

---

# OMSimulator Documentation

*Release v3.0.0.post106-g16bcf3f*

**Lennart Ochel**

**Oct 08, 2025**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>OMSimulator</b>	<b>3</b>
2.1	OMSimulator Flags . . . . .	3
2.2	Examples . . . . .	5
<b>3</b>	<b>OMSimulatorLib</b>	<b>7</b>
<b>4</b>	<b>C-API</b>	<b>9</b>
4.1	RunFile . . . . .	9
4.2	activateVariant . . . . .	9
4.3	addBus . . . . .	9
4.4	addConnection . . . . .	10
4.5	addConnector . . . . .	10
4.6	addConnectorToBus . . . . .	10
4.7	addResources . . . . .	10
4.8	addSignalsToResults . . . . .	10
4.9	addSubModel . . . . .	11
4.10	addSystem . . . . .	11
4.11	compareSimulationResults . . . . .	11
4.12	copySystem . . . . .	11
4.13	delete . . . . .	12
4.14	deleteConnection . . . . .	12
4.15	deleteConnectorFromBus . . . . .	12
4.16	deleteResources . . . . .	12
4.17	doStep . . . . .	13
4.18	duplicateVariant . . . . .	13
4.19	export . . . . .	13
4.20	exportDependencyGraphs . . . . .	14
4.21	exportSSMTemplate . . . . .	14
4.22	exportSSVTemplate . . . . .	14
4.23	exportSnapshot . . . . .	14
4.24	freeMemory . . . . .	15
4.25	getBoolean . . . . .	15
4.26	getBus . . . . .	15
4.27	getComponentType . . . . .	15
4.28	getConnections . . . . .	15
4.29	getConnector . . . . .	15
4.30	getDirectionalDerivative . . . . .	16

4.31	getElement . . . . .	16
4.32	getElements . . . . .	16
4.33	getFMUInfo . . . . .	16
4.34	getFixedStepSize . . . . .	16
4.35	getInteger . . . . .	16
4.36	getModelState . . . . .	17
4.37	getReal . . . . .	17
4.38	getResultFile . . . . .	17
4.39	getSolver . . . . .	17
4.40	getStartTime . . . . .	17
4.41	getStopTime . . . . .	17
4.42	getString . . . . .	18
4.43	getSubModelPath . . . . .	18
4.44	getSystemType . . . . .	18
4.45	getTime . . . . .	18
4.46	getTolerance . . . . .	18
4.47	getVariableStepSize . . . . .	18
4.48	getVersion . . . . .	19
4.49	importFile . . . . .	19
4.50	importSnapshot . . . . .	19
4.51	initialize . . . . .	19
4.52	instantiate . . . . .	19
4.53	list . . . . .	19
4.54	listUnconnectedConnectors . . . . .	20
4.55	listVariants . . . . .	20
4.56	loadSnapshot . . . . .	20
4.57	newModel . . . . .	21
4.58	newResources . . . . .	21
4.59	referenceResources . . . . .	21
4.60	removeSignalsFromResults . . . . .	22
4.61	rename . . . . .	22
4.62	replaceSubModel . . . . .	22
4.63	reset . . . . .	22
4.64	setActivationRatio . . . . .	23
4.65	setBoolean . . . . .	23
4.66	setBusGeometry . . . . .	23
4.67	setCommandLineOption . . . . .	23
4.68	setConnectionGeometry . . . . .	25
4.69	setConnectorGeometry . . . . .	25
4.70	setElementGeometry . . . . .	25
4.71	setFixedStepSize . . . . .	26
4.72	setInteger . . . . .	26
4.73	setLogFile . . . . .	26
4.74	setLoggingCallback . . . . .	26
4.75	setLoggingInterval . . . . .	26
4.76	setLoggingLevel . . . . .	26
4.77	setMaxLogFileSize . . . . .	27
4.78	setReal . . . . .	27
4.79	setRealInputDerivative . . . . .	27
4.80	setResultFile . . . . .	27
4.81	setSolver . . . . .	28

4.82	setStartTime	28
4.83	setStopTime	28
4.84	setString	28
4.85	setTempDirectory	28
4.86	setTolerance	28
4.87	setUnit	29
4.88	setVariableStepSize	29
4.89	setWorkingDirectory	29
4.90	simulate	29
4.91	simulate_realtime	29
4.92	stepUntil	30
4.93	terminate	30
<b>5</b>	<b>OMSimulatorLua</b>	<b>31</b>
5.1	Examples	31
5.1.1	Lua Scripting Commands	32
5.2	activateVariant	32
5.3	addBus	32
5.4	addConnection	32
5.5	addConnector	32
5.6	addConnectorToBus	33
5.7	addResources	33
5.8	addSignalsToResults	33
5.9	addSubModel	34
5.10	addSystem	34
5.11	compareSimulationResults	34
5.12	copySystem	34
5.13	delete	35
5.14	deleteConnection	35
5.15	deleteConnectorFromBus	35
5.16	deleteResources	35
5.17	duplicateVariant	36
5.18	export	36
5.19	exportDependencyGraphs	37
5.20	exportSSMTemplate	37
5.21	exportSSVTemplate	37
5.22	exportSnapshot	37
5.23	freeMemory	38
5.24	getBoolean	38
5.25	getDirectionalDerivative	38
5.26	getFixedStepSize	38
5.27	getInteger	38
5.28	getModelState	38
5.29	getReal	39
5.30	getSolver	39
5.31	getStartTime	39
5.32	getStopTime	39
5.33	getString	39
5.34	getSystemType	39
5.35	getTime	40
5.36	getTolerance	40

5.37	getVariableStepSize	40
5.38	getVersion	40
5.39	importFile	40
5.40	importSnapshot	40
5.41	initialize	41
5.42	instantiate	41
5.43	list	41
5.44	listUnconnectedConnectors	41
5.45	listVariants	41
5.46	loadSnapshot	42
5.47	newModel	42
5.48	newResources	42
5.49	referenceResources	43
5.50	removeSignalsFromResults	43
5.51	rename	44
5.52	replaceSubModel	44
5.53	reset	44
5.54	setActivationRatio	44
5.55	setBoolean	45
5.56	setCommandLineOption	45
5.57	setFixedStepSize	47
5.58	setInteger	47
5.59	setLogFile	47
5.60	setLoggingInterval	47
5.61	setLoggingLevel	47
5.62	setMaxLogFileSize	48
5.63	setReal	48
5.64	setRealInputDerivative	48
5.65	setResultFile	48
5.66	setSolver	49
5.67	setStartTime	49
5.68	setStopTime	49
5.69	setString	49
5.70	setTempDirectory	49
5.71	setTolerance	50
5.72	setUnit	50
5.73	setVariableStepSize	50
5.74	setWorkingDirectory	50
5.75	simulate	51
5.76	simulate_realtime	51
5.77	stepUntil	51
5.78	terminate	51
<b>6</b>	<b>OMSimulatorPython</b>	<b>53</b>
6.1	Examples	53
6.2	Python Scripting Commands	54
6.3	activateVariant	54
6.4	addBus	55
6.5	addConnection	55
6.6	addConnector	55
6.7	addConnectorToBus	56

6.8	addResources	56
6.9	addSignalsToResults	56
6.10	addSubModel	56
6.11	addSystem	57
6.12	compareSimulationResults	57
6.13	copySystem	57
6.14	delete	57
6.15	deleteConnection	58
6.16	deleteConnectorFromBus	58
6.17	deleteResources	58
6.18	doStep	59
6.19	duplicateVariant	59
6.20	export	59
6.21	exportDependencyGraphs	60
6.22	exportSSMTemplate	60
6.23	exportSSVTemplate	60
6.24	exportSnapshot	60
6.25	freeMemory	61
6.26	getBoolean	61
6.27	getDirectionalDerivative	61
6.28	getFixedStepSize	61
6.29	getInteger	61
6.30	getReal	61
6.31	getResultFile	62
6.32	getSolver	62
6.33	getStartTime	62
6.34	getStopTime	62
6.35	getString	62
6.36	getSubModelPath	62
6.37	getSystemType	63
6.38	getTime	63
6.39	getTolerance	63
6.40	getVariableStepSize	63
6.41	getVersion	63
6.42	importFile	63
6.43	importSnapshot	64
6.44	initialize	64
6.45	instantiate	64
6.46	list	64
6.47	listUnconnectedConnectors	64
6.48	listVariants	65
6.49	loadSnapshot	65
6.50	newModel	65
6.51	newResources	65
6.52	referenceResources	66
6.53	removeSignalsFromResults	67
6.54	rename	67
6.55	replaceSubModel	67
6.56	reset	68
6.57	setBoolean	68
6.58	setCommandLineOption	68

6.59	setFixedStepSize . . . . .	70
6.60	setInteger . . . . .	70
6.61	setLogFile . . . . .	70
6.62	setLoggingInterval . . . . .	70
6.63	setLoggingLevel . . . . .	71
6.64	setMaxLogFileSize . . . . .	71
6.65	setReal . . . . .	71
6.66	setRealInputDerivative . . . . .	71
6.67	setResultFile . . . . .	72
6.68	setSolver . . . . .	72
6.69	setStartTime . . . . .	72
6.70	setStopTime . . . . .	72
6.71	setString . . . . .	72
6.72	setTempDirectory . . . . .	73
6.73	setTolerance . . . . .	73
6.74	setUnit . . . . .	73
6.75	setVariableStepSize . . . . .	73
6.76	setWorkingDirectory . . . . .	74
6.77	simulate . . . . .	74
6.78	stepUntil . . . . .	74
6.79	terminate . . . . .	74
6.79.1	Example: Pi . . . . .	74
<b>7</b>	<b>OMSimulatorPython3</b>	<b>77</b>
7.1	Core Capabilities . . . . .	77
7.2	Quick start Example . . . . .	78
7.3	SSP . . . . .	79
7.4	SSD . . . . .	80
7.5	SSV . . . . .	80
7.6	SSM . . . . .	81
7.7	FMU . . . . .	82
7.8	Component . . . . .	83
7.9	addSystem . . . . .	84
7.10	addConnector . . . . .	85
7.11	addConnection . . . . .	85
7.12	addResource . . . . .	86
7.13	addComponent . . . . .	87
7.14	addSSVReference . . . . .	88
7.15	removeSSVReference . . . . .	89
7.16	swapSSVReference . . . . .	90
7.17	listSSVReference . . . . .	91
7.18	exportSSVTemplate . . . . .	92
7.19	exportSSVTemplate . . . . .	93
7.20	setValue . . . . .	94
7.21	getValue . . . . .	95
7.22	mapParameter . . . . .	96
7.23	duplicate and activate Variant . . . . .	98
7.24	getVariant . . . . .	99
7.25	listResource . . . . .	100
7.26	deleteResource . . . . .	100
7.27	delete . . . . .	101



7.28	instantiate . . . . .	102
7.29	setResultFile . . . . .	103
7.30	initialize . . . . .	104
7.31	simulate . . . . .	105
7.32	terminate and delete . . . . .	105
<b>8</b>	<b>OpenModelicaScripting</b>	<b>107</b>
8.1	Examples . . . . .	107
8.2	OpenModelica Scripting Commands . . . . .	108
8.3	addBus . . . . .	108
8.4	addConnection . . . . .	108
8.5	addConnector . . . . .	108
8.6	addConnectorToBus . . . . .	109
8.7	addSignalsToResults . . . . .	109
8.8	addSubModel . . . . .	109
8.9	addSystem . . . . .	109
8.10	compareSimulationResults . . . . .	109
8.11	copySystem . . . . .	110
8.12	delete . . . . .	110
8.13	deleteConnection . . . . .	110
8.14	deleteConnectorFromBus . . . . .	110
8.15	export . . . . .	111
8.16	exportDependencyGraphs . . . . .	111
8.17	exportSnapshot . . . . .	111
8.18	freeMemory . . . . .	111
8.19	getBoolean . . . . .	111
8.20	getFixedStepSize . . . . .	111
8.21	getInteger . . . . .	112
8.22	getModelState . . . . .	112
8.23	getReal . . . . .	112
8.24	getSolver . . . . .	112
8.25	getStartTime . . . . .	112
8.26	getStopTime . . . . .	112
8.27	getSubModelPath . . . . .	113
8.28	getSystemType . . . . .	113
8.29	getTime . . . . .	113
8.30	getTolerance . . . . .	113
8.31	getVariableStepSize . . . . .	113
8.32	getVersion . . . . .	113
8.33	importFile . . . . .	114
8.34	importSnapshot . . . . .	114
8.35	initialize . . . . .	114
8.36	instantiate . . . . .	114
8.37	list . . . . .	114
8.38	listUnconnectedConnectors . . . . .	114
8.39	loadSnapshot . . . . .	115
8.40	newModel . . . . .	115
8.41	removeSignalsFromResults . . . . .	115
8.42	rename . . . . .	115
8.43	reset . . . . .	115
8.44	setBoolean . . . . .	116

8.45	setCommandLineOption . . . . .	116
8.46	setFixedStepSize . . . . .	118
8.47	setInteger . . . . .	118
8.48	setLogFile . . . . .	118
8.49	setLoggingInterval . . . . .	118
8.50	setLoggingLevel . . . . .	118
8.51	setReal . . . . .	119
8.52	setRealInputDerivative . . . . .	119
8.53	setResultFile . . . . .	119
8.54	setSolver . . . . .	119
8.55	setStartTime . . . . .	120
8.56	setStopTime . . . . .	120
8.57	setTempDirectory . . . . .	120
8.58	setTolerance . . . . .	120
8.59	setVariableStepSize . . . . .	121
8.60	setWorkingDirectory . . . . .	121
8.61	simulate . . . . .	121
8.62	stepUntil . . . . .	121
8.63	terminate . . . . .	121
<b>9</b>	<b>Graphical Modelling</b>	<b>123</b>
9.1	New SSP Model . . . . .	123
9.2	Add System . . . . .	124
9.3	Add SubModel . . . . .	126
9.4	Simulate . . . . .	126
9.5	Dual Mass Oscillator Example . . . . .	127
<b>10</b>	<b>SSP Support</b>	<b>129</b>
10.1	Bus Connections . . . . .	129
	<b>Index</b>	<b>131</b>

## 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* and *Lua*.



## OMSIMULATOR

OMSimulator is a command line wrapper for the OMSimulatorLib library.

### 2.1 OMSimulator Flags

A brief description of all command line flags will be displayed using `OMSimulator --help`:

```
info:  Usage: OMSimulator [Options] [Lua script] [FMU] [SSP file]
       Options:
         --addParametersToCSV=<bool>      false      Export ↵
↵ parameters to a .csv file
         --algLoopSolver=<arg>            "kinsol"    Specifies the ↵
↵ loop solver method (fixedpoint, kinsol) used for algebraic loops spanning ↵
↵ multiple components.
         --clearAllOptions                Reset all ↵
↵ flags to their default values
         --CVODEMaxErrTestFails=<int>     100         Maximum number ↵
↵ of error test failures for CVODE
         --CVODEMaxNLSFailures=<int>      100         Maximum number ↵
↵ of nonlinear convergence failures for CVODE
         --CVODEMaxNLSIterations=<int>    5          Maximum number ↵
↵ of nonlinear solver iterations for CVODE
         --CVODEMaxSteps=<int>            1000        Maximum number ↵
↵ of steps for CVODE
         --deleteTempFiles=<bool>         true        Delete ↵
↵ temporary files as soon as they are no longer needed
         --directionalDerivatives=<bool>  true        Use ↵
↵ directional derivatives to calculate the Jacobian for algebraic loops
         --dumpAlgLoops=<bool>            false       Dump ↵
↵ information for algebraic loops
         --emitEvents=<bool>              true        Emit events ↵
↵ during simulation
         --help [-h]                     Display the ↵
↵ help text
         --ignoreInitialUnknowns=<bool>   false       Ignore initial ↵
↵ unknowns from the modelDescription.xml
         --initialStepSize=<double>        1e-6       Specify the ↵
↵ initial step size
```

(continues on next page)

(continued from previous page)

<code>--inputExtrapolation=&lt;bool&gt;</code>	<code>false</code>	Enable input_
↪ extrapolation using derivative information		
<code>--intervals=&lt;int&gt; [-i]</code>	<code>500</code>	Specify the_
↪ number of communication points (arg > 1)		
<code>--logFile=&lt;arg&gt; [-l]</code>	<code>""</code>	Specify the_
↪ log file (stdout is used <b>if</b> no log file is specified)		
<code>--logLevel=&lt;int&gt;</code>	<code>0</code>	Set the log_
↪ level (0: default, 1: debug, 2: debug+trace)		
<code>--master=&lt;arg&gt;</code>	<code>"ma"</code>	Specify the_
↪ master algorithm (ma)		
<code>--maxEventIteration=&lt;int&gt;</code>	<code>100</code>	Specify the_
↪ maximum number of iterations <b>for</b> handling a single event		
<code>--maxLoopIteration=&lt;int&gt;</code>	<code>10</code>	Specify the_
↪ maximum number of iterations <b>for</b> solving algebraic loops between system-		
↪ level components. Internal algebraic loops of components are not affected.		
<code>--minimumStepSize=&lt;double&gt;</code>	<code>1e-12</code>	Specify the_
↪ minimum step size		
<code>--mode=&lt;arg&gt; [-m]</code>	<code>"me"</code>	Force a_
↪ certain FMI mode <b>if</b> the FMU provides both cs and me (cs, me)		
<code>--numProcs=&lt;int&gt; [-n]</code>	<code>1</code>	Specify the_
↪ maximum number of processors to use (0=auto, 1=default)		
<code>--progressBar=&lt;bool&gt;</code>	<code>false</code>	Show a_
↪ progress bar <b>for</b> the simulation progress <b>in</b> the terminal		
<code>--realTime=&lt;bool&gt;</code>	<code>false</code>	Enable_
↪ experimental feature <b>for</b> (soft) real-time co-simulation		
<code>--resultFile=&lt;arg&gt; [-r]</code>	<code>"&lt;default&gt;"</code>	Specify the_
↪ name of the output result file		
<code>--skipCSVHeader=&lt;bool&gt;</code>	<code>true</code>	Skip exporting_
↪ the CSV delimiter <b>in</b> the header		
<code>--solver=&lt;arg&gt;</code>	<code>"cvmode"</code>	Specify the_
↪ integration method (euler, cvmode)		
<code>--solverStats=&lt;bool&gt;</code>	<code>false</code>	Add solver_
↪ stats to the result file, e.g., step size; not supported <b>for</b> all solvers		
<code>--startTime=&lt;double&gt; [-s]</code>	<code>0</code>	Specify the_
↪ start <b>time</b>		
<code>--stepSize=&lt;double&gt;</code>	<code>1e-3</code>	Specify the_
↪ (maximum) step size		
<code>--stopTime=&lt;double&gt; [-t]</code>	<code>1</code>	Specify the_
↪ stop <b>time</b>		
<code>--stripRoot=&lt;bool&gt;</code>	<code>false</code>	Remove the_
↪ root system prefix from all exported signals		
<code>--suppressPath=&lt;bool&gt;</code>	<code>false</code>	Suppress path_
↪ information <b>in</b> info messages; especially useful <b>for</b> testing		
<code>--tempDir=&lt;arg&gt;</code>	<code>""</code>	Specify the_
↪ temporary directory		
<code>--timeout=&lt;int&gt;</code>	<code>0</code>	Specify the_
↪ maximum allowed <b>time</b> <b>in</b> seconds <b>for</b> running a simulation (0 disables)		
<code>--tolerance=&lt;double&gt;</code>	<code>1e-4</code>	Specify the_
↪ relative tolerance		

(continues on next page)

(continued from previous page)

<code>--version [-v]</code>		Display <code>␣</code>
<code>↪version information</code>		
<code>--wallTime=&lt;bool&gt;</code>	<code>false</code>	Add wall <code>time</code> <code>␣</code>
<code>↪information to the result file</code>		
<code>--workingDir=&lt;arg&gt;</code>	<code>"."</code>	Specify the <code>␣</code>
<code>↪working directory</code>		
<code>--zeroNominal=&lt;bool&gt;</code>	<code>false</code>	Accept FMUs <code>␣</code>
<code>↪with invalid nominal values and replace the invalid nominal values with</code>	<code>1.0</code>	

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

## 2.2 Examples

```
OMSimulator --timeout 180 example.lua
```





## **OMSIMULATORLIB**

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



## 4.1 RunFile

Simulates a single FMU or SSP model.

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

## 4.2 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);
```

An example of activating the number of available variants in a ssp file

```
oms_newModel("model")      oms_addSystem("model.root",      "system_wc")
oms_addSubModel("model.root.A", "A.fmu") oms_duplicateVariant("model", "varA") //
varA will be the current variant oms_duplicateVariant("varA", "varB") // varB will be the
current variant oms_activateVariant("varB", "varA") // Reactivate the variant varB to varA
oms_activateVariant("varA", "model") // Reactivate the variant varA to model
```

## 4.3 addBus

Adds a bus to a given component.

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

## 4.4 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.

## 4.5 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);
```

## 4.6 addConnectorToBus

Adds a connector to a bus.

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

## 4.7 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)
```

## 4.8 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.

## 4.9 addSubModel

Adds a component to a system.

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

## 4.10 addSystem

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

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

## 4.11 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

## 4.12 copySystem

Copies a system.

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

## 4.13 delete

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

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

## 4.14 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.

## 4.15 deleteConnectorFromBus

Deletes a connector from a given bus.

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

## 4.16 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);
```

## 4.17 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_enumeration_t oms_doStep(const char* cref);
```

## 4.18 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_enumeration_t oms_duplicateVariant(const char* crefA, const char* crefB);
```

An example of creating a multi-variant modelling is presented below

```
oms_newModel("model")
oms_addSystem("model.root", "system_wc")
oms_addSubModel("model.root.A", "A.fmu")
oms_setReal("model.root.A.param1", "10")
oms_duplicateVariant("model", "varB")
oms_addSubModel("varB.root.B", "B.fmu")
oms_setReal("varB.root.A.param2", "20")
oms_export("varB", "variant.ssp")
```

The variant.ssp file will have the following structure

```
Variant.ssp
  SystemStructure.ssd
  varB.ssd
  resources\
    A.fmu
    B.fmu
```

## 4.19 export

Exports a composite model to a SPP file.

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

## 4.20 exportDependencyGraphs

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

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

## 4.21 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_enumeration_t oms_exportSSMTemplate(const char* cref, const char* filename)
```

## 4.22 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_enumeration_t oms_exportSSVTemplate(const char* cref, const char* filename)
```

## 4.23 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_enumeration_t oms_exportSnapshot(const char* cref, char** contents);
```



## 4.24 freeMemory

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

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

## 4.25 getBoolean

Get boolean value of given signal.

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

## 4.26 getBus

Gets the bus object.

```
oms_status_enu_t oms_getBus(const char* cref, oms_busconnector_t**  
↳ busConnector);
```

## 4.27 getComponentType

Gets the type of the given component.

```
oms_status_enu_t oms_getComponentType(const char* cref, oms_component_enu_t*  
↳ type);
```

## 4.28 getConnections

Get list of all connections from a given component.

```
oms_status_enu_t oms_getConnections(const char* cref, oms_connection_t***  
↳ connections);
```

## 4.29 getConnector

Gets the connector object of the given connector cref.

```
oms_status_enu_t oms_getConnector(const char* cref, oms_connector_t**  
↳ connector);
```

## 4.30 getDirectionalDerivative

This function computes the directional derivatives of an FMU.

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

## 4.31 getElement

Get element information of a given component reference.

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

## 4.32 getElements

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

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

## 4.33 getFMUInfo

Returns FMU specific information.

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

## 4.34 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_enu_t oms_getFixedStepSize(const char* cref, double* stepSize);
```

## 4.35 getInteger

Get integer value of given signal.

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

## 4.36 getModelState

Gets the model state of the given model cref.

```
oms_status_enu_t oms_getModelState(const char* cref, oms_modelState_enu_t*  
↪modelState);
```

## 4.37 getReal

Get real value.

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

## 4.38 getResultFile

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

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

## 4.39 getSolver

Gets the selected solver method of the given system.

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

## 4.40 getStartTime

Get the start time from the model.

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

## 4.41 getStopTime

Get the stop time from the model.

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

## 4.42 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);
```

## 4.43 getSubModelPath

Returns the path of a given component.

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

## 4.44 getSystemType

Gets the type of the given system.

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

## 4.45 getTime

Get the current simulation time from the model.

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

## 4.46 getTolerance

Gets the tolerance of a given system or component.

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

## 4.47 getVariableStepSize

Gets the step size parameters.

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

## 4.48 getVersion

Returns the library's version string.

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

## 4.49 importFile

Imports a composite model from a SSP file.

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

## 4.50 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);
```

## 4.51 initialize

Initializes a composite model.

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

## 4.52 instantiate

Instantiates a given composite model.

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

## 4.53 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);
```

## 4.54 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_enumer_t oms_listUnconnectedConnectors(const char* cref, char**  
↳contents);
```

## 4.55 listVariants

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

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

An example for finding the number of available variants in a ssp file

```
oms_newModel("model")  
oms_addSystem("model.root", "system_wc")  
oms_addSubModel("model.root.A", "A.fmu")  
oms_duplicateVariant("model", "varA")  
oms_duplicateVariant("varA", "varB")  
  
oms_listVariants("varB")
```

The API will list the available variants like below

```
<oms:Variants>  
  <oms:variant name="model" />  
  <oms:variant name="varB" />  
  <oms:variant name="varA" />  
</oms:Variants>
```

## 4.56 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_enumer_t oms_loadSnapshot(const char* cref, const char* snapshot,   
↳char** newCref);
```

## 4.57 newModel

Creates a new and yet empty composite model.

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

## 4.58 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)
```

## 4.59 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” seperately as the ssm file is binded to a ssv file. Hence it is not possible to switch the reference of “.ssm” file alone. So inorder 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);
```

## 4.60 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.

## 4.61 rename

Renames a model, system, or component.

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

## 4.62 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);
```

It is possible to import an partially developed fmu (i.e contains only modeldescription.xml without any binaries) in OMSimulator, and later can be replaced with a fully developed fmu. An example to use the API, oms\_addSubModel(“model.root.A”, “../resources/replaceA.fmu”) oms\_export(“model”, “test.ssp”) oms\_import(“test.ssp”) oms\_replaceSubModel(“model.root.A”, “../resources/replaceA\_extended.fmu”)

## 4.63 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);
```



## 4.64 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);
```

## 4.65 setBoolean

Sets the value of a given boolean signal.

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

## 4.66 setBusGeometry

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

## 4.67 setCommandLineOption

Sets special flags.

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

Available flags:

```
info:      Usage: OMSimulator [Options] [Lua script] [FMU] [SSP file]
           Options:
             --addParametersToCSV=<bool>      false      Export_
↳ parameters to a .csv file
             --algLoopSolver=<arg>            "kinsol"    Specifies the_
↳ loop solver method (fixedpoint, kinsol) used for algebraic loops spanning_
↳ multiple components.
             --clearAllOptions                  Reset all_
↳ flags to their default values
             --CVODEMaxErrTestFails=<int>      100        Maximum number_
↳ of error test failures for CVODE
             --CVODEMaxNLSFailures=<int>      100        Maximum number_
↳ of nonlinear convergence failures for CVODE
             --CVODEMaxNLSIterations=<int>     5         Maximum number_
↳ of nonlinear solver iterations for CVODE
             --CVODEMaxSteps=<int>            1000       Maximum number_
↳ of steps for CVODE
             --deleteTempFiles=<bool>         true       Delete_
↳ temporary files as soon as they are no longer needed
```

(continues on next page)

(continued from previous page)

<code>--directionalDerivatives=&lt;bool&gt;</code>	<code>true</code>	Use
↪directional derivatives to calculate the Jacobian <b>for</b> algebraic loops		
<code>--dumpAlgLoops=&lt;bool&gt;</code>	<code>false</code>	Dump
↪information <b>for</b> algebraic loops		
<code>--emitEvents=&lt;bool&gt;</code>	<code>true</code>	Emit events
↪during simulation		
<code>--help [-h]</code>		Display the
↪help text		
<code>--ignoreInitialUnknowns=&lt;bool&gt;</code>	<code>false</code>	Ignore initial
↪unknowns from the modelDescription.xml		
<code>--initialStepSize=&lt;double&gt;</code>	<code>1e-6</code>	Specify the
↪initial step size		
<code>--inputExtrapolation=&lt;bool&gt;</code>	<code>false</code>	Enable input
↪extrapolation using derivative information		
<code>--intervals=&lt;int&gt; [-i]</code>	<code>500</code>	Specify the
↪number of communication points (arg > 1)		
<code>--logFile=&lt;arg&gt; [-l]</code>	<code>""</code>	Specify the
↪log file (stdout is used <b>if</b> no log file is specified)		
<code>--logLevel=&lt;int&gt;</code>	<code>0</code>	Set the log
↪level (0: default, 1: debug, 2: debug+trace)		
<code>--master=&lt;arg&gt;</code>	<code>"ma"</code>	Specify the
↪master algorithm (ma)		
<code>--maxEventIteration=&lt;int&gt;</code>	<code>100</code>	Specify the
↪maximum number of iterations <b>for</b> handling a single event		
<code>--maxLoopIteration=&lt;int&gt;</code>	<code>10</code>	Specify the
↪maximum number of iterations <b>for</b> solving algebraic loops between system-		
↪level components. Internal algebraic loops of components are not affected.		
<code>--minimumStepSize=&lt;double&gt;</code>	<code>1e-12</code>	Specify the
↪minimum step size		
<code>--mode=&lt;arg&gt; [-m]</code>	<code>"me"</code>	Force a
↪certain FMI mode <b>if</b> the FMU provides both cs and me (cs, me)		
<code>--numProcs=&lt;int&gt; [-n]</code>	<code>1</code>	Specify the
↪maximum number of processors to use (0=auto, 1=default)		
<code>--progressBar=&lt;bool&gt;</code>	<code>false</code>	Show a
↪progress bar <b>for</b> the simulation progress <b>in</b> the terminal		
<code>--realTime=&lt;bool&gt;</code>	<code>false</code>	Enable
↪experimental feature <b>for</b> (soft) real-time co-simulation		
<code>--resultFile=&lt;arg&gt; [-r]</code>	<code>"&lt;default&gt;"</code>	Specify the
↪name of the output result file		
<code>--skipCSVHeader=&lt;bool&gt;</code>	<code>true</code>	Skip exporting
↪the CSV delimiter <b>in</b> the header		
<code>--solver=&lt;arg&gt;</code>	<code>"cvmode"</code>	Specify the
↪integration method (euler, cvmode)		
<code>--solverStats=&lt;bool&gt;</code>	<code>false</code>	Add solver
↪stats to the result file, e.g., step size; not supported <b>for</b> all solvers		
<code>--startTime=&lt;double&gt; [-s]</code>	<code>0</code>	Specify the
↪start time		
<code>--stepSize=&lt;double&gt;</code>	<code>1e-3</code>	Specify the
↪(maximum) step size		

(continues on next page)

(continued from previous page)

```

--stopTime=<double> [-t]          1          Specify the
↪ stop time
--stripRoot=<bool>                 false       Remove the
↪ root system prefix from all exported signals
--suppressPath=<bool>              false       Suppress path
↪ information in info messages; especially useful for testing
--tempDir=<arg>                    "."         Specify the
↪ temporary directory
--timeout=<int>                    0           Specify the
↪ maximum allowed time in seconds for running a simulation (0 disables)
--tolerance=<double>               1e-4        Specify the
↪ relative tolerance
--version [-v]                    Display
↪ version information
--wallTime=<bool>                  false       Add wall time
↪ information to the result file
--workingDir=<arg>                 "."         Specify the
↪ working directory
--zeroNominal=<bool>              false       Accept FMUs
↪ with invalid nominal values and replace the invalid nominal values with 1.0

```

## 4.68 setConnectionGeometry

```

oms_status_enumeration_t oms_setConnectionGeometry(const char* crefA, const char*
↪ crefB, const ssd_connection_geometry_t* geometry);

```

## 4.69 setConnectorGeometry

Set geometry information to a given connector.

```

oms_status_enumeration_t oms_setConnectorGeometry(const char* cref, const ssd_
↪ connector_geometry_t* geometry);

```

## 4.70 setElementGeometry

Set geometry information to a given component.

```

oms_status_enumeration_t oms_setElementGeometry(const char* cref, const ssd_element_
↪ geometry_t* geometry);

```

## 4.71 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_enu_t oms_setFixedStepSize(const char* cref, double stepSize);
```

## 4.72 setInteger

Sets the value of a given integer signal.

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

## 4.73 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_enu_t oms_setLogFile(const char* filename);
```

## 4.74 setLoggingCallback

Sets a callback function for the logging system.

```
void oms_setLoggingCallback(void (*cb)(oms_message_type_enu_t type, const_  
↳char* message));
```

## 4.75 setLoggingInterval

Set the logging interval of the simulation.

```
oms_status_enu_t oms_setLoggingInterval(const char* cref, double_  
↳loggingInterval);
```

## 4.76 setLoggingLevel

Enables/Disables debug logging (logDebug and logTrace).

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

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

## 4.77 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);
```

## 4.78 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.

## 4.79 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);
```

## 4.80 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.

## 4.81 setSolver

Sets the solver method for the given system.

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

## 4.82 setStartTime

Set the start time of the simulation.

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

## 4.83 setStopTime

Set the stop time of the simulation.

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

## 4.84 setString

Sets the value of a given string signal.

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

## 4.85 setTempDirectory

Set new temp directory.

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

## 4.86 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.

## 4.87 setUnit

Sets the unit of a given signal.

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

## 4.88 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);
```

## 4.89 setWorkingDirectory

Set a new working directory.

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

## 4.90 simulate

Simulates a composite model.

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

## 4.91 simulate\_realtime

Experimental feature for (soft) real-time simulation.

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

## 4.92 stepUntil

Simulates a composite model until a given time value.

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

## 4.93 terminate

Terminates a given composite model.

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



## OMSIMULATORLUA

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

### 5.1 Examples

```
oms_setTempDirectory("./temp/")
oms_newModel("model")
oms_addSystem("model.root", 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")
```

### 5.1.1 Lua Scripting Commands

## 5.2 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.

```
status = oms_activateVariant(crefA, crefB)
```

An example of activating the number of available variants in a ssp file

```
oms_newModel("model")          oms_addSystem("model.root",          "system_wc")
oms_addSubModel("model.root.A", "A.fmu") oms_duplicateVariant("model", "varA") //
varA will be the current variant oms_duplicateVariant("varA", "varB") // varB will be the
current variant oms_activateVariant("varB", "varA") // Reactivate the variant varB to varA
oms_activateVariant("varA", "model") // Reactivate the variant varA to model
```

## 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,

```
oms_causality_input
oms_causality_output
oms_causality_parameter
```

(continues on next page)

(continued from previous page)

```
oms_causality_bidir
oms_causality_undefined
```

The third argument `"type"`, should be any of the following,

```
oms_signal_type_real
oms_signal_type_integer
oms_signal_type_boolean
oms_signal_type_string
oms_signal_type_enum
oms_signal_type_bus
```

## 5.6 addConnectorToBus

Adds a connector to a bus.

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

## 5.7 addResources

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

```
status = oms_addResources(cref, path)

-- Example
oms_importFile("addExternalResources1.ssp")
-- add list of external resources from filesystem to ssp
oms_addResources("addExternalResources", "../resources/externalRoot.ssv")
oms_addResources("addExternalResources:externalSystem.ssv", "../resources/
↪externalSystem1.ssv")
oms_addResources("addExternalResources", "../resources/externalGain.ssv")
-- export the ssp with new resources
oms_export("addExternalResources", "addExternalResources1.ssp")
```

## 5.8 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.

## 5.9 addSubModel

Adds a component to a system.

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

## 5.10 addSystem

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

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

## 5.11 compareSimulationResults

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

```
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

## 5.12 copySystem

Copies a system.

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

## 5.13 delete

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

```
status = oms_delete(cref)
```

## 5.14 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.

## 5.15 deleteConnectorFromBus

Deletes a connector from a given bus.

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

## 5.16 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” seperately 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”

```
status = oms_deleteResources(cref)
```

```
-- Example
```

```
oms_importFile("deleteResources1.ssp")
```

```
-- delete only the references in ".ssd" file
```

```
oms_deleteResources("deleteResources.root:root.ssv")
```

```
-- delete both references and resources
```

(continues on next page)

(continued from previous page)

```
oms_deleteResources("deleteResources:root.ssv")
oms_export("deleteResources1.ssp")
```

## 5.17 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.

```
status = oms_duplicateVariant(crefA, crefB)
```

An example of creating a multi-variant modelling is presented below

```
oms_newModel("model")
oms_addSystem("model.root", "system_wc")
oms_addSubModel("model.root.A", "A.fmu")
oms_setReal("model.root.A.param1", "10")
oms_duplicateVariant("model", "varB")
oms_addSubModel("varB.root.B", "B.fmu")
oms_setReal("varB.root.A.param2", "20")
oms_export("varB", "variant.ssp")
```

The variant.ssp file will have the following structure

```
Variant.ssp
  SystemStructure.ssd
  varB.ssd
  resources\
    A.fmu
    B.fmu
```

## 5.18 export

Exports a composite model to a SPP file.

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

## 5.19 exportDependencyGraphs

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

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

## 5.20 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.

```
status = oms_exportSSMTemplate(cref, filename)
```

## 5.21 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.

```
status = oms_exportSSVTemplate(cref, filename)
```

## 5.22 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)
```

## 5.23 freeMemory

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

This function is neither needed nor available from the Lua interface.

## 5.24 getBoolean

Get boolean value of given signal.

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

## 5.25 getDirectionalDerivative

This function computes the directional derivatives of an FMU.

```
value, status = oms_getDirectionalDerivative(cref)
```

## 5.26 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_setFixedStepSize(cref)
```

## 5.27 getInteger

Get integer value of given signal.

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

## 5.28 getModelState

Gets the model state of the given model cref.

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



## 5.29 getReal

Get real value.

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

## 5.30 getSolver

Gets the selected solver method of the given system.

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

## 5.31 getStartTime

Get the start time from the model.

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

## 5.32 getStopTime

Get the stop time from the model.

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

## 5.33 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.

```
value, status = oms_getString(cref)
```

## 5.34 getSystemType

Gets the type of the given system.

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

### 5.35 getTime

Get the current simulation time from the model.

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

### 5.36 getTolerance

Gets the tolerance of a given system or component.

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

### 5.37 getVariableStepSize

Gets the step size parameters.

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

### 5.38 getVersion

Returns the library's version string.

```
version = oms_getVersion()
```

### 5.39 importFile

Imports a composite model from a SSP file.

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

### 5.40 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.

```
newCref, status = oms_importSnapshot(cref, snapshot)
```

## 5.41 initialize

Initializes a composite model.

```
status = oms_initialize(cref)
```

## 5.42 instantiate

Instantiates a given composite model.

```
status = oms_instantiate(cref)
```

## 5.43 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)
```

## 5.44 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)
```

## 5.45 listVariants

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

```
status = oms_listVariants(cref)
```

An example for finding the number of available variants in a ssp file

```
oms_newModel("model")
oms_addSystem("model.root", "system_wc")
oms_addSubModel("model.root.A", "A.fmu")
oms_duplicateVariant("model", "varA")
oms_duplicateVariant("varA", "varB")

oms_listVariants("varB")
```

The API will list the available variants like below

```
<oms:Variants>
  <oms:variant name="model" />
  <oms:variant name="varB" />
  <oms:variant name="varA" />
</oms:Variants>
```

## 5.46 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.

```
newCref, status = oms_loadSnapshot(cref, snapshot)
```

## 5.47 newModel

Creates a new and yet empty composite model.

```
status = oms_newModel(cref)
```

## 5.48 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”

```
status = oms_newResources(cref)

-- Example
oms_newModel("newResources")

oms_addSystem("newResources.root", oms_system_wc)
oms_addConnector("newResources.root.Input1", oms_causality_input, oms_signal_
↪type_real)
oms_addConnector("newResources.root.Input2", oms_causality_input, oms_signal_
↪type_real)

-- add Top level new resources, the filename is provided using the colon_
↪suffix ":root.ssv"
oms_newResources("newResources.root:root.ssv")
oms_setReal("newResources.root.Input1", 10)
-- export the ssp with new resources
oms_export("newResources", "newResources.ssp")
```

## 5.49 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” seperately as the ssm file is binded to a ssv file. Hence it is not possible to switch the reference of “.ssm” file alone. So inorder 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.

```
status = oms_referenceResources(cref, ssmFile)

-- Example
oms_importFile("referenceResources1.ssp")
-- delete only the references in ".ssd" file
oms_deleteResources("referenceResources1.root:root.ssv")
-- usage-1 switch with new references, only ssv file
oms_referenceResources("referenceResources1.root:Config1.ssv")
-- usage-2 switch with new references, both ssv and ssm file
oms_referenceResources("referenceResources1.root:Config1.ssv", "Config1.ssm")
oms_export("referenceResources1.ssp")
```

## 5.50 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.

## 5.51 rename

Renames a model, system, or component.

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

## 5.52 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.

```
status = oms_replaceSubModel(cref, fmuPath)
```

It is possible to import an partially developed fmu (i.e contains only modeldescription.xml without any binaries) in OMSimulator, and later can be replaced with a fully developed fmu. An example to use the API, `oms_addSubModel("model.root.A", "../resources/replaceA.fmu")` `oms_export("model", "test.ssp")` `oms_import("test.ssp")` `oms_replaceSubModel("model.root.A", "../resources/replaceA_extended.fmu")`

## 5.53 reset

Reset the composite model after a simulation run.

The FMUs go into the same state as after instantiation.

```
status = oms_reset(cref)
```

## 5.54 setActivationRatio

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

```
status = experimental_setActivationRatio(cref, k)
```

## 5.55 setBoolean

Sets the value of a given boolean signal.

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

## 5.56 setCommandLineOption

Sets special flags.

```
status = oms_setCommandLineOption(cmd)
```

Available flags:

```
info:      Usage: OMSimulator [Options] [Lua script] [FMU] [SSP file]
           Options:
             --addParametersToCSV=<bool>      false      Export
↪parameters to a .csv file
             --algLoopSolver=<arg>            "kinsol"     Specifies the
↪loop solver method (fixedpoint, kinsol) used for algebraic loops spanning
↪multiple components.
             --clearAllOptions                Reset all
↪flags to their default values
             --CVODEMaxErrTestFails=<int>     100          Maximum number
↪of error test failures for CVODE
             --CVODEMaxNLSFailures=<int>      100          Maximum number
↪of nonlinear convergence failures for CVODE
             --CVODEMaxNLSIterations=<int>    5           Maximum number
↪of nonlinear solver iterations for CVODE
             --CVODEMaxSteps=<int>            1000         Maximum number
↪of steps for CVODE
             --deleteTempFiles=<bool>         true         Delete
↪temporary files as soon as they are no longer needed
             --directionalDerivatives=<bool>   true         Use
↪directional derivatives to calculate the Jacobian for algebraic loops
             --dumpAlgLoops=<bool>            false        Dump
↪information for algebraic loops
             --emitEvents=<bool>              true         Emit events
↪during simulation
             --help [-h]                    Display the
↪help text
             --ignoreInitialUnknowns=<bool>   false        Ignore initial
↪unknowns from the modelDescription.xml
             --initialStepSize=<double>        1e-6         Specify the
↪initial step size
             --inputExtrapolation=<bool>       false        Enable input
↪extrapolation using derivative information
             --intervals=<int> [-i]           500          Specify the
↪number of communication points (arg > 1)
```

(continues on next page)

(continued from previous page)

<code>--logFile=&lt;arg&gt; [-l]</code>	<code>""</code>	Specify the
↳ log file (stdout is used <b>if</b> no log file is specified)		
<code>--logLevel=&lt;int&gt;</code>	<code>0</code>	Set the log
↳ level ( <code>0</code> : default, <code>1</code> : debug, <code>2</code> : debug+trace)		
<code>--master=&lt;arg&gt;</code>	<code>"ma"</code>	Specify the
↳ master algorithm (ma)		
<code>--maxEventIteration=&lt;int&gt;</code>	<code>100</code>	Specify the
↳ maximum number of iterations <b>for</b> handling a single event		
<code>--maxLoopIteration=&lt;int&gt;</code>	<code>10</code>	Specify the
↳ maximum number of iterations <b>for</b> solving algebraic loops between system-		
↳ level components. Internal algebraic loops of components are not affected.		
<code>--minimumStepSize=&lt;double&gt;</code>	<code>1e-12</code>	Specify the
↳ minimum step size		
<code>--mode=&lt;arg&gt; [-m]</code>	<code>"me"</code>	Force a
↳ certain FMI mode <b>if</b> the FMU provides both cs and me (cs, me)		
<code>--numProcs=&lt;int&gt; [-n]</code>	<code>1</code>	Specify the
↳ maximum number of processors to use ( <code>0</code> =auto, <code>1</code> =default)		
<code>--progressBar=&lt;bool&gt;</code>	<code>false</code>	Show a
↳ progress bar <b>for</b> the simulation progress <b>in</b> the terminal		
<code>--realTime=&lt;bool&gt;</code>	<code>false</code>	Enable
↳ experimental feature <b>for</b> (soft) real-time co-simulation		
<code>--resultFile=&lt;arg&gt; [-r]</code>	<code>"&lt;default&gt;"</code>	Specify the
↳ name of the output result file		
<code>--skipCSVHeader=&lt;bool&gt;</code>	<code>true</code>	Skip exporting
↳ the CSV delimiter <b>in</b> the header		
<code>--solver=&lt;arg&gt;</code>	<code>"cvmode"</code>	Specify the
↳ integration method (euler, cvmode)		
<code>--solverStats=&lt;bool&gt;</code>	<code>false</code>	Add solver
↳ stats to the result file, e.g., step size; not supported <b>for</b> all solvers		
<code>--startTime=&lt;double&gt; [-s]</code>	<code>0</code>	Specify the
↳ start time		
<code>--stepSize=&lt;double&gt;</code>	<code>1e-3</code>	Specify the
↳ (maximum) step size		
<code>--stopTime=&lt;double&gt; [-t]</code>	<code>1</code>	Specify the
↳ stop time		
<code>--stripRoot=&lt;bool&gt;</code>	<code>false</code>	Remove the
↳ root system prefix from all exported signals		
<code>--suppressPath=&lt;bool&gt;</code>	<code>false</code>	Suppress path
↳ information <b>in</b> info messages; especially useful <b>for</b> testing		
<code>--tempDir=&lt;arg&gt;</code>	<code>."</code>	Specify the
↳ temporary directory		
<code>--timeout=&lt;int&gt;</code>	<code>0</code>	Specify the
↳ maximum allowed time <b>in</b> seconds <b>for</b> running a simulation ( <code>0</code> disables)		
<code>--tolerance=&lt;double&gt;</code>	<code>1e-4</code>	Specify the
↳ relative tolerance		
<code>--version [-v]</code>		Display
↳ version information		
<code>--wallTime=&lt;bool&gt;</code>	<code>false</code>	Add wall time
↳ information to the result file		

(continues on next page)



(continued from previous page)

```
--workingDir=<arg>          "."          Specify the
↪working directory
--zeroNominal=<bool>        false        Accept FMUs
↪with invalid nominal values and replace the invalid nominal values with 1.0
```

## 5.57 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)
```

## 5.58 setInteger

Sets the value of a given integer signal.

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

## 5.59 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)
```

## 5.60 setLoggingInterval

Set the logging interval of the simulation.

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

## 5.61 setLoggingLevel

Enables/Disables debug logging (logDebug and logTrace).

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

```
oms_setLoggingLevel(logLevel)
```

## 5.62 setMaxLogFileSize

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

```
oms_setMaxLogFileSize(size)
```

## 5.63 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.

## 5.64 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)
```

## 5.65 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.

## 5.66 setSolver

Sets the solver method for the given system.

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

solver	Type	Description
oms_solver_sc_explicit_euler	sc-system	Explicit euler with fixed step size
oms_solver_sc_cvode	sc-system	CVODE with adaptive stepsize
oms_solver_wc_ma	wc-system	default master algorithm with fixed step size
oms_solver_wc_mav	wc-system	master algorithm with adaptive stepsize
oms_solver_wc_mav2	wc-system	master algorithm with adaptive stepsize (double-step)

## 5.67 setStartTime

Set the start time of the simulation.

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

## 5.68 setStopTime

Set the stop time of the simulation.

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

## 5.69 setString

Sets the value of a given string signal.

```
status = oms_setString(cref, value)
```

## 5.70 setTempDirectory

Set new temp directory.

```
status = oms_setTempDirectory(newTempDir)
```

## 5.71 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.

## 5.72 setUnit

Sets the unit of a given signal.

```
status = oms_setUnit(cref, value)
```

## 5.73 setVariableStepSize

Sets the step size parameters for methods with stepsize control.

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

## 5.74 setWorkingDirectory

Set a new working directory.

```
status = oms_setWorkingDirectory(newWorkingDir)
```

## 5.75 simulate

Simulates a composite model.

```
status = oms_simulate(cref)
```

## 5.76 simulate\_realtime

Experimental feature for (soft) real-time simulation.

```
status = experimental_simulate_realtime(ident)
```

## 5.77 stepUntil

Simulates a composite model until a given time value.

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

## 5.78 terminate

Terminates a given composite model.

```
status = oms_terminate(cref)
```



## OMSIMULATORPYTHON

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

Installation using `pip` is recommended:

```
> pip3 install OMSimulator --upgrade
```

### 6.1 Examples

```
from OMSimulator import OMSimulator

oms = OMSimulator()
oms.setTempDirectory("./temp/")
oms.newModel("model")
oms.addSystem("model.root", 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")
```

The python package also provides a more object oriented API. The following example is equivalent to the previous one:

```
import OMSimulator as oms

oms.setTempDirectory('./temp/')
model = oms.newModel("model")
root = model.addSystem('root', oms.Types.System.SC)

# instantiate FMUs
root.addSubModel('system1', 'FMUs/System1.fmu')
root.addSubModel('system2', 'FMUs/System2.fmu')

# add connections
root.addConnection('system1.y', 'system2.u')
root.addConnection('system2.y', 'system1.u')

# simulation settings
model.resultFile = 'results.mat'
model.stopTime = 0.1
model.fixedStepSize = 1e-4

model.instantiate()
model.setReal('root.system1.x_start', 2.5)
#or system.setReal('system1.x_start', 2.5)

model.initialize()
model.simulate()
model.terminate()
model.delete()
```

## 6.2 Python Scripting Commands

### 6.3 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.

```
status = oms.activateVariant(crefA, crefB)
```

An example of activating the number of available variants in a ssp file

```
oms_newModel("model")          oms_addSystem("model.root",          "system_wc")
oms_addSubModel("model.root.A", "A.fmu") oms_duplicateVariant("model", "varA") //
varA will be the current variant oms_duplicateVariant("varA", "varB") // varB will be the
current variant oms_activateVariant("varB", "varA") // Reactivate the variant varB to varA
oms_activateVariant("varA", "model") // Reactivate the variant varA to model
```



## 6.4 addBus

Adds a bus to a given component.

```
status = oms.addBus(cref)
```

## 6.5 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.

## 6.6 addConnector

Adds a connector to a given component.

```
status = oms.addConnector(cref, causality, type)
```

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

```
oms.input  
oms.output  
oms.parameter  
oms.bidir  
oms.undefined
```

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

```
oms.signal_type_real  
oms.signal_type_integer  
oms.signal_type_boolean  
oms.signal_type_string  
oms.signal_type_enum  
oms.signal_type_bus
```

## 6.7 addConnectorToBus

Adds a connector to a bus.

```
status = oms.addConnectorToBus(busCref, connectorCref)
```

## 6.8 addResources

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

```
status = oms.addResources(cref, path)

## Example
from OMSimulator import OMSimulator
oms = OMSimulator()
oms.importFile("addExternalResources1.ssp")
## add list of external resources from filesystem to ssp
oms.addResources("addExternalResources", "../resources/externalRoot.ssv")
oms.addResources("addExternalResources:externalSystem.ssv", "../resources/
↪externalSystem1.ssv")
oms.addResources("addExternalResources", "../resources/externalGain.ssv")
## export the ssp with new resources
oms_export("addExternalResources", "addExternalResources1.ssp")
```

## 6.9 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.

## 6.10 addSubModel

Adds a component to a system.

```
status = oms.addSubModel(cref, fmuPath)
```

## 6.11 addSystem

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

```
status = oms.addSystem(cref, type)
```

## 6.12 compareSimulationResults

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

```
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

## 6.13 copySystem

Copies a system.

```
status = oms.copySystem(source, target)
```

## 6.14 delete

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

```
status = oms.delete(cref)
```

## 6.15 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.

## 6.16 deleteConnectorFromBus

Deletes a connector from a given bus.

```
status = oms.deleteConnectorFromBus(busCref, connectorCref)
```

## 6.17 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” seperately 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”

```
status = oms.deleteResources(cref))
```

```
## Example
```

```
from OMSimulator import OMSimulator
oms = OMSimulator()
oms.importFile("deleteResources1.ssp")
## delete only the references in ".ssd" file
oms.deleteResources("deleteResources.root:root.ssv")
## delete both references and resources
oms.deleteResources("deleteResources:root.ssv")
oms.export("deleteResources1.ssp")
```

## 6.18 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.

```
status = oms.doStep(cref)
```

## 6.19 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.

```
status = oms.duplicateVariant(crefA, crefB)
```

An example of creating a multi-variant modelling is presented below

```
oms_newModel("model")
oms_addSystem("model.root", "system_wc")
oms_addSubModel("model.root.A", "A.fmu")
oms_setReal("model.root.A.param1", "10")
oms_duplicateVariant("model", "varB")
oms_addSubModel("varB.root.B", "B.fmu")
oms_setReal("varB.root.A.param2", "20")
oms_export("varB", "variant.ssp")
```

The variant.ssp file will have the following structure

```
Variant.ssp
  SystemStructure.ssd
  varB.ssd
  resources\
    A.fmu
    B.fmu
```

## 6.20 export

Exports a composite model to a SPP file.

```
status = oms.export(cref, filename)
```

## 6.21 exportDependencyGraphs

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

```
status = oms.exportDependencyGraphs(cref, initialization, event, simulation)
```

## 6.22 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.

```
status = oms.exportSSMTemplate(cref, filename)
```

## 6.23 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.

```
status = oms.exportSSVTemplate(cref, filename)
```

## 6.24 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)
```

## 6.25 freeMemory

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

```
oms.freeMemory(obj)
```

## 6.26 getBoolean

Get boolean value of given signal.

```
value, status = oms.getBoolean(cref)
```

## 6.27 getDirectionalDerivative

This function computes the directional derivatives of an FMU.

```
value, status = oms.getDirectionalDerivative(cref)
```

## 6.28 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)
```

## 6.29 getInteger

Get integer value of given signal.

```
value, status = oms.getInteger(cref)
```

## 6.30 getReal

Get real value.

```
value, status = oms.getReal(cref)
```

### 6.31 getResultFile

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

```
filename, bufferSize, status = oms.getResultFile(cref)
```

### 6.32 getSolver

Gets the selected solver method of the given system.

```
solver, status = oms.getSolver(cref)
```

### 6.33 getStartTime

Get the start time from the model.

```
startTime, status = oms.getStartTime(cref)
```

### 6.34 getStopTime

Get the stop time from the model.

```
stopTime, status = oms.getStopTime(cref)
```

### 6.35 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.

```
value, status = oms.getString(cref)
```

### 6.36 getSubModelPath

Returns the path of a given component.

```
path, status = oms.getSubModelPath(cref)
```



## 6.37 getSystemType

Gets the type of the given system.

```
type, status = oms.getSystemType(cref)
```

## 6.38 getTime

Get the current simulation time from the model.

```
time, status = oms.getTime(cref)
```

## 6.39 getTolerance

Gets the tolerance of a given system or component.

```
relativeTolerance, status = oms.getTolerance(cref)
```

## 6.40 getVariableStepSize

Gets the step size parameters.

```
initialStepSize, minimumStepSize, maximumStepSize, status = oms.  
↪getVariableStepSize(cref)
```

## 6.41 getVersion

Returns the library's version string.

```
oms = OMSimulator()  
oms.getVersion()
```

## 6.42 importFile

Imports a composite model from a SSP file.

```
cref, status = oms.importFile(filename)
```

## 6.43 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.

```
newCref, status = oms.importSnapshot(cref, snapshot)
```

## 6.44 initialize

Initializes a composite model.

```
status = oms.initialize(cref)
```

## 6.45 instantiate

Instantiates a given composite model.

```
status = oms.instantiate(cref)
```

## 6.46 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)
```

## 6.47 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)
```

## 6.48 listVariants

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

```
status = oms.listVariants(cref)
```

An example for finding the number of available variants in a ssp file

```
oms_newModel("model")
oms_addSystem("model.root", "system_wc")
oms_addSubModel("model.root.A", "A.fmu")
oms_duplicateVariant("model", "varA")
oms_duplicateVariant("varA", "varB")

oms_listVariants("varB")
```

The API will list the available variants like below

```
<oms:Variants>
  <oms:variant name="model" />
  <oms:variant name="varB" />
  <oms:variant name="varA" />
</oms:Variants>
```

## 6.49 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.

```
newCref, status = oms.loadSnapshot(cref, snapshot)
```

## 6.50 newModel

Creates a new and yet empty composite model.

```
status = oms.newModel(cref)
```

## 6.51 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”

```
status = oms.newResources(cref)

## Example
from OMSimulator import OMSimulator
oms = OMSimulator()
oms.newModel("newResources")

oms.addSystem("newResources.root", oms_system_wc)
oms.addConnector("newResources.root.Input1", oms.input, oms_signal_type_real)
oms.addConnector("newResources.root.Input2", oms.input, oms_signal_type_real)

## add Top level resources, the filename is provided using the colon suffix
→ ":root.ssv"
oms.newResources("newResources.root:root.ssv")
oms.setReal("newResources.root.Input1", 10)
## export the ssp with new resources
oms.export("newResources", "newResources.ssp")
```

## 6.52 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” seperately as the ssm file is binded to a ssv file. Hence it is not possible to switch the reference of “.ssm” file alone. So inorder 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.

```
status = oms.referenceResources(cref, ssmFile)

## Example
from OMSimulator import OMSimulator
oms = OMSimulator()
oms.importFile("referenceResources1.ssp")
## delete only the references in ".ssv" file
oms.deleteResources("referenceResources1.root:root.ssv")
## usage-1 switch with new references, only ssv file
oms.referenceResources("referenceResources1.root:Config1.ssv")
```

(continues on next page)

(continued from previous page)

```
## usage-2 switch with new references, both ssv and ssm file
oms.referenceResources("referenceResources1.root:Config1.ssv", "Config1.ssm")
```

## 6.53 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.

## 6.54 rename

Renames a model, system, or component.

```
status = oms.rename(cref, newCref)
```

## 6.55 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.

```
status = oms.replaceSubModel(cref, fmuPath)
```

It is possible to import an partially developed fmu (i.e contains only modeldescription.xml without any binaries) in OMSimulator, and later can be replaced with a fully developed fmu. An example to use the API, oms\_addSubModel(“model.root.A”, “./resources/replaceA.fmu”) oms\_export(“model”, “test.ssp”) oms\_import(“test.ssp”) oms\_replaceSubModel(“model.root.A”, “./resources/replaceA\_extended.fmu”)

## 6.56 reset

Reset the composite model after a simulation run.

The FMUs go into the same state as after instantiation.

```
status = oms.reset(cref)
```

## 6.57 setBoolean

Sets the value of a given boolean signal.

```
status = oms.setBoolean(cref, value)
```

## 6.58 setCommandLineOption

Sets special flags.

```
status = oms.setCommandLineOption(cmd)
```

Available flags:

```
info:      Usage: OMSimulator [Options] [Lua script] [FMU] [SSP file]
Options:
  --addParametersToCSV=<bool>      false      Export
↳ parameters to a .csv file
  --algLoopSolver=<arg>            "kinsol"      Specifies the
↳ loop solver method (fixedpoint, kinsol) used for algebraic loops spanning
↳ multiple components.
  --clearAllOptions                Reset all
↳ flags to their default values
  --CVODEMaxErrTestFails=<int>     100      Maximum number
↳ of error test failures for CVODE
  --CVODEMaxNLSFailures=<int>     100      Maximum number
↳ of nonlinear convergence failures for CVODE
  --CVODEMaxNLSIterations=<int>   5      Maximum number
↳ of nonlinear solver iterations for CVODE
  --CVODEMaxSteps=<int>           1000      Maximum number
↳ of steps for CVODE
  --deleteTempFiles=<bool>        true      Delete
↳ temporary files as soon as they are no longer needed
  --directionalDerivatives=<bool> true      Use
↳ directional derivatives to calculate the Jacobian for algebraic loops
  --dumpAlgLoops=<bool>           false      Dump
↳ information for algebraic loops
  --emitEvents=<bool>            true      Emit events
↳ during simulation
  --help [-h]                    Display the
```

(continues on next page)

(continued from previous page)

```

↪help text
    --ignoreInitialUnknowns=<bool>      false      Ignore initial
↪unknowns from the modelDescription.xml
    --initialStepSize=<double>           1e-6        Specify the
↪initial step size
    --inputExtrapolation=<bool>          false      Enable input
↪extrapolation using derivative information
    --intervals=<int> [-i]               500         Specify the
↪number of communication points (arg > 1)
    --logFile=<arg> [-l]                 ""          Specify the
↪log file (stdout is used if no log file is specified)
    --logLevel=<int>                     0           Set the log
↪level (0: default, 1: debug, 2: debug+trace)
    --master=<arg>                       "ma"        Specify the
↪master algorithm (ma)
    --maxEventIteration=<int>             100         Specify the
↪maximum number of iterations for handling a single event
    --maxLoopIteration=<int>              10          Specify the
↪maximum number of iterations for solving algebraic loops between system-
↪level components. Internal algebraic loops of components are not affected.
    --minimumStepSize=<double>           1e-12       Specify the
↪minimum step size
    --mode=<arg> [-m]                    "me"        Force a
↪certain FMI mode if the FMU provides both cs and me (cs, me)
    --numProcs=<int> [-n]                 1           Specify the
↪maximum number of processors to use (0=auto, 1=default)
    --progressBar=<bool>                  false      Show a
↪progress bar for the simulation progress in the terminal
    --realTime=<bool>                     false      Enable
↪experimental feature for (soft) real-time co-simulation
    --resultFile=<arg> [-r]               "<default>" Specify the
↪name of the output result file
    --skipCSVHeader=<bool>                 true       Skip exporting
↪the CSV delimiter in the header
    --solver=<arg>                        "cvmode"    Specify the
↪integration method (euler, cvmode)
    --solverStats=<bool>                   false      Add solver
↪stats to the result file, e.g., step size; not supported for all solvers
    --startTime=<double> [-s]              0          Specify the
↪start time
    --stepSize=<double>                    1e-3       Specify the
↪(maximum) step size
    --stopTime=<double> [-t]               1          Specify the
↪stop time
    --stripRoot=<bool>                     false      Remove the
↪root system prefix from all exported signals
    --suppressPath=<bool>                  false      Suppress path
↪information in info messages; especially useful for testing
    --tempDir=<arg>                        "."         Specify the

```

(continues on next page)

(continued from previous page)

```

↪temporary directory
    --timeout=<int>                                0                Specify the
↪maximum allowed time in seconds for running a simulation (0 disables)
    --tolerance=<double>                            1e-4            Specify the
↪relative tolerance
    --version [-v]                                  Display
↪version information
    --wallTime=<bool>                                false           Add wall time
↪information to the result file
    --workingDir=<arg>                              "."              Specify the
↪working directory
    --zeroNominal=<bool>                            false           Accept FMUs
↪with invalid nominal values and replace the invalid nominal values with 1.0

```

## 6.59 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)
```

## 6.60 setInteger

Sets the value of a given integer signal.

```
status = oms.setInteger(cref, value)
```

## 6.61 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)
```

## 6.62 setLoggingInterval

Set the logging interval of the simulation.

```
status = oms.setLoggingInterval(cref, loggingInterval)
```



## 6.63 setLoggingLevel

Enables/Disables debug logging (logDebug and logTrace).

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

```
oms.setLoggingLevel(logLevel)
```

## 6.64 setMaxLogFileSize

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

```
oms.setMaxLogFileSize(size)
```

## 6.65 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.

## 6.66 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)
```

## 6.67 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.

## 6.68 setSolver

Sets the solver method for the given system.

```
status = oms.setSolver(cref, solver)
```

solver	Type	Description
oms.solver_sc_explicit_euler	sc-system	Explicit euler with fixed step size
oms.solver_sc_cvode	sc-system	CVODE with adaptive stepsize
oms.solver_wc_ma	wc-system	default master algorithm with fixed step size
oms.solver_wc_mav	wc-system	master algorithm with adaptive stepsize
oms.solver_wc_mav2	wc-system	master algorithm with adaptive stepsize (double-step)

## 6.69 setStartTime

Set the start time of the simulation.

```
status = oms.setStartTime(cref, startTime)
```

## 6.70 setStopTime

Set the stop time of the simulation.

```
status = oms.setStopTime(cref, stopTime)
```

## 6.71 setString

Sets the value of a given string signal.

```
status = oms.setString(cref, value)
```

## 6.72 setTempDirectory

Set new temp directory.

```
status = oms.setTempDirectory(newTempDir)
```

## 6.73 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.

## 6.74 setUnit

Sets the unit of a given signal.

```
status = oms.setUnit(cref, value)
```

## 6.75 setVariableStepSize

Sets the step size parameters for methods with stepsize control.

```
status = oms.getVariableStepSize(cref, initialStepSize, minimumStepSize, ↵  
↵maximumStepSize)
```

## 6.76 setWorkingDirectory

Set a new working directory.

```
status = oms.setWorkingDirectory(newWorkingDir)
```

## 6.77 simulate

Simulates a composite model.

```
status = oms.simulate(cref)
```

## 6.78 stepUntil

Simulates a composite model until a given time value.

```
status = oms.stepUntil(cref, stopTime)
```

## 6.79 terminate

Terminates a given composite model.

```
status = oms.terminate(cref)
```

### 6.79.1 Example: Pi

This example uses a simple Modelica model and FMI-based batch simulation to approximate the value of pi.

A Modelica model is used to calculate two uniform distributed pseudo-random numbers between 0 and 1 based on a seed value and evaluates if the resulting coordinate is inside the unit circle or not.

```
model Circle
  parameter Integer globalSeed = 30020 "global seed to initialize random
  ↪number generator";
  parameter Integer localSeed = 614657 "local seed to initialize random
  ↪number generator";
  Real x;
  Real y;
  Boolean inside = x*x + y*y < 1.0;
protected
  Integer state128[4];
algorithm
  when initial() then
    state128 := Modelica.Math.Random.Generators.Xorshift128plus.
```

(continues on next page)

(continued from previous page)

```

↪ initialState(localSeed, globalSeed);
    (x, state128) := Modelica.Math.Random.Generators.Xorshift128plus.
↪ random(state128);
    (y, state128) := Modelica.Math.Random.Generators.Xorshift128plus.
↪ random(state128);
    end when;
    annotation(uses(Modelica(version="4.0.0")));
end Circle;

```

The model is then exported using the FMI interface and the generated FMU can then be used to run a million simulations in just a few seconds.

Listing 1: Batch simulation of the simple *Circle* model with different seed values. All OMSimulator-related commands are highlighted for convenience.

```

1  import math
2  import matplotlib.pyplot as plt
3  import OMSimulator as oms
4
5  # redirect logging to file and limit the file size to 65MB
6  oms.setLogFile('pi.log', 65)
7
8  model = oms.newModel('pi')
9  root = model.addSystem('root', oms.Types.System.SC)
10 root.addSubModel('circle', 'Circle.fmu')
11
12 model.resultFile = '' # no result file
13 model.instantiate()
14
15 results = list()
16 inside = 0
17
18 MIN = 100
19 MAX = 1000000
20 for i in range(0, MAX+1):
21     if i > 0:
22         model.reset()
23         model.setInteger('root.circle.globalSeed', i)
24         model.initialize()
25         if model.getBoolean("root.circle.inside"):
26             inside = inside + 1
27         if i >= MIN:
28             results.append(4.0*inside/i)
29 model.terminate()
30 model.delete()
31
32 plt.plot([MIN, MAX], [math.pi, math.pi], 'r--', range(MIN, MAX+1), results)
33 plt.xscale('log')
34 plt.ylabel('Approximation of pi')

```

(continues on next page)

(continued from previous page)

```
35 plt.savefig('pi.png')
```

The following figure shows the approximation of pi in relation to the number of samples.

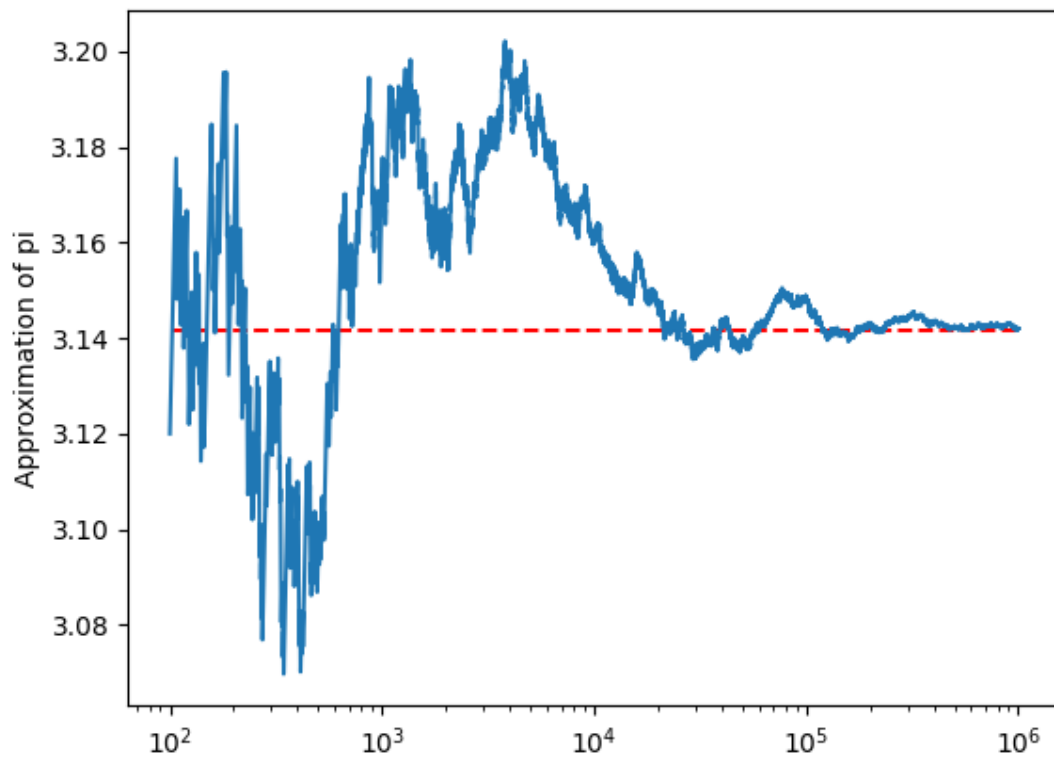


Fig. 1: Results of the above batch simulation which approximates the value of pi

## **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.

### **7.1 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.

<b>Category</b>	<b>Functionality</b>
<b>Model Management</b>	<ul style="list-style-type: none"><li>• Create new SSP models</li><li>• Create, activate, and duplicate variants</li><li>• Add or delete resources, components, systems, connectors</li><li>• Create nested systems</li><li>• Set parameters and solvers</li><li>• Swap SSV references</li></ul>
<b>Connectivity</b>	<ul style="list-style-type: none"><li>• Add components and connectors</li><li>• Define and manage connections</li></ul>
<b>SSP Standards</b>	<ul style="list-style-type: none"><li>• Import and export SSP models</li><li>• Create SSV and SSM descriptions</li><li>• Export SSV and SSM templates</li><li>• Validate SSP, SSV, and SSM</li></ul>
<b>FMU Integration</b>	<ul style="list-style-type: none"><li>• Import and validate FMUs as components</li></ul>
<b>Utilities</b>	<ul style="list-style-type: none"><li>• Error handling, logging, and debugging support</li></ul>

## 7.2 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',
```

(continues on next page)



(continued from previous page)

```

↪ '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)
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()

```

## 7.3 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()

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

# list the ssp components
model.list()

```

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.

## 7.4 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()
```

## 7.5 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")
```

## 7.6 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")
```

## 7.7 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()
```

(continues on next page)

(continued from previous page)

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

## 7.8 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)
```

(continues on next page)

(continued from previous page)

```
# 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')

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

## 7.9 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'))
```

## 7.10 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))
```

## 7.11 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')
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()
```

## 7.12 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')
```

(continues on next page)



(continued from previous page)

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

## 7.13 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
```

(continues on next page)

(continued from previous page)

```
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()
```

## 7.14 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
```

(continues on next page)

(continued from previous page)

```

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
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')

```

## 7.15 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()
```

## 7.16 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")
```

(continues on next page)

(continued from previous page)

```
# 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()
```

## 7.17 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(
```

(continues on next page)

(continued from previous page)

```
↪ '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'))}")
```

## 7.18 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")
```

(continues on next page)

(continued from previous page)

```
# 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')
```

## 7.19 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)
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')
```

## 7.20 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).
- 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*): 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()

# 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")
```

## 7.21 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'))}"
      ↪, flush=True)

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

## 7.22 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
model.mapParameter(CRef("default"), "connector_input", "input1")
model.mapParameter(CRef("default"), "connector_input", "input2")
```

## 7.23 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)

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

(continues on next page)

(continued from previous page)

```
# 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()
```

## 7.24 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

# Load an existing SSP model
model = SSP("exportSSMTemplate1.ssp")
## get list of available variants
```

(continues on next page)

(continued from previous page)

```
model.getAllVariantNames()
## get current active variant
varA = model.getVariant("varA")
```

## 7.25 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())
```

## 7.26 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()
```

## 7.27 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()
```

## 7.28 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)
```

## 7.29 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

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

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

## 7.30 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()
```

## 7.31 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()
```

## 7.32 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.

### 8.1 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();
```

## 8.2 OpenModelica Scripting Commands

### 8.3 addBus

Adds a bus to a given component.

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

### 8.4 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.

### 8.5 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"
```

## 8.6 addConnectorToBus

Adds a connector to a bus.

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

## 8.7 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.

## 8.8 addSubModel

Adds a component to a system.

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

## 8.9 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"
```

## 8.10 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

## 8.11 copySystem

Copies a system.

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

## 8.12 delete

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

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

## 8.13 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.

## 8.14 deleteConnectorFromBus

Deletes a connector from a given bus.

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



## 8.15 export

Exports a composite model to a SPP file.

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

## 8.16 exportDependencyGraphs

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

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

## 8.17 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);
```

## 8.18 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

## 8.19 getBoolean

Get boolean value of given signal.

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

## 8.20 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_setFixedStepSize(cref);
```

## 8.21 getInteger

Get integer value of given signal.

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

## 8.22 getModelState

Gets the model state of the given model cref.

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

## 8.23 getReal

Get real value.

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

## 8.24 getSolver

Gets the selected solver method of the given system.

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

## 8.25 getStartTime

Get the start time from the model.

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

## 8.26 getStopTime

Get the stop time from the model.

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

## 8.27 getSubModelPath

Returns the path of a given component.

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

## 8.28 getSystemType

Gets the type of the given system.

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

## 8.29 getTime

Get the current simulation time from the model.

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

## 8.30 getTolerance

Gets the tolerance of a given system or component.

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

## 8.31 getVariableStepSize

Gets the step size parameters.

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

## 8.32 getVersion

Returns the library's version string.

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

### 8.33 importFile

Imports a composite model from a SSP file.

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

### 8.34 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);
```

### 8.35 initialize

Initializes a composite model.

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

### 8.36 instantiate

Instantiates a given composite model.

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

### 8.37 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);
```

### 8.38 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);
```

## 8.39 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);
```

## 8.40 newModel

Creates a new and yet empty composite model.

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

## 8.41 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.

## 8.42 rename

Renames a model, system, or component.

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

## 8.43 reset

Reset the composite model after a simulation run.

The FMUs go into the same state as after instantiation.

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

## 8.44 setBoolean

Sets the value of a given boolean signal.

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

## 8.45 setCommandLineOption

Sets special flags.

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

Available flags:

```
info:      Usage: OMSimulator [Options] [Lua script] [FMU] [SSP file]
           Options:
             --addParametersToCSV=<bool>      false      Export
           ↪ parameters to a .csv file
             --algLoopSolver=<arg>             "kinsol"    Specifies the
           ↪ loop solver method (fixedpoint, kinsol) used for algebraic loops spanning
           ↪ multiple components.
             --clearAllOptions                  Reset all
           ↪ flags to their default values
             --CVODEMaxErrTestFails=<int>      100         Maximum number
           ↪ of error test failures for CVODE
             --CVODEMaxNLSFailures=<int>       100         Maximum number
           ↪ of nonlinear convergence failures for CVODE
             --CVODEMaxNLSIterations=<int>     5           Maximum number
           ↪ of nonlinear solver iterations for CVODE
             --CVODEMaxSteps=<int>             1000        Maximum number
           ↪ of steps for CVODE
             --deleteTempFiles=<bool>          true        Delete
           ↪ temporary files as soon as they are no longer needed
             --directionalDerivatives=<bool>    true        Use
           ↪ directional derivatives to calculate the Jacobian for algebraic loops
             --dumpAlgLoops=<bool>              false      Dump
           ↪ information for algebraic loops
             --emitEvents=<bool>                true        Emit events
           ↪ during simulation
             --help [-h]                       Display the
           ↪ help text
             --ignoreInitialUnknowns=<bool>    false      Ignore initial
           ↪ unknowns from the modelDescription.xml
             --initialStepSize=<double>         1e-6        Specify the
           ↪ initial step size
             --inputExtrapolation=<bool>        false      Enable input
           ↪ extrapolation using derivative information
             --intervals=<int> [-i]             500         Specify the
           ↪ number of communication points (arg > 1)
```

(continues on next page)

(continued from previous page)

<code>--logFile=&lt;arg&gt; [-l]</code>	<code>""</code>	Specify the
↪ log file (stdout is used <b>if</b> no log file is specified)		
<code>--logLevel=&lt;int&gt;</code>	<code>0</code>	Set the log
↪ level ( <code>0</code> : default, <code>1</code> : debug, <code>2</code> : debug+trace)		
<code>--master=&lt;arg&gt;</code>	<code>"ma"</code>	Specify the
↪ master algorithm (ma)		
<code>--maxEventIteration=&lt;int&gt;</code>	<code>100</code>	Specify the
↪ maximum number of iterations <b>for</b> handling a single event		
<code>--maxLoopIteration=&lt;int&gt;</code>	<code>10</code>	Specify the
↪ maximum number of iterations <b>for</b> solving algebraic loops between system-		
↪ level components. Internal algebraic loops of components are not affected.		
<code>--minimumStepSize=&lt;double&gt;</code>	<code>1e-12</code>	Specify the
↪ minimum step size		
<code>--mode=&lt;arg&gt; [-m]</code>	<code>"me"</code>	Force a
↪ certain FMI mode <b>if</b> the FMU provides both cs and me (cs, me)		
<code>--numProcs=&lt;int&gt; [-n]</code>	<code>1</code>	Specify the
↪ maximum number of processors to use ( <code>0</code> =auto, <code>1</code> =default)		
<code>--progressBar=&lt;bool&gt;</code>	<code>false</code>	Show a
↪ progress bar <b>for</b> the simulation progress <b>in</b> the terminal		
<code>--realTime=&lt;bool&gt;</code>	<code>false</code>	Enable
↪ experimental feature <b>for</b> (soft) real-time co-simulation		
<code>--resultFile=&lt;arg&gt; [-r]</code>	<code>"&lt;default&gt;"</code>	Specify the
↪ name of the output result file		
<code>--skipCSVHeader=&lt;bool&gt;</code>	<code>true</code>	Skip exporting
↪ the CSV delimiter <b>in</b> the header		
<code>--solver=&lt;arg&gt;</code>	<code>"cvmode"</code>	Specify the
↪ integration method (euler, cvmode)		
<code>--solverStats=&lt;bool&gt;</code>	<code>false</code>	Add solver
↪ stats to the result file, e.g., step size; not supported <b>for</b> all solvers		
<code>--startTime=&lt;double&gt; [-s]</code>	<code>0</code>	Specify the
↪ start time		
<code>--stepSize=&lt;double&gt;</code>	<code>1e-3</code>	Specify the
↪ (maximum) step size		
<code>--stopTime=&lt;double&gt; [-t]</code>	<code>1</code>	Specify the
↪ stop time		
<code>--stripRoot=&lt;bool&gt;</code>	<code>false</code>	Remove the
↪ root system prefix from all exported signals		
<code>--suppressPath=&lt;bool&gt;</code>	<code>false</code>	Suppress path
↪ information <b>in</b> info messages; especially useful <b>for</b> testing		
<code>--tempDir=&lt;arg&gt;</code>	<code>."</code>	Specify the
↪ temporary directory		
<code>--timeout=&lt;int&gt;</code>	<code>0</code>	Specify the
↪ maximum allowed time <b>in</b> seconds <b>for</b> running a simulation ( <code>0</code> disables)		
<code>--tolerance=&lt;double&gt;</code>	<code>1e-4</code>	Specify the
↪ relative tolerance		
<code>--version [-v]</code>		Display
↪ version information		
<code>--wallTime=&lt;bool&gt;</code>	<code>false</code>	Add wall time
↪ information to the result file		

(continues on next page)

(continued from previous page)

```
--workingDir=<arg>          "."          Specify the
↪working directory
--zeroNominal=<bool>        false        Accept FMUs
↪with invalid nominal values and replace the invalid nominal values with 1.0
```

## 8.46 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);
```

## 8.47 setInteger

Sets the value of a given integer signal.

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

## 8.48 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);
```

## 8.49 setLoggingInterval

Set the logging interval of the simulation.

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

## 8.50 setLoggingLevel

Enables/Disables debug logging (logDebug and logTrace).

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

```
oms_setLoggingLevel(logLevel);
```



## 8.51 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.

## 8.52 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);
```

## 8.53 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.

## 8.54 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_cvmode"
"OpenModelica.Scripting.oms_solver.oms_solver_sc_max"
"OpenModelica.Scripting.oms_solver.oms_solver_wc_min"
```

(continues on next page)

(continued from previous page)

```
"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"
```

## 8.55 setStartTime

Set the start time of the simulation.

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

## 8.56 setStopTime

Set the stop time of the simulation.

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

## 8.57 setTempDirectory

Set new temp directory.

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

## 8.58 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.

## 8.59 setVariableStepSize

Sets the step size parameters for methods with stepsize control.

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

## 8.60 setWorkingDirectory

Set a new working directory.

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

## 8.61 simulate

Simulates a composite model.

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

## 8.62 stepUntil

Simulates a composite model until a given time value.

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

## 8.63 terminate

Terminates a given composite model.

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



## GRAPHICAL MODELLING

OMSimulator has an optional dependency to OpenModelica in order to utilize the graphical modelling editor OMEdit. This feature requires to install the full OpenModelica tool suite, which includes OM-Simulator. The independent stand-alone version doesn't provide any graphical modelling editor.

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](#).

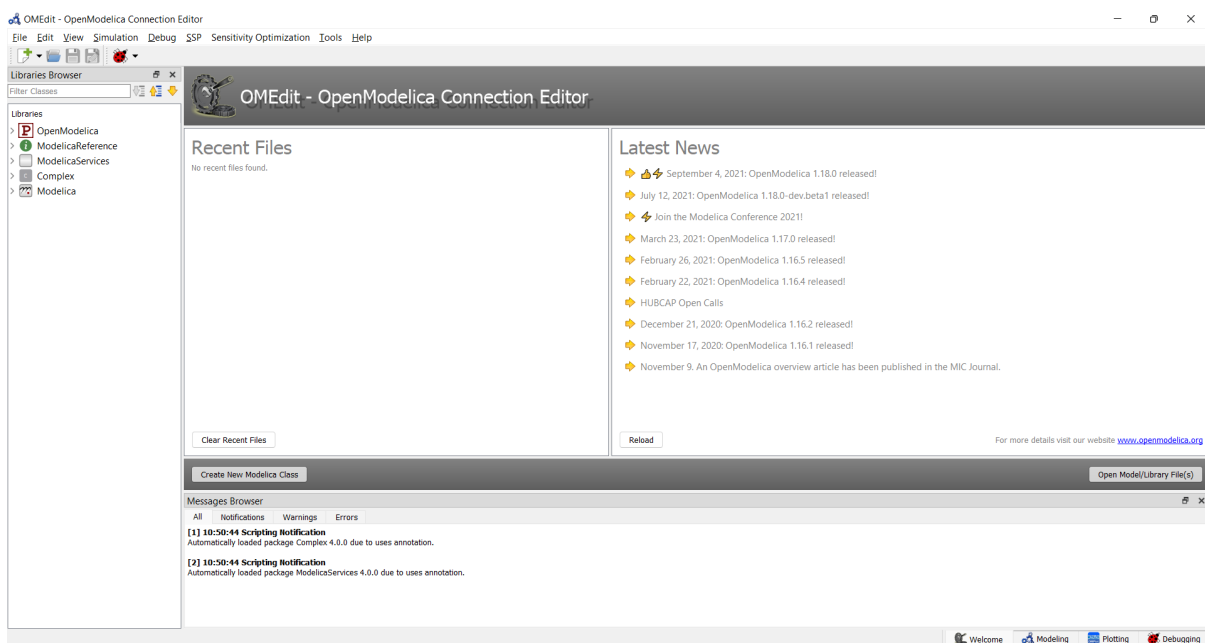


Fig. 1: OMEdit MainWindow and Browsers.

### 9.1 New SSP Model

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

That will open a dialog to enter the names of the model and the root system and to choose the root systems type.

**There are three types available:**

- TLM - Transmission Line Modeling System

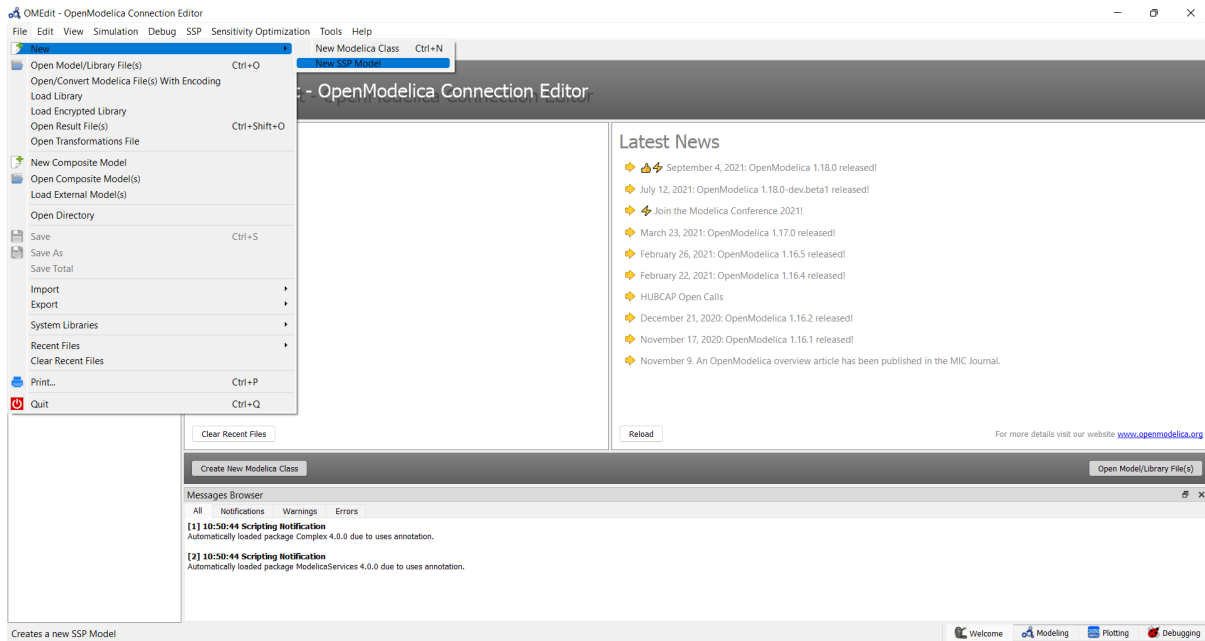


Fig. 2: OMEdit: New SSP Model

- Weakly Coupled - Connected Co-Simulation FMUs System
- Strongly Coupled - Connected Model-Exchange FMUs System

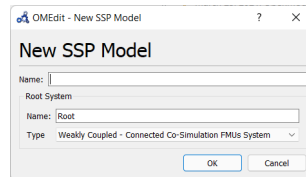


Fig. 3: OMEdit: New SSP Model Dialog

## 9.2 Add System

When a new model is created a root system is always generated. If you need to have another system in your root system you can add it with *SSP->Add System*.

For example only a weakly coupled system (Co-Simulation) can integrate strongly coupled system (Model Exchange). Therefore, the weakly coupled system must be selected from the Libraries Browser and the respective menu item can be selected:

That will pop-up a dialog to enter the names of the new system.

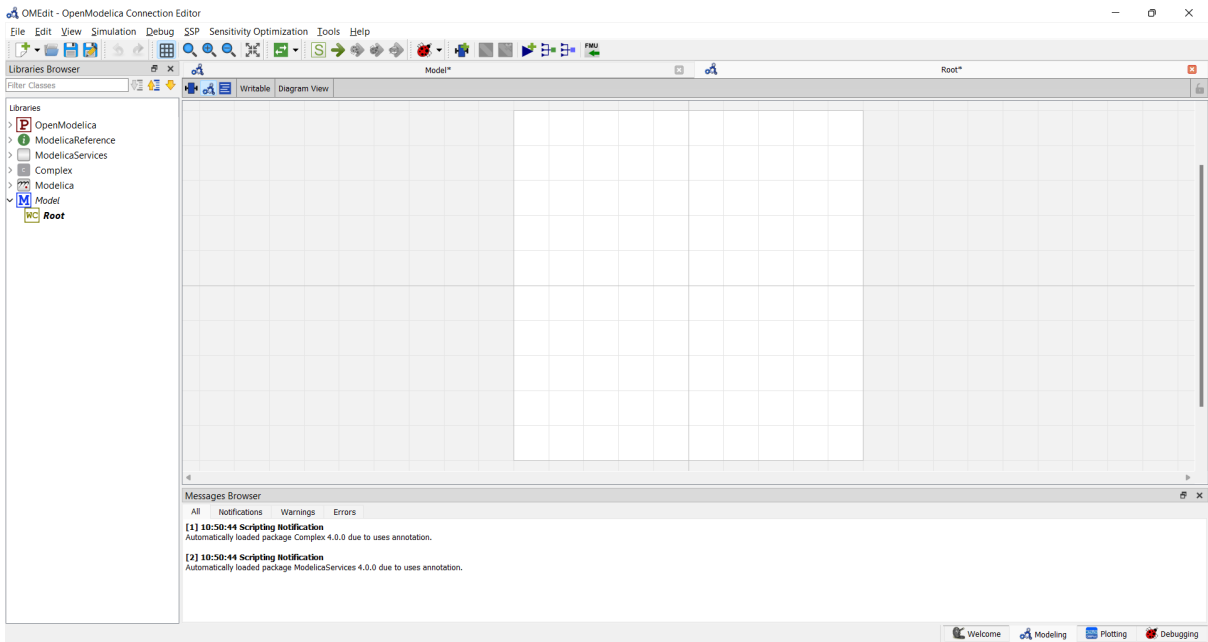


Fig. 4: OMEdit: Newly created empty root system of SSP model

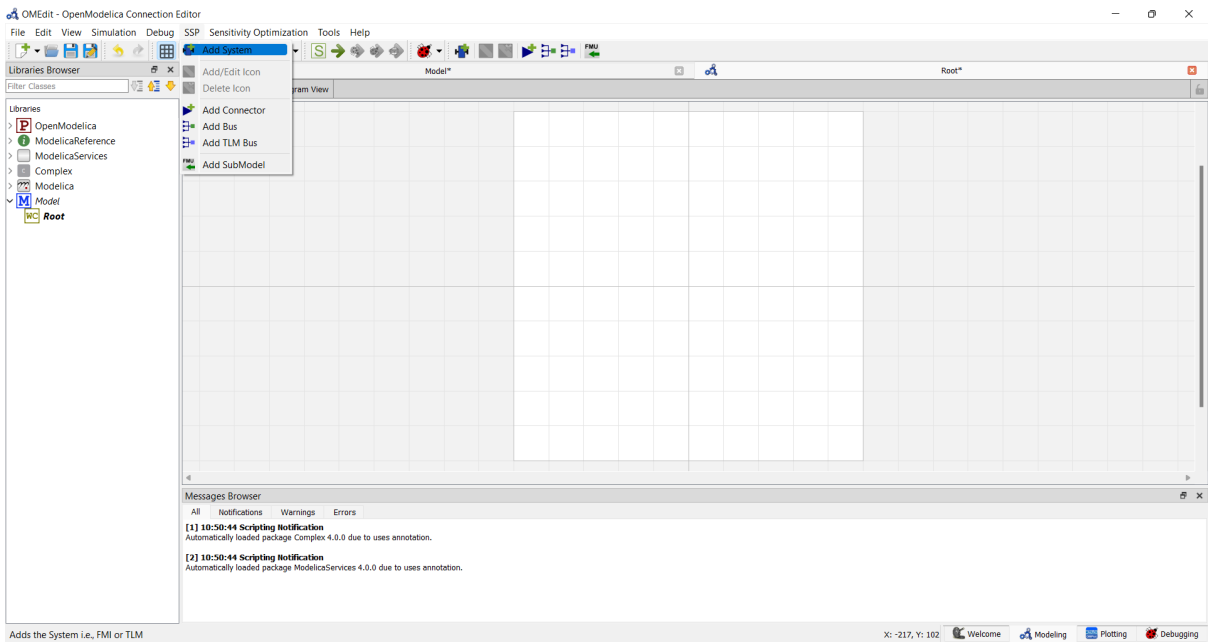


Fig. 5: OMEdit: Add System

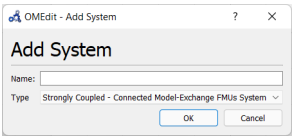


Fig. 6: OMEdit: Add System Dialog

## 9.3 Add SubModel

A sub-model is typically an FMU, but it also can be result file. In order to import a sub-model, the respective system must be selected and the action can be selected from the menu bar:

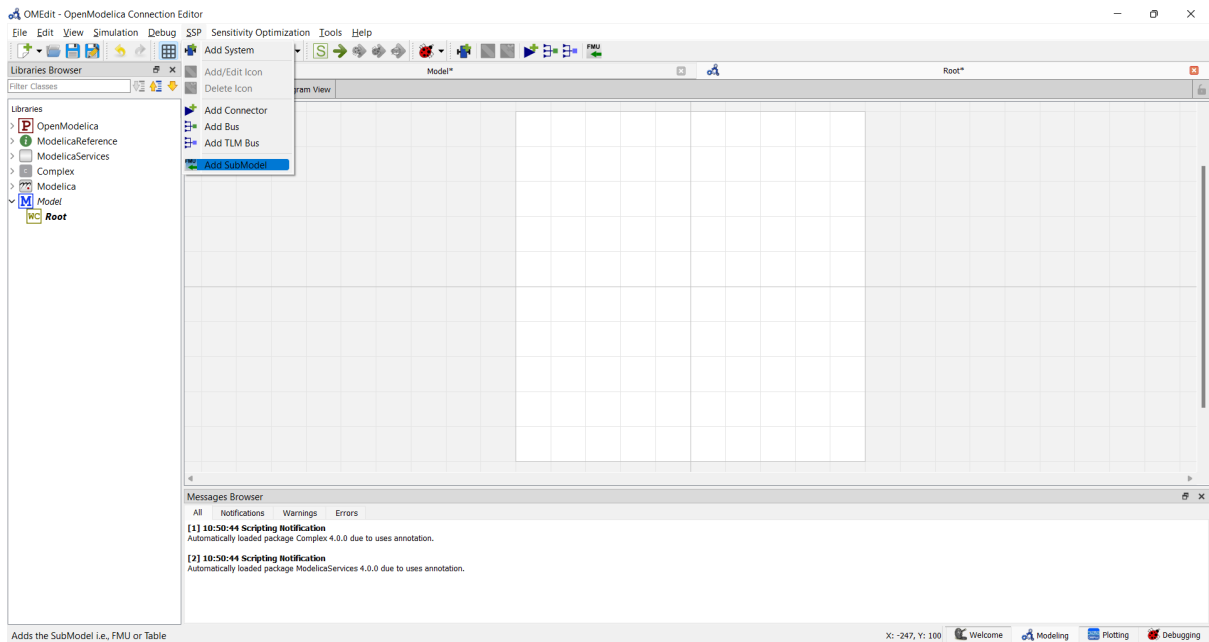


Fig. 7: OMEdit: Add SubModel

The file browser will open to select an FMU (.fmu) or result file (.csv) as a submodel. Then a dialog opens to choose the name of the new sub-model.

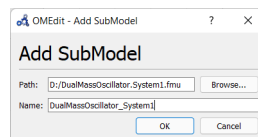


Fig. 8: OMEdit: Add SubModel Dialog

## 9.4 Simulate

Select the simulate button (symbol with green arrow) or select *Simulation->Simulate* from the menu in OMEdit to simulate the SSP model.



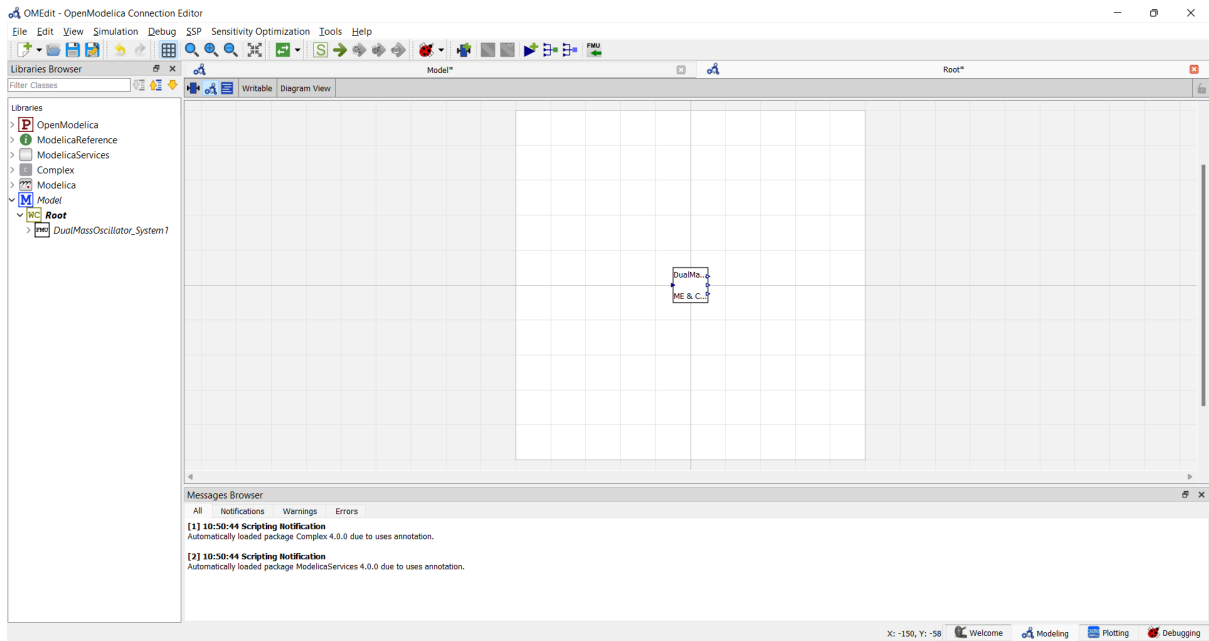


Fig. 9: OMEdit: Root system with added FMU.

## 9.5 Dual Mass Oscillator Example

The dual mass oscillator example from our testsuite is a simple example one can recreate using components from the Modelica Standard Library. After splitting the model into two models and exporting each as an Model-Exchange and Co-Simulation FMU.

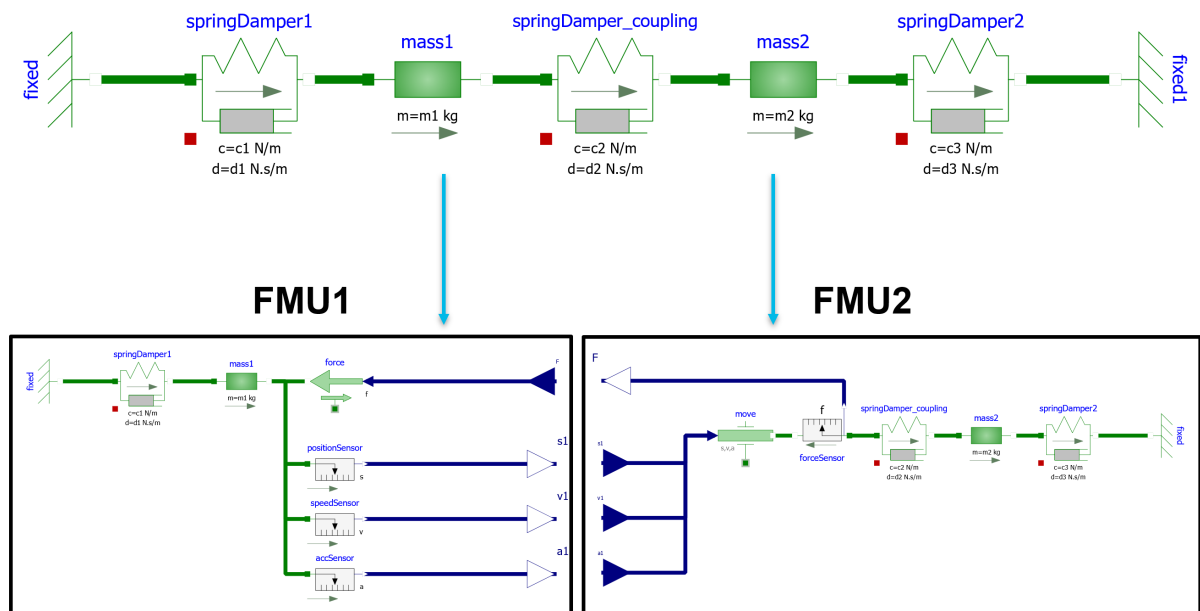


Fig. 10: Dual mass oscillator Modelica model (diagramm view) and FMUs

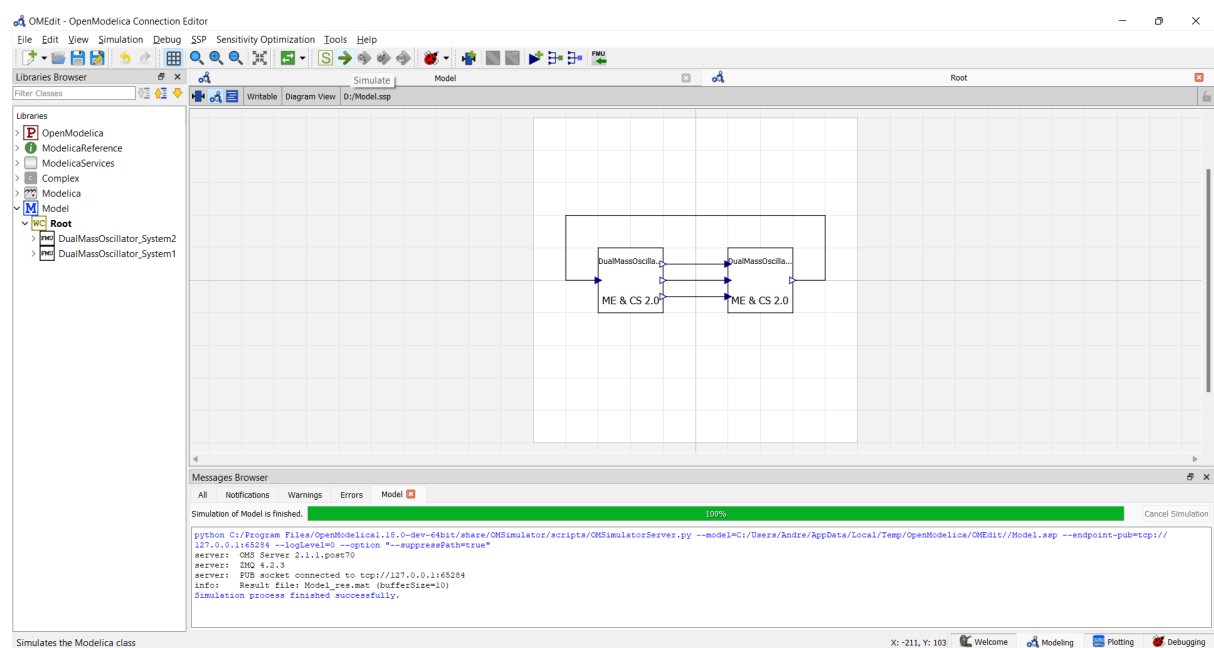


Fig. 11: OMEdit: Simulate Dual Mass Oscillator SSP model

## SSP SUPPORT

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

### 10.1 Bus Connections

Bus connections are saved as annotations to the SSD file. Bus connectors are only allowed in weakly coupled and strongly coupled systems. Bus connections can exist in any system type. Bus connectors are used to hide SSD connectors and bus connections are used to hide existing SSD connections in the graphical user interface. It is not required that all connectors referenced in a bus are connected. One bus may be connected to multiple other buses, and also to SSD connectors.

The example below contains a root system with two subsystems, WC1 and WC2. Bus connector WC1.bus1 is connected to WC2.bus2. Bus connector WC2.bus2 is also connected to SSD connector WC1.C3.

```
<?xml version="1.0" encoding="UTF-8"?>
<ssd:SystemStructureDescription name="Test" version="Draft20180219">
  <ssd:System name="Root">
    <ssd:Elements>
      <ssd:System name="WC2">
        <ssd:Connectors>
          <ssd:Connector name="C1" kind="input" type="Real"/>
          <ssd:Connector name="C2" kind="output" type="Real"/>
        </ssd:Connectors>
        <ssd:Annotations>
          <ssc:Annotation type="org.openmodelica">
            <oms:Bus name="bus2">
              <oms:Signals>
                <oms:Signal name="C1"/>
                <oms:Signal name="C2"/>
              </oms:Signals>
            </oms:Bus>
          </ssc:Annotation>
        </ssd:Annotations>
      </ssd:System>
      <ssd:System name="WC1">
        <ssd:Connectors>
          <ssd:Connector name="C1" kind="output" type="Real"/>
          <ssd:Connector name="C2" kind="input" type="Real"/>
```

(continues on next page)

(continued from previous page)

```
<ssd:Connector name="C3" kind="input" type="Real"/>
</ssd:Connectors>
<ssd:Annotations>
  <ssc:Annotation type="org.openmodelica">
    <oms:Bus name="bus1">
      <oms:Signals>
        <oms:Signal name="C1"/>
        <oms:Signal name="C2"/>
      </oms:Signals>
    </oms:Bus>
  </ssc:Annotation>
</ssd:Annotations>
</ssd:System>
</ssd:Elements>
<ssd:Connections>
  <ssd:Connection startElement="WC2" startConnector="C1"
    endElement="WC1" endConnector="C1"/>
  <ssd:Connection startElement="WC2" startConnector="C2"
    endElement="WC1" endConnector="C2"/>
  <ssd:Connection startElement="WC2" startConnector="C2"
    endElement="WC1" endConnector="C3"/>
</ssd:Connections>
<ssd:Annotations>
  <ssc:Annotation type="org.openmodelica">
    <oms:Connections>
      <oms:Connection startElement="WC1" startConnector="bus1"
        endElement="WC2" endConnector="bus2"/>
      <oms:Connection startElement="WC2" startConnector="bus2"
        endElement="WC1" endConnector="C3"/>
    </oms:Connections>
  </ssc:Annotation>
</ssd:Annotations>
</ssd:System>
</ssd:SystemStructureDescription>
```

## O

- OMEdit, [121](#)
- OMSimulator, [1](#)
  - Examples, [5](#)
  - Flags, [3](#)
- OMSimulatorLib, [5](#)
  - C-API, [7](#)
- OMSimulatorLua, [30](#)
  - Examples, [31](#)
  - Scripting Commands, [31](#)
- OMSimulatorPython, [51](#)
  - Examples, [53](#)
  - Scripting Commands, [54](#)
- OMSimulatorPython3, [76](#)
- OpenModelicaScripting, [106](#)
  - Examples, [107](#)
  - Scripting Commands, [107](#)

## S

- SSP, [127](#)
  - Bus connections, [129](#)