OpenModelica System Documentation

Preliminary Draft, 2008-02-20
for OpenModelica 1.4.4

February 2008

Peter Fritzson
Adrian Pop, Peter Aronsson,
David Akhvlediani, Bernhard Bachmann, Vasile Baluta,
Simon Björklén, Mikael Blom, David Broman,
Henrik Eriksson, Anders Fernström, Pavel Grozman, Daniel Hedberg,
Kim Jansson, Joel Klinghed, Magnus Leksell, Håkan Lundvall, Eric Meyers,
Kristoffer Norling, Klas Sjöholm, Kristian Stavåker, Constantin Belyaev

This document is part of OpenModelica: www.ida.liu.se/projects/OpenModelica
Contact: OpenModelica@ida.liu.se

Table of Contents

Preface

This system documentation has been prepared to simplify further development of the OpenModelica compiler as well as other parts of the environment. It contains contributions from a number of developers.

Chapter 1

Introduction

This document is intended as system documentation for the OpenModelica environment, for the benefit of developers who are extending and improving OpenModelica. For information on how to use the OpenModelica environment, see the OpenModelica users guide.

This system documentation, version May 2006, primarily includes information about the OpenModelica compiler. Short chapters about the other subsystems in the OpenModelica environment are also included.

1.1 OpenModelica Environment Structure

The OpenModelica environment consists of several interconnected subsystems, as depicted in Figure 1-1 below.

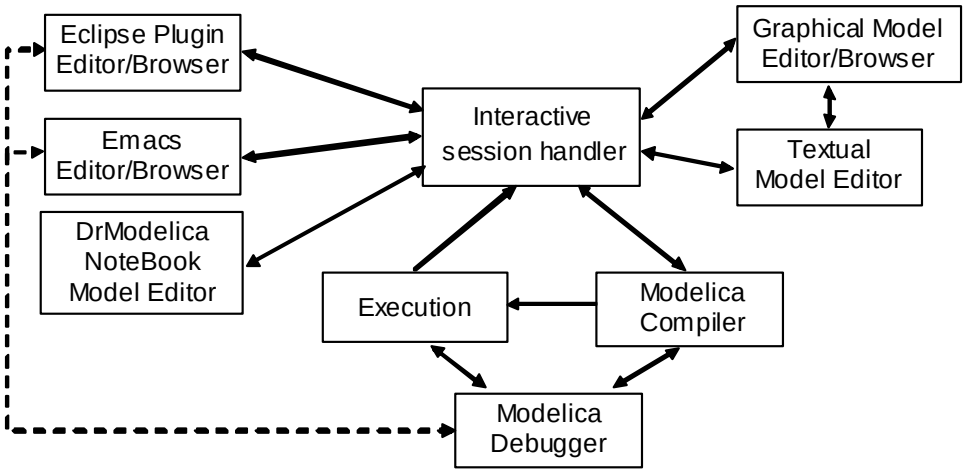


Figure 1-1. The overall architecture of the OpenModelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating commands and expressions that are translated and executed. Several subsystems provide different forms of browsing and textual editing of Modelica code. The debugger currently provides debugging of an extended algorithmic subset of Modelica, and uses Emacs or Eclipse for display and positioning. The graphical model editor is not really part of OpenModelica but integrated into the system and available from MathCore Engineering AB without cost for academic usage.

As mentioned above, this version of the system documentation only includes the OpenModelica compilation subsystem, translating Modelica to C code. The compiler also includes a Modelica interpreter for interactive usage and for command and constant expression evaluation. The subsystem includes facilities for building simulation executables linked with selected numerical ODE or DAE solvers. Currently the default solver is DASSL.

1.2 OpenModelica Compiler Translation Stages

The Modelica translation process is schematically depicted in Figure 1-2 below. Modelica source code (typically .mo files) input to the compiler is first translated to a so-called flat model. This phase includes type checking, performing all object-oriented operations such as inheritance, modifications etc., and fixing package inclusion and lookup as well as import statements. The flat model includes a set of equations declarations and functions, with all object-oriented structure removed apart from dot notation within names. This process is a *partial instantiation* of the model, called *code instantiation* or *elaboration* in subsequent sections.

The next two phases, the equation analyzer and equation optimizer, are necessary for compiling models containing equations. Finally, C code is generated which is fed through a C compiler to produce executable code.

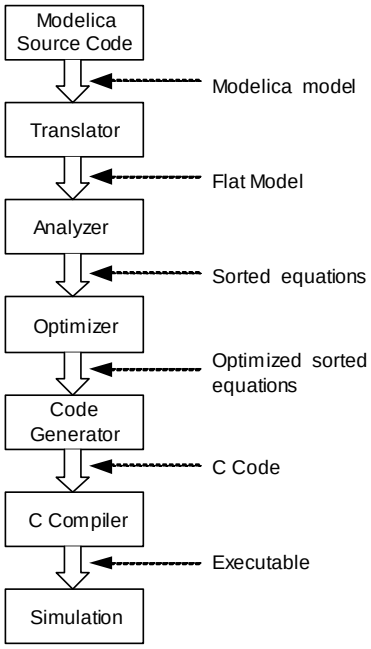


Figure 1-2. Translation stages from Modelica code to executing simulation.

1.3 Simplified Overall Structure of the Compiler

The OpenModelica compiler is separated into a number of modules, to separate different stages of the translation, and to make it more manageable. The top level function is called `main`, and appears as follows in simplified form that emits flat Modelica (leaving out the code generation and symbolic equation manipulation):

```
function main
  input String f;  // file name
algorithm
  ast := Parser.parse(f);
  scode1 := SCode.elaborate(ast);
  scode2 := Inst.elaborate(scode1);
  DAE.dump(scode2);
end main;
```

The simplified overall structure of the OpenModelica compiler is depicted in Figure 1-3, showing the most important modules, some of which can be recognized from the above `main` function. The total system contains approximately 40 modules.

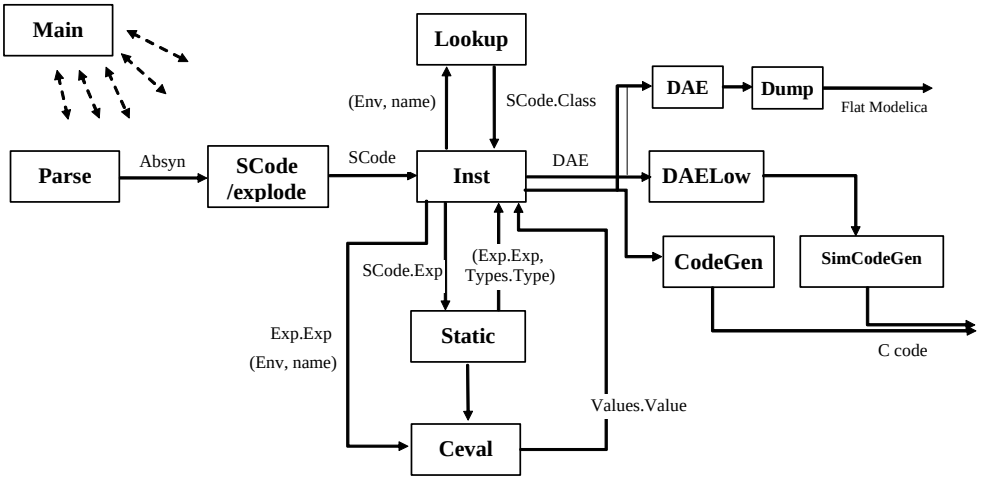


Figure 1-3. Some module connections and data flows in the OpenModelica compiler. The parser generates abstract syntax (Absyn) which is converted to the simplified (SCode) intermediate form. The code instantiation module (Inst) calls Lookup to find a name in an environment. It also generates the DAE equation representation which is simplified by DAELow. The Ceval module performs compile-time or interactive expression evaluation and returns values. The Static module performs static semantics and type checking. The DAELow module performs BLT sorting and index reduction. The DAE module internally uses Exp.Exp, Types.Type and Algorithm.Algorithm; the SCode module internally uses Absyn.

1.4 Parsing and Abstract Syntax

The function `Parser.parse` is actually written in C, and calls the parser generated from a grammar by the ANTLR parser generator tool (ANTLR 1998). This parser builds an abstract syntax tree (AST) from the source file, using the AST data types in a MetaModelica module called `Absyn`. The parsing stage is not really part of the semantic description, but is of course necessary to build a real translator.

1.5 Rewriting the AST into SCode

The AST closely corresponds to the parse tree and keeps the structure of the source file. This has several disadvantages when it comes to translating the program, and especially if the translation rules should be easy to read for a human. For this reason a preparatory translation pass is introduced which translates the AST into an intermediate form, called `SCode`. Besides some minor simplifications the `SCode` structure differs from the AST in the following respects:

- All variables are described separately. In the source and in the AST several variables in a class definition can be declared at once, as in `Real x, y[17];`. In the `SCode` this is represented as two unrelated declarations, as if it had been written `Real x; Real y[17];`.
- Class declaration sections. In a Modelica class declaration the public, protected, equation and algorithm sections may be included in any number and in any order, with an implicit public section first. In the `SCode` these sections are collected so that all public and protected sections are combined into one section, while keeping the order of the elements. The information about which elements were in a protected section is stored with the element itself.

One might have thought that more work could be done at this stage, like analyzing expression types and resolving names. But due to the nature of the Modelica language, the only way to know anything about how the names will be resolved during elaboration is to do a more or less full elaboration. It is possible to

analyze a class declaration and find out what the parts of the declaration would mean if the class was to be elaborated as-is, but since it is possible to modify much of the class while elaborating it that analysis would not be of much use.

1.6 Model Flattening and Instantiation

To be executed, classes in a model need to be instantiated, i.e., data objects are created according to the class declaration. There are two phases of instantiation:

- The symbolic, or compile time, phase of instantiation is usually called *flattening*/*elaboration* or *code instantiation*. No data objects are created during this phase. Instead the symbolic internal representation of the model to be executed/simulated is transformed, by performing inheritance operations, modification operations, aggregation operations, etc.
- The creation of the data object, usually called *instantiation* in ordinary object-oriented terminology. This can be done either at compile time or at run-time depending on the circumstances and choice of implementation.

The central part of the translation is the *code instantiation* or flattening/elaboration of the model. The convention is that the top-level model in the instance hierarchy in the source file is elaborated, which means that the equations in that model declaration, and all its subcomponents, are computed and collected.

The elaboration of a class is done by looking at the class definition, elaborating all subcomponents and collecting all equations, functions, and algorithms. To accomplish this, the translator needs to keep track of the class context. The context includes the lexical scope of the class definition. This constitutes the *environment* which includes the variables and classes declared previously in the same scope as the current class, and its parent scope, and all enclosing scopes. The other part of the context is the current set of modifiers which modify things like parameter values or redeclare subcomponents.

```
model M
  constant Real c = 5;
  model Foo
    parameter Real p = 3;
    Real x;
  equation
    x = p * sin(time) + c;
  end Foo;

  Foo f(p = 17);
end M;
```

In the example above, elaborating the model `M` means elaborating its subcomponent f, which is of type `Foo`. While elaborating `f` the current environment is the parent environment, which includes the constant `c`. The current set of modifications is (`p = 17`), which means that the parameter `p` in the component `f` will be 17 rather than 3.

There are many semantic rules that takes care of this, but only a few are shown here. They are also somewhat simplified to focus on the central aspects.

1.7 The instClass and instElement Functions

The function `instClass` elaborates a class. It takes five arguments, the environment `env`, the set of modifications `mod`, the prefix `inPrefix` which is used to build a globally unique name of the component in a hierarchical fashion, a collection of connection sets `csets`, and the class definition `inScodeclass`. It opens a new scope in the environment where all the names in this class will be stored, and then uses a function called `instClassIn` to do most of the work. Finally it generates equations from the connection sets collected while elaborating this class. The “result” of the function is the *elaborated* equations and some information about what was in the class. In the case of a function, regarded as a restricted class, the result is an algorithm section.

One of the most important functions is `instElement`, that elaborates an element of a class. An element can typically be a class definition, a variable or constant declaration, or an extends-clause. Below is shown *only* the rule in `instElement` for elaborating variable declarations.

The following are simplified versions of the `instClass` and `instElement` functions.

```
function instClass  "Symbolic instantiation of a class"
  input Env          inEnv;
  input Mod          inMod;
  input Prefix       inPrefix;
  input Connect.Sets inConnectsets;
  input Scode.Class  inScodeclass;
  output list<DAE.Element> outDAEelements;
  output Connect.Sets outConnectSets;
  output Types.Type   outType;
algorithm
  (outDAEelements, outConnectSets, outType) :=
  matchcontinue (inEnv,inMod,inPrefix,inConnectsets,inScodeclass)
    local
      Env env,env1;  Mod mod;  Prefix prefix;
      Connect.Sets connectSets,connectSets1;
      ... n,r;  list<DAE.Element> dae1,dae2;
    case (env,mod,pre,connectSets, scodeClass as SCode.CLASS(n,_,r,_))
      equation
        env1 = Env.openScope(env);
        (dae1,_,connectSets1,ciState1,tys) = instClassIn(env1,mod,prefix,
                                                         connectSets, scodeClass);
        dae2 = Connect.equations(connectSets1);
        dae  = listAppend(dae1,dae2);
        ty   = mktype(ciState1,tys);
     then (dae, {}, ty);
  end matchcontinue;
end instClass;


function instElement  "Symbolic instantiation of an element of a class"
  input  Env               inEnv;
  input  Mod               inMod;
  input  Prefix            inPrefix;
  input  Connect.Sets      inConnectSets;
  input  Scode.Element     inScodeElement;
  output list<DAE.Element> outDAEelement;
  output Env               outEnv;
  output Connect.Sets      outConnectSets;
  output list<Types.Var>   outTypesVar;
algorithm
  (outDAE,outEnv,outdConnectSets,outdTypesVar) :=
  matchcontinue (inEnv,inMod,inPrefix,inConnectSets,inScodeElement)
    local
      Env env,env1;  Mod mods;  Prefix pre;
      Connect.Sets csets,csets1;
      ...  n, final, prot, attr, t, m;
    ...
    case (env,mods,pre,csets, SCode.COMPONENT(n,final,prot,attr,t,m))
      equation
        vn = Prefix.prefixCrefCref(pre,Exp.CREF_IDENT(n,{} ));
        (cl,classmod) = Lookup.lookupClassClass(env,t)      // Find the class definition
        mm = Mod.lookupModification(mods,n);
        mod = Mod.merge(classmod,mm);                // Merge the modifications
        mod1 = Mod.merge(mod,m);
        pre1 = Prefix.prefixAddAdd(n,[],pre);          // Extend the prefix
        (dae1,csets1,ty,st) =
              instClass(env,mod1,pre1,csets,cl)     //   Elaborate the variable
        eq = Mod.modEquation(mod1); //   If the variable is declared with a default equation,
```

```
        binding = makeBinding (env,attr,eq,cl); // add it to the environment
                                                // with the variable.
        env1 = Env.extendFrameFrame_v(env,           // Add the variable binding to the
          Env.FRAMEVAR(n,attr,ty,binding));     // environment
        dae2 = instModEquation(env,pre,n,mod1); // Fetch the equation, if supplied
        dae = listAppendAppend(dae1, dae2);           // Concatenate the equation lists
      then (dae, env1,csets1, { (n,attr,ty) } )
    ...
  end matchcontinue;
end instElement;
```

1.8 Output

The equations, functions, and variables found during elaboration (symbolic instantiation) are collected in a list of objects of type `DAEcomp`:

```
uniontype DAEcomp
  record VAR  Exp.ComponentRef componentRef;  VarKind varKind;  end VAR;
  record EQUATION  Exp exp1;  Exp exp2; end EQUATION;
end DAEcomp;
```

As the final stage of translation, functions, equations, and algorithm sections in this list are converted to C code.

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`.

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 ++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.
- Codegen
  `cg` ??
  `cgtr` Tracing matching rules
  `codegen` Code generation.
- 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.

2.2 The OpenModelica Client-Server Architecture

The OpenModelica client-server architecture is schematically depicted in Figure 2-1, showing two typical clients: a graphic model editor and an interactive session handler for command interpretation.

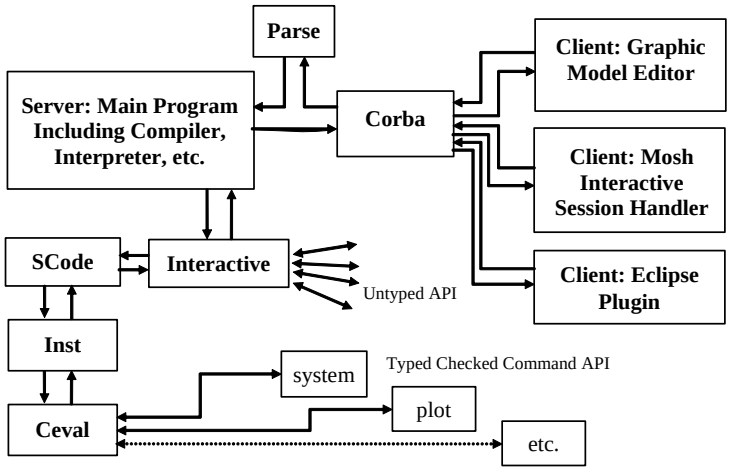


Figure 2-1. Client-Server interconnection structure of the compiler/interpreter main program and interactive tool interfaces. Messages from the Corba interface are of two kinds. The first group consists of expressions or user commands which are evaluated by the Ceval module. The second group are declarations 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) (??Optional arguments xrange and yrange not yet implemented)	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) (??partly implemented)	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;` *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:\OpenModelica1.2.1\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});
```

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. T

--- 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 (A1<string>)	Loads all models in the file. Also in typed API. Returns list of 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>)	Returns the nth (A2) equation of the model named by A1. e.g.

(??NotYetImplemented)	`"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 ---`	
`getVersion``()`	returns the OMC version, e.g. `"1.4.2"`

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[:];  // Problem, fails to instantiate if
                           // styles are given as modification
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 a file.	`saveTotalModel(<string>,<cref>)` or `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>)

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. (??Note that there is a Word bug that arbitrarily changes the width of the arrows)

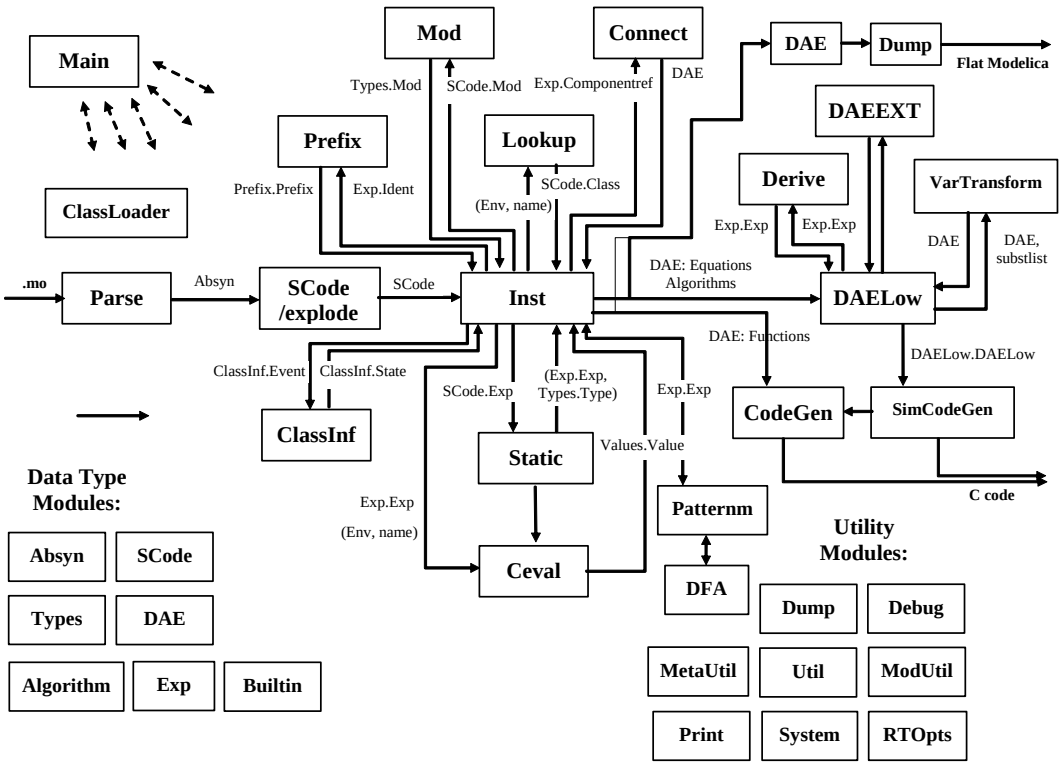


Figure 3-1. Module connections and data flows in the OpenModelica compiler.

One can see that there are three main kinds of modules:

- Function modules that perform a specified function, e.g. Lookup, code instantiation, etc.
- Data type modules that contain declarations of certain data types, e.g. Absyn that declares the abstract syntax.
- Utility modules that contain certain utility functions that can be called from any module, e.g. the Util module with list processing funtions.

Note that this functionality classification is not 100% clearcut, since certain modules performs several functions. For example, the SCode module primarily defines the lower-level SCode tree structure, but also transforms Absyn into SCode. The DAE module defines the DAE equation representation, but also has a few routines to emit equations via the Dump module.

We have the following approximate description:

- The Main program calls a number of modules, including the parser (Parse), SCode, etc.
- The parser generates abstract syntax (Absyn) which is converted to the simplified (SCode) intermediate form.
- The code instantiation module (Inst) is the most complex module, and calls many other modules. It calls Lookup to find a name in an environment, calls Prefix for analyzing prefixes in qualified variable designators (components), calls Mod for modifier analysis and Connect for connect equation analys. It also generates the DAE equation representation which is simplified by DAELow and fed to the SimCodeGen code generator for generating equation-based simulation code, or directly to CodeGen for compiling Modelica functions into C functions
- The Ceval module performs compile-time or interactive expression evaluation and returns values. The Static module performs static semantics and type checking.
- The DAELow module performs BLT sorting and index reduction. The DAE module internally uses Exp.Exp, Types.Type and Algorithm.Algorithm; the SCode module internally uses Absyn
- The Vartransform module called from DAELow performs variable substitution during the symbolic transformation phase (BLT and index reduction).
- The Patternm module performs compilation of pattern match expressions in the MetaModelica language extension, calling the DFA and MetaUtil modules.

3.2 OpenModelica Source Code Directory Structure

The following is a short summary of the directory structure of the OpenModelica compiler and interactive subsystem.

3.2.1 OpenModelica/Compiler/

Contains all MetaModelica files of the compiler, listed in Section ??.

3.2.2 OpenModelica/Compiler/runtime

This directory contains runtime modules, both for the compiler and for interactive system and communication needs. Mostly written in C.

`rtops.c`	Accessing compiler options.
`printimpl.c`	Print routines, e.g. for debug tracing.
`socketimpl.c`	Phased out. Should not be used. Socket communication between clients and the OpenModelica main program.
`corbaimpl.cpp`	Corba communication between clients and the OpenModelica main program.
`ptolemyio.cpp`	IO routines from the Ptolemy system to store simulation data for plotting, etc.

`systemimpl.c`	Operating system calls.
`daeext.cpp`	C++ routines for external DAE bit vector operations, etc.

3.2.3 OpenModelica/testsuite

This directory contains the Modelica testsuite consisting two subdirectories mofiles and mosfiles. The `mofiles` directory contains more than 200 test models. The `mosfiles` directory contains a few Modelica script files consisting of commands according to the general command API.

3.2.4 OpenModelica/OMShell

Files for the OpenModelica interactive shell, called `OMShell` for OpenModelica Shell.

3.2.5 OpenModelica/c_runtime – OpenModelica Run-time Libraries

This directory contains files for the Modelica runtime environment. The runtime contains a number of C files, for which object code versions are are packaged in of two libraries, `libc_runtime.a` and `libsim.a`. We group the C files under the respective library, even though the files occur directly under the `c_runtime` directory.

3.2.5.1 libc_runtime.a

The `libc_runtime` is used for executing Modelica functions that has been generated C code for. It contains the following files.

`boolean_array.*`	How arrays of booleans are represented in C.
`integer_array.*`	How arrays of integers are represented in C.
`real_array.*`	How arrays of reals are represented in C.
`string_array.*`	How arrays of strings are represented in C.
`index_spec.c`	Keep track of dimensionsizes of arrays.
`memory_pool.c`	Memory allocation for local variables.
`read_write.*`	Reading and writing of data to file.
`utility.c`	Utility functions

3.2.5.2 libsim.a

The library `libsim.a` is the runtime library for simulations, it contains solvers and a `main` function for the simulation. The following files are included:

`simulation_runtime.*`	Includes the `main` function, solver wrappers,etc.
`daux.f`	Auxiliary Fortran functions.
`ddasrt.f`	DDASRT solver.
`ddassl.f`	DASSL solver.
`dlamch.f`	Determine machine parameters for solvers.
`dlinpk.f`	Gaussian elimination routines, used by solvers.
`lsame.f`	LAPACK axuiliary routine LSAME.

Non-linear solver:

`hybrd1.f`	Non-linear solver with approximate jacobian.
`hybrj.f`	Non-linear solver with analythical jacobian.- alternative for non-linear solver.

`fdjac1.f`	Helper routines
`enorm.f`	Helper routines.
`dpmpar.f`	Helper routines
`dogleg.f`	Helper routines

3.3 Short Overview of Compiler Modules

The following is a list of the OpenModelica compiler modules with a very short description of their functionality. Chapter 3 describes these modules in more detail.

Absyn	Abstract Syntax
Algorithm	Data Types and Functions for Algorithm Sections
Builtin	Builtin Types and Variables
Ceval	Evaluation/interpretation of Expressions.
ClassInf	Inference and check of class restrictions for restricted classes.
ClassLoader	Loading of Classes from $OPENMODELICALIBRARY
Codegen	Generate C Code from functions in DAE representation.
Connect	Connection Set Management
Corba	Modelica Compiler Corba Communication Module
DAE	DAE Equation Management and Output
DAEEXT	External Utility Functions for DAE Management
DAELow	Lower Level DAE Using Sparse Matrises for BLT
Debug	Trace Printing Used for Debugging
Derive	Differentiation of Equations from DAELow
DFA	A deterministic finite automata (DFA) used by the pattern match algorithm in Patternm.
Dump	Abstract Syntax Unparsing/Printing
DumpGraphviz	Dump Info for Graph visualization of AST
Env	Environment Management
Exp	Typed Expressions after Static Analysis /*updated)
Graphviz	Graph Visualization from Textual Representation
Inst	Code Instantiation/Elaboration of Modelica Models
Interactive	Model management and expression evaluation – the function Interactive.evaluate. Keeps interactive symbol tables. Contains Graphic Model Editor API.
Lookup	Lookup of Classes, Variables, etc.
Main	The Main Program. Calls Interactive, the Parser, the Compiler, etc.
MetaUtil	MetaModelica Related Utility Functions
Mod	Modification Handling
ModSim	/*Depreciated, not used). Previously communication for Simulation, Plotting, etc.
ModUtil	Modelica Related Utility Functions
Parse	Parse Modelica or Commands into Abstract Syntax
Patternm	The MetaModelica pattern match compilation algorithm.
Prefix	Handling Prefixes in Variable Names
Print	Buffered Printing to Files and Error Message Printing
RTOpts	Run-time Command Line Options
SCode	Simple Lower Level Intermediate Code Representation.

SimCodegen	Generate simulation code for solver from equations and algorithm sections in DAE.
Socket	(Partly Depreciated) OpenModelica Socket Communication Module
Static	Static Semantic Analysis of Expressions
System	System Calls and Utility Functions
TaskGraph	Building Task Graphs from Expressions and Systems of Equations. Optional module.
TaskGraphExt	External Representation of Task Graphs. Optional module.
Types	Representation of Types and Type System Info
Util	General Utility Functions
Values	Representation of Evaluated Expression Values
VarTransform	Binary Tree Representation of Variable Transformations

3.4 Descriptions of OpenModelica Compiler Modules

The following are more detailed descriptions of the OpenModelica modules.

3.4.1 Absyn – Abstract Syntax

This module defines the abstract syntax representation for Modelica in MetaModelica. It primarily contains datatypes for constructing the abstract syntax tree (AST), functions for building and altering AST nodes and a few functions for printing the AST:

- Abstract Syntax Tree (Close to Modelica)
  - Complete Modelica 2.2
  - Including annotations and comments
- Primary AST for e.g. the Interactive module
  - Model editor related representations (must use annotations)
- Functions
  - A few small functions, only working on Absyn types, e.g.:
    - `pathToCref(Path) => ComponentRef`
    - `joinPaths(Path, Path) => (Path)`
    - `etc.`

The constructors defined by the Absyn module are primarily used by the walker (`Compiler/absyn_builder/walker.g`) which takes an ANTLR internal syntax tree and converts it into an MetaModelica abstract syntax tree. When the AST has been built, it is normally used by the SCode module in order to build the `SCode` representation. It is also possible to send the AST to the unparser (Dump) in order to print it.

For details regarding the abstract syntax tree, check out the grammar in the Modelica language specification.

The following are the types and datatypes that are used to build the AST:

An *identifier*, for example a variable name:

```
type Ident = String;
```

Info attribute type.

The `Info` attribute type is not needed to represent Modelica language constructs or for the semantics. Instead, `Info` contains various pieces of information needed by tools for debugging and browsing support.

```
uniontype Info
 "Modextension: Various pieces of information needed for debugging and browsing"
```

```
  record INFO
    String  fileName   "fileName where the class is defined in" ;
    Boolean isReadOnly "isReadOnly : (true|false). Should be true for libraries" ;
    Integer lineNumberStart;
    Integer columnNumberStart;
    Integer lineNumberEnd;
    Integer columnNumberEnd;
  end INFO;

end Info;
```

Programs, the top level construct:

A program is simply a list of class definitions declared at top level in the source file, combined with a `within` clause. that indicates the hierarchical position of the program.

Nodes such as `BEGIN_DEFINITION` and `END_DEFINITION` can be used for representing packages and classes that are entered piecewise, e.g., first entering the package head (as `BEGIN_DEFINITION`), then the contained definitions, then an end package repesented as `END_DEFINITION`.

```
uniontype Program
  record PROGRAM
    list<Class>  classes  "List of classes" ;
    Within       within_  "Within clause" ;
  end PROGRAM;

  record BEGIN_DEFINITION
    Path         path  "path for split definitions" ;
    Restriction  restriction   "Class restriction" ;
    Boolean      partial_      "true if partial" ;
    Boolean      encapsulated_ "true if encapsulated" ;
  end BEGIN_DEFINITION;

  record END_DEFINITION
    Ident  name   "name for split definitions" ;
  end END_DEFINITION;

  record COMP_DEFINITION
    ElementSpec  element    "element for split definitions" ;
    Option<Path> insertInto "insert into, Default: NONE" ;
  end COMP_DEFINITION;

  record IMPORT_DEFINITION
    ElementSpec   importElementFor "For split definitions" ;
    Option<Path>  insertInto       "Insert into, Default: NONE" ;
  end IMPORT_DEFINITION;

end Program;
```

Within Clauses:

```
uniontype Within
  record WITHIN
    Path path;
  end WITHIN;

  record TOP end TOP;

end Within;
```

Classes:

A class definition consists of a name, a flag to indicate if this class is declared as `partial`, the declared class restriction, and the body of the declaration.

```
uniontype Class
  record CLASS
    Ident name;
    Boolean     partial_   "true if partial" ;
    Boolean     final_     "true if final" ;
    Boolean     encapsulated_ "true if encapsulated" ;
    Restriction restricion  "Restriction" ;
    ClassDef    body;
    Info        info    "Information: FileName the class is defined in +
               isReadOnly bool + start line no + start column no +
               end line no + end column no";
  end CLASS;

end Class;
```

ClassDef:

The `ClassDef` type contains the definition part of a class declaration. The definition is either explicit, with a list of parts (`public`, `protected`, `equation`, and `algorithm`), or it is a definition derived from another class or an enumeration type.

For a derived type, the type contains the name of the derived class and an optional array dimension and a list of modifications.

```
uniontype ClassDef
  record PARTS
    list<ClassPart>  classParts;
    Option<String>   comment;
  end PARTS;

  record DERIVED
    TypeSpec         typeSpec "typeSpec specification includes array dimensions";
    ElementAttributes attributes ;
    list<ElementArg>  arguments;
    Option<Comment>   comment;
  end DERIVED;

  record ENUMERATION
    EnumDef          enumLiterals;
    Option<Comment>  comment;
  end ENUMERATION;

  record OVERLOAD
    list<Path>       functionNames;
    Option<Comment>  comment;
  end OVERLOAD;

  record CLASS_EXTENDS
    Ident            name  "class to extend" ;
    list<ElementArg> arguments;
    Option<String>   comment;
    list<ClassPart>  parts;
  end CLASS_EXTENDS;

  record PDER
    Path         functionName;
    list<Ident>  vars  "derived variables" ;
  end PDER;

end ClassDef;
```

EnumDef:

The definition of an enumeration is either a list of literals or a colon, :, which defines a supertype of all enumerations.

```
uniontype EnumDef
  record ENUMLITERALS
    list<EnumLiteral> enumLiterals "enumLiterals" ;
  end ENUMLITERALS;

  record ENUM_COLON end ENUM_COLON;

end EnumDef;
```

EnumLiteral:

An enumeration type contains a list of `EnumLiteral`, which is a name in an enumeration and an optional comment.

```
uniontype EnumLiteral

  record ENUMLITERAL
    Ident           literal
    Option<Comment> comment
  end ENUMLITERAL;

end EnumLiteral;
```

ClassPart:

A class definition contains several parts. There are public and protected component declarations, type definitions and extends-clauses, collectively called elements. There are also equation sections and algorithm sections. The `EXTERNAL` part is used only by functions which can be declared as external C or FORTRAN functions.

```
uniontype ClassPart

  record PUBLIC
    list<ElementItem> contents;
  end PUBLIC;

  record PROTECTED
    list<ElementItem> contents;
  end PROTECTED;

  record EQUATIONS
    list<EquationItem> contents;
  end EQUATIONS;

  record INITIALEQUATIONS
    list<EquationItem> contents;
  end INITIALEQUATIONS;

  record ALGORITHMS
    list<AlgorithmItem> contents;
  end ALGORITHMS;

  record INITIALALGORITHMS
    list<AlgorithmItem> contents;
  end INITIALALGORITHMS;

  record EXTERNAL
    ExternalDecl       externalDecl;
    Option<Annotation> annotation_;
```

```
  end EXTERNAL;

end ClassPart;
```

ElementItem:

An element item is either an element or an annotation

```
uniontype ElementItem

  record ELEMENTITEM
    Element element;
  end ELEMENTITEM;

  record ANNOTATIONITEM
    Annotation annotation_;
  end ANNOTATIONITEM;

end ElementItem;
```

Element:

The basic element type in Modelica.

```
uniontype Element

  record ELEMENT
    Boolean final_;
    Option<RedeclareKeywords> redeclareKeywords "i.e., replaceable or redeclare" ;
    InnerOuter innerOuter " inner / outer" ;
    Ident      name;
    ElementSpec specification " Actual element specification" ;
    Info      info "The File name the class is defined in + line no + column no" ;
    Option<ConstrainClass> constrainClass "only valid for classdef and component";
  end ELEMENT;

  record TEXT
    Option<Ident> optName " optional name of text, e.g. model with syntax error.
                            We need the name to be able to browse it..." ;
    String   string;
    Info     info;
  end TEXT;

end Element;
```

Constraining type:

Constraining type (i.e., not inheritance), specified using the extends keyword.

```
uniontype ConstrainClass

  record CONSTRAINCLASS
    ElementSpec      elementSpec "must be extends" ;
    Option<Comment>  comment;
  end CONSTRAINCLASS;

end ConstrainClass;
```

ElementSpec:

An element is something that occurs in a public or protected section in a class definition. There is one constructor in the `ElementSpec` type for each possible element type. There are class definitions (`CLASSDEF`), extends clauses (`EXTENDS`) and component declarations (`COMPONENTS`).

As an example, if the element `extends TwoPin;` appears in the source, it is represented in the AST as `EXTENDS(IDENT("TwoPin"),{} )`.

```
uniontype ElementSpec

  record CLASSDEF
    Boolean replaceable_  "true if replaceable";
    Class   class_;
  end CLASSDEF;

  record EXTENDS
    Path             path;
    list<ElementArg> elementArg;
  end EXTENDS;

  record IMPORT
    Import          import_;
    Option<Comment> comment;
  end IMPORT;

  record COMPONENTS
    ElementAttributes   attributes;
    Path                typeName;
    list<ComponentItem> components;

  end COMPONENTS;

end ElementSpec;
```

InnerOuter:

One of the keywords `inner` or `outer` or the combination `inner outer` can be given to reference an inner, outer or inner outer component. Thus there are four disjoint possibilities.

```
uniontype InnerOuter

  record INNER end INNER;

  record OUTER end OUTER;

  record INNEROUTER end INNEROUTER;

  record UNSPECIFIED end UNSPECIFIED;

end InnerOuter;
```

Import:

Import statements of different kinds.

```
uniontype Import

  record NAMED_IMPORT
    Ident name "name" ;
    Path path "path" ;
  end NAMED_IMPORT;

  record QUAL_IMPORT
    Path path "path" ;
  end QUAL_IMPORT;

  record UNQUAL_IMPORT
```

```
    Path path "path" ;
  end UNQUAL_IMPORT;

end Import;
```

ComponentItem:

Collection of component and an optional comment.

```
uniontype ComponentItem

  record COMPONENTITEM
    Component                   component;
    Option<ComponentCondition> condition;
    Option<Comment>             comment;
  end COMPONENTITEM;

end ComponentItem;
```

ComponentCondition:

A ComponentItem can have a condition that must be fulfilled if the component should be instantiated.

```
type ComponentCondition = Exp;
```

Component:

A component represents some kind of Modelica entity (object or variable). Note that several component declarations can be grouped together in one `ElementSpec` by writing them in the same declaration in the source. However, this type contains the information specific to one component.

```
uniontype Component

  record COMPONENT
    Ident                name      "component name" ;
    ArrayDim             arrayDim  "Array dimensions, if any" ;
    Option<Modification> modification "Optional modification" ;
  end COMPONENT;

end Component;
```

EquationItem:

```
uniontype EquationItem

  record EQUATIONITEM
    Equation         equation_;
    Option<Comment>  comment;
  end EQUATIONITEM;

  record EQUATIONITEMANN
    Annotation  annotation_;
  end EQUATIONITEMANN;

end EquationItem;
```

AlgorithmItem:

Info specific for an algorithm item.

```
uniontype AlgorithmItem

  record ALGORITHMITEM
    Algorithm        algorithm_;
    Option<Comment>  comment;
  end ALGORITHMITEM;

  record ALGORITHMITEMANN
    Annotation  annotation_;
  end ALGORITHMITEMANN;

end AlgorithmItem;
```

Equation:

Information on one (kind) of equation, different constructors for different kinds of equations

```
uniontype Equation

  record EQ_IF
    Exp                ifExp                "Conditional expression" ;
    list<EquationItem> equationTrueItems    "true branch" ;
    list<tuple<Exp, list<EquationItem>>> elseIfBranches;
    list<EquationItem> equationElseItems    "Standard 2-side eqn" ;
  end EQ_IF;

  record EQ_EQUALS
    Exp leftSide;
    Exp rightSide "rightSide Connect eqn" ;
  end EQ_EQUALS;

  record EQ_CONNECT
    ComponentRef connector1;
    ComponentRef connector2;
  end EQ_CONNECT;

  record EQ_FOR
    Ident forVariable;
    Exp forExp;
    list<EquationItem> forEquations;
  end EQ_FOR;

  record EQ_WHEN_E
    Exp  whenExp;
    list<EquationItem>  whenEquations;
    list<tuple<Exp, list<EquationItem>>>  elseWhenEquations;
  end EQ_WHEN_E;

  record EQ_NORETCALL
    Ident functionName;
    FunctionArgs functionArgs "fcalls without return value" ;
  end EQ_NORETCALL;

end Equation;
```

Algorithm:

The `Algorithm` type describes one algorithm statement in an algorithm section. It does not describe a whole algorithm. The reason this type is named like this is that the name of the grammar rule for algorithm statements is `algorithm`.

```
uniontype Algorithm
```

```
  record ALG_ASSIGN
    ComponentRef assignComponent;
    Exp value;
  end ALG_ASSIGN;

  record ALG_TUPLE_ASSIGN
    Exp tuple_;
    Exp value;
  end ALG_TUPLE_ASSIGN;

  record ALG_IF
    Exp ifExp;
    list<AlgorithmItem> trueBranch;
    list<tuple<Exp, list<AlgorithmItem>>> elseIfAlgorithmBranch;
    list<AlgorithmItem> elseBranch;
  end ALG_IF;

  record ALG_FOR
    Ident forVariable;
    Exp forStmt;
    list<AlgorithmItem> forBody;
  end ALG_FOR;

  record ALG_WHILE
    Exp whileStmt;
    list<AlgorithmItem> whileBody;
  end ALG_WHILE;

  record ALG_WHEN_A
    Exp whenStmt;
    list<AlgorithmItem> whenBody;
    list<tuple<Exp, list<AlgorithmItem>>> elseWhenAlgorithmBranch;
  end ALG_WHEN_A;

  record ALG_NORETCALL
    ComponentRef functionCall;
    FunctionArgs functionArgs " general fcalls without return value" ;
  end ALG_NORETCALL;

end Algorithm;
```

Modifications:

Modifications are described by the `Modification` type. There are two forms of modifications: redeclarations and component modifications.

```
uniontype Modification

  record CLASSMOD
    list<ElementArg>  elementArgLst;
    Option<Exp>       expOption;
  end CLASSMOD;

end Modification;
```

ElementArg:

Wrapper for things that modify elements, modifications and redeclarations.

```
uniontype ElementArg

  record MODIFICATION
    Boolean finalItem;
    Each each_;
```

```
    ComponentRef componentReg;
    Option<Modification> modification;
    Option<String> comment;
  end MODIFICATION;

  record REDECLARATION
    Boolean           finalItem;
    RedeclareKeywords redeclareKeywords "keywords redeclare, or replaceable" ;
    Each        each_;
    ElementSpec elementSpec;
    Option<ConstrainClass> constrainClass "class definition or declaration" ;
  end REDECLARATION;

end ElementArg;
```

RedeclareKeywords:

The keywords `redeclare` and `replaceable` can be given in three different combinations, each one by themselves or both combined.

```
uniontype RedeclareKeywords
  record REDECLARE end REDECLARE;
  record REPLACEABLE end REPLACEABLE;
  record REDECLARE_REPLACEABLE end REDECLARE_REPLACEABLE;
end RedeclareKeywords;
```

Each:

The `Each` attribute represented by the `each` keyword can be present in both `MODIFICATION`'s and `REDECLARATION`'s.

```
uniontype Each
  record EACH end EACH;
  record NON_EACH end NON_EACH;
end Each;
```

ElementAttributes:

This represents component attributes which are properties of components which are applied by type prefixes. As an example, declaring a component as `input Real x;` will give the attributes `ATTR( {},false,VAR,INPUT)`.

```
uniontype ElementAttributes

  record ATTR
    Boolean flow_ "flow" ;
    Variability variability "variability ; parameter, constant etc." ;
    Direction direction "direction" ;
    ArrayDim arrayDim "arrayDim" ;
  end ATTR;

end ElementAttributes;
```

Variability:

Component/variable attribute variability:

```
uniontype Variability
  record VAR end VAR;
  record DISCRETE end DISCRETE;
  record PARAM end PARAM;
```

```
  record CONST end CONST;
end Variability;
```

Direction:

Component/variable attribute `Direction`.

```
uniontype Direction
  record INPUT end INPUT;
  record OUTPUT end OUTPUT;
  record BIDIR end BIDIR;
end Direction;
```

ArrayDim:

Array dimensions are specified by the type `ArrayDim`. Components in Modelica can be scalar or arrays with one or more dimensions. This datatype is used to indicate the dimensionality of a component or a type definition.

```
type ArrayDim = list<Subscript>;
```

Exp:

The `Exp` datatype is the container for representing a Modelica expression.

```
uniontype Exp

  record INTEGER
    Integer value;
  end INTEGER;

  record REAL
    Real value;
  end REAL;

  record CREF
    ComponentRef componentReg;
  end CREF;

  record STRING
    String value;
  end STRING;

  record BOOL
    Boolean value ;
  end BOOL;

  record BINARY  "Binary operations, e.g. a*b, a+b, etc."
    Exp       exp1;
    Operator  op;
    Exp       exp2;
  end BINARY;

  record UNARY  "Unary operations, e.g. -(x)"
    Operator  op;
    Exp       exp;
  end UNARY;

  record LBINARY  "Logical binary operations: and, or"
    Exp       exp1;
    Operator  op;
    Exp       exp2;
  end LBINARY;
```

```
  record LUNARY  "Logical unary operations: not"
    Operator  op;
    Exp       exp;
  end LUNARY;

  record RELATION  "Relations, e.g. a >= 0"
    Exp       exp1;
    Operator  op;
    Exp       exp2  ;
  end RELATION;

  record IFEXP  "If expressions"
    Exp   ifExp;
    Exp   trueBranch;
    Exp   elseBranch;
    list<tuple<Exp, Exp>>  elseIfBranch  ;
  end IFEXP;

  record CALL  "Function calls"
    ComponentRef function_;
    FunctionArgs functionArgs  ;
  end CALL;

  record ARRAY  "Array construction using { } or array()"
    list<Exp> arrayExp  ;
  end ARRAY;

  record MATRIX  "Matrix construction using [ ]"
    list<list<Exp>>  matrix;
  end MATRIX;

  record RANGE  "matrix Range expressions, e.g. 1:10 or 1:0.5:10"
    Exp         start;
    Option<Exp> step;
    Exp         stop;
  end RANGE;

  record TUPLE  "Tuples used in function calls returning several values"
    list<Exp>  expressions;
  end TUPLE;

  record END  "Array access operator for last element, e.g. a[end]:=1;"
  end END;

  record CODE  "Modelica AST Code constructors"
    Code  code;
  end CODE;

end Exp;
```

Code:

The `Code` datatype is a proposed meta-programming extension of Modelica. It originates from the Code quoting mechanism, see paper in the Modelica'2003 conference.

```
 uniontype Code

  record C_TYPENAME
    Path  path;
  end C_TYPENAME;

  record C_VARIABLENAME
    ComponentRef  componentRef;
```

```
  end C_VARIABLENAME;

  record C_EQUATIONSECTION
    Boolean             boolean;
    list<EquationItem>  equationItemLst;
  end C_EQUATIONSECTION;

  record C_ALGORITHMSECTION
    Boolean              boolean;
    list<AlgorithmItem>  algorithmItemLst;
  end C_ALGORITHMSECTION;

  record C_ELEMENT
    Element  element;
  end C_ELEMENT;

  record C_EXPRESSION
    Exp  exp;
  end C_EXPRESSION;

  record C_MODIFICATION
    Modification  modification;
  end C_MODIFICATION;

end Code;
```

FunctionArgs:

The `FunctionArgs` datatype consists of a list of positional arguments followed by a list of named arguments.

```
uniontype FunctionArgs

  record FUNCTIONARGS
    list<Exp>       args;
    list<NamedArg>  argNames;
  end FUNCTIONARGS;

  record FOR_ITER_FARG
    Exp    from;
    Ident  var;
    Exp    to;
  end FOR_ITER_FARG;

end FunctionArgs;
```

NamedArg:

The `NamedArg` datatype consist of an Identifier for the argument and an expression giving the value of the argument.

```
uniontype NamedArg

  record NAMEDARG
    Ident argName "argName" ;
    Exp argValue "argValue" ;
  end NAMEDARG;

end NamedArg;
```

Operator:

The `Operator` type can represent all the expression operators, binary or unary.

```
uniontype Operator "Expression operators"
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
  record POW end POW;
  record UPLUS end UPLUS;
  record UMINUS end UMINUS;
  record AND end AND;
  record OR end OR;
  record NOT end NOT;
  record LESS end LESS;
  record LESSEQ end LESSEQ;
  record GREATER end GREATER;
  record GREATEREQ end GREATEREQ;
  record EQUAL end EQUAL;
  record NEQUAL end NEQUAL;
end Operator;
```

Subscript:

The `Subscript` data type is used both in array declarations and component references. This might seem strange, but it is inherited from the grammar. The `NOSUB` constructor means that the dimension size is undefined when used in a declaration, and when it is used in a component reference it means a slice of the whole dimension.

```
uniontype Subscript

  record NOSUB end NOSUB;

  record SUBSCRIPT
    Exp subScript "subScript" ;
  end SUBSCRIPT;

end Subscript;
```

ComponentRef:

A component reference is the fully or partially qualified name of a component. It is represented as a list of identifier-subscript pairs.

```
uniontype ComponentRef

  record CREF_QUAL
    Ident            name;
    list<Subscript>  subScripts;
    ComponentRef     componentRef;
  end CREF_QUAL;

  record CREF_IDENT
    Ident            name;
    list<Subscript>  subscripts;
  end CREF_IDENT;

end ComponentRef;
```

Path:

The type `Path` is used to store references to class names, or names inside class definitions.

```
uniontype Path
```

```
  record QUALIFIED
    Ident  name;
    Path   path;
  end QUALIFIED;

  record IDENT
    Ident  name;
  end IDENT;

end Path;
```

Restrictions:

These constructors each correspond to a different kind of class declaration in Modelica, except the last four, which are used for the predefined types. The parser assigns each class declaration one of the restrictions, and the actual class definition is checked for conformance during translation. The predefined types are created in the `Builtin` module and are assigned special restrictions.

```
uniontype Restriction
  record R_CLASS end R_CLASS;
  record R_MODEL end R_MODEL;
  record R_RECORD end R_RECORD;
  record R_BLOCK end R_BLOCK;
  record R_CONNECTOR end R_CONNECTOR;
  record R_EXP_CONNECTOR end R_EXP_CONNECTOR;
  record R_TYPE end R_TYPE;
  record R_PACKAGE end R_PACKAGE;
  record R_FUNCTION end R_FUNCTION;
  record R_ENUMERATION end R_ENUMERATION;
  record R_PREDEFINED_INT end R_PREDEFINED_INT;
  record R_PREDEFINED_REAL end R_PREDEFINED_REAL;
  record R_PREDEFINED_STRING end R_PREDEFINED_STRING;
  record R_PREDEFINED_BOOL end R_PREDEFINED_BOOL;
  record R_PREDEFINED_ENUM end R_PREDEFINED_ENUM;
end Restriction;
```

Annotation:

An Annotation is a class_modification.

```
uniontype Annotation

  record ANNOTATION
    list<ElementArg>  elementArgs;
  end ANNOTATION;

end Annotation;
```

Comment:

```
uniontype Comment

  record COMMENT
    Option<Annotation>  annotation_;
    Option<String>      comment;
  end COMMENT;

end Comment;
```

ExternalDecl:

The type ExternalDecl is used to represent declaration of an external function wrapper.

```
uniontype ExternalDecl

  record EXTERNALDECL
    Option<Ident>       funcName  "The name of the external function" ;
    Option<String>      lang      "Language of the external function" ;
    Option<ComponentRef> output_  "output parameter as return value" ;
    list<Exp>           args   "only positional arguments, i.e. expression list" ;
    Option<Annotation>  annotation_;
  end EXTERNALDECL;

end ExternalDecl;
```

Dependencies:

Module dependencies of the Absyn module: Debug, Dump, Util, Print.

3.4.2 Algorithm – Data Types and Functions for Algorithm Sections

This module contains data types and functions for managing algorithm sections. The algorithms in the AST are analyzed by the Inst module which uses this module to represent the algorithm sections. No processing of any kind, except for building the data structure is done in this module. It is used primarily by the Inst module which both provides its input data and uses its "output" data.

Module dependencies: Exp, Types, SCode, Util, Print, Dump, Debug.

3.4.3 Builtin – Builtin Types and Variables

This module defines the builtin types, variables and functions in Modelica. The only exported functions are `initial_env` and `simple_initial_env`. There are several builtin attributes defined in the builtin types, such as unit, start, etc.

Module dependencies: Absyn, SCode, Env, Types, ClassInf, Debug, Print.

3.4.4 Ceval – Constant Evaluation of Expressions and Command Interpretation

This module handles constant propagation and expression evaluation, as well as interpretation and execution of user commands, e.g. plot(...). When elaborating expressions, in the Static module, expressions are checked to find out their type. This module also checks whether expressions are constant. In such as case the function `ceval` in this module will then evaluate the expression to a constant value, defined in the Values module.

Input:

Env: Environment with bindings.
Exp: Expression to check for constant evaluation.
Bool flag determines whether the current instantiation is implicit.
InteractiveSymbolTable is optional, and used in interactive mode, e.g. from mosh.

Output:

Value: The evaluated value
InteractiveSymbolTable: Modified symbol table.

Subscript list : Evaluates subscripts and generates constant expressions.

Module dependencies: Absyn, Env, Exp, Interactive, Values, Static, Print, Types, ModUtil, System, SCode, Inst, Lookup, Dump, DAE, Debug, Util, Modsim, ClassInf, RTOpts, Parse, Prefix, Codegen, ClassLoader.

3.4.5 ClassInf – Inference and Check of Class Restrictions

This module deals with class inference, i.e., determining if a class definition adheres to one of the class restrictions, and, if specifically declared in a restricted form, if it breaks that restriction.

The inference is implemented as a finite state machine. The function `start` initializes a new machine, and the function `trans` signals transitions in the machine. Finally, the state can be checked against a restriction with the `valid` function.

Module dependencies: Absyn, SCode, Print.

3.4.6 ClassLoader – Loading of Classes from $OPENMODELICALIBRARY

This module loads classes from $OPENMODELICALIBRARY. It exports only one function: the `loadClassClass` function. It is used by module Ceval when using the `loadClass` function in the interactive environment.

Module dependencies: Absyn, System, Lookup, Interactive, Util, Parse, Print, Env, Dump.

3.4.7 Codegen – Generate C Code from DAE

Generate C code from DAE (Flat Modelica) for Modelica functions and algorithms (SimCodeGen is generating code from equations). This code is compiled and linked to the simulation code or when functions are called from the interactive environment.

Input: DAE

Output: (generated code output by the Print module)

Module dependencies: Absyn, Exp, Types, Inst, DAE, Print, Util, ModUtil, Algorithm, ClassInf, Dump, Debug.

3.4.8 Connect – Connection Set Management

Connections generate connection sets (represented using the datatype `Set` defined in this module) which are constructed during code instantiation. When a connection set is generated, it is used to create a number of equations. The kind of equations created depends on the type of the set.

The Connect module is called from the Inst module and is responsible for creation of all connect-equations later passed to the DAE module.

Module dependencies: Exp, Env, Static, DAE.

3.4.9 Corba – Modelica Compiler Corba Communication Module

The Corba actual implementation differs between Windows and Unix versions. The Windows implementation is located in `./winruntime` and the Unix version lies in `./runtime`.

OpenModelica does not in itself include a complete CORBA implementation. You need to download one, for example MICO from `http://www.mico.org`. There also exists some options that can be sent to `configure` concerning the usage of CORBA:

- `--with-CORBA=/location/of/corba/library`
- `--without-CORBA`

No module dependencies.

3.4.10 DAE – DAE Equation Management and Output

This module defines data structures for DAE equations and declarations of variables and functions. It also exports some help functions for other modules. The DAE data structure is the result of flattening, containing only flat Modelica, i.e., equations, algorithms, variables and functions.

```
uniontype DAElist "A DAElist is a list of Elements. Variables, equations,
                   functions, algorithms, etc. are all found in this list."
  record DAE
    list<Element> elementLst;
  end DAE;

end DAElist;

type Ident = String;
type InstDims = list<Exp.Subscript>;
type StartValue = Option<Exp.Exp>;

uniontype VarKind
  record VARIABLE end VARIABLE;
  record DISCRETE end DISCRETE;
  record PARAM end PARAM;
  record CONST end CONST;
end VarKind;

uniontype Type
  record REAL end REAL;
  record INT end INT;
  record BOOL end BOOL;
  record STRING end STRING;
  record ENUM end ENUM;

  record ENUMERATION
    list<String> stringLst;
  end ENUMERATION;

end Type;

uniontype Flow "The Flow of a variable indicates if it is a Flow variable or not,
or if
   it is not a connector variable at all."
  record FLOW end FLOW;
  record NON_FLOW end NON_FLOW;
  record NON_CONNECTOR end NON_CONNECTOR;
end Flow;

uniontype VarDirection
  record INPUT end INPUT;
  record OUTPUT end OUTPUT;
  record BIDIR end BIDIR;
end VarDirection;

uniontype Element
```

```
record VAR
  Exp.ComponentRef componentRef;
  VarKind          varible "variable name" ;
  VarDirection     variable "variable, constant, parameter, etc." ;
  Type             input_ "input, output or bidir" ;
  Option<Exp.Exp>  one "one of the builtin types" ;
  InstDims         binding "Binding expression e.g. for parameters" ;
  StartValue       dimension "dimension of original component" ;
  Flow             value "value of start attribute" ;
  list<Absyn.Path> flow_ "Flow of connector variable. Needed for
                                    unconnected flow variables" ;
  Option<VariableAttributes> variableAttributesOption;
  Option<Absyn.Comment> absynCommentOption;
end VAR;

record DEFINE
  Exp.ComponentRef componentRef;
  Exp.Exp exp;
end DEFINE;

record INITIALDEFINE
  Exp.ComponentRef componentRef;
  Exp.Exp exp;
end INITIALDEFINE;

record EQUATION
  Exp.Exp exp;
  Exp.Exp scalar "Scalar equation" ;
end EQUATION;

record ARRAY_EQUATION
  list<Integer> dimension "dimension sizes" ;
  Exp.Exp exp;
  Exp.Exp array "array equation" ;
end ARRAY_EQUATION;

record WHEN_EQUATION
  Exp.Exp condition "Condition" ;
  list<Element> equations "Equations" ;
  Option<Element> elsewhen_ "Elsewhen should be of type" ;  end WHEN_EQUATION;

record IF_EQUATION
  Exp.Exp condition1 "Condition" ;
  list<Element> equations2 "Equations of true branch" ;
  list<Element> equations3 "Equations of false branch" ;
end IF_EQUATION;

record INITIAL_IF_EQUATION
  Exp.Exp condition1 "Condition" ;
  list<Element> equations2 "Equations of true branch" ;
  list<Element> equations3 "Equations of false branch" ;
end INITIAL_IF_EQUATION;

record INITIALEQUATION
  Exp.Exp exp1;
  Exp.Exp exp2;
end INITIALEQUATION;

record ALGORITHM
  Algorithm.Algorithm algorithm_;
end ALGORITHM;

record INITIALALGORITHM
  Algorithm.Algorithm algorithm_;
end INITIALALGORITHM;
```

```
  record COMP
    Ident ident;
    DAElist dAElist "a component with subelements, normally
                     only used at top level." ;
  end COMP;

  record FUNCTION
    Absyn.Path path;
    DAElist dAElist;
    Types.Type type_;
  end FUNCTION;

  record EXTFUNCTION
    Absyn.Path path;
    DAElist dAElist;
    Types.Type type_;
    ExternalDecl externalDecl;
  end EXTFUNCTION;

  record ASSERT
    Exp.Exp exp;
  end ASSERT;

  record REINIT
    Exp.ComponentRef componentRef;
    Exp.Exp exp;
  end REINIT;

end Element;

uniontype VariableAttributes
  record VAR_ATTR_REAL
    Option<String> quantity "quantity" ;
    Option<String> unit "unit" ;
    Option<String> displayUnit "displayUnit" ;
    tuple<Option<Real>, Option<Real>> min "min , max" ;
    Option<Real> initial_ "Initial value" ;
    Option<Boolean> fixed "fixed - true: default for parameter/constant, false -
default for other variables" ;
    Option<Real> nominal "nominal" ;
    Option<StateSelect> stateSelectOption;
  end VAR_ATTR_REAL;

  record VAR_ATTR_INT
    Option<String> quantity "quantity" ;
    tuple<Option<Integer>, Option<Integer>> min "min , max" ;
    Option<Integer> initial_ "Initial value" ;
    Option<Boolean> fixed "fixed - true: default for parameter/constant, false -
default for other variables" ;
  end VAR_ATTR_INT;

  record VAR_ATTR_BOOL
    Option<String> quantity "quantity" ;
    Option<Boolean> initial_ "Initial value" ;
    Option<Boolean> fixed "fixed - true: default for parameter/constant, false -
default for other variables" ;
  end VAR_ATTR_BOOL;

  record VAR_ATTR_STRING
    Option<String> quantity "quantity" ;
    Option<String> initial_ "Initial value" ;
  end VAR_ATTR_STRING;

  record VAR_ATTR_ENUMERATION
```

```
    Option<String> quantity "quantity" ;
    tuple<Option<Exp.Exp>, Option<Exp.Exp>> min "min , max" ;
    Option<Exp.Exp> start "start" ;
    Option<Boolean> fixed "fixed - true: default for parameter/constant, false -
default for other variables" ;
  end VAR_ATTR_ENUMERATION;

end VariableAttributes;

uniontype StateSelect
  record NEVER end NEVER;
  record AVOID end AVOID;
  record DEFAULT end DEFAULT;
  record PREFER end PREFER;
  record ALWAYS end ALWAYS;
end StateSelect;

uniontype ExtArg
  record EXTARG
    Exp.ComponentRef componentRef;
    Types.Attributes attributes;
    Types.Type type_;
  end EXTARG;

  record EXTARGEXP
    Exp.Exp exp;
    Types.Type type_;
  end EXTARGEXP;

  record EXTARGSIZE
    Exp.ComponentRef componentRef;
    Types.Attributes attributes;
    Types.Type type_;
    Exp.Exp exp;
  end EXTARGSIZE;

  record NOEXTARG end NOEXTARG;

end ExtArg;

uniontype ExternalDecl
  record EXTERNALDECL
    Ident ident;
    list<ExtArg>  external_ "external function name" ;
    ExtArg        parameters "parameters" ;
    String        return "return type" ;
    Option<Absyn.Annotation> language "language e.g. Library" ;
  end EXTERNALDECL;

end ExternalDecl;
```

Som of the more important functions for unparsing (dumping) flat Modelica in DAE form:

The function `dump` unparses (converts into string or prints) a `DAElist` into the standard output format by calling `dumpFunctionFunction` and `dumpCompElement`. We also have (?? explain more):

```
dumpStrStr: DAElist => string
dumpGraphvizGraphviz: DAElist => ()
dumpDebugDebug
```

`dumpCompElement` (classes) calls `dumpElementsElements`, which calls:

```
dumpVarsVars
dumpListList equations
dumpListList algorithm
```

```
dumpListList compElement (classes)
...
```

Module dependencies: Absyn, Exp, Algorithm, Types, Values.

3.4.11 DAEEXT – External Utility Functions for DAE Management

The DAEEXT module is an externally implemented module (in file `runtime/daeext.cpp`) used for the BLT and index reduction algorithms in DAELow. The implementation mainly consists of bit vector datatypes and operations implemented using `std::vector<bool>` since such functionality is not available in MetaModelica.

No module dependencies.

3.4.12 DAELow – Lower Level DAE Using Sparse Matrises for BLT

This module handles a lowered form of a DAE including equations, simple equations with equal operator only, and algorithms, in three separate lists: equations, simple equations, algorithms. The variables are divided into two groups: 1) known variables, parameters, and constants; 2) unknown variables including state variables and algebraic variables.

The module includes the BLT sorting algorithm which sorts the equations into blocks, and the index reduction algorithm using dummy derivatives for solving higher index problems. It also includes an implementation of the Tarjan algorithm to detect strongly connected components during the BLT sorting.

Module dependencies: DAE, Exp, Values, Absyn, Algorithm.

3.4.13 Debug – Trace Printing Used for Debugging

Printing routines for debug output of strings. Also flag controlled printing. When flag controlled printing functions are called, printing is done only if the given flag is among the flags given in the runtime arguments to the compiler.

If the +d-flag, i.e., if `+d=inst,lookup` is given in the command line, only calls containing these flags will actually print something, e.g.: `fprint("inst", "Starting instantiation...")`. See `runtime/rtopts.c` for implementation of flag checking.

Module dependencies: Rtopts, Dump, Print.

3.4.14 Derive – Differentiation of Equations from DAELow

This module is responsible for symbolic differentiation of equations and expressions. It is currently (2004-09-28) only used by the solve function in the Exp module for solving equations.

The symbolic differentiation is used by the Newton-Raphson method and by the index reduction.

Module dependencies: DAELow, Exp, Absyn, Util, Print.

3.4.15 DFA – MetaModelica Pattern Matching

This module is part of the MetaModelica language extension. This module contains a deterministic finite automata (DFA) and a matrix data structure. These are used by the pattern match algorithm found in Patternm. There are also several functions for handling DFAs (for instance a function for adding a new arc

to a DFA) and matrices (functions for adding a row to matrix, singling out the first row of a matrix, removing the first row of a matrix, etc.). The union type `RenamedPat` can also be found in this module.

A renamed pattern is a pattern (an Absyn expression) tagged with a variable name (an Absyn identifier).

This module also contains the functions that transforms a DFA into a value block expression with nested if-elseif-else nodes. The function `fromDFAtoIfNodes` is the entry point for this transformation; `generateIfElseifAndElse`, `fromStatetoAbsynCode`, etc. are then invoked.

See the OMC MetaModelica extension chapter (chapter 4) for more information.

Module dependencies: Absyn, Util, Env, Lookup, Types, SCode, ClassInf

3.4.16 Dump – Abstract Syntax Unparsing/Printing

Printing routines for unparsing and debugging of the AST. These functions do nothing but print the data structures to the standard output.

The main entry point for this module is the function `dump` which takes an entire program as an argument, and prints it all in Modelica source form. The other interface functions can be used to print smaller portions of a program.

Module dependencies: Absyn, Interactive, ClassInf, Rtopts, Print, Util, Debug..

3.4.17 DumpGraphviz – Dump Info for Graph visualization of AST

Print the abstract syntax into a text form that can be read by the GraphViz tool (www.graphviz.org) for drawing abstract syntax trees.

Module dependencies: Absyn, Debug, Graphviz, ClassInf, Dump.

3.4.18 Env – Environment Management

This module contains functions and data structures for environment management.

"Code instantiation is made in a context which consists of an *environment* an an *ordered set of parents*", according to the Modelica Specification

An environment is a stack of frames, where each frame contains a number of class and variable bindings. Each frame consist of the following:

- A frame name (corresponding to the class partially instantiated in that frame).
- A binary tree/hash table?? containing a list of classes.
- A binary tree/hash table?? containing a list of functions (functions are overloaded so that several identical function names corresponding to different functions can exist).
- A list of unnamed items consisting of import statements.

```
type Env = list<Frame>;

uniontype Frame
 record FRAME
  Option<Ident> class_1 "Class name" ;
    BinTree list_2 "List of uniquely named classes and variables" ;
    BinTree list_3 "List of types, which DOES NOT be uniquely named, eg. size have
several types" ;
  list<Item> list_4 "list of unnamed items (imports)" ;
    list<Frame> list_5 "list of frames for inherited elements" ;
    list<Exp.ComponentRef> current6 "current connection set crefs" ;
    Boolean encapsulated_7 "encapsulated bool=true means that FRAME is created due
to encapsulated class" ;
  end FRAME;
```

```
end Frame;

uniontype Item
  record VAR
    Types.Var instantiated "instantiated component" ;
    Option<tuple<SCode.Element, Types.Mod>> declaration "declaration if not fully
instantiated." ;
    Boolean if_ "if it typed/fully instantiated or not" ;
    Env env "The environment of the instantiated component
                        Contains e.g. all sub components
                  " ;
  end VAR;

  record CLASS
    SCode.Class class_;
    Env env;
  end CLASS;

  record TYPE
    list<Types.Type> list_ "list since several types with the same name can exist
in the same scope (overloading)" ;
  end TYPE;

  record IMPORT
    Absyn.Import import_;
  end IMPORT;

end Item;
```

The binary tree data structure `BinTree` used for the environment is generic and can be used in any application. It is defined as follows:

```
uniontype BinTree "The binary tree data structure
  The binary tree data structure used for the environment is generic and can
  be used in any application."
  record TREENODE
    Option<TreeValue> value  "Value" ;
    Option<BinTree>   left   "left subtree" ;
    Option<BinTree>   right  "right subtree" ;
  end TREENODE;

end BinTree;
```

Each node in the binary tree can have a value associated with it.

```
uniontype TreeValue
  record TREEVALUE
    Key key;
    Value value;
  end TREEVALUE;

end TreeValue;

type Key = Ident "Key" ;

type Value = Item;

constant Env emptyEnv;
```

As an example lets consider the following Modelica code:

```
package A
  package B
    import Modelica.SIunits.*;
    constant Voltage V=3.3;

    function foo
    end foo;

    model M1
      Real x,y;
    end M1;

    model M2
    end M2;

  end B;
end A;
```

When instantiating `M1` we will first create the environment for its surrounding scope by a recursive instantiation on `A.B` giving the environment:

```
{
  FRAME("A", {Class:B},{},{},false) ,
  FRAME("B", {Class:M1, Class:M2, Variable:V}, {Type:foo},
         {import Modelica.SIunits.*},false)
}
```

Then, the class `M1` is instantiated in a new scope/Frame giving the environment:

```
{
  FRAME("A", {Class:B},{},{},false) ,
  FRAME("B", {Class:M1, Class:M2, Variable:V}, {Type:foo},
    {Import Modelica.SIunits.*},false),
  FRAME("M1, {Variable:x, Variable:y},{},{},false)
}
```

Note: The instance hierarchy (components and variables) and the class hierarchy (packages and classes) are combined into the same data structure, enabling a uniform lookup mechanism.

The most important functions in Env:

```
  function newFrame : (Boolean) => Frame
  function openScope      : (Env,Boolean, Option<Ident>) => Env
  function extendFrameC   : (Env, SCode.Class) => Env
  function extendFrameClasses   : (Env, SCode.Program) => Env
  function extendFrameV   : (Env, Types.Var,
                          Option<tuple<SCode.Element,Types.Mod>>, Boolean) => Env
  function updateFrameV   : (Env, Types.Var,bool) => Env
  function extendFrameT : (Env,Ident,Types.Type) => Env
  function extendFrameI : (Env, Absyn.Import) => Env
  function topFrame : Env => Frame
  function getEnvPath: (Env) => Absyn.Path option
```

Module dependencies: Absyn, Values, SCode, Types, ClassInf, Exp, Dump, Graphviz, DAE, Print, Util, System.

3.4.19 Exp – Expression Handling after Static Analysis

This file contains the module Exp, which contains data types for describing expressions, after they have been examined by the static analyzer in the module Static. There are of course great similarities with the expression types in the Absyn module, but there are also several important differences.

No overloading of operators occur, and subscripts have been checked to see if they are slices. Deoverloading of overloaded operators such as `ADD` (+) is performed, e.g. to operations `ADD_ARR`, `ADD(REAL)`, `ADD(INT)`. Slice operations are also identified, e.g.:

```
model A Real b; end A;

model B
  A a[10];
equation
  a.b=fill(1.0,10); // a.b is a slice
end B;
```

All expressions are also type consistent, and all implicit type conversions in the AST are made explicit here, e.g. `Real(1)+1.5` converted from `1+1.5`.

Functions:

Some expression simplification and solving is also done here. This is used for symbolic transformations before simulation, in order to rearrange equations into a form needed by simulation tools. The functions `simplify`, `solve`, `expContainsContains`, `expEqual`, `extendCref`, etc. perform this functionality, e.g.:

```
extendCrefCref (ComponentRef, Ident, list<Subscript>) => ComponentRef
simplify(Exp) => Exp
```

The `simplify` function simplifies expressions that have been generated in a complex way, i.e., not a complete expression simplification mechanism.

This module also contains functions for printing expressions, for IO, and for conversion to strings. Moreover, graphviz output is supported.

Identifiers :

```
type Ident = String;
```

Define `Ident` as an alias for `String` and use it for all identifiers in Modelica.

Basic types:

```
uniontype Type
  record INT end INT;
  record REAL end REAL;
  record BOOL end BOOL;
  record STRING end STRING;
  record ENUM end ENUM;
  record OTHER "e.g. complex types, etc." end OTHER;

  record T_ARRAY
    Type type_;
    list<Integer> arrayDimensions;
  end T_ARRAY;

end Type;
```

These basic types are not used as expression types (see the Types module for expression types). They are used to parameterize operators which may work on several simple types.

Expressions:

The `Exp` union type closely corresponds to the `Absyn.Exp` union type, but is used for statically analyzed expressions. It includes explicit type promotions and typed (non-overloaded) operators. It also contains expression indexing with the `ASUB` constructor. Indexing arbitrary array expressions is currently not supported in Modelica, but it is needed here.

```
uniontype Exp "Expressions
  record ICONST
    Integer integer "Integer constants" ;
  end ICONST;
```

```
record RCONST
  Real real "Real constants" ;
end RCONST;

record SCONST
  String string "String constants" ;
end SCONST;

record BCONST
  Boolean bool "Bool constants" ;
end BCONST;

record CREF
  ComponentRef componentRef;
  Type component "component references, e.g. a.b[2].c[1]" ;
end CREF;

record BINARY
  Exp exp;
  Operator operator;
  Exp binary "Binary operations, e.g. a+4" ;
end BINARY;

record UNARY
  Operator operator;
  Exp unary "Unary operations, -(4x)" ;
end UNARY;

record LBINARY
  Exp exp;
  Operator operator;
  Exp logical "Logical binary operations: and, or" ;
end LBINARY;

record LUNARY
  Operator operator;
  Exp logical "Logical unary operations: not" ;
end LUNARY;

record RELATION
  Exp exp;
  Operator operator;
  Exprelation_ "Relation, e.g. a <= 0" ;
end RELATION;

record IFEXP
  Exp exp1;
  Expexp2;
  Exp if_3 "If expressions" ;
end IFEXP;

record CALL
  Absyn.Path path;
  list<Exp> expLst;
  Boolean tuple_ "tuple" ;
  Boolean builtin "builtin Function call" ;
end CALL;

record ARRAY
  Type type_;
  Boolean scalar "scalar for codegen" ;
  list<Exp> array "Array constructor, e.g. {1,3,4}" ;
end ARRAY;

record MATRIX
```

```
    Type type_;
    Integer integer;
    list<list<tuple<Exp, Boolean>>> scalar "Matrix constructor. e.g. [1,0;0,1]" ;
  end MATRIX;

  record RANGE
    Type type_;
   exp;
    Option<Exp> expOption;
    Exp range "Range constructor, e.g. 1:0.5:10" ;
  end RANGE;

  record TUPLE
    list<Exp> PR "PR. Tuples, used in func calls returning several
                                                        arguments" ;
  end TUPLE;

  record CAST
    Type type_;
    Exp cast "Cast operator" ;
  end CAST;

  record ASUB
    Exp exp;
    Integer array "Array subscripts" ;
  end ASUB;

  record SIZE
    Exp exp;
    Option<Exp> the "The ssize operator" ;
  end SIZE;

  record CODE
    Absyn.Code code;
    Type modelica "Modelica AST constructor" ;
  end CODE;

  record REDUCTION
    Absyn.Path path;
    Exp expr "expr" ;
    Ident ident;
    Exp range "range Reduction expression" ;
  end REDUCTION;

  record END "array index to last element, e.g. a[end]:=1;" end END;

end Exp;
```

Operators:

Operators which are overloaded in the abstract syntax are here made type-specific. The `Integer` addition operator `ADD(INT)` and the `Real` addition operator `ADD(REAL)` are two distinct operators.

```
uniontype Operator

  record ADD
    Type type_;
  end ADD;

  record SUB
    Type type_;
  end SUB;

  record MUL
    Type type_;
```

```
end MUL;

record DIV
  Type type_;
end DIV;

record POW
  Type type_;
end POW;

record UMINUS
  Type type_;
end UMINUS;

record UPLUS
  Type type_;
end UPLUS;

record UMINUS_ARR
  Type type_;
end UMINUS_ARR;

record UPLUS_ARR
  Type type_;
end UPLUS_ARR;

record ADD_ARR
  Type type_;
end ADD_ARR;

record SUB_ARR
  Type type_;
end SUB_ARR;

record MUL_SCALAR_ARRAY
  Type a "a  { b, c }" ;
end MUL_SCALAR_ARRAY;

record MUL_ARRAY_SCALAR
  Type type_ "{a, b}  c" ;
end MUL_ARRAY_SCALAR;

record MUL_SCALAR_PRODUCT
  Type type_ "{a, b}  {c, d}" ;
end MUL_SCALAR_PRODUCT;

record MUL_MATRIX_PRODUCT
  Type type_ "{{..},..}  {{..},{..}}" ;
end MUL_MATRIX_PRODUCT;

record DIV_ARRAY_SCALAR
  Type type_ "{a, b} / c" ;
end DIV_ARRAY_SCALAR;

record POW_ARR
  Type type_;
end POW_ARR;

record AND end AND;

record OR end OR;

record NOT end NOT;

record LESS
```

```
    Type type_;
  end LESS;

  record LESSEQ
    Type type_;
  end LESSEQ;

  record GREATER
    Type type_;
  end GREATER;

  record GREATEREQ
    Type type_;
  end GREATEREQ;

  record EQUAL
    Type type_;
  end EQUAL;

  record NEQUAL
    Type type_;
  end NEQUAL;

  record USERDEFINED
    Absyn.Path the "The fully qualified name of the overloaded operator function";
  end USERDEFINED;

end Operator;
```

Component references:

```
uniontype ComponentRef "- Component references
    CREF_QUAL(...) is used for qualified component names, e.g. a.b.c
    CREF_IDENT(..) is used for non-qualifed component names, e.g. x "
  record CREF_QUAL
    Ident ident;
    list<Subscript> subscriptLst;
    ComponentRef componentRef;
  end CREF_QUAL;

  record CREF_IDENT
    Ident ident;
    list<Subscript> subscriptLst;
  end CREF_IDENT;

end ComponentRef;
```

The `Subscript` and `ComponentRef` datatypes are simple translations of the corresponding types in the Absyn module.

```
uniontype Subscript

  record WHOLEDIM "a[:,1]" end WHOLEDIM;

  record SLICE
    Exp a "a[1:3,1], a[1:2:10,2]" ;
  end SLICE;

  record INDEX
    Exp a "a[i+1]" ;
  end INDEX;

end Subscript;
```

Module dependencies: Absyn, Graphviz, Rtopts, Util, Print, ModUtil, Derive, System, Dump.

3.4.20 Graphviz – Graph Visualization from Textual Representation

Graphviz is a tool for drawing graphs from a textual representation. This module generates the textual input to Graphviz from a tree defined using the data structures defined here, e.g. Node for tree nodes. See http://www.research.att.com/sw/tools/graphviz/ .

Input: The tree constructed from data structures in Graphviz
Output: Textual input to graphviz, written to stdout.

3.4.21 Inst – Code Instantiation/Elaboration of Modelica Models

This module is responsible for code instantiation of Modelica models. Code instantiation is the process of elaborating and expanding the model component representation, flattening inheritance, and generating equations from connect equations.

The code instantiation process takes Modelica AST as defined in SCode and produces variables and equations and algorithms, etc. as defined in the DAE module

This module uses module Lookup to lookup classes and variables from the environment defined in Env. It uses the Connect module for generating equations from `connect` equations. The type system defined in Types is used for code instantiation of variables and types. The Mod module is used for modifiers and merging of modifiers.

3.4.21.1 Overview:

The Inst module performs most of the work of the *flattening* of models:

1. Build empty initial environment.
2. Code instantiate certain classes *implicitly*, e.g. functions.
3. Code instantiate (last class or a specific class) in a program explicitly.

The process of code instantiation consists of the following:

1. Open a new scope => a new environment
2. Start the class state machine to recognize a possible restricted class.
3. Instantiate class in environment.
4. Generate equations.
5. Read class state & generate Type information.

3.4.21.2 Code Instantiation of a Class in an Environment

(?? Add more explanations)

Function: `instClassdef`
PARTS: `instElementListList`
DERIVED (i.e `class A=B(mod);`):

1. `lookup` class
2. `elabModMod`
3. Merge modifications
4. `instClassIn` (…,mod, …)

3.4.21.3 InstElementListList & Removing Declare Before Use

The procedure is as follows:

1. First implicitly declare all local classes and add component names (calling `extendComponentsToEnvComponentsToEnv`), Also merge modifications (This is done by saving modifications in the environment and postponing to step 3, since type information is not yet available).
2. Expand all `extends` nodes.
3. Perform instantiation, which results in DAE elements.

Note: This is probably the most complicated parts of the compiler!

Design issue: How can we simplify this? The complexity is caused by the removal of Declare-before-use in combination with sequential translation structure (Absyn->Scode->(Exp,Mod,Env)).

3.4.21.4 The InstElement Function

This is a huge function to handle element instantiation in detail, including the following items:

- Handling `extends` clauses.
- Handling component nodes (the function `update_components_in_env` is called if used before it is declared).
- Elaborated dimensions (?? explain).
- `InstVar` called (?? explain).
- ClassDefs (?? explain).

3.4.21.5 The InstVar Function

The `instVar` function performs code instantiation of all subcomponents of a component. It also instantiates each array element as a scalar, i.e., expands arrays to scalars, e.g.:

`Real x[2]` => `Real x[1]; Real x[2];` in flat Modelica.

3.4.21.6 Dependencies

Module dependencies: Absyn, ClassInf, Connect, DAE, Env, Exp, SCode, Mod, Prefix, Types.

3.4.22 Interactive – Model Management and Expression Evaluation

This module contain functionality for model management, expression evaluation, etc. in the interactive environment. The module defines a symbol table used in the interactive environment containing the following:

- Modelica models (described using Absyn abstract syntax).
- Variable bindings.
- Compiled functions (so they do not need to be recompiled).
- Instantiated classes (that can be reused, not implemented. yet).
- Modelica models in SCode form (to speed up instantiation. not implemented. yet).

The most important data types:

```
uniontype InteractiveSymbolTable "The Interactive Symbol Table"
  record SYMBOLTABLE
    Absyn.Program ast                   "The ast" ;
    SCode.Program explodedAst           "The exploded ast" ;
    list<InstantiatedClass> instClsLst  "List of instantiated classes" ;
    list<InteractiveVariable> lstVarVal "List of variables with values" ;
    list<tuple<Absyn.Path, Types.Type>> compiledFunctions "List of compiled
                                 functions, fully qualified name + type" ;
  end SYMBOLTABLE;
end InteractiveSymbolTable;
```

```
uniontype InteractiveStmt "The Interactive Statement:
                           An Statement given in the interactive environment
                           can either be an Algorithm statement or an expression"
  record IALG
    Absyn.AlgorithmItem algItem;
  end IALG;

  record IEXP
    Absyn.Exp exp;
  end IEXP;
end InteractiveStmt;

uniontype InteractiveStmts "The Interactive Statements:
                            Several interactive statements are used in the
                            Modelica scripts"
  record ISTMTS
    list<InteractiveStmt> interactiveStmtLst "interactiveStmtLst" ;
    Boolean semicolon "when true, the result will not be shown in
                       the interactive environment" ;
  end ISTMTS;
end InteractiveStmts;


uniontype InstantiatedClass "The Instantiated Class"
  record INSTCLASS
    Absyn.Path qualName " The fully qualified name of the inst:ed class";
    list<DAE.Element> daeElementLst " The list of DAE elements";
    Env.Env env "The env of the inst:ed class";
  end INSTCLASS;
end InstantiatedClass;

uniontype InteractiveVariable "- Interactive Variable"
  record IVAR
    Absyn.Ident varIdent "The variable identifier";
    Values.Value value "The expression containing the value";
    Types.Type type_ " The type of the expression";
  end IVAR;
end InteractiveVariable;
```

Two of the more important functions and their input/output:

```
function evaluate
  input InteractiveStmts inInteractiveStmts;
  input InteractiveSymbolTable inInteractiveSymbolTable;
  input Boolean inBoolean;
  output String outString;
  output InteractiveSymbolTable outInteractiveSymbolTable;
algorithm
  ...
end evaluate;
```

```
function updateProgram
  input Absyn.Program inProgram1;
  input Absyn.Program inProgram2;
  output Absyn.Program outProgram;
algorithm
  ...
end updateProgram;
```

Module dependencies: Absyn, SCode, DAE, Types, Values, Env, Dump, Debug, Rtops, Util, Parse, Prefix, Mod, Lookup, ClassInf, Exp, Inst, Static, ModUtil, Codegen, Print, System, ClassLoader, Ceval.

3.4.23 Lookup – Lookup of Classes, Variables, etc.

This module is responsible for the lookup mechanism in Modelica. It is responsible for looking up classes, types, variables, etc. in the environment of type Env by following the lookup rules.

The important functions are the following:

- `lookupClassClass` – to find a class.
- `lookupTypeType` – to find types (e.g. functions, types, etc.).
- `lookupVarVar` – to find a variable in the instance hierarchy.

Concerning builtin types and operators:

- Built-in types are added in `initialEnvEnv` => same lookup for all types.
- Built-in operators, like `size`(...), are added as functions to `initialEnvEnv`.

Note the difference between Type and Class: the type of a class is defined by ClassInfo state + variables defined in the Types module.

Module dependencies: Absyn, ClassInf, Types, Exp, Env, SCode.

3.4.24 Main – The Main Program

This is the main program in the OpenModelica system. It either translates a file given as a command line argument (see Chapter 2) or starts a server loop communicating through CORBA or sockets. (The Win32 implementation only implements CORBA). It performs the following functions:

- Calls the parser
- Invokes the Interactive module for command interpretation which in turn calls to Ceval for expression evaluation when needed.
- Outputs flattened DAEs if desired.
- Calls code generation modules for C code generation.

Module dependencies: Absyn, Modutil, Parse, Dump, Dumpgraphviz, SCode, DAE, DAElow, Inst, Interactive, Rtopts, Debug, Codegen, Socket, Print, Corba, System, Util, SimCodegen.

Optional dependencies for parallel code generation: ??

3.4.25 MetaUtil – MetaModelica Handling

This module is part of the MetaModelica language extension. This module contains several functions that handles different MetaModelica extensions such as the list construct and the union type construct. These functions have been moved to this module in order to more clearly separate the MetaModelica extension code from the rest of the code in the compiler.

See the OMC MetaModelica extension chapter (chapter 4) for more information.

Module dependencies: Types, Exp, Util, Lookup, Debug, Env, Absyn, SCode, DAE

3.4.26 Mod – Modification Handling

Modifications are simply the same kind of modifications used in the Absyn module.

This type is very similar to `SCode.Mod`. The main difference is that it uses `Exp.Exp` in the Exp module for the expressions. Expressions stored here are prefixed and type checked.

The datatype itself (`Types.Mod`) has been moved to the Types module to prevent circular dependencies.

A few important functions:

- `elabModMod(Env.Env, Prefix.Prefix, Scode.Mod) => Mod` Elaborate modifications.
- `merge(Mod, Mod) => Mod` Merge of Modifications according to merging rules in Modelica.

Module dependencies: Absyn, Env, Exp, Prefix, SCode, Types, Dump, Debug, Print, Inst, Static, Values, Util.

3.4.27 ModSim – Communication for Simulation, Plotting, etc.

This module communicates with the backend (through files) for simulation, plotting etc. Called from the Ceval module.

Module dependencies: System, Util.

3.4.28 ModUtil – Modelica Related Utility Functions

This module contains various utility functions. For example converting a path to a string and comparing two paths. It is used pretty much everywhere. The difference between this module and the Util module is that ModUtil contains Modelica related utilities. The Util module only contains “low-level” “generic” utilities, for example finding elements in lists.

Module dependencies: Absyn, DAE, Exp, Rtopts, Util, Print.

3.4.29 Parse – Parse Modelica or Commands into Abstract Syntax

Interface to external code for parsing Modelica text or interactive commands. The parser module is used for both parsing of files and statements in interactive mode. Some functions never fails, even if parsing fails. Instead, they return an error message other than "Ok".

Input: String to parse

Output: Absyn.Program or InteractiveStmts

Module dependencies: Absyn, Interactive.

3.4.30 Patternm – MetaModelica Pattern Matching

This module is part of the MetaModelica extension. This module contains a big part of the pattern match algorithm. This module contains the functions that transforms a matchcontinue/match expression (an Absyn expression) into a deterministic finite automata (DFA). The DFA is transformed into a value block expression by functions in the DFA module. The "main" function of this module is `matchMain`, which calls a number of functions.

See the OMC MetaModelica extension chapter (chapter 4) for more information.

Input: Absyn.Exp

Output: Absyn.Exp

Module dependencies: Absyn, DFA, Util, Env, SCode, Lookup

3.4.31 Prefix – Handling Prefixes in Variable Names

When performing code instantiation of an expression, there is an instance hierarchy prefix (not package prefix) that for names inside nested instances has to be added to each variable name to be able to use it in the flattened equation set.

An instance hierarchy prefix for a variable x could be for example `a.b.c` so that the fully qualified name is `a.b.c.x`, if `x` is declared inside the instance `c`, which is inside the instance `b`, which is inside the instance `a`.

Module dependencies: Absyn, Exp, Env, Lookup, Util, Print..

3.4.32 Print – Buffered Printing to Files and Error Message Printing

This module contains a buffered print function to be used instead of the builtin print function, when the output should be redirected to some other place. It also contains print functions for error messages, to be used in interactive mode.

No module dependencies.

3.4.33 RTOpts – Run-time Command Line Options

This module takes care of command line options. It is possible to ask it what flags are set, what arguments were given etc. This module is used pretty much everywhere where debug calls are made.

No module dependencies.

3.4.34 SCode – Lower Level Intermediate Representation

This module contains data structures to describe a Modelica model in a more convenient way than the Absyn module does. The most important function in this module is `elaborate` which turns an abstract syntax tree into an `SCode` representation. The `SCode` representation is used as input to the Inst module.

- Defines a lower-level elaborated AST.
- Changed types:
  - Modifications
  - Expressions (uses Exp module)
  - ClassDef (PARTS divided into equations, elements and algorithms)
  - Algorithms uses Algorithm module
  - Element Attributes enhanced.
- Three important public Functions
  - `elaborate (Absyn.Program) => Program`
  - `elabClassClass: Absyn.Class => Class`
  - `buildModMod (Absyn.Modification option, bool) => Mod`

Module dependencies: Absyn, Dump, Debug, Print.

3.4.35 SimCodegen – Generate Simulation Code for Solver

This module generates simulation code to be compiled and executed to a (numeric) solver. It outputs the generated simulation code to a file with a given filename.

Input: DAELow.

Output: To file

Module dependencies: Absyn, DAElow, Exp, Util, RTOpts, Debug, System, Values.

3.4.36 Socket – (Depreciated) OpenModelica Socket Communication Module

This module is partly depreciated and replaced by the Corba implementation. It is the socket connection module of the OpenModelica compiler, still somewhat useful for debugging, and available for Linux and CygWin. Socket is used in interactive mode if the compiler is started with +d=interactive. External implementation in C is in ./runtime/soecketimpl.c.

This socket communication is not implemented in the Win32 version of OpenModelica. Instead, for Win32 build using +d=interactiveCorba.

No module dependencies.

3.4.37 Static – Static Semantic Analysis of Expressions

This module performs static semantic analysis of expressions. The analyzed expressions are built using the constructors in the Exp module from expressions defined in Absyn. Also, a set of properties of the expressions is calculated during analysis. Properties of expressions include type information and a boolean indicating if the expression is constant or not. If the expression is constant, the Ceval module is used to evaluate the expression value. A value of an expression is described using the Values module.

The main function in this module is `eval_exp` which takes an `Absyn.Exp` abstract syntax tree and transforms it into an `Exp.Exp` tree, while performing type checking and automatic type conversions, etc.

To determine types of builtin functions and operators, the module also contain an elaboration handler for functions and operators. This function is called `elabBuiltinHandler`. Note: These functions should only determine the type and properties of the builtin functions and operators and not evaluate them. Constant evaluation is performed by the `Ceval` module.

The module also contain a function for deoverloading of operators, in the `deoverload` function. It transforms operators like '+' to its specific form, `ADD`, `ADD_ARR`, etc.

Interactive function calls are also given their types by `elabExpExp`, which calls `elabCallInteractiveCallInteractive`.

Elaboration for functions involve checking the types of the arguments by filling slots of the argument list with first positional and then named arguments to find a matching function. The details of this mechanism can be found in the Modelica specification. The elaboration also contain function deoverloading which will be added to Modelica in the future when lookup of overloaded user-defined functions is supported.

We summarize a few of the functions:

Expression analysis:

- `elabExpExp: Absyn.Exp => (Exp.Exp, Types.Properties)` – Static analysis, finding out properties.
- `elabGraphicsExp` – for graphics annotations.
- `elabCrefCref` – check component type, constant binding.
- `elabSubscripts`: `Absyn.Subscript` => `Exp.Subscript` – Determine whether subscripts are constant

Constant propagation

- `ceval`

The `elabExpExp` function handles the following:

- constants: integer, real, string, bool

- binary and unary operations, relations
- conditional: ifexp
- function calls
- arrays: array, range, matrix

The `ceval` function:

- Compute value of a constant expressions
- Results as `Values.Value` type

The `canonCrefCref` function:

- Convert `Exp.ComponentRef` to canonical form
- Convert subscripts to constant values

The `elabBuiltinHandlerBuiltinHandler` function:

- Handle builtin function calls such as `size`, `zeros`, `ones`, `fill`, etc.

Module dependencies: Absyn, Exp, SCode, Types, Env, Values, Interactive, ClassInf, Dump, Print, System, Lookup, Debug, Inst, Codegen, Modutil, DAE, Util, RTOpts, Parse, ClassLoader, Mod, Prefix, CEval

3.4.38 System – System Calls and Utility Functions

This module contain a set of system calls and utility functions, e.g. for compiling and executing stuff, reading and writing files, operations on strings and vectors, etc., which are implemented in C. Implementation in runtimesystemimpl.c In comparison, the Util module has utilities implemented in MetaModelica.

Module dependencies: Values.

3.4.39 TaskGraph – Building Task Graphs from Expressions and Systems of Equations

This module is used in the optional `modpar` part of OpenModelica for bulding task graphs for automatic parallelization of the result of the BLT decomposition.

The exported function `build_taskgraph` takes the lowered form of the DAE defined in the DAELow module and two assignments vectors (which variable is solved in which equation) and the list of blocks given by the BLT decomposition.

The module uses the TaskGraphExt module for the task graph datastructure itself, which is implemented using the Boost Graph Library in C++.

Module dependencies: Exp, DAELow, TaskGraphExt, Util, Absyn, DAE, CEval, Values, Print.

3.4.40 TaskGraphExt – The External Representation of Task Graphs

This module is the interface to the externally implemented task graph using the Boost Graph Library in C++.

Module dependencies: Exp, DAELow.

3.4.41 Types – Representation of Types and Type System Info

This module specifies the Modelica Language type system according to the Modelica Language specification. It contains an MetaModelica type called `Type` which defines types. It also contains functions for determining subtyping etc.

There are a few known problems with this module. It currently depends on `SCode.Attributes`, which in turn depends on `Absyn.ArrayDim`. However, the only things used from those modules are constants that could be moved to their own modules.

Identifiers:

```
type Ident = string
```

Variables:

```
uniontype Var "- Variables"
  record VAR
    Ident name "name" ;
    Attributes attributes "attributes" ;
    Boolean protected_ "protected" ;
    Type type_ "type" ;
    Binding binding " equation modification" ;
  end VAR;
end Var;

uniontype Attributes "- Attributes"
  record ATTR
    Boolean flow_ "flow" ;
    SCode.Accessibility accessibility "accessibility" ;
    SCode.Variability parameter_ "parameter" ;
    Absyn.Direction direction "direction" ;
  end ATTR;
end Attributes;

uniontype Binding "- Binding"
  record UNBOUND end UNBOUND;

  record EQBOUND
    Exp.Exp exp "exp" ;
    Option<Values.Value> evaluatedExp "evaluatedExp; evaluated exp" ;
    Const constant_ "constant" ;
  end EQBOUND;

  record VALBOUND
    Values.Value valBound "valBound" ;
  end VALBOUND;
end Binding;
```

Types:

```
type Type = tuple<TType, Option<Absyn.Path>> "A Type is a tuple of a TType
                                              (containing the actual type)
                                              and a optional classname
                                              for the class where the
                                              type originates from.";

uniontype TType "-TType contains the actual type"
  record T_INTEGER
    list<Var> varLstInt "varLstInt" ;
  end T_INTEGER;

  record T_REAL
    list<Var> varLstReal "varLstReal" ;
  end T_REAL;
```

```
  record T_STRING
    list<Var> varLstString "varLstString" ;
  end T_STRING;

  record T_BOOL
    list<Var> varLstBool "varLstBool" ;
  end T_BOOL;

  record T_ENUM end T_ENUM;

  record T_ENUMERATION
    list<String> names "names" ;
    list<Var> varLst "varLst" ;
  end T_ENUMERATION;

  record T_ARRAY
    ArrayDim arrayDim "arrayDim" ;
    Type arrayType "arrayType" ;
  end T_ARRAY;

  record T_COMPLEX
    ClassInf.State complexClassType " The type of. a class" ;
    list<Var> complexVarLst " The variables of a complex type" ;
    Option<Type> complexTypeOption " A complex type can be a subtype of another
                                     primitive) type (through extends).
                                     In that case the varlist is empty" ;
  end T_COMPLEX;

  record T_FUNCTION
    list<FuncArg> funcArg "funcArg" ;
    Type funcResultType "Only single-result" ;
  end T_FUNCTION;

  record T_TUPLE
    list<Type> tupleType " For functions returning multiple values.
                           Used when type is not yet known" ;
  end T_TUPLE;

  record T_NOTYPE end T_NOTYPE;

  record T_ANYTYPE
    Option<ClassInf.State> anyClassType "Used for generic types. When class state
                                         present the type is assumed to be a
                                         complex type which has that restriction";
  end T_ANYTYPE;

end TType;

uniontype ArrayDim "- Array Dimensions"
  record DIM
    Option<Integer> integerOption;
  end DIM;

end ArrayDim;

type FuncArg = tuple<Ident, Type> "- Function Argument" ;
```

Expression properties:

A tuple has been added to the `Types` representation. This is used by functions returning multiple arguments.

Used by `splitPropsProps`:

```
uniontype Const " Variable properties: The degree of constantness of an expression
is determined by the Const datatype.
    Variables declared as 'constant' will get C_CONST constantness.
    Variables declared as \'parameter\' will get C_PARAM constantness and
    all other variables are not constant and will get C_VAR constantness."
  record C_CONST end C_CONST;

  record C_PARAM "\'constant\'s, should always be evaluated" end C_PARAM;

  record C_VAR "\'parameter\'s, evaluated if structural not constants,
        never evaluated"
  end C_VAR;
end Const;

uniontype TupleConst "A tuple is added to the Types.
                      This is used by functions whom returns multiple arguments.
                      Used by split_props"
  record CONST
    Const const;
  end CONST;

  record TUPLE_CONST
    list<TupleConst> tupleConstLst "tupleConstLst" ;
  end TUPLE_CONST;
end TupleConst;

uniontype Properties "Expression properties:
                      For multiple return arguments from functions,
                      one constant flag for each return argument.
                      The datatype `Properties\' contain information about an
                      expression.  The properties are created by analyzing the
                      expressions."
  record PROP
    Type type_ "type" ;
    Const constFlag "if the type is a tuple, each element have a const flag.";
  end PROP;

  record PROP_TUPLE
    Type type_;
    TupleConst tupleConst " The elements might be tuple themselfs.";
  end PROP_TUPLE;

end Properties;
```

The datatype `Properties` contains information about an expression. The properties are created by analyzing the expressions.

To generate the correct set of equations, the translator has to differentiate between the primitive types `Real`, `Integer`, `String`, `Boolean` and types directly derived from then from other, complex types. For arrays and matrices the type `T_ARRAY` is used, with the first argument being the number of dimensions, and the second being the type of the objects in the array. The `Type` type is used to store information about whether a class is derived from a primitive type, and whether a variable is of one of these types.

Modification datatype:

```
uniontype EqMod "To generate the correct set of equations, the translator has to
                 differentiate between the primitive types `Real\', `Integer\',
                 `String\', `Boolean\' and types directly derived from then from
                 other, complex types.  For arrays and matrices the type
                 `T_ARRAY\' is used, with the first argument being the number of
                 dimensions, and the second being the type of the objects in the
                 array.  The `Type\' type is used to store information about
                 whether a class is derived from a primitive type, and whether a
```

```
                variable is of one of these types.
  record TYPED
    Exp.Exp modifierAsExp "modifierAsExp ; modifier as expression" ;
    Option<Values.Value> modifierAsValue " modifier as Value option" ;
    Properties properties "properties" ;
  end TYPED;

  record UNTYPED
    Absyn.Exp exp;
  end UNTYPED;
end EqMod;

uniontype SubMod "-Sub Modification"
  record NAMEMOD
    Ident ident;
    Mod mod;
  end NAMEMOD;

  record IDXMOD
    list<Integer> integerLst;
    Mod mod;
  end IDXMOD;
end SubMod;

uniontype Mod "Modification"
  record MOD
    Boolean final_ "final" ;
    Absyn.Each each_;
    list<SubMod> subModLst;
    Option<EqMod> eqModOption;
  end MOD;

  record REDECL
    Boolean final_ "final" ;
    list<tuple<SCode.Element, Mod>> tplSCodeElementModLst;
  end REDECL;

  record NOMOD end NOMOD;
end Mod;
```

Module dependencies: Absyn, Exp, ClassInf, Values, SCode, Dump, Debug, Print, Util.

3.4.42 Util – General Utility Functions

This module contains various utility functions, mostly list operations. It is used pretty much everywhere. The difference between this module and the ModUtil module is that ModUtil contains Modelica related utilities. The Util module only contains “low-level” general utilities, for example finding elements in lists.

This modules contains many functions that use type variables. A type variable is exactly what it sounds like, a type bound to a variable. It is used for higher order functions, i.e., in MetaModelica the possibility to pass a "handle" to a function into another function. But it can also be used for generic data types, like in C++ templates.

A type variable in MetaModelica is written as ??? `'a`.

For instance, in the function `list_fill ('a,int) => 'a list` the type variable `'a` is here used as a generic type for the function `list_fill`, which returns a list of n elements of a certain type.

No module dependencies.

3.4.43 Values – Representation of Evaluated Expression Values

The module Values contains data structures for representing evaluated constant Modelica values. These include integer, real, string and boolean values, and also arrays of any dimensionality and type.

Multidimensional arrays are represented as arrays of arrays.

```
uniontype Value
  record INTEGER Integer integer; end INTEGER;
  record REAL Real real; end REAL;
  record STRING String string; end STRING;
  record BOOL Boolean boolean; end BOOL;
  record ENUM String string; end ENUM;
  record ARRAY list<Value> valueLst; end ARRAY;
  record TUPLE list<Value> valueLst; end TUPLE;

  record RECORD
    Absyn.Path record_ "record name" ;
    list<Value> orderd "orderd set of values" ;
    list<Exp.Ident> comp "comp names for each value" ;
  end RECORD;

  record CODE
    Absyn.Code A "A record consist of value  Ident pairs" ;
  end CODE;
end Value;
```

Module dependencies: Absyn, Exp.

3.4.44 VarTransform – Binary Tree Representation of Variable Transformations

VarTransform contains Binary Tree representation of variables and variable replacements, and performs simple variable subsitutions and transformations in an efficient way. Input is a DAE and a variable transform list, output is the transformed DAE.

Module dependencies: Exp, DAELow, System, Util, Algorithm.

Chapter 4

MetaModelica Pattern Matching Compilation

This chapter gives a more detailed description of the methods used for compilation of pattern matching as implemented in the modules Patternm and DFA.

In addition to the pattern matching, several other language constructs have been added to the OpenModelica Compiler (OMC). A majority of these constructs are MetaModelica constructs. This chapter describes the implementation of these constructs in order to ease the continuous implementation.

The most important construct that has been added to the OMC is the matchcontinue expression. It has been implemented using an algorithm for pattern matching developed by Mikael Pettersson (former PELAB member). This algorithm first transforms the matchcontinue expression into a Deterministic Finite Automata (DFA). This DFA is then transformed into if-elseif-else nodes.

Other constructs that have been added (or are currently being added) include the MetaModelica list type, MetaModelica union type and the MetaModelica tuple type.

A value block expression has been added to the OMC. The value block expression is simply an expression consisting of a local variable declaration section, an equation or algorithm section and a return statement. Similar block constructs may be found in languages such as Java and C. This construct is only available internally and not for the end-user. The matchcontinue expression makes use of the value block expression.

A number of modules have been altered. The implementation of the value block expression resulted in the altering of many modules since it created circular dependencies in the compiler and a number of data structures and functions had to be replicated. This replication, however, should only be seen as a temporary solution. A later version of the OMC will hopefully be able to handle circular dependencies better.

4.1 MetaModelica Matchcontinue Expression

The matchcontinue expression is transformed from an `Absyn.Exp` into a new `Absyn.Exp`, namely a value block (see section 4.2). The matchcontinue expression is first encountered in the function `instStatement` in the Inst module. From here the expression is dispatched to the function `matchMain` in Patternm. Patternm contains the code that transforms the `Absyn.Exp` into a DFA.

The DFA data structure can be found in the module DFA. The DFA module also contains functions that convert the DFA into a value block with if-elseif-else nodes. The pattern matching code is clearly separated from the rest of the code since there is only one point of entry, in Inst, and the rest of the algorithm is located in DFA and Patternm.

4.1.1 Modules Involved

4.1.1.1 Absyn

The abstract syntax for the matchcontinue expression was added to Absyn by Adrian Pop.

4.1.1.2 Inst

Two new cases have been added to the function `instStatement`, one for the case `(var1,...,varN) := matchcontinue () ...` (tuple assignment) and one for the case `var := matchcontinue () ...` (single variable assignment). The pattern match algorithm is invoked (this algorithm has its entry point in the function `matchMain` in the module Patternm) and a value block expression is given in return. The reason why we single out the matchcontinue expression in this function and this module (instead of in `Static.elabExp`) is that we need to know the return type(s) of the value block that we create (and the names of the assigned variables). The return type(s) is given by the types of the variables on the left side of the assignments. As of now, the left-hand side variables are used as the return variables of the value block/matchcontinue expression so that no new variables have to be created.

4.1.1.3 Patternm

This module contains most of the pattern match algorithm. This module contains the functions that takes a matchcontinue expression and transforms it into a DFA. The DFA is transformed into a value block expression by functions in DFA.

The "main" function of this module is `matchMain`, this function calls several functions. First it calls `ASTtoMatrixForm` which transforms the matchcontinue expression into a matrix and a vector/list. The matrix contains renamed patterns (patterns containing “path” variables). The vector contains right-hand side records (records containing equations and variables belonging to a right-hand side of the initial matchcontinue expression).

After `ASTtoMatrixForm` the function `matchFuncHelper` is called. This function takes care of all the pattern matching and transforms the renamed pattern matrix and right-hand side list into a DFA. The last thing `matchMain` does is to call `DFA.fromDFAtoIfNodes` which transforms the DFA into a value block expression.

The function `ASTtoMatrixForm` goes through each and every case-clause in the matchcontinue expression, adds path variables to the patterns, singles out the right-hand sides and takes care of all the as-bindings (a pattern such as `e as Absyn.INTEGER(1)` will result in a new variable assignment in the corresponding right-hand side, involving the path variable and the variable e).

The function `extractFromMatchAST` simply creates one list of patterns and one vector of right-hand sides out of the matchcontinue expression. A matrix which contains renamed patterns is then created.

This matrix is then filled with renamed patterns by the function `fillMatrix`. This function takes one tuple at a time from the list of patterns, rename all the patterns (add path variables) and then add a new row to the matrix.

The function `addRow` adds a new row to the matrix after it has invoked the function `renameMain` on each pattern in the row.

The function `renameMain` recursively adds path variables to a pattern. The function `renamePatList` calls `renameMain` on each pattern in a list of patterns.

The function `matchFuncHelper` is the workhorse of the pattern match algorithm. This function dispatches to a number of cases. Which case that should be executed is determined by the upper row of the matrix. If the matrix, and thus the upper row, is empty, a final state is created. This can be seen as the stop condition of the algorithm. A final state is a state that contains the variables and equations from a right-hand side record. There are three other main cases as given below. The `matchFuncHelper` function will assign a unique number, a stamp, to each state.

- **Case 1, all of the top-most patterns consist of wildcards**. The leftmost wildcard is used to create an arc to a new state. The function `matchFuncHelper` is invoked on this new state with what is left of the upper row (actually, since this row only contains wildcards we can discard all these wildcards and go directly to a final state). An else arc to a new state is created; `matchFuncHelper` is invoked on this new state with the rest of the matrix with the upper-row removed.

- **Case 2, the top-most column consists of wildcards and constants but no constructors (record calls or cons expressions)**. Select the left-most column with a constant at the uppermost position. If this is the only column in the matrix do the following: Create a new arc with the constant and a new (final) state. Create an else branch and a new state and invoke `matchFuncHelper` on this new state with what is left of the column. Otherwise if there is more than one column left in the matrix: Create one new arc and state for each constant and one new arc and state for all the wildcards. This is done by calling the functions `addNewArcForEachC` and `addNewArcForWildcards`.
- **Case 3, there is a column whose top-most pattern is a constructor**. Select this column. The function `matchFuncHelper` calls the function `matchCase3`. We create a new arc for each constructor c. For each constructor c: Select the rows that match c (wildcards included). Extract the sub patterns, create a new arc and state and invoke `matchFuncHelper` on what is left on the matrix appended with the extracted sub patterns. This is mainly done in the function `addNewArcForEachCHelper`. If this is the only column in the matrix do the following: Create an else arc and a new "union" state for all the wildcards and constants. This is done by the function `createUnionState`. Otherwise if there is more than one column left in the matrix: create an arc and state for each constant, in the same way as for the constructors. Create one new arc and state for all the wildcards.

An array containing states already created is passed along in the pattern match algorithm. Whenever a new state is about to be created, we search in this array to see whether an equal state already has been created. If this is the case we simply create a goto-state containing the name of the old state. We use the stamps/numbers assigned to each state to jump between equal states and to access the array.

4.1.1.4 DFA

This module contains the DFA data structure. The DFA data structure has the following components.

- A DFA record which contains the start state, the number of states the DFA contains, an optional else case, and a list of variables that will be added to the local variable section in the resulting value block.
- A state record which contains a state stamp (identifier), a list of outdoing arcs, and an optional right-hand side (if the state is a final state). There is also a goto-state record; it simply contains the name of the state to jump to.
- An arc record which contains the state the arc is leading to, a list of numbers representing all the right-hand sides that this arc leads to down the path, the name of the arc, and an optional renamed pattern (the arc may be an else arc which means it does not have a renamed pattern).

This module also contains the functions that transform a DFA into a value block expression with nested if-elseif-else nodes. The entry point is the function `fromDFAtoIfNodes`. This function will start by creating some variables that are mostly needed for the failure handling (a case-clause in a matchcontinue expression may fail which leads to the matching of the next case).

After this the function `generateAlgorithmBlock` is invoked. The function `fromStatetoAbsynCode` will be called with the start state of the DFA. Depending on whether an else-case exists or not we might need to generate some extra code in `generateAlgorithmBlock`.

The function `fromStatetoAbsynCode` will take a state as input, extract the outgoing arcs from this state, create an if-elseif-else statement for all the arcs and recursively invoke itself on each state that each arc leads to.

The recursive call is made by the function `generateIfElseifAndElse` which is the function that creates the if-elseif-else statements. The function `generateIfElseifAndElse` is a function that takes a

list of arcs as input and accumulates if-elseif cases in a list until the list of arcs is empty and the actual if-elseif-else statement is created.

The function `fromStatetoAbsynCode` must keep track of the type of the incoming arc to the current state. If the incoming arc was a constructor then new path variables must be declared and initialized to the field values of the record. This is done by the function `generatePathVarDeclerations`. This function looks up the type and name of each field in the record so that a new variable may be declared.

The module DFA also contains the renamed patterns union type. A renamed pattern is similar to an `Absyn.Exp` except that we have added a path variable to each pattern. This module also contains functions for handling matrices: adding a row to a matrix, picking out the first row of a matrix, removing the first row of a matrix, singling out a column from a matrix, etc..

In order to handle matchcontinue failures (a case-clause may fail which should lead to the matching of the next case-clause) the following scheme is used.

- As mentioned earlier, the numbers of the right-hand sides that each arc eventually leads to are saved in a list in the arc record.
- An array of Boolean values is added to the final value block. The array contains one entry for each right-hand side.
- Whenever a right-hand side section fails, we catch this failure and set the corresponding entry in the Boolean array to false.
- In every if-else-elseif statement, in the generated code, we access the Boolean array to see whether all the right-hand sides that this arc leads to already have been visited.

An example follows.

```
y := matchcontinue(x)
    case (1) equation … <code1> fail(); <code2> … then 1;
    case (2) equation … <code3> … then 2;
end matchcontinue;
```

The code above would result in the following C-code (note that the code is somewhat simplified).

```
{
Bool BOOLVAR[2] = {true,true};
Int LASTFINALSTATE = 0;
Bool NOTDONE = true;

while(1)
{
    try {
          if (x == 1 && BOOLVAR[1]) {
                LASTFINALSTATE = 1;
                <code1>
                throw 1; //fail
                <code2>
                …
                NOTDONE = false;
          }
          else if (x == 2 && BOOLVAR[2]) {
                LASTFINALSTATE = 2;
                <code3>
                NOTDONE = false;
          }
```

```
      }
      catch (…) {
            BOOLVAR[LASTFINALSTATE] = false;
      }
      if (!NOTDONE) break;

  }
  }
```

4.2 Value block Expression

The value block expression makes it possible to have equations and algorithm statements nested within another equation or algorithm statement. This fact makes the implementation of this construct rather complicated. Circular dependencies arise in the compiler. The compiler design also becomes unclean in the sense that the original patterns of design are altered: we may find pieces of code in places we did not expect.

4.2.1 Modules Involved

4.2.1.1 Absyn

A value block record has been added to `Absyn.Exp`. This record consists of a list of `elementItems` (local variable declarations), a `ValueblockBody` union type (this union type consists of two records, one representing a list of equations and the other one representing a list of algorithm statements) and a result expression.

4.2.1.2 Exp

A value block record has been added to this module. Since a value block may contain variable declarations and algorithm statements (if any equations exist at the outset these are converted into algorithm assignment statements by a function in the Static module) and since we do not want circular dependencies we had to duplicate many data structure into Exp. We had to move (duplicate) type data structures from Types, DAE and Algorithm. In Static when the value block is first encountered these data structures are converted from being union types of Types, DAE and Algorithm into being union types of Exp. In Codegen they are then converted back. This converting is done by the module Convert, see the next paragraph.

4.2.1.3 Convert

This module contains functions that convert union types from Types, DAE and Algorithm into corresponding union types in Exp, and then back again.

4.2.1.4 Static

The value block expression is first encountered in this module in the function `elabExp`. First a new scope is added to the environment. After this the local variable list is elaborated and the variables are added to the environment. After this the algorithm section is instantiated and the return expression is elaborated. However, in order to avoid circular dependencies we had to add some extra data structures to Exp as mentioned above. Therefore we must call functions in the module Convert that converts these data structures. If we have a value block with an equation section instead of an algorithm section we simply use the function `fromEquationsToAlgAssignments` to transform each equation into an algorithm assignment statement.

4.2.1.5 Prefix

In the function `prefixExp` we must now handle a value block expression. New functions that can add prefixes to elements and algorithm section have been added: `prefixDecls`, `prefixAlgorithm` and `prefixStatements`.

4.2.1.6 Codegen

The value block expression (an `Exp.Exp` record) is encountered in the function `generateExpression`. First the list of elements and algorithm statements are converted from Exp union types into DAE, Types and Algorithm union types. After this the C code is generated in a rather straightforward fashion.

4.3 MetaModelica list

The MetaModelica language contains a list construct, similar to the one found in languages like Lisp.

```
list<Integer> listInt;
...
listInt = {1,2,3,4};
listInt = cons(1,{1,2,3});
listInt = (1 :: {1,2,3}); // :: is the cons operator
```

This list type has now been added to the OMC. The C code that is generated consists of void pointers and function calls to the C runtime functions `mk_nil` and `mk_cons`.

4.3.1 Modules Involved

4.3.1.1 Absyn

The :: operator is represented by the `CONS` record in the Exp union type in Absyn. A `LIST` record has also been added to the Exp union type. This one is used internally in the compiler to represent an `Absyn.ARRAY` (the parser cannot decide whether curly brackets, { ... }, denotes a list or an array constructor). In some places in the code (where type information is available), an `Absyn.ARRAY` expression is replaced by an `Absyn.LIST` expression.

4.3.1.2 Codegen

C code is generated for the `Exp.LIST` and `Exp.CONS` expressions in the function `generateExpression`. `DAE.Type` and `Types.T_LIST` are handled in several places in this module and C void pointers are generated.

4.3.1.3 DAE

A list type has been added to the union type `DAE.Type`.

4.3.1.4 DFA

The handling of lists has been added to this module. A renamed cons pattern should result in an appropriate if-statement. Given a list variable, we must create two new variables that should be assigned the car and cdr parts of the list variable. An example follows.

```
matchcontinue (x)
      case (1 :: {}) ...
```

The above example should result in the following (somewhat simplified) code.

```
      if () {
            Type x1 = car(x);
            list<Type> x2 = cdr(x);
            if (x1 == 1) {
                  if (x2 == {}) {...}
```

```
            }
        }
```

An extra environment variable must be passed along. This environment contains the types of the variables generated from a cons pattern (such as x1 and x2 above). This is needed because when we encounter a path variable such as x1 and x2 (that have been generated from a cons pattern) we need to know the type of this variable.

4.3.1.5 Inst

Extra clauses have been added to the functions `instElement` and `instStatement`. In the function `instElement`, a list element must be dealt with separately. The basic underlying type of the list is handled as usual and at the end the `Types.T_LIST` is added to the resulting DAE element. Nested lists, for instance `list<list<Integer>>`, are also supported.

4.3.1.6 Metautil

This module contains a number of functions that deals with the list construct. These functions are invoked from Inst, Static and Codegen. This module was added so that the code dealing with MetaModelica constructs would be more strictly separated from the rest of the code.

4.3.1.7 Patternm

The cons and empty-list patterns are handled in `renameMain` and in a few other functions.

4.3.1.8 Static

Several extra clauses have been added to the function `elabExp`. When the MetaModelica flag is set, we must go through all the arguments to a function call to see if there are any `Absyn.ARRAY` expressions. If this is the case and the underlying type is a list, we must replace this `Absyn.ARRAY` expression with an `Absyn.LIST` expression. In the function `elabExp` we also handle the `Absyn.LIST` and `Absyn.CONS` records. The elaboration of these records results in an `Exp.LIST` or `Exp.CONS` record.

4.3.1.9 Types

A `T_LIST` record has been added to the `TType` union type. This record is handled by for instance the function `subtype`.

4.3.1.10 Values

A list value has been added to this module. However, it is not used as of now (and may never have to be used in the future).

4.4 MetaModelica Union Type

NA.

Chapter 5

OMNotebook and OMShell

This chapter covers the OpenModelica electronic notebook subsystem, called OMNotebook. Both OMNotebook and OMShell uses the development framework Qt.

5.1 Qt

Qt is an object-oriented, platform independent, C++ development framework created and maintained by Trolltech. Qt includes a comprehensive class library, with more then 400 classes, and several tools for development. The Qt API has a rich set of classes and functionality for several types of development and programming. In OMNotebook Qt have been used for GUI programming, file handling and XML, but Qt can be used for database programming, networking, internationalization, OpenGL integration and much more.

Qt is consistent across all supported platforms, which enable developers to create truly platform independent applications. Using Qt, developers can create native applications for Windows, Mac and X11 platforms. Qt requires no virtual machines, emulation layers or bulky runtime environments. Instead Qt writes directly to low-level graphics function like native applications, which allows Qt applications to run natively. Trolltech have designed Qt to be easy and intuitive to use.

5.2 HTML documentation

Using Doxygen a HTML documentation have been generated from the source files. This documentation contatins information about the different classes, functions and files belonging to OMNotebook. The documentation is found on the SVN under OMNotebook/Doxygen_doc.

5.3 Mathematica Notebook Parser

OMNotebook have a parser implemented that can read Mathematica notebooks. This parser is generated by ANTLR using grammar descriptions. This is an EBNF grammar for the Mathematica notebook fullform format, taken from the grammar definition for the Mathematica notebook parser.

```
document          ::=  <expr>

expr              ::=  (FrontEnd´)* <exprheader>
                     |  <value>
                     |  <attribute>
```

```
exprheader        ::=
    Notebook [ <expr> (, <rule>)* ]
   |List [ (<listbody>)* (, <listbody>)* ]
   |list [ (<listbody>)* (, <listbody>)* ]
   |Cell [ <expr> (, <expr>)? (, <rule>)* ]
   |CellGroupData [ <expr> (, Open|Closed)) ]
   |TextData [ <expr> (, <expr>)* (, <rule>)* ]
   |StyleBox [ <expr> (, <expr>)* (, <rule>)* ]
   |StyleData [ <expr> (, <expr>)* (, <rule>)* ]
   |SuperscriptBox [ <expr>, <expr> ]
   |SubscriptBox [ <expr>, <expr> ]
   |SubsuperscriptBox [ <expr> (, <expr>)* (, <rule>)* ]
   |UnderscriptBox [ <expr> (, <expr>)* (, <rule>)* ]
   |OverscriptBox [ <expr> (, <expr>)* (, <rule>)* ]
   |UnderoverscriptBox [ <expr> (, <expr>)* (, <rule>)* ]
   |FractionBox [ <expr> (, <expr>)* (, <rule>)* ]
   |SqrtBox [ <expr> (, <expr>)* (, <rule>)* ]
   |RadicalBox [ <expr> (, <expr>)* (, <rule>)* ]
   |RowBox [ <expr> (, <expr>)* (, <rule>)* ]
   | GridBox [ <expr> (, <expr>)* (, <rule>)* ]
   |FormBox [ <expr> (, <expr>)* (, <rule>)* ]
   |TagBox [ <expr> (, <expr>)* (, <rule>)* ]
   |CounterBox [ <expr> (, <expr>)* (, <rule>)* ]
   |AdjustmentBox [ <expr> (, <expr>)* (, <rule>)* ]
   |ButtonBox [ <expr> (, <expr>)* (, <rule>)* ]
   |InterpretationBox [ <expr>, <expr> ]
   |Annotation [ <expr> (, <expr>)* (, <rule>)* ]
   |Equal [ <expr> (, <expr>)* (, <rule>)* ]
   |Diagram [ <expr> (, <expr>)* (, <rule>)* ]
   |Icon [ <expr> (, <expr>)* (, <rule>)* ]
   |Polygon [ <expr> (, <expr>)* (, <rule>)* ]
   |Ellipse [ <expr> (, <expr>)* (, <rule>)* ]
   |Line [ <expr> (, <expr>)* (, <rule>)* ]
   |GreyLevel [ <expr> (, <expr>)* (, <rule>)* ]
   | OLEData [ <expr> (, <expr>)* (, <rule>)* ]
   |RGBColor [ Number, Number, Number ]
   |Filename [ <expr> (, <expr>)* (, <rule>)* ]
   |BoxData [ <expr> (, <expr>)* (, <rule>)* ]
   |GraphicsData [ String, String (, <rule>)* ]
   |DirectedInfinity [ Number ]
   |StartModelEditor [ ]
   |ParentDirectory [ ]

listbody          ::=  (<expr>|<rule>)

rule              ::=  Rule [ <expr> (, <expr>) ]
                     |   rule [<expr> (, <expr>) ]
                     |   RuleDelayed [ <expr> (, <expr>) ]

value             ::=  String
                     |   Number
                     |   True
```

```
                    |   False
                    |   Right
                    |   Left
                    |   Center
                    |   Smaller
                    |   Inherited
                    |   PaperWidth
                    |   WindowWidth
                    |   TraditionalForm
                    |   StandardForm
                    |   InputForm
                    |   OutputForm
                    |   DefaultInputFormatType
                    |   Automatic
                    |   None
                    |   Null
                    |   All

attribute         ::=   FontSlant
                    |   FontSize
                    |   FontColor
                    |   FontWeight
                    |   FontFamily
                    |   FontVariation
                    |   TextAlignment
                    |   TextJustification
                    |   InitializationCell
                    |   FormatType
                    |   PageWidth
                    |   PageHeaders
                    |   PageHeaderLines
                    |   PageFooters
                    |   PageFooterLines
                    |   PageBreakBelow
                    |   PageBreakWithin
                    |   BoxMargins
                    |   BoxBaselineShift
                    |   LineSpacing
                    |   Hyphenation
                    |   Active
                    |   Visible
                    |   Evaluatable
                    |   ButtonFuncion
                    |   ButtonData
                    |   ButtonEvaluator
                    |   ButtonStyle
                    |   CharacterEncoding
                    |   ShowStringCharacters
                    |   ScreenRectangle
                    |   AutoGeneratedPackage
                    |   AutoItalicWords
                    |   InputAutoReplacements
                    |   ScriptMinSize
```

```
|   StyleMenuListing
|   CounterIncrements
|   CounterAssignments
|   PrivateEvaluationOptions
|   GroupPageBreakWithin
|   DefaultFormatType
|   NumberMarks
|   LinebreakAdjustments
|   VisioLineFormat
|   VisioFillFormat
|   Extent
|   NamePosition
|   CellTags
|   CellFrame
|   CellFrameColor
|   CellFrameLabels
|   CellFrameMargins
|   CellFrameLabelMargins
|   CellLabelMargins
|   CellLabelPositioning
|   CellMargins
|   CellDingbat
|   CellHorizontalScrolling
|   CellOpen
|   GeneratedCell
|   ShowCellBracket
|   ShowCellLabel
|   CellBracketOptions
|   Editable
|   Background
|   CellGroupingRules
|   WindowSize
|   WindowMargins
|   WindowFrame
|   WindowElements
|   WindowTitle
|   WindowToolbars
|   WindowMoveable
|   WindowFloating
|   WindowClickSelect
|   StyleDefinitions
|   FrontEndVersion
|   ScreenStyleEnvironment
|   PrintingStyleEnvironment
|   PrintingOptions
|   PrintingCopies
|   PrintingPageRange
|   PrivateFontOptions
|   Magnification
|   GenerateCell
|   CellAutoOverwrite
|   ImageSize
|   ImageMargins
```

```
|   ImageRegion
|   ImageRangeCache
|   ImageCache
|   ModelEditor
```

5.4 File list

This file list lists all source files belonging to OMNotebook in alphabetical order with a short description. In addition to these files a set of files are also generated by Qt and ANTLR, but those files are not listed below. The lines of code (LOC) specified for each file is with comments and blank rows (counted May 2006).

File	Description	LOC
application.h	Describe interface for the core application.	88
cell.cpp	Implementation of the Cell class.	923
cell.h	Definition of the Cell class, superclass for all cells.	234
cellapplication.cpp	Implementation of the CellApplication class.	706
cellapplication.h	Definition of the CellApplication class, the main application class.	106
cellcommandcenter.cpp	Implementation of the CellCommandCenter class.	134
cellcommandcenter.h	Definition of the CellCommandCenter class, responsible for storing and executing commands.	77
cellcommands.cpp	Implementation of all commands on cell level.	766
cellcommands.h	Definition of all commands on cell level.	201
cellcursor.cpp	Implementation of the CellCursor class.	580
cellcursor.h	Definition of the CellCursor class, a subclass of Cell used as a cursor within a document.	131
celldocument.cpp	Implementation of the CellDocument class.	1359
celldocument.h	Definition of the CellDocument class, represent a document, contains all cells.	218
celldocumentview.h	Describe interface for a notebook window. [deprecated]	93
cellfactory,cpp	Implementation of the CellFactory class.	208
cellfactory.h	Definition of the CellFactory class, responsible for creating all cells.	85
cellgrammar.cpp	Small text application, to test grammar description. [deprecated]	109
cellgroup.cpp	Implementation of the CellGroup class.	500
cellgroup.h	Definition of the CellGroup, a subclass of Cell used to group together cells.	129
cellparserfactory.cpp	Implementation of the CellParserFactory class.	96
cellstyle.h	Definition and Implementation of the CellStyle class, holds different style options for cells.	131
chaptercountervisitor.cpp	Implementation of the ChapterCounterVisitor class.	187
chaptercountercisitor.h	Definition of the ChapterCounterVisitor class, responsible for updating chapter counters.	92
command.h	Describe interface for a commands.	134
commandcenter.h	Describe interface for a command center.	74
commandcompletion.cpp	Implementation of the CommandCompletion class.	408
commandcompletion.h	Definition of the CommandCompletion class, responsible for command completion.	103
commands.xml	XML file containing all commands and keywords for CommandCompletion class.	114
commandunit.h	Definition and Implementation of the CellStyle class,	116

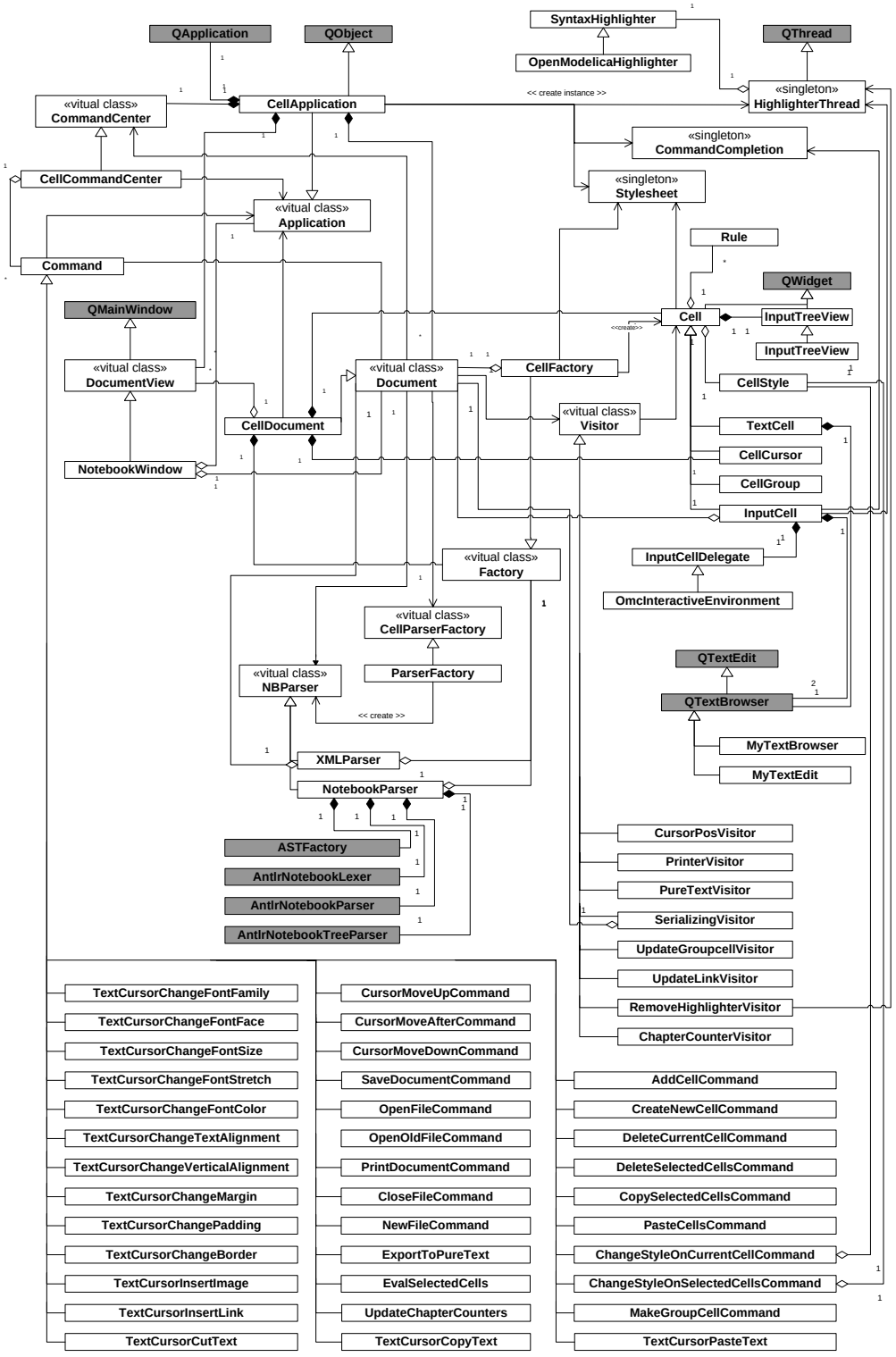
holds a command/keyword for command completion.

copytest.cpp	Small text application, to test copy function for cells. [deprecated]	78
cursorcommands.h	Definition and implementation of all commands on cursor level.	227
cursorposvisitor.h	Definition and implementation of the CursorPosVisitor class, responsible for calculate cell cursor position.	135
document.h	Describe interface for a document.	180
documentview.h	Describe interface for a notebook window.	87
factory.h	Describe interface for a cell factory.	84
highlighterthread.cpp	Implementation of the HighlighterThread class.	283
highlighterthread.h	Definition of the HighlighterThread class, responsible for running the syntax highlighter.	95
imagesizedlg.h	Definition and implementation of the ImageSizeDlg class, a dialog for selecting size of an image.	126
ImageSizeDlg.iu	Define user interface for ImageSizeDlg class.	114
inputcell.cpp	Implementation of the InputCell class.	1592
inputcell.h	Definition of the InputCell class, a subclass of Cell used to enter code in.	210
inputcelldelegate.h	Describe the interface for an input cell delegate.	81
lexer.g	Grammar file for ANTLR, describe tokens.	330
modelicacolors.xml	Specifies color and font settings for the highlighter.	47
nbparser.h	Describe interface for a parser.	66
notebook.cpp	Implementation of the NotebookWindow class.	3348
notebook.h	Definition of the NotebookWindow class, main window used to display a document.	350
notebookcommands.h	Definition and implementation of all commands on document/notebook level.	500
notebookparser.cpp	Implementation of the NotebookParser class.	171
notebookparser.h	Definition of the NotebookParser class, responsible for loading Mathematica notebooks saved in fullform.	76
notebooksocket.cpp	Implementation of the NotebookSocket class.	299
notebooksocket.h	Definition of the NotebookSocket class, for communi-cation between different OMNotebook processes.	63
omc_communicator.cpp	Implementation of the OmcCommunicator class.	1420
omc_communicator.hpp	Definition of the OmcCommunicator class, responsible for low level communication with OMC.	201
omcinteractiveenvironment.cpp	Implementation of the OmcInteractiveEnvironment class.	297
omcinteractiveenvironment.h	Definition of the OmcInteractiveEnvironment class, a interactive environment for evaluation with OMC.	79
OMNotebookHelp.onb	Help documentation about OMNotebook.	---
openmodelicahighlighter.cpp	Implementation of the OpenModelicaHighlighter class.	543
openmodelicahighlighter.h	Definition of the OpenModelicaHighlighter class, a syntax highlighter for modelica code.	124
otherdlg.h	Definition and implementation of the OtherDlg class, a dialog for selecting an integer value.	116
OtherDlg.ui	Define user interface for OtherDlg class.	114
parser.g	Grammar file for ANTLR, describe grammar rules.	226
parserfactory.h	Describe interface for a parser factory. Definition of the CellParserFactory, responsible for creating correct parser for a given file.	83
printervisitor.cpp	Implementation of the PrinterVisitor class.	302
printervisitor.h	Definition of the PrinterVisitor class, creates the document that is sent to a printer.	101

puretextvisitor.cpp	Implementation of the PureTextVisitor class.	179
puretextvisitor.h	Definition of the PureTextVisitor class, extracts document contents and save it as pure text.	95
qtapp.cpp	Contains the main() function.	87
removehighlightervisitor.h	Definition and implementation of the RemoveHighlighterVisitor class, remove documents cells from the highlighter thread.	97
rule.h	Implementation and definition of the Rule class, holds format rules for cells and styles.	101
serializingvisitor.cpp	Implementation of the SerializingVisitor class.	331
serializingvisitor.h	Definition of the SerializingVisitor class, responsible for saving a document in .onb format.	111
stripstring.h	Static functions for text manipulation, used in walker.g.	353
stylesheet.cpp	Implementation of the Stylesheet class.	521
stylesheet.h	Definition of the Stylesheet class, holds and manages the different cell styles.	108
stylesheet.xml	XML file containing specification of ass cell styles.	146
syntaxhighlighter.h	Define interface for a syntax highlighter.	85
textcell.cpp	Implementation of the TextCell class.	871
textcell.h	Definition of the TextCell class, a subclass of Cell used to write normal text in.	167
textcursorcommands.cpp	Implementation of all commands on text cursor level.	604
textcursorcommands.h	Definition of all commands on text cursor level.	271
treeview.cpp	Implementation of the TreeView class.	220
treeview.h	Definition of the TreeView class, represents an item in the tree view of documents.	115
updategroupcellvisitor.cpp	Implementation of the UpdateGroupcellVisitor class.	123
updategroupcellvisitor.h	Definition of the UpdateGroupcellVisitor class, responsible for updating groupcell state when loading.	86
updatelinkvisitor.cpp	Implementation of the UpdateLinkVisitor class.	176
updatelinkvisitor.h	Definition of the UpdateLinkVisitor class, responsible for updating links when needed.	95
visitor.h	Describe interface for a visitor.	96
walker.g	Grammar file for ANTLR, describe how to walk to created tree and create a cell structure.	953
xmlnodename.h	Define all xml name used in the .onb file format.	85
xmlparser.cpp	Implementation of the XMLParser class.	600
xmlparser.h	Definition of the XMLParser class, responsible for loading files saved in .onb format.	111
	Sum:	*27 037*

5.5 Class overview

The following diagram contains the complete static structure of OMNotebook.



5.6 References

Anders Fernström. Extending OMNotebook – An Interactive Notebook for Structured Modelica Documents.Final thesis to be presented spring 2006, Dept. Computer and Information Science, Linköping University, Sweden.

Trolltech, Qt Product Overview, http://www.trolltech.com/products/qt/index.html.

van Heesch, Dimitri, www.doxygen.org (2006), Doxygen, http://www.doxygen.org.

ANTLR, About The Parser Generator ANTLR, http://www.antlr.org/about.html.

Chapter 6

OpenModelica Eclipse Plugin – MDT

To be updated, until then, consult the Modelica Development Tooling (MDT) website:

http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/MDT

Chapter 7

How to Write Test Cases for OpenModelica Development

This chapter is a "how-to" guide to aid in developing testcases for the omc testsuite. At the end of the file there are examples to illustrate the guide.

7.1 Getting Started

In case you plan to develop several testcases it might be beneficial to have a separate working directory in the testsuite directory.To set this up you need to copy some files to that directory. Copy `rtest`, `translation_template.mo`, `translation_failed_template.mo`, `simulation_template.mos`, and `simulation_failed_template.mos`.

Depending on where in the directory hierarchy you put your subdirectory `<DIRECTORY>` including the rtest script, you may need to modify the path "`../../build/bin/omc`" in the following line in the `rtest` file:

```
system "MODELICAUSERCFLAGS=$info{cflags} ../../build/bin/omc $f >$log2>&1";
```

In order to test your testcase you want to be able to run just a single case at the time. To do this, edit `Makefile.omdev.mingw` under the OpenModelica directory. Add the following two lines (perhaps also including dependencies?):

```
mytest:
      (cd testsuite/<DIRECTORY>; rtest -v XXX.mos)
```

Here `<DIRECTORY>` is the specific directory where your testcase is saved.

Then in order to run your testcase, simply type the command `mytest` when you build the project using the Eclipse MDT plugin (Ctrl + B).

7.2 Developing a Test Case

A complete testcase consists of 2 separate files. The .mo file containing the model you are running your tests on and a .mos file containing the test script.

7.2.1 Creating the .mo File

Open `translation_template.mo` or `translation_failed_template.mo`, depending on if the translation should fail or not.

- Save the file with a name of your choice. (Don't just copy the content to the new file since it might result in errors.)

- Change the XXX to appropriate names.
- Write the code for the test model. In case your model is supposed to translate add the flat code at the bottomof the file (as seen in the template file).

In order to obtain the flat file, enter the following command:

```
>omc.exe XXX.mo
```

at the command prompt. Copy the result to the bottom of your .mo file. It is important that you maintain all information from the flattened file, including white spaces.

When commenting the flattened code as seen in the template ensure that there is a white space after each '//' (as in the template).

7.2.2 Creating the .mos File

Open one of the templates simulation_template.mos, simulation_failed_template.mos depending on whether your testcase should be simulated successfully or not. Save it with preferably the same name as the .mo file.

7.2.2.1 Simulation not Failing

The `simulation_template.mos` file is used when the simulation should not fail.

- Change <XXX> in loadfile to the .mo file name.
- Change <XXX> in the rest of the file to the class or model name that should be simulated (the last model/class in the mo file)
- Add appropriate `startTime`, `stopTime`, and `numberOfIntervals` in simulate.
- Change the variables in `readSimulationResult` to the variables you want to test/check.

To get all the values from a variable in the simulation you use res[1] for the first variable you added in readSimulationResult and res[2] for the second one and so on.

The `res[X]` is an array of all simulated values from that variable with the size of `readSimulationResultSize("<XXX>_res.plt");` The size of the simulation result depends on the interval set in `simulate`. To get a specific value in the set/array you use `res[X,Y]`.

To get a value at a specific time in the simulation you must manually look it up in the `<XXX>_res.plt` file.

To do that you have to out comment the line `system("rm ....")` in the .mos file and run the test. Then the result files will not be removed.

This is not very practical. There is a script function called val that can get the value for a specific time. It's used like `val(variableName,time)`. However, the function currently works only on scalar variables, not array elements.

Get the values you are going to test as described above. In the template file there is an example of how you can round the values to 3 digits/decimals.

```
x:=res[1]; // get the values
x:=1000*x; // multiply the values with 1000
x:=floor(x); // remove the decimals
echo(true); // turns on output
x/1000.0; // divide it with 1000 -> 3 digits/decimals and prints it.
```

Remove:

```
// {1.0,1.654,2.169,2.62,3.032,3.418}
// {2.0,2.0,2.0,2.0,2.0,2.0}
// {3.0,2.545,2.23,1.979,1.767,1.581}
```

and add the expected result for your test variables. One way to obtain the expected values is to simulate the model in another simulator or compute the results manually.

7.2.2.2 Simulation Fail

The `simulation_failed_template.mos` is used when the simulation should fail.

- Change <XXX> in loadfile to the .mo file name.
- Change <XXX> in simulate to the class or model name that should be simulate (the last class/model in the .mo file)

Then remove

```
//"#Error, too few equations. Underdetermined system
// The model has 3 variables and 2 equations
```

and replace it with the error message expected for your model.

Note::

The expected values and the errormessage will be matched towards the printout from the simulation. Thus the expected values and error messages have to be exactly the same as the printout or the test will fail.

Hints:

change the template mos file.

```
size:=readSimulationResultSize("<XXX>_res.plt");
res:=readSimulationResult("<XXX>_res.plt",{x,y,z},size);
```

7.3 Status of Simulated Test Cases

7.3.1 Status for .mo Files

There are three different cases of mo.files.

1. The .mo file is correct and translates. Then status shall be correct.

2. The .mo file is inaccurate and thus it won't translate. Status shall then be incorrect.

3. The .mo file is correct according to the modelica language specification but it has features not yet implemented in the omc compiler. Status shall be set to correct. These tests however will be added differently to the testsuite.

7.3.2 Status for .mos Files

Status on `.mos` files should always be set to correct.

7.4 Adding Test Cases to the Suite

Move the files to the dir where they should be and add the new mo and. mos files to the makefile. Normal correct testcases should be added at the TESTCASE label (like example 1 below). Testcases that are using features yet not implemented in OMC should be added to the failing test label.

For testcases that have 'planted' errors in the mo-file and a 'simulation_failed' .mos file (like example 2 below), the mo-file should be added as a failing test and the .mos file as a normal test file.

7.5 Examples

7.5.1 Correct Test

MO-FILE

```
// name:     Example1
// keywords:
// status:   correct
//
// Simple example
//

model Ex1
    Integer x;
    equation
      x = 2+3;
end Ex1;

// fclass Ex1
// Integer x;
// equation
//   x = 5;
// end Ex1;
```

MOS-file

```
// name:     Example1
// keywords:
// status:   correct
//
// Simple example
loadFile("Example1.mo");
simulate(Ex1,startTime=0.0, stopTime=1.0, numberOfIntervals=2); // 2 intervals ==
3 values
echo(false); // turns of output
size := readSimulationResultSize("Ex1_res.plt");
res:=readSimulationResult("Ex1_res.plt",{x},size);
x1:=res[1,1]; //Gets the simulated value of the model variable x at the time 0
x2:=res[1,size]; //Gets the value of the model variable x at stoptime.

echo(true); // turns on output

x1; //prints x1, expecting 5.0
x2; //prints x2, expecting 5.0

readFile("output.log"); // Check that output log is emtpy
system("rm -rf Ex1_* Ex1.exe Ex1.cpp Ex1.makefile Ex1.libs Ex1.log output.log");

// Result:
// true
// record
//     resultFile = "Ex1_res.plt"
// end record
// true
// 5.0
// 5.0
// ""
// 0
// endResult
```

7.5.2 Failing Test

MO-FILE

```
// name:     Example2
// keywords:
// status:   incorrect
//
// Simple example
//

model Ex2
    Integer x = 5.5; //Type mismatch
    equation
      x = 5;
end Ex2;
```

MOS-FILE

```
// name:     Example2
// keywords:
// status:   correct
//
// Simple example

loadFile("Example2.mo");
simulate(Ex2,startTime=0.0, stopTime=1.0, numberOfIntervals=2);
// 2 intervals == 3 values
getErrorString(); // simulation failed, check error string.
// Result:
// true
// record
//     resultFile = "Simulation failed.
// Type mismatch in modifier, expected Integer, got modifier =5.5 of type Real
// Error occured while flattening model Ex2
// "
// end record
// ""
// endResult
```

Appendix A

Exercises (?? Incomplete, version 070204)

The following are some exercises mostly related to the OpenModelica Compiler (omc), but also about writing a test script and using the Corba client-server interface.

A.1 Exercise SimpleTestCase – Write a Simple Test Case

Write your own testcase MyHelloWorld.mo as a MyHelloWorld.mos file and add it to the test suite. For example, modify the existing HelloWorld.mo, e.g. by changing the equation, run it within OMNotebook or OMShell, check the values at a few points using the val-function – val(x,time). Use these to design your own .mos file.

Also read Chapter 7 in this document which gives more detailed instructions.

Below is the .mos file that runs and compares with the values in the comments at the end of the file. In the .mo file there is also a flattened version of the file for checking the flattening.

HelloWorld.mos:

```
// name:      HelloWorld
// keywords:  equation
// status:    correct
//
// Equation handling
//
loadFile("HelloWorld.mo");
simulate(HelloWorld, startTime=0.0, stopTime=1.0, numberOfIntervals=2);
echo(false);
size := readSimulationResultSize("HelloWorld_res.plt");
res:=readSimulationResult("HelloWorld_res.plt",{x},size);
x := res[1];
x := 1000*x;
x := floor(x);  ??? Should perhaps be re-written using the val-function?
echo(true);
x/1000.0;
readFile("output.log");
system("rm -rf HelloWorld_* HelloWorld.exe HelloWorld.cpp HelloWorld.makefile
HelloWorld.libs HelloWorld.log output.log");
// Result:
// true
// record
//     resultFile = "HelloWorld_res.plt"
// end record
// true
// {1.0,0.999,0.999,0.606,0.367}
// ""
// 0
// endResult
```

HelloWorld.mo:

```
// name:     HelloWorld
// keywords: equation
// status:   correct
```

```
//
// Equation handling
//

model HelloWorld
      Real x(start = 1);
      parameter Real a = 1;
equation
      der(x) = - a * x;
end HelloWorld;

// fclass HelloWorld
// Real x(start = 1.0);
// parameter Real a = 1;
// equation
//   der(x) = -(a * x);
// end HelloWorld;
```

A.2 Exercise UseAPIFunctions – Call Some OMC API Functions

Take a look at the API table in Section 2.4.3 and in the notebook QueryAPIExamples in the testcases directory under the OpenModelica installation.

```
** Call a few API function.
```

A.3 Exercise OMCCorbaJava – Commands via Corba from a Java Client

In this exercise you will send commands to the OMC compiler via the Corba interface. Please switch to the Java perspective for this exercise. In this exercise you just play around with the Java Corba interface to omc.

A.3.1 How Corba Communication Works

When OMC is started with: `omc[.exe] +d=interactiveCorba`, it writes a file in the temporary directory with its Corba Object reference. The file is called differently depending on the OS. In Windows: `openmodelica.objid` and in Linux: `openmodelica.USERNAME.objid` where USERNAME is the name of the current user. The Corba clients check if this file exists, read it and use it to initialize the Corba code that connects to OMC. The code in general looks like this:

```
ORB orb;
OmcCommunication omcc;

orb = ORB.init(args, null);

/* Convert string to object. */
org.omg.CORBA.Object obj = orb.string_to_object(stringifiedObjectReference);

/* Convert object to OmcCommunication object. */
omcc = OmcCommunicationHelper.narrow(obj);
```

In the code above the variable `stringifiedObjectReference` represents the contents read from the `openmodelica.[USERNAME.]objid` file.

All the `OmcCommunication*.java` files are generated using an Corba IDL compiler from a very simple `omc_coomunication.idl` file with the following contents:

```
// As simple as can be omc communication, sending and recieving of strings.
interface OmcCommunication {
  string sendExpression( in string expr);
  string sendClass( in string model);
};
```

Please reffer to Corba documentation (for example http://www.mico.org) for more information about the IDL Compiler and ORB.

A.3.2 OMCProxy.java

Provides implementation for:

- starting the OpenModelica compiler: omc[.exe] depending on the platform (Windows/Linux). See method: `startServer()`.
- sending expressions to OMC and receiving results. See method: `String sendExpression(String e)`.
- initialization of Corba communication. See method: `setupOmcc(String objReference)`.

A.4 Corba Clients for C++ and Python

If you are interested in calling OpenModelica compiler OMC from other languages we have available OMC clients for C++ and Python here: http://www.ida.liu.se/~adrpo/omc/corba/

A.5 Exercise newAPIFunction – Write a new Simple OMC API Function

Write your own simple function myOwnAPIFunction() with no arguments that returns the string "myString"

- Look in the file `Interactive.mo`.
- Locate function `evaluateGraphicalApi2`.
- Look at the cases for some existing API functions, e.g. the one below.
- Add your own case for a simple function `myOwnAPIFunction()`.

Below you find a case rule for one of the existing functions `getEnvironmentVar(...)`:

```
algorithm
  (outString,outInteractiveSymbolTable):=
  matchcontinue (inInteractiveStmts,inInteractiveSymbolTable)

   case (ISTMTS(interactiveStmtLst = {IEXP(exp = Absyn.CALL(
   function_ = Absyn.CREF_IDENT(name = "getEnvironmentVar"),
   functionArgs = Absyn.FUNCTIONARGS(args = {Absyn.STRING(value = name)},
                                     argNames = {}))})),
   (st as SYMBOLTABLE(ast = p,explodedAst = s,instClsLst = ic,
    lstVarVal = iv,compiledFunctions = cf))
   )
      equation
        resstr = System.readEnv(name);
      then
        (resstr,st);
```

A.6 Exercise ASTExpTransform – Write A Small Exp AST Transformation

Write a small AST transformation, e.g. in the Exp package, for example to simplify an expression. For example, you can transform small powers of 3, e.g. x^3, to corresponding multiplications, e.g. x*x*x.

A.7 Exercise CodeGen – Generate Code for a new Builtin Function

Make a small change in the code generator. (e.g. add a compiler-known builtin function twice(x) that generates the code x+x, or mySin2(x) for computing sin(x)+2, or change an existing function (floor), or something of your choice, etc.)

Depending on your ambitions, you need to change two or more of the following files. Changes to at least Builtin.mo and Codegen.mo are necessary.

- Builtin.mo – This package creates a top-level environment with all predefined classes and types.
- Static.mo – This package performs type checking and certain cases of symbolic simplification.
- Ceval.mo – This package performs evaluation of constant expressions.
- Codegen.mo – This package performs code generation.

A simple method is to search for the string `"fill"` for the builtin function `fill` in the above .mo-files. Then you easily find the places where to insert code for your own builtin function.

A.8 Exercise getClassNamesRecursive – Recursive Printout of Class Names in a Model Hierarchy

Write an API function: `getClassNamesRecursive(cref)` where cref=Component Reference.

This function should display all the loaded classes/packages hierarchically to the last depth

- each level should be indented
- An example of output is given below

Example call:

```
loadModel(Modelica)
getClassNamesRecursive(Modelica)
```

Output:

```
Modelica [package]
  Blocks [package]
    Continous [package]
      Der [block]
      Derivative [block]
      ....
    Discrete [package]
  Constants  [package]
  Electrical [package]
  Icons      [package]
  Math       [package]
  Mechanics  [package]
  SIunits    [package]
  UsersGuide [package]
```

Hints:

- Start from “getClassNames” and think about how you can write some functions to get the output above. See also getClassRestriction(cref) .

Appendix B

Solutions to Exercises (??Incomplete)

The following are solutions to some exercises in Appendix A.

B.1 Solution SimpleTestCase – Write a Simple Test Case

One possible solution (?? need to update this)

MyHelloWorld.mos:

```
// name:      HelloWorld
// keywords:  equation
// status:    correct
//
// Equation handling
//
loadFile("HelloWorld.mo");
simulate(HelloWorld, startTime=0.0, stopTime=1.0, numberOfIntervals=2);
echo(false);
size := readSimulationResultSize("HelloWorld_res.plt");
res:=readSimulationResult("HelloWorld_res.plt",{x},size);
x := res[1];
x := 1000*x;
x := floor(x);  ??? Should perhaps be re-written using the val-function?
echo(true);
x/1000.0;
readFile("output.log");
system("rm -rf HelloWorld_* HelloWorld.exe HelloWorld.cpp HelloWorld.makefile
HelloWorld.libs HelloWorld.log output.log");
// Result:
// true
// record
//     resultFile = "HelloWorld_res.plt"
// end record
// true
// {1.0,0.999,0.999,0.606,0.367}
// ""
// 0
// endResult
```

HelloWorld.mo:

```
// name:     HelloWorld
// keywords: equation
// status:   correct
//
// Equation handling
//

model HelloWorld
      Real x(start = 1);
      parameter Real a = 1;
equation
      der(x) = - a * x;
end HelloWorld;
```

```
// fclass HelloWorld
// Real x(start = 1.0);
// parameter Real a = 1;
// equation
//   der(x) = -(a * x);
// end HelloWorld;
```

B.2 Solution UseAPIFunctions – Call Some OMC API Functions

?? fill in

```
** Call a few API functions.
```

B.3 Solution OMCCorbaJava – Commands via Corba from a Java Client

No solution. Just play around with the existing Java Corba communication.

B.4 Solution Corba Clients for C++ and Python

No solution. Just play around with the existing C++ or Python Corba communication implementation.

B.5 Solution newAPIFunction – Write a new Simple OMC API Function

```
case (ISTMTS(interactiveStmtLst = {
  IEXP(exp = Absyn.CALL(function_ = Absyn.CREF_IDENT(name = "myOwnAPIFunc")))}),
  (st as SYMBOLTABLE(ast = p,explodedAst = s,instClsLst = ic,
     lstVarVal = iv,compiledFunctions = cf)))
      equation
        resstr = "returned from myOwnAPIFunc";
      then
        (resstr,st);
```

B.6 Solution ASTExpTransform – Write A Small Exp AST Transformation

?? fill in.

B.7 Solution CodeGen – Generate Code for a new Builtin Function

?? fill in.

B.8 Solution getClassNamesRecursive – Recursive Printout of Class Names in a Model Hierarchy

Note: This solution does not display the restriction after the class name. We leave that implementation part for the reader.

Inserted into the function `evaluateGraphicalAPI` in `Interactive.mo`:

```
    case (ISTMTS(interactiveStmtLst = {IEXP(exp = Absyn.CALL(function_ =
Absyn.CREF_IDENT(name = "getClassNamesRecursive"),
  functionArgs = Absyn.FUNCTIONARGS(args = {Absyn.CREF(componentReg = cr)}))))}),
(st as SYMBOLTABLE(ast = p,explodedAst = s,instClsLst = ic,
 lstVarVal = iv,compiledFunctions = cf)))
      local Absyn.Path path;
      equation
        path = Absyn.crefToPath(cr);
```

```
        resstr = getClassNamesRecursive(path, p, "");
      then
        (resstr,st);


protected function getClassnamesInClassList
  input Absyn.Path inPath;
  input Absyn.Program inProgram;
  input Absyn.Class inClass;
  output list<String> outString;
algorithm
  outString:=
  matchcontinue (inPath,inProgram,inClass)
    local
      list<String> strlist;
      list<String> res;
      list<Absyn.ClassPart> parts;
      Absyn.Class cdef;
      Absyn.Path newpath,inmodel,path;
      Absyn.Program p;
    case (_,_,Absyn.CLASS(body = Absyn.PARTS(classParts = parts)))
      equation
        strlist = getClassnamesInParts(parts);
      then
        strlist;
    case (inmodel,p,Absyn.CLASS(body = Absyn.DERIVED(path = path)))
      equation
        (cdef,newpath) = lookupClassdef(path, inmodel, p);
        res = getClassnamesInClassList(newpath, p, cdef);
      then
        res;
  end matchcontinue;
end getClassnamesInClassList;


protected function joinPaths
  input String child;
  input Absyn.Path parent;
  output Absyn.Path outPath;
algorithm
  outPaths:=
  matchcontinue (child, parent)
    local
      Absyn.Path r, res;
      String c;
    case (c, r)
      equation
        res = Absyn.joinPaths(r, Absyn.IDENT(c));
      then res;
  end matchcontinue;
end joinPaths;


protected function getClassNamesRecursive "function: getClassNamesRecursive
  Returns a string with all the classes for a given path.
"
  input Absyn.Path inPath;
  input Absyn.Program inProgram;
  input String indent;
  output String outString;
algorithm
  outString:=
  matchcontinue (indent,inPath,inProgram)
    local
```

```
      Absyn.Class cdef;
      String s1,res, parent_string, result;
      list<String> strlst;
      Absyn.Path pp, modelpath;
      Absyn.Program p;
      String indent;
      list<Absyn.Path> result_path_lst;
    case (pp,p,indent)
      equation
        cdef = getPathedClassInProgram(pp, p);
        strlst = getClassnamesInClassList(pp, p, cdef);
        parent_string = Absyn.pathString(pp);
        result_path_lst = Util.listMap1(strlst, joinPaths, pp);
        indent = indent +& "   ";
        result = Util.stringAppendList(Util.listMap2(result_path_lst,
          getClassNamesRecursive, p, indent));
        res = Util.stringAppendList({parent_string,"\n",indent, result});
      then
        res;
    case (_,_,_) then "Error";
  end matchcontinue;
end getClassNamesRecursive;
```

Appendix C

Contributors to OpenModelica

This Appendix lists the individuals who have made significant contributions to OpenModelica, in the form of software development, design, documentation, project leadership, tutorial material, etc. The individuals are listed for each year, from 1998 to the current year: the project leader and main author/editor of this document followed by main contributors followed by contributors in alphabetical order.

C.1 OpenModelica Contributors 2008

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.

David Akhvlediani, PELAB, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Vasile Baluta, PELAB, Linköping University, Linköping, Sweden.
Mikael Blom, PELAB, Linköping University, Linköping, Sweden.
Kristoffer Norling, PELAB, Linköping University, Linköping, Sweden.
Klas Sjöholm, PELAB, Linköping University, Linköping, Sweden.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Henrik Eriksson, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Kim Jansson, PELAB, Linköping University, Linköping, Sweden.
Joel Klinghed, PELAB, Linköping University, Linköping, Sweden.
Simon Björklén, PELAB, Linköping University, Linköping, Sweden
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
Anders Sandholm, PELAB, Linköping University, Linköping, Sweden.
Eric Meyers, Pratt & Whitney Rocketdyne, Palm City, Florida, USA.
Pavel Grozman, Equa AB, Stockholm, Sweden.
Constantin Belyaev, Bashpromavtomatika Ltd., Ufa, Russia

C.2 OpenModelica Contributors 2007

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.

David Akhvlediani, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.

Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Kristoffer Norling, Linköping University, Linköping, Sweden.
Anders Sandholm, PELAB, Linköping University, Linköping, Sweden.
Klas Sjöholm, Linköping University, Linköping, Sweden.
Simon Björklén, PELAB, Linköping University, Linköping, Sweden
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
William Spinelli, Politecnico di Milano, Milano, Italy
Stefan Vorkoetter, MapleSoft, Waterloo, Canada.
Constantin Belyaev, Bashpromavtomatika Ltd., Ufa, Russia

C.3 OpenModelica Contributors 2006

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.

David Akhvlediani, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Elmir Jagudin, PELAB, Linköping University, Linköping, Sweden.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Kaj Nyström, PELAB, Linköping University, Linköping, Sweden.
Lucian Popescu, MathCore Engineering AB, Linköping, Sweden.
Andreas Remar, PELAB, Linköping University, Linköping, Sweden.
Anders Sandholm, PELAB, Linköping University, Linköping, Sweden.

C.4 OpenModelica Contributors 2005

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, PELAB, Linköping University and MathCore Engineering AB, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.

Ingemar Axelsson, PELAB, Linköping University, Linköping, Sweden.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Kaj Nyström, PELAB, Linköping University, Linköping, Sweden.
Lucian Popescu, MathCore Engineering AB, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.

C.5 OpenModelica Contributors 2004

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, Linköping University, Linköping, Sweden.

Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
Peter Bunus, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Emma Larsdotter Nilsson, PELAB, Linköping University, Linköping, Sweden.

Kaj Nyström, PELAB, Linköping University, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Lucian Popescu, MathCore Engineering AB, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.

C.6 OpenModelica Contributors 2003

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, Linköping University, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.

Peter Bunus, PELAB, Linköping University, Linköping, Sweden.
Vadim Engelson, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, Linköping University, Linköping, Sweden.
Eva-Lena Lengquist-Sandelin, PELAB, Linköping University, Linköping, Sweden.
Susanna Monemar, PELAB, Linköping University, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Erik Svensson, MathCore Engineering AB, Linköping, Sweden.

C.7 OpenModelica Contributors 2002

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, Linköping University, Linköping, Sweden.

Daniel Hedberg, Linköping University, Linköping, Sweden.
Henrik Johansson, PELAB, Linköping University, Linköping, Sweden
Andreas Karström, PELAB, Linköping University, Linköping, Sweden

C.8 OpenModelica Contributors 2001

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, Linköping University, Linköping, Sweden.

C.9 OpenModelica Contributors 2000

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

C.10 OpenModelica Contributors 1999

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden

–

Peter Rönnquist, PELAB, Linköping University, Linköping, Sweden.

C.11 OpenModelica Contributors 1998

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

David Kågedal, PELAB, Linköping University, Linköping, Sweden.

Vadim Engelson, PELAB, Linköping University, Linköping, Sweden.

Index

Error! No index entries found.