Chapter 2

Invoking omc – the OpenModelica Compiler/Interpreter Subsystem

The OpenModelica Compiler/Interpreter subsystem (omc) can be invoked in two ways:

- As a whole program, called at the operating-system level, e.g. as a command.
- As a server, called via a Corba client-server interface from client applications.

In the following we will describe these options in more detail.

2.1 Command-Line Invokation of the Compiler/Interpreter

The OpenModelica compilation subsystem is called omc (OpenModelica Compiler). The compiler can be given file arguments as specified below, and flags that are described in the subsequent sections.

`omc file.mo`	Return flat Modelica by code instantiating the last class in the file `file.mo`
`omc file.mof`	Put the flat Modelica produced by code instantiation of the last class within `file.mo` in the file named `file.mof`.
`omc file.mos`	Run the Modelica script file called `file.mos`.
`omc`	Calling omc with no parameters will display the help:

```
 $ ./omc
OpenModelica Compiler 1.8.1 (r11525) Copyright Linkoping University 1997-2012
Distributed under OMSC-PL and GPL, see www.openmodelica.org

Usage: omc [-runtimeOptions +omcOptions] (Model.mo | Script.mos) [Libraries | .mo-files]
* Libraries: Fully qualified names of libraries to load before processing Model or Script.
*            The libraries should be separated by spaces: Lib1 Lib2 ... LibN.
* runtimeOptions: call omc -help to see runtime options
* omcOptions:
  +d, +debug                 Sets debug flags. Use +help=debug to see available
                              flags.
      +help                  Displays the help text.
                               Valid options: debug, optmodules
      +running-testsuite     Used when running the testsuite.
 ++v, +version               Print the version and exit.
      +target                Sets the target compiler to use.
                               Valid options: gcc, msvc
  +g, +grammar               Sets the grammar and semantics to accept.
                               Valid options: Modelica, MetaModelica, ParModelica
      +annotationVersion     Sets the annotation version that should be used.
                               Valid options: 1.x, 2.x, 3.x
      +std                   Sets the language standard that should be used.
                               Valid options: 1.x, 2.x, 3.1, 3.2, 3.3
      +showErrorMessages     Show error messages immediately when they happen.
      +showAnnotations       Show annotations in the flattened code.
      +noSimplify            Do not simplify expressions if set.
      +preOptModules         Sets the pre optimisation modules to use in the
                              back end. See +help=optmodules for more info.
                               Valid options:
                                * removeSimpleEquationsFast
                                * removeSimpleEquations
                                * inlineArrayEqn
                                * removeFinalParameters
```

```
                                * removeEqualFunctionCalls
                                * removeProtectedParameters
                                * removeUnusedParameter
                                * removeUnusedVariables
                                * partitionIndependentBlocks
                                * collapseIndependentBlocks
                                * expandDerOperator
                                * residualForm

     +indexReductionMethod  Sets the index reduction method to use.
                              Valid options:
                               * dummyDerivative
                               * DynamicStateSelection

     +postOptModules        Sets the post optimisation modules to use in the
                             back end. See +help=optmodules for more info.
                              Valid options:
                               * lateInline
                               * removeSimpleEquationsFast
                               * removeSimpleEquations
                               * removeEqualFunctionCalls
                               * inlineArrayEqn
                               * removeUnusedParameter
                               * constantLinearSystem
                               * dumpComponentsGraphStr

     +simCodeTarget         Sets the target language for the code generation
                              Valid options: CSharp, Cpp, Adevs, QSS, C, c, Dump
     +orderConnections      Orders connect equations alphabetically if set.
  +t, +typeinfo             Prints out extra type information if set.
  +a, +keepArrays           Sets whether to split arrays or not.
  +m, +modelicaOutput
  +p, +paramsStruct
  +q, +silent               Turns on silent mode.
  +c, +corbaSessionName     Sets the name of the corba session if
                             +d=interactiveCorba is used.
  +n, +numProcs             Sets the number of processors to use.
  +l, +latency              Sets the latency for parallel execution.
  +b, +bandwidth            Sets the bandwidth for parallel execution.
  +i, +instClass            Instantiate the class given by the fully qualified
                             path.
  +v, +vectorizationLimit   Sets the vectorization limit, arrays and matrices
                             larger than this will not be vectorized.
  +s, +simulationCg         Turns on simulation code generation.
     +evalAnnotationParams  Sets whether to evaluate parameters in annotations
                             or not.
     +generateLabeledSimCode Turns on labeled SimCode generation for reduction
                             algorithms.
     +reduceTerms           Turns on reducing terms for reduction algorithms.
     +reductionMethod       Sets the reduction method to be used.
                              Valid options: deletion, substitution, linearization
     +plotSilent            Defines whether plot commands should open OMPlot
                             or just output results.

* Examples:
       omc Model.mo             will produce flattened Model on standard output
       omc +s Model.mo          will produce simulation code for the model:
                                 * Model.c           the model C code
                                 * Model_functions.c the model functions C code
                                 * Model.makefile    the makefile to compile the model.
                                 * Model_init.xml    the initial values
       omc Script.mos           will run the commands from Script.mos
       omc Model.mo Modelica    will first load the Modelica library and then produce
                                flattened Model on standard output
       omc Model1.mo Model2.mo  will load both Model1.mo and Model2.mo, and produce
                                flattened Model1 on standard output
       *.mo (Modelica files)
       *.mos (Modelica Script files)
```

2.1.1 General Compiler Flags

The following are general flags for uses not specifically related to debugging or tracing:

`omc +s file.mo/.mof`	Generate simulation code for the model last in `file.mo` or `file.mof`. The following files are generated: `modelname.cpp`, `modelname.h`, `modelname_init.txt`, `modelname.makefile`.
`omc +q`	Quietly run the compiler, no output to stdout.
`omc +d=blt`	Perform BLT transformation of the equations.
`omc +d=interactive`	Run the compiler in interactive mode with Socket communication. This functionality is depreciated and is replaced by the newer Corba communication module, but still useful in some cases for debugging communication. This flag only works under Linux and Cygwin.
`omc +d=interactiveCorba`	Run the compiler in interactive mode with Corba communication. This is the standard communication that is used for the interactive mode.
`omc +i=classpath`	Instantiates the class given by the fully qualified path classpath, instead of the last class in the file as default.
`omc ++v`	Returns the version number of the OMC compiler.

2.1.1.1 Example of Generating Stand-alone Simulation Code

To run `omc` from the command line and generate simulation code use the following flag:

```
omc +s model.mo
```

Currently the classloader does not load packages from MODELICAPATH automatically, so the .mo file must contain all used classes, i.e., a “total model” must be created.

Once you have generated the C code (and makefile, etc.) you can compile the model using

```
make –f modelname.makefile
```

2.1.2 Compiler Debug Trace Flags

Run omc with a comma separated list of flags without spaces,

```
"omc +d=flg1,flg2,..."
```

Here `flg1`,`flg2`,... are one of the flag names in the leftmost column of the flag description below. The special flag named `all` turns on all flags.

A debug trace printing is turned on by giving a flag name to the print function, like:

```
Debug.fprint("li", "Lookup information:...")
```

If omc is run with the following:

```
omc +d=foo,li,bar, ...
```

this line will appear on stdout, otherwise not. For backwards compatibility for debug prints not yet sorted out, the old debug print call:

```
Debug.print
```

has been changed to a call like the following:

```
Debug.fprint("olddebug",...)
```

Thus, if `omc` is run with the debug flag `olddebug` (or `all`), these messages will appear. The calls to `Debug.print` should eventually be changed to appropriately flagged calls.

Moreover, putting a `"-"` in front of a flag turns off that flag, i.e.:

```
omc +d=all,-dump
```

This will turn on all flags except `dump`.

Using Graphviz for visualization of abstract syntax trees, can be done by giving one of the graphviz flags, and redirect the output to a file. Then run `"dot -Tps filename -o filename.ps"` or `"dotty filename"`.

The following is a short description of all available debug trace flags. There is less of a need for some of these flags now when the recently developed interactive debugger with a data structure viewer is available.

- All debug tracing

`all`	Turn on all debug tracing.
`none`	This flag has default value true if no flags are given.

- General

`info`	General information.
`olddebug`	Print messages sent to the old `Debug.print`

- Dump

`parsedump`	Dump the parse tree.
`dump`	Dump the absyn tree.
`dumpgraphviz`	Dump the absyn tree in graphviz format.
`daedump`	Dump the DAE in printed form.
`daedumpgraphv`	Dump the DAE in graphviz format.
`daedumpdebug`	Dump the DAE in expression form.
`dumptr`	Dump trace.
`beforefixmodout`	Dump the PDAE in expression form before moving the modification equations into the `VAR` declarations.

- Types

`tf`	Types and functions.
`tytr`	Type trace.

- Lookup

`li`	Lookup information.
`lotr`	Lookup trace.
`locom`	Lookup compare.

- Static

`sei`	Information
`setr`	Trace

- SCode

`ecd`	Trace of `elab_classdef`.

- Instantiation

`insttr`	Trace of code instantiation.

- Env

`envprint`	Dump the environment at each class instantiation.
`envgraph`	Same as envprint, but using graphviz.
`expenvprint`	Dump environment at equation elaboration.

expenvgraph dump environment at equation elaboration.

2.1.2.1 Example of Generating Log Information

```
$ omc +s +simCodeTarget=Dump MyModel.mo Modelica +i=My.Model.Name
```

Prints a log like:

```
when (#2): time > 0.5
i = integer(r)
[a.mo:7:5-7:19]
partOfLst: within M;
instanceOptLst:
connectEquationOptLst:
typeLst:
operations (0):
```

2.1.3 Compiler Optimization Flags

A simplified schema of the back end optimization phases is sketched in Figure 2-1.

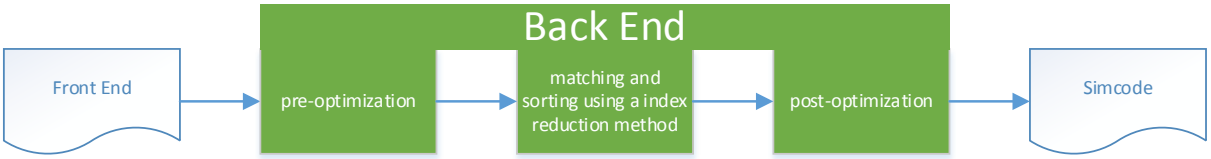

Figure 2-1: simplified schema of the back end optimization phases

In both the pre-optimization phase and post-optimization phase several optimization modules are applied one after another. Hence, the order is highly fundamental for this process.

2.1.3.1 Pre Optimization Modules

Run omc with a comma separated list of flags without spaces to specify a custom pre-optimization phase:

```
"omc +preOptModules=flg1,flg2,…"
```

Here `flg1,flg2,...` are one of the flag names in the leftmost column of the flag description. The flag description can be generated using `"omc +help=preOptModules"`. The following is the list of modules that are used per default:

```
"omc +preOptModules=evaluateReplaceFinalEvaluateParameters,
simplifyIfEquations,removeEqualFunctionCalls,partitionIndependentBlocks,
expandDerOperator,findStateOrder,replaceEdgeChange,inlineArrayEqn,
removeSimpleEquations"
```

Valid options are:

```
 * removeSimpleEquations
 * removeAllSimpleEquations
 * inlineArrayEqn
 * evaluateFinalParameters
 * evaluateEvaluateParameters
 * evaluateFinalEvaluateParameters
 * evaluateReplaceFinalParameters
 * evaluateReplaceEvaluateParameters
```

```
* evaluateReplaceFinalEvaluateParameters
* removeEqualFunctionCalls
* removeProtectedParameters
* removeUnusedParameter
* removeUnusedVariables
* partitionIndependentBlocks
* collapseIndependentBlocks
* expandDerOperator
* simplifyIfEquations
* replaceEdgeChange
* residualForm
* addInitialStmtsToAlgorithms
```

2.1.3.2 Transformation Phase

This phase combines matching and sorting using an index reduction method. Therefore, the following flags can be used:

cheapmatchingAlgorithm

```
"omc +cheapmatchingAlgorithm=0"
```

Sets the cheap matching algorithm to use. A cheap matching algorithm gives a jump start matching by heuristics. Valid options are:

```
* 0
* 1
* 3
```

matchingAlgorithm

```
"omc +matchingAlgorithm=PFPlus"
```

Sets the matching algorithm to use. See +help=optmodules for more info. Valid options are:

```
* BFSB
* DFSB
* MC21A
* PF
* PFPlus
* HK
* HKDW
* ABMP
* PR
* DFSBExt
* BFSBExt
* MC21AExt
* PFExt
* PFPlusExt
* HKExt
* HKDWExt
* ABMPExt
* PRExt
```

matchingAlgorithm

```
"omc +indexReductionMethod=uode"
```

Sets the index reduction method to use. See +help=optmodules for more info. Valid options are:
* uode
* dynamicStateSelection

2.1.3.3 Post Optimization Modules

Run omc with a comma separated list of flags without spaces to specify a custom post-optimization phase:

```
"omc +postOptModules=flg1,flg2,…"
```

Here flg1,flg2,... are one of the flag names in the leftmost column of the flag description. The flag description can be generated using "omc +help=postOptModules". The following is the list of modules that are used per default:

```
"omc +postOptModules=relaxSystem,inlineArrayEqn,constantLinearSystem,
simplifysemiLinear,removeSimpleEquations,encapsulateWhenConditions,
tearingSystem,countOperations,removeUnusedFunctions,inputDerivativesUsed,
detectJacobianSparsePattern,removeConstants"
```

Valid options are:

```
* encapsulateWhenConditions
* lateInlineFunction
* removeSimpleEquationsFast
* removeSimpleEquations
* evaluateFinalParameters
* evaluateEvaluateParameters
* evaluateFinalEvaluateParameters
* evaluateReplaceFinalParameters
* evaluateReplaceEvaluateParameters
* evaluateReplaceFinalEvaluateParameters
* removeEqualFunctionCalls
* inlineArrayEqn
* removeUnusedParameter
* constantLinearSystem
* tearingSystem
* relaxSystem
* countOperations
* dumpComponentsGraphStr
* generateSymbolicJacobian
* generateSymbolicLinearization
* collapseIndependentBlocks
* removeUnusedFunctions
* simplifyTimeIndepFuncCalls
* inputDerivativesUsed
* simplifysemiLinear
* removeConstants
* optimizeInitialSystem
* detectJacobianSparsePattern
* partitionIndependentBlocks
* addInitialStmtsToAlgorithms
```

2.1.4 Simulation Initialization Flags

You can use different initialization configurations by the following simulation flags:

"-iim state": [default] normal initialization will be performed
"-iim none": no initialization will be performed –> start values are used
"-iom nelder_mead_ex": [default] the new optimization method will be used –> global homotopy
"-iom nelder_mead_ex2": the new optimization method will be used –> without global homotopy
"-iom simplex": the old simplex-initialization from OM1.8.0 will be used, just with the new event handling and so on.
"-iif <matfile>": imports start values for all variables from given matlab-file (time 0.0 or –iit <time>)
"-iit <time>": point in time for iif flag above

2.2 The OpenModelica Client-Server Architecture

The OpenModelica client-server architecture is schematically depicted in Figure 2-2, showing two typical clients: a graphic model editor and an interactive session handler for command interpretation.

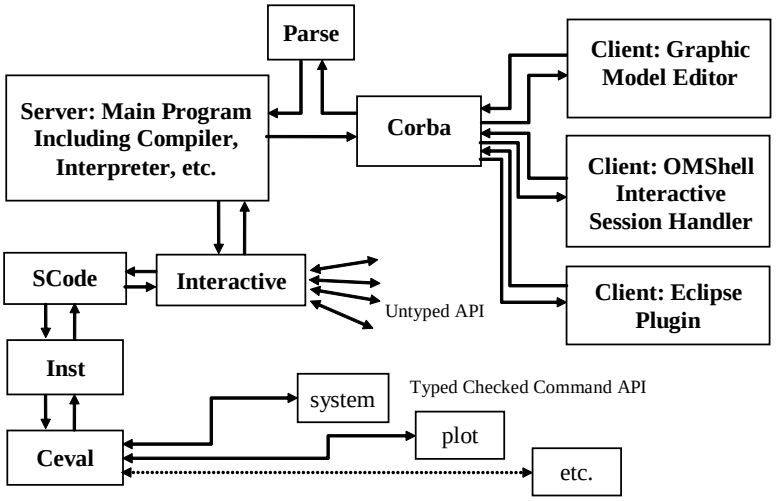

Figure 2-2. Client-Server interconnection structure of the compiler/interpreter main program and interactive tool interfaces. Messages from the Corba interface are of two kinds. The first group consists of expressions or user commands which are evaluated by the Ceval module. The second group are declarations of classes, variables, etc., assignments, and client-server API calls that are handled via the Interactive module, which also stores information about interactively declared/assigned items at the top-level in an environment structure.

The SCode module simplifies the Absyn representation, public components are collected together, protected ones together, etc. The Interactive modul serves the untyped API, updates, searches, and keeps the abstract syntax representation. An environment structure is not kept/cached, but is built by Inst at each call. Call Inst for more exact instantion lookup in certain cases. The whole Absyn AST is converted into Scode when something is compiled, e.g. converting the whole standard library if something.

Commands or Modelica expressions are sent as text from the clients via the Corba interface, parsed, and divided into two groups by the main program:

- All kinds of declarations of classes, types, functions, constants, etc., as well as equations and assignment statements. Moreover, function calls to the untyped API also belong to this group – a

function name is checked if it belongs to the API names. The Interactive module handles this group of declarations and untyped API commands.

- Expressions and type checked API commands, which are handled by the Ceval module.

The reason the untyped API calls are not passed via SCode and Inst to Ceval is that Ceval can only handle typed calls – the type is always computed and checked, whereas the untyped API prioritizes performance and typing flexibility. The Main module checks the name of a called function name to determine if it belongs to the untyped API, and should be routed to Interactive.

Moreover, the Interactive module maintains an environment of all interactively given declarations and assignments at the top-level, which is the reason such items need to be handled by the Interactive module.

2.3 Client-Server Type-Checked Command API for Scripting

The following are short summaries of typed-checked scripting commands/ interactive user commands for the OpenModelica environment.

The emphasis is on safety and type-checking of user commands rather than high performance run-time command interpretation as in the untyped command interface described in Section 2.4.

These commands are useful for loading and saving classes, reading and storing data, plotting of results, and various other tasks.

The arguments passed to a scripting function should follow syntactic and typing rules for Modelica and for the scripting function in question. In the following tables we briefly indicate the types or character of the formal parameters to the functions by the following notation:

- `String` typed argument, e.g. `"hello"`, `"myfile.mo"`.
- `TypeName` – class, package or function name, e.g. `MyClass`, `Modelica.Math`.
- `VariableName` – variable name, e.g. `v1`, `v2`, `vars1[2].x`, etc.
- `Integer` or `Real` typed argument, e.g. `35`, `3.14`, `xintvariable`.
- `options` – optional parameters with named formal parameter passing.

The following are brief descriptions of the most common scripting commands available in the OpenModelica environment. Se also some example calls in the file

`animate``(className, options)` (NotYetImplemented)	Display a 3D visaulization of the latest simulation. *Inputs*: `TypeName className;` *Outputs*: `Boolean res;`
`cd``(dir)`	Change directory. *Inputs*: `String dir;` *Outputs*: `Boolean res;`
`cd``()`	Return current working directory. *Outputs*: `String res;`
`checkModel``(className)` (NotYetImplemented)	Instantiate model, optimize equations, and report errors. *Inputs*: `TypeName className;` *Outputs*: `Boolean res;`
`clear``()`	Clears everything: symboltable and variables. *Outputs*: `Boolean res;`
`clearClasses``()` (NotYetImplemented)	Clear all class definitions from symboltable. *Outputs*: `Boolean res;`
`clearLog``()` (NotYetImplemented)	Clear the log. *Outputs*: `Boolean res;`
`clearVariables``()`	Clear all user defined variables. *Outputs*: `Boolean res;`
`closePlots``()`(NotYetImplemented)	Close all plot windows. *Outputs*: `Boolean res;`
`getLog``()`(NotYetImplemented)	Return log as a string. *Outputs*: `String log;`
`instantiateModel``(className)`	Instantiate model, resulting in a `.mof` file of flattened Modelica. *Inputs*: `TypeName className;` *Outputs*: `Boolean res;`
`list``(className)`	Print class definition. *Inputs*: `TypeName className;` *Outputs*: `String classDef;`
`list``()`	Print all loaded class definitions. *Output*: `String classdefs;`
`listVariables``()`	Print user defined variables. *Outputs*: `VariableName res;`
`loadFile``(fileName)`	Load models from file. *Inputs*: `String fileName` *Outputs*: `Boolean res;`
`loadModel``(className)`	Load the file corresponding to the class, using the Modelica class name-to-file-name mapping to locate the file. *Inputs*: `TypeName className` *Outputs*: `Boolean res;`
`plot``(variables, options)`	Plots vars, which is a vector of variable names.

	Inputs: `VariableName variables; String title; Boolean legend; Boolean gridLines; Real xrange[2]` i.e. `{xmin,xmax}; Real yrange[2]` i.e. `{ymin,ymax};` *Outputs*: `Boolean res;`
plot(var, options)	Plots variable with name `var`. *Inputs*: `VariableName var; String title; Boolean legend; Boolean gridLines; Real xrange[2]` i.e. `{xmin,xmax}; Real yrange[2]` i.e. `{ymin,ymax};` *Outputs*: `Boolean res;`
plotParametric(vars1, vars2, options)	Plot each pair of corresponding variables from the vectors of variables `vars1`, `vars2` as a parametric plot. *Inputs*: `VariableName vars1[:]; VariableName vars2[size(variables1,1)]; String title; Boolean legend; Boolean gridLines; Real range[2,2];` *Outputs*: `Boolean res;`
plotParametric(var1, var2, options)	Plot the variable `var2` against `var1` as a parametric plot. *Inputs*: `VariableName var1; VariableName var2; String title; Boolean legend; Boolean gridLines; Real range[2,2];` *Outputs*: `Boolean res;`
plotVectors(v1, v2, options) (??NotYetImplemented)	Plot vectors `v1` and `v2` as an x-y plot. *Inputs*: `VariableName v1; VariableName v2;` *Outputs*: `Boolean res;`
readMatrix(fileName, matrixName) (??NotYetImplemented)	Read a matrix from a file given filename and matrixname. *Inputs*: `String fileName; String matrixName;` *Outputs*: `Boolean matrix[:,:];`
readMatrix(fileName, matrixName, nRows, nColumns) (??NotYetImplemented)	Read a matrix from a file, given file name, matrix name, #rows and #columns. *Inputs*: `String fileName; String matrixName; int nRows; int nColumns;` *Outputs*: `Real res[nRows,nColumns];`
readMatrixSize(fileName, matrixName) (??NotYetImplemented)	Read the matrix dimension from a *file* given a *matrix name*. *Inputs*: `String fileName; String matrixName;` *Outputs*: `Integer sizes[2];`
readSimulationResult(fileName, variables, size)	Reads the simulation result for a list of variables and returns a matrix of values (each column as a vector or values for a variable.) Size of result is also given as input. *Inputs*: `String fileName; VariableName variables[:]; Integer size;` *Outputs*: `Real res[size(variables,1),size)];`
readSimulationResultSize(fileName) (??NotYetImplemented)	Read the size of the trajectory vector from a file. *Inputs*: `String fileName;` *Outputs*: `Integer size;`
runScript(fileName)	Executes the script file given as argument. *Inputs*: `String fileName;` *Outputs*: `Boolean res;`
saveLog(fileName) (??NotYetImplemented)	Save the log to a file. *Inputs*: `String fileName;` *Outputs*: `Boolean res;`
saveModel(fileName, className) (NotYetImplemented)	Save class definition in a file. *Inputs*: `String fileName; TypeName className` *Outputs*: `Boolean res;`
save(className)	Save the model (A1) into the file it was loaded from.

	Inputs: `TypeName className`
saveTotalModel(fileName, className) (??NotYetImplemented)	Save total class definition into file of a class. *Inputs*: `String fileName; TypeName className` *Outputs*: `Boolean res;`
simulate(className, options)	Simulate model, optionally setting simulation values. *Inputs*: `TypeName className; Real startTime; Real stopTime; Integer numberOfIntervals; Real outputInterval; String method; Real tolerance; Real fixedStepSize; String outputFormat;` *Outputs*: `SimulationResult simRes;`
system(fileName)	Execute system command. *Inputs*: `String fileName;` *Outputs*: `Integer res;`
translateModel(className) (??NotYetImplemented)	Instantiate model, optimize equations, and generate code. *Inputs*: `TypeName className;` *Outputs*: `SimulationObject res;`
writeMatrix(fileName, matrixName, matrix) (??NotYetImplemented)	Write matrix to file given a matrix name and a matrix. *Inputs*: `String fileName; String matrixName; Real matrix[:,:];` *Outputs*: `Boolean res;`

2.3.1 Examples

The following session in OpenModelica illustrates the use of a few of the above-mentioned functions.

```
>> model test Real x; end test;
  Ok
>> s:=list(test);
>> s
"model test
  Real x;
equation
  der(x)=x;
end test;
"
>> instantiateModel(test)
"fclass test
Real x;
equation
  der(x) = x;
end test;
"
>> simulate(test)
record
    resultFile = "C:\OpenModelica1.6.0\test_res.plt"
 end record

>> a:=1:10
  {1,2,3,4,5,6,7,8,9,10}
>> a*2
  {2,4,6,8,10,12,14,16,18,20}
>> clearVariables()
  true
>> list(test)
"model test
  Real x;
equation
  der(x)=x;
```

```
end test;
"
>> clear()
  true
>> list()
  {}
```

The common combination of a simulation followed by a plot:

```
> simulate(mycircuit, stopTime=10.0);
> plot({R1.v});
```

There are several output format possibilities. “plt” is default, and plt is currently the only format capable of using val() or plot() functions. csv (comma separated values) is roughly twice as fast on data-heavy simulations, and doesn't require all output data allocated in RAM during simulation. Empty does no output at all and should be by far the fastest.

```
simulate(... , outputFormat="csv")
simulate(... , outputFormat="plt")
simulate(... , outputFormat="empty")
```

2.4 Client-Server Untyped High Performance API for Model Query

The following API is primarily designed for clients calling the OpenModelica compiler/interpreter via the Corba (or socket) interface to obtain information about and manipulate the model structure, but the functions can also be invoked directly as user commands and/or scripting commands. The API has the following general properties:

- Untyped, no type checking is performed. The reason is high performance, low overhead per call.
- All commands are sent as strings in Modelica syntax; all results are returned as strings.
- Polymorphic typed commands. Commands are internally parsed into Modelica Abstract syntax, but in a way that does not enforce uniform typing (analogous to what is allowed for annotations). For example, vectors such as {true, 3.14, "hello"} can be passed even though the elements have mixed element types, here (`Boolean`, `Real`, `String`), which is currently not allowed in the Modelica type system.

The API for interactive/incremental development consist of a set of Modelica functions in the Interactive module. Calls to these functions can be sent from clients to the interactive environment as plain text and parsed using an expression parser for Modelica. Calls to this API are parsed and routed from the Main module to the Interactive module if the called function name is in the set of names in this API. All API functions return strings, e.g. if the value true is returned, the text `"true"` will be sent back to the caller, but without the string quotes.

- When a function fails to perform its action the string `"-1"` is returned.
- All results from these functions are returned as strings (without string quotes).

The API can be used by human users when interactively building models, directly, or indirectly by using scripts, but also by for instance a model editor who wants to interact with the symbol table for adding/changing/removing models and components, etc.

(??Future extension: Also describe corresponding internal calls from within OpenModelica)

2.4.1 Definitions

An	Argument no. n, e.g. A1 is the first argument, A2 is the second, etc.
<ident>	Identifier, e.g. A or Modelica.
<string>	Modelica string, e.g. "Nisse" or "foo".
<expr>	Arbitrary Modelica expression..
<cref>	Class reference, i.e. the name of a class, e.g. Resistor.

2.4.2 Examples of Calls

Calls fulfill the normal Modelica function call syntax. For example:

```
saveModel("MyResistorFile.mo",MyResistor)
```

will save the model MyResistor into the file "MyResistorFile.mo".

For creating new models it is most practical to send a model declaration to the API, since the API also accepts Modelica declarations and Modelica expressions. For example, sending:

```
model Foo end Foo;
```

will create an empty model named Foo, whereas sending:

```
connector Port end Port;
```

will create a new empty connector class named Port.

Many more API example calls can be found in the OMNotebook file ModelQueryAPIexamples.onb in the OpenModelica testmodels directory.

2.4.3 Untyped API Functions for Model Query and Manipulation

The following are brief descriptions of the untyped API functions available in the OpenModelica environment for obtaining information about models and/or manipulate models. API calls are decoded by evaluateGraphicalApi and evaluateGraphicalApi2 in the Interactive package. Results from a call are returned as as a text string (without the string delimiters ""). The functions in the typed API (Section 2.3) are handled by the Ceval package.

Executable example calls to these functions are available in the file ModelQueryAPIexample.onb in the OpenModelica testmodels directory.

Additional, more extensive documentation with examples, including some functions not mentioned below, is available in the separate file OMC_API-HowTo.pdf.

--- Source Files ---	
getSourceFile (A1<string>)	Gets the source file of the class given as argument (A1).
setSourceFile (A1<string>, A2<string>)	Associates the class given as first argument (A1) to a source file given as second argument (A2)
--- Environment Variables ---	
getEnvironmentVar(A1<string>)	Retrieves an evironment variable with the specified name.
setEnvironmentVar(A1<string>, A2<string>)	Sets the environment variable with the specified name (A1) to a given value (A2).
--- Classes and Models **---**	
loadFile(A1<string>)	Loads all models in the file. Also in typed API. Returns list of names of top level classes in the loaded files.
loadFileInteractiveQualified	Loads all models in the file. Also in typed API. Returns list of

(A1<string>)	qualified names of top level classes in the loaded files.
loadFileInteractive(A1<string>)	Loads the file given as argument into the compiler symbol table. ??What is the difference to loadFile??
loadModel(A1<cref>)	Loads the model (A1) by looking up the correct file to load in $OPENMODELICALIBRARY. Loads all models in that file into the symbol table.
saveModel(A1<string>,A2<cref>)	Saves the model (A2) in a file given by a string (A1). This call is also in typed API. NOTE: ?? Not yet completely implemented.
save(A1<cref>)	Saves the model (A1) into the file it was previously loaded from. This call is also in typed API.
deleteClass(A1<cref>)	Deletes the class from the symbol table.
renameClass(A1<cref>, A2<cref>)	Renames an already existing class with *from_name* A1 to *to_name* (A2). The rename is performed recursively in all already loaded models which reference the class A1. NOTE: ??The implementation is currently buggy/very slow.
--- Class Attributes ---	
getElementsInfo(A1<cref>)	Retrieves the Info attribute of all elements within the given class (A1). This contains information of the element type, filename, isReadOnly, line information, name etc., in the form of a vector containing element descriptors on record constructor form rec(...), e.g.: "{rec(attr1=value1, attr2=value2 ...), ..., rec(attr1=value1, attr2=value2 ...)}"
setClassComment(A1<cref>,A2<string>)	Sets the class (A1) string comment (A2).
addClassAnnotation(A1<cref>, annotate=<expr>)	Adds annotation given by A2(in the form annotate= classmod(...)) to the model definition referenced by A1. Should be used to add Icon Diagram and Documentation annotations.
getIconAnnotation(A1<cref>)	Returns the Icon Annotation of the class named by A1.
getDiagramAnnotation(A1<cref>)	Returns the Diagram annotation of the class named by A1. NOTE1: Since the Diagram annotations can be found in base classes a partial code instantiation is performed that flattens the inheritance hierarchy in order to find all annotations. NOTE2: Because of the partial flattening, the format returned is not according the Modelica standard for Diagram annotations.
getPackages(A1<cref>)	Returns the names of all Packages in a class/package named by A1 as a list, e.g.: {Electrical,Blocks,Mechanics, Constants,Math,SIunits}
getPackages()	Returns the names of all package definitions in the global scope.
getClassNames(A1<cref>)	Returns the names of all class defintions in a class/package.
getClassNames()	Returns the names of all class definitions in the global scope.
getClassNamesForSimulation()	Returns a list of all “open models” in client that are candidates for simulation.

setClassNamesForSimulation(A1<string>)	Set the list of all “open models” in client that are candidates for simulation. The string must be on format: “{model1,model2,model3}”
getClassAttributes(A1<cref>)	Returns all the possible class information in the following form: `rec( attr1=value1, attr2=value2 ... )`
getClassRestriction(A1<cref>)	Returns the kind of restricted class of <cref>, e.g. `"model"`, `"connector"`, `"function"`, `"package"`, etc.
getClassInformation(A1<cref>)	Returns a list of the following information about the class A1: {"restriction","comment","filename.mo",{bool,bool,bool},{"readonly|writable",int,int,int,int}}
--- Restricted Class Predicates	
isPrimitive(A1<cref>)	Returns `"true"` if class is of primitive type, otherwise `"false"`.
isConnector(A1<cref>)	Returns `"true"` if class is a connector, otherwise `"false"`.
isModel(A1<cref>)	Returns `"true"` if class is a model, otherwise `"false"`.
isRecord(A1<cref>)	Returns `"true"` if class is a record, otherwise `"false"`.
isBlock(A1<cref>)	Returns `"true"` if class is a block, otherwise `"false"`.
isType(A1<cref>)	Returns `"true"` if class is a type, otherwise `"false"`.
isFunction(A1<cref>)	Returns `"true"` if class is a function, otherwise `"false"`.
isPackage(A1<cref>)	Returns `"true"` if class is a package, otherwise `"false"`.
isClass(A1<cref>)	Returns `"true"` if A1 is a class, otherwise `"false"`.
isParameter(A1<cref>)	Returns `"true"` if A1 is a parameter, otherwise `"false"`. NOTE: ??Not yet implemented.
isConstant(A1<cref>)	Returns `"true"` if A1 is a constant, otherwise `"false"`. NOTE: ??Not yet implemented.
isProtected(A1<cref>)	Returns `"true"` if A1 is protected, otherwise `"false"`. NOTE: ??Not yet implemented.
existClass(A1<cref)	Returns `"true"` if class exists in symbolTable, otherwise `"false"`.
--- Components ---	
getComponents(A1<cref>)	Returns a list of the component declarations within class A1: `"{{Atype,varidA,"commentA"},{Btype,varidB,"commentB"}, {...}}"`
setComponentProperties(A1<cref>, A2<cref>, A3<Boolean>, A4<Boolean>, A5<Boolean>, A6<Boolean>, A7<String>, A8<{Boolean, Boolean}>, A9<String>)	Sets the following properties of a component (A2) in a class (A1). - A3 final (true/false) - A4 flow (true/false) - A5 protected(true) or public(false) - A6 replaceable (true/false) - A7 variablity: `"constant”` or `"discrete"` or `"parameter"` or `""` - A8 dynamic_ref: {inner, outer} - two boolean values.

	- A9 causality: "input" or "output" or ""
getComponentAnnotations(A1<cref>)	Returns a list {...} of all annotations of all components in A1, in the same order as the components, one annotation per component.
getCrefInfo(A1<cref>)	Gets the component reference file and position information. Returns a list: {file, readonly|writable, start line, start column, end line, end column} >> getCrefInfo(BouncingBall) {C:/OpenModelica1.4.1/testmodels/BouncingBall.mo,writable,1,1,20,17}
addComponent(A1<ident>,A2<cref>, A3<cref>,annotate=<expr>)	Adds a component with name (A1), type (A2), and class (A3) as arguments. Optional annotations are given with the named argument annotate.
deleteComponent(A1<ident>, A2<cref>)	Deletes a component (A1) within a class (A2).
updateComponent(A1<ident>, A2<cref>, A3<cref>,annotate=<expr>)	Updates an already existing component with name (A1), type (A2), and class (A3) as arguments. Optional annotations are given with the named argument annotate.
renameComponent(A1<cref>, A2<ident>, A3<ident>)	Renames an already existing component with name A2 defined in a class with name (A1), to the new name (A3). The rename is performed recursively in all already loaded models which reference the component declared in class A2. NOTE: ??The implementation is currently buggy/very slow.
getNthComponentAnnotation(A1<cref>,A2<int>)	Returns the flattened annotation record of the nth component (A2) (the first is has no 1) within class/component A1. Consists of a comma separated string of 15 values, see Annotations in Section 2.4.4 below, e.g "false,10,30,..."
getNthComponentModification(A1<cref>,A2<int>)	Returns the modification of the nth component (A2) where the first has no 1) of class/component A1.
getComponentModifierValue(A1<cref>, A2<cref)	Returns the value of a component (e.g. variable, parameter, constant, etc.) (A2) in a class (A1).
setComponentModifierValue(A1<cref>, A2<cref>,A3<exp>)	Sets the modfier value of a component (e.g. variable, parameter, constant, etc.) (A2) in a class (A1) to an expression (unevaluated) in A3.
getComponentModifierNames(A1<cref>, A2<cref>)	Retrieves the names of ?? all components in the class.
--- Inheritance ---	
getInheritanceCount(A1<cref>)	Returns the number (as a string) of inherited classes of a class.
getNthInheritedClass(A1<cref>, A2<int>)	Returns the type name of the nth inherited class of a class. The first class has number 1.
getExtendsModifierNames(A1<cref>)	Return the modifier names of a modification on an extends clause. For instance: "model test extends A(p1=3,p2(z=3)); end test;" getExtendsModifierNames(test,A) => {p1,p2}
getExtendsModifierValue(A1<cref>)	Return the submodifier value of an extends clause for

	instance, `"model test extends A(p1=3,p2(z=3));end test;" getExtendsModifierValue(test,A,p1)` => =3
`--- Connections ---`	
`getConnectionCount``(A1<cref>)`	Returns the number (as a string) of connections in the model.
`getNthConnection``(A1<cref>, A2<int>)`	Returns the nth connection, as a comma separated pair of connectors, e.g. `"R1.n,R2.p"`. The first has number 1.
`getNthConnectionAnnotation``( A1<cref>,A2<int>)`	Returns the nth connection annotation as comma separated list of values of a flattened record, see Annotations in Section 2.4.4 below.
`addConnection``(A1<cref>,A2<cref>, A3<cref>, annotate=<expr>)`	Adds connection `connect(A1,A2)` to model `A3`, with annotation given by the named argument `annotate`.
`updateConnection``(A1<cref>, A2<cref>,A3<cref>, annotate=<expr>)`	Updates an already existing connection.
`deleteConnection``(A1<cref>, A2<cref>,A3<cref>)`	Deletes the connection `connect(A1,A2)` in class given by `A3`.
`--- Equations ---`	
`addEquation``(A1<cref>,A2<expr>, A3<expr>)`(??NotYetImplemented)	Adds the equation `A2=A3` to the model named by A1.
`getEquationCount``(A1<cref>)` (??NotYetImplemented)	Returns the number of equations (as a string) in the model named `A1`. (This includes connections)
`getNthEquation``(A1<cref>,A2<int>)` (??NotYetImplemented)	Returns the nth (`A2`) equation of the model named by A1. e.g. `"der(x)=-1"` or `"connect(A.b,C.a)"`. The first has number 1.
`deleteNthEquation``(A1<cref>, A2<int>)`(??NotYetImplemented)	Deletes the nth (`A2`) equation in the model named by `A1`. The first has number 1.
`--- Misc ---`	
`checkSettings``()`	Improved version of getSettings(). Used for debugging a user's settings. It checks that a compiler is installed and working, that environment variables are set, which OS is used, and more.
`getVersion``()`	returns the OMC version, e.g. `"1.4.2"`
`getAstAsCorbaString``([filename=<String>])`	This command unparses the internal AST of all the loaded files as text using the Java CORBA format for uniontypes. If a filename is given, the text is dumped to that file instead of sent over CORBA. This is useful because you can save the internal AST on file for future use. If you have problems sending the large text over CORBA, you can also use the file as intermediate output to overcome bugs and limitations in CORBA or RML implementations on 32-bit platforms.
`dumpXMLDAE``(`*`modelname`*`[,asInSimulationCode=<Boolean>] [,filePrefix=<String>] [,storeInTemp=<Boolean>] [,addMathMLCode =<Boolean>])`	This command dumps the mathematical representation of a model using an XML representation, with optional parameters *Inputs:* TypeName className; Boolean asInSimulationCode; String filePrefix; Boolean storeInTemp; Boolean addMathMLCode; *Outputs:* String xmlFile In particular, `asInSimulationCode` defines where to stop in

	the translation process (before dumping the model), the other options are relative to the file storage: `filePrefix` for specifying a different name and `storeInTemp` to use the temporary directory. The optional parameter `addMathMLCode` gives the possibility to don't print the MathML code within the xml file, to make it more readable.Usage is trivial, just: `addMathMLCode`=true/false (default value is false). For an example, See Section 2.5.5.
`exportDAEtoMatlab`(*`modelname`*)	Dumps the incidence matrix of model in a Matlab format. See Section 2.5.6.
`setDebugFlags`(`A1 = <String>`)	Enables a debug flag in an interactive session. Useful to enable failtrace even if omc was not started with the flag set (most interactive clients start omc without any flag set).

2.4.3.1 ERROR Handling

When an error occurs in any of the above functions, the string `"-1"` is returned.

2.4.4 Annotations

Annotations can occur for several kinds of Modelica constructs.

2.4.4.1 Variable Annotations

Variable annotations (i.e., component annotations) are modifications of the following (flattened) Modelica record:

```
record Placement
  Boolean visible = true;
  Real transformation.x=0;
  Real transformation.y=0;
  Real transformation.scale=1;
  Real transformation.aspectRatio=1;
  Boolean transformation.flipHorizontal=false;
  Boolean transformation.flipVertical=false;
  Real transformation.rotation=0;
  Real iconTransformation.x=0;
  Real iconTransformation.y=0;
  Real iconTransformation.scale=1;
  Real iconTransformation.aspectRatio=1;
  Boolean iconTransformation.flipHorizontal=false;
  Boolean iconTransformation.flipVertical=false;
  Real iconTransformation.rotation=0;
end Placement;
```

2.4.4.2 Connection Annotations

Connection annotations are modifications of the following (flattened) Modelica record:

```
record Line
  Real points[2][:];
  Integer color[3]={0,0,0};
  enumeration(None,Solid,Dash,Dot,DashDot,DashDotDot) pattern = Solid;
  Real thickness=0.25;
  enumeration(None,Open,Filled,Half) arrow[2] = {None, None};
  Real arrowSize=3.0;
```

```
  Boolean smooth=false;
end Line;
```

This is the Flat record Icon, used for Icon layer annotations

```
record Icon
  Real  coordinateSystem.extent[2,2] = {{-10, -10}, {10, 10}});
  GraphicItem[:] graphics;
end Icon;
```

The textual representation of this flat record is somewhat more complicated, since the graphics vector can conceptually contain different subclasses, like `Line`, `Text`, `Rectangle`, etc. To solve this, we will use record constructor functions as the expressions of these. For instance, the following annotation:

```
annotation (
Icon(coordinateSystem={{-10,-10}, {10,10}},
graphics={Rectangle(extent={{-10,-10}, {10,10}}),
Text({{-10,-10}, {10,10}}, textString="Icon")}));
```

will produce the following string representation of the flat record `Icon`:

```
{{{-10,10},{10,10}},{Rectangle(true,{0,0,0},{0,0,0},
LinePattern.Solid,FillPattern.None,0.25,BorderPattern.None,
{{-10,-10},{10,10}},0),Text({{-10,-10},{10,10}},textString="Icon")}}
```

The following is the flat record for the `Diagram` annotation:

```
record Diagram
  Real  coordinateSystem.extent[2,2] = {{-10, -10}, {10, 10}});
  GraphicItem[:] graphics;
end Diagram;
```

The flat records string representation is identical to the flat record of the `Icon` annotation.

2.4.4.3 Flat records for Graphic Primitives

```
record Line
  Boolean visible = true;
  Real points[2,:];
  Integer color[3] = {0,0,0};
  LinePattern pattern = LinePattern.Solid;
  Real thickness = 0.25;
  Arrow arrow[2] = {Arrow.None, Arrow.None};
  Real arrowSize = 3.0;
  Boolean smooth = false;
end Line;

record Polygon
  Boolean visible = true;
  Integer lineColor[3]={0,0,0};
  Integer fillColor[3]={0,0,0};
  LinePattern pattern = LinePattern.Solid;
  FillPattern fillPattern = FillPattern.None;
  Real lineThickness = 0.25;
  Real points[2,:];
  Boolean smooth = false;
end Polygon;

record Rectangle
  Boolean visible=true;
  Integer lineColor[3]={0,0,0};
  Integer fillColor[3]={0,0,0};
  LinePattern pattern = LinePattern.Solid;
  FillPattern fillPattern = FillPattern.None;
  Real lineThickness = 0.25;
```

```
  BorderPattern borderPattern = BorderPattern.None;
  Real extent[2,2];
  Real radius;
end Rectangle;

record Ellipse
  Boolean visible = true;
  Integer lineColor[3]={0,0,0};
  Integer fillColor[3]={0,0,0};
  LinePattern pattern = LinePattern.Solid;
  FillPattern fillPattern = FillPattern.None;
  Real lineThickness = 0.25;
  Real extent[2,2];
end Ellipse;

record Text
  Boolean visible = true;
  Integer lineColor[3]={0,0,0};
  Integer fillColor[3]={0,0,0};
  LinePattern pattern = LinePattern.Solid;
  FillPattern fillPattern = FillPattern.None;
  Real lineThickness = 0.25;
  Real extent[2,2];
  String textString;
  Real fontSize;
  String fontName;
  TextStyle textStyle[:];
end Text;

record BitMap
  Boolean visible = true;
  Real extent[2,2];
  String fileName;
  String imageSource;
end BitMap;
```

2.5 Discussion on Modelica Standardization of the Typed Command API

An interactive function interface could be part of the Modelica specification or Rationale. In order to add this, the different implementations (OpenModelica, Dymola, and others) need to agree on a common API. This section presents some naming conventions and other API design issues that need to be taken into consideration when deciding on the standard API.

2.5.1 Naming conventions

Proposal: function names should begin with a Non-capital letters and have a Capital character for each new word in the name, e.g.

```
loadModel
openModelFile
```

2.5.2 Return type

There is a difference between the currently implementations. The OpenModelica untyped API returns strings, `"OK"`, `"-1"`, `"false"`, `"true"`, etc., whereas the typed OpenModelica command API and Dymola returns Boolean values, e.g `true` or `false`.

Proposal: All functions, not returning information, like for instance `getModelName`, should return a Boolean value. (??Note: This is not the final solution since we also need to handle failure indications for functions returning information, which can be done better when exception handling becomes available).

2.5.3 Argument types

There is also a difference between implementations regarding the type of the arguments of certain functions. For instance, Dymola uses strings to denote model and variable references, while OpenModelica uses model/variable references directly.

For example, `loadModel("Resistor")` in Dymola, but `loadModel(Resistor)` in OpenModelica.

One could also support both alternatives, since Modelica will probably have function overloading in the near future.

2.5.4 Set of API Functions

The major issue is of course which subset of functions to include, and what they should do.

Below is a table of Dymola and OpenModelica functions merged together. The table also contains a proposal for a possible standard.

```
<s> == string
<cr> == component reference
[] == list constructor, e.g. [<s>] == vector of strings
```

Dymola	*OpenModelica*	*Description*	*Proposal*
`list()`	`listVariables()`	List all user-defined variables.	`listVariables()`
`listfunctions()`	-	List builtin function names and descriptions.	`listFunctions()`
-	`list()`	List all loaded class definitions.	`list()`
-	`list(<cref>)`	List model definition of <cref>.	`list(<cref>)` or `list(<string>)`
`classDirectory()`	`cd()`	Return current directory.	`currentDirectory()`
`eraseClasses()`	`clearClasses()`	Removes models.	`clearClasses()`
`clear()`	`clear()`	Removes all, including models and variables.	`clearAll()`
-	`clearVariables()`	Removes all user defined variables.	`clearVariables()`
-	`clearClasses()`	Removes all class definitions.	`clearClasses()`
`openModel(<string>)`	`loadFile(<string>)`	Load all definitions from file.	`loadFile(<string>)`
`openModelFile(<string>)`	`loadModel (<cref>)`	Load file that contains model.	`loadModel(<cref>)`, `loadModel(<string>)`
`saveTotalModel(<string>,<string>)`	-	Save total model definition of a model in	`saveTotalModel(<string>,<cref>)` or

		a file.	saveTotalModel(<string>,<string>)
-	saveModel(<cref>, <string>)	Save model in a file.	saveModel(<string>,<cref>) or saveModel(<string>,<string>)
-	createModel(<cref>)	Create new empty model.	createModel(<cref>) or createModel(<string>)
eraseClasses({<string>})	deleteModel(<cref>)	Remove model(s) from symbol table.	deleteModel(<cref>) or deleteModel(<string>)
instantiateModel(<string>	instantiateClass(<cref>)	Perform code instantiation of class.	instantiateClass(<cref>) or instantiateClass(<string>)

2.5.5 Example of Exporting XML from a Model

The following is an example of using the function dumpXMLDAE to export an XML representation of a model.

```
model Circuit1
  parameter Real C(min=5e-07, max=2e-06)=1e-06;
  parameter Real R1=50;
  parameter Real R2=50;
  parameter Real R3(min=500, max=2000)=1000;
  input Real i0;
  Real i1;
  Real i3;
  Real v1;
  Real v2;
  output Real v3;

equation
  C*der(v1)=i0 - i1;
  v1 - v2=i1*R1;
  v2 - v3=i1*R2;
  C*der(v3)=i1 - i3;
  v3=R3*i3;
end Circuit1;

loadFile('../path_to_mo_file/Circuit1.mo');
dumpXMLDAE(Circuit1);
```

will produce the following result:

```
{"<?xml version="1.0" encoding="UTF-8"?>
<dae xmlns:p1="http://www.w3.org/1998/Math/MathML" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://home.dei.polimi.it/donida/Projects/AutoEdit/Images/DAE.xsd">
 <variables dimension="11">
  <orderedVariables dimension="6">
   <variablesList>
    <variable id="1" name="v3" variability="continuousState" direction="output"
              type="Real" index="-1" origName="v3" fixed="true" flow="NonConnector">
     <classesNames> <element>Circuit1 </element> </classesNames>
    </variable>
    <variable id="2" name="v2" variability="continuous" direction="none"
```

```
                type="Real" index="-1" origName="v2" fixed="false" flow="NonConnector">
      <classesNames> <element>Circuit1 </element> </classesNames>
     </variable>
     <variable id="3" name="v1" variability="continuousState" direction="none"
                type="Real" index="-1" origName="v1" fixed="true" flow="NonConnector">
      <classesNames> <element>Circuit1 </element>  </classesNames>
     </variable>
     <variable id="4" name="i3" variability="continuous" direction="none"
                type="Real" index="-1" origName="i3" fixed="false" flow="NonConnector">
      <classesNames>  <element>Circuit1 </element>  </classesNames>
     </variable>
     <variable id="5" name="i1" variability="continuous" direction="none"
                type="Real" index="-1" origName="i1" fixed="false" flow="NonConnector">
      <classesNames> <element>Circuit1 </element> </classesNames>
     </variable>
     <variable id="6" name="$dummy" variability="continuousState" direction="none"
                type="Real" index="-1" origName="$dummy" fixed="true" flow="NonConnector">
      <attributesValues>
       <fixed string="true">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <apply> <true/> </apply> </math> </MathML>
       </fixed>
      </attributesValues>
     </variable>
    </variablesList>
   </orderedVariables>
   <knownVariables dimension="5">
    <variablesList>
     <variable id="1" name="i0" variability="continuous" direction="input"
                type="Real" index="-1" origName="i0" fixed="false" flow="NonConnector">
      <classesNames> <element>Circuit1 </element> </classesNames>
     </variable>
     <variable id="2" name="R3" variability="parameter" direction="none"
                type="Real" index="-1" origName="R3" fixed="true" flow="NonConnector">
      <bindValueExpression>
       <bindExpression string="1000">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="integer">1000 </cn> </math> </MathML>
       </bindExpression>
      </bindValueExpression>
      <classesNames> <element>Circuit1 </element> </classesNames>
      <attributesValues>
       <minValue string="500.0">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="real">500.0 </cn> </math> </MathML>
       </minValue>
       <maxValue string="2000.0">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="real">2000.0 </cn> </math> </MathML>
       </maxValue>
      </attributesValues>
     </variable>
     <variable id="3" name="R2" variability="parameter" direction="none"
                type="Real" index="-1" origName="R2" fixed="true" flow="NonConnector">
      <bindValueExpression>
       <bindExpression string="50">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="integer">50 </cn> </math> </MathML>
       </bindExpression>
      </bindValueExpression>
      <classesNames> <element>Circuit1 </element> </classesNames>
     </variable>
     <variable id="4" name="R1" variability="parameter" direction="none"
                type="Real" index="-1" origName="R1" fixed="true" flow="NonConnector">
      <bindValueExpression>
       <bindExpression string="50">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="integer">50 </cn> </math> </MathML>
       </bindExpression>
      </bindValueExpression>
      <classesNames> <element>Circuit1 </element> </classesNames>
     </variable>
     <variable id="5" name="C" variability="parameter" direction="none"
                type="Real" index="-1" origName="C" fixed="true" flow="NonConnector">
      <bindValueExpression>
       <bindExpression string="1e-06">
        <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="real">1e-06 </cn> </math> </MathML>
       </bindExpression>
      </bindValueExpression>
```

```
        <classesNames> <element>Circuit1 </element> </classesNames>
        <attributesValues>
         <minValue string="5e-07">
          <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="real">5e-07 </cn> </math> </MathML>
         </minValue>
         <maxValue string="2e-06">
          <MathML> <math xmlns="http://www.w3.org/1998/Math/MathML"> <cn type="real">2e-06 </cn> </math> </MathML>
         </maxValue>
        </attributesValues>
       </variable>
     </variablesList>
   </knownVariables>
  </variables>
  <equations dimension="6">
   <equation id="1">
    C * der(v1) = i0 - i1       <MathML>
     <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply> <equivalent/>
       <apply>
        <times/>  <ci>C </ci> <apply> <diff/> <ci>v1 </ci> </apply> </apply> <apply> <minus/> <ci>i0 </ci> <ci>i1 </ci>
       </apply>
      </apply>
     </math>
    </MathML>
   </equation>
   <equation id="2">
    v1 - v2 = i1 * R1       <MathML>
     <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply> <equivalent/>
       <apply> <minus/> <ci>v1 </ci> <ci>v2 </ci> </apply>
       <apply> <times/> <ci>i1 </ci> <ci>R1 </ci> </apply>
      </apply>
     </math>
    </MathML>
   </equation>
   <equation id="3">
    v2 - v3 = i1 * R2       <MathML>
     <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>  <equivalent/>
       <apply> <minus/> <ci>v2 </ci> <ci>v3 </ci> </apply>
       <apply> <times/> <ci>i1 </ci> <ci>R2 </ci> </apply>
      </apply>
     </math>
    </MathML>
   </equation>
   <equation id="4">
    C * der(v3) = i1 - i3       <MathML>  <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply> <equivalent/>
       <apply> <times/> <ci>C </ci> <apply> <diff/> <ci>v3 </ci> </apply> </apply>
       <apply> <minus/> <ci>i1 </ci> <ci>i3 </ci> </apply>
      </apply>
     </math>
    </MathML>
   </equation>
   <equation id="5">
    v3 = R3 * i3       <MathML>  <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply> <equivalent/> <ci>v3 </ci> <apply> <times/> <ci>R3 </ci> <ci>i3 </ci> </apply> </apply>
     </math>
    </MathML>
   </equation>
   <equation id="6">
    der($dummy) = sin(time * 628.318530717) <MathML>  <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply> <equivalent/> <apply> <diff/> <ci>$dummy </ci> </apply>
       <apply> <sin/> <apply> <times/> <ci>time </ci> <cn type="real">628.318530717 </cn> </apply> </apply>  </apply>
     </math>
    </MathML>
   </equation>
  </equations>
</dae> ","The model has been dumped to xml file: Circuit1.xml"}
```

2.5.6 Example of Exporting Matlab from a Model

The command export dumps an XML representation of a model, according to several optional parameters.

```
exportDAEtoMatlab(modelname);
```

This command dumps the mathematical representation of a model using a Matlab representation. Example:

```
$ cat daequery.mos
loadFile("BouncingBall.mo");
exportDAEtoMatlab(BouncingBall);
readFile("BouncingBall_imatrix.m");

$ omc daequery.mos
true
"The equation system was dumped to Matlab file:BouncingBall_imatrix.m"
"
% Incidence Matrix
% ===================================
% number of rows: 6
IM={[3,-6],[1,{'if', 'true','==' {3},{},}],[2,{'if', 'edge(impact)'
{3},{5},}],[4,2],[5,{'if', 'true','==' {4},{},}],[6,-5]};
VL = {'foo','v_new','impact','flying','v','h'};

EqStr = {'impact = h <= 0.0;','foo = if impact then 1 else 2;','when {h <= 0.0 AND
v <= 0.0,impact} then v_new = if edge(impact) then (-e) * pre(v) else 0.0; end
when;','when {h <= 0.0 AND v <= 0.0,impact} then flying = v_new > 0.0; end
when;','der(v) = if flying then -g else 0.0;','der(h) = v;'};

OldEqStr={'fclass BouncingBall','parameter Real e = 0.7 "coefficient of
restitution";','parameter Real g = 9.81 "gravity acceleration";','Real h(start =
1.0) "height of ball";','Real v "velocity of ball";','Boolean flying(start = true)
"true, if ball is flying";','Boolean impact;','Real v_new;','Integer
foo;','equation','  impact = h <= 0.0;','  foo = if impact then 1 else 2;','
der(v) = if flying then -g else 0.0;','  der(h) = v;','  when {h <= 0.0 AND v <=
0.0,impact} then','  v_new = if edge(impact) then (-e) * pre(v) else 0.0;','
flying = v_new > 0.0;','  reinit(v,v_new);','  end when;','end
BouncingBall;',''};"
```

Chapter 3

Detailed Overview of OpenModelica Packages

This chapter gives overviews of all packages in the OpenModelica compiler/interpreter and server functionality, as well as the detailed interconnection structure between the modules.

3.1 Detailed Interconnection Structure of Compiler Packages

A fairly detailed view of the interconnection structure, i.e., the main data flows and and dependencies between the modules in the OpenModelica compiler, is depicted in ??Figure 3-1 below.

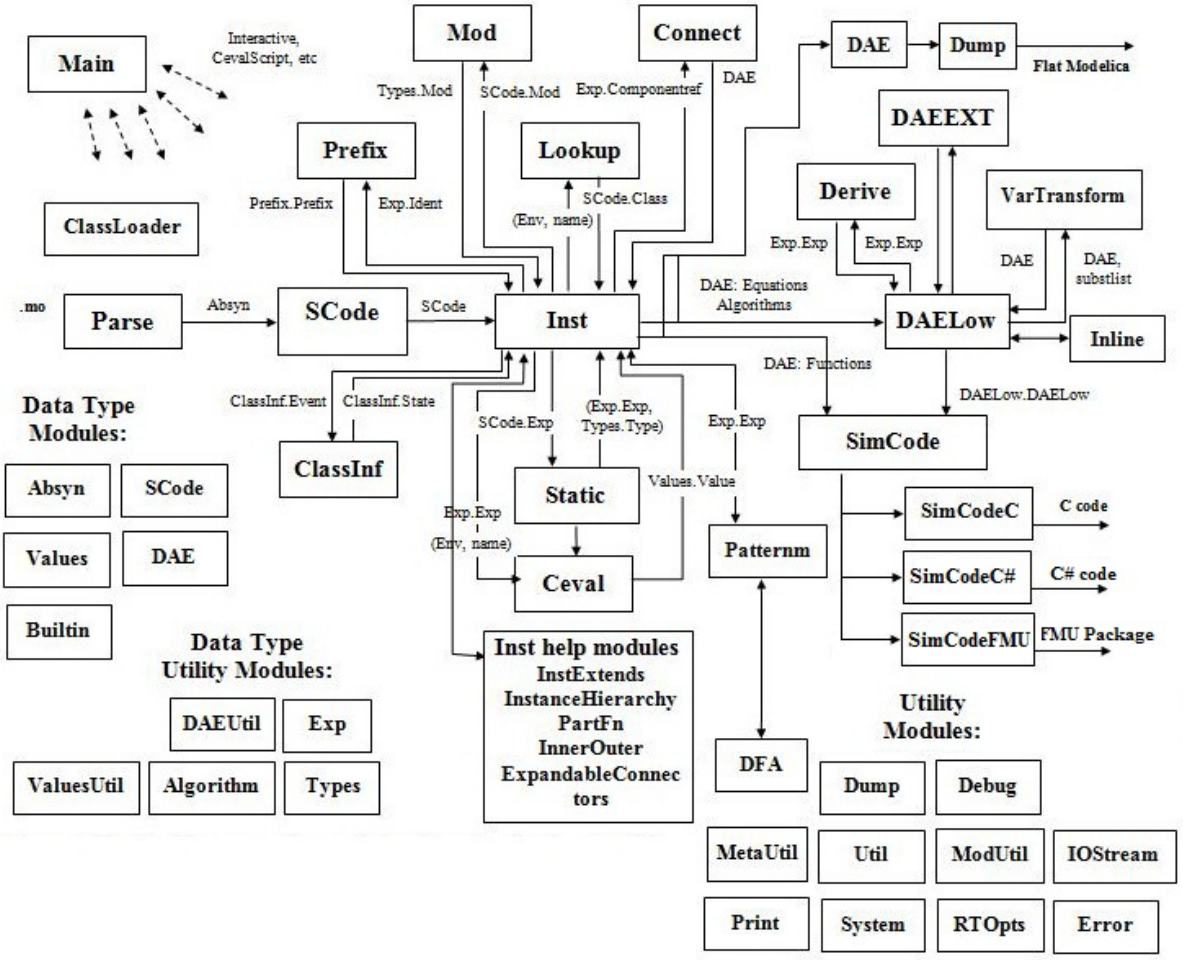

Figure 3-1. Module connections and data flows in the OpenModelica compiler. Note: many of the new modules are not yet included in this picture.