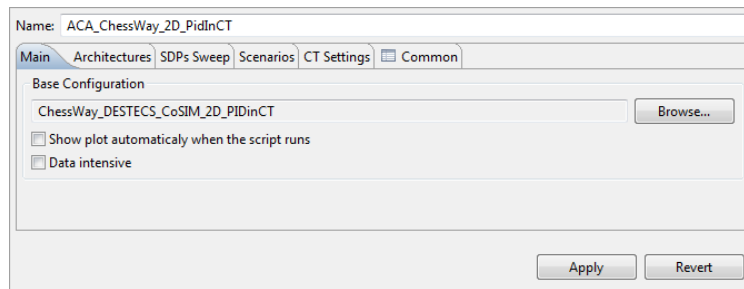


Figure 7.4: ACA Launch - Main tab.

By pressing the button Browse it is possible to browse through the Co-Sim Launches present in the Crescendo Tool and chose one. This configuration will be the base configuration for all the ones generated by the ACA. The ACA will take the base configuration and combine it in all possible ways depending on what the user set on the other tabs.

7.2.2 The Architecture Tab - Deployment Architectures

In this tab it is possible to select which Controller Architectures will be used in the ACA run (see Figure 7.5). For more information on Controller Architectures and how to define them please see Section 7.6.

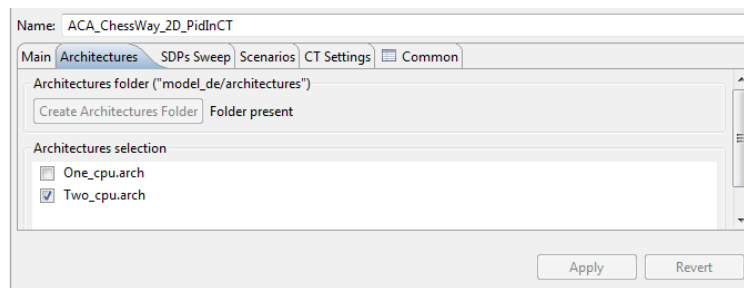


Figure 7.5: ACA Launch - Architecture tab.

7.2.3 Shared Design Parameters Tab

In the Shared Design Parameters tab it is possible to make a value “sweep” of the shared design parameters (see Figure 7.6).

The incremental Sweep

In the first column it is possible to select from a drop-down the shared design parameter to sweep (see Figure 7.7). In the second column (From), it is possible to select the value which the sweep should start from. The third column (To) indicates where the sweep should end and the forth column (Increment By) indicates the increment to be used in the sweep.



Name	From	To	Increment by	Clear
V	1.0	2.0	0.05	Delete
dist_b	0	0.4	0.2	Delete

Name	Values	Clear
mu	0.5;0.65;	Delete

Apply Revert

Figure 7.6: ACA Launch – Shared Design Parameters tab.

Name	From	To	Increment by
linefollow_lateral_offset	0.01	0.05	0.02
linefollow_longitudinal_offset	0.01	0.13	0.06

Figure 7.7: Incremental sweeping.

The value set Sweep

In the first column it is possible to select from a drop-down the shared design parameter to sweep (see Figure 7.8). In the second column a list of double values should be introduced, separated by (;).

It is possible to sweep by value set complex variables as shown in Figure 7.9.

The behaviour of complex SDPs is a bit different from the atomic SDPs. For example, the configuration on the picture above will generate 2 ACA runs for the variable “initial_Position”.

- 1st run: initial_Position = [-1.448,-1.110]
- 2nd run: initial_Position = [-1.736,X*] - * where X is the value defined in the base debug configuration for initial_Position[2].

The values defined in the value sweep are put together according to the order they appear, if

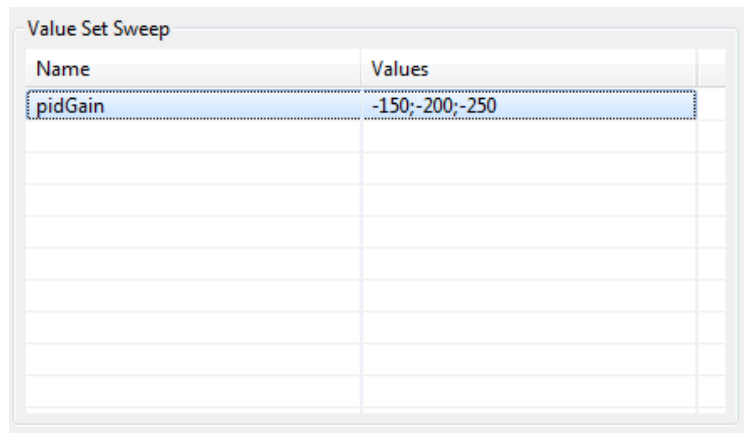


Figure 7.8: Selecting values of SDP variables to sweep over.

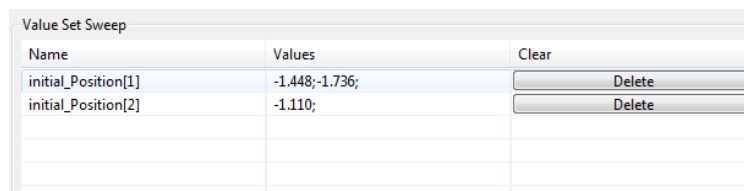


Figure 7.9: Sweeping over complex variables.

a for one of the indexes is missing (like in this case the second value of `initial_Position[2]`), the value from the original debug configuration will be used.

7.2.4 Scenario Tab

In the scenarios tab it is possible to select which scenarios will be used in the ACA run (see Figure 7.10). The scenarios present in the “scenarios” folder in the root of the project will be presented on the “Scenario selection” table. It is then possible to check which scenarios will be used in the ACA.

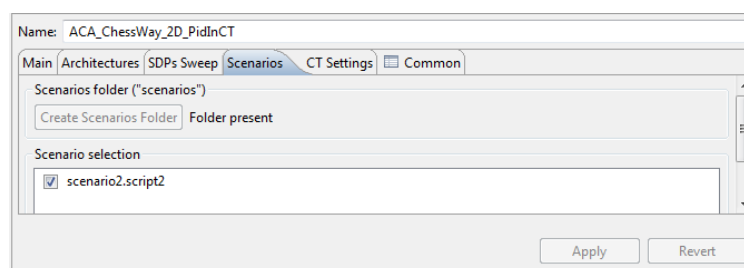


Figure 7.10: Possibility for choosing multiple scenarios for ACA.



7.2.5 CT Settings Tab

The CT side settings works in a similar fashion to the ones in the normal Crescendo launch (see Figure 7.11). The only difference is that it is possible to select multiple options instead of one. In the ACA Settings tab it is only possible to select options which have limited alternatives (i.e., enumerations).

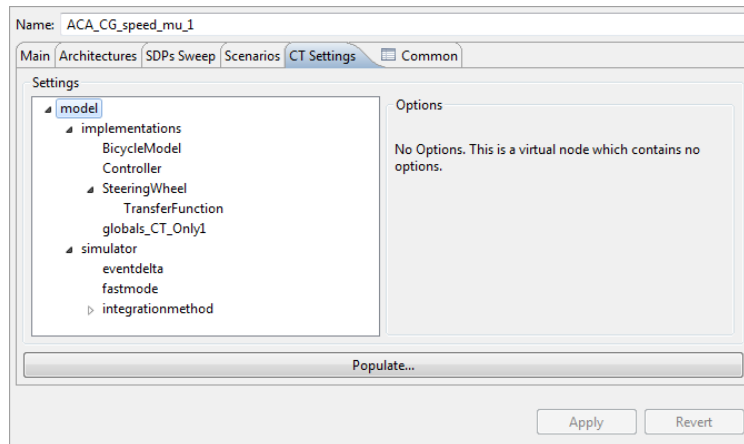


Figure 7.11: Selecting CT Settings for ACA.

7.2.6 Common Tab

The common tab settings here work much in a similar fashion to the ones in the normal Crescendo launch (see Figure 7.12).

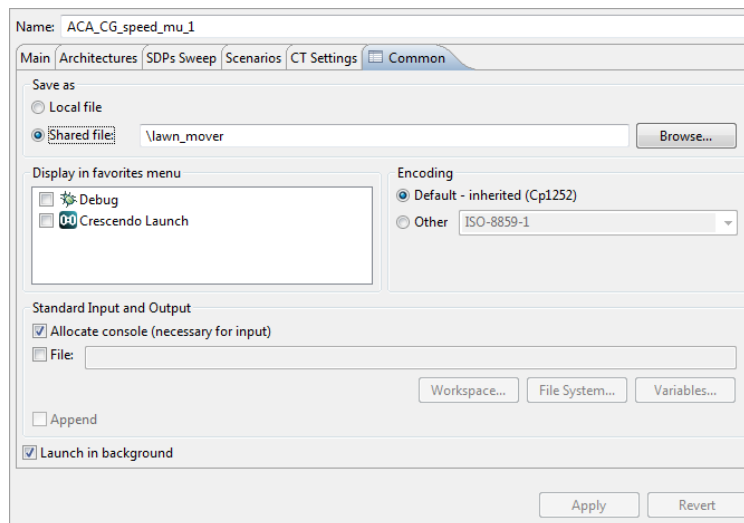


Figure 7.12: Common tab for ACA.



7.3 Repeating a Single Launch Part of an ACA

After a successful ACA launch, the `output` folder will contain information regarding what was run in a specific ACA launch. Each ACA run has generated a file named `xxx.dlaunch`.

By right-clicking a “.dlaunch” file and selecting the option *Crescendo* and then selecting the *Create and Launch* option, the single selected run will be launched again (see Figure 7.13). This single launch configuration is also stored together with the other launch configurations, typically its name is prefixed by “generated”.

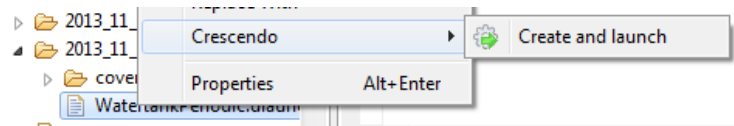


Figure 7.13: Relaunching single ACA experiments.

7.4 Folder Launch Configuration

This is a new way of launching an ACA by selecting a folder containing `.launch` files. The user has to produce its own `.launch` files. The options to select are the project and the folder containing the launch files. This is shown in Figure 7.14.

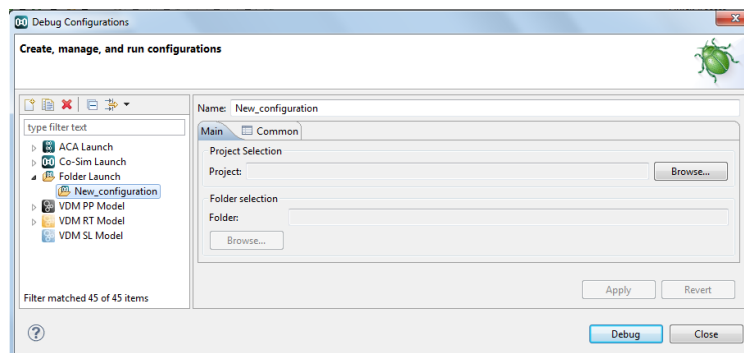


Figure 7.14: Launching with a Folder of launch files.

7.5 Control Library

In order to help build controllers in VDM that can handle low-level proportional control in addition to supervisory control, a control library has been included in the Crescendo tool. This library provides classes that are equivalent to the *P*, *PD*, *PI* and *PID* blocks of the 20-sim library under *Signal\Control\PID Control\Discrete*.



7.5.1 Accessing the Control Library

To use the control library, the class definitions must be imported into the project.

- Right-click on the project and select *New* and *Other...*
- Under the *Crescendo* folder, select *Add Crescendo Library* and click *Next*.
- Then check the box marked *Control Library*.

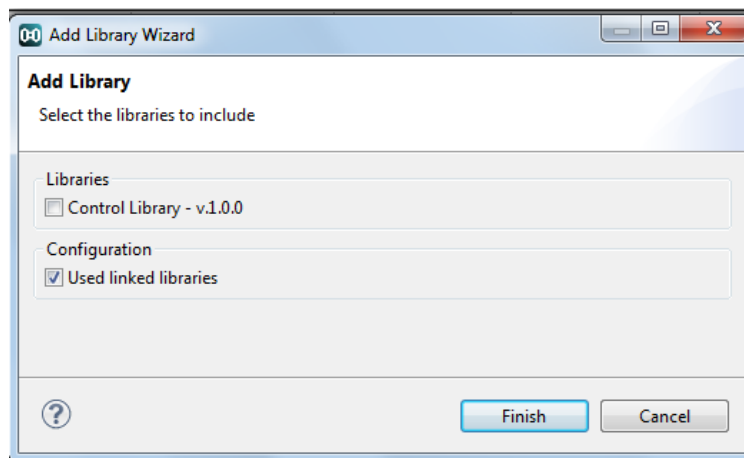


Figure 7.15: Adding Crescendo Control Libraries

Unless you want to edit the class files, leave “Use linked libraries” checked (default). The classes will now be added to your co-model (see Figure 7.15).

7.5.2 Using the Control Library

Basic Use

To use a class from the library, simply define a variable of the correct type, instantiate it with a constructor, call “SetSampleTime” and then call “Output” in your control loop. All of the control library classes have an operation called *Output*, which takes in an error and returns a control value, with the following form:

```
public Output: real ==> real
Output(err) == ...

class Controller

instance variables

-- controller object
```




```

private pid: PID;

-- setpoint
private SP: real;

-- shared variables
private MV: real;
private out: real

operations
-- constructor for Controller
public Controller: () ==> Controller
Controller() ==
(
pid := new PID(10, 1, 0.1);
pid.SetSampleTime(SAMPLE_TIME)
);

-- control loop
public Step: () ==> ()
Step() ==
(
dcl err: real := SP - MV;
out := pid.Output(err)
);

-- 100Hz control loop
values SAMPLE_TIME = 0.01;
thread periodic(10E6, 0, 0, 0)(Step);

end Controller

```

Also, all of the classes have an operation called *SetSampleTime*, which takes a sample time in seconds:

```

public SetSampleTime: real ==> ()
SetSampleTime(s) ==

```

Unlike 20-sim, VDM does not have a **sampletime** keyword, so it is necessary to explicitly tell the object what sample time to use in calculations. Therefore, for all control objects (except P) you must call *SetSampleTime* before the "Output" is used. This only needs to be done once and it is recommended that it is called immediately after the constructor. If this is not done, the co-simulation will fail with a pre-condition violation the first time *Output* is called.



7.5.3 Advanced Use

All of the controller classes in the library are subclasses of a single class called “DTControl” (Discrete-Time control). This class contains the definitions for “SetSampleTime” and “Output” and enforces a consistent interface. It is possible to use the various controller classes without making reference to `DTControl`. However, if it is desirable to test different controllers, variables can be defined as type `DTControl`, meaning that only the call to the constructor needs to be changed in order to use a different controller implementation. This is also useful if control objects are passed to controllers. In the following example, the `Controller` class can accept any control object (P, PID etc.):

```
class Controller

instance variables

-- controller object
private ctrl: DTControl;

operations

-- constructor for Controller
public Controller: DTControl ==> Controller
Controller(c) ==
(
  ctrl := c;
  ctrl.SetSampleTime(SAMPLE_TIME)
);

...
```

7.5.4 Constructors

P

The P class has the following constructors:

```
-- set k
public P: real ==> P
P(k) == ...

-- default: k = 0.2
public P: () ==> P
P() == ...
```



PD

The PD class has the following constructors:

```
-- set k, tauD, beta
public PD: real * real * real ==> PD
PD(k, tauD, beta) == ...

-- set k, tauD, beta = 0.1
public PD: real * real ==> PD
PD(k, tauD) == ...

-- default: k = 0.2, tauD = 1.0, beta = 0.1
public PD: () ==> PD
PD() == ...
```

PI

The PI class has the following constructors:

```
-- set k, tauI
public PI: real * real ==> PI
PI(k, tauI) == ...

-- default: k = 0.2, tauI = 0.5
public PI: () ==> PI
PI() == ...
```

PID

The PID class has the following constructors:

```
-- set k, tauI, tauD, beta
public PID: real * real * real * real ==> PID
PID(k, tauI, tauD, beta) == ...

-- set k, tauI, tauD, beta = 0.1
public PID: real * real * real ==> PID
PID(k, tauI, tauD) == ...

-- default: k = 0.2, tauI = 0.5, tauD = 0.5, beta = 0.1
public PID: () ==> PID
PID() == ...
```



7.6 DE Architecture

This feature allows the selection of the hardware and deployment to be specified in a separate file from the VDM system class.

In order to do this separation, the following steps need to be done:

- The System class must be cleaned of CPU and BUS declarations and deployments of the objects
- Annotations need to be added to the system class that indicate where the architecture and deployment statements. The architecture tag must be placed under an instance variables block:

```
-- ## Architecture ## --
```

The deployment tab must be placed in the constructor where the deployment normally will have been specified:

```
-- ## Deployment ## --
```

Architecture files (.arch), is placed in a folder called "model_de/architectures" in the project root. The architecture files should have the following form:

```
-- ## Architecture ## --
instance variables
cpu1: CPU := new CPU(<FCFS>, 1000000 /* Hz */);
cpu2: CPU := new CPU(<FCFS>, 1000000 /* Hz */);
cpu3: CPU := new CPU(<FCFS>, 1000000 /* Hz */);
bus1: BUS := new BUS(<FCFS>, 1000 /* bits/s */, {cpu1,cpu2,cpu3});
-- ## Deployment ## --
cpu1.deploy(mmi);
cpu3.deploy(navigation);
cpu2.deploy(radio);
```

When an architecture file like this is selected, the architecture and deployment declaration is inserted in the "right" place (under the tags in the *system* file), creating a "complete" system just before the co-simulation starts.

7.7 Events

Events can be triggered in the CT world. They will stop the simulation before the allowed time slice is completed. The co-simulation engine will then allow the DE simulator to take action but only until the point where the event has been raised. The events are used in the contract in order to support event-based triggering and not just time-triggered scheduling.



7.7.1 Simulation setup

Events in the contract

For events to be considered during a simulation the event must be defined in Section 5.3:

$\langle \text{events} \rangle ::= \text{'event'}, \langle \text{identifier} \rangle, \text{';'}$

Events in the link file

Events must be connected to a **public async** operation in VDM. This is done by linking the event name specified in the contract to the fully qualified operation name in VDM in the link file:

$\langle \text{events} \rangle ::= \text{'event'}, \langle \text{identifier} \rangle = \text{'System'}, \text{'.'}, \langle \text{identifier} \rangle, (\text{'.'}, \langle \text{identifier} \rangle)^+$

An example of this could be:

```
event event1=System.eventHandler.event1;
```

Where:

- “event1” is the event name from the contract.
- System is the **system** class.
- eventHandler is the class holding the **public async** operation to execute.
- “event1” (the last event1) is the **async** operation which to execute when the event occurs.

7.7.2 Events in CT

Events need to be marked using the keyword **'event'**, this marks the variable that it used as return value of the event function to be an event variable. The keywords **'eventdown'** and **'eventup'** are used as in standalone 20-sim models. See more under Events. Example:

```
variables
  boolean minLevelReached ('event');

equations
  maxLevelReached = eventup(levelIn-maxlevel);
```



7.7.3 Events in DE

The scheduler in VDM do not schedule events in the same way as for instance a micro controller would do where the current executing job is suspended in favour of the interrupt routine. However, it is possible to get a similar behaviour by creating and deploying an object to a CPU that contains the job to run when an interrupt occurs and then call this from the **async** operation which is triggered when event occurs. It is just important that no objects having a periodic threads are deployed to the same CPU since this will delay the event by exactly one periodic loop.

Events are linked to VDM through **async** operations and made assessable through the **system** class.

```
system System
instance variables
  eventHandler: EventHandler;
end System
```

The **async** operation must be specified in a class that is not deployed to a CPU. This makes the evaluation instant. This means that the event operation it self do not take time to run.

```
class EventHandler

operations

public async event1: () ==> ()
  event1() == skip;

end EventHandler
```

Note about the VDM scheduler: *The VDM scheduler uses priorities to select which thread should run, each thread is then executed with a limited allowed number of expressions/statements it can execute before another thread has to be scheduled and executed. The priority defined how many expressions/statements a thread can execute at a time. A thread will always continue executing until it is blocked or finished the allowed number of expressions/statements.*

Chapter 8

Post-Analysis Possibilities

8.1 Octave

GNU Octave is a high-level interpreted language, primarily intended for numerical computations. It provides capabilities for the numerical solution of linear and non-linear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation. Octave is normally used through its interactive command line interface, but it can also be used to write non-interactive programs. The Octave language is quite similar to Matlab so that most programs are easily portable. <http://www.gnu.org/software/octave/>

8.1.1 Octave Version

It is very important that the correct version of Octave is used to run the scripts generated by Crescendo. The correct version can be found in Chessforge together with the Crescendo Releases.

8.1.2 Octave use in Crescendo

After each co-simulation run, an Octave script is generated in the output dir. The script contains Octave code that reads the variable logs produced by both VDM and 20-sim during the run.

8.1.3 Show Plot Automatically when Script is Run

There is one option available in the debug configuration that affects the script. This option appears in two places both in the normal Crescendo run and in ACA. If enabled, when the script is executed, a plot (or several, depending on the amount of variables selected) will automatically be drawn. In the case of an ACA run being executed, for the same variable, the several runs will juxtaposed.

Figure 8.1 shows where to find this option for a normal run (Post-Processing tab).

Figure 8.2 show the ACA launch (Main Tab)

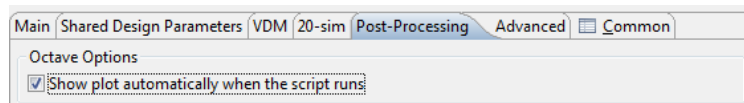


Figure 8.1: Normal Octave Selection.

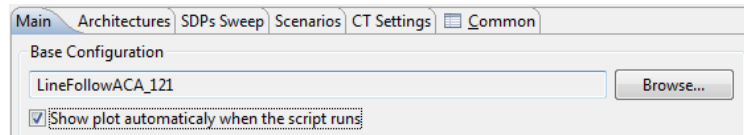


Figure 8.2: Octave with use of ACA.

8.1.4 Invoking Octave from Crescendo

It is possible to invoke Octave from the Crescendo IDE. Right-clicking on an Octave (.m) file reveals the option “Run Octave”. If you want to use this command, be sure to tick the “Show plot automatically when the script runs” box to get the result shown in Figure 8.3.

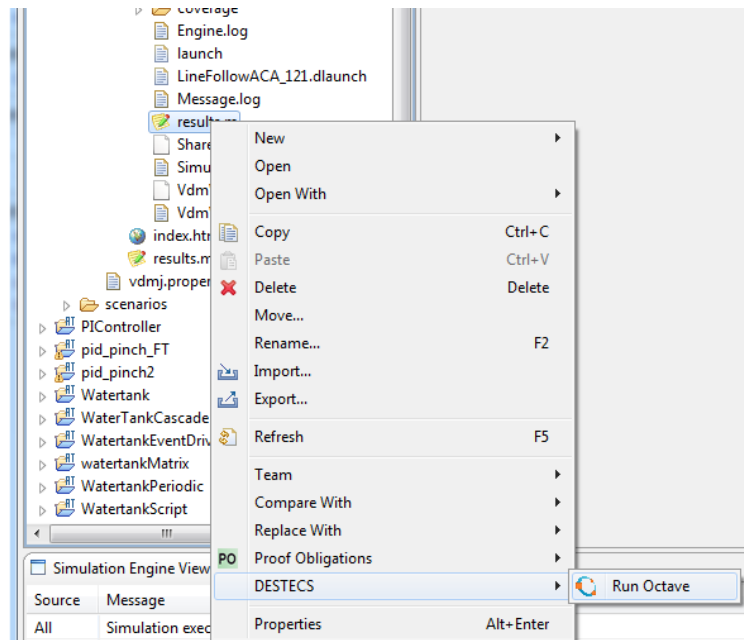


Figure 8.3: Octave in action.

If the option to show plot automatically was chosen, a plot will be drawn and shown in a window like in Figure 8.4.

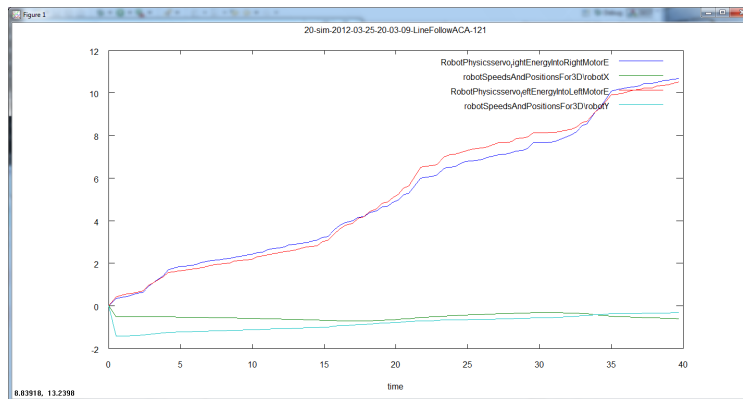


Figure 8.4: Octave plotting in separate window.

8.1.5 Setting Octave path

If your Octave installation was not made in the default installer path, the path to Octave must be corrected in the Crescendo settings for the feature mentioned above to work. The settings below can be found by navigating to the Window->Preferences menu as shown in Figure 8.5.

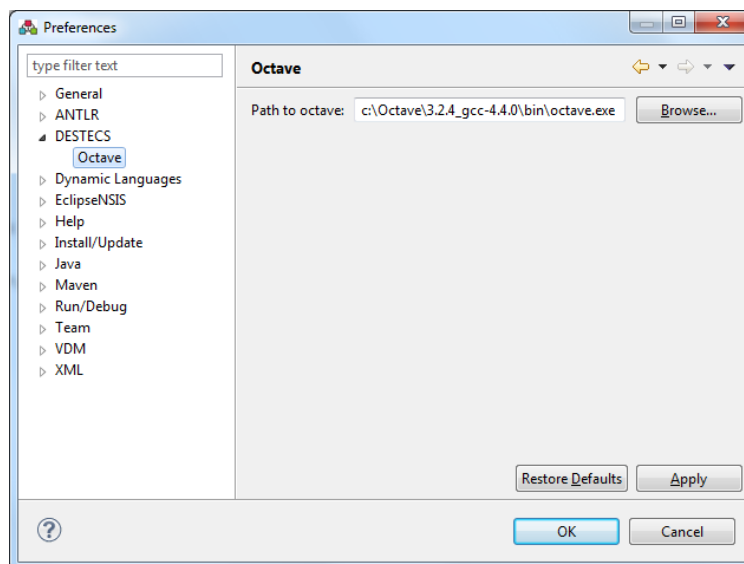


Figure 8.5: Setting of Octave usage.



Bibliography

- [ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.
- [BJ78] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.
- [BLV⁺10] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and Wouters F. Design Support and Tooling for Dependable Embedded Control Software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*, pages 77–82. ACM, April 2010.
- [CCFJ99] Tim Clement, Ian Cottam, Peter Froome, and Claire Jones. The Development of a Commercial “Shrink-Wrapped Application” to Safety Integrity Level 2: the DUST-EXPERT Story. In *Safecomp’99*, Toulouse, France, September 1999. Springer Verlag. LNCS 1698, ISBN 3-540-66488-2.
- [Con13] Controllab products. <http://www.20sim.com/>, January 2013. 20-Sim official website.
- [FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
- [FL09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.
- [FLM⁺05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.
- [FLV08] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.
- [FLV13] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2013.



- [IEE00] IEEE 100 The Authoritative Dictionary of IEEE Standards Terms Seventh Edition. *IEEE Std 100-2000*, 2000.
- [ISO96] Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.
- [Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.
- [Kle09] C. Kleijn. *20-sim 4.1 Reference Manual*. Controllab Products B.V., Enschede, First edition, 2009. ISBN 978-90-79499-05-2.
- [KN09] T. Kurita and Y. Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3):343–355, October 2009.
- [LH96] Peter Gorm Larsen and Bo Stig Hansen. Semantics for underdetermined expressions. *Formal Aspects of Computing*, 8(1):47–66, January 1996.
- [LLB⁺13] Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgerald, Sune Wolff, and Shin Sahara. VDM-10 Language Manual. Technical Report TR-001, The Overture Initiative, www.overturetool.org, April 2013.
- [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.
- [MBD⁺00] Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.
- [Rob04] Stewart Robinson. *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, 2004.
- [VLH06] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.

Appendix A

Glossary

As might be expected in such an interdisciplinary project, terms and concepts that are well known in one discipline may be unknown or understood quite differently in another. This page therefore contains common descriptions of core concepts agreed with the partners that are used consistently within the project.

abstract class (in object oriented programming) a class where one or more methods are defined abstractly using the text **is subclass responsibility** as their body.

actuator a component that produces a physical output in response to a signal [IEE00].

aggregate (in object oriented programming) the act of bringing together several objects into a single whole.

automated co-model analysis tool support for the selection of a single design from a set of design alternatives (including definition of scenarios, execution of co-simulations, and visualisation and analysis of co-simulation results).

automated co-model execution as automated co-model analysis except that it does not perform any analysis of the test results produced by the simulations

bond (in bond graphs) a directed point-to-point connection between power ports on submodels. Represents the sharing of both *flow* and *effort* by those ports.

bond graph a domain independent idealised physical model based on the representing energy and its exchange between submodels.

causality (in bond graphs) dictates which variable of a power port is the input (cause) for sub-model's equations and which is the output (effect).

class (in object oriented programming) the definition of the data field and methods an object of that class will contain.



code generation the process of implementing a system controller by automatically translating a model into a representation (in some programming language) which can then be executed on the real hardware of the system.

co-model a model comprising two constituent models (a DE submodel and a CT submodel) and a contract describing the communication between them.

consistency a co-model is consistent if the constituent models are both syntactically and semantically consistent.

constituent model one of the two submodels in a co-model.

continuous-time simulation a form of simulation where “the state of the system changes continuously through time” [Rob04, p. 15].

contract a description of the communication between the constituent models of a co-model, given in terms of shared design parameters, shared variables, and common events.

controlled variable a variable that a controller changes in order to perform control actions.

controller the part of the system that controls the plant.

controller architecture the allocation of software processes to CPUs and the configuration of those CPUs over a communications infrastructure.

co-sim launch the type of debug configuration used in the Crescendo tool to define and launch a single scenario.

co-simulation baseline the set of elements (co-model, scenario, test results etc.) required to reproduce a specific co-simulation.

co-simulation engine a program that supervises a co-simulation.

co-simulation the simulation of a co-model.

cost function a function which calculates the “cost” of a design.

debug config (Eclipse term) the place in Eclipse where a simulation scenario is defined.

design alternatives where two or more co-models represent different possible solutions to the same problem.

design parameter a property of a model that affects its behaviour, but which remains constant during a given simulation.

design space exploration the (iterative) process of constructing co-models, performing co-simulations and evaluating the results in order to select co-models for the next iteration.



- design step** a co-model which is considered to be a significant evolution of a previous co-model.
- discrete-event simulation** a form of simulation where “only the points in time at which the state of the system changes are represented” [Rob04, p. 15].
- disturbance** a stimulus that tends to deflect the plant from desired behaviour.
- edges** (in bond graphs) see *bond*.
- effort** (in bond graphs) one of the variables exposed by a power port. Represents physical concepts such as electrical voltage, mechanical force or hydraulic pressure.
- environment** everything that is outside of a given system.
- error** part of the system state that may lead to a failure [ALRL04].
- event** an action that is initiated in one constituent model of a co-model, which leads to an action in the other constituent model.
- executable model** a model that can be simulated.
- failure** a system’s delivered service deviates from specification [ALRL04].
- fault injection** the act of triggering faulty behaviour during simulation.
- fault modelling** the act of extending a model to encompass faulty behaviours.
- fault** the adjudged or hypothesized cause of an error [ALRL04].
- fault behaviour** a model of a component’s behaviour when a fault has been triggered and emerges as a failure to adhere to the component’s specification.
- fault-like phenomena** any behaviour that can be modelled like a fault (e.g. disturbance).
- flow** (in bond graphs) one of the variables exposed by a power port. Represents physical concepts such as electrical current, mechanical velocity, fluid flow.
- ideal behaviour** a model of a component that does not account for disturbances.
- inheritance** (in object oriented programming) the mechanism by which a subclass contains all public and protected data fields and methods of its superclass.
- input** a signal provided to a model.
- interface** (in object oriented programming) a class which defines the signatures of but no bodies for any of its methods. Should not be instantiated.
- junction** (in bond graphs) a point in a bond graph where the sum of flow (1-junction) or effort (0-junction) of all bonds to that point is zero.



log data written to a file during a simulation.

metadata information that is associated with, and gives information about, a piece of data.

model base the collection of artefacts gathered during a development (including various models and co-models; scenarios and test results; and documentation).

model management the activity of organizing co-models within a model base.

model structuring the activity of organizing elements within a model.

model synthesis see **code generation**.

model a more or less abstract representation of a system or component of interest.

modelling the activity of creating models.

modularisation construction of self-contained units (modules) that can be combined to form larger models.

monitored variable a variable that a controller observes in order to inform control actions.

object (in object oriented programming) an instantiation of a class, contains data fields and methods.

objective function see **cost function**.

ontology a structure that defines the relationships between concepts.

operation (in object oriented programming) defines an operation that an object may perform on some data. Operations may be *private*, *public* or *protected*.

output the states of a model as observed during (and after) simulation.

non-normative behaviour behaviour that is judged to deviate from specification.

physical concept (in bond graphs) a class of component or phenomena that could exist or be observed in the real world, e.g. an electrical resistor or mechanical friction.

plant the part of the system which is to be controlled [IEE00].

power port (in bond graphs) the port type connected in a bond graph. Contains two variables, *effort* and *flow*. A power port exchanges energy with its connected models.

private (in object oriented programming, VDM) the method or data field may only be accessed from within the containing class.

protected (in object oriented programming, VDM) the method or data field may only be accessed by its containing class or any of its subclasses.



public (in object oriented programming, VDM) the method or data field may be accessed by any other class.

ranking function a function that assigns a value to a design based on its ability to meet requirements defined by the engineer.

realistic behaviour a model of a component which includes disturbances defined by the tolerances associated with that component.

repository a shared store of data or files.

response a change in the state of a system as a consequence of a stimulus.

revision control the activity of managing changes (revisions) to computer data or files.

scenario test of a co-model.

signal domain where models share a single value or array at each port and where those ports are uni-directional, unlike bond graphs where the ports are bi-directional.

sensor a component whose input is a physical phenomenon and whose output is a quantitative measure of the phenomenon.

shared design parameter a design parameter that appears in both constituent models of a co-model.

shared variable a variable that appears in and can be accessed from both constituent models of a co-model.

simulation symbolic execution of a model.

semantically consistent the state when the constituent models of a co-model agree on the semantics of the variables, parameters and events they share. The nature of these semantics is not yet described.

static analysis a method for checking some property of a model without executing that model.

state event an event triggered by a change within a model.

stimulus a phenomenon that effects a change in the state of a system.

subclass (in object oriented programming) a class that is defined as extending another class. The other class becomes its superclass. The subclass inherits all non private data fields and methods.

submodel a distinct part of a larger model.

superclass (in object oriented programming) the class from which a subclass is defined.



syntactically consistent the state when the constituent models of a co-model agree on the identities and data types of all shared variables, parameters and events.

system boundary the common frontier between a system and its environment.

system under test (SUT) the part of a model that represents the system we wish to build, as opposed to parts of the model which are not part of this system.

system an entity that interacts with other entities, including hardware, software, humans and the physical world [ALRL04].

tag to associate metadata with a piece of data.

test result a record of the output from a simulation of a model (see also **log**).

time event an expected event that occurs at a predetermined time.

variable part of a model that may change during a given simulation.

vertices (in bond graphs) the joining points of bonds. May be manifested as either a *junction* or a submodel.