DACCOSIM NG 2018
USER’S GUIDE

Last update December 7th, 2018

Jean-Philippe Tavella, EDF Lab Paris-Saclay
José Évora-Gómez, Monentia SL
José Juan Hernández-Cabrera, SIANI
Stéphane Vialle, CentraleSupélec

jean-philippe.tavella@edf.fr
jose.evora@monentia.com
josejuanhernandez@siani.es
stephane.vialle@centralesupelec.fr

Final version is DACCOSIM NG v2.2.0-stable

Copyright © 2018 EDF, Monentia
1 What is DACCOSIM NG? .................................................................4
  1.1 DACCOSIM NG project contributors ....................................5
  1.2 DACCOSIM NG licensing mode ...........................................5

2 Overview of DACCOSIM NG 2018 .............................................6
  2.1 Main DACCOSIM NG features ..............................................6
  2.2 DACCOSIM NG control architecture ...................................7

3 Recommended installation procedures .....................................9
  3.1 Required Java installation ..................................................9
  3.2 Installation for Windows environment ...................................9
  3.3 Installation for Linux environment ......................................10
  3.4 PowerShell script execution ..............................................10

4 DACCOSIM NG 2018 Editor .....................................................11
  4.1 Launching the Editor ......................................................11
  4.2 Interface description ......................................................11
  4.3 Menus bar and toolbar .....................................................12
     4.3.1 “File” menu .........................................................12
     4.3.2 “Edit” menu .......................................................12
     4.3.3 “Simulation” menu ...............................................13
     4.3.4 “Help” menu .....................................................13
  4.4 Palette ............................................................................14
     4.4.1 FMU .................................................................14
        4.4.1.1 FMU block properties ......................................14
        4.4.1.2 FMU model description ................................15
     4.4.2 External Input .......................................................17
     4.4.3 External Output ....................................................17
     4.4.4 Adder ...............................................................18
     4.4.5 Offset ................................................................19
     4.4.6 Multiplier ...........................................................20
     4.4.7 Gain ..................................................................20
  4.5 Canvas ............................................................................21
     4.5.1 Drawing connections ...............................................21
     4.5.2 Defining variables on connections ...............................22
     4.5.3 Co-simulation configuration .....................................24
        4.5.3.1 “General” tab ...............................................25
        4.5.3.2 “Vars to export” tab ......................................26
        4.5.3.3 “Stepper vars” tab ........................................27
     4.5.4 Running a co-simulation ..........................................28
  4.6 Annotation area ..............................................................29
  4.7 Matryoshka FMU .............................................................29
     4.7.1 Internal dependencies ..............................................30
     4.7.2 Exporting an FMU ..................................................30

5 DACCOSIM NG 2018 Shell .....................................................32
  5.1 Launching the Shell ..........................................................32
  5.2 Co-simulation scripting ....................................................32
6 Deploying and launching co-simulations on a cluster resource ........................................... 39

7 Information collected during a co-simulation ........................................................................... 40

7.1 Tracking information ........................................................................................................... 40

7.2 Results data ......................................................................................................................... 41

7.2.1 Post-mortem visualization ............................................................................................... 41

7.2.1.1 Post-mortem visualization with OMPlot in a Windows environment ........................... 42

7.2.1.2 Post-mortem visualization with OMPlot in a Linux environment ............................... 42

7.2.1.3 Post-mortem visualization with OMEdit ...................................................................... 43

7.2.2 Real time visualization within the Editor ........................................................................... 43

8 Use cases added to the distribution ......................................................................................... 44

8.1 Co-initialization examples .................................................................................................... 45

8.1.1 equationsPair case ........................................................................................................... 45

8.1.1.1 Graphs and simulation properties ................................................................................... 46

8.1.1.2 Discussion on the results ............................................................................................... 46

8.1.2 tripleEquation case ............................................................................................................ 46

8.1.2.1 Graphs and simulation properties ................................................................................... 47

8.1.2.2 Discussion on the results ............................................................................................... 47

8.1.3 complexCoinitCase1 ........................................................................................................... 47

8.1.3.1 Graphs and simulation properties ................................................................................... 48

8.1.3.2 Discussion on the results ............................................................................................... 48

8.1.4 complexCoinitCase2 ........................................................................................................... 48

8.1.4.1 Graphs and simulation properties ................................................................................... 49

8.1.4.2 Discussion on the results ............................................................................................... 49

8.2 Academic co-simulation examples ......................................................................................... 49

8.2.1 tankYPipe case .................................................................................................................. 49

8.2.1.1 Graphs and simulation properties ................................................................................... 50

8.2.1.2 Discussion on the results ............................................................................................... 50

8.2.2 tankYPipeWithRequirements case .................................................................................... 50

8.2.3 events case ........................................................................................................................ 51

8.2.3.1 Graphs and simulation properties ................................................................................... 51

8.2.3.2 Discussion on the results ............................................................................................... 51

8.2.4 stairBouncingBall case ........................................................................................................ 52

8.2.4.1 Graphs and simulation properties ................................................................................... 52

8.2.4.2 Discussion on the results ............................................................................................... 52

8.2.5 dCACInverter case ............................................................................................................ 53

8.2.5.1 Graphs and simulation properties ................................................................................... 53

8.2.5.2 Discussion on the results ............................................................................................... 53

8.2.6 tankBarrel case ................................................................................................................ 53

8.2.6.1 Graphs and simulation properties ................................................................................... 54

8.2.6.2 Discussion on the results ............................................................................................... 54

8.2.7 tankBarrelWithTelecoms case ........................................................................................... 55
8.2.7.1 Graphs and simulation properties ................................................................. 55
8.2.7.2 Discussion on the results .............................................................................. 56
8.2.8 dynamicEquationsPair case ........................................................................... 56
8.2.8.1 Graphs and simulation properties ................................................................. 56
8.2.8.2 Discussion on the results .............................................................................. 57
8.2.9 heatedRoom case ........................................................................................... 57
8.2.9.1 Graphs and simulation properties ................................................................. 57
8.2.9.2 Discussion on the results .............................................................................. 57
8.2.10 heatedSystemWithTwoBuildingsAndTelecoms case ...................................... 58
8.2.10.1 Graphs and simulation properties ............................................................... 58
8.2.10.2 Discussion on the results ............................................................................ 59
8.2.11 multipleFMU case ......................................................................................... 59
8.2.11.1 Graphs and simulation properties ............................................................... 59
8.2.11.2 Discussion on the results ............................................................................ 60
8.2.12 multipleCnx case .......................................................................................... 60
8.2.12.1 Graphs and simulation properties ............................................................... 60
8.2.12.2 Discussion on the results ............................................................................ 60

8.3 Industrial co-simulation examples ........................................................................ 61
8.3.1 DistrictParis900s0000000100shade case ........................................................... 61
8.3.1.1 Graphs and simulation properties ................................................................. 61
8.3.1.2 Discussion on the results ............................................................................. 62

9 Some known limits ............................................................................................... 64

10 Annex about variable stepping implemented in DACCOSIM NG 2018 ................ 65
10.1 Quantization methods for variable stepping .................................................... 65
10.2 Euler and Adams-Bashforth-Moulton method ................................................... 65
10.2.1 Calculating two approximations with embedded methods ............................. 65
10.2.1.1 Application to the Euler method ................................................................. 65
10.2.1.2 Application to the Adams-Bashforth-Moulton method ............................... 66
What is DACCOSIM NG?

DACCOSIM NG is the second generation of DACCOSIM (Distributed Architecture for Controlled Co-Simulation) that is a powerful FMI\(^1\) compatible Master Algorithm designed as a collaborative development effort between EDF (France), CentraleSupélec (France), EIFER - European Institute for Energy Research (Germany), SIANI institute (Spain), and Monentia S.L. (Spain).

EDF R&D\(^2\) and CentraleSupélec\(^3\) jointly launched in December 2012 the RISEGrid institute\(^4\) as a laboratory of the Paris Saclay campus dedicated to the study, modelling and simulation of smart distribution power grids and their interactions with the whole energetic system.

EIFER, SIANI and Monentia closely work with RISEGrid and common research topics cover both theoretical aspects and more applicative and industrial ones. RISEGrid aims to be in perfect accordance with the strong and challenging evolutions of the electric power systems in Europe: opening of electricity markets, development of decentralized production, ubiquitous information and communication means, decentralized control and regulation algorithms...

More generally speaking, all entities carry out researches combining the multidisciplinary aspects of energetic systems (a kind of systems including multi-physics, control commands, telecommunication networks...) mixing new modelling languages and methods, and more efficient simulations of very wide and complex systems.

Very early, EDF bet on the co-simulation principle to associate heterogeneous active components exported from various and heterogeneous modeling tools. This choice is particularly relevant as energetic systems are a particular class of CPSs (Cyber-Physical Systems) composed of numerous subsystems to be equally taken into account to get a fairly realistic representation of the whole system behavior.

DACCOSIM NG is a major step towards a multi-domain co-simulation tool including both time-continuous dynamic systems (for multi-physics purpose) and discrete time simulators (for ICT subsystems).

DACCOSIM NG is based on the FMI-CS (FMI for Co-Simulation) standard in order to couple active components (simulators) within a global co-simulation environment. These simulators are executable codes exported from different tools as distribution files called FMUs (Functional Mock-up Unit), each including a binary form of a model combined with a numerical solver.

DACCOSIM NG objective is to take advantage of all the features available in the FMI standard, especially, from FMI 2.0 July 25\(^{th}\), 2014, variable communication step size, inputs extrapolation, FMU state and rollback, and directional derivatives. DACCOSIM NG also strives to follow the progresses of the standard in order to experiment future features like hybrid co-simulation in the next FMI 3.0 version.

First public versions of DACCOSIM were released in October 2015 (DACCOSIM 2015) and January 2017 (DACCOSIM 2017). Note these obsolete versions are no more downloadable and are to be replaced by DACCOSIM NG new versions, starting with DACCOSIM NG 2018.

DACCOSIM NG aims to be extremely simple to install and very user-friendly. The code has been completely rewritten by Monentia to offer a more robust tool with better performances than the previous releases.

---


1.1 DACCOSIM NG project contributors

DACCOSIM NG is a collaborative project permanently improved, tested and maintained by the following core-team:

- Jean-Philippe Tavella (simulation expert & research engineer at EDF Lab Paris-Saclay, France).
- Pr Stéphane Vialle (LRI, GeorgiaTech-CNRS, CentraleSupelec, Université Paris-Saclay, France).
- Dr José Évora-Gómez (MONENTIA, Las Palmas de GC, España).
- Dr José-Juan Hernández (SIANI, Universidad de Las Palmas de Gran Canaria, España).
- Dr Enrique Kremers (EIFER, European Institute for Energy Research, Deutschland).

In addition, every year, one-time contributors participate to the development of DACCOSIM NG, especially students or PhDs from RISEGrid and various high schools trainees at EDF Lab Paris-Saclay.

The DACCOSIM NG project team plan to deliver each year at least one new release of the DACCOSIM NG tool suite including the tool itself and some use cases as illustrative examples.

**DACCOSIM v2.2.0-stable** is the final version of DACCOSIM NG for 2018 and it is available for downloading at [https://bitbucket.org/simulage/daccosim](https://bitbucket.org/simulage/daccosim), which is the new DACCOSIM project repository.

1.2 DACCOSIM NG licensing mode

DACCOSIM NG is released under the GNU LGPL-v3 license: [http://www.gnu.org/licenses/lgpl](http://www.gnu.org/licenses/lgpl).

The DACCOSIM NG tool uses the “fmu-wrapper” and “fmu-builder” that are components of the JavaFMI tool suite developed and maintained by SIANI, University of Las Palmas, Spain. Refer to [https://bitbucket.org/siani/javafmi](https://bitbucket.org/siani/javafmi) for more information on this tool suite.
2 Overview of DACCOSIM NG 2018

DACCOSIM NG 2018 emphasizes a complete and user-friendly interface for configuring and performing local or distributed co-simulations with potentially many heterogeneous FMUs compliant with the co-simulation part of the FMI standard (FMI-CS) from and after version 2.0.

2.1 Main DACCOSIM NG features

The DACCOSIM NG tool suite is composed of complementary parts:

- An Editor as a Graphical User Interface (GUI) designed to achieve the preparation and execution of co-simulations, especially:
  - Designing co-simulation graphs (i.e. the blocks to be considered and the data to be exchanged between them);
  - Setting co-simulation parameters (i.e. starting time, ending time, stepping strategy...);
  - Mapping co-simulations on calculation resources either on a single multi-core machine or in a multi-node cluster environment;
  - Interfacing execution with real-time tracking.

- A Shell underlying computation package able to manage in batch mode execution of co-simulations on various environments (single machine or cluster).

- **In a future development, a Java extension named DacRun (DACCOSIM Runner) will be developed based on the Java Message Service to automate the pre-processing distribution of codes, the execution of co-simulations, and the post-processing (collect of results and log files) on distributed Linux computing resources compliant with OAR and SLURM cluster managers.**

- A viewer able to visualize results of co-simulations.

⚠ Editor and Shell can run under 64-bit or 32-bit Windows or Linux environment. Nevertheless, mixing 64-bit and 32-bit FMUs is not allowed with DACCOSIM NG 2018.

![Figure 2.1: Overview of the main features covered by DACCOSIM NG](image-url)

Copyright © 2018 EDF, Monentia
2.2 DACCOSIM NG control architecture

The approach used in DACCOSIM NG is to define co-simulation graphs composed of blocks (mainly FMUs) that are connected by logical links and potentially distributed on different computation nodes. Each graph is then translated into a data structure transferred to an optimized master code developed in Java in conformance with the features described in the FMI-CS standard.

More precisely, DACCOSIM NG 2018 relies on the FMI-CS capabilities in order to offer:

- **Global causal dependency graph** automatically generated using both all FMUs internal dependencies and the co-simulation graph external dependencies in order to determine the causal chains and loops.
- **Global and distributed co-initialization method** based on a generalized Newton-Raphson algorithm, even for hybrid dependency graphs mixing loops and chains.
- **Distributed master kernels** operating with constant or variable stepping. Error control is performed implementing quantization single-step methods (future development) or Adams-Bashforth multiple-step methods.
- **Input data extrapolation** in conformance with the order of the variable stepping integration method used.
- **Exact event detection** according to a hybrid co-simulation behavior to be soon included in the next FMI 3.0 version with early return on internal events (future development).
- **Automatic generation of a "Matryoshka"**, which is a multithreaded super-FMU encapsulating a DACCOSIM NG co-simulation graph and its inner FMUs.

DACCOSIM NG master code follows a *centralized hierarchical approach* where a *unique global master* located on one cluster node is in charge of handling the control data flowing from several *local masters* distributed on the other cluster nodes and taking step by step decisions based on these information. Every master, either global or local, aggregates these control data that are coming from each FMU wrapper present on its cluster node. This is done before communicating synthesized information to the global master.

![Diagram](image-url)

*Figure 2.2: Illustration of the distributed global/local master architecture of DACCOSIM NG*
The data exchanged between masters and also between FMUs and masters are called “vertical data”. Of course when a co-simulation is run on a single machine, only one master code is generated by DACCOSIM NG.

The originality of the DACCOSIM NG architecture lies in the fact that the data to be exchanged between FMUs at each communication step (“horizontal data”) are directly transmitted from the senders to the receivers without passing by any master.

The masters, the wrapped blocks (mainly FMUs with wrappers) as well as the communication channels between them are automatically configured by DACCOSIM NG by translating the multi-FMU co-simulation graph into an internal efficient data structure.
3  Recommended installation procedures

DACCOSIM NG 2018 is a set of Java applications distributed as .jar files for the Editor and the Shell. These applications are then usable either on Windows or Linux environment.

For Windows environment, subsidiary applications are supplied as .exe files encapsulating the .jar Editor file either for 64-bit or 32-bit machines.

3.1  Required Java installation

A JRE (Java Runtime Environment) is required to run the .jar files. It is not necessary to install a JDK (Java Development Kit) as no code generation and compilation is required with DACCOSIM NG; this is one of the most important difference from the previous DACCOSIM versions.

The recommended Java environment to use is the Java SE Version 8 that can be downloaded from https://www.java.com. The JRE installation should be consistent with the host machine, i.e. a 64-bit JRE should be installed on a 64-bit computer and a 32-bit JRE should be installed on a 32-bit one.

3.2  Installation for Windows environment


The installers have been built with Inno Setup, a free installer builder for Windows applications. Refer to http://www.jrsoftware.org/isinfo.php for more information on this tool.

Figure 3.1: Setup to install the DACCOSIM NG suite

After installation, and depending on the chosen version, the installation folder should contain:

- A 64-bit (resp. 32-bit) execution file for the Editor named daccosim-editor-2018-xxx-64bits.exe (resp. daccosim-editor-2018-xxx-32bits.exe),
- A .jar file for the Shell named daccosim-shell-2018-xxx.jar,
- A readme file as daccosim-2018-readme.txt,
- An uninstaller unins000.exe with its data as unins000.dat,
- A folder named daccosim-documentation containing the most recent version of the DACCOSIM NG 2018 User’s Guide, a flyer and some publications.
Note an icon can be added on the Desktop depending on the user options selected at the installation.

To limit the size of downloadable files, available free use cases are to be separately downloaded. They are zipped in a file `daccosim-use-case-windows-2018xxx.zip` that must be unzipped in a folder `daccosim-use-cases-windows` to be located below the DACCOSIM NG installation folder.

After a complete installation (tools and use cases), the DACCOSIM NG root directory should look like this:

```
- daccosim-documentation
- daccosim-use-cases-windows
- daccosim-2018-readme.txt
- daccosim-editor-2018-v2.0.1-stable-64bits.exe
- daccosim-shell-2018-v2.0.1-stable.jar
- unins000.dat
- unins000.exe
```

![Figure 3.2: Overview of the DACCOSIM NG 2018 root folder after a complete installation (on a 64-bit machine)](image)

### 3.3 Installation for Linux environment

No installer is supplied for Linux environment.


To limit the size of downloadable files, available free use cases are to be separately downloaded. They are zipped in a file `daccosim-use-case-linux-2018xxx.zip` that must be unzipped in a folder `daccosim-use-cases-linux` to be located below the DACCOSIM NG installation folder.

### 3.4 PowerShell script execution

Some scripts are supplied within the distribution, for example to run the available free use cases in batch mode or to visualize results after execution.

For portability reason between Windows and Linux, PowerShell has been chosen to write these scripts that can be freely adapted by the user. Refer to the website [https://github.com/PowerShell/PowerShell](https://github.com/PowerShell/PowerShell) for more information.

Whatever the environment (Window or Linux), the machine where the given PowerShell scripts are to be launched should be configured to accept local scripting. This configuration requires “Administrator right” to run the following recommended command:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope {CurrentUser | LocalMachine}
```

- oOo -
4 DACCOSIM NG 2018 Editor

The purpose of the DACCOSIM NG Editor is to offer a GUI (Graphical User Interface) to make co-simulation studies more comfortable, from the pre-processing stage (configuration) to the post-processing (results visualization) passing by an efficient and parallel execution.

Before getting started with the Editor, it is recommended to follow the installation procedure described in previous chapter.

4.1 Launching the Editor

Under Windows, the Editor can be run by the installed `daccosim-editor-2018-xxx-64bits.exe` or `daccosim-editor-2018-xxx-32bits.exe` application. Optionally, a shortcut has been created on the Desktop by the installer for a direct execution.

Under Linux, the Editor can be run by an on-line command from the Terminal console, as shown here:

```
java -jar daccosim-editor-2018-xxx.jar
```

⚠️ Note several Editors can be launched and simultaneously played on the same machine.

4.2 Interface description

When the Editor is launched, a splash screen is shortly displayed and the main window appears.

The different parts of this main window are the menus bar (A), the toolbar (B), the palette (C), the canvas (D), and the annotation area (E). In following sections, each of these areas is fully described.

![Figure 4.1: General view of the Editor main window](image-url)
4.3 Menus bar and toolbar

The menus bar allows to make operations relevant to the co-simulation graph currently defined in the canvas. These operations are available as commands in the menus “File”, “Edit”, “Simulation”, and “Help”.

The most often used commands are also available in the toolbar in order to provide graphical shortcuts to the user.

Next subsections explain more deeply the operations available via each menu command.

4.3.1 “File” menu

The commands of the “File” menu are general operations to manage co-simulation graphs stored in files:

- **New**: this command clears the canvas and provides a new empty co-simulation graph to start modelling. If a previous graph is not saved, the user is asked to save or discard it.

- **Open**: this command opens a previously stored co-simulation graph by loading an existing .simx file. If a previous graph is not saved, the user is asked to save or discard it.

  This command can also be launched via a keyboard shortcut (Ctrl+O).

- **Save**: this command saves the currently edited co-simulation graph. In case the graph has never been saved, the user will be asked the name and the place where the co-simulation graph is to be saved.

  This command can also be launched via a keyboard shortcut (Ctrl+S).

- **Save as**: this command saves the currently edited co-simulation graph in a new place or with a new name.

- **Import dngx file (with .dng)**: this command loads a previously stored co-simulation graph by importing an existing .dngx file. A graph is automatically created in the canvas with default position for each object contained in the file. This graph must be saved by the user as a .simx file before execution in the Editor.

- **Export as FMU**: this command exports the current co-simulation graph as a single parallel FMU (also known as Matryoshka). This super-FMU can then be used in other tools (like Dymola) or as a component of a wider co-simulation graph inside DACCOSIM NG. Refer to Matryoshka sub-section for more information. The user will be asked for the name of the resulting file and the compatibility with the operating systems that is desired.

- **Exit**: this command closes the Editor. If the current graph is not saved, the user is asked to save or discard it.

Note that the New, Open... and Save commands are also available as graphical shortcuts in the toolbar.

4.3.2 “Edit” menu

The “Edit” menu offers usual edition commands to design co-simulation graphs:

- **Undo**: this command undoes the last edition operation made into the currently edited co-simulation graph.

  It can also be launched via a keyboard shortcut (Ctrl+Z).

- **Redo**: this command redoes the last edition operation that was undone.

  It can also be launched via a keyboard shortcut (Ctrl+Y).

- **Cut**: this command cuts a selection of blocks and arrows done from the graph.

  It can also be launched via a keyboard shortcut (Ctrl+X).

- **Copy**: this command copies a selection of blocks and arrows done from the graph.

  It can also be launched via a keyboard shortcut (Ctrl+C).

- **Paste**: this command pastes an existing selection gotten from a Cut or Copy command. It can also be launched via a keyboard shortcut (Ctrl+V).

- **Delete**: this command removes a selection of blocks and arrows.

- **Select All**: this command makes a selection including all blocks and arrows visible in the currently edited co-simulation graph.
• **Select None**: this command unselects all blocks and arrows previously selected in the graph.
• **Fill color...**: this command changes the color of the blocks included in a selection.
• **Font color...**: this command changes the color of the text in the blocks included in a selection.
• **Properties...**: this command opens a dialog to edit the property of a single object (block or arrow). This dialog is different depending on the kind of object (refer to the palette section).

Note that the Undo, Redo, Cut, Copy, Paste and Delete commands are also available as graphical shortcuts in the toolbar.

### 4.3.3 “Simulation” menu

The commands of the “Simulation” menu enables checking, configuration, and running of co-simulation graphs:

• **Validate graph**: this command checks if the current co-simulation graph is correctly defined and candidate for running. In case it is not, a dialog is shown informing about the error(s).

• **Settings**: this command allows the user to configure the parameters to be used to run the current co-simulation graph. Settings dialog is fully described further.

• **Run**: this command executes the current co-simulation graph providing a follow-up window of the execution. When finished, a link is available at the bottom of this window in order to open the results file either via an OMPlot application (if correctly installed from an OpenModelica distribution) or otherwise by using the .csv default system tool defined on the machine.

Note these three commands are also available as graphical shortcuts in the toolbar.

### 4.3.4 “Help” menu

The “Help” menu only holds one command:

• **About**: displays which version of the Editor is being used and other interesting parameters to consider for running like the current Operating System and JVM version. Furthermore, some helpful links are available:
  
  o [https://www.gnu.org/licenses/lgpl](https://www.gnu.org/licenses/lgpl) points to the content of the GNU Lesser General Public License version 3,
  
  o [https://bitbucket.org/siani/javafmi](https://bitbucket.org/siani/javafmi) is the JavaFMI repository,
  
  o [https://bitbucket.org/simulage/daccosim](https://bitbucket.org/simulage/daccosim) is the DACCOSIM NG repository.

![Figure 4.2: Help information](image-url)
4.4 Palette

The palette exhibits the different objects that can be incorporated into a co-simulation graph. Concretely, objects can be dragged from the palette and dropped into the canvas. As already stated, objects visible in the canvas can also be selected for Copy-Paste or Cut operations.

Next sub-sections explain what the purpose of these objects is and how to configure them.

4.4.1 FMU

The “Fmu” object represents a FMU block of the co-simulation graph. Once dragged and dropped into the canvas, a file chooser is shown to select the .fmu file that will be represented by the dragged “Fmu” object instance:

![Figure 4.3: Selection of an .fmu file for a FMU block instance](image)

4.4.1.1 FMU block properties

After selecting a valid FMU file, a dialog shows editable fields and clickable buttons for an “Fmu” object instance:

![Figure 4.4: “Fmu” block instance setting](image)
Properties for an “Fmu” object instance are:

- **Label**: names an FMU in the co-simulation graph. This field is editable and will be used to identify inner variables in the results file.
- **... button**: reloads the same FMU which is helpful when the FMU has been updated or to load another FMU to be attached to the current FMU block.
- **A table** of variables belonging to the *initialization list* of the FMU. This table is composed of four columns named ‘Type’, ‘Mode’, ‘Variable name’ and ‘Init value’:
  - **Type**: is the type (Real, Boolean, Integer, String) of the current variable (for information);
  - **Mode**: defines the initialization mode (“Before” or “In Init”) of the current variable (for information);
  - **Variable name**: is the name of current variable (for information);
  - **Init value**: is the default value of the current variable if available in the *modelDescription.xml* file of the FMU. Double-click the field to change it.
- **+ button**: adds one or several new variables into the *initialization list*. A filter can be applied on the listed variables to select a subset of them whose name contains a given string (case is ignored). After selection, the **Ok button** must be pressed.
- **- button**: removes one or several selected variables from the *initialization list*.
- **Ok button**: assigns the currently edited fields and closes the “FMU properties” window.
- **Cancel button**: closes the “FMU properties” window without taking into account the most recent changes done after the last assignment.

Note the variables exposed by the tool to integrate the *initialization list* are selected in conformance with the FMI-CS 2.0 standard where some variables can be initialized either before or during the *Initialization Mode* of the FMU:

- The Mode “Before” is given to any variable that can be initialized before entering the *Initialization Mode* (after a *fmi2Instantiate()* call and before a *fmi2EnterInitializationMode()* call) that is to say:
  - any variable with (variability="fixed" or variability="tunable") and causality="parameter";
  - any variable with (variability="fixed" or variability="tunable") and (causality="calculated parameter" or causality="local") and initial="approx";
  - any variable with (variability="discrete" or variability="continuous") and (causality="local") and (initial="exact" or initial="approx").
- The Mode “In Init” is given to the variables that can only be initialized during the *Initialization Mode* (after a *fmi2EnterInitializationMode()* call and before a *fmi2ExitInitializationMode()* call) that is to say say any variable with causality="input".

Also note propagation of inputs values to outputs is done in conformance with the internal dependencies of each FMU as they are exposed in its *modelDescription.xml* file. When an FMU does not expose internal dependencies (which is unfortunately allowed by the FMI standard), the assumption is made that each external output depends on all external inputs of the FMU. These default dependencies cannot be modified by the user.

### 4.4.1.2 FMU model description

Information on the FMU can be accessed by a right-click on any “Fmu” block instance and selecting the *Model description*... command in the contextual menu. These information are extracted from the *modelDescription.xml* file of the FMU:
Figure 4.5: Main attributes describing a FMU block instance

The displayed attributes of an “Fmu” block instance are:

- **modelName**: name of the FMU. Don’t confuse with the Label.
- **generationTool**: optional name of the tool that generated the FMU.
- **fmiVersion**: version of the FMI standard used to export the FMU. DACCOSIM NG supports the FMI standard versions 2.0 and 3.0 (future development).
- **generationDateAndTime**: date and time when the FMU was generated.
- **canBeInstantiatedOnlyOncePerProcess**: indication if a FMU can be instantiated more than once on a cluster node. If false, several copies of the same FMU can be used within the co-simulation graph.
- **needsExecutionTool**: indication if a tool is needed to execute the model. The FMU just contains the communication path to this tool.
- **canHandleVariableCommunicationStepSize**: indication if the FMU can be run with variable simulation time steps. If false, no variable step simulation method can be used within the cosimulation graph.
- **canInterpolateInputs**: indication if the FMU can interpolate its inputs when derivatives are provided. If false, there will be no input extrapolation within the co-simulation graph.
- **canRunAsynchronously**: indication if the FMU can carry out the fmi2DoStep() call asynchronously. Note that asynchronous mode is not handled by DACCOSIM NG.
- **canNotUseMemoryManagementFunctions**: indication if the FMU can use its own functions for memory allocation and freeing only.
- **canGetAndSetFMUState**: indication if the FMU can take and restore snapshots of its internal and external states. If false, no variable step simulation method can be used within the cosimulation graph.
- **canSerializeFMUState**: indication if the FMU can serialize the internal FMU state. Not used with DACCOSIM NG.
- **providesDirectionalDerivative**: indication if the FMU can provide the directional derivative of the equations at communication points.
- **maxOutputDerivativeOrder**: indication if the FMU can provide derivatives of outputs with a given maximum order.
- **defaultExperimentTolerance**: preferred default experiment tolerance.
- **defaultExperimentStartTime**: preferred default start time.
- **defaultExperimentStopTime**: preferred default stop time.
- **defaultExperimentStepSize**: preferred default step size with constant stepping.
4.4.2 External Input

The “External Input” object allows to define a set of external inputs that can be used to feed other blocks of the co-simulation graph. A dialog shows editable fields and clickable buttons for an “External Input” object instance:

From the point of view of the graph, an “External Input” provides data from environment to some internal blocks. Once this block is dragged into the canvas, properties can be accessed by double-clicking the block or using the right-click button and selecting Properties… These properties are:

- **Label**: the name of the external input in the co-simulation graph. A default name is proposed but the field is editable and will be used to identify inner variables in the results file.
- A **table** of variables belonging to the *outputs list* of the block. This table is composed of four columns named ‘Type’, ‘Output name’, ‘Variability’, and ‘Initial value’:
  - **Type**: the type (Real, Boolean, Integer, String) of a variable defined as an item of the table. Click the field to change it.
  - **Output name**: the name of the variable to output from the block. A default name is proposed: double-click on this field to change it.
  - **Variability**: the variability (constant, discrete, continuous) of the current variable. Double-click on this field to change it.
  - **Initial value**: the default value of the current variable. Double-click on this field to change it. Note the assigned value will be checked considering the variable type set.
- **+ button**: adds a new output in the block (a new row in the table).
- **- button**: removes the selected output(s) of the block (removes rows in the table).
- **Ok button**: assigns the currently edited fields and closes the “External input properties” window.
- **Cancel button**: closes the “External input properties” window without taking in account the most recent changes done after the last assignment.

4.4.3 External Output

The “External Output” object allows to define a set of external outputs coming from other blocks of the co-simulation graph. A dialog shows editable fields and clickable buttons for an “External Output” object instance:
From the point of view of the graph, an “External Output” provides data to the environment from some internal blocks. Once this block is dragged into the canvas, properties can be accessed by double-clicking the block or using the right-click button and selecting Properties... These properties are:

- **Label**: the name of the external input in the co-simulation graph. A default name is proposed but the field is editable and will be used to identify inner variables in the results file.

- A **table** of variables belonging to the *inputs list* of the block. This table is composed of four columns named ‘Type’, ‘Input name’, ‘Variability’, and ‘Value’:
  - **Type**: the type (Real, Boolean, Integer, String) of a variable defined as an item of the table. Click the field to change it.
  - **Input name**: the name of the variable to output from the block. A default name is proposed: double-click on this field to change it.
  - **Variability**: the variability (discrete, continuous) of the current variable. Double-click on this field to change it.
  - **Initial value**: the default value of the current output. Double-click on this field to change it. Note the assigned value will be checked considering the variable type set.

- **+ button**: adds a new output in the block (a new row in the table).
- **- button**: removes the selected output(s) of the block (removes rows in the table).
- **Ok button**: assigns the currently edited fields and closes the “External output properties” window.
- **Cancel button**: closes the “External output properties” window without taking in account the most recent changes done after the last assignment.

### 4.4.4 Adder

The “Adder” object allows to combine two or more inputs from the same type. This type can be Real or Integer (in this case, the combination is an addition) or Boolean (in this case, the combination is a Boolean OR operation). A dialog shows editable fields and clickable buttons for an “Adder” object instance:
The properties of this block can be accessed from the contextual command Properties... or by double clicking it. These properties are:

- **Label**: the name of the block that will be used. A default name is proposed but the field is editable and will be used to identify inner variables in the results file.
- **Type**: the common type of the values that will be added (Real, Integer or Boolean).
- **A table** of variables belonging to the *inputs list* of the block. This table is composed of one column named ‘Inputs’:
  - **Inputs**: the current list of declared inputs. A default name is proposed: double-click on this field to change it.
- **+ button**: adds a new input in the block (a new row in the table).
- **- button**: removes the selected input(s) of the block (removes rows in the table).
- **Ok button**: assigns the currently edited fields and closes the “Adder properties” window.
- **Cancel button**: closes the “Adder properties” window without taking in account the most recent changes done after the last assignment.

### 4.4.5 Offset

The “Offset” object allows to combine an input with a constant value from the same type. This type can be Real or Integer (in this case, the combination is an addition) or Boolean (in this case, the combination is a Boolean OR operation). A dialog shows editable fields and clickable buttons for an “Offset” object instance:

The properties of this block can be accessed from the contextual command Properties... or by double clicking it. These properties are:

- **Label**: the name of the block that will be used. A default name is proposed but the field is editable and will be used to identify inner variables in the results file.
- **Type**: the type of the values that will be added (Real, Integer or Boolean).
- **Offset value**: the constant value to be combined to the single input of the block.
- **Ok button**: assigns the currently edited fields and closes the “Offset properties” window.
- **Cancel button**: closes the “Offset properties” window without taking in account the most recent changes done after the last assignment.
4.4.6 Multiplier

The “Multiplier” object allows to combine two or more inputs from the same type. This type can be Real or Integer (in this case, the combination is a multiplication) or Boolean (in this case, the combination is a Boolean AND operation). A dialog shows editable fields and clickable buttons for a “Multiplier” object instance:

![Multiplier properties](image)

Figure 4.10: “Multiplier” block instance setting

The properties of this block can be accessed from the contextual command Properties... or by double clicking it. These properties are:

- **Label**: the name of the block that will be used. A default name is proposed but the field is editable and will be used to identify inner variables in the results file.
- **Type**: the common type of the values that will be added (Real, Integer or Boolean).
- **A table** of variables belonging to the inputs list of the block. This table is composed of one column named ‘Inputs’:
  - **Inputs**: the current list of declared inputs. A default name is proposed: double-click on this field to change it.
- **+ button**: adds a new input in the block (a new row in the table).
- **- button**: removes the selected input(s) of the block (removes rows in the table).
- **Ok button**: assigns the currently edited fields and closes the “Multiplier properties” window.
- **Cancel button**: closes the “Multiplier properties” window without taking in account the most recent changes done after the last assignment.

4.4.7 Gain

The “Gain” object allows to combine an input with a constant value from the same type. This type can be Real or Integer (in this case, the combination is a multiplication) or Boolean (in this case, the combination is a Boolean AND operation). A dialog shows editable fields and clickable buttons for a “Gain” object instance:

![Gain properties](image)

Figure 4.11: “Gain” block instance setting

The properties of this block can be accessed from the contextual command Properties... or by double clicking it. These properties are:
- **Label**: the name of the block that will be used. A default name is proposed but the field is editable and will be used to identify inner variables in the results file.
- **Type**: the type of the values that will be added (Real, Integer or Boolean).
- **Gain value**: the constant value to be combined to the single input of the block.
- **Ok button**: assigns the currently edited fields and closes the “Gain properties” window.
- **Cancel button**: closes the “Gain properties” window without taking in account the most recent changes done after the last assignment.

4.5 Canvas

As already stated, objects are dragged from the palette and dropped into the canvas to graphically create a co-simulation graph. Then when objects are present on the canvas, connections can be drawn to define the data flow in the graph.

4.5.1 Drawing connections

A connection is a channel between two blocks able to hold one or several variables to be exchanged between these blocks.

Drawing a connection is possible when a source block is selected. A block selection is only possible when the cursor is approximately overlying the center of the chosen block. Under this condition a bright green frame surrounds the block, and the block selection is effective when the left button of the mouse is pressed.

Keeping the left button of the mouse pressed, the user has to reach a destination block. When this destination block is selected (also with a bright green frame surrounding the block), a direct link should appear between the two connected blocks and the mouse button can be released.

![Figure 4.12: Drawing a connection between two blocks: source selection on the left and destination selection on the right](image)

Each link can be moved to change its starting and ending connection points. These changes are only possible by dragging the green point present in the middle of the arrow. **The blue points must not be used as they may remove the current link.**

![Figure 4.13: Updating the position of a connection between two blocks](image)

In a very similar way, links can be drawn between other blocks like “External Input”, “External Output”, “Adder”, “Offset”, “Multiplier” or “Gain”. Just be aware that some pair of blocks cannot be connected together, as for example two “External Input” blocks or two “External Output” blocks.
It is also possible to precisely define how the arrow representing the connection exits or enters the source or target node in case the green point is not enough to draw the arrow as desired. To this end, it is possible to do a right-click with the arrow selected and click on “Fix source/target points”. A dialog will appear in which it is possible to define how the arrow exits or enters the block by indicating the percentage of the axes X and Y of the blocks.

4.5.2 Defining variables on connections

Once a connection has been established between two blocks, the user can select the output variables of the source block that must be associated to the input variables of the destination block.

This action is done by double clicking on a drawn arrow, leading to the opening of an “Arrow properties” window:
Figure 4.16: Setting of variables to transmit through a connection

This window contains some informative and non-editable fields:

- **Type**: indicates the type of each pair of input and output connected variables.
- **<blockname> outputs**: lists the outputs already connected to inputs.
- **<blockname> inputs**: lists the inputs already connected to outputs.

The same window also exposes clickable buttons or areas to select/unselect input variables paired to output variables:

- **+ button**: after selection of one or several rows in the table, selects the same input variable to be paired to all output variables of the selection. The same action is possible by double-clicking on any of the selected rows in the table.
- **- button**: after selection of one or several rows in the table, unselects the input variable paired to all output variables of the selection.
- **Ok button**: assigns the currently defined pairs of input/output variables and closes the “Arrow properties” window.
- **Cancel button**: closes the “Arrow properties” window without taking into account the most recent changes done after the last assignment.

When a complete pairing has been done for the connection, the “Arrow properties” window should look like this one:
Note that at the bottom and right part of the “Arrow properties” window, an indicator gives the number of assigned pairs of input/output variables regarding the number of inputs variables defined in the destination block. Complete pairing for one connection occurs when all inputs of the destination block are assigned to get a checkable co-simulation graph.

Also note a tooltip appears as an information box with an extract of the already defined variables when the user hovers a link between two blocks with his mouse pointer.

4.5.3 Co-simulation configuration
When a co-simulation graph is defined with all required blocks and well-paired connections, the data flow between objects is complete, and the properties of the co-simulation have finally to be set. This setting is possible when clicking on the button on the right.

A “Co-simulation configuration” window appears with three different tabs, and the tab “General” is by default selected. Whatever the selected tab, main buttons at the bottom of the “Co-simulation configuration” window are:
• **Ok button**: assigns the currently edited fields and closes the “Co-simulation configuration” window.

• **Cancel button**: closes the “Co-simulation configuration” window without taking into account the most recent changes done after the last assignment.

### 4.5.3.1 “General” tab

The “General” tab allows the definition of the co-simulation parameters:

**Figure 4.19: “General” tab of the “Co-simulation configuration” window**

The editable fields from this tab are:

- **Start time**: time instant indicating the beginning of the co-simulation (must be real and positive). Unit is FMU-dependent. Default value is 0.0.

- **Stop time**: time instant indicating the end of the co-simulation (must be a real and positive number higher than or equal to the start time value). Unit is FMU-dependent. Default value is 0.0.

- **(Co-init) Tolerance**: the co-initialization is a global method adapted to pure acyclic dependency graphs, pure cyclic dependency graphs and hybrid (cyclic/acyclic) dependency graphs. A tolerance is to be given for the Newton-Raphson algorithm used to calculate initial input values to propagate in each SCC (Strongly Connected Component). Default value is $10^{-5}$.

- **(Co-init) Max Iterations**: maximum number of steps considered by the Newton-Raphson algorithm for each SCC during the co-initialization phase. Default value is 100.

- **(Stepper) Method**: method used to control the time stepping among:
  - **Constant**: constant time stepping (default value),
  - **Euler**: variable time stepping controlled by the Euler method. **This method will be replaced in the future by several quantization methods.**
  - **Adams Bashforth**: variable time stepping controlled by a multi-steps Adams-Bashforth method. More information on the variable stepper methods implemented in DACCOSIM NG 2018 can be found in Annex 1.

- **(Stepper) Step size**: step size when the stepper method is **Constant**. Unit is FMU-dependent. Default value is 60.

- **(Stepper) Min step**: value of the time step under which the selected control method cannot go. This field is not accessible with the **Constant** stepper method. Unit is FMU-dependent. Default value is 1. **This parameter will be automatically set by the tool in the future.**

- **(Stepper) Safety factor**: safety factor used to limit the growth of the time step calculated by an adaptive method. A consensual value is often chosen in the range $[0.8, 0.9]$. This field is not accessible with the **Constant** stepper method. Default value is 0.9.
- **(Stepper) Order**: order (between 3 and 9) for the adaptive multi-steps *Adams-Bashforth* method when this method has been chosen. Default value is 3.
- **(Stepper) Max No. steps**: maximum number of simulation time steps. Default value 0 is to be interpreted as no limitation.
- **Export**: this check box indicates if results files are to be generated by the co-simulation. In this case, the tool checks that at least one variable is defined for exportation in results files. Default value is Export box checked.
- **(Export) Cell sep**: cell separator for results files. Possible values are “;” and “,”. This field is not accessible when the Export box is not checked. Default value is “;”.
- **(Export) Decimal sep**: decimal mark for results files. Possible values are “,” and “.”. But “.” is the only possibility when the cell separator has been chosen to “,”. This field is not accessible when the Export box is not checked. Default value is “.”.
- **Default button**: resets the default values for all the parameters in the “General” tab.

### 4.5.3.2 “Vars to export” tab

The “Vars to export” tab allows the definition of the variables to export as results at the end of a co-simulation run. This option is only available when the export checkbox is checked. The dialog to configure the variables to export is shown below.

![Figure 4.20: “Vars to export” tab of the “Co-simulation configuration” window](image)

This tab exposes a table containing the list of the variables to export as time-dependent variables of the co-simulation:
• **Variable name**: defines the list of variables to export.

The tab also exposes clickable buttons to add or remove variables:

• **+ button**: allows to add one or several new variables to export in results file. A filter can be applied on the listed variables to select a subset of them whose name contains a given string (case is ignored). After selection, the **Ok button** must be pressed.

• **- button**: removes one or several variables according to a beforehand selection.

⚠️ Note all the variables exposed from blocks of the co-simulation graph can be logged whatever their nature (inputs, outputs, internal variables or parameters). Each variable has a full-name with a prefix for the block name (its Label) and a suffix for the name of the variable as it is defined in the block.

⚠️ Also note there is no default selection; in particular, block inputs and outputs are not logged by default.

### 4.5.3.3 “Stepper vars” tab

The “Stepper vars” tab is only accessible when a variable stepping is chosen within the “General” tab. Its purpose is to allow the definition of the variables participating to the variable stepper handling:

![Co-simulation configuration window](image)

**Figure 4.21**: “Stepper vars” tab of the “Co-simulation configuration” window

This tab exposes a table containing the list of the variables participating to the variable stepper handling:

- **Variable name**: defines the list of variables participating to the variable stepping handling.
- **Tolerance**: defines the tolerance attached to the corresponding variable. His editable field has a default value set to “1E-5”.

Copyright © 2018 EDF, Monentia
The tab also exposes clickable buttons to add or remove variables:

- **+ button**: allows to add one or several new variables for variable stepping purpose. A filter can be applied on the listed variables to select a subset of them whose name contains a given string (case is ignored). After selection, the **Ok button** must be pressed.
- **- button**: removes one or several variables according to a beforehand selection.

The variables that are set by default in this dialog are those whose derivatives are calculated and available in the FMU. For instance, if there is a variables called `speed` and another variable called `der(speed)`, then `speed` variable is candidate to be tracked in the dynamic stepping procedure and it will be added by default.

### 4.5.4 Running a co-simulation

When a co-simulation graph is defined with connections, data flow between objects and the properties correctly defined, a check can be done to control a co-simulation graph is runnable.

This check is possible when clicking on the **✓** button. The checks that are made to validate the graph are:

- The co-simulation graph must have blocks,
- The blocks of the graph must have distinct labels,
- All defined arrows have, at least, one connection declared,
- All operator blocks have, at least, one arrow to provide input values,
- All operator blocks have, at least, one arrow to connect the output value with inputs,
- All external output blocks have, at least, one arrow to provide input values,
- All external input blocks have, at least, one arrow to connect the output value with inputs,
- All variables defined as exported are existing,
- All variables used to define a variable stepper are existing,
- If exportation is enabled, the list of variables to export cannot be empty.

Apart of these checks, note that many other checks are made when editing the co-simulation graph. For instance, arrows are deleted if they are not well defined concerning the source and target blocks. Another example is about initial values that must be in conformance with the type of variables.

After checking, a co-simulation graph can be run by clicking on the **▶** button. During the execution, a log windows appears with information on the running progress.
4.6 Annotation area

A free text can be associated to a co-simulation graph. This editable text located at the bottom of the Editor main window will be seen by the tool as a comment or an annotation.

4.7 Matryoshka FMU

As already stated, the “Export as FMU...” command of the “File” menu is dedicated to the exporting of a co-simulation as a parallel FMU, what is called here a “super-FMU” or a Matryoshka.
When a valid co-simulation is exported as a "super-FMU" from the Editor, it exposes:

- Inputs when external inputs are defined,
- Outputs when external outputs are defined,
- Parameters exposed by each inner FMU (future development),
- Variables exposed by each inner FMU (future development).

A Matryoshka always implements a co-initialization algorithm. About the step size, it can handle variable stepping only if the graph from where it has been exported has been defined with a variable stepping method.

### 4.7.1 Internal dependencies

When a valid co-simulation including both external inputs and outputs is exported as a Matryoshka, the dependencies of external outputs regarding its external inputs are automatically computed according to the causality graph of the co-simulation graph. When an FMU does not expose internal dependencies (which is unfortunately allowed by the FMI standard), the assumption is made that each external output depends on all external inputs of the FMU. These default dependencies cannot be modified by the user.

### 4.7.2 Exporting an FMU

The command "Export as FMU..." from the File menu of the Editor is used to export a single node co-simulation as a powerful multi-threaded FMU. By default, the base name of the Matryoshka FMU is derived from the .simx file edited on the Editor sheet.

A user interface appears allowing to select the execution operating system to consider for the generated FMU. It can be Windows 64-bit, Linux 64-bit or both. By default, the operating system of the hosting machine is proposed in this interface. Selecting “Cancel” means exiting the generation process.

The `modelDescription.xml` file of this “superFMU” is generated with the following main attributes:

- fmiVersion is "2.0" ("3.0" in future development);
- generationTool is "Daccosim NG 2018-v2.2.0-stable";
- description is "Daccosim NG Matryoshka FMU";
- modelName is derived from the co-simulation name.

`guid` and `generationDateAndTime` attributes are also automatically set when the FMU export is done from the DACCOSIM NG Editor.
In addition here are the exposed attributes of the element “CoSimulation” for this modelDescription.xml file:

- needsExecutionTool="false";
- canBeInstantiatedOnlyOncePerProcess="true" (in future development, it will be possible to get multiple instantiations per process);
- canHandleVariableStepSize="true";
- canInterpolateInputs="true";
- maxOutputDerivativeOrder="1";
- canGetAndSetFMUstate="false" (true in future development);
- canSerializeFMUstate="false";
- providesDirectionalDerivative="false" (true in future development).

The needed resources for a “superFMU” built as an exported FMU from the DACCOSIM NG Editor are located in the resources folder. These resources files are a global Matryoshka.jar file and all the inner FMUs as .fmu files.

⚠️ Windows .dll and Linux .so can be both present in the binaries folder. But 32-bit architectures are not supported for the Matryoshkas. Therefore, only 64-bit FMUs can compose the internal co-simulation graph.
5 DACCOSIM NG 2018 Shell

The purpose of the DACCOSIM NG Shell is to run co-simulation files in batch mode.

Before getting started with the Shell, it is recommended to follow the installation procedure described in previous chapter.

5.1 Launching the Shell

The Shell can be run by an on-line command from the Windows console (resp. the Terminal console) on Windows (resp. Linux) machines.

A co-simulation file representing the case to co-simulate is to be given to the command as parameter, as shown here:

```
java -jar daccosim-shell-2018-xxx.jar input:<myfile>
   [multithreaded:{true | false}] [outputBuffer:zzz]
```

input is a mandatory parameter with value `<myfile>` designing a valid and accessible co-simulation file with extension .simx, .dngx or .dng.

multithreaded is an optional parameter to force execution on a single core of the host machine when value is `false`. Default value is `true`.

outputBuffer:zzz is an optional parameter to define the size of the I/O buffer to `zzz` KB. Default value is 8 KB.

When a .dng file is used, the FMUs are not included and should be addressed either with absolute paths or with paths relative to the folder where the java command is invoked.

.simx files and .dngx files contain a script file (extension .dng) and a folder fmu where the FMUs required for the co-simulation are located.

In comparison to .dngx files, .simx files also contain a .sim file and a .dsg file. The .sim file includes all the information needed for graphical representation of the graph into the canvas of the Editor. The .dsg file contains the file serialized according to the JSON format (serialized representation of the objects defined in the graph).

.simx and .dngx files are heavy self-supporting compressed files and they can be executed on any platform where a DACCOSIM NG Shell is available. Conversely, .dng files are widely much lighter but also less portable on other machines, because the FMUs should be exchanged by another mean between the exporting and the importing machines.

5.2 Co-simulation scripting

The DACCOSIM NG Editor is a user-friendly tool to graphically elaborate co-simulation graphs.

However, when the number of blocks or the number of connections between them is too large, a graphical approach may become very tedious. For this reason, a DSL (Domain Specific Language) has been designed within DACCOSIM NG to help handling large co-simulations schemes.

A DACCOSIM NG script is a textual file with extension .dng. This .dng file can be handmade built or generated by a preprocessing tool like a Python code.

When a valid co-simulation graph is saved from the Editor, a zipped file with extension .simx is created including a .dng script file automatically generated by the Editor. For this reason, any .simx file built from the Editor can also be run by the Shell.

Examples of .simx files are supplied in the `daccosim-use-cases-windows-2018xxx.zip` and `daccosim-use-cases-linux-2018xxx.zip` files of the distribution.
When a .dng script is generated outside the Editor, it can be zipped with all the FMUs participating to the co-simulation in a folder named fmu. The .dngx extension is advocated for this kind of zipped co-simulation files in order not to confuse them with complete .simx files from the Editor.

Examples of .dngx files are supplied in the daccosim-use-cases-windows-2018xxx.zip and daccosim-use-cases-linux-2018xxx.zip files of the distribution.

![DACCOSIM NG script language](image)

The script command lines available within DACCOSIM NG 2018 are listed in the following sub-sections.

5.2.1 Commenting a script

A co-simulation script can be commented. Each end of line starting with // is seen as a comment.

Example:

```
FMU tank1 "fmu/tankwin3264.fmu" // This is my FMU
```

When a co-simulation graph has been generated by the Editor, the header of its .dng file contains two lines similar to these ones:

```
// 2018-04-18T07:35:14.015Z
// Generated with Daccosim NG v2.0.0
```

5.2.2 Managing clusters

With DACCOSIM NG 2018, a script can only be run on a single machine.

5.2.3 Defining blocks

FMUs and factory blocks are two kind of blocks that can be defined in a co-simulation script.

The syntax for declaring an FMU is:

```
FMU <FMU-id> <file-path>
```

The file path can be either absolute or relative to a local folder named fmu and present in the archive.

Examples:
The factory blocks (Adder, Offset, Multiplier, or Gain) are not to be declared as their name is implicit. For example, when three adders are used in a graph, they are automatically named “adder”, “adder_0” and “adder_1”.

### 5.2.4 Defining external input or output connectors

External connectors are able to bring one or several variables. Input connectors define external variables entering into a co-simulation graph. The syntax is:

```
ExternalInput <connector-id>
```

Output connectors define external variables exiting from a co-simulation graph. The syntax is:

```
ExternalOutput <connector-id>
```

Examples:

- `ExternalInput maxLevel`
- `ExternalOutput water`

### 5.2.5 Declaring variables to FMUs or external connectors

Each input variable to be attached to a FMU input or an external input connector should be defined as:

```
Input <FMU-or-connector-id> <var-id> <Real | Integer | Boolean | String>
```

Examples:

- `ExternalOutput water` // This connector owns three real variables
- `Input water water-Dymola Real`
- `Input water water-CB Real`
- `Input water water-Papyrus Real`

Each output variable to be attached to a FMU output or an external output connector should be defined as:

```
Output <FMU-or-connector-id> <var-id> <Real | Integer | Boolean | String>
```

```
FMU barrell "fmu/DF2DF486257C9AE0954C17C32C29DD6A.fmu"
Output barrell Water Real
```

> Each existing FMU input should be declared within an Input command line. Only the FMU outputs to be connected according to the co-simulation graph should be declared within an Output command line.

### 5.2.6 Declaring exposed FMU variables

Each exposed FMU variable should be defined as:

```
Variable <FMU-id> <var-id> <Real | Integer | Boolean | String>
```

Example:

```
Variable Bldgl eqTradWinZn1Wall14.svf Real
```
5.2.7 Setting initial values to variables

Variables that have already been declared via an Input, Output or Variable command line can receive an initial value.

When these variables are associated to external inputs, initial setting is like:

```
InitialValue <connector-id>.<var-id> <value>
```

Example:

```
ExternalOutput water
Input water water-CB Real
InitialValue water.water-CB 0.5 // Initial value is 0.5
```

When these variables are associated to FMU parameters, the keyword beforeInit should be added so that initial setting is like:

```
InitialValue <FMU-id>.<var-id> <value> beforeInit
```

Example:

```
InitialValue Bldg1.eqTradWinZn1Wall14.svf 0.500666 beforeInit
```

When these variables are associated to FMU inputs, the keyword inInit should be added so that initial setting is like:

```
InitialValue <FMU-id>.<var-id> <value> inInit
```

Example:

```
FMU tank1 "fmu/F02FBDB3E283C958B0A38918A63F1084.fmu"
Input tank1 Valve Boolean
InitialValue tank1.Valve 0 inInit // Initial value is 0 (false)
```

5.2.8 Connecting blocks

Variables that have already been declared via an Input or Output command line can be connected together:

```
Connection <FMU-or-connector-id>.<output-var-id> < FMU-or-connector-id >.<input-var-id>
```

Example:

```
FMU tank1 "fmu/tankwin3264.fmu"
Output tank1 outputRate Real
FMU ypipe "fmu/ypipewin3264.fmu"
Input ypipe inputRate1 Real
Connection tank1.outputRate ypipe.inputRate1 // Two connected variables
```
5.2.9 Defining variables to log and logging properties

Variables that have already been declared via an Input, Output or Variable command line can be logged. Variables to log are listed according to this syntax:

\[
\text{Log } \langle \text{FMU-or-connector-id}.\langle \text{var-id} \rangle \rangle \{\langle \text{FMU-or-connector-id}.\langle \text{var-id} \rangle \rangle \}^* | \text{Log all}
\]

The keyword \text{Log all} means the automatic selection of all the variables that have already been declared into the FMU or connector.

Example:

Log water.water-CB water.water-Dymola water.water.water-Papyrus

As result files are .csv files, a cell separator (default value is “,”) and a decimal mark (default value is “." ) have to be defined according to the syntax:

\[
\text{Export } \langle \text{cell-separator} \rangle \langle \text{decimal-mark} \rangle
\]

Nevertheless note this command is not required and can be avoided if the default values are to be kept.

Example:

Export , .

5.2.10 Setting co-initialization parameters

Co-initialization is characterized by the definition of a maximum number of steps and a tolerance for the Newton-Raphson algorithm used to calculate initial input values to propagate in each SCC of the dependency graph.

The syntax is:

\[
\text{CoInit } \langle \text{max-iter} \rangle \langle \text{tolerance} \rangle
\]

Example:

CoInit 100 1.0E-5

5.2.11 Setting co-simulation parameters

Co-simulation is characterized by a time range defining a start time and a stop time according to syntax:

\[
\text{Simulation } \langle \text{start time} \rangle \langle \text{stop-time} \rangle
\]

Example:

Simulation 0.0 18.0

In addition, the integration method should be given either with constant or variable stepping.

For constant stepping, the syntax is:

\[
\text{ConstantStepper } \langle \text{step-size} \rangle
\]
For variable stepping, the exact syntax depends on the required integration method:

- **EulerStepper** `<min-step> <safety-factor> <max-nb-steps>`
- **AdamsStepper** `<min-step> <safety-factor> <max-nb-steps> <order>`

Examples:

- ConstantStepper 0.001
- EulerStepper 0.1 0.9 2147483647
- AdamsStepper 0.1 0.9 2147483647 3

For variable stepping, a tolerance can be set for each variable participating to the triggering rollbacks. The syntax is:

- **StepperVariable** `<var-id> <tolerance>`

Example:

- StepperVariable tank1.level 1.0E-5

5.2.12 Real time visualization

To be defined later (future development).

5.2.13 Running a co-simulation script

The Shell can run co-simulation scripts after unzipping .simx or .dngx file or directly by interpreting .dng files.

Syntax errors are detected before execution and reported by the tool as stated in the next figure.

![Figure 5.2: Shell execution of a co-simulation with syntactical errors (e.g. an incorrect FMU line is encountered)](image)

When a co-simulation script is correctly executed, logged information are similar to the same execution with the Editor. These information look like to:
5.2.14 An example of a co-simulation script

Here is the script illustrating the case tankYPipe available in the folder *daccosim-use-cases-windows* or *daccosim-use-cases-linux*:

```plaintext
// 2016-04-09T14:21:36.657Z
// Generated with Daccosim NG Preview1 v2.0.0
FMU tank1 "fmu/tankwin3264.fmu"
Output tank1 level Real
Output tank1 outputRate Real
Input tank1 inputRate Real
InitialValue tank1.inputRate 0.5 inInit
FMU tank2 "fmu/tankwin3264.fmu"
Output tank2 level Real
Output tank2 outputRate Real
Input tank2 inputRate Real
InitialValue tank2.inputRate 0.1 inInit
FMU ypipe "fmu/ypipewin3264.fmu"
Output ypipe outputRate Real
Input ypipe inputRate1 Real
Input ypipe inputRate2 Real
FMU tank3 "fmu/tankwin3264.fmu"
Output tank3 level Real
Output tank3 outputRate Real
Input tank3 inputRate Real
Connection ypipe.outputRate tank3.inputRate
Connection tank1.outputRate ypipe.inputRate1
Connection tank2.outputRate ypipe.inputRate2
Export , ,
Log tank1.level tank2.level tank3.level
CoInit 100 1.0E-5
ConstantStepper 0.5
Simulation 0.0 100.0
```

Figure 5.4: Example of a .dng script

---

Figure 5.3: Example of a correct Shell execution of a co-simulation from the Windows console
6 Deploying and launching co-simulations on a cluster resource

The distributed master architecture of DACCOSIM NG enables executions on distributed calculation nodes (PC clusters). This allows to take advantage of additional scalability and to improve the performance of heavy computation co-simulations.

Deploying a large co-simulation on a cluster becomes quickly complex if performed manually.

This chapter is under development (reported as future development in DACCOSIM 2019).
7 Information collected during a co-simulation

Information is available after each successful co-simulation done either by the Editor or the Shell. This information is of two kinds:

- Tracking information displayed during the co-simulation.
- Post-mortem results data computed all along the co-simulation.

7.1 Tracking information

This information is interactively displayed by the Editor or the Shell in a log window. The same information is also logged in a .log file.

The name of this .log file derives from the name of the co-simulation file used (either a .simx, .dngx or .dng file) extended with a timestamp. For example, each run of the DistrictParis900s000000100shade.dngx file will generate a .log file named DistrictParis900s000000100shade-yyyyymmddhhmmss.log.

Most important tracked information are:

- A recall of the configuration parameters (Start time, Stop time, ColInit max iterations, ColInit tolerance, Stepper method, additional parameters depending on the stepper method, multithreading information, Output buffer size, CSV decimal mark and cell separator, Output file name for results).
- A set of time stamped [INFO] messages.
- Additional information about the number of SCCs (Strongly Coupled Components) encountered in the co-simulation graph and the number of steps done to iterate the co-initialization.

The time stamped [INFO] messages are detailed here:

- Message “FMU loading time…” is displayed after:
  - Parallel unzipping .fmu file and first fmi2Instantiate() on each different physical FMU,
  - Serialized other fmi2Instantiate() on previous FMUs when multiple instantiations are required, and fmi2SetupExperiment() executed for each FMU,
- Message “FMU init time…” is displayed after:
  - Serialized propagation of inputs on outputs for all FMUs,
  - Parallel Newton-Raphson algorithm executed on each SCC,
- If the run is long enough, a series of messages “Simulation time:…” is intermediately displayed. Each message is logged every elapsed 30 s indicating the reached simulation time.
- Message “FMU co-sim time…” is displayed after the complete loop of fmi2DoStep() framed by fmi2Getxxx() and fmi2Setxxx() on all the FMUs,
- Message “FMU termination time…” is displayed after achieving the fmi2Terminate() and fmi2FreeInstance() for all the FMUs,
- Message “Co-sim finished with…” detailing the number of steps really executed and summing the total time of the run (load + init + co-sim + terminate).

Figure 7.1: Example of information logged by a co-simulation run (file .log)
Note some other information can be logged depending on the co-simulation conditions.

7.2 Results data

Results data computed by a co-simulation are registered in a .csv file.

The two first columns of this result file are always reserved to the system variables time and dt (i.e. Δt as the difference between two consecutive time steps). The other columns reference all the variables that have been selected for exportation in the “Co-simulation configuration” window.

When the running machine is a single calculation node, only one .csv result file is generated per co-simulation run. The name of this file derives from the name of the co-simulation file used (a .simx, .dngx or .dng file) extended with a timestamp. For example, each run of the tankPipe.simx file will generate a result file named tankPipe-yyyymmddhhmmss.csv.

When a cluster is used, each calculation node generates a .csv result file from the FMUs that have been running on it, as if only one “super FMU” was launched on the node. No global .csv file is built by DACCOSIM NG and if required it’s the responsibility of the user to merge all the node results files of a distributed co-simulation.

By default in the “Co-simulation configuration” window, .csv result files are configured with:

- The decimal mark as “;”
- The cell separator as “.”

All variable types defined by the standard FMI can be exported in .csv result files by DACCOSIM NG. Nevertheless, the Boolean values are represented as integers, with “0” for false and “1” for true. Likewise the enumerated values are replaced by integer values as defined in the original models.

![Figure 7.2: Extract of a result file as an example](image)

7.2.1 Post-mortem visualization

Depending on the configured decimal mark and cell separator, result files can be read by external tools like Excel knowing that conventions for this Microsoft tool are depending on languages and countries. For example, in France, Excels is often configured with the “;” as cell separator and “,” as decimal mark.
Another smart solution is to use the open-source Modelica-based modelling and simulation environment OpenModelica (https://openmodelica.org) as this tool is very effective to visualize .csv files both in Windows or Linux environment.

Note that to be readable by OpenModelica, .csv result files are to be configured in the “Co-simulation configuration” window with:

- The cell separator as “,”
- The decimal mark as “.”

For viewing, OpenModelica can be used either directly by its OMPlot application or through the editor OMEdit.

7.2.1.1 Post-mortem visualization with OMPlot in a Windows environment

After execution with the Editor, a PowerShell script is automatically invoked when clicking from the log window to visualize results with the OMPlot application distributed in the OpenModelica distribution.

⚠️ Note this script is launched only if the cell separator and decimal mark are correctly set, and the OMPlot application is available on the machine.

⚠️ Also note the system variables time and dt are also plotted but they can easily be removed if they bother the viewing of the other exposed variables by clicking on them in the legend at the right side of the graph.

After execution with the Shell, a .bat script named daccosim-viewer.bat is supplied in the folder corresponding to each use case supplied in the directory daccosim-use-cases-windows of the delivery.

This .bat script launches a .ps1 PowerShell script named daccosim-viewer.ps1 that can be copied and adapted by the user if needed. By default, this .ps1 script selects the most recent .csv result file located in the folder where the .bat and .ps1 are placed and it visualizes the content of this result file with the OMPlot application.

![Figure 7.3: a results file with variables plotted with OMPlot (time and dt are erased)](image)

7.2.1.2 Post-mortem visualization with OMPlot in a Linux environment

PowerShell Core 6.0 is a new edition of PowerShell that is cross-platform (Windows, macOS, and Linux) and open-source.

So all scripts available either in the Editor or in the directory daccosim-use-cases-linux are executable.
7.2.1.3 Post-mortem visualization with OMEdit

After a batch execution, another possibility is given by the “OpenModelica Connection Editor” (available on Windows or Linux machines) where .csv files can be opened using the “Open Result File(s)” command in the “File” menu of the OMEdit application.

The plot windows can be parameterized thanks to the “Variables Browser” as variables are structured in a tree form, each variable having a checkbox. Ticking the checkbox will plot the variable values. There is a find box on the top for filtering the variable in the tree. The filtering can be done using regular expression, wildcard and fixed string. The complete “Variables Browser” can be collapsed and expanded using the “Collapse All” and “Expand All” buttons.

The viewing with OMEdit is smart because it is possible to open and visualize several .csv files either in different plot windows or on the same graph. It is very interesting when several cluster nodes .csv files have to be simultaneously opened and variables from these different files have to be represented together.

This plotting mechanism is also very helpful when variables in result files must be compared with some other files with or without the same time series and possibly coming from other tools like Dymola, often seen as a reference in modeling and simulation.

7.2.2 Real time visualization within the Editor

In a future development, a dedicated GUI will be proposed in DACCOSIM NG for real time visualization.

This solution will also be usable for post-mortem visualization.
8 Use cases added to the distribution

Several examples are supplied beside the DACCOSIM NG 2018 main delivery. After correct installation, they should be located in the folder `daccosim-use-cases-windows` (resp. `daccosim-use-cases-linux`) when a Windows (resp. Linux) environment is used.

For memory, these folders can be downloaded at: [https://bitbucket.org/simulage/daccosim/downloads](https://bitbucket.org/simulage/daccosim/downloads) regardless DACCOSIM NG releases. Refer to chapter 3 for installation details.

Some cases from these folders are trivial examples while others are more business-oriented use cases.

All the referenced FMUs are license-free and have been built using the most recent version of Dymola. All the Modelica source models are also supplied. One case also contains FMUs exported from ControlBuild and Papyrus, but without the source models.

All the cases can be executed either on Windows or Linux environment. Each of them can be run:

- Either interactively when opened in the Editor (.simx files),
- Or in batch mode with the Shell (.simx or .dngx files).

For batch mode, a PowerShell script named `daccosim-launcher.ps1` is supplied to simplify the launching. For a Windows environment, this script can be invoked thanks to a supplied batch script `daccosim-launcher.bat`. All these scripts are present in each folder where .simx or .dngx files can be run.

Whatever the mode used (either the interactive mode with the Editor or the batch mode with the Shell), a post-mortem visualization is possible with the PowerShell script named `daccosim-viewer.ps1` under the condition the OpenModelica tool ([https://openmodelica.org/](https://openmodelica.org/)) has previously been installed on the machine. This script is able to visualize the most recent .csv file in the current folder.

Note each FMU name contains information on the runnable OS to be used on it; e.g. `tankwin3264.fmu` is able to run on a 64-bit or 32-bit Windows machine while the same FMU exported from Linux is named `tanklin3264.fmu`.  

Copyright © 2018 EDF, Monentia
Also note it is possible to run in one time all the use cases supplied with the delivery with a PowerShell script named `daccosim-launcher-all.ps1` available in the folder `daccosim-use-cases-windows` or `daccosim-use-cases-linux`. For a Windows environment, this script can be invoked thanks to a supplied batch script `daccosim-launcher-all.bat`.

The purpose of this .ps1 script is to list all the .simx and .dngx files accessible in the current directory and its subdirectories and sequentially run each of them.

### 8.1 Co-initialization examples

This paragraph details some co-initialization examples available in the sub-folder `daccosim-use-cases/1-coinit-only` of the DACCOSIM NG 2018 distribution.

Note for these cases, a post-mortem visualization script is not useful as only the starting point is calculated.

#### 8.1.1 equationsPair case

This example illustrates the co-initialization of a system composed of two coupled FMUs (resolution of a system with two equations and two unknowns):

- equation1 model calculates $x_2$ from $x_1$ according to the equation: $2x_1^2 + 5x_2 = 42$
- equation2 model calculates $x_1$ from $x_2$ according to the equation: $x_1 - 6x_2 = 4$

This example is interesting as the system resolution implies to solve an algebraic loop between the two FMUs.
8.1.1.1 Graphs and simulation properties

Here is the co-simulation graph for the equationsPair case:

![Co-simulation graph](image)

Figure 8.2: Co-simulation graph from the Editor for the equationsPair case

Default co-simulation parameters are enough for this case.

Exported variables are:
- `equation1.x1, equation1.x2` from FMU equation1
- `equation2.x1, equation2.x2` from FMU equation2

8.1.1.2 Discussion on the results

The `equationsPair-xxx.csv` file shows the co-initialization result as the resolution of the two equations and two unknowns system: \((x_1, x_2) \approx (4.56, 0.09)\).

The `equationsPair-xxx.log` file shows that 8 iterations have been done to convergence in solving the single contained SCC.

8.1.2 tripleEquation case

This example illustrates the co-initialization of a system composed of three coupled FMUs (intersection between three planes):
- plane1 model calculates \(z\) from \(x\) and \(y\) according to the equation: \(6x - y + z = 30\)
- plane2 model calculates \(y\) from \(x\) and \(z\) according to the equation: \(2x + 2y - 3z = 27\)
- plane3 model calculates \(x\) from \(y\) and \(z\) according to the equation: \(x - 3y + 2z = -9\)

As for the previous case, this example is interesting as it implies to solve several algebraic loops between the three FMUs.
8.1.2.1 Graphs and simulation properties

Here is the co-simulation graph for the tripleEquation case:

![Co-simulation graph](image)

Figure 8.3: Co-simulation graph from the Editor for the tripleEquation case

Default co-simulation parameters are enough for this case.

Exported variables are:
- `plane1.x, plane1.y, plane1.z` from FMU plane1
- `plane2.x, plane2.y, plane2.z` from FMU plane2
- `plane3.x, plane3.y, plane3.z` from FMU plane3

8.1.2.2 Discussion on the results

The `tripleEquation-xxx.csv` file shows the co-initialization result as the resolution of the calculated coordinates of the three planes intersection point: \((x, y, z) \approx (6, 3, -3)\).

The `tripleEquation-xxx.log` file shows that only one iteration has been enough to convergence in solving the single contained SCC.

8.1.3 complexCoinitCase1

This example illustrates a more complex propagation of data from a source FMU to an external output in a graph mixing FMUs and factory blocks:
- source model generates a constant output with value 3
- each propagation model generates an output as a multiple of its input value (e.g. propagation3 outputs 3 times its input value)
8.1.3.1 Graphs and simulation properties

Here is the co-simulation graph for the complexCoinitCase1 case:

![Co-simulation graph for complexCoinitCase1](image)

**Figure 8.4: Co-simulation graph from the Editor for the complexCoinitCase1 case**

Default co-simulation parameters are enough for this case.

Exported variable is:
- `myOutput.value.value` from external output `myOutput`

8.1.3.2 Discussion on the results

The `case1-xxx.csv` file shows the co-initialization result as a propagation and combination of data on an external output: `myOutput.value = 477`.

The `case1-xxx.log` file shows that no iteration has been done as the case contains no SCC.

8.1.4 complexCoinitCase2

This example illustrates a more complex propagation of data from several source FMUs to an external output in a graph mixing FMUs and factory blocks with some algebraic loops:

- each pair of source models generates a vector of values \((p_1, p_2)\) as inputs for an associated pair of equations:
  - for source11 and source 21 models, \((p_1, p_2) = (0, 0)\)
  - for source12 and source 22 models, \((p_1, p_2) = (1, 5)\)
  - for source13 and source 23 models, \((p_1, p_2) = (2, 10)\)
- each pair of equation models solves a system composed of \(\begin{cases} 2x_1 - x_2 = p_1 \\ 4x_1 + x_2 = 6 + p_2 \end{cases}\)
- all calculated \(x_1\) (resp. \(x_2\)) values are then summed and the two summations are finally multiplied
8.1.4.1 Graphs and simulation properties

Here is the co-simulation graph for the complexCoinitCase2 case:

![Co-simulation graph](image)

**Figure 8.5: Co-simulation graph from the Editor for the complexCoinitCase2 case**

Default co-simulation parameters are enough for this case.

Exported variable is:

- `myOutput.value.value from external output myOutput`

8.1.4.2 Discussion on the results

The `case2-xxx.csv` file shows the co-initialization result as a propagation and combination of data with algebraic loops on an external output: `myOutput.value ≈ 54`.

The `case2-xxx.log` file shows that only one iteration has been enough for each of the 3 SCCs contained in the case.

8.2 Academic co-simulation examples

This paragraph details some academic co-simulation examples available in the sub-folder `daccosim-use-cases/2-academic` of the DACCOSIM NG 2018 distribution.

8.2.1 tankYPipe case

This example illustrates a non-stiff case without internal events. It is composed of two FMUs, one being duplicated three times.

Each tank has a 2 m² floor area with a hole of 0.1 m² and an initial water level of 3 m. tank1 and tank2 each calculates their output flow rate with respectively an initial input flow rate of 0.5 m³ and 0.1 m³. They pour water in a Y-pipe model that simply adds the two input rates as an output flow rate entering into tank3.
tank1, tank2 and tank3 are three instances of the same Modelica model `tank.mo` with a low data processing weight. Several variants of this model are given (only for Windows) with dummy differential equations included to artificially increase the time calculation of the numerical integration. Of course all the results are exactly the same when the co-simulation is done replacing one or several instances of the small model by some of the variant models.

These model variants all have 40,000 dummy variables but the proportion of algebraic and differential equations depends on each model. For example tank15000.mo model contains $30,000 = 15,000 \times 2$ differential equations and $10,000 = 40,000 - 30,000$ algebraic ones.

### 8.2.1.1 Graphs and simulation properties

Here is the co-simulation graph for the tankYPipe case:

![Co-simulation graph](image)

**Figure 8.6: Co-simulation graph from the Editor for the tankYPipe case**

Default co-simulation parameters are enough apart from the followings:

- **stop time** = 100 s
- **constant step size** = 0.5 s
- **cell separator** = ,

### 8.2.1.2 Discussion on the results

The obtained results in the `tankYPipe-xxx.csv` file seem realistic and in line with the expectations.

![Results graph](image)

**Figure 8.7: Results for the academic tankYPipe case**

### 8.2.2 tankYPipeWithRequirements case

To be defined later.
8.2.3  **events case**

This example illustrates the capability to co-simulate in one run independent systems and also to deal with internal events in FMUs. Each model contains different type of events:

- *m1* model is the famous bouncing ball case with several state events
- *m2* model contains two state events
- *m3* model contains three time events

#### 8.2.3.1  Graphs and simulation properties

Here is the co-simulation graph for the events case:

![Co-simulation graph from the Editor for the events case](image)

Default co-simulation parameters are enough apart from the followings:

- \textit{stop time} = 10 \, s
- \textit{constant step size} = 0.01 \, s
- \textit{cell separator} = ,

#### 8.2.3.2  Discussion on the results

Waiting for an exact internal event detection, the visualization from the \texttt{events-xxx.csv} file shows all the events are correctly detected only under the condition a small enough constant step size is used.

![Events handing for the academic events case](image)
Exact events handling is under consideration in the FMI Project managed by the Modelica Association. In advance, some improvements are unofficially being tested by a working group composed of Dassault Systèmes, EDF, CentraleSupélec and MONENTIA (not yet implemented in DACCOSIM NG 2018).

8.2.4 stairBouncingBall case

This example is detailed in the paper “Discrete Event Simulation of Hybrid Systems” published by E. Kofman in 2004. It represents a typical discontinuous example as the ball moves in two dimensions bouncing downstairs and the bouncing condition depends on both variables (length and height).

The case is an example of non-stiff problem with FMU internal events.

8.2.4.1 Graphs and simulation properties

Here is the co-simulation graph for the stairBouncingBall case:

![Co-simulation graph](image)

Default co-simulation parameters are enough apart from the followings:

- **stop time** = 10 s
- **constant step size** = 0.01 s
- **cell separator** = ,

8.2.4.2 Discussion on the results

The obtained results from the **stairBouncingBall-xxx.csv** file are in conformance with the expectations, where the ball trajectory is visible in relation to the level of the stairs.

![Ball trajectory](image)
8.2.5 dCACInverter case
This example is detailed in the paper “Discrete Event Simulation of Hybrid Systems” published by E. Kofman in 2004. It represents an inverter circuit coupled with a pulse width modulation component and illustrates the case of a stiff problem where a very small step size is required to get enough accuracy on results.

8.2.5.1 Graphs and simulation properties
Here is the co-simulation graph for the dCACInverter case:

![Co-simulation graph from the Editor for the dCACInverter case](image)

Default co-simulation parameters are enough apart from the followings:
- stop time = 1.5 s
- constant step size = 0.0001 s
- cell separator = ,

8.2.5.2 Discussion on the results
The obtained results in the dCACInverter-xxx.csv file seems realistic and in line with the expectations.

8.2.6 tankBarrel case
This example is detailed in the paper “Hybrid Co-Simulation of FMUs using DEV&DESS in MECSYCO” published by B. Camus in 2016.

The case illustrates a hybrid co-simulation mixing control and physics exhibiting FMUs exported from heterogeneous tools (Dymola, ControlBuild or Papyrus) and whose models are built with different modelling languages (Modelica, IEC 61131-3 or UML/SysML) for the control part of the case.

Please note no FMUs from ControlBuild have been exported for Linux.

Copyright © 2018 EDF, Monentia
8.2.6.1 Graphs and simulation properties

Here is the co-simulation graph for the tankBarrel case:

![Co-simulation graph](image)

**Figure 8.16: Co-simulation graph from the Editor for the tankBarrel case (Windows only)**

Default co-simulation parameters are enough apart from the followings:

- `stop time = 18 s`
- `constant step size = 0.001 s`
- `cell separator = ,`

8.2.6.2 Discussion on the results

The obtained results from the `tankBarrel-xxx.csv` file are in conformance with the control behavior on the physics part of the system.
8.2.7 tankBarrelWithTelecoms case

This example is very similar to the previous case tankBarrel with an additional FMU developed with components from the EDF TelSysPro library in order to model a 100 ms latency effect on control commands.

A comparison between the two systems (with and without latency induced by a very simple telecom network) can be easily done as the previous tankBarrel is co-simulated in the same run.

8.2.7.1 Graphs and simulation properties

Here is the co-simulation graph for the tankBarrelWithTelecoms case:

```
\begin{figure}
  \centering
  \includegraphics[width=\textwidth]{co_simulation_graph.png}
  \caption{Co-simulation graph from the Editor for the tankBarrelWithTelecoms case}
\end{figure}
```

Default co-simulation parameters are enough apart from the followings:

- \textit{stop time} = 18 s
- \textit{constant step size} = 0.001 s
- \textit{cell separator} = ,
8.2.7.2 Discussion on the results
The obtained results from the tankBarrelWithTelecoms-xxx.csv file illustrate the impact of latency for commands from the control part of the system.

![Figure 8.19: Impact of latency for the academic tankBarrelWithTelecoms case](image)

8.2.8 dynamicEquationsPair case
This example extends the case about the resolution of a system with two equations and two unknowns with time-dependent equations:

- equation1 model calculates $x_2$ from $x_1$ according to the equation: $(x_2 - 2) \times time + x_1 + x_2 = 10$
- equation2 model calculates $x_1$ from $x_2$ according to the equation: $2 \times x_1 - (x_2 - \sqrt{time}) = 11$

This example is interesting as the system resolution implies to solve an algebraic loop between the two FMUs at initialization and then to permanently check the deviations during the whole co-simulation.

8.2.8.1 Graphs and simulation properties
Here is the co-simulation graph for the dynamicEquationsPair case:

![Figure 8.20: Co-simulation graph from the Editor for the dynamicEquationsPair case](image)

Default co-simulation parameters are enough apart from the followings:

- stop time = 18 s
- constant step size = 0.001 s
- cell separator = ,
8.2.8.2 Discussion on the results

The obtained results from the `dynamicEquationsPair-xxx.csv` file are in conformance with the cont

![Figure 8.21: Dynamic system resolution for the academic dynamicEquationsPair case](image)

8.2.9 heatedRoom case

This example is detailed in the paper “Efficient Monte Carlo simulation of stochastic hybrid systems” published by M. Bouissou in 2015.

The system implements a stochastic behavior for one of these components as the heater has a constant failure rate $\lambda = 0.01/h$, and a constant repair rate $\mu = 0.1/h$.

⚠️ Note this case requires the use of the commercial library Design (edited by Dassault Systèmes) for rebuilding the FMUs if needed.

8.2.9.1 Graphs and simulation properties

Here is the co-simulation graph for the heatedRoom case:

![Figure 8.22: Co-simulation graph from the Editor for the heatedRoom case](image)

Default co-simulation parameters are enough apart from the followings:

- `stop time = 1000 h`
- `constant step size = 0.01 h`
- `cell separator = ,`
Exported variable are:

- `heatedRoom.T` from FMU `heatedRoom`
- `heatedController.T_high`, `heatedController.T_low` from FMU `heatedController`

### 8.2.9.2 Discussion on the results

The obtained results from the `heatedRoom-xxx.csv` file are in conformance with the expectations.

![Figure 8.23: Random effect on the room temperature given for the academic heatedRoom case](image)

### 8.2.10 heatedSystemWithTwoBuildingsAndTelecoms case

This case illustrates the capability to handle a long duration co-simulation (30 days in this example).

The system implements a stochastic behavior for each of the telecom network developed with components from the EDF TelSysPro library (user equipments, links and intermediate nodes), each having a constant failure rate $\lambda = 0.01/h$, and a constant repair rate $\mu = 0.1/h$.

### 8.2.10.1 Graphs and simulation properties

Here is the co-simulation graph for the heatedSystemWithTwoBuildingsAndTelecoms case:

![Figure 8.24: Co-simulation graph from the Editor for the heatedSystemWithTwoBuildingsAndTelecoms case](image)

Default co-simulation parameters are enough apart from the followings:

- `stop time` = 2592000 s
- `constant step size` = 60 s
- `cell separator` = ,

Copyright © 2018 EDF, Monentia
8.2.10.2 Discussion on the results

Due to the randomness of the telecom components, the results are different at each run. The impact on the internal temperature of rooms in the two buildings is visible in the `heatedSystemWithTwoBuildingsAndTelecoms-xxx.csv` file as this temperature can be lower than the minimal comfort temperature required by users (288.15 $K$) and even lower than the minimal temperature defined at the controller side (285.15 $K$).

Figure 8.25: Internal temperature in one room for the academic heatedSystemWithTwoBuildingsAndTelecoms case

8.2.11 multipleFMU case

This theoretical case illustrates the capability to instantiate hundreds of times the same FMU whose flag `canBeInstantiatedOnlyOncePerProcess` is set to false.

8.2.11.1 Graphs and simulation properties

No co-simulation graph is available for the multipleFMU case as the co-simulation has been defined by a handmade .dng script file zipped with the FMUs in a .dngx file. This script instantiates 1000 times the FMU from stairBouncingBall model.

Default co-simulation parameters are enough apart from the followings:

- `stop time = 10 s`
- `constant step size = 0.01 s`
- `cell separator = ,`

Exported variables are:

- `stairBouncingBall001.y, stairBouncingBall001.stair from FMU instance stairBouncingBall001`
- `stairBouncingBall1000.y, stairBouncingBall1000.stair from FMU instance stairBouncingBall1000`
8.2.11.2 Discussion on the results

The results from the `multipleFMU-xxx.csv` file are identical for the two plotted instances and they are identical to the reference basic case `stairBouncingBall`.

![Figure 8.26: Ball trajectory for the academic multipleFMU case](image)

8.2.12 multipleCnx case

This theoretical case illustrates the capability to define thousands of connections between two FMUs.

8.2.12.1 Graphs and simulation properties

No co-simulation graph is available for the multipleCnx case as the co-simulation has been defined by a handmade `.dng` script file zipped with the FMUs in a `.dngx` file. This script defines 10000 connections aiming to transfer 10000 variables of real type:

- gene1toN model generates 10000 outputs, each values being a unique integer in the range \([1, 10000]\)
- sigmaN model calculates the sum of all these inputs and compares it to the theoretical result being \(\frac{10000 \times 10001}{2}\)

Default co-simulation parameters are enough apart from the followings:

- \(\text{stop time} = 1 \text{s}\)
- \(\text{cell separator} = ,\)

Exported variables are:

- \(\text{sigmaN.sigmaN sigmaN.zero from FMU sigma}\)

8.2.12.2 Discussion on the results

As expected, the variable sigmaN.zero in the `multipleCnx-xxx.csv` file is equal to zero.
8.3 Industrial co-simulation examples

This paragraph details some industrial co-simulation examples available in the sub-folder *daccosim-use-cases/3-industrial* of the DACCOSIM NG 2018 distribution.

8.3.1 DistrictParis900s000000100shade case

The purpose of this example is to illustrate a real business use case related to district heating and cooling energy in buildings.

The system represents a district composed of 23 buildings (with only 2 adjoining walls) with inter-building long-wave radiation coupling and solar flux pre-processed per facade to account for shadings and reflections.

This test case was notably used in “Adapting building heating and cooling power need models at the district scale” published by Loïc Frayssinet as his PhD thesis, INSA de Lyon in 2018. More information about the district characteristics can be found in this PhD reference.

Some internal components of the building models are extracted from BuildSysPro, an open source Modelica library developed by EDF for buildings, districts and energy systems modelling ([https://github.com/edf-enerbat/BuildSysPro](https://github.com/edf-enerbat/BuildSysPro)). The building energy models are automatically generated at district scale from geographical information system with the MoDEM plateform, also introduced in the previous reference.

A *globalscript.mos* script is supplied in folder *ScriptToGenFMUs* to activate other .mos scripts in order to generate the FMUs from the supplied Modelica source models (given in folder *ModelicaModels*). Please note:

- these .mos scripts are to be adapted to the directory definition of the running machine
- some parameters are set before the FMU translation in the .mos files as they refer to external resources or they modify the size of some structure input vectors and hence cannot be modified after the translation. As usual, more static parameters are directly set in the DACCOSIM NG script file
- Each FMU is built with the following resources:
  - a weather file defined during the FMU parameterisation (.txt file)
  - solar fluxes defined within the Modelica Model (.csv file)

8.3.1.1 Graphs and simulation properties

No co-simulation graph is available for the DistrictParis900s000000100shade case as the co-simulation script file has been automatically generated by the MoDEM platform (Python codes).
Default co-simulation parameters are enough apart from the followings:

- \( \text{stop time} = 34128000 \, \text{s} \)
- \( \text{constant step size} = 900 \, \text{s} \)
- \( \text{cell separator} = , \)

Note: the normal duration for this example is one year (about 34128000 s) but time has been reduced to 604800 s (one week) in the given file to keep a reasonable execution time.

Exported variables correspond to the heating and cooling power needs (W) of each building of the district. They are as follow:


8.3.1.2 Discussion on the results

Results are in conformance with the business expectations.

![Figure 8.28: Power needs for each building for the industrial DistrictParis900s000000100shade case (one week duration)](image)
Figure 8.29: Power needs for each building for the industrial District Paris 900 s000000100s shade case (one year duration)
9 Some known limits

Representation for Reals

DACCOSIM NG calculations are based on floating-point numbers to be in conformance with the \texttt{fmi2DoStep()} primitive of the FMI-CS >2.0 standard where both \texttt{currentCommunicationPoint} and \texttt{communicationStepSize} are represented as Real (double).

Floating-point represent numbers with a fixed number of bits. For example, 53 bits are used in double-precision floating-point, so the otherwise infinite representation is rounded to 53 significant bits.

0.1 is a typical example of inaccuracy in representation for floating points in double-precision.

Let’s write it in binary truncated to 57 significant bits:

0.0001100110011001100110011001100110011001100110011001

Bits 54 and beyond total to greater than half the value of bit position 53, so this rounds up to:

0.0001100110011001100110011001100110011001100110011001101

In decimal, this is equivalent to:

0.10000000000000000000005551115123125782702181583404541015625

which is slightly greater than 0.1

This default is well known in data computing but may induce confusion in user mind as it is especially visible in DACCOSIM NG results when a constant step size method is required for the co-simulation.

Very large co-simulations

Some memory overflows can be due to a limitation of the maximum Java Heap (JavaHeapSpace) that can be tuned and changed when invoking the shell.jar by using the Xmx parameter. For instance:

```
java -Xmx8000m -jar daccosim-shell-2018-xxx.jar input:<myfile>
```

will reserve 8 GB of RAM for the process. The user can increase it as much as desired but he must also pay attention to have enough RAM to avoid swapping.

The limit for the number of FMU blocks involved in a co-simulation has not been pinned down and is very dependent on the FMUs size and the memory available on the machine. Note that when hundreds and hundreds of FMUs are to be used, a cluster environment should be preferable.

- oOo -
10 Annex about variable stepping implemented in DACCOSIM NG 2018

Variable stepping methods implemented in DACCOSIM NG 2018 are single-step quantization methods or Adams-Bashforth multiple-step methods.

10.1 Quantization methods for variable stepping

This chapter will be written in 2019 when hysteretic quantization functions will be implemented in DACCOSIM NG derived from QSS methods.

These new methods will replace the historical Euler method implemented for years in DACCOSIM.

10.2 Euler and Adams-Bashforth-Moulton method

Assuming an FMU supplies a numerical method \( \Phi \) with order \( m \), DACCOSIM NG 2018 implements a variable stepping method based on the calculation of two approximations with same step size but another embedded method of different consistency order.

10.2.1 Calculating two approximations with embedded methods

Denoting \( Y(t_{i+1}) \) as the exact solution of the integration at the communication point \( t_{i+1} \) and \( \Phi \) (resp. \( \Phi' \)) the \( m \)-order (resp. \( p \)-order) numerical methods embedded to calculate an approximation at the same communication point, the equations are:

\[
\begin{align*}
Y(t_{i+1}) &= \Phi(h_i) + K h_i^{m+1} + O(h_i^{m+1}) \\
Y(t_{i+1}) &= \Phi'(h_i) + K' h_i^{p+1} + O(h_i^{p+1})
\end{align*}
\]

In this system, \( \Phi \) stands for the numerical method implemented in the FMU and \( \Phi' \) for another embedded method.

DACCOSIM NG 2018 implements two different additional methods for \( \Phi' \):

- The simple explicit Euler method;
- And a more sophisticated and multi-step order method.

10.2.1.1 Application to the Euler method

Assuming that \( p < m \):

\[
\begin{align*}
Y(t_{i+1}) &= \Phi(h_i) + O(h_i^{p+1}) \\
Y(t_{i+1}) &= \Phi'(h_i) + K' h_i^{p} + O(h_i^{p+1})
\end{align*}
\]

Subtracting the second equation from the first one in the system, we can find an evaluation of the local error:

\[ K' h_i^{p} \approx \Phi(h_i) - \Phi'(h_i) \]

This quantity is also an estimation of the truncation error done with the numerical embedded methods \( \Phi \) and \( \Phi' \): \( \text{EST} \approx |K' h_i^{p}| \).

Knowing \( TOL \) as the acceptable tolerance, the optimum step size \( h_{opt} \) can be calculated via the same kind of formula: \( TOL \approx |K' h_{opt}^{p}| \).

By dividing the two quantities, we can set: \( h_{opt} = h_i \left( \frac{TOL}{\text{EST}} \right)^{\frac{1}{p}} \).

As the Euler method is 1-order, \( p = 1 \) and \( h_{opt} = h_i \left( \frac{TOL}{\text{EST}} \right) \).

In DACCOSIM NG 2018, the maximum theoretical step size is calculated for each output and each exposed continuous variable \( j \) with a formula very similar to the one used for the Richardson extrapolation, that is to say, with the same notations:

\[
h_{\text{max},j} = \begin{cases} 
\max \left( \min_{\text{step}} \min \left( s_f \cdot h_i \left( \frac{TOL_j}{\text{EST}_j} \right), \max_{\text{step}} \right) \right) \\
h_i \cdot \text{IF} \left( s_f \cdot h_i \left( \frac{TOL_j}{\text{EST}_j} \right) < h_i \ \text{AND} \ (\text{EST}_j < TOL_j) \right) 
\end{cases}
\]

DACCOSIM NG 2018 then select the next step size for the FMU as: \( h_{\text{max}} = \min_j (h_{\text{max},j}) \).

And the rollback conditions are exactly the same as for the Richardson extrapolation.

10.2.1.2 Application to the Adams-Bashforth-Moulton method

k-step Adams-Bashforth formula are known to be explicit k-order methods while k-step Adams-Moulton ones are implicit k+1-order. To make this formula implicit, we can use a predictor-corrector pair expressed as a variable step explicit k-step Adams-Bashforth predictor and a k-step Adams-Moulton corrector. As given by Lopez et al., 2010 the predictor and the corrector are:

\[
\begin{align*}
p(t_{i+1}) &= \Phi(h_i) + h_i \sum_{j=0}^{k-1} g_j(i) \beta_j(i) \varphi_j(i) \\
\varphi^*(h_i) &= p(t_{i+1}) + h_i g_k(i) \varphi_k(i+1)
\end{align*}
\]

Where the different coefficients are given by the following recurrences:

\[
\begin{align*}
\beta_0(i) &= 1 \\
\beta_j(i) &= \beta_{j-1}(i) \frac{t_{i+1} - t_{i-j+1}}{t_i - t_{i-j}} \\
\varphi_0(i) &= \Phi(h_i) \\
\varphi_j(i) &= \varphi_{j-1}(i) - \beta_{j-1}(i-1) \varphi_{j-1}(i-1) \\
c_{0,q}(t_{i+1}) &= \frac{1}{q} \\
c_{j,q}(t_{i+1}) &= c_{j-1,q}(t_{i+1}) - c_{j-1,q+1}(t_{i+1}) \frac{h_i}{t_{i+1} - t_{i-j+1}} \quad g_j(i) = c_{j,1}(t_{i+1})
\end{align*}
\]

It is easy to check that:

- the coefficients \( \beta_j(i) \) and \( g_j(i) \) are scalar and independent from any function values \( \Phi(h_i) \);
- the \( \varphi_j(i) \) coefficients depend on the \( k - 1 \) previous function values \( \Phi(h_i) \).

Assuming the second embedded method \( \Phi^* \) is a p-order one with \( p > m \), the system becomes:

\[
\begin{align*}
Y(t_{i+1}) &= \Phi(h_i) + K h_i^m + O(h_i^{m+1}) \\
Y(t_{i+1}) &= \Phi^*(h_i) + O(h_i^{m+1})
\end{align*}
\]

And the optimum step size \( h_{\text{opt}} \) is calculated as: \( h_{\text{opt}} = s_f h_i \left( \frac{TOL}{\text{EST}} \right)^{\frac{1}{m}} \).

To limit the number of steps to keep in memory, we can choice \( p = m + 1 \) and then adopt a m-step Adams-Moulton formula.

In DACCOSIM NG 2018 the maximum theoretical step size is calculated for each output and each exposed continuous variable \( j \) with a formula exactly identical to the one used for the Richardson extrapolation, that is to say, with the same notations:

\[
h_{\text{max},j} = \begin{cases} 
\max \left( \min_{\text{step}} \min \left( s_f \cdot h_i \left( \frac{TOL_j}{\text{EST}_j} \right)^{\frac{1}{m}}, \max_{\text{step}} \right) \right) \\
h_i \cdot \text{IF} \left( s_f \cdot h_i \left( \frac{TOL_j}{\text{EST}_j} \right)^{\frac{1}{m}} < h_i \ \text{AND} \ (\text{EST}_j < TOL_j) \right) 
\end{cases}
\]

With:

- \( \max_{\text{step}} \) being an upper value of the step size, \( \min_{\text{step}} \) a lower value of the step size and \( s_f \) a safety factor, all given as parameters in the tool;

Copyright © 2018 EDF, Monentia
• \( h_i \) being the current step size;
• \( TOL_j \) being the given relative tolerance and \( EST_j \) the calculated truncation of the relative error, for each \( j \) indexed variable.

DACCOSIM NG 2018 then select the next step size for the FMU as: \( h_{\text{max}} = \min_j(h_{\text{max},j}) \).

When to rollback?

When \( h_{\text{max}} \geq h_i \), no rollback is required and \( h_{\text{max}} \) becomes the step size for the next iteration: \( h_{i+1} = h_{\text{max}} \). When \( h_{i+1} < h_i \), a rollback is to be done, the iteration is erased and performed again with a smaller step size: \( h_i = h_{\text{max}} \).

Note the IF condition in the formula giving \( h_{\text{max},j} \) is empirically set to avoid too frequent rollbacks.