
OMSimulator Documentation

Release v3.0.0.post203-gc218ca5

Lennart Ochel

Oct 07, 2026

CONTENTS

1	Introduction	1
2	OMSimulator	3
2.1	OMSimulator Flags	3
2.2	Examples	6
2.2.1	Running a Python script	6
2.2.2	Running an SSP file	6
2.2.3	Running an FMU	7
3	OMSimulatorLib	9
3.1	RunFile	9
3.2	activateVariant	9
3.3	addBus	9
3.4	addConnection	9
3.5	addConnector	10
3.6	addConnectorToBus	10
3.7	addResources	10
3.8	addSignalsToResults	10
3.9	addSubModel	10
3.10	addSystem	10
3.11	compareSimulationResults	10
3.12	copySystem	11
3.13	delete	11
3.14	deleteConnection	11
3.15	deleteConnectorFromBus	11
3.16	deleteResources	11
3.17	doStep	12
3.18	duplicateVariant	12
3.19	export	12
3.20	exportDependencyGraphs	12
3.21	exportSSMTemplate	13
3.22	exportSSVTemplate	13
3.23	exportSnapshot	13
3.24	freeMemory	13
3.25	getBoolean	13
3.26	getBus	13
3.27	getComponentType	14
3.28	getConnections	14
3.29	getConnector	14

3.30	getDirectionalDerivative	14
3.31	getElement	14
3.32	getElements	14
3.33	getFMUInfo	15
3.34	getFixedStepSize	15
3.35	getInteger	15
3.36	getModelState	15
3.37	getReal	15
3.38	getResultFile	15
3.39	getSolver	15
3.40	getStartTime	16
3.41	getStopTime	16
3.42	getString	16
3.43	getSubModelPath	16
3.44	getSystemType	16
3.45	getTime	16
3.46	getTolerance	16
3.47	getVariableStepSize	17
3.48	getVersion	17
3.49	importFile	17
3.50	importSnapshot	17
3.51	initialize	17
3.52	instantiate	17
3.53	list	17
3.54	listUnconnectedConnectors	18
3.55	listVariants	18
3.56	loadSnapshot	18
3.57	newModel	18
3.58	newResources	18
3.59	referenceResources	18
3.60	removeSignalsFromResults	19
3.61	rename	19
3.62	replaceSubModel	19
3.63	reset	19
3.64	setActivationRatio	20
3.65	setBoolean	20
3.66	setBusGeometry	20
3.67	setCommandLineOption	20
3.68	setConnectionGeometry	20
3.69	setConnectorGeometry	20
3.70	setElementGeometry	20
3.71	setFixedStepSize	21
3.72	setInteger	21
3.73	setLogFile	21
3.74	setLoggingCallback	21
3.75	setLoggingInterval	21
3.76	setLoggingLevel	21
3.77	setMaxLogFileSize	21
3.78	setReal	22
3.79	setRealInputDerivative	22
3.80	setResultFile	22

3.81	setSolver	22
3.82	setStartTime	22
3.83	setStopTime	22
3.84	setString	23
3.85	setTempDirectory	23
3.86	setTolerance	23
3.87	setUnit	23
3.88	setVariableStepSize	23
3.89	setWorkingDirectory	24
3.90	simulate	24
3.91	simulate_realtime	24
3.92	stepUntil	24
3.93	terminate	24
4	OMSimulatorPython3	25
4.1	Core Capabilities	25
4.2	Command Line Interface	25
4.2.1	Examples	26
4.3	Quick start Example	26
4.4	SSP	28
4.5	SSD	29
4.6	SSV	29
4.7	SSM	30
4.8	FMU	30
4.9	Component	32
4.10	addSystem	33
4.11	addConnector	33
4.12	addConnection	34
4.13	addResource	35
4.14	addComponent	35
4.15	addSSVReference	37
4.16	removeSSVReference	38
4.17	swapSSVReference	39
4.18	listSSVReference	40
4.19	exportSSVTemplate	40
4.20	exportSSVTemplate	42
4.21	setValue	43
4.22	getValue	45
4.23	mapParameter	46
4.24	duplicate and activate Variant	47
4.25	getVariant	48
4.26	listResource	49
4.27	deleteResource	49
4.28	delete	50
4.29	instantiate	51
4.30	setResultFile	52
4.31	initialize	53
4.32	Solver	53
4.32.1	Available Solvers	54
4.32.2	Solver Settings	54
4.32.3	Solver Assignment	55

4.33	simulate	56
4.34	terminate and delete	56
5	OpenModelicaScripting	59
5.1	Examples	59
5.2	OpenModelica Scripting Commands	59
5.3	addBus	59
5.4	addConnection	60
5.5	addConnector	60
5.6	addConnectorToBus	60
5.7	addSignalsToResults	60
5.8	addSubModel	61
5.9	addSystem	61
5.10	compareSimulationResults	61
5.11	copySystem	61
5.12	delete	62
5.13	deleteConnection	62
5.14	deleteConnectorFromBus	62
5.15	export	62
5.16	exportDependencyGraphs	62
5.17	exportSnapshot	62
5.18	freeMemory	62
5.19	getBoolean	63
5.20	getFixedStepSize	63
5.21	getInteger	63
5.22	getModelState	63
5.23	getReal	63
5.24	getSolver	63
5.25	getStartTime	63
5.26	getStopTime	63
5.27	getSubModelPath	64
5.28	getSystemType	64
5.29	getTime	64
5.30	getTolerance	64
5.31	getVariableStepSize	64
5.32	getVersion	64
5.33	importFile	64
5.34	importSnapshot	65
5.35	initialize	65
5.36	instantiate	65
5.37	list	65
5.38	listUnconnectedConnectors	65
5.39	loadSnapshot	65
5.40	newModel	65
5.41	removeSignalsFromResults	66
5.42	rename	66
5.43	reset	66
5.44	setBoolean	66
5.45	setCommandLineOption	66
5.46	setFixedStepSize	66
5.47	setInteger	67

5.48	setLogFile	67
5.49	setLoggingInterval	67
5.50	setLoggingLevel	67
5.51	setReal	67
5.52	setRealInputDerivative	67
5.53	setResultFile	68
5.54	setSolver	68
5.55	setStartTime	68
5.56	setStopTime	68
5.57	setTempDirectory	68
5.58	setTolerance	69
5.59	setVariableStepSize	69
5.60	setWorkingDirectory	69
5.61	simulate	69
5.62	stepUntil	69
5.63	terminate	70
6	Graphical Modelling	71
6.1	OMEdit	71
6.1.1	Architecture	71
6.1.2	New SSP Model	71
6.1.3	Open SSP Model	71
6.1.4	Add System and Sub-Systems	73
6.1.5	Add SubModel	73
6.1.6	Replace SubModel	73
6.1.7	Simulation Setup	77
6.1.8	Simulate	82
6.1.9	Dual Mass Oscillator Example	82
6.2	OMSimulatorGui	82
6.2.1	Requirements and Enabling the Build	82
6.2.2	Installing with pip	83
6.2.3	Launching	84
6.2.4	Main Window	84
6.2.5	Creating and Opening Models	85
6.2.6	The System Tree	85
6.2.7	The Diagram Canvas	87
6.2.8	Element (Component) Properties	88
6.2.9	Parameter Files (SSV/SSM)	88
6.2.10	Resources	94
6.2.11	Variants	94
6.2.12	Simulation Settings	96
6.2.13	Options (Working Directory and Command Line Options)	96
6.2.14	Simulating and Viewing Results	98
6.2.15	Current Limitations	99

INTRODUCTION

The OMSimulator project is a FMI-based co-simulation tool. It supports large-scale simulation and virtual prototyping using models from multiple sources utilizing the FMI standard. It is integrated into OpenModelica but also available stand-alone, i.e., without dependencies to Modelica specific models or technology. OMSimulator provides an industrial-strength open-source FMI-based modelling and simulation tool. Input/output ports of FMUs can be connected, ports can be grouped to buses, FMUs can be parameterized and composed, and composite models can be exported according to the SSP (System Structure and Parameterization) standard. Efficient FMI based simulation is provided for both model exchange and co-simulation. An external API is available for use from other tools and scripting languages such as *Python*.

OMSIMULATOR

OMSimulator is a command line interface for the OMSimulatorLib library. It can be used to simulate Functional Mock-up Units (FMUs) and System Structure and Parameterization (SSP) files, as well as execute dedicated Python simulation scripts using the OMSimulator Python API.

2.1 OMSimulator Flags

A brief description of all command line flags can be displayed using

OMSimulator --help:

```
usage: OMSimulator [-h] [--validate] [--version]
                  [--addParametersToCSV | --no-addParametersToCSV]
                  [--algLoopSolver {fixedpoint,kinsol}] [--clearAllOptions]
                  [--CVODEMaxErrTestFails CVODEMAXERRTESTFAILS]
                  [--CVODEMaxNLSFailures CVODEMAXNLSFAILURES]
                  [--CVODEMaxNLSIterations CVODEMAXNLSITERATIONS]
                  [--CVODEMaxSteps CVODEMAXSTEPS]
                  [--deleteTempFiles | --no-deleteTempFiles]
                  [--directionalDerivatives | --no-directionalDerivatives]
                  [--dumpAlgLoops | --no-dumpAlgLoops]
                  [--emitEvents | --no-emitEvents]
                  [--ignoreInitialUnknowns | --no-ignoreInitialUnknowns]
                  [--initialStepSize INITIALSTEPSize]
                  [--inputExtrapolation | --no-inputExtrapolation]
                  [--intervals INTERVALS] [--logFile LOGFILE]
                  [--logLevel LOGLEVEL] [--master MASTER]
                  [--maxEventIteration MAXEVENTITERATION]
                  [--maxLoopIteration MAXLOOPITERATION]
                  [--minimumStepSize MINIMUMSTEPSize] [--mode {cs,me}]
                  [--numProcs NUMPROCS] [--progressBar | --no-progressBar]
                  [--realTime | --no-realTime] [--resultFile RESULTFILE]
                  [--skipCSVHeader | --no-skipCSVHeader]
                  [--solver {euler,cvode}] [--solverStats | --no-solverStats]
                  [--startTime STARTTIME] [--stepSize STEPSize]
                  [--stopTime STOPTIME] [--stripRoot | --no-stripRoot]
                  [--suppressPath | --no-suppressPath] [--tempDir TMPDIR]
                  [--timeout TIMEOUT] [--tolerance TOLERANCE]
                  [--wallTime | --no-wallTime] [--workingDir WORKINGDIR]
                  [--zeroNominal | --no-zeroNominal]
```

(continues on next page)

(continued from previous page)

model

Command line entry point for OMSimulator. Allows running a model file directly, e.g.: OMSimulator test.ssp OMSimulator model.fmu OMSimulator test.py running a Python driver script without having to write a driver script. All simulation settings (start/stop time, result file, tolerance, ...) are taken from the model itself; use --resultFile/--startTime/--stopTime/--tolerance/--stepSize to override them. For FMUs that export both model exchange and co-simulation, use --mode to pick which one to run. Pass --stripRoot to drop the "model.root" prefix from exported signal names. Files are validated against their schema (FMI for .fmu, SSP for .ssp) before simulation; pass --validate to only validate the file and skip simulation e.g.: OMSimulator --validate test.ssp OMSimulator --validate model.fmu

positional arguments:

model Path to a .ssp or .fmu file to simulate

options:

-h, --help show this help message and exit
 --validate Only validate the file against its schema; do not simulate
 --version show program's version number and exit
 --addParametersToCSV, --no-addParametersToCSV Export parameters to a .csv file (default: false)
 --algLoopSolver {fixedpoint,kinsol} Specifies the loop solver method used for algebraic loops spanning multiple components (default: kinsol)
 --clearAllOptions Reset all flags to their default values
 --CVODEMaxErrTestFails CVODEMAXERRTESTFAILS Maximum number of error test failures for CVODE (default: 100)
 --CVODEMaxNLSFailures CVODEMAXNLSFAILURES Maximum number of nonlinear convergence failures for CVODE (default: 100)
 --CVODEMaxNLSIterations CVODEMAXNLSITERATIONS Maximum number of nonlinear solver iterations for CVODE (default: 5)
 --CVODEMaxSteps CVODEMAXSTEPS Maximum number of steps for CVODE (default: 1000)
 --deleteTempFiles, --no-deleteTempFiles Delete temporary files as soon as they are no longer needed (default: true)
 --directionalDerivatives, --no-directionalDerivatives Use directional derivatives to calculate the Jacobian for algebraic loops (default: true)
 --dumpAlgLoops, --no-dumpAlgLoops Dump information for algebraic loops (default: false)
 --emitEvents, --no-emitEvents Emit events during simulation (default: true)

(continues on next page)

(continued from previous page)

```
--ignoreInitialUnknowns, --no-ignoreInitialUnknowns
    Ignore initial unknowns from the modelDescription.xml
    (default: false)
--initialStepSize INITIALSTEPSize
    Specify the initial step size (default: 1e-06)
--inputExtrapolation, --no-inputExtrapolation
    Enable input extrapolation using derivative
    information (default: false)
--intervals INTERVALS
    Specify the number of communication points (arg > 1)
    (default: 500)
--logFile LOGFILE
    Specify the log file (default: stdout)
--logLevel LOGLEVEL
    Set the log level (0: default, 1: debug, 2:
    debug+trace) (default: 0)
--master MASTER
    Specify the master algorithm (ma) (default: ma)
--maxEventIteration MAXEVENTITERATION
    Specify the maximum number of iterations for handling
    a single event (default: 100)
--maxLoopIteration MAXLOOPITERATION
    Specify the maximum number of iterations for solving
    algebraic loops between system-level components.
    Internal algebraic loops of components are not
    affected. (default: 10)
--minimumStepSize MINIMUMSTEPSize
    Specify the minimum step size (default: 1e-12)
--mode {cs,me}
    Force 'cs' (co-simulation) or 'me' (model exchange)
    for FMUs that export both kinds (.fmu only) (default:
    co-simulation)
--numProcs NUMPROCS
    Specify the maximum number of processors to use
    (0=auto, 1=default) (default: 1)
--progressBar, --no-progressBar
    Show a progress bar for the simulation progress in the
    terminal (default: false)
--realTime, --no-realTime
    Enable experimental feature for (soft) real-time co-
    simulation (default: false)
--resultFile RESULTFILE
    Specify the name of the output result file (default:
    the model name plus '_res.mat')
--skipCSVHeader, --no-skipCSVHeader
    Skip the CSV delimiter row in the header of .csv
    result files (default: true)
--solver {euler,cvode}
    Set the ODE solver for model-exchange FMUs (.fmu,
    mode=me only) (default: cvode)
--solverStats, --no-solverStats
    Add solver stats to the result file, e.g., step size;
    not supported for all solvers (default: false)
--startTime STARTTIME
```

(continues on next page)

(continued from previous page)

```
--stepSize STEPSIZE      Specify the start time (default: from the model)
                          Specify the (maximum) step size (.fmu only) (default:
                          from the model)
--stopTime STOPTIME      Specify the stop time (default: from the model)
--stripRoot, --no-stripRoot
                          Remove the root system prefix from all exported
                          signals (default: false)
--suppressPath, --no-suppressPath
                          Suppress path information in info messages; especially
                          useful for testing (default: true)
--tempDir TMPDIR         Specify the temporary directory (default: the working
                          directory)
--timeout TIMEOUT        Specify the maximum allowed time in seconds for
                          running a simulation (default: 0 (disabled))
--tolerance TOLERANCE    Specify the relative tolerance (default: from the
                          model)
--wallTime, --no-wallTime
                          Add wall time information to the result file (default:
                          false)
--workingDir WORKINGDIR  Specify the working directory (default: the current
                          directory)
--zeroNominal, --no-zeroNominal
                          Accept FMUs with invalid nominal values and replace
                          the invalid nominal values with 1.0 (default: false)
```

To use the `logLevel` flag with option `debug` (`--logLevel=1`) or `debug+trace` (`--logLevel=2`), OMSimulator needs to be built with debug configuration enabled. Refer to the [OMSimulator README on GitHub](#) for further instructions.

2.2 Examples

OMSimulator supports several ways of running simulations.

2.2.1 Running a Python script

A dedicated Python script can be passed directly to OMSimulator. The script can use the OMSimulator Python API to create and configure a simulation.

```
OMSimulator example.py
```

This is useful when the simulation setup is created programmatically or when a simulation needs additional Python logic.

2.2.2 Running an SSP file

An SSP file can be passed directly to OMSimulator. OMSimulator loads the system structure from the SSP file and executes the simulation.

```
OMSimulator example.ssp
```

2.2.3 Running an FMU

An FMU can also be passed directly to OMSimulator.

```
OMSimulator example.fmu
```

This provides a simple way to simulate an FMU without creating an SSP system explicitly.

OMSIMULATORLIB

This library is the core of OMSimulator and provides a C interface that can easily be utilized to handle co-simulation scenarios.

The following is a list of the available C-API functions:

3.1 RunFile

Simulates a single FMU or SSP model.

```
oms_status_enu_t oms_RunFile(const char* filename);
```

3.2 activateVariant

This API provides support to activate a multi-variant modelling from an ssp file [(e.g). SystemStructure.ssd, VarA.ssd, VarB.ssd] from a ssp file. By default when importing a ssp file the default variant will be “SystemStructure.ssd”. The users can be able to switch between other variants by using this API and make changes to that particular variant and simulate them.

```
oms_status_enu_t oms_activateVariant(const char* crefA, const char* crefB);
```

3.3 addBus

Adds a bus to a given component.

```
oms_status_enu_t oms_addBus(const char* cref);
```

3.4 addConnection

Adds a new connection between connectors *A* and *B*. The connectors need to be specified as fully qualified component references, e.g., “*model.system.component.signal*”.

```
oms_status_enu_t oms_addConnection(const char* crefA, const char* crefB, bool_↪  
suppressUnitConversion);
```

The two arguments *crefA* and *crefB* get swapped automatically if necessary. The third argument *suppressUnitConversion* is optional and the default value is *false* which allows automatic unit conversion between connections, if set to *true* then automatic unit conversion will be disabled.

3.5 addConnector

Adds a connector to a given component.

```
oms_status_enu_t oms_addConnector(const char* cref, oms_causality_enu_t  
↪ causality, oms_signal_type_enu_t type);
```

3.6 addConnectorToBus

Adds a connector to a bus.

```
oms_status_enu_t oms_addConnectorToBus(const char* busCref, const char*  
↪ connectorCref);
```

3.7 addResources

Adds an external resources to an existing SSP. The external resources should be a “.ssv” or “.ssm” file

```
oms_status_enu_t oms_addResources(const char* cref_, const char* path)
```

3.8 addSignalsToResults

Add all variables that match the given regex to the result file.

```
oms_status_enu_t oms_addSignalsToResults(const char* cref, const char* regex);
```

The second argument, i.e. regex, is considered as a regular expression (C++11). “.*” and “(.)*” can be used to hit all variables.

3.9 addSubModel

Adds a component to a system.

```
oms_status_enu_t oms_addSubModel(const char* cref, const char* fmuPath);
```

3.10 addSystem

Adds a (sub-)system to a model or system.

```
oms_status_enu_t oms_addSystem(const char* cref, oms_system_enu_t type);
```

3.11 compareSimulationResults

This function compares a given signal of two result files within absolute and relative tolerances.

```
int oms_compareSimulationResults(const char* filenameA, const char* filenameB,  
↪ const char* var, double relTol, double absTol);
```

The following table describes the input values:

Input	Type	Description
filenameA	String	Name of first result file to compare.
filenameB	String	Name of second result file to compare.
var	String	Name of signal to compare.
relTol	Number	Relative tolerance.
absTol	Number	Absolute tolerance.

The following table describes the return values:

Type	Description
Integer	1 if the signal is considered as equal, 0 otherwise

3.12 copySystem

Copies a system.

```
oms_status_enu_t oms_copySystem(const char* source, const char* target);
```

3.13 delete

Deletes a connector, component, system, or model object.

```
oms_status_enu_t oms_delete(const char* cref);
```

3.14 deleteConnection

Deletes the connection between connectors *crefA* and *crefB*.

```
oms_status_enu_t oms_deleteConnection(const char* crefA, const char* crefB);
```

The two arguments *crefA* and *crefB* get swapped automatically if necessary.

3.15 deleteConnectorFromBus

Deletes a connector from a given bus.

```
oms_status_enu_t oms_deleteConnectorFromBus(const char* busCref, const char* ↵
↵connectorCref);
```

3.16 deleteResources

Deletes the reference and resource file in a SSP. Deletion of “.ssv” and “.ssm” files are currently supported. The API can be used in two ways.

- 1) deleting only the reference file in “.ssd”.

- 2) deleting both reference and resource files in “.ssp”.

To delete only the reference file in ssd, the user should provide the full qualified cref of the “.ssv” file associated with a system or subsystem or component (e.g) “model.root:root1.ssv”.

To delete both the reference and resource file in ssp, it is enough to provide only the model cref of the “.ssv” file (e.g) “model:root1.ssv”.

When deleting only the references of a “.ssv” file, if a parameter mapping file “.ssm” is binded to a “.ssv” file then the “.ssm” file will also be deleted. It is not possible to delete the references of “.ssm” separately as the ssm file is binded to a ssv file.

The filename of the reference or resource file is provided by the users using colon suffix at the end of cref. (e.g) “:root.ssv”

```
oms_status_enu_t oms_deleteResources(const char* cref);
```

3.17 doStep

Simulates a macro step of the given composite model. The step size will be determined by the master algorithm and is limited by the defined minimal and maximal step sizes.

```
oms_status_enu_t oms_doStep(const char* cref);
```

3.18 duplicateVariant

This API provides support to develop a multi-variant modelling in OMSimulator [(e.g). SystemStructure.ssd, VarA.ssd, VarB.ssd]. When duplicating a variant, the new variant becomes the current variant and all the changes made by the users are applied to the new variants only, and all the ssv and ssm resources associated with the new variant will be given new name based on the variant name provided by the user. This allows the bundling of multiple variants of a system structure definition referencing a similar set of packaged resources as a single SSP. However there must still be one SSD file named SystemStructure.ssd at the root of the ZIP archive which will be considered as default variant.

```
oms_status_enu_t oms_duplicateVariant(const char* crefA, const char* crefB);
```

3.19 export

Exports a composite model to a SPP file.

```
oms_status_enu_t oms_export(const char* cref, const char* filename);
```

3.20 exportDependencyGraphs

Export the dependency graphs of a given model to dot files.

```
oms_status_enu_t oms_exportDependencyGraphs(const char* cref, const char* _  
↳ initialization, const char* event, const char* simulation);
```

3.21 exportSSMTemplate

Exports all signals that have start values of one or multiple FMUs to a SSM file that are read from modelDescription.xml with a mapping entry. The mapping entry specifies a single mapping between a parameter in the source and a parameter of the system or component being parameterized. The mapping entry contains two attributes namely source and target. The source attribute will be empty and needs to be manually mapped by the users associated with the parameter name defined in the SSV file, the target contains the name of parameter in the system or component to be parameterized. The function can be called for a top level model or a certain FMU component. If called for a top level model, start values of all FMUs are exported to the SSM file. If called for a component, start values of just this FMU are exported to the SSM file.

```
oms_status_enu_t oms_exportSSMTemplate(const char* cref, const char* filename)
```

3.22 exportSSVTemplate

Exports all signals that have start values of one or multiple FMUs to a SSV file that are read from modelDescription.xml. The function can be called for a top level model or a certain FMU component. If called for a top level model, start values of all FMUs are exported to the SSV file. If called for a component, start values of just this FMU are exported to the SSV file.

```
oms_status_enu_t oms_exportSSVTemplate(const char* cref, const char* filename)
```

3.23 exportSnapshot

Lists the SSD representation of a given model, system, or component.

Memory is allocated for *contents*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
oms_status_enu_t oms_exportSnapshot(const char* cref, char** contents);
```

3.24 freeMemory

Free the memory allocated by some other API. Pass the object for which memory is allocated.

```
void oms_freeMemory(void* obj);
```

3.25 getBoolean

Get boolean value of given signal.

```
oms_status_enu_t oms_getBoolean(const char* cref, bool* value);
```

3.26 getBus

Gets the bus object.

```
oms_status_enumer_t oms_getBus(const char* cref, oms_busconnector_t**  
↪ busConnector);
```

3.27 GetComponentType

Gets the type of the given component.

```
oms_status_enumer_t oms_getComponentType(const char* cref, oms_component_enumer_t*  
↪ type);
```

3.28 getConnections

Get list of all connections from a given component.

```
oms_status_enumer_t oms_getConnections(const char* cref, oms_connection_t***  
↪ connections);
```

3.29 getConnector

Gets the connector object of the given connector cref.

```
oms_status_enumer_t oms_getConnector(const char* cref, oms_connector_t**  
↪ connector);
```

3.30 getDirectionalDerivative

This function computes the directional derivatives of an FMU.

```
oms_status_enumer_t oms_getDirectionalDerivative(const char* cref, double*  
↪ value);
```

3.31 getElement

Get element information of a given component reference.

```
oms_status_enumer_t oms_getElement(const char* cref, oms_element_t** element);
```

3.32 getElements

Get list of all sub-components of a given component reference.

```
oms_status_enumer_t oms_getElements(const char* cref, oms_element_t*** elements);
```

3.33 getFMUInfo

Returns FMU specific information.

```
oms_status_enumeration_t oms_getFMUInfo(const char* cref, const oms_fmu_info_t**  
↪ fmuInfo);
```

3.34 getFixedStepSize

Gets the fixed step size. Can be used for the communication step size of co-simulation systems and also for the integrator step size in model exchange systems.

```
oms_status_enumeration_t oms_getFixedStepSize(const char* cref, double* stepSize);
```

3.35 getInteger

Get integer value of given signal.

```
oms_status_enumeration_t oms_getInteger(const char* cref, int* value);
```

3.36 getModelState

Gets the model state of the given model cref.

```
oms_status_enumeration_t oms_getModelState(const char* cref, oms_modelState_enumeration_t*  
↪ modelState);
```

3.37 getReal

Get real value.

```
oms_status_enumeration_t oms_getReal(const char* cref, double* value);
```

3.38 getResultFile

Gets the result filename and buffer size of the given model cref.

```
oms_status_enumeration_t oms_getResultFile(const char* cref, char** filename, int*  
↪ bufferSize);
```

3.39 getSolver

Gets the selected solver method of the given system.

```
oms_status_enumeration_t oms_getSolver(const char* cref, oms_solver_enumeration_t* solver);
```

3.40 getTime

Get the start time from the model.

```
oms_status_enu_t oms_getStartTime(const char* cref, double* startTime);
```

3.41 getStopTime

Get the stop time from the model.

```
oms_status_enu_t oms_getStopTime(const char* cref, double* stopTime);
```

3.42 getString

Get string value.

Memory is allocated for *value*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
oms_status_enu_t oms_getString(const char* cref, char** value);
```

3.43 getSubModelPath

Returns the path of a given component.

```
oms_status_enu_t oms_getSubModelPath(const char* cref, char** path);
```

3.44 getSystemType

Gets the type of the given system.

```
oms_status_enu_t oms_getSystemType(const char* cref, oms_system_enu_t* type);
```

3.45 getTime

Get the current simulation time from the model.

```
oms_status_enu_t oms_getTime(const char* cref, double* time);
```

3.46 getTolerance

Gets the tolerance of a given system or component.

```
oms_status_enu_t oms_getTolerance(const char* cref, double*   
↪relativeTolerance);
```

3.47 getVariableStepSize

Gets the step size parameters.

```
oms_status_enu_t oms_getVariableStepSize(const char* cref, double*  
↪ initialStepSize, double* minimumStepSize, double* maximumStepSize);
```

3.48 getVersion

Returns the library's version string.

```
const char* oms_getVersion();
```

3.49 importFile

Imports a composite model from a SSP file.

```
oms_status_enu_t oms_importFile(const char* filename, char** cref);
```

3.50 importSnapshot

Loads a snapshot to restore a previous model state. The model must be in virgin model state, which means it must not be instantiated.

```
oms_status_enu_t oms_importSnapshot(const char* cref, const char* snapshot,  
↪ char** newCref);
```

3.51 initialize

Initializes a composite model.

```
oms_status_enu_t oms_initialize(const char* cref);
```

3.52 instantiate

Instantiates a given composite model.

```
oms_status_enu_t oms_instantiate(const char* cref);
```

3.53 list

Lists the SSD representation of a given model, system, or component.

Memory is allocated for *contents*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
oms_status_enu_t oms_list(const char* cref, char** contents);
```

3.54 listUnconnectedConnectors

Lists all unconnected connectors of a given system.

Memory is allocated for *contents*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
oms_status_enu_t oms_listUnconnectedConnectors(const char* cref, char**  
↪ contents);
```

3.55 listVariants

This API shows the number of variants available [(e.g). SystemStructure.ssd, VarA.ssd, VarB.ssd] from a ssp file.

```
oms_status_enu_t oms_listVariants(const char* cref);
```

3.56 loadSnapshot

Loads a snapshot to restore a previous model state. The model must be in virgin model state, which means it must not be instantiated.

```
oms_status_enu_t oms_loadSnapshot(const char* cref, const char* snapshot,  
↪ char** newCref);
```

3.57 newModel

Creates a new and yet empty composite model.

```
oms_status_enu_t oms_newModel(const char* cref);
```

3.58 newResources

Adds a new empty resources to the SSP. The resource file is a “.ssv” file where the parameter values set by the users using “oms_setReal()”, “oms_setInteger()” and “oms_setReal()” are writtern to the file. Currently only “.ssv” files can be created.

The filename of the resource file is provided by the users using colon suffix at the end of cref. (e.g) “:root.ssv”

```
oms_status_enu_t oms_newResources(const char* cref)
```

3.59 referenceResources

Switches the references of “.ssv” and “.ssm” in a SSP file. Referencing of “.ssv” and “.ssm” files are currently supported. The API can be used in two ways.

- 1) Referencing only the “.ssv” file.
- 2) Referencing both the “.ssv” along with the “.ssm” file.

This API should be used in combination with “oms_deleteResources”. To switch with a new reference, the old reference must be deleted first using “oms_deleteResources” and then reference with new resources.

When deleting only the references of a “.ssv” file, if a parameter mapping file “.ssm” is binded to a “.ssv” file, then the reference of “.ssm” file will also be deleted. It is not possible to delete the references of “.ssm” separately as the ssm file is binded to a ssv file. Hence it is not possible to switch the reference of “.ssm” file alone. So in order to switch the reference of “.ssm” file, the users need to bind the reference of “.ssm” file along with the “.ssv”.

The filename of the reference or resource file is provided by the users using colon suffix at the end of cref (e.g) “:root.ssv”, and the “.ssm” file is optional and is provided by the user as the second argument to the API.

```
oms_status_enu_t oms_referenceResources(const char* cref, const char* ↵
↵ssmFile);
```

3.60 removeSignalsFromResults

Removes all variables that match the given regex to the result file.

```
oms_status_enu_t oms_removeSignalsFromResults(const char* cref, const char* ↵
↵regex);
```

The second argument, i.e. regex, is considered as a regular expression (C++11). “.*” and “(.*?)” can be used to hit all variables.

3.61 rename

Renames a model, system, or component.

```
oms_status_enu_t oms_rename(const char* cref, const char* newCref);
```

3.62 replaceSubModel

Replaces an existing fmu component, with a new component provided by the user. When replacing the fmu checks are made in all ssp concepts like in ssd, ssv and ssm, so that connections and parameter settings are not lost. It is possible that the namings of inputs and parameters match, but the start values might have been changed, in such cases new start values will be applied in ssd, ssv and ssm. In case if the Types of inputs and outputs and parameters differed, then the variables are updated according to the new changes and the connections will be removed with warning messages to user. In case when replacing a fmu, if the fmu contains parameter mapping associated with the ssv file, then only the ssm file entries are updated and the start values in the ssv files will not be changed.

```
oms_status_enu_t oms_replaceSubModel(const char* cref, const char* fmuPath);
```

3.63 reset

Reset the composite model after a simulation run.

The FMUs go into the same state as after instantiation.

```
oms_status_enu_t oms_reset(const char* cref);
```

3.64 setActivationRatio

Experimental feature for setting the activation ratio of FMUs for experimenting with multi-rate master algorithms.

```
oms_status_enu_t experimental_setActivationRatio(const char* cref, int k);
```

3.65 setBoolean

Sets the value of a given boolean signal.

```
oms_status_enu_t oms_setBoolean(const char* cref, bool value);
```

3.66 setBusGeometry

```
oms_status_enu_t oms_setBusGeometry(const char* bus, const ssd_connector_
↳ geometry_t* geometry);
```

3.67 setCommandLineOption

Sets special flags.

```
oms_status_enu_t oms_setCommandLineOption(const char* cmd);
```

3.68 setConnectionGeometry

```
oms_status_enu_t oms_setConnectionGeometry(const char* crefA, const char*
↳ crefB, const ssd_connection_geometry_t* geometry);
```

3.69 setConnectorGeometry

Set geometry information to a given connector.

```
oms_status_enu_t oms_setConnectorGeometry(const char* cref, const ssd_
↳ connector_geometry_t* geometry);
```

3.70 setElementGeometry

Set geometry information to a given component.

```
oms_status_enu_t oms_setElementGeometry(const char* cref, const ssd_element_
↳ geometry_t* geometry);
```

3.71 setFixedStepSize

Sets the fixed step size. Can be used for the communication step size of co-simulation systems and also for the integrator step size in model exchange systems.

```
oms_status_enumer_t oms_setFixedStepSize(const char* cref, double stepSize);
```

3.72 setInteger

Sets the value of a given integer signal.

```
oms_status_enumer_t oms_setInteger(const char* cref, int value);
```

3.73 setLogFile

Redirects logging output to file or std streams. The warning/error counters are reset.

filename="" to redirect to std streams and proper filename to redirect to file.

```
oms_status_enumer_t oms_setLogFile(const char* filename);
```

3.74 setLoggingCallback

Sets a callback function for the logging system.

```
void oms_setLoggingCallback(void (*cb)(oms_message_type_enumer_t type, const char* message));
```

3.75 setLoggingInterval

Set the logging interval of the simulation.

```
oms_status_enumer_t oms_setLoggingInterval(const char* cref, double loggingInterval);
```

3.76 setLoggingLevel

Enables/Disables debug logging (logDebug and logTrace).

0 default, 1 default+debug, 2 default+debug+trace

```
void oms_setLoggingLevel(int logLevel);
```

3.77 setMaxLogFileSize

Sets maximum log file size in MB. If the file exceeds this limit, the logging will continue on stdout.

```
void oms_setMaxLogFileSize(const unsigned long size);
```

3.78 setReal

Sets the value of a given real signal.

```
oms_status_enu_t oms_setReal(const char* cref, double value);
```

This function can be called in different model states:

- Before instantiation: *setReal* can be used to set start values or to define initial unknowns (e.g. parameters, states). The values are not immediately applied to the simulation unit, since it isn't actually instantiated.
- After instantiation and before initialization: Same as before instantiation, but the values are applied immediately to the simulation unit.
- After initialization: Can be used to force external inputs, which might cause discrete changes of continuous signals.

3.79 setRealInputDerivative

Sets the first order derivative of a real input signal.

This can only be used for CS-FMU real input signals.

```
oms_status_enu_t oms_setRealInputDerivative(const char* cref, double value);
```

3.80 setResultFile

Set the result file of the simulation.

```
oms_status_enu_t oms_setResultFile(const char* cref, const char* filename, ↵
↪int bufferSize);
```

The creation of a result file is omitted if the filename is an empty string.

3.81 setSolver

Sets the solver method for the given system.

```
oms_status_enu_t oms_setSolver(const char* cref, oms_solver_enu_t solver);
```

3.82 setStartTime

Set the start time of the simulation.

```
oms_status_enu_t oms_setStartTime(const char* cref, double startTime);
```

3.83 setStopTime

Set the stop time of the simulation.

```
oms_status_enu_t oms_setStopTime(const char* cref, double stopTime);
```

3.84 setString

Sets the value of a given string signal.

```
oms_status_enu_t oms_setString(const char* cref, const char* value);
```

3.85 setTempDirectory

Set new temp directory.

```
oms_status_enu_t oms_setTempDirectory(const char* newTempDir);
```

3.86 setTolerance

Sets the tolerance for a given model or system.

```
oms_status_enu_t oms_setTolerance(const char* cref, double relativeTolerance);
```

Default values are $1e-4$ for both relative and absolute tolerances.

A tolerance specified for a model is automatically applied to its root system, i.e. both calls do exactly the same:

```
oms_setTolerance("model", relativeTolerance);
oms_setTolerance("model.root", relativeTolerance);
```

Component, e.g. FMUs, pick up the tolerances from there system. That means it is not possible to define different tolerances for FMUs in the same system right now.

In a strongly coupled system (*oms_system_sc*), the relative tolerance is used for CVODE and the absolute tolerance is used to solve algebraic loops.

In a weakly coupled system (*oms_system_wc*), both the relative and absolute tolerances are used for the adaptive step master algorithms and the absolute tolerance is used to solve algebraic loops.

3.87 setUnit

Sets the unit of a given signal.

```
oms_status_enu_t oms_setUnit(const char* cref, const char* value);
```

3.88 setVariableStepSize

Sets the step size parameters for methods with stepsize control.

```
oms_status_enu_t oms_getVariableStepSize(const char* cref, double*
↪initialStepSize, double* minimumStepSize, double* maximumStepSize);
```

3.89 setWorkingDirectory

Set a new working directory.

```
oms_status_enu_t oms_setWorkingDirectory(const char* newWorkingDir);
```

3.90 simulate

Simulates a composite model.

```
oms_status_enu_t oms_simulate(const char* cref);
```

3.91 simulate_realtime

Experimental feature for (soft) real-time simulation.

```
oms_status_enu_t experimental_simulate_realtime(const char* ident);
```

3.92 stepUntil

Simulates a composite model until a given time value.

```
oms_status_enu_t oms_stepUntil(const char* cref, double stopTime);
```

3.93 terminate

Terminates a given composite model.

```
oms_status_enu_t oms_terminate(const char* cref);
```

OMSIMULATORPYTHON3

This section documents the new **oms3 Python API**, designed to improve modularity and reduce overhead, while providing tighter integration with Python. The updated architecture separates the simulation core from the SSP (System Structure and Parameterization) standards, offering greater flexibility, interoperability and efficient model management.

4.1 Core Capabilities

The **oms3 Python API** provides an extensive set of commands for building, configuring, and validating SSP models. The following table summarizes the main functionalities provided by the package.

Category	Functionality
Model Management	• Create new SSP models • Create, activate, and duplicate variants • Add or delete resources, components, systems, connectors • Create nested systems • Set parameters and solvers • Swap SSV references
Connectivity	• Add components and connectors • Define and manage connections
SSP Standards	• Import and export SSP models • Create SSV and SSM descriptions • Export SSV and SSM templates • Validate SSP, SSV, and SSM
FMU Integration	• Import and validate FMUs as components
Utilities	• Error handling, logging, and debugging support

4.2 Command Line Interface

In addition to the Python API, the **oms3 Python** package installs a `OMSimulator` command line entry point that can run a `.ssp` or `.fmu` or `.py` file directly, without having to write a driver script:

```
OMSimulator model.ssp
OMSimulator model.fmu
```

All simulation settings (start/stop time, tolerance, ...) are taken from the model itself; use the flags below to override them. Both file types are validated against their schema (FMI for `.fmu`, SSP for `.ssp`) before simulation.

The full list of command line arguments can also be printed at any time with:

```
OMSimulator --help
```

The table below summarizes the most commonly used flags. For a complete list, refer to *OMSimulator Flags*.

Table 1: OMSimulator command line flags

Flag	Value	Description
<code>--resultFile</code>	FILE	Override the result file name.
<code>--startTime</code>	TIME	Override the simulation start time.
<code>--stopTime</code>	TIME	Override the simulation stop time.
<code>--tolerance</code>	TOL	Override the solver tolerance (.fmu only).
<code>--stepSize</code>	SIZE	Override the (maximum) simulation step size (.fmu only).
<code>--mode</code>	cs me	Force co-simulation or model exchange for FMUs that export both kinds (.fmu only).
<code>--solver</code>	euler cvode	Set the ODE solver for model-exchange FMUs (.fmu, requires <code>--mode me</code>).
<code>--stripRoot</code>	—	Remove the root system prefix from exported signal names (applies to both .fmu and .ssp).
<code>--validate</code>	.fmu .ssp	Only validate the file against its schema; do not simulate.

4.2.1 Examples

```
# Simulate an FMU using its own declared defaults
OMSimulator model.fmu

# Force model-exchange mode with the Euler solver and a custom step size
OMSimulator model.fmu --mode me --solver euler --stepSize 1e-3

# Only validate an FMU or SSP against its schema, without simulating
OMSimulator model.fmu --validate
OMSimulator model.ssp --validate

# Override the stop time and result file of an SSP
OMSimulator model.ssp --stopTime 10 --resultFile out.mat

# Drop the "model.root" prefix from exported signal names (.fmu or .ssp)
OMSimulator model.fmu --stripRoot
OMSimulator model.ssp --stripRoot
```

4.3 Quick start Example

The following example demonstrates how to

- Create a new SSP model
- Add FMU components as resources
- Establish connections between components

- Export the model to an SSP file
- Re-import the exported SSP file
- Instantiate the model, set parameter values, and run a simulation

```
from OMSimulator import SSP, CRef, Settings
Settings.suppressPath = True

# This example creates a new SSP file with an FMU instantiated as a component.
# It then exports the SSP file and re-imports and set run time value on the
↳ instantiated model and the simulates the model.

model = SSP()
model.addResource('../resources/Modelica.Blocks.Math.Add.fmu', new_name=
↳ 'resources/Add.fmu')
model.addResource('../resources/Modelica.Blocks.Math.Gain.fmu', new_name=
↳ 'resources/Gain.fmu')

component1 = model.addComponent(CRef('default', 'Add1'), 'resources/Add.fmu')
component3 = model.addComponent(CRef('default', 'Gain1'), 'resources/Gain.fmu
↳ ')

model.addConnection(CRef('default', 'Gain1', 'y'), CRef('default', 'Add1', 'u1
↳ '))

#model.list()
model.export('SimpleSimulation5.ssp')
## import the exported SSP file
model2 = SSP('SimpleSimulation5.ssp')

instantiated_model = model2.instantiate()
instantiated_model.setResultFile("SimpleSimulation5_res.mat")
instantiated_model.setValue(CRef('default', 'Gain1', 'k'), 2.0)
instantiated_model.setValue(CRef('default', 'Gain1', 'u'), 6.0)

print(f"info: After instantiation:")
print(f"info:    default.Gain1.k: {instantiated_model.getValue(CRef('default',
↳ 'Gain1', 'k'))}", flush=True)
print(f"info:    default.Gain1.u: {instantiated_model.getValue(CRef('default',
↳ 'Gain1', 'u'))}", flush=True)
print(f"info:    default.Gain1.y: {instantiated_model.getValue(CRef('default',
↳ 'Gain1', 'y'))}", flush=True)
print(f"info:    default.Add1.u1: {instantiated_model.getValue(CRef('default',
↳ 'Add1', 'u1'))}", flush=True)

instantiated_model.initialize()
instantiated_model.simulate()
print(f"info: After simulation:")
print(f"info:    default.Gain1.k: {instantiated_model.getValue(CRef('default',
↳ 'Gain1', 'k'))}", flush=True)
```

(continues on next page)

(continued from previous page)

```
print(f"info:    default.Gain1.u: {instantiated_model.getValue(CRef('default',
↪ 'Gain1', 'u'))}", flush=True)
print(f"info:    default.Gain1.y: {instantiated_model.getValue(CRef('default',
↪ 'Gain1', 'y'))}", flush=True)
print(f"info:    default.Add1.u1: {instantiated_model.getValue(CRef('default',
↪ 'Add1', 'u1'))}", flush=True)

instantiated_model.terminate()
instantiated_model.delete()
```

4.4 SSP

The `ssp` module provides a high-level interface for creating, importing, and managing SSP (System Structure and Parameterization) models. SSP models are used to describe complex system architectures by connecting multiple components and defining their parameters, enabling seamless simulation and co-simulation workflows.

```
from OMSimulator import SSP, Settings
Settings.suppressPath = True

# Create a new, empty SSP model instance
model = SSP()

# Create a new, empty SSP model with a custom model name and root system name
model = SSP(model_name="model", system_name="root")

# Load an existing SSP model from a file
model = SSP("PIController.ssp")

# list the ssp components
model.list()
```

`model_name` and `system_name` are optional. `model_name` defaults to "default" and `system_name` defaults to `model_name`. The model name is the first element of every component reference, so a model created with `model_name="model"` is addressed as `CRef('model', 'Add1')`. Both arguments are ignored when an existing SSP file is loaded, because the names are then read from the file.

Once created or loaded, an SSP model instance allows you to:

- Add and configure components.
- Connect signals between components.
- Define parameters and experiment settings.
- Export the model for simulation or further analysis.

4.5 SSD

The **SSV** class provides a high-level interface for importing, and managing SSD (System Structure Description) models which are unpacked from SSP files. We can use the SSD class to load an SSD file, inspect its structure, and manipulate its elements programmatically.

```
from OMSimulator import SSD, Settings
Settings.suppressPath = True
ssd = SSD.importFromFile('../resources/LOC/SystemStructure.ssd')
ssd.list()
```

4.6 SSV

The **SSV** class provides an interface for creating and managing *System Structure Parameter Values* (SSV) files. These files are used to define parameter values, units, and variability information for components within an SSP model.

Typical use cases include:

- Defining scalar variables and their default values
- Assigning units and data types to parameters
- Storing both numeric and string-based parameter values
- Exporting parameter sets to *.ssv* files for reuse across simulations

Key Methods

- `setValue(name, value, unit=None)` Assigns a parameter value. Optionally, a unit can be specified.
- `export(filename)` Writes the defined parameters to an external *.ssv* file.

Example

```
from OMSimulator import SSV, Settings

Settings.suppressPath = True

# Create a new SSV file and define parameters
ssv1 = SSV()
ssv1.setValue("k1", 2.0, "m")
ssv1.setValue("k2", 3.0)
ssv1.setValue("k3", 3)
ssv1.setValue("k4", False)
ssv1.setValue("param3", "hello")

# Export to file
ssv1.export("myfile1.ssv")
```

4.7 SSM

The **SSM** class provides an interface for creating and managing *System Structure and Mapping* (SSM) files. These files define how parameters declared in an SSV file are mapped to component parameters within an SSP model. This enables separation of parameter definitions from their usage, improving reusability and modularity.

Typical use cases include:

- Linking global parameter definitions (from SSV files) to component parameters in an SSP model.
- Supporting multiple parameter mappings for the same input across different components.
- Creating flexible configurations by swapping or updating parameter mappings without modifying the model structure.

Key Methods

- `mapParameter(source, target)` Creates a mapping between a parameter defined in an SSV file (`source`) and a component parameter or input signal (`target`).
- `export(filename)` Writes the defined parameter mappings to an external `.ssm` file.

Example

```
from OMSimulator import SSV, SSM, Settings

Settings.suppressPath = True

# Define parameters in SSV
ssv1 = SSV()
ssv1.setValue("Input1", 2.0)
ssv1.setValue("param1", 3.0)
ssv1.setValue("param2", 3)
ssv1.export("mapping1.ssv")

# Create parameter mappings in SSM
ssm1 = SSM()
ssm1.mapParameter("Input1", "Add1.u1")
ssm1.mapParameter("Input1", "u2")
ssm1.mapParameter("Input1", "u3")
ssm1.mapParameter("param1", "k1")
ssm1.mapParameter("param2", "k2")
ssm1.export("mapping2.ssm")
```

4.8 FMU

The **FMU** class provides an interface for loading and inspecting *Functional Mock-up Units* (FMUs). This class allows users to access key metadata, states, and variables of an FMU directly from Python, enabling integration, validation and simulation.

Typical use cases include:

- Loading an FMU into memory for inspection or simulation.

- Accessing FMU metadata such as name, GUID, and FMI version.
- Iterating through state variables and retrieving their properties.
- Using variable metadata (signal type, causality, variability, value references) for analysis or parameter mapping.
- set default experiment settings such as start time, stop time, step size, and tolerance.
- set parameters and inputs before simulation.

Syntax

```
fmu = FMU(fmuPath: str, instanceName: str = "")
```

Arguments

- *fmuPath*: Path to the FMU file.
- *instanceName*: Optional name for the FMU instance. If not provided, it defaults to the model name of the FMU. The instance name is used in the result file as prefix for the variable names.

Example

```
from OMSimulator import FMU

# Load FMU
fmu = FMU('../resources/Modelica.Electrical.Analog.Examples.
↳CauerLowPassAnalog.fmu')

# Print metadata
print("name:", fmu.modelName)
print("guid:", fmu.guid)
print("fmi version:", fmu.fmiVersion)

# Inspect state variables
print("States:")
for var in sorted(fmu.states, key=lambda x: x.name):
    print({
        'name': var.name,
        'signal_type': var.signal_type.name,
        'valueReference': var.valueReference,
        'variability': var.variability,
        'causality': var.causality.name
    })

## instantiate the FMU
fmu.instantiate()

## set result file
fmu.setResultFile('CauerLowPass_res.mat')

## set parameter and input values
fmu.setValue('R1', 1000.0)
fmu.setValue('C1.u', 1e-6)

## initialize the FMU
fmu.initialize()

## simulate the FMU
fmu.simulate()
```

(continues on next page)

(continued from previous page)

```
fmu.terminate()
fmu.delete()
```

4.9 Component

The **Component** class provides a flexible interface for defining modular building blocks in an SSP model. It supports **architectural modeling**, allowing components to represent actual FMUs or “dummy fmus” with expected connectors and parameter values. This makes it possible to design and validate system architectures before all FMUs are available.

Key Features:

- **Flexible instantiation** — create a component from an FMU or as a dummy placeholder.
- **Connector support** — define inputs, outputs, and parameters with explicit causality and signal type.
- **Parameter management** — assign values to parameters or inputs directly within the component.
- **SSV integration** — link components to SSV files to define and manage parameter sets externally.
- **System integration** — add the component to SSP models for hierarchical or flat system architectures.

```
from OMSimulator import CRef, Connector, Causality, SignalType, SSV
from OMSimulator.component import Component

# Create a dummy component
component = Component(CRef('add'), 'dummy.fmu')

# Add connectors
component.addConnector(Connector('input', Causality.input, SignalType.Real))
component.addConnector(Connector('output', Causality.output, SignalType.Real))
component.addConnector(Connector('parameter', Causality.parameter, SignalType.
    ↳Real))

# Set parameter values
component.setValue('parameter', 1.0)
component.setValue('input', 2.0)

# Create an SSV file for parameter initialization
ssv = SSV()
ssv.setValue('parameter', 1.0)
ssv.setValue('input', 2.0)
ssv.export('dummy.ssv')

# Link SSV file to component
component.addSSVReference('dummy.ssv')
```

(continues on next page)

(continued from previous page)

```
# List component details
component.list()
```

4.10 addSystem

The **addSystem** method creates a new system within an SSP model. Systems serve as containers for components and connectors, and can be organized hierarchically to support **nested-system architectures**. This enables modular, scalable, and reusable system designs.

Key Features:

- **System instantiation** — create new systems within the SSP model.
- **Hierarchical modeling** — use structured CRef paths to create nested systems (e.g., a sub-system inside another subsystem).
- **Organization** — group related components, connectors, and resources within a defined sub-system for clarity and reusability.
- **Scalability** — build complex system hierarchies with multiple levels.

Syntax

```
addSystem(cref)
```

Parameters:

- **cref** (*CRef*): The system reference, specifying the hierarchical path and system name within the SSP model.

Example

```
from OMSimulator import SSP, CRef

model = SSP()

# Create systems
model.addSystem(CRef('default', 'sub-system'))
model.addSystem(CRef('default', 'sub-system2'))

# Create a nested subsystem inside 'sub-system2'
model.addSystem(CRef('default', 'sub-system2', 'sub-sub-system'))
```

4.11 addConnector

The **addConnector** API is used to define external interfaces for systems and components within an SSP model. A connector specifies the signal type and causality (e.g., input, output, or parameter) of a variable, allowing subsystems and components to communicate through well-defined ports.

Connectors can be added at different hierarchy levels:

- **Top-level system connectors**, which expose inputs, outputs, and parameters of the entire SSP model.

- **Subsystem connectors**, which define interaction points within nested systems and enable modular system design.

Syntax:

```
addConnector(Connector(name, causality, signal_type))
```

Parameters:

- `name` (*str*): Name of the connector.
- `causality` (*Causality*): Role of the connector (input, output, or parameter).
- `signal_type` (*SignalType*): Type of signal (e.g., Real, Integer, ``Boolean``, String).

Example:

```
from OMSimulator import SSP, CRef, Connector, Causality, SignalType

model = SSP()
# Add a top-level system connector
model.activeVariant.system.addConnector(Connector('input1', Causality.input,
↪SignalType.Real))
# Add a connector to a subsystem
model.addSystem(CRef('default', 'sub-system'))
model.activeVariant.system.elements[CRef('sub-system')].
↪addConnector(Connector('output', Causality.output, SignalType.Real))
```

4.12 addConnection

The **addConnection** API is used to link the output of one component to the input of another within an SSP model. Connections define the signal flow between components, ensuring that data is transmitted correctly during simulation.

Connections are defined using source and target references:

- **Source:** Specifies the output variable of a component.
- **Target:** Specifies the input variable of another component.

Syntax:

```
addConnection(source: CRef, target: CRef)
```

Parameters:

- `source` (*CRef*): Reference to the output variable of the source component.
- `target` (*CRef*): Reference to the input variable of the target component.

Example:

```
from OMSimulator import SSP, CRef, Settings

model = SSP()
model.addResource('../resources/Modelica.Blocks.Math.Add.fmu', new_name=
↪'resources/Add.fmu')
```

(continues on next page)

(continued from previous page)

```
model.addResource('../resources/Modelica.Blocks.Math.Gain.fmu', new_name=
↳ 'resources/Gain.fmu')

model.addComponent(CRef('default', 'Add1'), 'resources/Add.fmu')
model.addComponent(CRef('default', 'Gain1'), 'resources/Gain.fmu')
## add a connection from Gain1.y to Add1.u1
model.addConnection(CRef('default', 'Gain1', 'y'), CRef('default', 'Add1', 'u1
↳ '))
## list the components in the model and connections
model.list()
```

4.13 addResource

The **addResource** method allows you to add external files, such as FMUs, SSV, SSM, SRMD etc.. to an SSP model's resource directory. This ensures that all required model files are bundled with the SSP package and can be referenced consistently during simulation.

Key Features:

- **File management** — copy or rename external FMUs and resources into the SSP project structure only once and use the instances multiple times in the model.
- **Integration with components** — added resources can then be used to instantiate components within the SSP model.

Syntax

```
addResource(source_path, new_name=None)
```

Parameters:

- **source_path** (*str*): Path to the external resource file (e.g., FMU)
- **new_name** (*str, optional*): New name for the resource within the SSP project structure. If not provided, the original filename is used.

Example

```
from OMSimulator import SSP, CRef, Settings

# Add an FMU to the SSP model's resources folder
model.addResource('../resources/Modelica.Blocks.Math.Add.fmu', new_name=
↳ 'resources/Add.fmu')
model.addResource('../resources/Modelica.Blocks.Math.Gain.fmu', new_name=
↳ 'resources/Gain.fmu')
```

4.14 addComponent

The **addComponent** method allows you to instantiate a component within an SSP model. Components can be FMUs or dummy placeholders, enabling flexible system architecture design and simulation.

Key Features:

- **Component instantiation** — create components based on FMU resources or dummy placeholders.
- **Hierarchical modeling** — components can be placed in nested systems using a structured *CRef* path.
- **Flexible integration** — components can later be connected via connectors, linked to SSV files, or assigned custom solver settings.
- **Supports multiple components** — multiple instances of the same FMU can be added with unique identifiers.

Syntax

```
addComponent(cref, resource_path)
```

Parameters:

- **cref** (*CRef*): The component reference, specifying the hierarchical path and component name within the SSP model.
- **resource_path** (*str*): Path to the FMU or model resource that the component should instantiate.

Returns:

- The newly created **Component** object.

Example

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True

model = SSP()

# Add resources to the SSP model
model.addResource('../resources/Modelica.Blocks.Math.Add.fmu', new_name=
↳ 'resources/Add.fmu')
model.addResource('../resources/Modelica.Blocks.Math.Gain.fmu', new_name=
↳ 'resources/Gain.fmu')

# Instantiate components
component1 = model.addComponent(CRef('default', 'Add1'), 'resources/Add.fmu')
# instantiate multiple instances of the same FMU
component2 = model.addComponent(CRef('default', 'Add2'), 'resources/Add.fmu')
component3 = model.addComponent(CRef('default', 'Gain1'), 'resources/Gain.fmu
↳ ')
component4 = model.addComponent(CRef('default', 'Gain2'), 'resources/Gain.fmu
↳ ')
# list the components in the model
model.list()
```

4.15 addSSVReference

The **addSSVReference** API is used to associate an SSV (System Structure Parameterer and values) file with a specific component in an SSP model. This allows parameter values defined in the SSV file to be applied directly to the component, enabling flexible configuration and reuse of parameter sets.

The API also supports an optional third argument for specifying a **parameter mapping file** (.ssm), which defines how the variables in the SSV file map to the parameters of the component.

Syntax:

```
addSSVReference(cref, ssv_path: str, ssm_path: str = None)
```

Parameters:

- **cref** (*CRef*): Reference to a system, sub-system or component to which the SSV file should be applied.
- **ssv_path** (*str*): Path to the SSV file relative to the SSP model resources.
- **ssm_path** (*str, optional*): Path to a parameter mapping file (.ssm) that defines how SSV variables correspond to component parameters.

Example:

```
from OMSimulator import SSP, CRef, Settings, SSV, SSM, Connector, Causality, ↵
↵SignalType

Settings.suppressPath = True

# This example creates a new SSP file with an FMU instantiated as a component. ↵
↵and
# set parameter values to ssv file and map some parameters to ssm file

model = SSP()
model.addResource('../resources/Modelica.Blocks.Math.Add.fmu', new_name=
↵'resources/Add.fmu')
component1 = model.addComponent(CRef('default', 'Add1'), 'resources/Add.fmu')
component2 = model.addComponent(CRef('default', 'Add2'), 'resources/Add.fmu')

## create a ssv file
ssv1 = SSV()
ssv1.setValue("connector_param", 2.0)
ssv1.setValue("connector_input", 3.0)
ssv1.export("myfile3.ssv")

## create a ssm file
ssm = SSM()
ssm.mapParameter("connector_param", "u1")
ssm.mapParameter("connector_input", "u2")
ssm.export("myfile4.ssm")

## Add SSV resource to the model
```

(continues on next page)

(continued from previous page)

```
model.addResource("myfile3.ssv", "resources/myfile3.ssv")
## Add ssm resources to the model
model.addResource("myfile3.ssv", "resources/myfile4.ssm")

# Reference the SSV to a specific component
model.addSSVReference(CRef('default', 'Add1'), 'resources/myfile3.ssv')

# With an optional SSM mapping file
model.addSSVReference(CRef('default', 'Add2'), 'resources/myfile4.ssv',
↪ 'resources/myfile4.ssm')
```

4.16 removeSSVReference

Removes the reference to a given SSV (System Structure parameter and values) file from a specific component or subsystem within the SSP model. This method is useful when a component or subsystem should no longer be associated with a parameter set defined in an SSV file.

Notes:

- The SSV file itself remains in the resources/ folder of the SSP.
- Only the *reference* from the specified component is removed.
- If a parameter mapping file (.ssm) is associated with the SSV, it is also removed.
- If the component had multiple SSV references, only the matching one is cleared.

Syntax:

```
removeSSVReference(cref, ssv_path)
```

Parameters:

- `cref` (*CRef*): Reference to a system, sub-system or component to which the SSV file should be applied.
- `ssv_path` (*str*): Path to the SSV file relative to the SSP model resources.

Example usage:

```
# Load an existing SSP model
model = SSP("swapSSV1.ssp")

# Remove SSV reference from top-level component Add1
model.removeSSVReference(CRef("default", "Add1"), "resources/swap1.
↪ ssv")

# Remove SSV reference from Add3 inside the subsystem
model.removeSSVReference(CRef("default", "sub-system", "Add3"),
↪ "resources/swap1.ssv")
# List the model details to verify removal
model.list()
```

4.17 swapSSVReference

Swaps the reference to an existing SSV (System Structure parameter and values) file with a new one for a specific component or subsystem within the SSP model.

This method is essentially a combination of `removeSSVReference` and `addSSVReference`. It ensures that the old SSV reference is removed and replaced with the new SSV reference in a single step, reducing the risk of leaving the system without any SSV mapping during the transition.

Notes:

- The old SSV file remains in the `resources/` folder of the SSP model.
- Only the *reference* is swapped — no SSV files are deleted.
- If a parameter mapping file (.ssm) is associated with the old SSV, it will also be swapped accordingly.
- This method is equivalent to calling:

```
model.removeSSVReference(cref, old_ssv)
model.addSSVReference(cref, new_ssv)
```

Syntax:

```
swapSSVReference(cref, old_ssv, new_ssv)
```

Parameters:

- `cref` (*CRef*): Reference to a system, sub-system, or component where the SSV file should be swapped.
- `old_ssv` (*str*): Path to the currently referenced SSV file relative to the SSP model resources.
- `new_ssv` (*str*): Path to the new SSV file relative to the SSP model resources.

Example usage:

```
# Load an existing SSP model
model = SSP("swapSSV3.ssp")

# Swap SSV reference from default system (swap5.ssv → swap6.ssv)
model.swapSSVReference(CRef("default"), "resources/swap5.ssv",
↪ "resources/swap6.ssv")

# Swap SSV reference from sub-system (swap6.ssv → swap5.ssv)
model.swapSSVReference(CRef("default", "sub-system"), "resources/
↪ swap6.ssv", "resources/swap5.ssv")

print("After swapping swap6.ssv to default and swap5.ssv to sub-
↪ system")
print(
↪ "=====")
model.list()
```

4.18 listSSVReference

Retrieves the list of SSV (System Structure parameter and values) files currently referenced by a system, sub-system, or component within the SSP model.

Notes:

- Only the references are listed — the method does not open or parse the SSV files.
- If no SSV file is associated with the given scope, an empty list is returned.
- The returned paths are relative to the `resources/` directory of the SSP archive.

Syntax:

```
listSSVReference(cref) -> list[dict]
```

Parameters:

- `cref (CRef)`: Reference to a system, sub-system, or component for which SSV references should be listed.

Returns:

- (`list[dict]`): A list of dict referenced by the given system, sub-system, or component.
 - Each dict contains:
 - * `ssv (str)`: Path to the referenced SSV file relative to the SSP model resources.
 - * `ssm (str | None)`: Path to the associated SSM file if one exists; otherwise, `None`.

Example usage:

```
model = SSP("listSSVReference1.ssp")
model.list()

print("List of SSV references")
print("=====")
print(f"Top level system SSV resources      : {model.listSSVReference(CRef(
    ↪ 'default'))}")
print(f"Top level sub-system SSV resources: {model.listSSVReference(CRef(
    ↪ 'default', 'sub-system'))}")
print(f"Component Add1 SSV resources        : {model.listSSVReference(CRef(
    ↪ 'default', 'Add1'))}")
print(f"Component Add2 SSV resources        : {model.listSSVReference(CRef(
    ↪ 'default', 'Add2'))}")
print(f"Component Add3 SSV resources        : {model.listSSVReference(CRef(
    ↪ 'default', 'sub-system', 'Add3'))}")
```

4.19 exportSSVTemplate

Exports a template SSV (System Structure Parameter Values) file for a given system, sub-system, or component within the SSP model.

The generated SSV template contains all parameters that can be configured for the referenced scope (system or component), but without assigned values. This is useful for creating new parameter sets, preparing configuration files, or documenting the available tunable parameters in a model.

Notes:

- The exported SSV file contains parameter definitions but not their actual values.
- **The scope of the export depends on the provided *CRef***
 - Root system → exports all parameters for the entire SSP.
 - Sub-system → exports parameters for that sub-system and its components.
 - Component → exports only the parameters of the selected component.
- The output file is always written to the specified path; existing files will be overwritten.

Syntax:

```
exportSSVTemplate(cref, ssv_path)
```

Parameters:

- *cref* (*CRef*): Reference to a system, sub-system, or component for which the SSV template should be generated.
- *ssv_path* (*str*): Path where the exported SSV template will be saved.

Example usage:

```
from OMSimulator import SSP, CRef

# Load an existing SSP model
model = SSP("exportSSVTemplate1.ssp")

# Export template for the entire SSP
model.exportSSVTemplate(CRef("default"), "exportSSVTemplate1.ssv")

# Export template for a sub-system
model.exportSSVTemplate(CRef("default", "sub-system"),
↳ "exportSSVTemplate2.ssv")

# Export template for a single component (Add1)
model.exportSSVTemplate(CRef("default", "Add1"), "exportSSVTemplate3.
↳ ssv")

# Export template for another component (Add2)
model.exportSSVTemplate(CRef("default", "Add2"), "exportSSVTemplate4.
↳ ssv")
```

Example usage with fmu component:

```
from OMSimulator import FMU
fmu = FMU('../resources/Modelica.Blocks.Math.Add.fmu')
## export the start values in fmu to ssv template
fmu.exportSSVTemplate('startValuesSSVTemplate.ssv')
```

4.20 exportSSVTemplate

Exports a template SSM (System Structure Parameter mapping) file for a given system, sub-system, or component within the SSP model.

The generated SSM template contains all parameters that can be configured for the referenced scope (system or component), with source and target variables. The source attribute in ssm template will be empty and should be set by user to map to the desired variable in the SSV file. This is useful for creating new parameter mapping files, preparing configuration files.

Notes:

- The exported SSM file contains source and target definitions but source attribute will be empty.
- **The scope of the export depends on the provided *CRef***
 - Root system → exports all parameters for the entire SSP.
 - Sub-system → exports parameters for that sub-system and its components.
 - Component → exports only the parameters of the selected component.
- The output file is always written to the specified path; existing files will be overwritten.

Syntax:

```
exportSSMTemplate(cref, ssm_path)
```

Parameters:

- *cref* (*CRef*): Reference to a system, sub-system, or component for which the SSV template should be generated.
- *ssm_path* (*str*): Path where the exported SSV template will be saved.

Example usage:

```
from OMSimulator import SSP, CRef

# Load an existing SSP model
model = SSP("exportSSMTemplate1.ssp")

# Export template for the entire SSP
model.exportSSMTemplate(CRef("default"), "exportSSMTemplate1.ssm")

# Export template for a sub-system
model.exportSSMTemplate(CRef("default", "sub-system"),
    ↪ "exportSSMTemplate2.ssm")

# Export template for a single component (Add1)
model.exportSSMTemplate(CRef("default", "Add1"), "exportSSMTemplate3.
    ↪ ssm")

# Export template for another component (Add2)
```

(continues on next page)

(continued from previous page)

```
model.exportSSMTemplate(CRef("default", "Add2"), "exportSSMTemplate4.  
→ssm")
```

Example usage with fmu component:

```
from OMSimulator import FMU  
fmu = FMU('../resources/Modelica.Blocks.Math.Add.fmu')  
## export the start values in fmu to ssv template  
fmu.exportSSMTemplate('startValuesSSVTemplate.ssm')
```

4.21 setValue

Assigns a numerical value to a parameter of a component within the SSP model.

This method is used to directly set start values or parameter values of model variables defined in an FMU component. The values are stored in the SSP model and will be applied when the model is simulated or exported.

Notes:

- The parameter must exist in the referenced FMU/component or a top level system connectors; otherwise, an error will be raised.
- Values are stored at the SSP level and exported together with the model structure.
- Supported data types are numerical (float, int, str, bool).
- Special FMI-3.0 data types like Float64, Float32, Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64 are handled via special classes in OMSimulator like Float64, Int32, etc. These classes can be imported from OMSimulator and used to set parameter values with specific fmi3 data types.
- Parameters can be set before or after instantiation.

Syntax:

```
setValue(cref, value)
```

Parameters:

- cref (*CRef*): Reference to the parameter variable within a component. The CRef must specify the system, component, and parameter name (e.g., CRef("default", "Add1", "k1")).
- value (*float | int | str | bool | Float64 | Float32 | Int8 | Int16 | Int32 | Int64 | UInt8 | UInt16 | UInt32 | UInt64*): The value to assign to the parameter.

Example usage:

```
from OMSimulator import SSP, CRef, Settings  
  
Settings.suppressPath = True  
  
# Create a new SSP model  
model = SSP()
```

(continues on next page)

(continued from previous page)

```
# Add FMU resource
model.addResource("../resources/Modelica.Blocks.Math.Add.fmu", new_name="Add.
↪fmu")

# Add two components
model.addComponent(CRef("default", "Add1"), "Add.fmu")
model.addComponent(CRef("default", "Add2"), "Add.fmu")

# Set parameter values for Add1
model.setValue(CRef("default", "Add1", "k1"), 2.0)
model.setValue(CRef("default", "Add1", "k2"), 3.0)

# Set parameter values for Add2
model.setValue(CRef("default", "Add2", "k1"), 100.0)
model.setValue(CRef("default", "Add2", "k2"), 300.0)

# Display model details
model.list()

# Export the SSP with parameter values applied
model.export("setValue2.ssp")
```

Example of setting FMI-3.0 specific data types:

```
from OMSimulator import SSP, CRef, Settings, Float64, Int32, UInt16, Float32,
↪Int8, Int16, Int64, UInt8, UInt32, UInt64

model = SSP()
model.addResource("../resources/Feedthrough3.fmu", new_name="resources/
↪Feedthrough3.fmu")
model.addComponent(CRef("default", "Feedthrough1"), "resources/Feedthrough3.
↪fmu")

model.setValue(CRef("default", "Feedthrough1", "Float64_continuous_input"),
↪Float64(3.5))
model.setValue(CRef("default", "Feedthrough1", "Int64_input"), Int64(7))
model.setValue(CRef("default", "Feedthrough1", "Float32_continuous_input"),
↪Float32(1.5))
model.setValue(CRef("default", "Feedthrough1", "Int8_input"), Int8(3))
model.setValue(CRef("default", "Feedthrough1", "Boolean_input"), True)

model.setValue(CRef("default", "Feedthrough1", "Float32_discrete_input"),
↪Float32(2.5))
model.setValue(CRef("default", "Feedthrough1", "Enumeration_input"), 2)

model.setValue(CRef("default", "Feedthrough1", "Int16_input"), Int16(1000))
model.setValue(CRef("default", "Feedthrough1", "UInt16_input"), UInt16(2000))
model.setValue(CRef("default", "Feedthrough1", "UInt32_input"), UInt32(3000))
model.setValue(CRef("default", "Feedthrough1", "UInt64_input"), UInt64(4000))
```

4.22 getValue

Retrieves the current value of a parameter or variable from a system, sub-system, or component within the SSP model.

This method is typically used to query parameter values that have been set using `setValue`, or to inspect default values provided by FMUs or the SSP model. It can be applied both before and after simulation, depending on whether the SSP has been instantiated and executed.

Notes:

- If called before instantiation, it will return the value stored in the SSP model (either default or set explicitly by the user).
- If called after instantiation, it return the updated value depending on the FMU state.
- Works for system-level, sub-system, and component parameters.
- Returns a Python type corresponding to the variable type (float for Real, int for Integer, bool for Boolean, etc.).

Syntax:

```
getValue(cref) -> value
```

Parameters:

- `cref (CRef)`: Reference to the parameter or variable whose value should be retrieved. For example, `CRef("default", "Add1", "k1")` refers to parameter `k1` of component `Add1`.

Returns:

- *(float | int | bool | str)*: The current value of the requested parameter or variable.

Example usage:

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True

# Load an existing SSP model
model = SSP("setValue2.ssp")

# Retrieve values from top-level system parameter
print(f"info:    default.param1 : {model.getValue(CRef('default', 'param1'))}"
      "\n", flush=True)

# Retrieve values from FMU components
print(f"info:    Add1.k1 : {model.getValue(CRef('default', 'Add1', 'k1'))}",
      "\n", flush=True)
print(f"info:    Add2.k2 : {model.getValue(CRef('default', 'Add2', 'k2'))}",
      "\n", flush=True)
```

4.23 mapParameter

Creates a parameter mapping between a connector and one or more parameters within the SSP model.

Notes:

- Parameter mapping enables hierarchical parameterization of systems and sub-systems.
- A single connector can be mapped to multiple parameters (fan-out mapping).
- Mappings can be stored inline or in the associated SSM (System Structure Mapping) file.

Syntax:

```
mapParameter(cref, connector, parameter)
```

Parameters:

- `cref` (*CRef*): Reference to the system, sub-system, or component where the mapping is applied. Typically, this is the root system (e.g., `CRef("default")`).
- `connector` (*str*): Name of the connector providing the source value.
- `parameter` (*str*): Name of the target parameter or input to be mapped to the connector.

Example usage:

```
from OMSimulator import SSP, CRef, Connector, Causality, SignalType

# Create a new SSP model
model = SSP()

# Add top-level system connectors
model.activeVariant.system.addConnector(Connector("param1", Causality.
    ↳parameter, SignalType.Real))
model.activeVariant.system.addConnector(Connector("param2", Causality.
    ↳parameter, SignalType.Real))
model.activeVariant.system.addConnector(Connector("param3", Causality.
    ↳parameter, SignalType.Real))
model.activeVariant.system.addConnector(Connector("input1", Causality.input,
    ↳SignalType.Real))
model.activeVariant.system.addConnector(Connector("input2", Causality.input,
    ↳SignalType.Real))

# Assign values to virtual connectors
model.setValue(CRef("default", "connector_param"), 2.0)
model.setValue(CRef("default", "connector_input"), 3.0)

# Map top-level parameters
model.mapParameter(CRef("default"), "connector_param", "param1")
model.mapParameter(CRef("default"), "connector_param", "param2")
model.mapParameter(CRef("default"), "connector_param", "param3")

# Map inputs
```

(continues on next page)

(continued from previous page)

```
model.mapParameter(CRef("default"), "connector_input", "input1")
model.mapParameter(CRef("default"), "connector_input", "input2")
```

4.24 duplicate and activate Variant

Creates a copy of an existing variant (SSD file) within an SSP model and assigns it a new name.

A variant in SSP represents a specific configuration of a system (stored as an SSD file). By duplicating a variant, users can create an alternative configuration without modifying the original one. This is particularly useful for scenario testing, parameter studies, or creating multiple versions of a system with different inputs, parameters, or components.

Notes:

- The duplicate is a *deep copy* of the variant — all systems, components, and parameter values are copied.
- The duplicate can be modified independently of the original (e.g., by adding components, changing parameter values).
- Duplicated variants must be explicitly added back to the model using `model.add(variant)`.
- The active variant of a model can be switched using `model.activeVariantName = "VariantName"`.

Syntax:

```
variant_copy = model.variants['Variant-Name'].duplicate(new_name)
```

Parameters:

- `new_name` (*str*): Name for the duplicated variant.

Returns:

- (*SSD*): A new SSD object representing the duplicated variant.

Example usage:

```
from OMSimulator import SSD, SSP, Settings, CRef

Settings.suppressPath = True

# Create a new SSP model with a default variant
model = SSP(temp_dir="./tmp-duplicateVariant2/")
model.addResource("../resources/Modelica.Blocks.Math.Add.fmu")
model.addComponent(CRef("default", "Add1"), "resources/Modelica.Blocks.Math.
↪Add.fmu")

# Set parameter values in default variant
model.setValue(CRef("default", "Add1", "u1"), 10.0)
model.setValue(CRef("default", "Add1", "u2"), 20.0)
model.setValue(CRef("default", "Add1", "k1"), 30.0)
```

(continues on next page)

(continued from previous page)

```
# Duplicate the default variant as "Variant-B"
variantB = model.variants["default"].duplicate("Variant-B")
model.add(variantB)

# Modify parameter values in Variant-B
variantB.setValue(CRef("default", "Add1", "u1"), 100.0)
variantB.setValue(CRef("default", "Add1", "u2"), 200.0)

# Export SSP with both variants
model.export("duplicateVariant2.ssp")

# Load the model again and duplicate Variant-B as Variant-C
model2 = SSP("duplicateVariant2.ssp")
variantC = model2.variants["Variant-B"].duplicate("Variant-C")
model2.add(variantC)

# Modify Variant-C independently
variantC.setValue(CRef("default", "Add1", "u1"), -100.0)
variantC.addComponent(CRef("default", "Add2"), "resources/Modelica.Blocks.
↳Math.Add.fmu")

# Switch active variant to Variant-C
model2.activeVariantName = "Variant-C"

# Add new component only to Variant-C
model2.addComponent(CRef("default", "Add3"), "resources/Modelica.Blocks.Math.
↳Add.fmu")
variantC.setValue(CRef("default", "Add3", "u1"), -200.0)
variantC.setValue(CRef("default", "Add3", "k1"), -30.0)

# Print the structure of the model
model2.list()
```

4.25 getVariant

Get the list of available variants for a given system or sub-system within the SSP model.

Syntax:

```
getVariant(cref : None | str) -> SSD | None
```

Parameters:

- cref : variant name (e.g) SystemStructure, VarA, VarB, if None is given then it will return the current active variant.

Example usage:

```
from OMSimulator import SSP, CRef
```

(continues on next page)

(continued from previous page)

```
# Load an existing SSP model
model = SSP("exportSSMTemplate1.ssp")
## get list of available variants
model.getAllVariantNames()
## get current active variant
varA = model.getVariant("varA")
```

4.26 listResource

Lists all resource files stored in the `resources/` folder of the SSP model.

This method is useful for inspecting the files available in the model, including FMUs, SSV parameter files, SSM mapping files, and other artifacts. It provides a quick overview of which resources are packaged inside the SSP archive.

Syntax:

```
listResource() -> list[str]
```

Returns:

- (`list[str]`): A list of resource file paths contained in the SSP model.

Example usage:

```
model = SSP("deleteResources1.ssp")
model.list()

print("list of Resources in SSP:")
print("=====")
print(model.listResource())
```

4.27 deleteResource

Removes a resource file from the `resources/` folder of the SSP model.

This method deletes the specified resource file from the model archive and updates the SSP structure accordingly. It is useful for cleaning up unused parameter sets, mapping files, or FMUs.

Notes:

- The file is physically removed from the SSP archive — it cannot be recovered unless re-added.
- If the resource is still referenced by components, connectors, or mappings, those references will become invalid and may cause errors.
- Can be used for any type of resource (e.g., FMU, SSV, SSM, or custom files).

Syntax:

```
deleteResource(resource_path)
```

Parameters:

- `resource_path (str)`: Path to the resource file to be deleted, relative to the SSP resources/ directory.

Example usage:

```
model = SSP("deleteResources1.ssp")
model.list()

# Delete a parameter set file
model.deleteResource("resources/delete3.ssv")
print("After deleting delete3.ssv:")
model.list()

# Delete another SSV file
model.deleteResource("resources/delete4.ssv")
print("After deleting delete4.ssv:")
model.list()

# Delete an FMU resource
model.deleteResource("resources/Add.fmu")
print("After deleting resources/Add.fmu:")
model.list()
```

4.28 delete

Deletes a connector, component, or subsystem from the SSP model.

This method removes the specified element from the model hierarchy. It can be used to: - Remove individual connectors (parameters or inputs/outputs) from a system or component. - Remove components from a subsystem or top-level system. - Remove entire subsystems, including all their child components and connectors.

Notes:

- Deleting a connector removes it from the system/component, but does not affect other connectors or components.
- Deleting a component removes all its connectors and any associated parameter values.
- Deleting a subsystem removes the subsystem and all nested components and connectors.
- Any SSV/SSM references pointing to deleted elements may become invalid.
- The operation modifies the active variant; other variants are unaffected unless explicitly modified.

Syntax:

```
delete(cref)
```

Parameters:

- **`cref (CRef)`: Reference to the element to delete. Can point to a connector, component, or subsystem. Examples:**

- Connector: CRef("default", "param1")
- Component: CRef("default", "Add1")
- Subsystem: CRef("default", "sub-system")

Example usage:

```
from OMSimulator import SSP, CRef

# Load an existing SSP model
model2 = SSP("deleteConnector1.ssp")

# Delete individual connectors
model2.delete(CRef("default", "param1"))
model2.delete(CRef("default", "sub-system", "input"))
model2.delete(CRef("default", "sub-system", "Add2", "u1"))
model2.delete(CRef("default", "Add1", "u1"))

print("## After deleting connectors:")
model2.list()

# Delete an entire component
model2.delete(CRef("default", "sub-system", "Add2"))
print("## After deleting component Add2 from sub-system:")
model2.list()

# Delete an entire sub-system
model2.delete(CRef('default', 'sub-system'))
print("## After deleting sub-system:")
model2.list()

# Delete the entire system
model2.delete(CRef('default'))
print("## After deleting full system:")
model2.list()
```

4.29 instantiate

Instantiates an SSP model for simulation by creating an internal runtime representation. After instantiation, users can modify parameter values on the instantiated model and specify the simulation result file before running the simulation.

Notes:

- All start values from SSV files and inline SSP `setValue` calls are applied initially.
- Values set via `setValue` on the instantiated model override the initial values.
- The instantiated model is independent of the SSP file on disk
- After instantiation, simulation methods such as `initialize()`, `simulate()`, `terminate()`, and `delete()` can be used.
- The instantiated model should be deleted using `delete()` to free resources after simulation.

- `setResultFile()` should be called before simulation to define the output file for logged results.

Syntax:

```
instantiated_model = model.instantiate() -> InstantiatedSSP
```

Returns:

- (*InstantiatedSSP*): A runtime object representing the SSP model, ready for simulation.

Example usage:

```
from OMSimulator import SSP, CRef, Settings

# import SSP
model2 = SSP("SimpleSimulation2.ssp")
model2.list()

# Instantiate the model for simulation
instantiated_model = model2.instantiate() # Start values from SSV and inline_
↳ setValue are applied

# Set result file
instantiated_model.setResultFile("SimpleSimulation6_res.mat")

# Override parameter values at runtime
instantiated_model.setValue(CRef("default", "Gain1", "k"), 2.0)
instantiated_model.setValue(CRef("default", "Gain1", "u"), 6.0)
```

4.30 setResultFile

Specifies the file where simulation results of an instantiated SSP model will be stored.

Notes:

- The file path can be relative or absolute.
- Supported formats depend on the simulator backend (commonly .mat or .csv files).
- Must be called before *simulate()* to ensure proper logging of results.

Syntax:

```
setResultFile(file_path)
```

Parameters:

- `file_path` (*str*): Path to the simulation result file.

Example usage:

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True
```

(continues on next page)

(continued from previous page)

```
# Load SSP and instantiate model
model = SSP("SimpleSimulation2.ssp")
instantiated_model = model.instantiate()

# Specify the result file
instantiated_model.setResultFile("SimpleSimulation6_res.mat")
```

4.31 initialize

Initializes an instantiated SSP model and prepares it for simulation.

Notes:

- Must be called **after** *instantiate()* and **before** *simulate()*.
- Applies all parameter and connector start values at runtime.
- Performs consistency checks on component connections and initial conditions.
- Initializes all FMU components and sets the model to a “ready-to-simulate” state.
- Does not perform the actual simulation; only prepares the model for it.

Syntax:

```
initialize()
```

Example usage:

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True

# Load SSP and instantiate model
model = SSP("SimpleSimulation2.ssp")
instantiated_model = model.instantiate()

# Apply result file and runtime parameters
instantiated_model.setResultFile("SimpleSimulation6_res.mat")
instantiated_model.setValue(CRef("default", "Gain1", "k"), 2.0)
instantiated_model.setValue(CRef("default", "Gain1", "u"), 6.0)

# Initialize the model before simulation
instantiated_model.initialize()
```

4.32 Solver

OMSimulator supports different solver methods depending on the FMI kind of the components in the system. Solvers are configured by name in the Simulation Setup dialog and assigned to individual FMU components.

4.32.1 Available Solvers

Co-Simulation (CS) Solvers

Co-simulation solvers act as masters that drive the co-simulation protocol. They are used for FMUs that implement the co-simulation interface (CS FMUs).

Method	Description
oms_ma	Master Algorithm — fixed-step co-simulation master. The step size is controlled by the <code>fixedStepSize</code> setting.
oms_mav	Master Algorithm with Variable Step — variable-step co-simulation master. Uses <code>initialStepSize</code> , <code>minimumStepSize</code> , and <code>maximumStepSize</code> to control the step size, and <code>relativeTolerance</code> for error control.
oms_mav2	Master Algorithm with Variable Step (v2) — an improved variable-step co-simulation master with enhanced error estimation. Supports the same settings as <code>oms_mav</code> .

Model-Exchange (ME) Solvers

Model-exchange solvers are ODE integrators used for FMUs that implement the model-exchange interface (ME FMUs). They integrate the continuous-time equations provided by the FMU.

Method	Description
ccode	CVODE — variable-step, variable-order ODE solver from the SUNDIALS suite. Suitable for stiff and non-stiff systems. Supports <code>initialStepSize</code> , <code>minimumStepSize</code> , <code>maximumStepSize</code> , and <code>relativeTolerance</code> .
euler	Explicit Euler — simple fixed-step explicit Euler integrator. Suitable for non-stiff systems. Controlled by <code>fixedStepSize</code> .

4.32.2 Solver Settings

The following settings can be configured for each named solver:

Setting	Applies to	Description
<code>relativeTolerance</code>	<code>oms_mav</code> , <code>oms_mav2</code> , <code>ccode</code>	Relative error tolerance used for step-size control.
<code>fixedStepSize</code>	<code>oms_ma</code> , <code>euler</code>	The fixed communication / integration step size.
<code>initialStepSize</code>	<code>oms_mav</code> , <code>oms_mav2</code> , <code>ccode</code>	The initial step size at the start of simulation.
<code>minimumStepSize</code>	<code>oms_mav</code> , <code>oms_mav2</code> , <code>ccode</code>	The minimum allowed step size.
<code>maximumStepSize</code>	<code>oms_mav</code> , <code>oms_mav2</code> , <code>ccode</code>	The maximum allowed step size.

4.32.3 Solver Assignment

Each FMU component in a system can be assigned one named solver configuration. The available solvers are filtered based on the FMI kind of the component:

- **CS FMUs** — only co-simulation solvers (`oms_ma`, `oms_mav`, `oms_mav2`) are available.
- **ME FMUs** — only model-exchange solvers (`cvode`, `euler`) are available.
- **me+cs FMUs** — all solvers are available.

If no solver is assigned to a component, the system default `oms_ma` is applied.

Example

The following example creates two named solver configurations — one using the fixed-step Euler method and one using the variable-step CVODE solver — and assigns them to individual components.

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True

model = SSP()

# Add FMU resources
model.addResource('../resources/Modelica.Blocks.Math.Add.fmu', new_name=
    ↪ 'resources/Add.fmu')
model.addResource('../resources/Modelica.Blocks.Math.Gain.fmu', new_name=
    ↪ 'resources/Gain.fmu')

# Instantiate components
model.addComponent(CRef('default', 'Add1'), 'resources/Add.fmu')
model.addComponent(CRef('default', 'Add2'), 'resources/Add.fmu')
model.addComponent(CRef('default', 'Gain1'), 'resources/Gain.fmu')
model.addComponent(CRef('default', 'Gain2'), 'resources/Gain.fmu')

# Define solver configurations
solver1 = {'name': 'solver1', 'method': 'euler', 'fixedStepSize': 1e-3}
model.newSolver(solver1)

solver2 = {'name': 'solver2', 'method': 'cvode', 'relativeTolerance': 1e-4,
    'initialStepSize': 1e-6, 'minimumStepSize': 1e-12, 'maximumStepSize':
    ↪ 0.001}
model.newSolver(solver2)

# Assign solvers to components
model.setSolver(CRef('default', 'Add1'), 'solver1')
model.setSolver(CRef('default', 'Add2'), 'solver2')
model.setSolver(CRef('default', 'Gain1'), 'solver1')
model.setSolver(CRef('default', 'Gain2'), 'solver2')
```

4.33 simulate

Runs the simulation of an instantiated SSP model.

Notes:

- Must be called **after** *initialize()*.
- Simulation uses start values from SSV files and any parameters overridden via *setValue*.
- Logged results are saved in the format defined by *setResultFile()* (commonly *.mat*).
- Simulation is performed on the active variant of the SSP model.

Syntax:

```
simulate()
```

Example usage:

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True

# Load SSP and instantiate model
model = SSP("SimpleSimulation2.ssp")
instantiated_model = model.instantiate()

# Apply result file and runtime parameters
instantiated_model.setResultFile("SimpleSimulation6_res.mat")
instantiated_model.setValue(CRef("default", "Gain1", "k"), 2.0)
instantiated_model.setValue(CRef("default", "Gain1", "u"), 6.0)

# Initialize before simulation
instantiated_model.initialize()

# Run the simulation
instantiated_model.simulate()
```

4.34 terminate and delete

Terminates the simulation of an instantiated SSP model. It releases internal simulation resources, closes loggers, and ensures that the model state is marked as terminated.

Syntax:

```
terminate()
delete()
```

Example usage:

```
from OMSimulator import SSP, CRef, Settings

Settings.suppressPath = True

# Load SSP and instantiate model
model = SSP("SimpleSimulation2.ssp")
instantiated_model = model.instantiate()

# Apply result file and runtime parameters
instantiated_model.setResultFile("SimpleSimulation6_res.mat")
instantiated_model.setValue(CRef("default", "Gain1", "k"), 2.0)
instantiated_model.setValue(CRef("default", "Gain1", "u"), 6.0)

# Initialize and run simulation
instantiated_model.initialize()
instantiated_model.simulate()

# Terminate the simulation
instantiated_model.terminate()
# Clean up resources
instantiated_model.delete()
```


OPENMODELICASCRIPTING

This is a shared library that provides a OpenModelica Scripting interface for the OMSimulatorLib library.

5.1 Examples

```
loadOMSimulator();
oms_setTempDirectory("./temp/");
oms_newModel("model");
oms_addSystem("model.root", OpenModelica.Scripting.oms_system.oms_system_sc);

// instantiate FMUs
oms_addSubModel("model.root.system1", "FMUs/System1.fmu");
oms_addSubModel("model.root.system2", "FMUs/System2.fmu");

// add connections
oms_addConnection("model.root.system1.y", "model.root.system2.u");
oms_addConnection("model.root.system2.y", "model.root.system1.u");

// simulation settings
oms_setResultFile("model", "results.mat");
oms_setStopTime("model", 0.1);
oms_setFixedStepSize("model.root", 1e-4);

oms_instantiate("model");
oms_setReal("model.root.system1.x_start", 2.5);

oms_initialize("model");
oms_simulate("model");
oms_terminate("model");
oms_delete("model");
unloadOMSimulator();
```

5.2 OpenModelica Scripting Commands

5.3 addBus

Adds a bus to a given component.

```
status := oms_addBus(cref);
```

5.4 addConnection

Adds a new connection between connectors *A* and *B*. The connectors need to be specified as fully qualified component references, e.g., “*model.system.component.signal*”.

```
status := oms_addConnection(crefA, crefB, suppressUnitConversion);
```

The two arguments *crefA* and *crefB* get swapped automatically if necessary. The third argument *suppressUnitConversion* is optional and the default value is *false* which allows automatic unit conversion between connections, if set to *true* then automatic unit conversion will be disabled.

5.5 addConnector

Adds a connector to a given component.

```
status := oms_addConnector(cref, causality, type);
```

The second argument “*causality*”, should be any of the following,

```
"OpenModelica.Scripting.oms_causality.oms_causality_input"  
"OpenModelica.Scripting.oms_causality.oms_causality_output"  
"OpenModelica.Scripting.oms_causality.oms_causality_parameter"  
"OpenModelica.Scripting.oms_causality.oms_causality_bidir"  
"OpenModelica.Scripting.oms_causality.oms_causality_undefined"
```

The third argument *type*, should be any of the following,

```
"OpenModelica.Scripting.oms_signal_type.oms_signal_type_real"  
"OpenModelica.Scripting.oms_signal_type.oms_signal_type_integer"  
"OpenModelica.Scripting.oms_signal_type.oms_signal_type_boolean"  
"OpenModelica.Scripting.oms_signal_type.oms_signal_type_string"  
"OpenModelica.Scripting.oms_signal_type.oms_signal_type_enum"  
"OpenModelica.Scripting.oms_signal_type.oms_signal_type_bus"
```

5.6 addConnectorToBus

Adds a connector to a bus.

```
status := oms_addConnectorToBus(busCref, connectorCref);
```

5.7 addSignalsToResults

Add all variables that match the given regex to the result file.

```
status := oms_addSignalsToResults(cref, regex);
```

The second argument, i.e. regex, is considered as a regular expression (C++11). “.” and “(.)” can be used to hit all variables.

5.8 addSubModel

Adds a component to a system.

```
status := oms_addSubModel(cref, fmuPath);
```

5.9 addSystem

Adds a (sub-)system to a model or system.

```
status := oms_addSystem(cref, type);
```

The second argument **type**, should be any of the following,

```
"OpenModelica.Scripting.oms_system.oms_system_none"
"OpenModelica.Scripting.oms_system.oms_system_wc"
"OpenModelica.Scripting.oms_system.oms_system_sc"
```

5.10 compareSimulationResults

This function compares a given signal of two result files within absolute and relative tolerances.

```
status := oms_compareSimulationResults(filenameA, filenameB, var, relTol,
↪absTol);
```

The following table describes the input values:

Input	Type	Description
filenameA	String	Name of first result file to compare.
filenameB	String	Name of second result file to compare.
var	String	Name of signal to compare.
relTol	Number	Relative tolerance.
absTol	Number	Absolute tolerance.

The following table describes the return values:

Type	Description
Integer	1 if the signal is considered as equal, 0 otherwise

5.11 copySystem

Copies a system.

```
status := oms_copySystem(source, target);
```

5.12 delete

Deletes a connector, component, system, or model object.

```
status := oms_delete(cref);
```

5.13 deleteConnection

Deletes the connection between connectors *crefA* and *crefB*.

```
status := oms_deleteConnection(crefA, crefB);
```

The two arguments *crefA* and *crefB* get swapped automatically if necessary.

5.14 deleteConnectorFromBus

Deletes a connector from a given bus.

```
status := oms_deleteConnectorFromBus(busCref, connectorCref);
```

5.15 export

Exports a composite model to a SPP file.

```
status := oms_export(cref, filename);
```

5.16 exportDependencyGraphs

Export the dependency graphs of a given model to dot files.

```
status := oms_exportDependencyGraphs(cref, initialization, event, simulation);
```

5.17 exportSnapshot

Lists the SSD representation of a given model, system, or component.

Memory is allocated for *contents*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
(contents, status) := oms_exportSnapshot(cref);
```

5.18 freeMemory

Free the memory allocated by some other API. Pass the object for which memory is allocated.

This function is not needed for OpenModelicaScripting Interface

5.19 getBoolean

Get boolean value of given signal.

```
(value, status) := oms_getBoolean(cref);
```

5.20 getFixedStepSize

Gets the fixed step size. Can be used for the communication step size of co-simulation systems and also for the integrator step size in model exchange systems.

```
(stepSize, status) := oms_getFixedStepSize(cref);
```

5.21 getInteger

Get integer value of given signal.

```
(value, status) := oms_getInteger(cref);
```

5.22 getModelState

Gets the model state of the given model cref.

```
(modelState, status) := oms_getModelState(cref);
```

5.23 getReal

Get real value.

```
(value, status) := oms_getReal(cref);
```

5.24 getSolver

Gets the selected solver method of the given system.

```
(solver, status) := oms_getSolver(cref);
```

5.25 getStartTime

Get the start time from the model.

```
(startTime, status) := oms_getStartTime(cref);
```

5.26 getStopTime

Get the stop time from the model.

```
(stopTime, status) := oms_getStopTime(cref);
```

5.27 getSubModelPath

Returns the path of a given component.

```
(path, status) := oms_getSubModelPath(cref);
```

5.28 getSystemType

Gets the type of the given system.

```
(type, status) := oms_getSystemType(cref);
```

5.29 getTime

Get the current simulation time from the model.

```
(time, status) := oms_getTime(cref);
```

5.30 getTolerance

Gets the tolerance of a given system or component.

```
(relativeTolerance, status) := oms_getTolerance(cref);
```

5.31 getVariableStepSize

Gets the step size parameters.

```
(initialStepSize, minimumStepSize, maximumStepSize, status) := oms_  
↪getVariableStepSize(cref);
```

5.32 getVersion

Returns the library's version string.

```
version := oms_getVersion();
```

5.33 importFile

Imports a composite model from a SSP file.

```
(cref, status) := oms_importFile(filename);
```

5.34 importSnapshot

Loads a snapshot to restore a previous model state. The model must be in virgin model state, which means it must not be instantiated.

```
status := oms_importSnapshot(cref, snapshot);
```

5.35 initialize

Initializes a composite model.

```
status := oms_initialize(cref);
```

5.36 instantiate

Instantiates a given composite model.

```
status := oms_instantiate(cref);
```

5.37 list

Lists the SSD representation of a given model, system, or component.

Memory is allocated for *contents*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
(contents, status) := oms_list(cref);
```

5.38 listUnconnectedConnectors

Lists all unconnected connectors of a given system.

Memory is allocated for *contents*. The caller is responsible to free it using the C-API. The Lua and Python bindings take care of the memory and the caller doesn't need to call free.

```
(contents, status) := oms_listUnconnectedConnectors(cref);
```

5.39 loadSnapshot

Loads a snapshot to restore a previous model state. The model must be in virgin model state, which means it must not be instantiated.

```
status := oms_loadSnapshot(cref, snapshot);
```

5.40 newModel

Creates a new and yet empty composite model.

```
status := oms_newModel(cref);
```

5.41 removeSignalsFromResults

Removes all variables that match the given regex to the result file.

```
status := oms_removeSignalsFromResults(cref, regex);
```

The second argument, i.e. regex, is considered as a regular expression (C++11). “.” and “(.)” can be used to hit all variables.

5.42 rename

Renames a model, system, or component.

```
status := oms_rename(cref, newCref);
```

5.43 reset

Reset the composite model after a simulation run.

The FMUs go into the same state as after instantiation.

```
status := oms_reset(cref);
```

5.44 setBoolean

Sets the value of a given boolean signal.

```
status := oms_setBoolean(cref, value);
```

5.45 setCommandLineOption

Sets special flags.

```
status := oms_setCommandLineOption(cmd);
```

5.46 setFixedStepSize

Sets the fixed step size. Can be used for the communication step size of co-simulation systems and also for the integrator step size in model exchange systems.

```
status := oms_setFixedStepSize(cref, stepSize);
```

5.47 setInteger

Sets the value of a given integer signal.

```
status := oms_setInteger(cref, value);
```

5.48 setLogFile

Redirects logging output to file or std streams. The warning/error counters are reset.

filename="" to redirect to std streams and proper filename to redirect to file.

```
status := oms_setLogFile(filename);
```

5.49 setLoggingInterval

Set the logging interval of the simulation.

```
status := oms_setLoggingInterval(cref, loggingInterval);
```

5.50 setLoggingLevel

Enables/Disables debug logging (logDebug and logTrace).

0 default, 1 default+debug, 2 default+debug+trace

```
oms_setLoggingLevel(logLevel);
```

5.51 setReal

Sets the value of a given real signal.

```
status := oms_setReal(cref, value);
```

This function can be called in different model states:

- Before instantiation: *setReal* can be used to set start values or to define initial unknowns (e.g. parameters, states). The values are not immediately applied to the simulation unit, since it isn't actually instantiated.
- After instantiation and before initialization: Same as before instantiation, but the values are applied immediately to the simulation unit.
- After initialization: Can be used to force external inputs, which might cause discrete changes of continuous signals.

5.52 setRealInputDerivative

Sets the first order derivative of a real input signal.

This can only be used for CS-FMU real input signals.

```
status := oms_setRealInputDerivative(cref, value);
```

5.53 setResultFile

Set the result file of the simulation.

```
status := oms_setResultFile(cref, filename);  
status := oms_setResultFile(cref, filename, bufferSize);
```

The creation of a result file is omitted if the filename is an empty string.

5.54 setSolver

Sets the solver method for the given system.

```
status := oms_setSolver(cref, solver);
```

The second argument "solver" should be any of the following,

```
"OpenModelica.Scripting.oms_solver.oms_solver_none"  
"OpenModelica.Scripting.oms_solver.oms_solver_sc_min"  
"OpenModelica.Scripting.oms_solver.oms_solver_sc_explicit_euler"  
"OpenModelica.Scripting.oms_solver.oms_solver_sc_cvode"  
"OpenModelica.Scripting.oms_solver.oms_solver_sc_max"  
"OpenModelica.Scripting.oms_solver.oms_solver_wc_min"  
"OpenModelica.Scripting.oms_solver.oms_solver_wc_ma"  
"OpenModelica.Scripting.oms_solver.oms_solver_wc_mav"  
"OpenModelica.Scripting.oms_solver.oms_solver_wc_mav2"  
"OpenModelica.Scripting.oms_solver.oms_solver_wc_max"
```

5.55 setStartTime

Set the start time of the simulation.

```
status := oms_setStartTime(cref, startTime);
```

5.56 setStopTime

Set the stop time of the simulation.

```
status := oms_setStopTime(cref, stopTime);
```

5.57 setTempDirectory

Set new temp directory.

```
status := oms_setTempDirectory(newTempDir);
```

5.58 setTolerance

Sets the tolerance for a given model or system.

```
status := oms_setTolerance(const char* cref, double relativeTolerance);
```

Default values are $1e-4$ for both relative and absolute tolerances.

A tolerance specified for a model is automatically applied to its root system, i.e. both calls do exactly the same:

```
oms_setTolerance("model", relativeTolerance);  
oms_setTolerance("model.root", relativeTolerance);
```

Component, e.g. FMUs, pick up the tolerances from there system. That means it is not possible to define different tolerances for FMUs in the same system right now.

In a strongly coupled system (*oms_system_sc*), the relative tolerance is used for CVODE and the absolute tolerance is used to solve algebraic loops.

In a weakly coupled system (*oms_system_wc*), both the relative and absolute tolerances are used for the adaptive step master algorithms and the absolute tolerance is used to solve algebraic loops.

5.59 setVariableStepSize

Sets the step size parameters for methods with stepsize control.

```
status := oms_getVariableStepSize(cref, initialStepSize, minimumStepSize, ↵  
↵maximumStepSize);
```

5.60 setWorkingDirectory

Set a new working directory.

```
status := oms_setWorkingDirectory(newWorkingDir);
```

5.61 simulate

Simulates a composite model.

```
status := oms_simulate(cref);
```

5.62 stepUntil

Simulates a composite model until a given time value.

```
status := oms_stepUntil(cref, stopTime);
```

5.63 terminate

Terminates a given composite model.

```
status := oms_terminate(cref);
```

GRAPHICAL MODELLING

OMSimulator offers two graphical editors for composite (SSP) models, covering different needs:

- *OMEdit* – OpenModelica’s own Connection Editor. Requires a full OpenModelica installation, and integrates SSP editing alongside Modelica modelling, simulation and plotting.
- *OMSimulatorGui* – a lightweight, standalone GUI with no omc dependency, for users who only need to edit and simulate SSP models directly.

Both import and export composite models in the System Structure Description (SSD) format, part of the System Structure and Parameterization (SSP) standard.

6.1 OMEdit

OMSimulator provides a graphical modelling environment through OMEdit, the OpenModelica Connection Editor. This feature requires a full OpenModelica installation that includes OMSimulator.

Composite models are imported and exported in the System Structure Description (SSD) format, which is part of the System Structure and Parameterization (SSP) standard.

See also [FMI documentation](#) and [SSP documentation](#).

6.1.1 Architecture

OMEdit communicates with OMSimulator through a Python-based ZMQ server:

- **OMSimulatorGuiServer.py** — handles all GUI-driven requests (model management, element queries, solver settings, connections, etc.)
- **OMSimulatorSimulationServer.py** — handles simulation execution and result streaming

The GUI server is started automatically when the first SSP model is opened or created. A notification in the Messages Browser confirms the server is running, including the script path and ZMQ endpoint.

6.1.2 New SSP Model

A new and empty SSP model can be created from **File -> New -> SSP** menu item.

A dialog opens to enter the model name and the name of the root system.

6.1.3 Open SSP Model

An existing SSP file (.ssp) can be opened from **File -> Open Model/Library**. If a model with the same name is already loaded, an error is reported and the file is not loaded again.

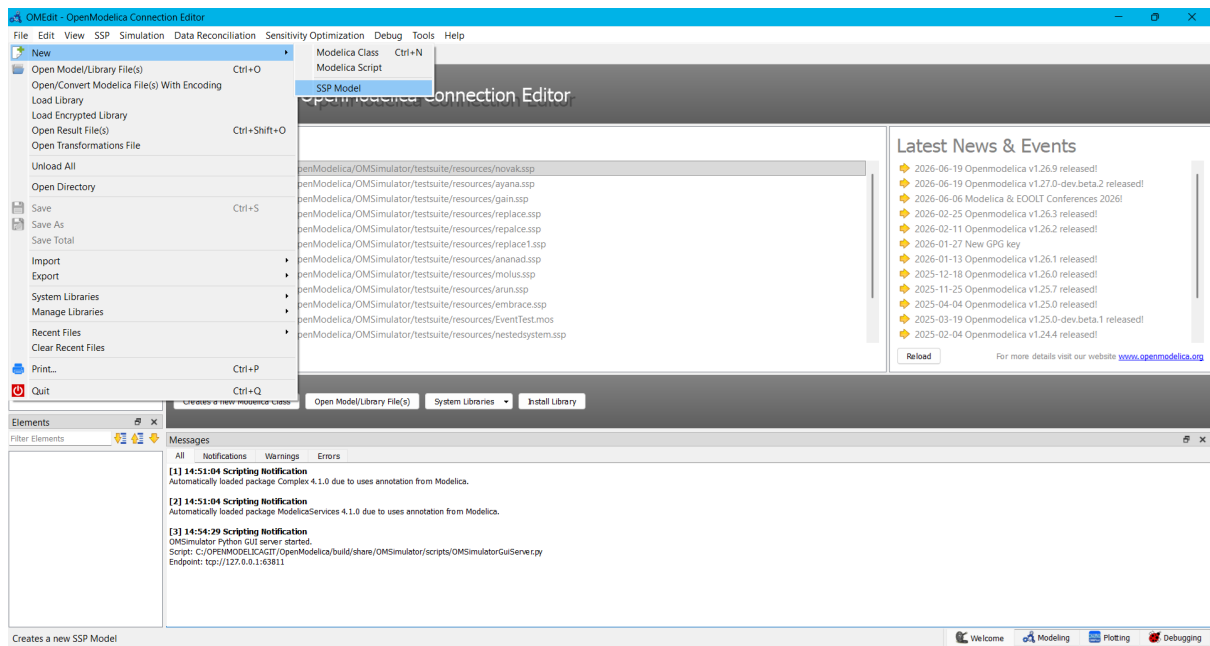


Fig. 1: OMEdit: New SSP Model

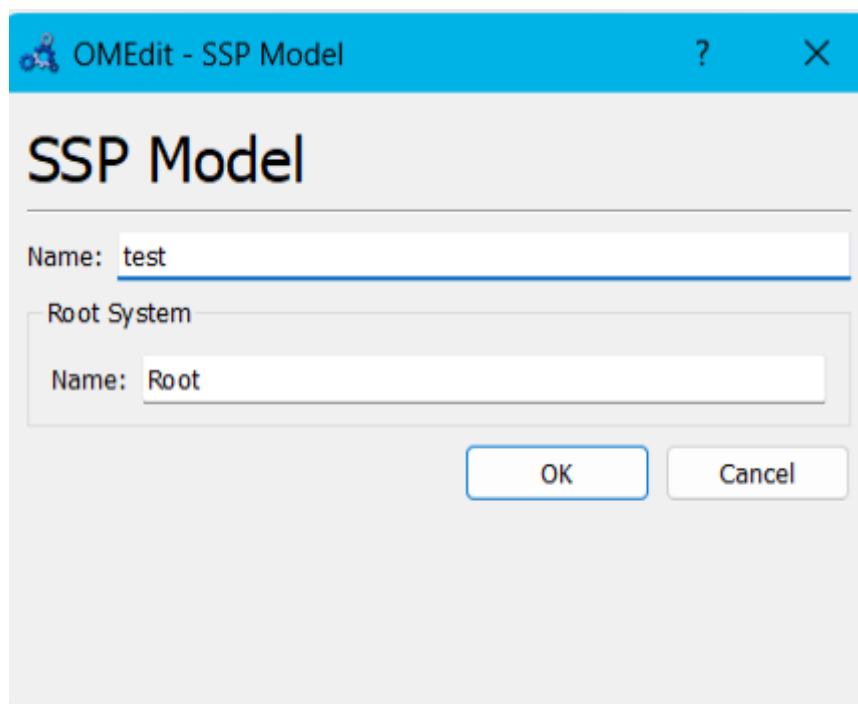


Fig. 2: OMEdit: New SSP Model Dialog

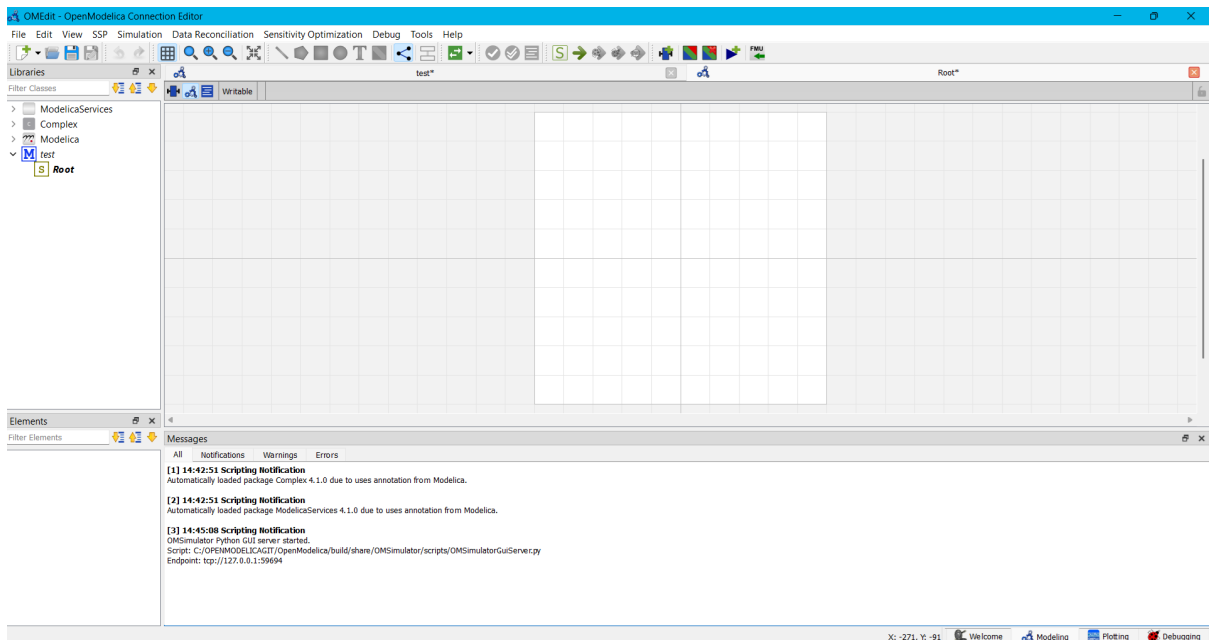


Fig. 3: OMEdit: Newly created empty root system of SSP model

6.1.4 Add System and Sub-Systems

A root system is always created together with the model. Additional subsystems can be added inside the root system via SSP -> Add System.

A dialog opens to enter the name of the new system.

6.1.5 Add SubModel

A sub-model is typically an FMU, but it can also be a result file (CSV). To add a sub-model, select the target system in the Libraries Browser and choose SSP -> Add SubModel.

A file browser opens to select an FMU (.fmu) or result file (.csv). A dialog then opens to set the name for the new sub-model.

6.1.6 Replace SubModel

An existing sub-model can be replaced with a different FMU by right-clicking on the component in the diagram view and selecting Replace SubModel from the context menu.

A file browser opens to select the new FMU. The replacement is performed in two steps:

1. **Dry run** (dryRun=true) — OMEdit first performs a dry run without making any changes to the model. It checks for interface differences between the old and new FMU, such as added, removed, or modified connectors and parameters. If any changes are detected, they are reported in the Messages Browser and a warning dialog is shown. The model remains unchanged at this stage, allowing the user to review the impact before deciding to proceed.
2. **Replacement** (dryRun=false) — If the user confirms, the replacement is carried out and all detected interface changes are applied to the model. Existing connections to removed or renamed connectors are dropped, and new connectors are available for connecting.

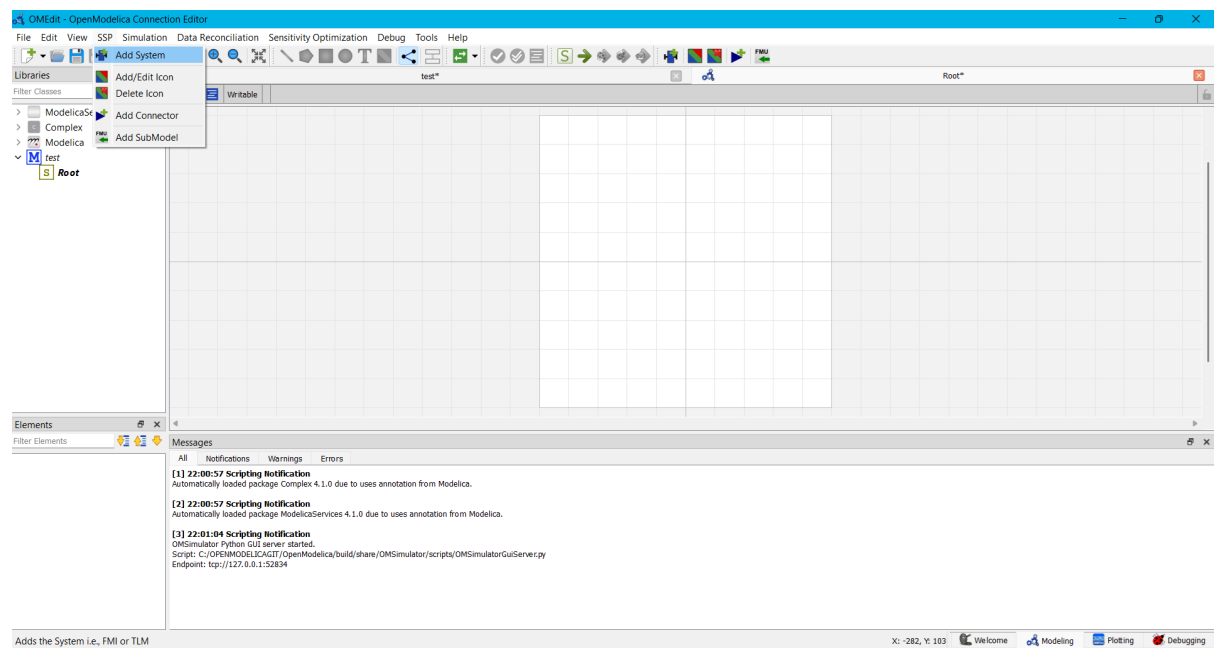


Fig. 4: OMEdit: Add System

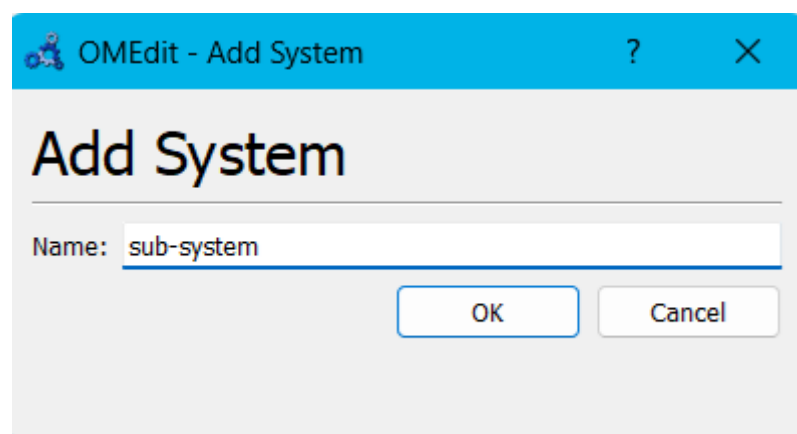


Fig. 5: OMEdit: Add System Dialog

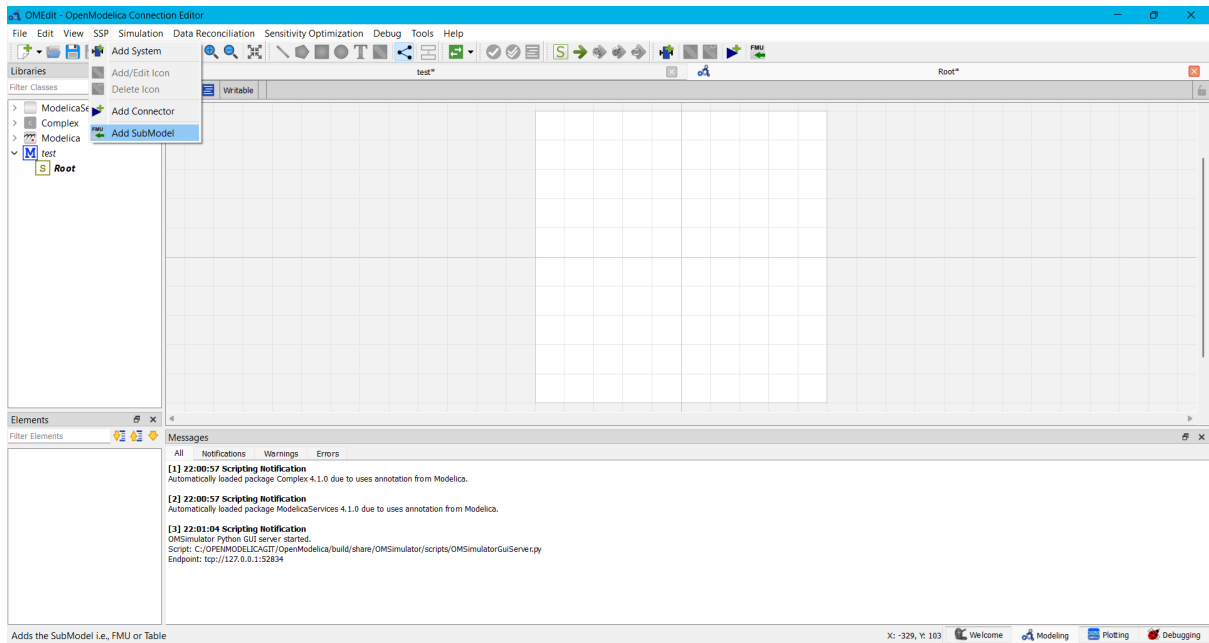


Fig. 6: OMEdit: Add SubModel

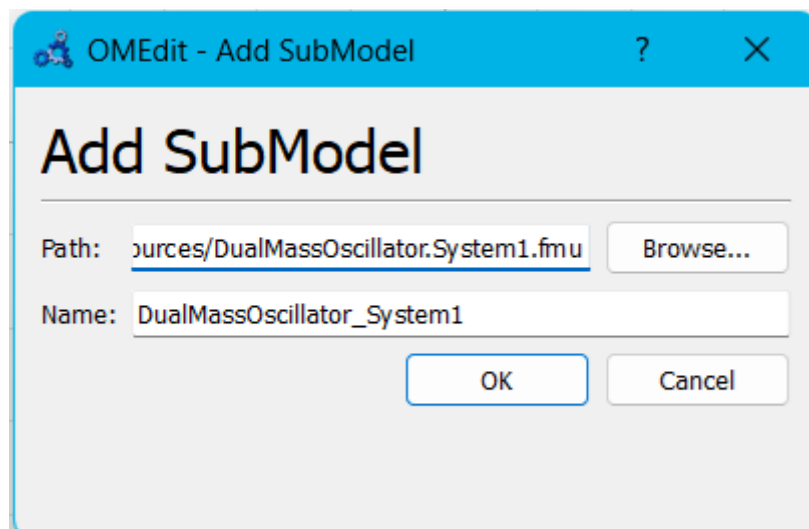


Fig. 7: OMEdit: Add SubModel Dialog

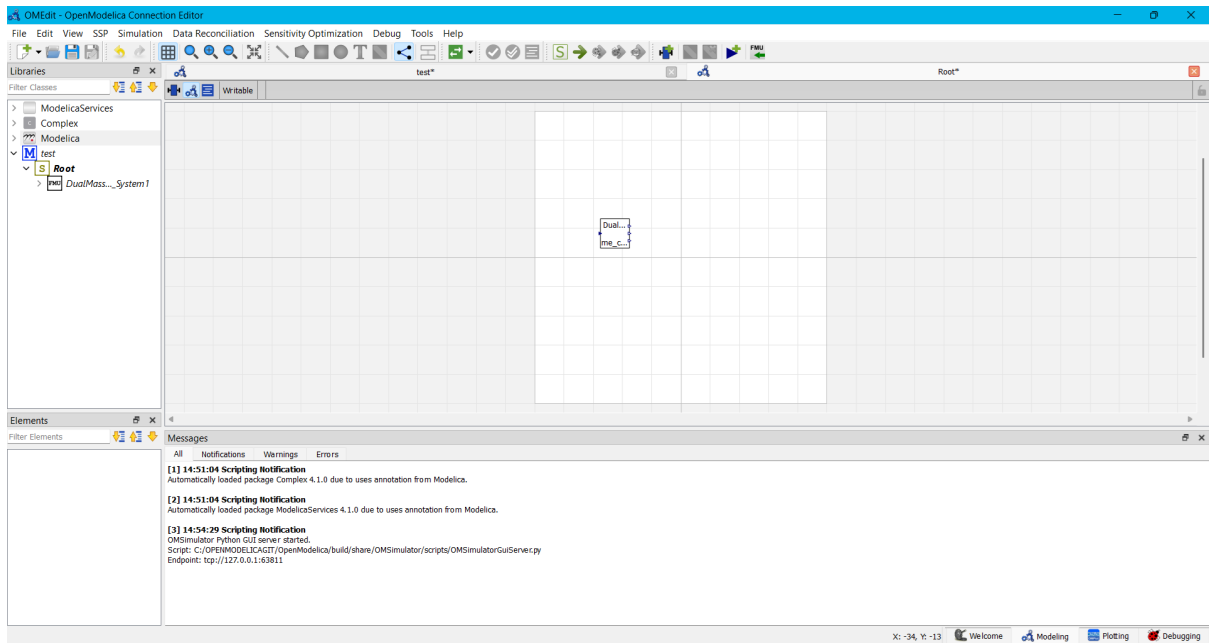


Fig. 8: OMEdit: Root system with added FMU.

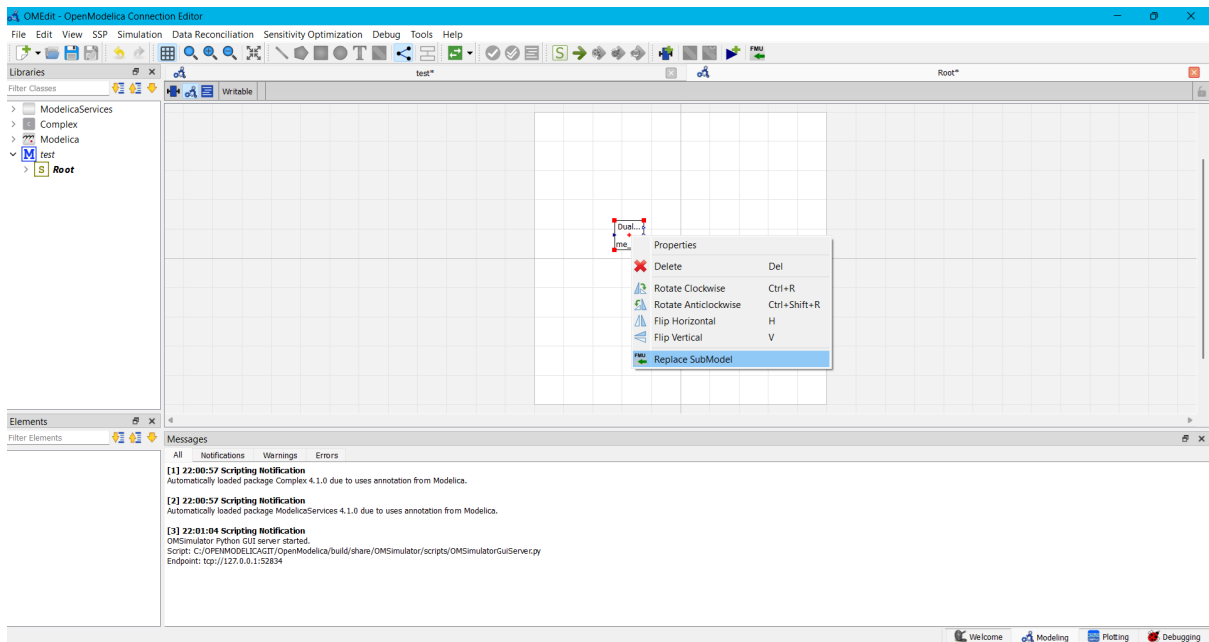


Fig. 9: OMEdit: Replace SubModel Dialog.

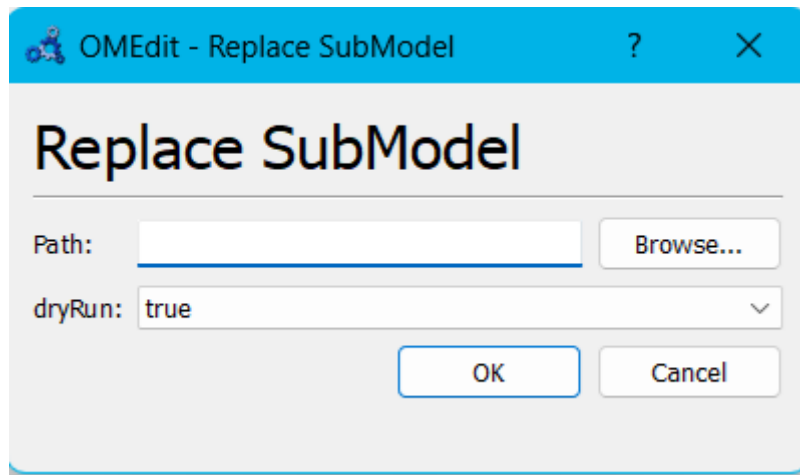


Fig. 10: OMEdit: Replace SubModel context menu.

6.1.7 Simulation Setup

Select **Simulation** -> **Simulation Setup** to configure simulation parameters before running. The dialog has two tabs: **General** and **Solver Settings**.

The **General** tab covers start time, stop time, and result file settings.

The **Solver Settings** tab is divided into two parts:

Solver Configurations

This table lists the named solver configurations available in the model. Each configuration has a name and a method. Click **Add** to create a new solver configuration — a new row appears in the table with a default method of oms-ma. The method can be changed using the combo box in the Method column.

To fine-tune a solver's numerical parameters, select the row and click **Edit**. This opens the Solver Settings dialog:

The available fields depend on the solver method:

- **Fixed-step solvers** (oms-ma, euler) — Fixed Step Size is editable; Initial Step Size, Minimum Step Size, and Maximum Step Size are disabled.
- **Variable-step solvers** (oms-mav, oms-mav-2, cvode) — Initial Step Size, Minimum Step Size, Maximum Step Size, and Relative Tolerance are editable; Fixed Step Size is disabled.

Click **Remove** to delete the selected solver configuration.

Component Assignments

This table lists all FMU components in the model. Each component can be assigned one of the named solver configurations using the combo box in the Solver column. The available solvers are filtered by the FMI kind of each component:

- **Co-Simulation FMUs** — only co-simulation masters are selectable (oms-ma, oms-mav, oms-mav-2).
- **Model-Exchange FMUs** — only ODE integrators are selectable (cvode, euler).
- **me+cs FMUs** — all solvers are available.

Incompatible solvers are shown as disabled in the combo box with a tooltip explaining why. If no solver is assigned, (none) is used and the system default applies.

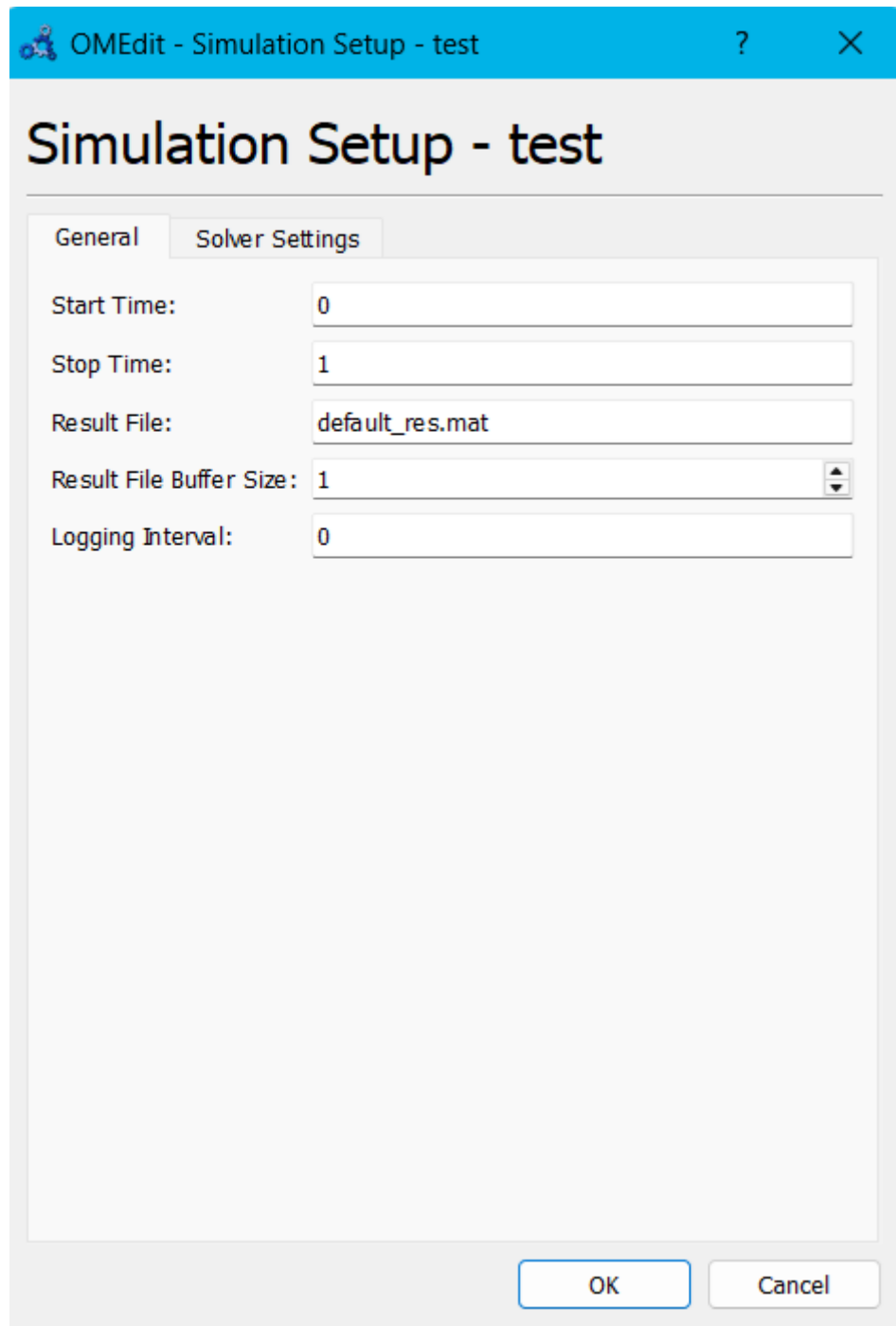


Fig. 11: OMEdit: Simulation Setup — Solver Settings tab with a solver configuration.

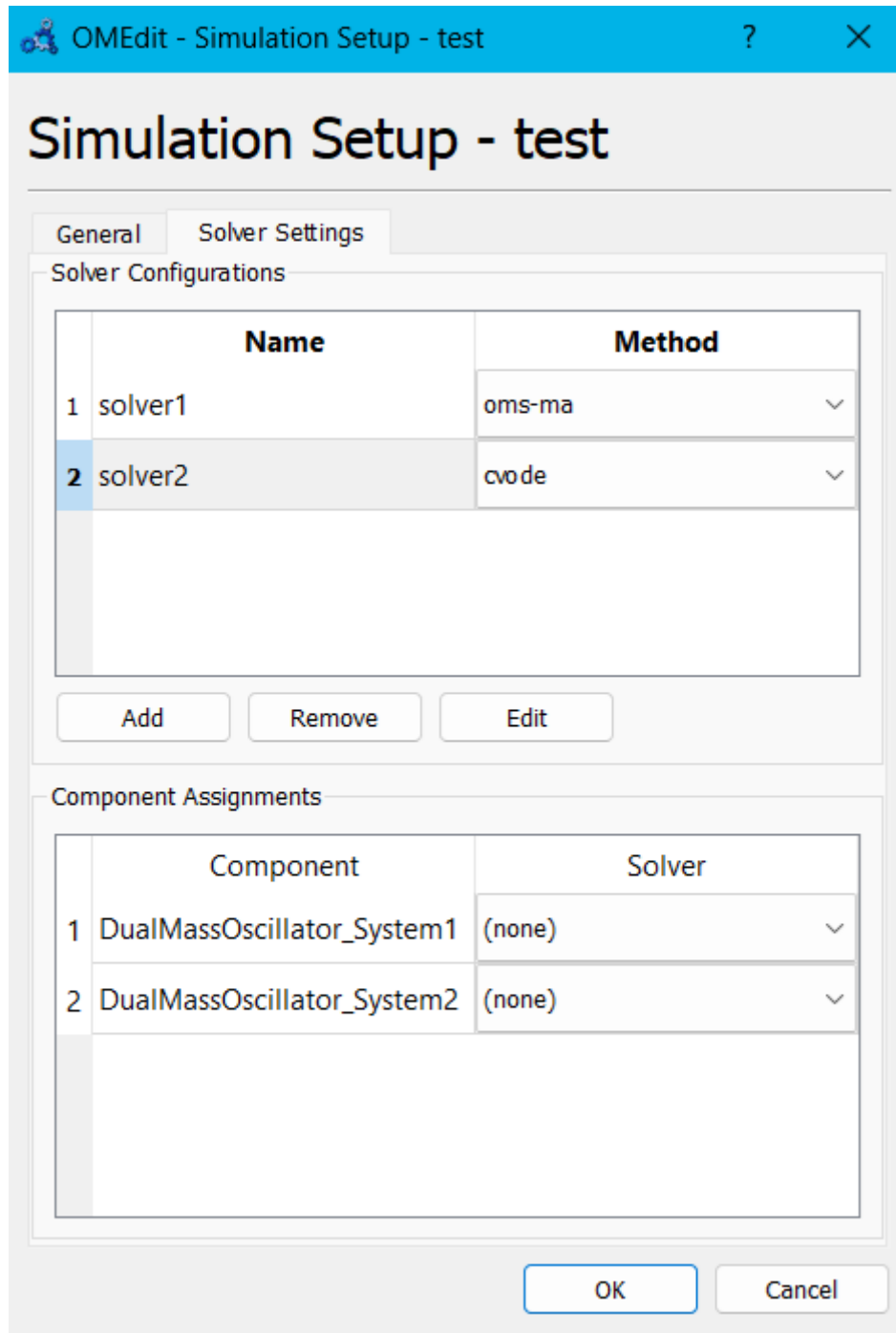


Fig. 12: OMEdit: Simulation Setup — Solver Settings tab with a solver configuration.

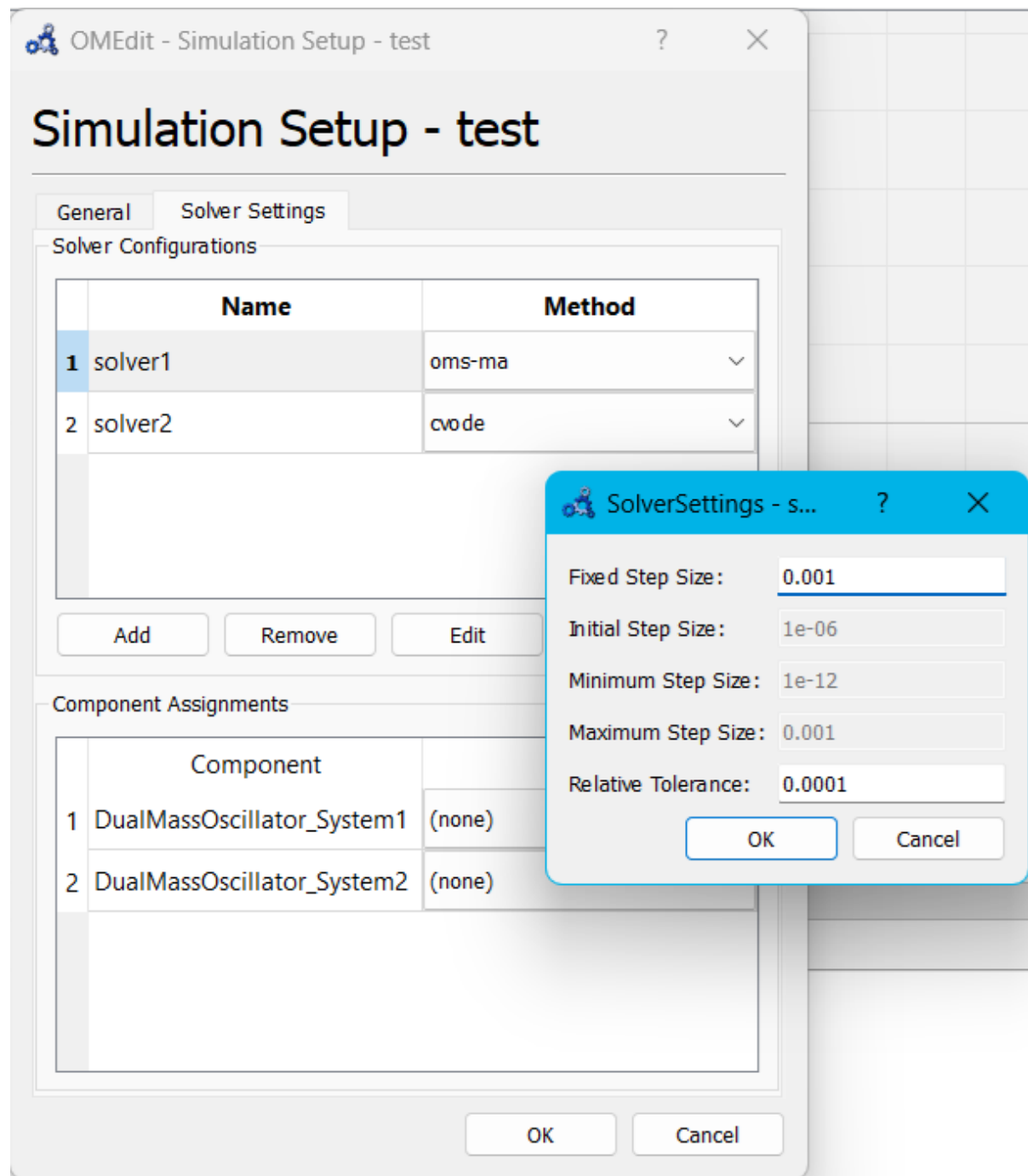


Fig. 13: OMEdit: Solver Settings dialog.

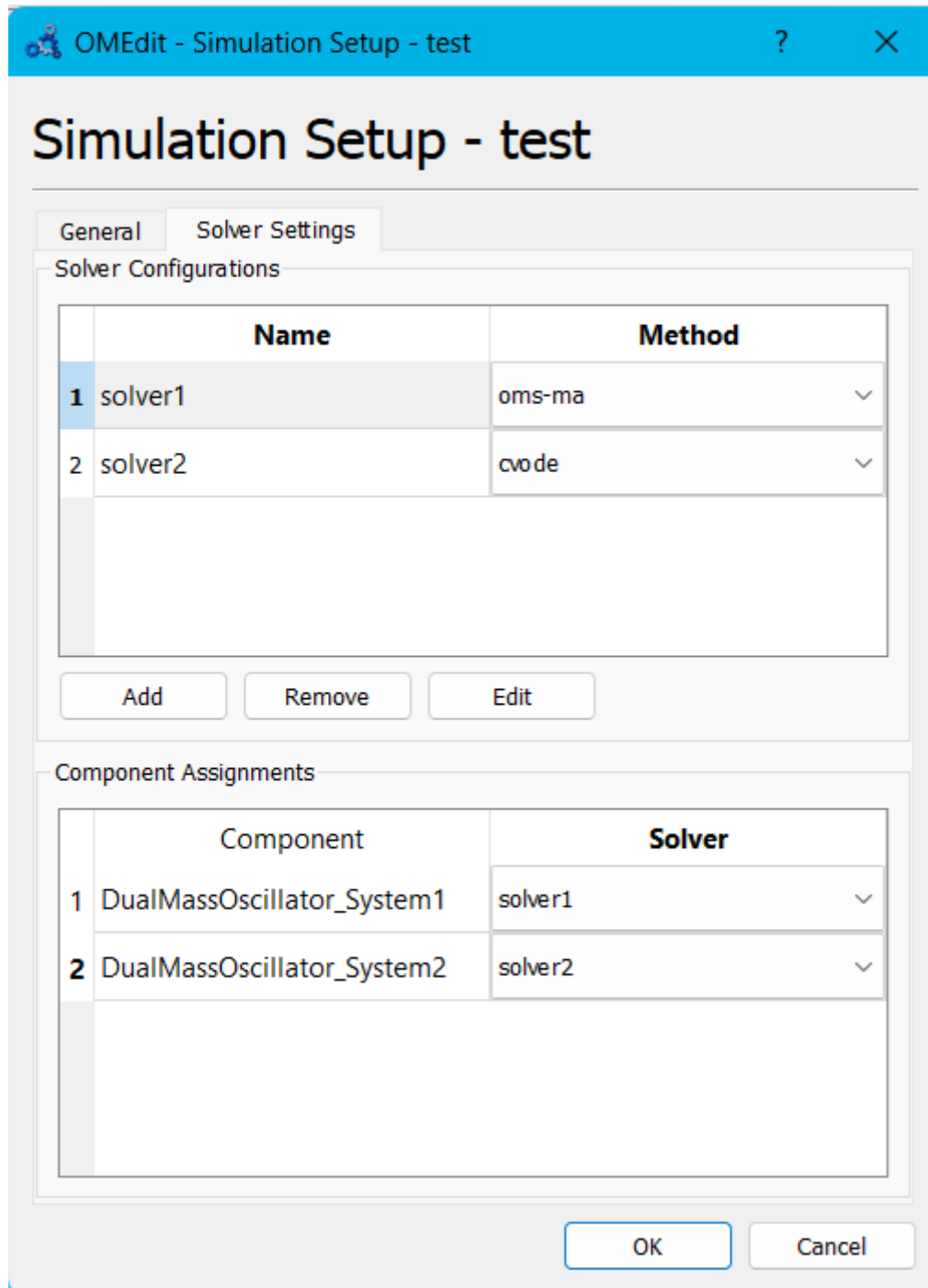


Fig. 14: OMEdit: Solver Assignments in Components

6.1.8 Simulate

Select the simulate button (green arrow) or **Simulation** -> **Simulate** to run the SSP model. Results are streamed back in real time via the simulation server.

6.1.9 Dual Mass Oscillator Example

The dual mass oscillator example from the test suite demonstrates a typical SSP workflow. The model is split into two sub-models, each exported as an FMU, and then connected in an SSP.

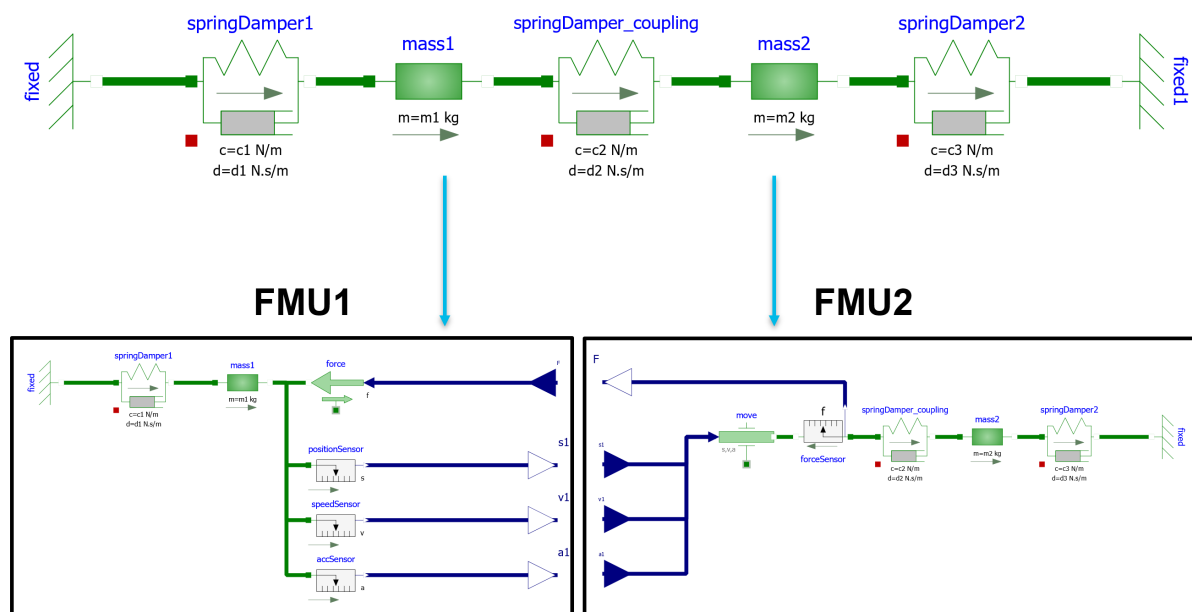


Fig. 15: Dual mass oscillator Modelica model (diagram view) and FMUs

6.2 OMSimulatorGui

OMSimulatorGui is a lightweight, standalone graphical editor for SSP models, built directly on top of the OMSimulator Python API (*OMSimulatorPython3*). Unlike *OMEdit*, it has no dependency on omc or the rest of the OpenModelica compiler – it only needs a plain OMSimulator installation with a couple of Python GUI packages on top, and it calls the same in-process Python API used by scripts, with no server process or ZMQ layer in between for editing.

It covers the everyday SSP editing workflow: building up a system structure with subsystems, FMU components and connectors, wiring them together on a diagram canvas, attaching parameter values and mappings (SSV/SSM), configuring solvers, and running a simulation with live progress and a result plot. It does not (yet) include a read-only XML/text viewer for the underlying SSD/SSV/SSM files – see *Current Limitations* below.

6.2.1 Requirements and Enabling the Build

OMSimulatorGui is installed together with OMSimulator:

```
cmake -S . -B build
cmake --build build --target install
```

It has no compiled component – it is pure Python, installed alongside the OMSimulator Python package – but it does need a couple of extra Python packages that the core OMSimulator CLI/API does not require:

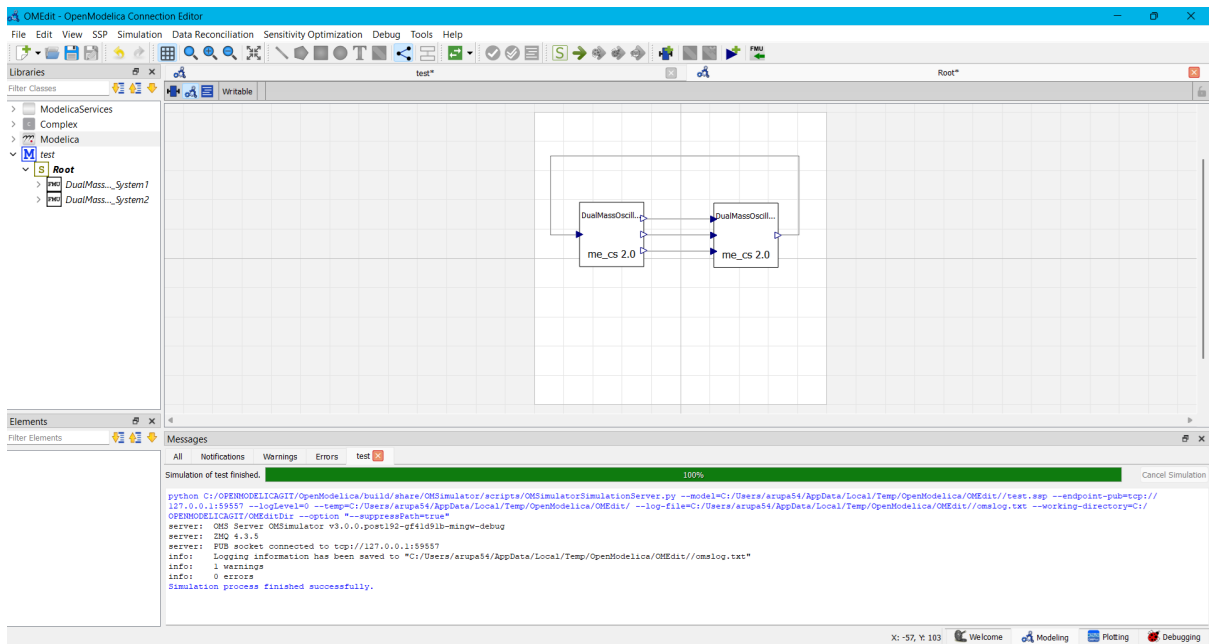


Fig. 16: OMEdit: Simulate Dual Mass Oscillator SSP model

- `PySide6` – the Qt widgets, the system tree, and the diagram canvas.
- `pyqtgraph` – the result plot.
- `scipy` – reading `.mat` result files.

Install them into whichever Python environment OMSimulator itself runs under:

```
pip install -r src/OMSimulatorGui/requirements.txt
```

6.2.2 Installing with pip

The OMSimulator package on PyPI already contains the GUI: the OMSimulatorGui Python package is installed next to OMSimulator in site-packages.

```
pip install OMSimulator
```

The GUI's own requirements (PySide6, pyqtgraph and scipy) are *not* installed by default, so the core package stays small. If you start the GUI without them, it tells you what is missing. Install them with the gui extra:

```
pip install OMSimulator[gui]
```

Installing the extra is also enough when OMSimulator is already installed: pip only adds the missing requirements. Then start the GUI with:

```
python -m OMSimulatorGui
```

6.2.3 Launching

A build with OMSimulatorGui enabled installs an OMSimulatorGui launcher script next to OMSimulator itself:

```
OMSimulatorGui
```

This sets up PYTHONPATH/PATH the same way OMSimulator.bat/.sh does for the CLI, then runs `python3 -m OMSimulatorGui`. It can also be invoked directly once the OMSimulator/OMSimulatorGui packages are importable:

```
python -m OMSimulatorGui
```

6.2.4 Main Window

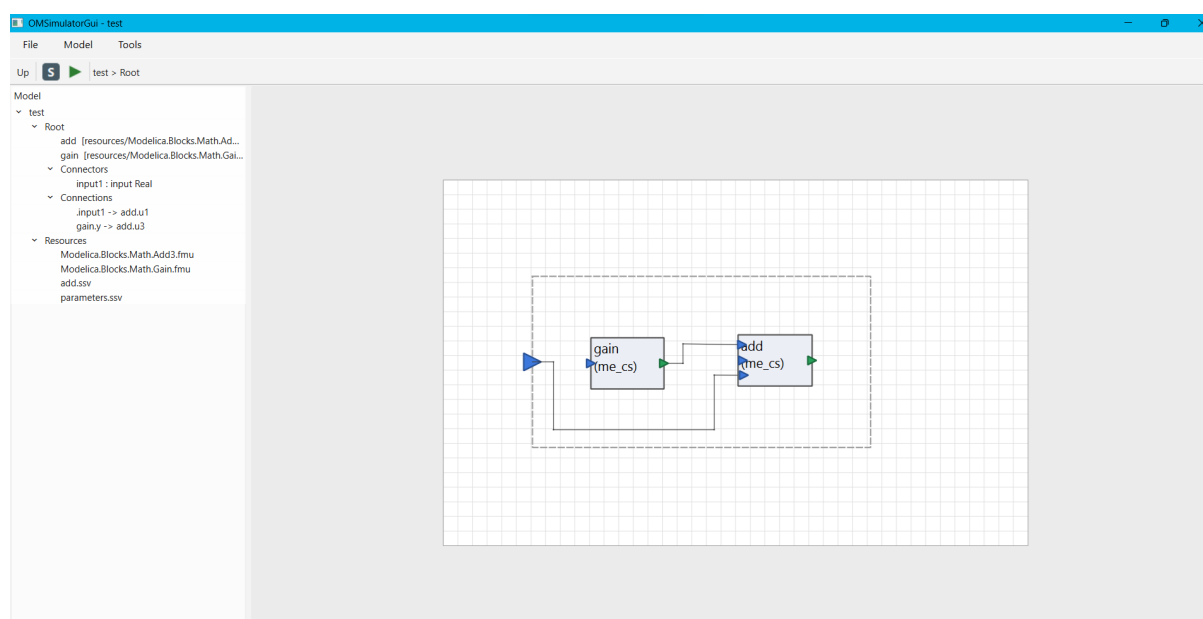


Fig. 17: OMSimulatorGui: main window with a model open, showing the system tree (left), the diagram canvas drilled into the root system “Root” (right), and the breadcrumb “test > Root” in the toolbar.

The main window has three parts:

- A **System Tree** on the left, showing every open model as its own top-level row, each with its root system, connectors, connections, parameter files and registered resources underneath.
- A **Diagram Canvas** on the right, showing one system’s contents at a time as boxes and connection lines – systems, components (FMUs) and connectors on its boundary.
- A **toolbar** above the canvas with an **Up** button (leave the current system and go back to its parent), a **Simulation Settings** button, a **Simulate** button (green arrow), and a breadcrumb showing the current position in the system hierarchy.

Three menus tie everything together:

- **File** – New SSP Model... (Ctrl+N), Open... (Ctrl+O), Save As... (Ctrl+Shift+S), Exit (Ctrl+Q).
- **Model** – Simulation Settings..., Variants..., Simulate (Ctrl+R) – all act on whichever model is currently active.

- **Tools** – Options... (see *Options (Working Directory and Command Line Options)* below).
- **Help** – About OMSimulatorGui..., showing the running OMSimulator version (the same one OMSimulator --version reports).

The tree and the canvas always show the same underlying model: editing on one immediately refreshes the other.

6.2.5 Creating and Opening Models

File -> New SSP Model... opens a dialog asking for the model's name and the name of its root system, then creates a new, empty SSP model and opens it as a new tab in the tree.

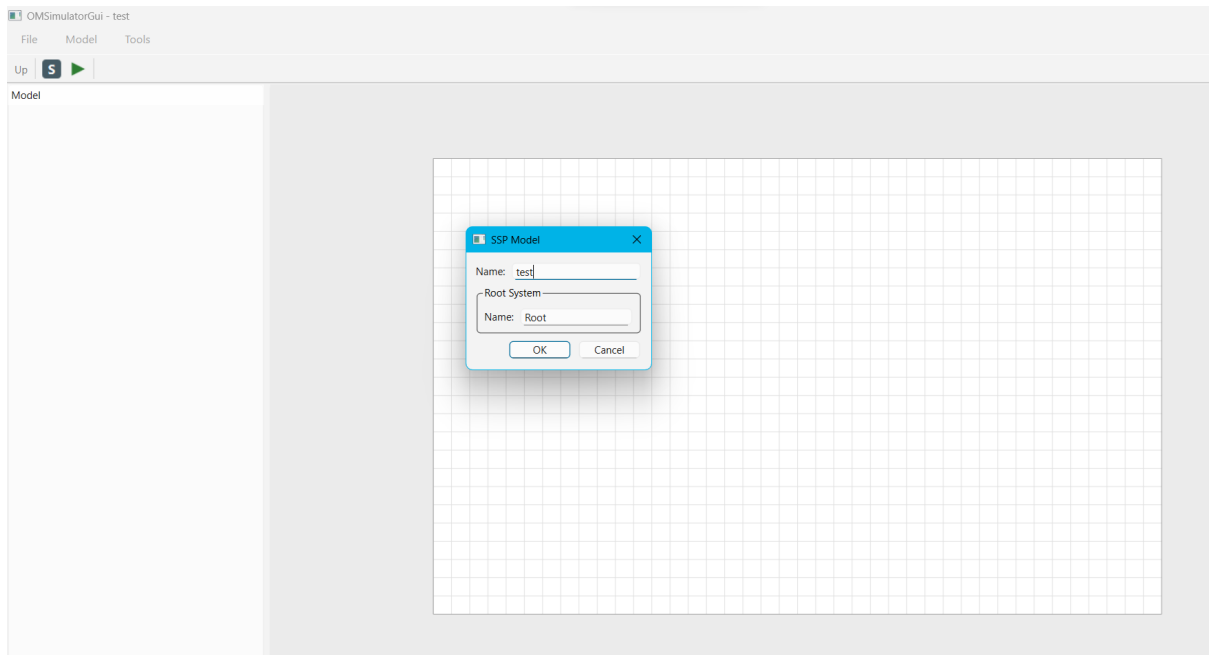


Fig. 18: OMSimulatorGui: New SSP Model dialog, entering the model name and the root system's own name.

File -> Open... opens a file browser to load an existing .ssp file. Several models can be open at the same time; each gets its own top-level row in the tree, and the diagram canvas and toolbar always reflect whichever model was last clicked on in the tree.

File -> Save As... exports the currently active model to a chosen .ssp path.

Multiple models can be open side by side. To close one, select its model row (or its root system row) in the tree and either press Delete or use **Delete** from its right-click menu – both close the whole model, not just one part of it.

6.2.6 The System Tree

Each open model's row expands into its root system, which in turn shows:

- **Systems and Components** – nested subsystems, FMU components, and result-file lookup tables (all three added via the same **Add Component...** action – see The Diagram Canvas below), each with their own Connectors/Connections/Parameter Files/Resources underneath as applicable.
- **Connectors** – the system's own boundary ports.
- **Connections** – wires between two connectors at this level.

- **Parameter Files** – SSV/SSM resources attached to this system or component (see *Parameter Files (SSV/SSM)* below).
- **Resources** – every file registered in the SSP’s shared resource pool (FMUs, SSV/SSM files, tables, ...), shown once under the model’s own row.

Right-clicking a row opens a context menu appropriate to that row’s kind:

- A **System** row offers Add System..., Add Component..., Add Connector..., Add Parameter File..., Export SSV/SSM Template..., Rename..., and Delete.
- A **Component** row offers Properties..., Add Parameter File..., Export SSV/SSM Template..., Rename..., and Delete.
- A **Component Table** row (a result-file lookup table, not an FMU) offers the same actions minus Properties... – it has no editable metadata of its own.
- A **Connector** row offers Delete.
- A **Parameter File** row offers Edit Values..., Swap Resource..., and Remove.
- A **Resource** row offers Remove.
- A model’s own top-level row offers Add Resource..., Rename..., and Delete (closes the whole model).

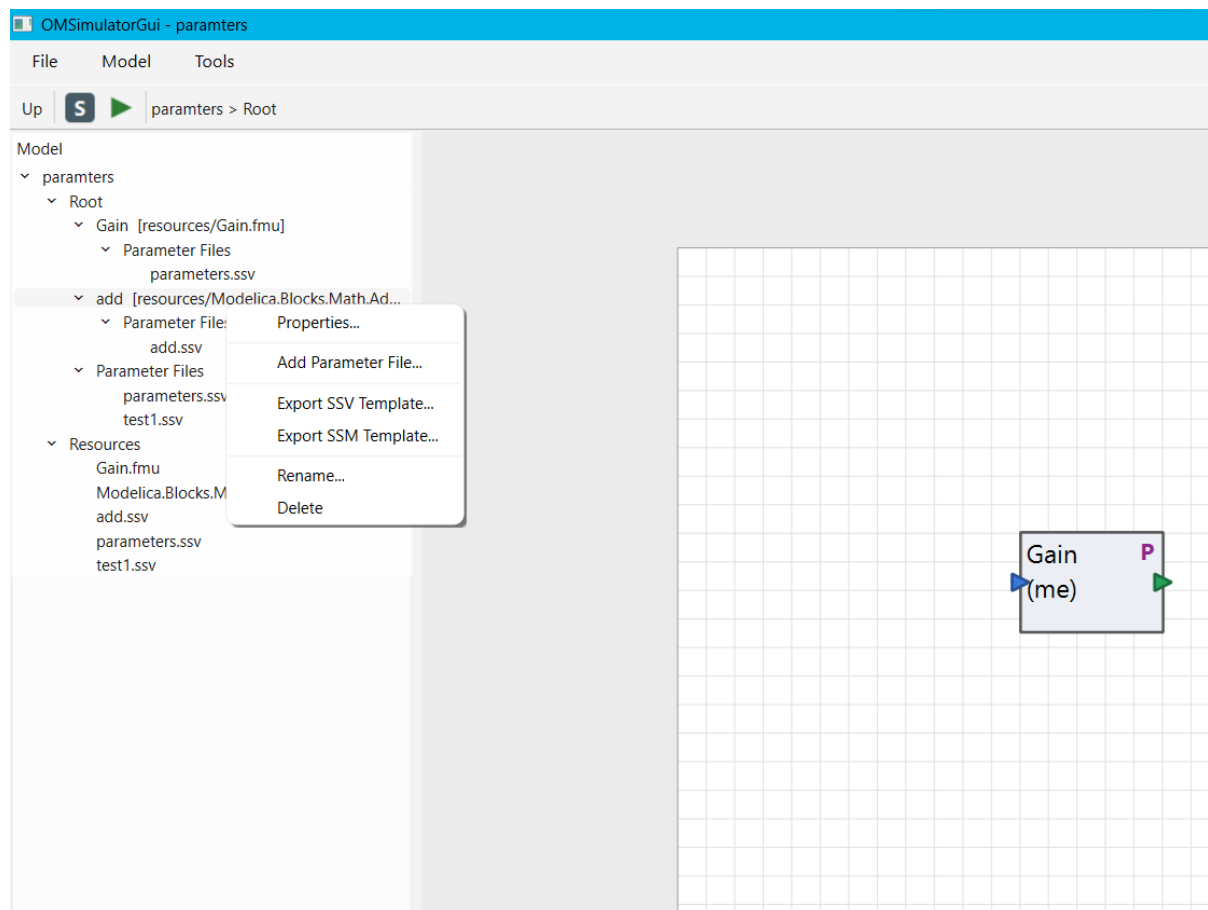


Fig. 19: OMSimulatorGui: right-click context menu on a Component row (“add”) in the tree – Properties..., Add Parameter File..., Export SSV/SSM Template..., Rename..., Delete. Right-clicking the same component’s box on the canvas offers the identical set of actions.

Double-clicking a Component row opens its Properties dialog; double-clicking a Parameter File row opens Edit Values. Double-clicking a Component Table row does nothing (there is no Properties dialog for one).

6.2.7 The Diagram Canvas

The canvas shows one system's contents at a time. Double-click a subsystem or component box to drill into it (an FMU component opens its Properties dialog instead, since it has no contents of its own; a component table box does nothing on double-click); use the toolbar's **Up** button, or the breadcrumb, to go back out.

Right-clicking empty canvas space offers **Add System...**, **Add Component...** and **Add Connector...**, placing the new element at the click position. Right-clicking an existing box offers the same system/component actions as the tree (Add Parameter File..., Export Template..., Rename..., Delete), plus **Properties...** for an FMU component.

Add Component... accepts either an FMU (.fmu) or a result-file lookup table (.csv/.mat) through the same dialog and file browser – matching OMEdit's own unified Add SubModel. Which kind is created depends only on the file's own extension: an FMU becomes a regular Component, while a .csv/.mat becomes a **Component Table** whose signals are always read from the file's header and always given **output** causality (there is nothing to configure on one – no Properties dialog, no double-click action, matching OMEdit).

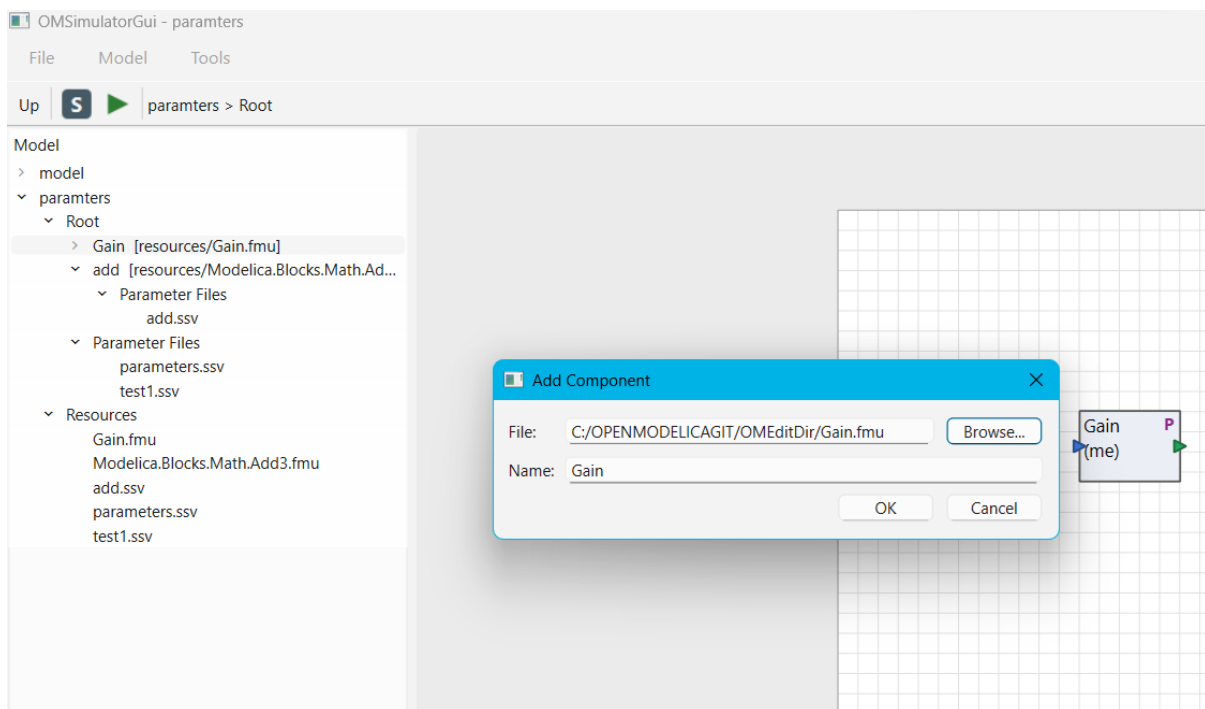


Fig. 20: OMSimulatorGui: Add Component dialog – a file (an FMU or a result-file table, via Browse...) and a name for the new component.

To connect two ports, drag from one to the other – the connection is only created if the two connectors are causality-compatible (e.g. an output to an input), otherwise nothing happens. A connection defaults to an automatic routed path but can be reshaped by hand: drag anywhere along it to insert or move a waypoint, right-click a waypoint to remove it. Moving a box shifts any connections attached to it by the same amount, so a manually-reshaped route does not go stale.

Select a box, port, or connection and press **Delete** to remove it, or use **Delete** from its context menu.

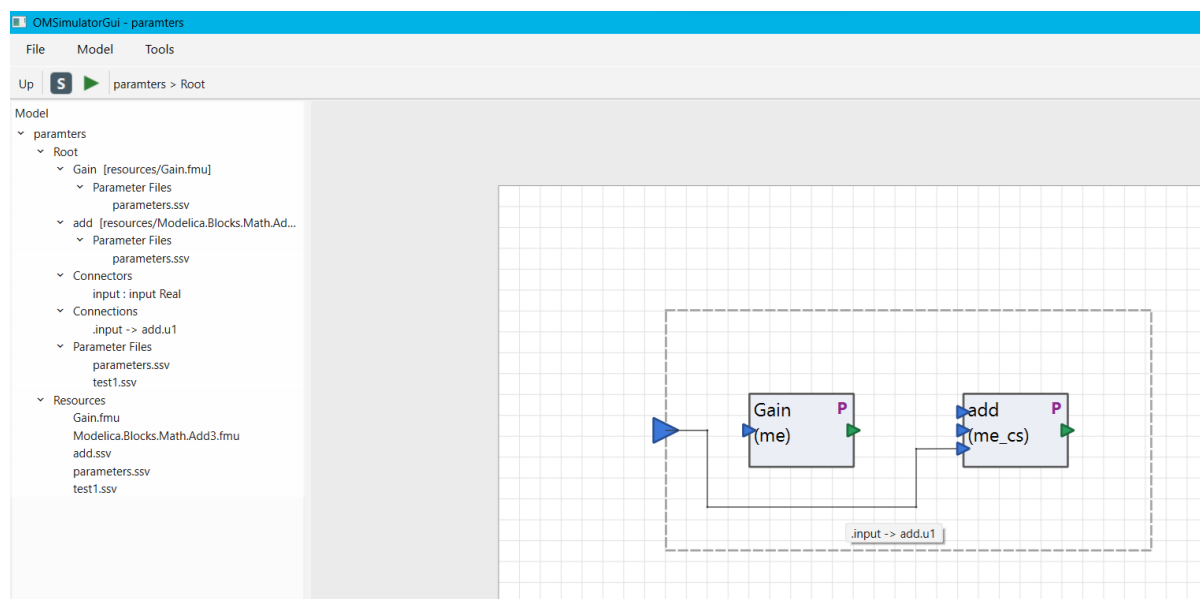


Fig. 21: OMSimulatorGui: a manually reshaped connection (“`.input -> add.u1`”) routed below and around both boxes instead of a straight line, with its identity shown in a hover tooltip.

Double-clicking a connection itself has no effect (it does not open or delete anything).

An input or output connector with no incoming connection can be given a constant value by double-clicking it directly on the canvas – this opens a small **Set Value** dialog. Leaving the value field blank clears any set value (the backend then defaults it to 0); no unit needs to be entered.

6.2.8 Element (Component) Properties

Double-clicking an FMU component (in the tree or on the canvas) opens its **Properties** dialog, with three tabs. A component table (a `.csv/ .mat` added via the same **Add Component...** action) has no such dialog – double-clicking one does nothing, since it has no metadata or parameters to edit.

- **Info** – FMU path, FMI kind (model-exchange/co-simulation/both) and other read-only metadata.
- **Parameters** – editable parameter values (Real/Integer/Boolean; String parameters are not yet supported, matching OMEdit).
- **Inputs** – editable input values, same types as above.

Only values the user actually changes are written back to the model – an untouched parameter keeps using whatever default the FMU itself declares.

6.2.9 Parameter Files (SSV/SSM)

A **parameter value file** (SSV) holds a named set of parameter/input values; a **parameter mapping file** (SSM) maps those names onto a different system or component’s own connector names (e.g. renaming or rescaling a value as it crosses into a subsystem). Both are ordinary SSP resources that get attached to a system or component – never to the “whole model” as a free-floating box, so on the canvas they only ever show as a small “P” badge on the system/component box they are attached to.

Add Parameter File... (on a System or Component row, or its canvas box) opens a dialog to either:

- create a brand-new SSV file by entering values directly, optionally also creating a companion SSM mapping (which can be saved empty, to be filled in later), or

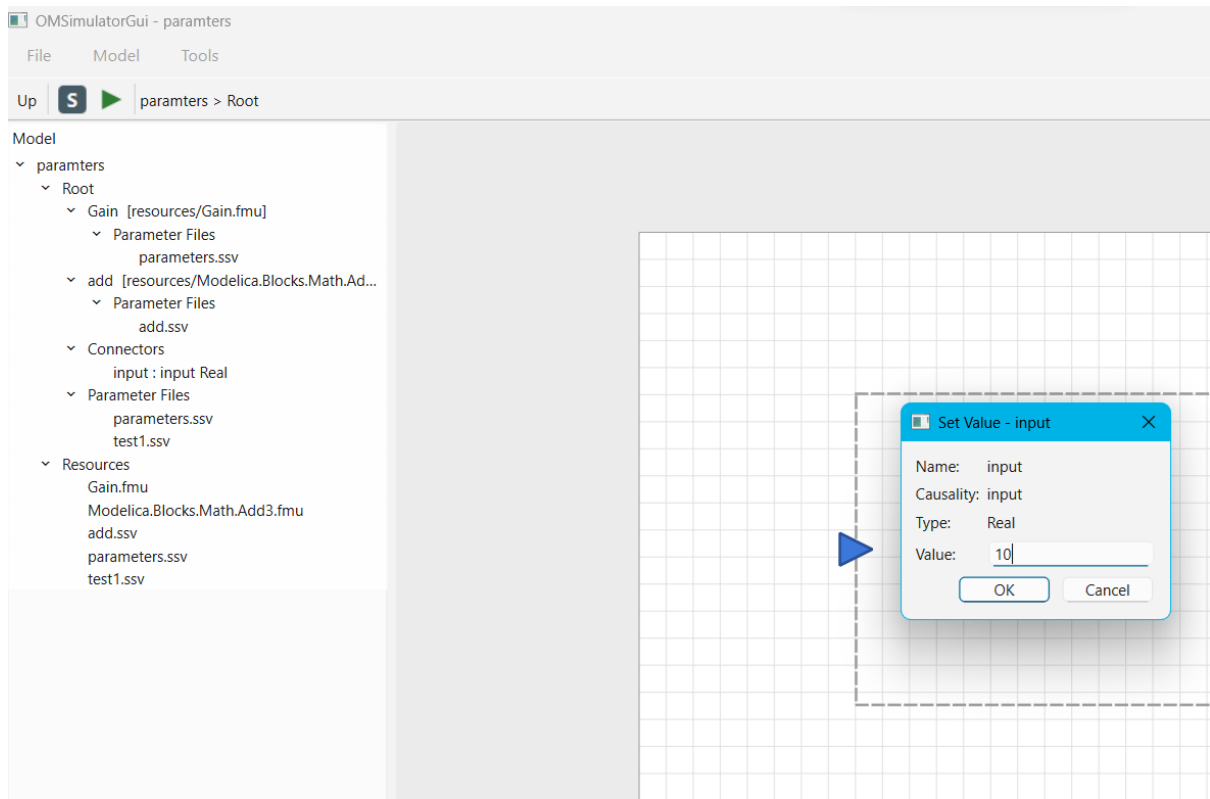


Fig. 22: OMSimulatorGui: Set Value dialog for the root system’s own “input” connector.

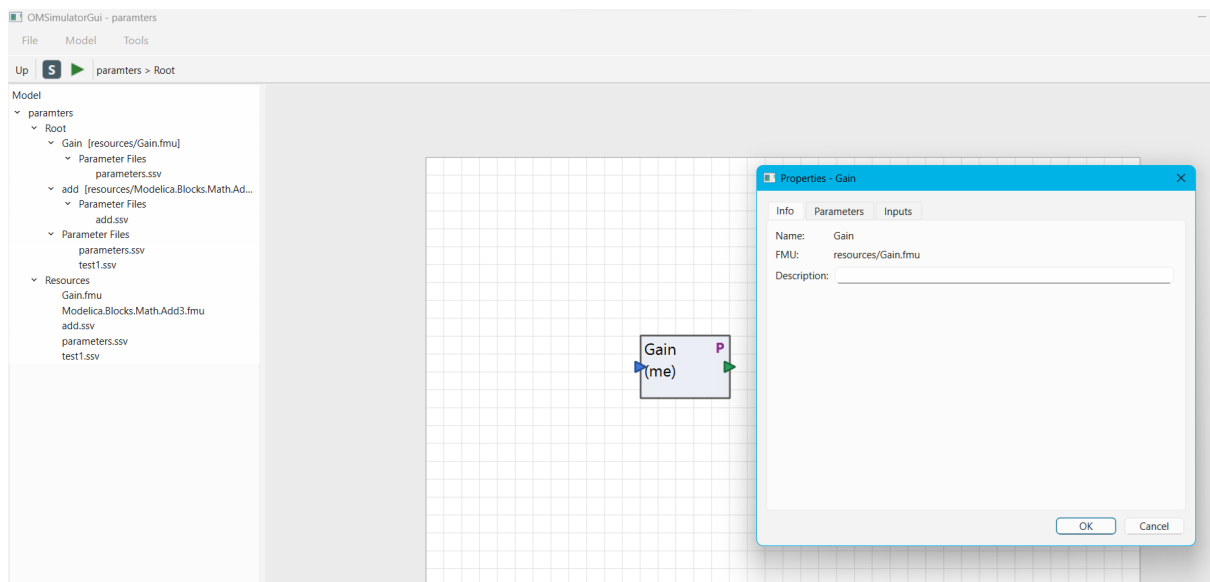


Fig. 23: OMSimulatorGui: Properties dialog for the “Gain” component, Info tab – the Parameters and Inputs tabs sit alongside it.

- browse to an existing .ssv/.ssm file on disk, or
- pick a resource **already registered** in the SSP (see Resources below) instead of creating or browsing a new one.

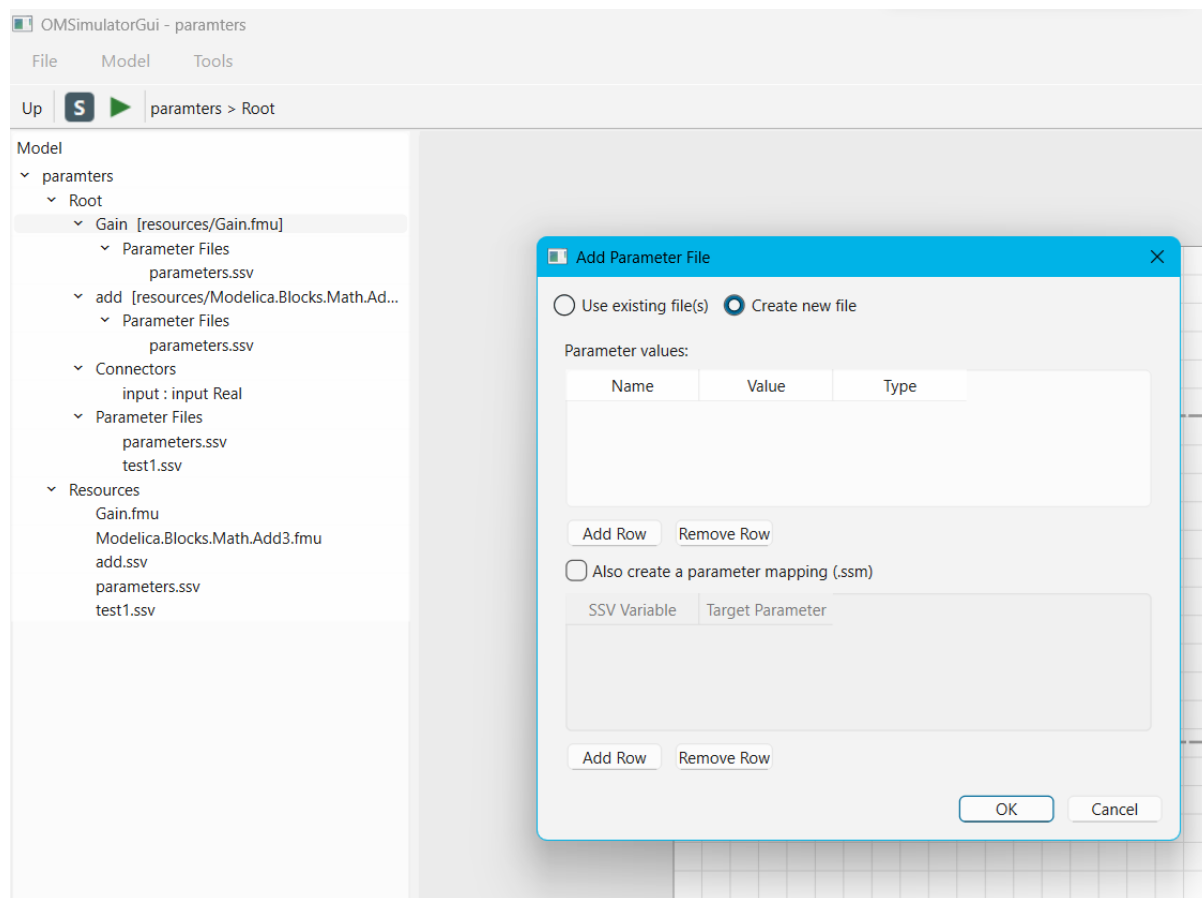


Fig. 24: OMSimulatorGui: Add Parameter File dialog in “Create new file” mode – a Name/Value/Type table for the SSV, plus an optional “Also create a parameter mapping (.ssm)” table underneath.

Edit Values... (double-click a Parameter File row, or its badge on the canvas) reopens the attached SSV’s own values for editing, and can also attach or re-point the companion SSM the same way Add Parameter File does (create new / browse / pick an existing resource).

Swap Resource... re-points an already-attached SSV reference at a *different, already-registered* SSV resource, without touching its values – useful for switching between two alternative presets that were both added via Add Resource beforehand. Swapping drops any attached SSM mapping (a warning is shown first if one is present), since the mapping is specific to the file being swapped out.

Export SSV Template.../Export SSM Template... (on a System or Component row) writes a ready-made external file listing every parameter/connector target reachable under that row – an SSV template with each target’s *current* value, or an SSM template with every target listed with an empty `source=""` for hand-editing. Neither of these attaches anything to the model; they are a scaffold to hand-edit externally and bring back in later via Add Parameter File’s “pick an existing resource” option.

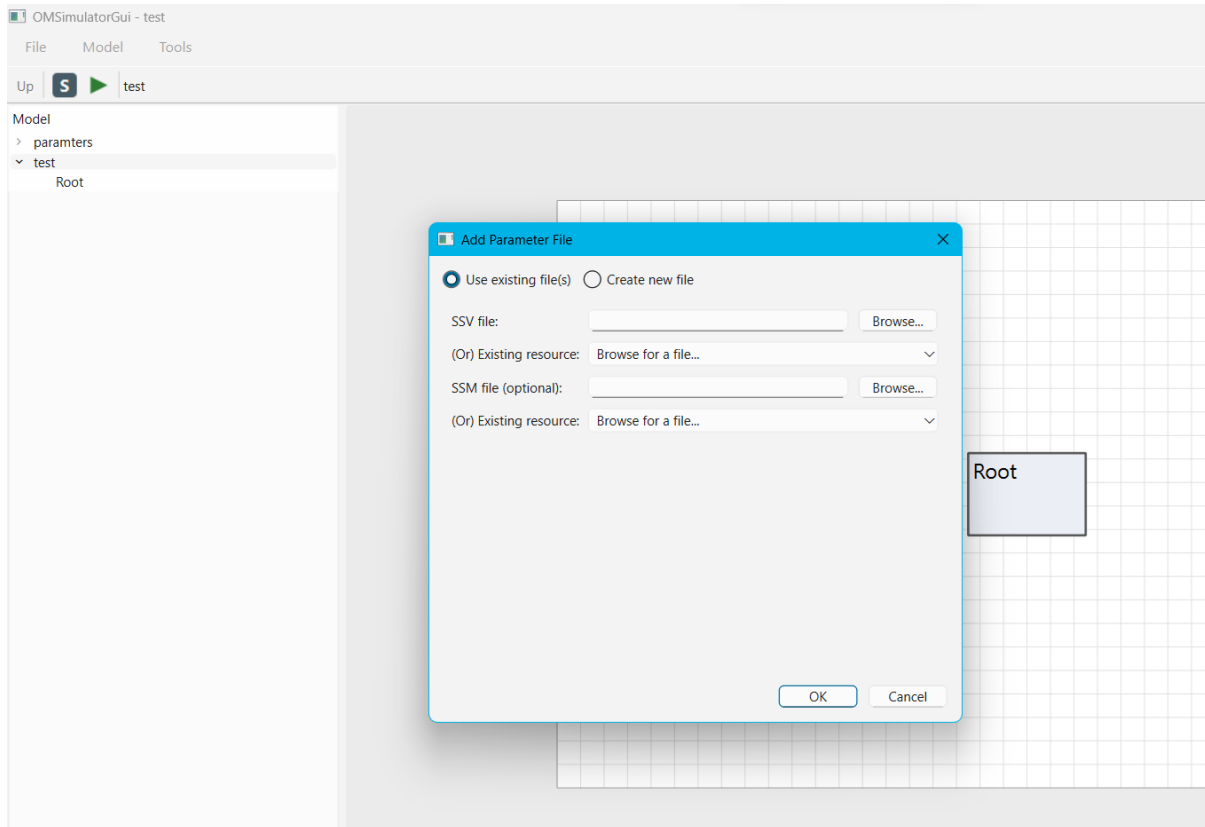


Fig. 25: OMSimulatorGui: Add Parameter File dialog in “Use existing file(s)” mode – an SSV file (with an “(Or) Existing resource” combo above it) and an optional SSM file, each browsable or pickable from the SSP’s own registered resources.

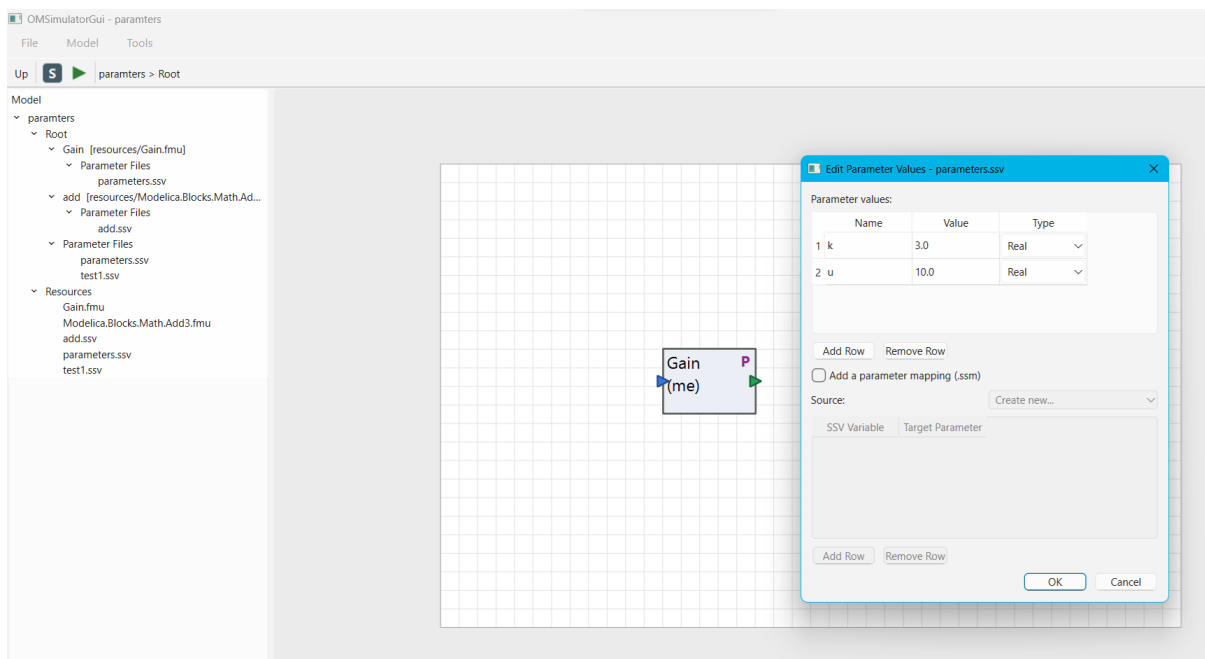


Fig. 26: OMSimulatorGui: tree with each system/component’s own “Parameter Files” group expanded, the “P” badge on the “Gain” box on the canvas, and the Edit Parameter Values dialog reopened on it – with the “Add a parameter mapping (.ssm)” checkbox and Source picker visible underneath the values table.

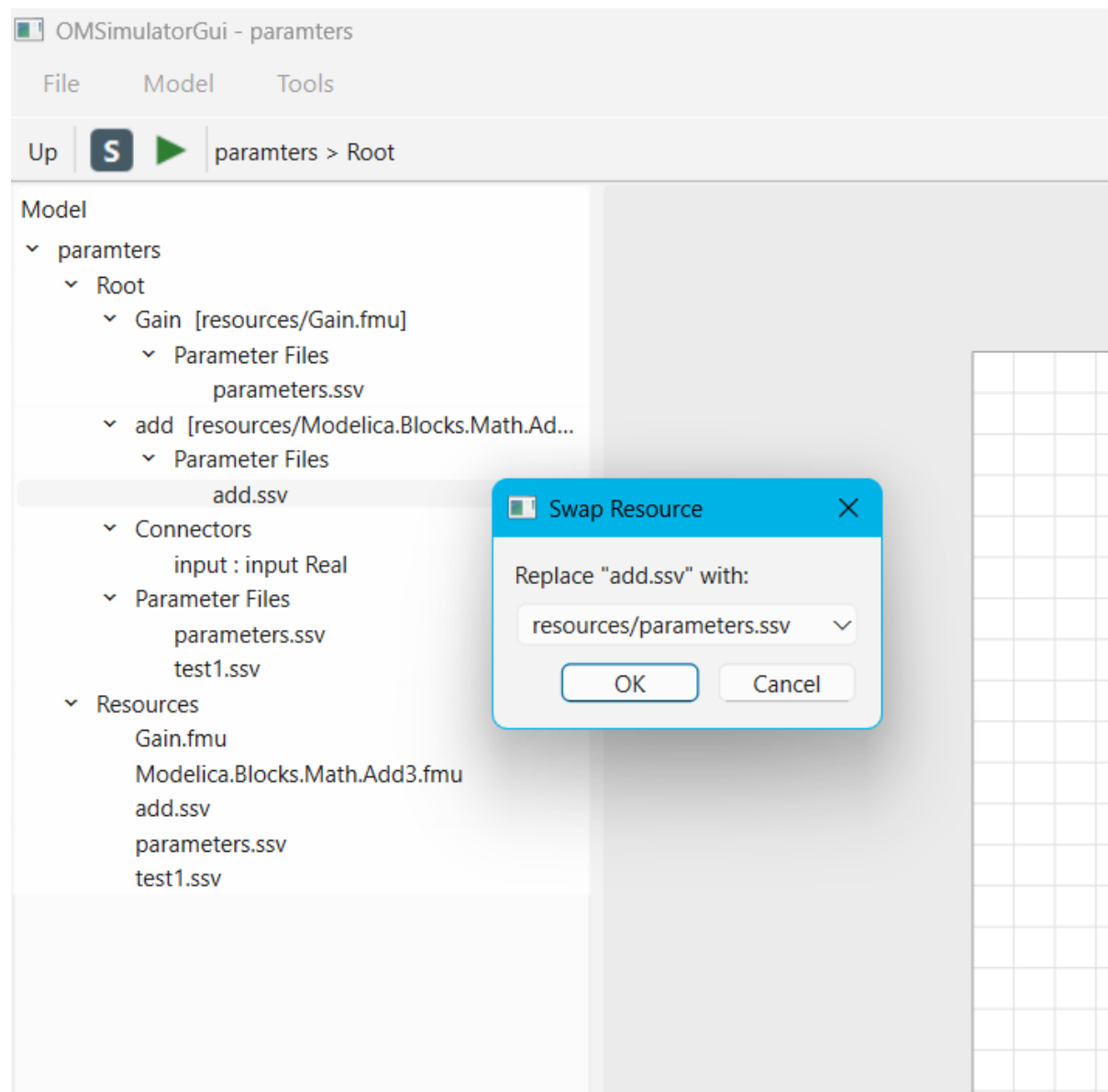


Fig. 27: OMSimulatorGui: Swap Resource picker, replacing “add.ssv” with another already-registered .ssv resource.

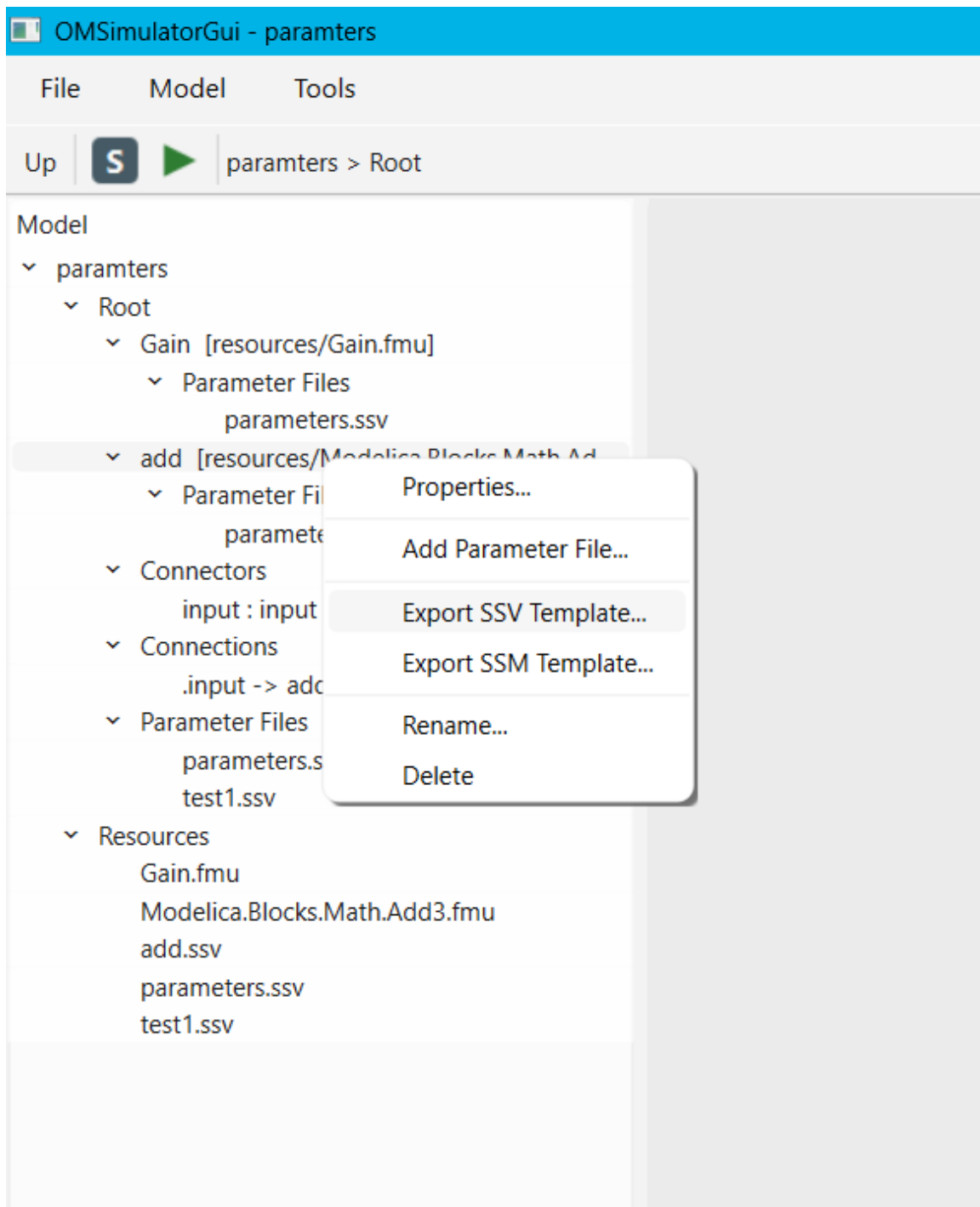


Fig. 28: OMSimulatorGui: Export SSV Template.../Export SSM Template... on a Component row's context menu. Picking either one opens a standard Save File dialog, pre-filled with a default name.

6.2.10 Resources

Every FMU, SSV/SSM file, or other resource referenced anywhere in the model is registered once in the SSP's shared resource pool and listed under the model's own **Resources** group in the tree.

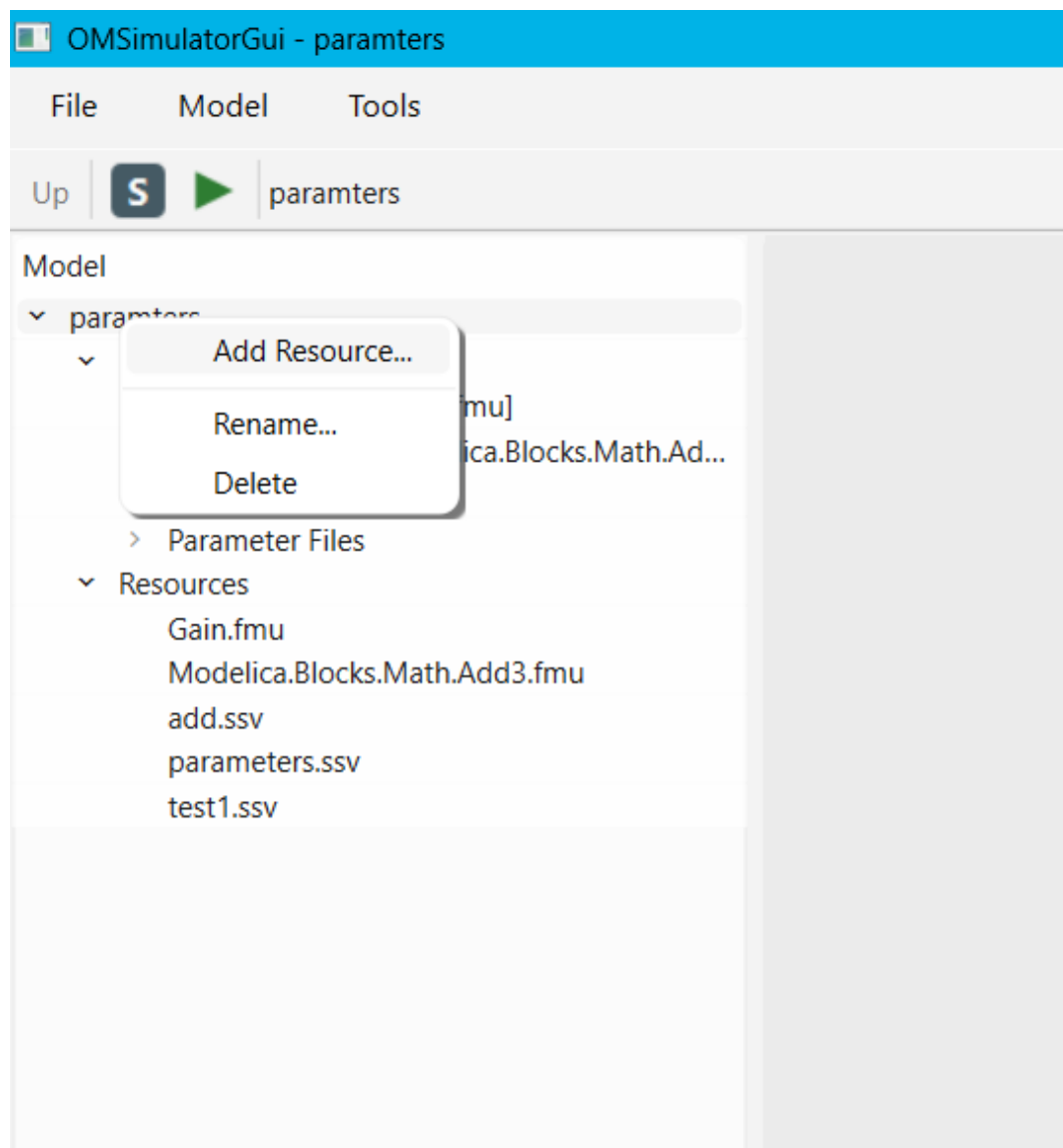


Fig. 29: OMSimulatorGui: model row's right-click menu (Add Resource.../Rename.../ Delete) and the Resources group listing every file registered in the SSP.

Add Resource... (on the model's own row) registers an arbitrary file without attaching it to anything – useful for staging a file (e.g. a lookup table for a component table) before wiring it up elsewhere, or for building up a pool of SSV/SSM presets ahead of time so Add Parameter File's/Swap Resource's "existing resource" pickers have something to offer. **Remove** unregisters a resource; it fails with an in-place error if something in the model still references it.

6.2.11 Variants

Model -> Variants... opens the SSP Variants dialog, listing every variant (alternate top-level system structure) stored in the model, letting the user activate a different one or duplicate the active one under a new name.

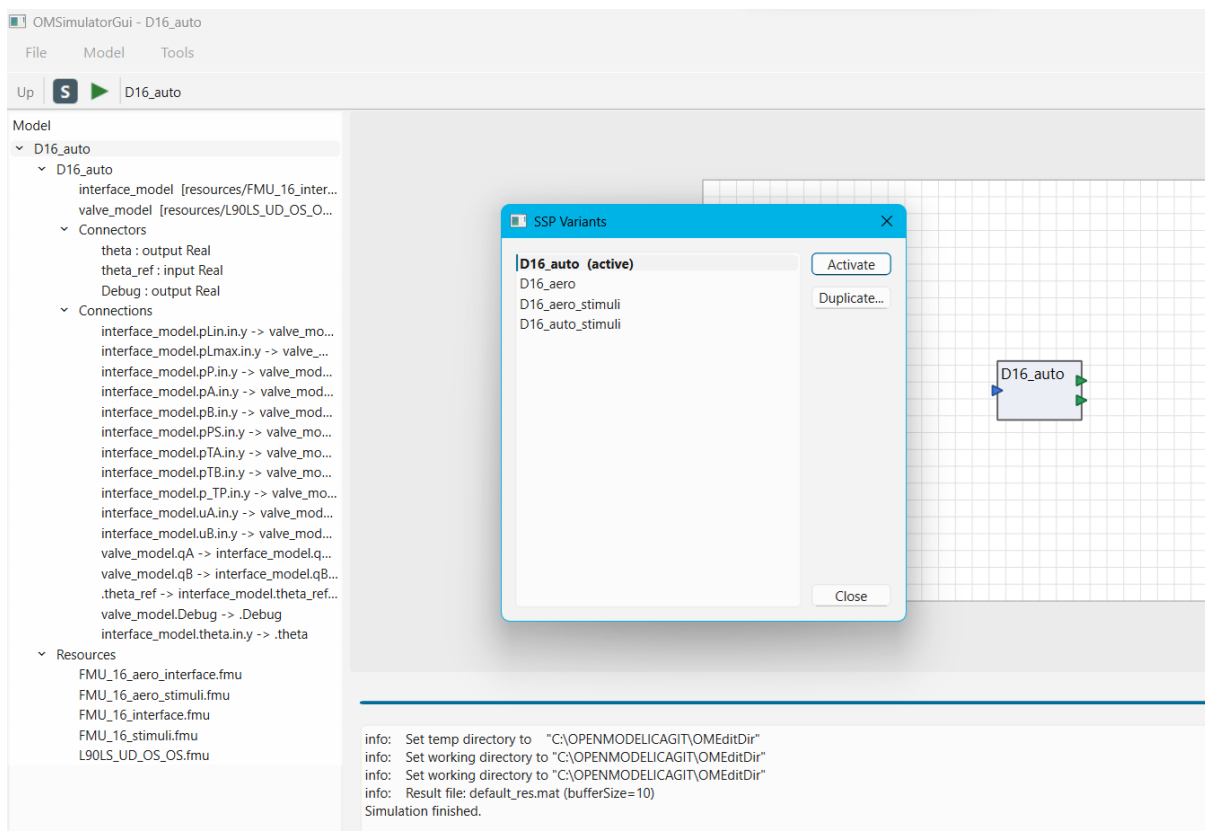


Fig. 30: OMSimulatorGui: SSP Variants dialog for a model with four variants – the active one marked “(active)”, with Activate and Duplicate... actions.

Activating a variant swaps the root system the tree and canvas are built from – the model’s own tree row always tracks whichever variant is currently active by name.

6.2.12 Simulation Settings

Model -> Simulation Settings... (or the toolbar’s “S” button) opens a per-model dialog with two tabs:

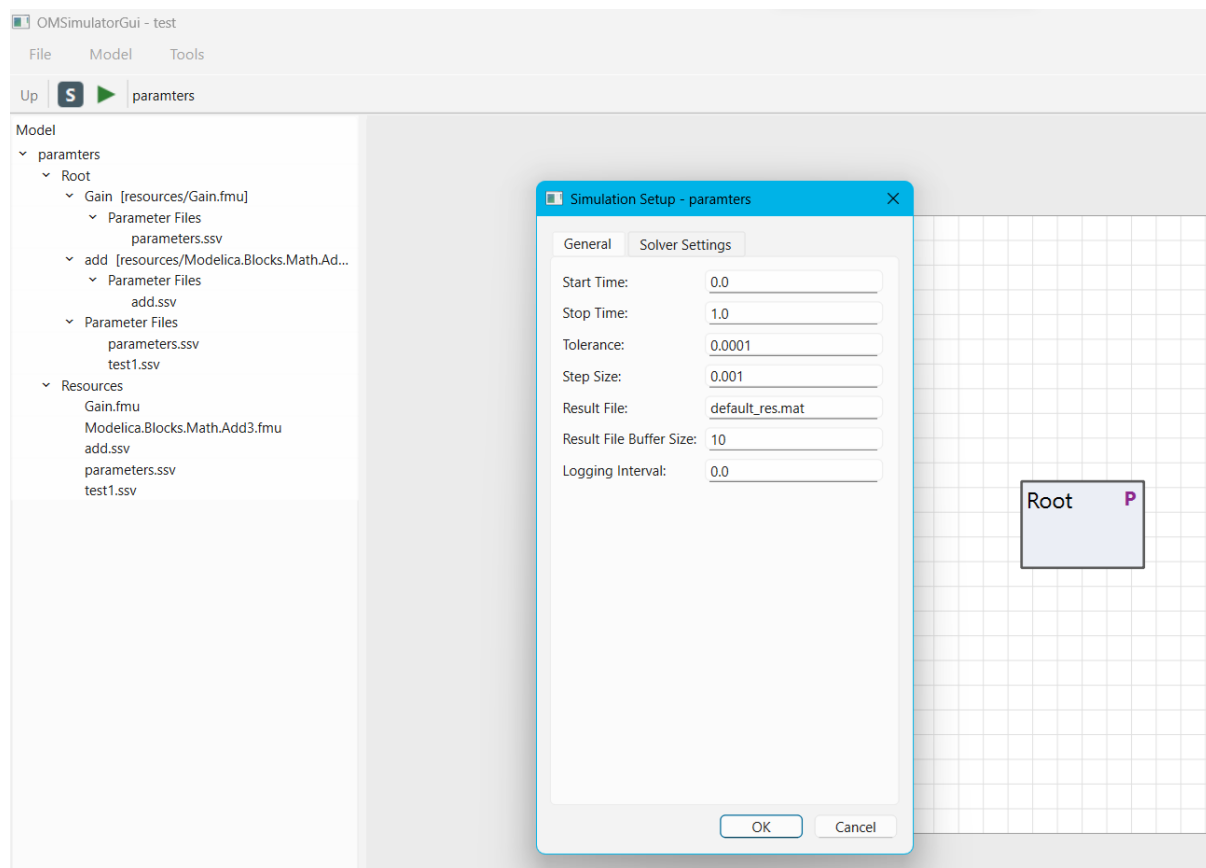


Fig. 31: OMSimulatorGui: Simulation Setup dialog, General tab.

- **General** – start time, stop time, tolerance, (maximum) step size, result file name, result file buffer size, and logging interval.
- **Solver Settings** – named solver configurations (a name plus a method, e.g. oms-ma, oms-mav, ccode, euler) and a per-component assignment table listing every FMU component in the model. A component left unassigned falls back to the automatic default for its FMI kind.

6.2.13 Options (Working Directory and Command Line Options)

Tools -> Options... holds two global preferences, applied to every simulation run regardless of which model is open:

- **Working Directory** – when set, every simulation exports the model into this fixed folder and runs there, so the result file lands somewhere predictable and gets overwritten on each re-run. Left blank (the default), each run instead uses a fresh temporary directory, same as before this setting existed.
- **Command Line Options** – free text, forwarded as one or more native flags before the model is instantiated – the same flags and the same underlying mechanism *OMSimulator’s own CLI* uses

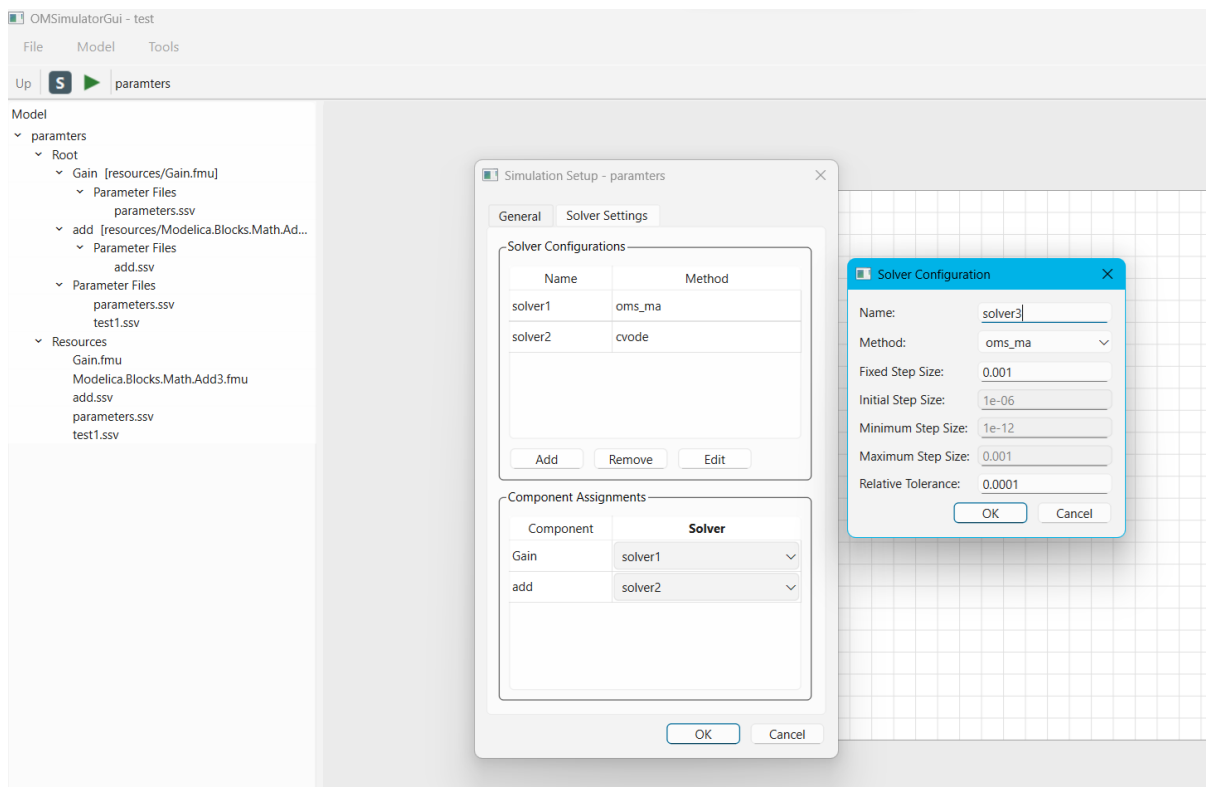


Fig. 32: OMSimulatorGui: Simulation Setup dialog, Solver Settings tab – two named solver configurations, each FMU component assigned to one – with the Solver Configuration dialog open on top for editing a configuration’s own numerical parameters (step sizes, relative tolerance).

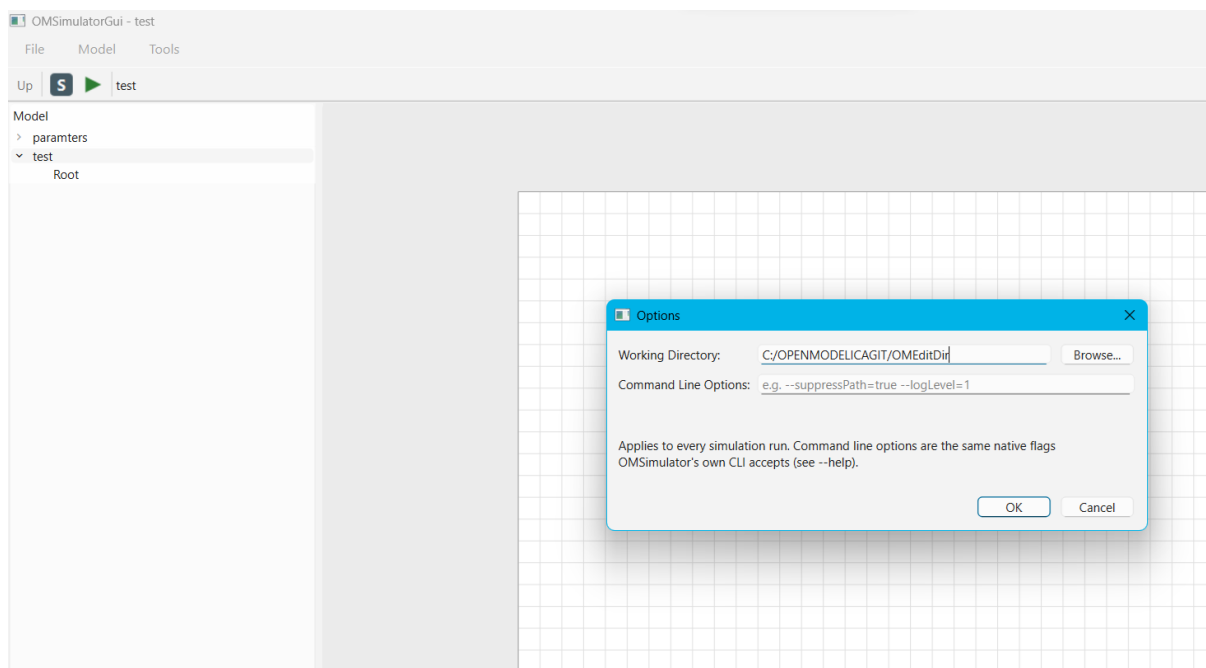


Fig. 33: OMSimulatorGui: Tools -> Options dialog, with a Working Directory set.

(e.g. `--suppressPath=true`, `--logLevel=1`). An unrecognized flag makes the simulation fail immediately with the native flag parser’s own error message.

6.2.14 Simulating and Viewing Results

Simulate (the toolbar’s green arrow, `Ctrl+R`, or `Model -> Simulate`) exports the active model’s current state and runs it in a separate subprocess – crash isolation from a misbehaving FMU never takes the GUI down with it. A progress panel appears at the bottom of the window with a progress bar and **Pause**, **Stop** and **Hide** buttons; pausing and resuming take effect immediately.

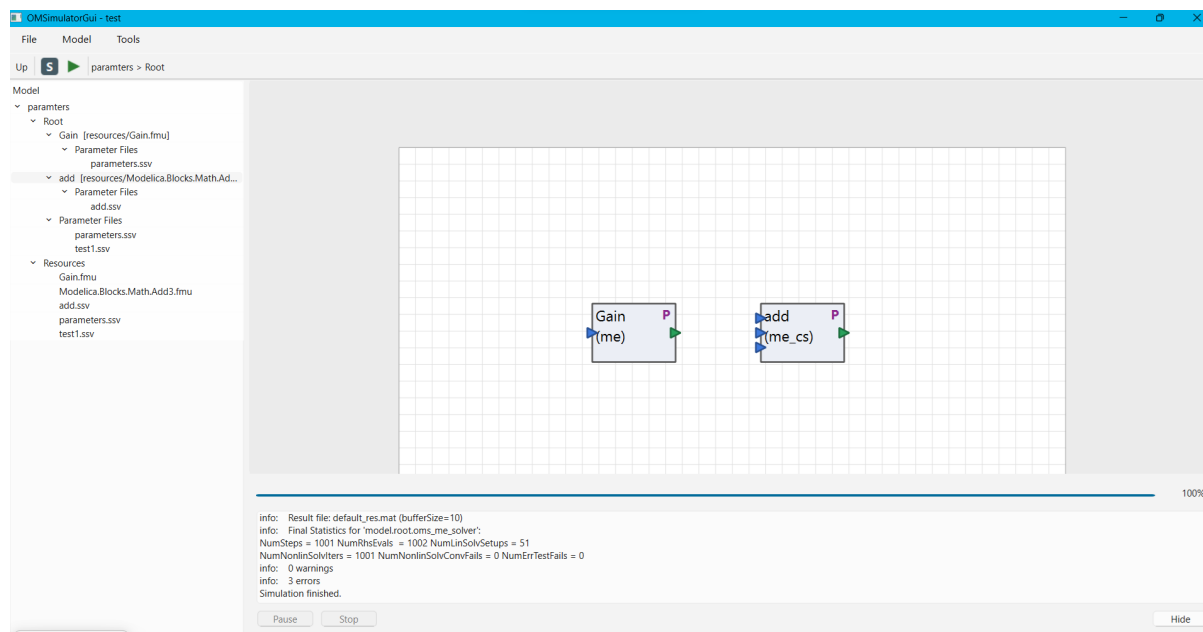


Fig. 34: OMSimulatorGui: simulation progress panel after a run has finished – progress at 100%, the solver’s own log output above it, and Pause/Stop disabled now that there is nothing left to control.

When the simulation finishes, its result file (.mat or .csv) opens automatically in a **Results** window: a checkable list of every signal on the left, plotted on the right as they are checked.

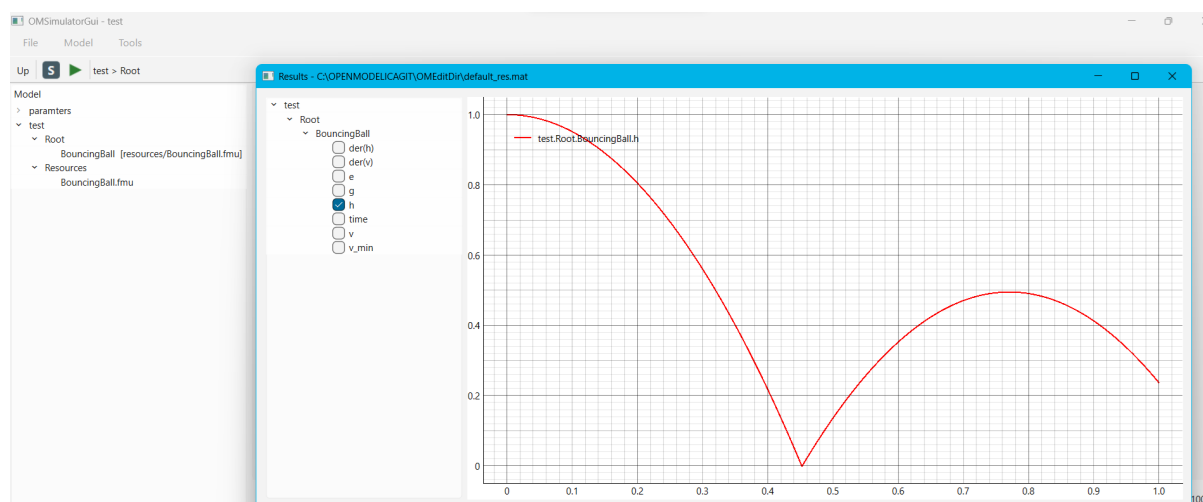


Fig. 35: OMSimulatorGui: Results window for a bouncing ball model – the “h” (height) signal checked in the tree on the left, plotted on the right.

6.2.15 Current Limitations

- No read-only XML/text viewer for the underlying SSD/SSV/SSM files yet (planned as a low-risk, self-contained addition).
- String-valued parameters/inputs are not editable (matches OMEdit's own current limitation).
- Only one variant's system structure is shown/edited at a time; switching variants resets the diagram canvas back to the model level.

G

Graphical Modelling, 70

O

OMEdit, 71

OMSimulator, 1

 Command Line Interface Examples, 26

 Examples, 6

 Flags, 3

OMSimulatorGui, 82

 Creating and Opening Models, 85

 Diagram Canvas, 87

 Element Properties, 88

 Launching, 83

 Main Window, 84

 Options, 96

 Parameter Files, 88

 pip, 83

 Requirements, 82

 Resources, 90

 Simulate, 98

 Simulation Settings, 96

 System Tree, 85

 Variants, 94

OMSimulatorLib, 7

OMSimulatorPython3, 24

 Command Line Interface, 25

OpenModelicaScripting, 57

 Examples, 59

 Scripting Commands, 59