OpenModelica System Documentation

Version 0.2, April 2005

Preliminary Incomplete Draft, 2005-04-18

PELAB – Programming Environment Laboratory
Department of Computer and Information Science
Linköping University, Sweden

The OSM (OpenModelica) License (Version 1.1 of March 4, 2005)

License Disclaimer

Table of Contents

Preface

This system documentation has been prepared to simplify further development of the OpenModelica compiler. 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 April 2005, primarily includes information about the OpenModelica compiler. Sections about the other subsystems in the OpenModelica environment will be added in the future.

1.1 OpenModelica Environment Structure

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

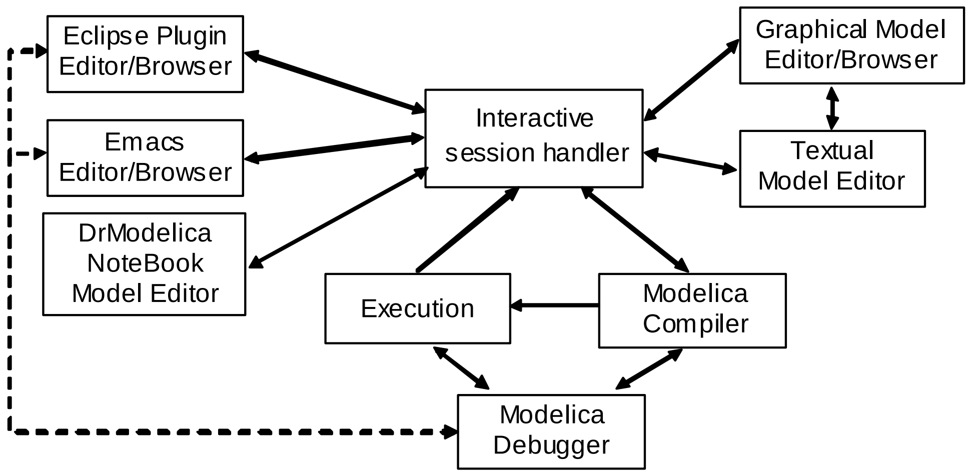
Eclipse Plugin Editor/Browser
Emacs Editor/Browser
DrModelica NoteBook Model Editor
Interactive session handler
Graphical Model Editor/Browser
Textual Model Editor
Execution
Modelica Compiler
Modelica Debugger

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.

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.

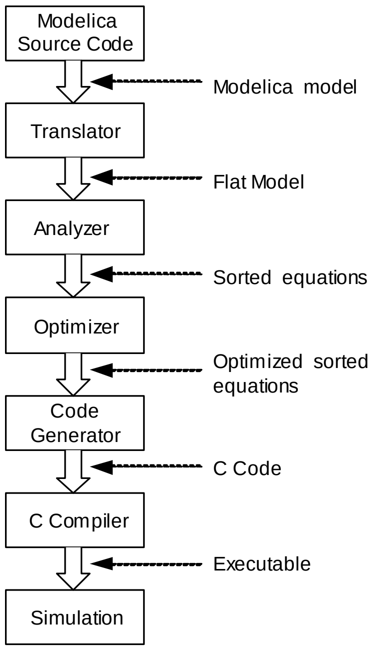

Figure 1-2. Translation stages from Modelica code to executing simulation.

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):

```
relation main =

  rule  Parser.parse(f) => ast &
        SCode.elaborate(ast) => scode1 &
        Inst.elaborate(scode1) => scode2 &
        DAE.dump(scode2) &
        ----------------
        main([f])
end
```

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.

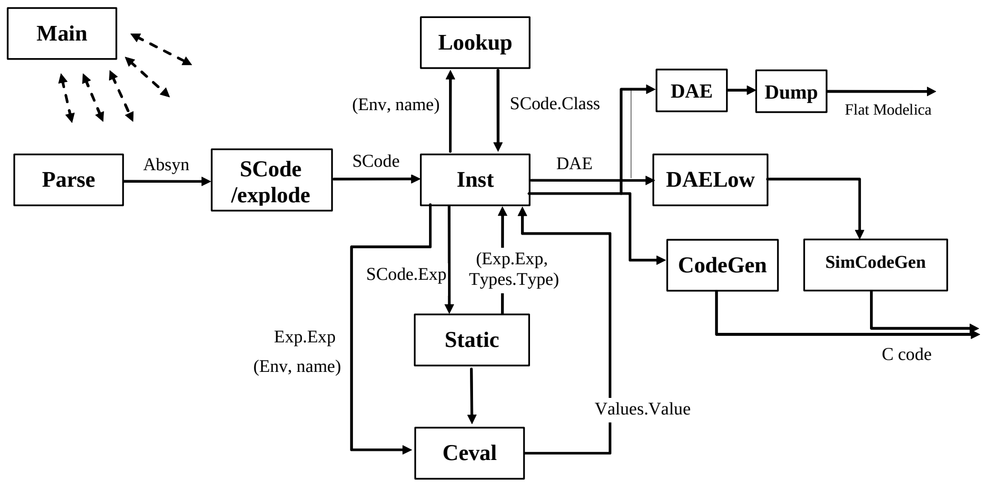

Figure 1-3. Some module connections and data flows in the OpenModelica compiler. The parser generates abstract syntax (Absyn) which is converted to the simplified (SCode) intermediate form. The code instantiation module (Inst) calls Lookup to find a name in an environment. It also generates the DAE equation 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 RML 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 Code 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 *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 elaboration of the model. The convention is that the last model 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 inst_class and inst_element Functions

The function `inst_class` elaborates a class. It takes five arguments, the environment `env`, the set of modifications `mod`, the prefix `pre` 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 `c`. It opens a new scope in the environment where all the names in this class will be stored, and then uses a function called `inst_class_in` 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 `inst_element`, 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 for elaborating variable declarations.

The following are simplified versions of the inst_class and inst_element functions.

```
relation inst_class: (Env, Mod, Prefix, Connect.Sets, Scode.Class) =>
(DAE.Element list, Connect.Sets, Types.Type) =

  rule Env.open_scope(env) => env' &
     elab_class_in(env',mod,pre,csets,c) => (dae1,_,csets',ci_state', tys) &
     Connect.equations csets' => dae2 & list_append(dae1, dae2) => dae &
     mktype(ci_state',tys) => ty
     -------------------------------------------------------------------
     elab_class(env,mod,pre,csets,c as SCode.CLASS(n,_,r,_)) => (dae, [], ty)
end
```

```
relation inst_element: (Env, Mod, Prefix, Connect.Sets, Scode.Element) =>
(DAE.Element list, Env, Connect.Sets, Types.Var list) =
...
rule Prefix.prefix_cref(pre,Exp.CREF_IDENT(n,[]))  => vn &
     Lookup.lookup_class(env,t)  => (cl,classmod) & Find the class definition
     Mod.lookup_modification(mods,n)=>mm &
     Mod.merge(classmod,mm) => mod &               Merge the modifications
     Mod.merge(mod,m) => mod' &
     Prefix.prefix_add(n,[],pre) => pre' &         Extend the prefix
     elab_class(env,mod',pre',csets,cl)            Elaborate the variable
          => (dae1,csets',ty,st) &
     Mod.mod_equation mod' => eq &  If the variable is declared with a default equation,
     make_binding (env,attr,eq,cl)   add it to the environment with the variable.
          => binding &
     Env.extend_frame_v(env,           Add the variable binding to the environment
          Env.FRAMEVAR(n,attr,ty,binding))
          => env' &
     elab_mod_equation(env,pre,n,mod')          Fetch the equation, if supplied
          => dae2 &
     list_append(dae1, dae2) => dae                Concatenate the equation lists
     --------------------------
     elab_element(env,mods,pre,csets,
                       SCode.COMPONENT(n,final,prot,attr,t,m))
          => (dae, env',csets',[(n,attr,ty)])
...
end
```

1.8 Output

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

```
datatype DAEcomp = VAR of Exp.ComponentRef * VarKind
                  | EQUATION of Exp * Exp
                    ...
```

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

Chapter 2

Invoking the OpenModelica Compiler/Interpreter Subsystem

The OpenModelica Compiler/Interpreter subsystem 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 currently called modeq (Modelica equation compiler). The name will probably be changed in the future. The compiler can be given file arguments as specified below, and flags that are described in the subsequent sections.

`modeq file.mo`	Return flat Modelica by code instantiating the last class in the file `file.mo`
`modeq file.mof`	Put the flat Modelica produced by code instantiation of the last class within `file.mo` in the file named `file.mof`.
`modeq 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:

`modeq +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`.
`modeq +q`	Quietly run the compiler, no output to stdout.
`modeq +d=blt`	Perform BLT transformation of the equations.
`modeq +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.
`modeq +d=interactiveCorba`	Run the compiler in interactive mode with Corba communication. This is the standard communication that is used for the interactive mode.

2.1.2 Compiler Debug Trace Flags

Run modeq with comma separated list of flags without spaces,

```
"modeq +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 modeq is run with the following:

```
modeq +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 `modeq` 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.:

```
modeq 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.

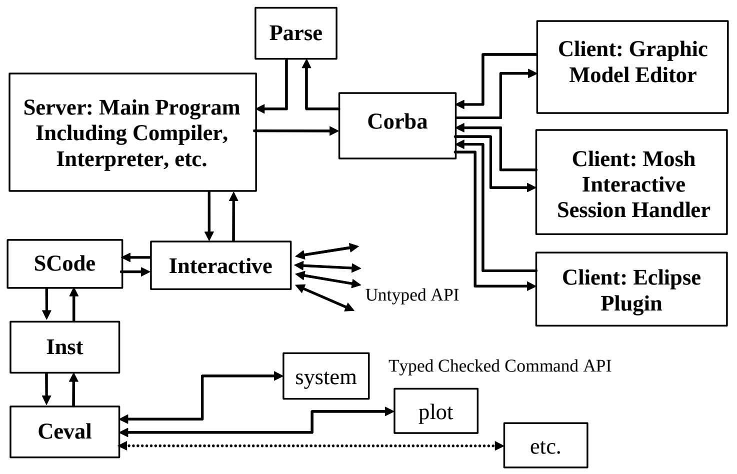
Parse
Server: Main Program Including Compiler, Interpreter, etc.
Corba
Client: Graphic Model Editor
Client: Mosh Interactive Session Handler
Client: Eclipse Plugin
SCode
Interactive
Untyped API
Inst
Ceval
system
Typed Checked Command API
plot
etc.

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.

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.

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;`
`plotParametric``(variables1, variables2, options)` (NotYetImplemented)	Plot each pair of corresponding variables from the vectors of variables `vars1`, `vars2` as a parametric plot. *Inputs*: `VariableName variables1[:]; VariableName variables2[size(variables1,1)]; 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)` (NotYetImplemented)	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)`	Save class definition in a file. *Inputs*: `String fileName; TypeName className` *Outputs*: `Boolean res;`
`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

The following API is primarily designed for clients calling the OpenModelica compiler/interpreter via the Corba interface, 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 Example 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`.

2.4.3 Untyped API Functions

The following are brief descriptions of the untyped API functions available in the OpenModelica environment.

`saveModel``(A1<string>,A2<cref>)`	Saves the model (A2) in a file given by a string (A1). This call is also in typed API.
`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.
`loadModel``(A1<cref>)`	Loads the model (A1) by looking up the correct file to load in `$MODELICAPATH`. Loads all models in that file into the symbol table.
`deleteClass``(A1<cref>)`	Deletes the class from the symbol table.
`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`.
`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.
`getComponents``(A1<cref>)`	Returns a list of the component declarations within class A1: `{{Atype,varidA,"commentA"},{Btype,varidB,"commentB"}, {...}}`
`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.
`getComponentCount``(A1<cref>)`	Returns the number (as a string) of components in a class, e.g return `"2"` if there are 2 components.
`getNthComponent``(A1<cref>,A2<int>)`	Returns the belonging class, component name and type name of the nth component of a class, e.g. `"A.B.C,R2,Resistor"`, where the first component is numbered 1.
`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.
`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.
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.
addEquation(A1<cref>,A2<expr>, A3<expr>) (NotYetImplemented)	Adds the equation A2=A3 to the model named by A1.
getEquationCount(A1<cref>) (NotYetImplemented)	Returns the number of equations (as a string) in the model named A1. (This includes connections)
getNthEquation(A1<cref>,A2<int>) (NotYetImplemented)	Returns the nth (A2) equation of the model named by A1. e.g. `"der(x)=-1"` or `"connect(A.b,C.a)"`. The first has number 1.
deleteNthEquation(A1<cref>, A2<int>) (NotYetImplemented)	Deletes the nth (A2) equation in the model named by A1. The first has number 1.
getConnectorCount(A1<cref>)	Returns the number of connectors (as a string) of a class A1. NOTE: partial code instantiation of inheritance is performed before investigating the connector count, in order also to get the inherited connectors.
getNthConnector(A1<cref>,A2<int>)	Returns the name of the nth connector, e.g `"n"`. The first connector has number 1.
getNthConnectorIconAnnotation(A1<cref>,A2<int>)	Returns the nth connector icon layer annotation as comma separated list of values of a flat record, see Annotation below. NOTE: Since connectors can be inherited, a partial instantiation of the inheritance structure is performed. The first has number 1.
getNthConnectorDiagramAnnotation (A1<cref>,A2<int>) (NotYetImplemented)	Returns the nth connector diagram layer annotation as comma separated list of values of a flat record, see Annotation below. NOTE: Since connectors can be inherited, a partial instantiation of the inheritance structure is performed. The first has number 1.
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. NOTE: 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.
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.
isType(A1<cref>)	Returns `"true"` if class is a type, otherwise `"false"`.
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"`.
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"`.
isConstant(A1<cref>)	Returns `"true"` if A1 is a constant, otherwise `"false"`.
isProtected(A1<cref>)	Returns `"true"` if A1 is protected, otherwise `"false"`.
existClass(A1<cref)	Returns `"true"` if class exists in symbolTable, otherwise `"false"`.
existModel(A1<cref>)	Returns `"true"` if class exists in symbol table and has restriction model, otherwise `"false"`.
existPackage(A1<cref>)	Returns `"true"` if class exists in symbol table and has restriction package, otherwise `"false"`.

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 Modules

This chapter gives overviews of all modules 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 Modules

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)

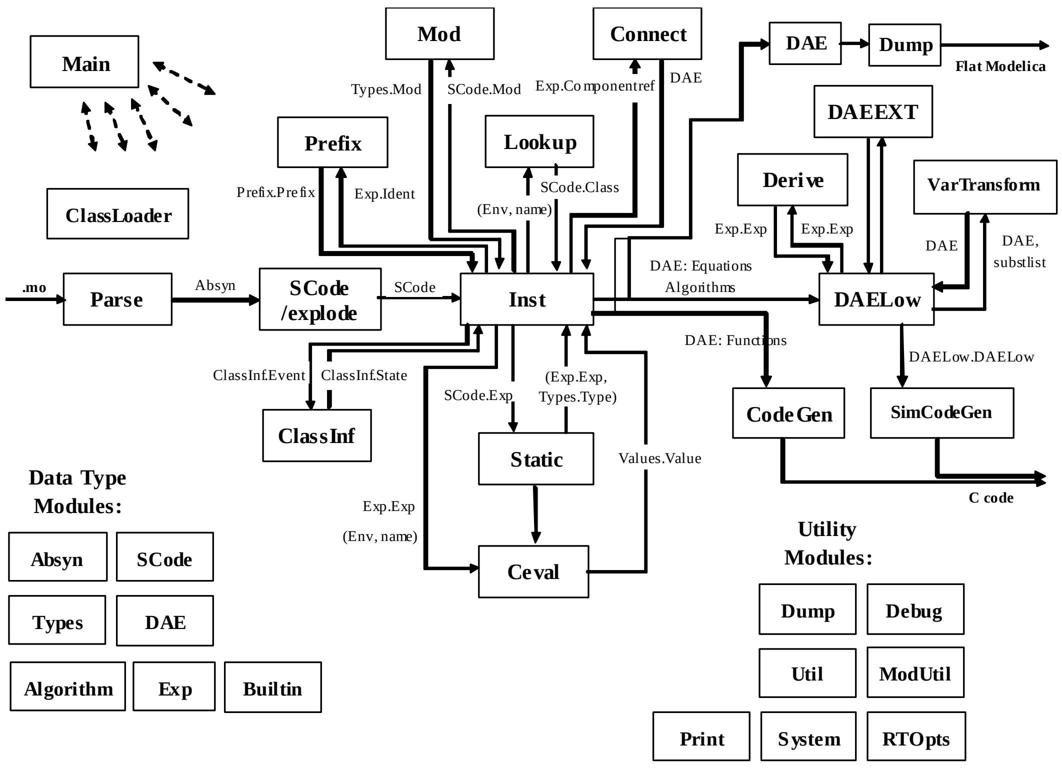

Figure 3-1. Module connections and data flows in the OpenModelica compiler.

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).

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 modelica/modeq/

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

3.2.2 modelica/modeq/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 modelica/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 modelica/mosh

Files for the OpenModelica interactive shell, called `MOSH` for Modelica Shell.

3.2.5 modelica/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 $MODELICAPATH
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
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.
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
Prefix	Handling Prefixes in Variable Names
Print	Buffered Printing to Files and Error Message Printing

RTOpts	Run-time Command Line Options
SCode	File explode.rml, Simple Lower Level Intermediate Code Representation. (*explode.rml *)
SimCodegen	Generate simulation code for solver from equations and algorithm sections in DAE.
Socket	(*Depreciated, removed) OpenModelica Socket Communication Module
Static	Static Semantic Analysis of Expressions (* file staticexp.rml *)
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 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 RML. It primarily contains datatypes for constructing the abstract syntax tree (AST), functions for building and altering datatypes and a few functions for printing the AST:

- Abstract Syntax Tree (Close to Modelica)
  - Complete Modelica 2.1 (*almost*)
  - 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.:
    - `path_to_cref(Path) => ComponentRef`
    - `join_paths(Path, Path) => (Path)`
    - `etc.`

The constructors defined by the Absyn module are primarily used by the walker (`modeq/absyn_builder/walker.g`) which takes an ANTLR internal syntax tree and converts it into an RML abstract syntax tree. When the AST has been built, it is normally used by the SCode/explode 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
```

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` statement that indicates the hierarchical position of the program.

```
datatype Program = PROGRAM of Class list      (* List of classes *)
                        * Within              (* Within statement *)
              | BEGIN_DEFINITION   of Path    (* For split definitions*)
                        * Restriction         (* Class restriction *)
                        * bool                (* Partial *)
                        * bool                (* Encapsulated *)
              | END_DEFINITION of Ident        (* For split definitions *)
              | COMP_DEFINITION of ElementSpec (* For split definitions*)
                              * Path option (* insert into.
                                             Default, NONE *)
              | IMPORT_DEFINITION of ElementSpec(* For split definitions*)
                              * Path option (* insert into.
                                             Default, NONE *)
```

Within statements:

```
datatype Within = WITHIN of Path | TOP
```

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.

```
datatype Class = CLASS of Ident           (* Name *)
                   * bool                 (* Partial *)
                   * bool                 (* Final *)
                   * bool                 (* Encapsulated *)
                   * Restriction          (* Restricion *)
                   * ClassDef             (* Body *)
```

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. An enumeration type contains a list of ??.

```
datatype ClassDef = PARTS of ClassPart list
                        * string option          (* string comment *)
               | DERIVED of Path
                        * ArrayDim option     (* *)
                        * ElementAttributes
                        * ElementArg list
                        * Comment option         (* comment *)
               | ENUMERATION of EnumLiteral list
                        * Comment option         (* comment*)
               | OVERLOAD of Path list           (* function names *)
                        * Comment option
```

EnumLiteral:

`EnumLiteral`, which is a name in an enumeration and an optional `Comment`.

```
datatype EnumLiteral = ENUMLITERAL of Ident      (* Literal *)
                      * Comment option              (* comment *)
```

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.

```
datatype ClassPart = PUBLIC of ElementItem list
                | PROTECTED of ElementItem list
                | EQUATIONS of EquationItem list
                | INITIALEQUATIONS of EquationItem list
                | ALGORITHMS of AlgorithmItem list
                | INITIALALGORITHMS of AlgorithmItem list
                | EXTERNAL of ExternalDecl * Annotation option
```

ElementItem:

An element item is either an element or an annotation

```
datatype ElementItem = ELEMENTITEM of Element
                  | ANNOTATIONITEM of Annotation
```

Elements:

The basic element type in Modelica.

```
datatype Element =

 ELEMENT of bool          (* final *)
      * bool              (* replaceable *)
      * InnerOuter        (* inner/outer *)
      * Ident             (* Element name *)
      * ElementSpec       (* Actual element specification*)
      * string            (* Source code file *)
      * int               (* Line number *)
      * ConstrainClass option (* only valid for classdef and component*)
```

Constraining type:

Constraining type, must be specified using the extends keyword (not inheritance).

```
type 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"),[])`.

```
datatype ElementSpec = CLASSDEF of bool          (* replaceable *)
                              * Class
                    | EXTENDS of Path * ElementArg list
                    | IMPORT of Import * Comment option
               | COMPONENTS of ElementAttributes (*1.1 also contains Arraydim *)
```

```
                                * Path              (* type name *)
                                * ComponentItem list
```

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

```
datatype InnerOuter = INNER | OUTER | UNSPECIFIED
```

Import:

Import statements of different kinds.

```
datatype Import = NAMED_IMPORT of Ident * Path
            | QUAL_IMPORT of Path
            | UNQUAL_IMPORT of Path
```

ComponentItem:

Collection of component and an optional comment.

```
datatype ComponentItem = COMPONENTITEM of Component * Comment option
```

Component:

Some kind of Modelica entity (object or variable).

```
datatype Component = COMPONENT of Ident           (* component name *)
                          * ArrayDim              (* Array dimensions, if any *)
                          * Modification option   (* Optional modification *)
```

Several component declarations can be grouped together in one ElementSpec by writing them on the same line in the source. This type contains the information specific to one component.

EquationItem:

```
datatype EquationItem = EQUATIONITEM of Equation * Comment option
                      | EQUATIONITEMANN of Annotation
```

AlgorithmItem:

Info specific for an algorithm item.

```
datatype AlgorithmItem = ALGORITHMITEM  of Algorithm * Comment option
                  |  ALGORITHMITEMANN of Annotation
```

Equation:

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

```
datatype Equation = EQ_IF of Exp                 (* Conditional expression *)
                      * EquationItem list        (* true branch *)
                      * (Exp * EquationItem list) list (* elseif branches *)
                      * EquationItem list        (* else branch *)
              | EQ_EQUALS of Exp * Exp           (* Standard 2-side eqn*)
              | EQ_CONNECT of ComponentRef * ComponentRef (* Connect stmt *)
              | EQ_FOR of Ident * Exp * EquationItem list (* For-loops *)
              | EQ_WHEN_E of Exp  (* Condition *)
                        * EquationItem list (* Loop body *)
                        * (Exp * EquationItem list) list (* else when *)
```

```
        | EQ_NORETCALL of Ident * FunctionArgs  (* fcalls without return value *)
```

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`.

```
datatype Algorithm = ALG_ASSIGN of ComponentRef * Exp
                | ALG_TUPLE_ASSIGN of Exp          (*tuple*)
                             * Exp        (* value*)
                | ALG_IF of Exp
                        * AlgorithmItem list           (* true branch *)
                        * (Exp * AlgorithmItem list) list (* elseif *)
                        * AlgorithmItem list           (* else branch *)
                | ALG_FOR of Ident * Exp * AlgorithmItem list
                | ALG_WHILE of Exp * AlgorithmItem list
                | ALG_WHEN_A of Exp
                             * AlgorithmItem list
                             * (Exp * AlgorithmItem list) list (* elsewhen *)
                | ALG_NORETCALL of ComponentRef * FunctionArgs  (* general
                                                 fcalls without return value *)
```

Modifications:

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

```
datatype Modification = CLASSMOD of ElementArg list * Exp option
```

Wrapper for things that modify elements, modifications and redeclarations.

```
datatype ElementArg = MODIFICATION of bool * Each * ComponentRef *
                         Modification option * string option
     | REDECLARATION of bool * Each * ElementSpec * ConstrainClass option
```

Each attribute:

The `each` keyword can be present in both `MODIFICATION`'s and `REDECLARATION`'s.

```
datatype Each = EACH | NON_EACH
```

Component attributes:

Component attributes 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)`.

```
datatype ElementAttributes = ATTR of bool   (* flow *)
                            * Variability  (* parameter, constant etc. *)
                            * Direction
                            * ArrayDim
```

Data attributes:

Variability and direction.

```
datatype Variability = VAR | DISCRETE | PARAM | CONST

datatype Direction = INPUT | OUTPUT | BIDIR
```

Array dimensions:

```
type ArrayDim = Subscript list
```

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.

Expressions:

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

```
datatype Exp = INTEGER of int
           | REAL of real
           | CREF of ComponentRef
           | STRING of string
           | BOOL of bool
           | BINARY of Exp * Operator * Exp (* Binary operations, e.g. a*b *)
           | UNARY of Operator * Exp (* Unary operations, e.g. -(x) *)
           | LBINARY of Exp*Operator*Exp (* Logical binary operations: and,or*)
           | LUNARY of Operator * Exp      (* Logical unary operations: not *)
           | RELATION of Exp * Operator * Exp (* Relations, e.g. a >= 0 *)
           | IFEXP of Exp * Exp * Exp * (Exp * Exp) list  (* If expressions *)
           | CALL of ComponentRef * FunctionArgs      (* Function calls *)
           | ARRAY of Exp list                        (* Array constructor *)
           | MATRIX of Exp list list
           | RANGE of Exp * Exp option * Exp (* Range expressions,
                                              e.g. 1:10 or 1:0.5:10 *)
           | TUPLE of Exp list         (* Tuples used in function calls
                                         returning several values *)
           | END (* array access operator for last element, e.g. a[end]:=1; *)
           | CODE of Code (* Modelica AST Code constructors *)
```

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.

```
datatype Code = C_TYPENAME of Path
           | C_VARIABLENAME of ComponentRef
           | C_EQUATIONSECTION of bool * EquationItem list
           | C_ALGORITHMSECTION of bool * AlgorithmItem list
           | C_ELEMENT of Element
           | C_EXPRESSION of Exp
           | C_MODIFICATION of Modification
```

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

```
datatype FunctionArgs =  FUNCTIONARGS of Exp list * NamedArg list
                 | FOR_ITER_FARG of Exp * Ident * Exp
```

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

```
datatype NamedArg = NAMEDARG of Ident * Exp
```

Operators:

```
datatype Operator = ADD   | SUB    | MUL     | DIV       | POW
               | UPLUS | UMINUS
               | AND   | OR
```

```
                    | NOT
                    | LESS  | LESSEQ | GREATER | GREATEREQ | EQUAL | NEQUAL
```

Subscripts:

The `Subscript` datatype 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.

```
datatype Subscript = NOSUB
                   | SUBSCRIPT of Exp
```

Component references and paths:

A component reference is the fully or partially qualified name of a component. It is represented as a list of identifier--subscript pairs. The type `Path', on the other hand, is used to store references to class names, or names inside class definitions.

```
datatype ComponentRef = CREF_QUAL of Ident * (Subscript list) * ComponentRef
                      | CREF_IDENT of Ident * (Subscript list)

datatype Path = QUALIFIED of Ident * Path
          | IDENT of Ident
```

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.

```
datatype Restriction = R_CLASS
                     | R_MODEL
                     | R_RECORD
                     | R_BLOCK
                     | R_CONNECTOR
                     | R_TYPE
                     | R_PACKAGE
                     | R_FUNCTION
                     | R_ENUMERATION
                     | R_PREDEFINED_INT
                     | R_PREDEFINED_REAL
                     | R_PREDEFINED_STRING
                     | R_PREDEFINED_BOOL
                     | R_PREDEFINED_ENUM
```

Annotation:

An Annotation is a class_modification.

```
datatype Annotation = ANNOTATION of ElementArg list
```

Comment:

```
datatype Comment = COMMENT of Annotation option * string option
```

ExternalDecl:

```
datatype ExternalDecl = EXTERNALDECL of
            Ident option  *      (* The name of the external function *)
            string option *       (* Language of the external function *)
            ComponentRef option * (* output parameter as return value*)
            Exp list      (* only positional arguments, i.e. expression list*)
```

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/explode, 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/explode, 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/explode, 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/explode, Print.

3.4.6 ClassLoader – Loading of Classes from $MODELICAPATH

This module loads classes from $MODELICAPATH. It exports only one function: the load_class function. It is currently (2004-09-27) only 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 implementaton. 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.

```
datatype DAElist = DAE of Element list

type Ident = string
type InstDims = Exp.Subscript list

type StartValue = Exp.Exp option

datatype VarKind = VARIABLE | DISCRETE | PARAM | CONST

datatype Type = REAL | INT | BOOL | STRING | ENUM | ENUMERATION of string list
datatype Flow = FLOW | NON_FLOW

datatype VarDirection = INPUT | OUTPUT | BIDIR   (* Variables in functions *)


datatype Element =
                 VAR of
                   Exp.ComponentRef *
                   VarKind *
                   VarDirection *
                   Type *
                   Exp.Exp option * (* Binding expression e.g. for parameters*)
                   InstDims *
                   StartValue * (* value of start attribute *)
                   Flow * (* Flow of connector variable. Needed for
                         unconnected flow variables *)
                   Absyn.Path list (* The class the variable is instantiated
from *)

                | DEFINE of Exp.ComponentRef * Exp.Exp
                | INITIALDEFINE of Exp.ComponentRef * Exp.Exp
                | EQUATION of Exp.Exp * Exp.Exp
                | WHEN_EQUATION of Exp.Exp        (* Condition *) *
                                 Element list   (* Equations *) *
                                 Element option (* Elsewhen should be of type
                                                  WHEN_EQUATION*)
                | IF_EQUATION   of Exp.Exp      (* Condition *) *
                                 Element list (* Equations of true branch*) *
                                 Element list (* Equations of false branch*)
                | INITIALEQUATION of Exp.Exp * Exp.Exp
                | ALGORITHM of Algorithm.Algorithm
                | INITIALALGORITHM of Algorithm.Algorithm
                | COMP of Ident * DAElist
                | FUNCTION of Absyn.Path * DAElist * Types.Type
                | EXTFUNCTION of Absyn.Path * DAElist * Types.Type * ExternalDecl
                | ASSERT of Exp.Exp

datatype ExtArg = EXTARG of Exp.ComponentRef * Types.Attributes * Types.Type
                | EXTARGEXP of Exp.Exp * Types.Type
                | EXTARGSIZE of Exp.ComponentRef *
                              Types.Attributes *
                              Types.Type * Exp.Exp
                | NOEXTARG

 datatype ExternalDecl = EXTERNALDECL of Ident * (* external function name *)
                                 ExtArg list * (* parameters *)
                                 ExtArg * (* return type *)
                                 string (* language *)
```

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 `dump_function` and `dump_comp_element`. We also have (?? explain more):

```
dump_str: DAElist => string
dump_graphviz: DAElist => ()
dump_debug
```

`dump_comp_element` (classes) calls `dump_elements`, which calls:

```
dump_vars
dump_list equations
dump_list algorithm
dump_list comp_element (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 RML.

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 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..

3.4.16 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.

3.4.17 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 Ident = string

type Env = Frame list

datatype Frame =
   FRAME of
     Ident option *  (* Class name *)
     BinTree * (* List of classes and variables which must be uniquely named *)
```

```
       BinTree * (* List of types, which DO NOT be uniquely named,
                    eg. size(...) is overloaded and has several types *)
       Item list * (* list of unnamed items (imports)* *)
       bool   (* encapsulated *)
   (* the boolean true means that FRAME is created due to encapsulated class *)

   datatype Item = VAR of Types.Var *
                  (SCode.Element*Types.Mod) option *
                   bool
            | CLASS of SCode.Class * Env
            | TYPE of Types.Type list  (* list since several types with the same
                              name can exist in the same scope (overloading) *)
            | IMPORT of Absyn.Import
```

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

```
 datatype BinTree = TREENODE of
   TreeValue option * (* Value *)
   BinTree option *   (* left subtree *)
   BinTree option     (* right subtree *)
```

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

```
   datatype TreeValue = TREEVALUE of Key *    (* Key *)
                                    Value    (* Value *)
   type Key = Ident
   type Value = Item

   val empty_env   : Env
```

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:

```
  relation new_frame : (bool) => Frame
  relation open_scope       : (Env,bool, Ident option) => Env
  relation extend_frame_c    : (Env, SCode.Class) => Env
  relation extend_frame_classes    : (Env, SCode.Program) => Env
  relation extend_frame_v    : (Env, Types.Var,(SCode.Element*Types.Mod)
option,bool) => Env
  relation update_frame_v    : (Env, Types.Var,bool) => Env
  relation extend_frame_t : (Env,Ident,Types.Type) => Env
  relation extend_frame_i : (Env, Absyn.Import) => Env
  relation top_frame : Env => Frame
  relation get_env_path: (Env) => Absyn.Path option
```

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

3.4.18 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`, `exp_contains`, `exp_equal`, `extend_cref`, etc. perform this functionality, e.g.:

```
extend_cref (ComponentRef, Ident, Subscript list) => 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:

```
datatype Type = INT | REAL | BOOL | STRING | ENUM | OTHER
```

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:

```
datatype Exp =

  ICONST of int                       (* Integer constants *)
| RCONST of real                      (* Real constants *)
| SCONST of string                    (* String constants *)
| BCONST of bool                      (* Bool constants *)
| CREF of ComponentRef * Type         (* component references, e.g. a.b[2].c[1] *)
| BINARY of Exp * Operator * Exp      (* Binary operations, e.g. a+4 *)
| UNARY of Operator * Exp             (* Unary operations, -(4*x) *)
| LBINARY of Exp * Operator * Exp     (* Logical binary operations: and, or *)
| LUNARY of Operator * Exp            (* Logical unary operations: not *)
| RELATION of Exp * Operator * Exp   (* Relation, e.g. a <= 0 *)
| IFEXP of Exp * Exp * Exp            (* If expressions *)
| CALL of Absyn.Path * Exp list * bool (* tuple *) * bool
                                     (* builtin *) (* Function call *)
| ARRAY of Type * bool * Exp list       (* Array constructor, e.g. {1,3,4} *)
| MATRIX of Type * int * (Exp*bool) list list  (* Matrix constructor. e.g.
                                                 [1,0;0,1] *)
| RANGE of Type * Exp * Exp option * Exp  (* Range constructor, e.g. 1:0.5:10 *)
| TUPLE of Exp list (*PR.*) (* Tuples, used in func calls returning several
                                                        arguments *)
(* New constructors *)
| CAST of Type * Exp                    (* Cast operator *)
| ASUB of Exp * int                     (* Array subscripts *)
| SIZE of Exp * Exp option              (* The size operator *)
| CODE of Absyn.Code * Type                    (* Modelica AST constructor *)
| REDUCTION of Absyn.Path * Exp (*expr*) * Ident * Exp (*range*)  (* Reduction
expression *)
| END                        (* array index to last element, e.g. a[end]:=1; *)
```

The `Exp` datatype closely corresponds to the `Absyn.Exp` datatype, 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.

Operators:

```
  datatype Operator =
                  ADD       of Type
                | SUB       of Type
                | MUL       of Type
                | DIV       of Type
                | POW       of Type
                | UMINUS    of Type
                | UPLUS     of Type
                | UMINUS_ARR  of Type
                | UPLUS_ARR   of Type
                | ADD_ARR     of Type
                | SUB_ARR     of Type
                | MUL_SCALAR_ARRAY of Type   (* a * { b, c }            *)
```

```
                    | MUL_ARRAY_SCALAR of Type   (* {a, b} * c              *)
                    | MUL_SCALAR_PRODUCT of Type (* {a, b} * {c, d}         *)
                    | MUL_MATRIX_PRODUCT of Type (* {{..},..} * {{..},{..}} *)
                    | DIV_ARRAY_SCALAR of Type   (* {a, b} / c              *)
                    | POW_ARR     of Type
                    | AND | OR
                    | NOT
                    | LESS        of Type
                    | LESSEQ      of Type
                    | GREATER     of Type
                    | GREATEREQ   of Type
                    | EQUAL       of Type
                    | NEQUAL      of Type
      | USERDEFINED of Absyn.Path (* The fully qualified name of the
                                     overloaded operator function *)
```

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.

Component references:

```
datatype ComponentRef = CREF_QUAL of Ident * (Subscript list) * ComponentRef
                 | CREF_IDENT of Ident * (Subscript list)

datatype Subscript = WHOLEDIM      (* a[:,1] *)
                | SLICE of Exp     (* a[1:3,1], a[1:2:10,2] *)
                | INDEX of Exp
```

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

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

3.4.19 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.

3.4.20 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.

3.4.20.1 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.

3.4.20.2 Code Instantiation of a Class in an Environment

(?? Add more explanations)

Relation: `inst_classdef`
PARTS: `inst_element_list`
DERIVED (i.e `class A=B(mod);`):

1. `lookup` class
2. `elab_mod`
3. Merge modifications
4. `inst_class_in` (…,mod, …)

3.4.20.3 Inst_element_list & Removing Declare Before Use

The procedure is as follows:

1. First implicitly declare all local classes and add component names (calling extend_components_to_env), 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)).

3.4.20.4 The Inst_element Function

This is a huge relation 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).
- `Inst_var` called (?? explain).
- ClassDefs (?? explain).

3.4.20.5 The Inst_var Function

The `inst_var` 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.

3.4.20.6 Dependencies

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

3.4.21 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:

```
datatype InteractiveSymbolTable
  = SYMBOLTABLE of
    Absyn.Program *                (* The ast *)
    SCode.Program *                (* The exploded ast *)
    InstantiatedClass list *       (* List of instantiated classes*)
    InteractiveVariable list *     (* List of variables with values*)
    (Absyn.Path * Types.Type) list (* List of compiled functions,
                                      fully qualified name + type *)
datatype InteractiveStmt
  = IALG of Absyn.AlgorithmItem
  | IEXP of Absyn.Exp

datatype InteractiveStmts
  = ISTMTS of InteractiveStmt list *
             bool (* true if output result == no semicolon*)

datatype InstantiatedClass
  = INSTCLASS of Absyn.Path * (* The fully qualifiedname of the inst:ed class*)
    DAE.Element list *        (* The list of DAE elements *)
    Env.Env                   (* The env of the inst:ed class*)

datatype InteractiveVariable
  = IVAR of Absyn.Ident *  (* The variable identifier *)
    Values.Value *         (* The expression containing the value *)
    Types.Type             (* The type of the expression *)
```

Two of the more important functions:

```
relation evaluate: (InteractiveStmts, InteractiveSymbolTable) =>
                                            (string,InteractiveSymbolTable)
```

```
relation update_program: (Absyn.Program,Absyn.Program) => Absyn.Program
```

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

3.4.22 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:

- `lookup_class` – to find a class.
- `lookup_type` – to find types (e.g. functions, types, etc.).
- `lookup_var` – to find a variable in the instance hierarchy.

Concerning builtin types and operators:

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

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/explode.

3.4.23 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/explode, DAE, DAElow, Inst, Interactive, Rtopts, Debug, Codegen, Socket, Print, Corba, System, Util, SimCodegen.

Optional dependencies for parallel code generation: ??

3.4.24 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:

- `elab_mod(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/explode, Types, Dump, Debug, Print, Inst, Static, Values, Util.

3.4.25 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.

3.4.26 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.

3.4.27 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.

3.4.28 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..

3.4.29 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.

3.4.30 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.

3.4.31 SCode – File explode.rml – 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 Relations
  - `elaborate (Absyn.Program) => Program`
  - `elab_class: Absyn.Class => Class`
  - `build_mod (Absyn.Modification option, bool) => Mod`

(?? The module is called SCode whereas the file is called explode, i.e., inconsistent naming)

Module dependencies: Absyn, Dump, Debug, Print.

3.4.32 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.

3.4.33 Socket – (Depreciated) OpenModelica Socket Communication Module

This module is being 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.

3.4.34 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 `elab_builtin_handler`. 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 `elab_exp`, which calls `elab_call_interactive`.

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:

- `elab_exp:  Absyn.Exp => Exp.Exp * Types.Properties` – Static analysis, finding out properties.
- `elab_graphics_exp` – for graphics annotations.
- `elab_cref` – check component type, constant binding.
- `elab_subscripts`: `Absyn.Subscript` => `Exp.Subscript` – Determine whether subscripts are constant

Constant propagation

- `ceval`

The elab_exp 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 `canon_cref` function:

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

The `elab_builtin_handler` function:

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

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

3.4.35 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 RML.

Module dependencies: Values.

3.4.36 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.

3.4.37 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.

3.4.38 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 RML 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:

```
datatype Var = VAR of Ident             (* name *)
                * Attributes           (* attributes *)
                * bool                  (* protected *)
                * Type                 (* type *)
                * Binding              (* equation modification *)

datatype Attributes = ATTR of bool   (* flow *)
                       * SCode.Accessibility
                       * SCode.Variability  (* parameter *)
                       * Absyn.Direction

datatype Binding = UNBOUND
                | EQBOUND of Exp.Exp * bool (* bool true for constant *)
                | VALBOUND of Values.Value
```

Types:

```
type Type = (TType * Absyn.Path option)

datatype TType = T_INTEGER of Var list
            | T_REAL of Var list
            | T_STRING of Var list
            | T_BOOL of Var list
            | T_ENUM
            | T_ENUMERATION of string list * Var list
            | T_ARRAY of ArrayDim * Type
            | T_COMPLEX of ClassInf.State
                      * Var list
            | T_FUNCTION of FuncArg list
                      * Type (* Only single-result *)
            | T_TUPLE of Type list (* Used by functions who return multiple
values. *)
            | T_NOTYPE

datatype ArrayDim = DIM of int option
type FuncArg = Ident * Type
```

Expression properties:

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

Used by `split_props`:

```
datatype Const = CONST of bool |
              TUPLE_CONST  of Const list

datatype Properties = PROP of Type * (* type *)
                         bool (* if the type is a tuple, each element
                                  have a const flag. *)

(* Type is meant to be T_TUPLE *)
                   | PROP_TUPLE of Type * Const (* The elements might be
                                                    tuple themselves *)

 (* Used for multiple return arguments from functions,
  *  one constant flag for each return argument.
  *)
```

The datatype `Properties` contains information about an expression. The properties are created by analyzing the expressions. (?? Where is this datatype?)

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, was originally in Mod:

```
datatype EqMod = TYPED of Exp.Exp * Properties |
             UNTYPED of Absyn.Exp
datatype SubMod = NAMEMOD of Ident * Mod
            | IDXMOD of int list * Mod
and Mod = MOD of bool * (SubMod list) * EqMod option
      | REDECL of bool * (SCode.Element*Mod) list
      | NOMOD
```

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

3.4.39 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 RML 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 RML 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.

3.4.40 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.

```
datatype Value = INTEGER of int
             | REAL of real
             | STRING of string
             | BOOL of bool
             | ENUM of string
             | ARRAY of Value list
             | TUPLE of Value list
             | RECORD of Value list * Exp.Ident list
             | CODE of Absyn.Code
             (* A record consist of value * Ident pairs *)
```

Module dependencies: Absyn, Exp.

3.4.41 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.

Index

Error! No index entries found.