Using Modelica under Scilab/Scicos

Sébastien FURIC
Imagine

Agenda

- Overview of the Modelica language
  - Basic concepts
  - Building models using Modelica
- Modelicac, a Modelica compiler
  - Overview
  - Generating C code from a Modelica specification using Modelicac

Overview of the Modelica language

Basic concepts

Structuring knowledge

- Modelica enables the creation of:
  - Structured types
  - Connectors
  - Blocks
  - Models
  - Functions
  - Packages

Basic language elements

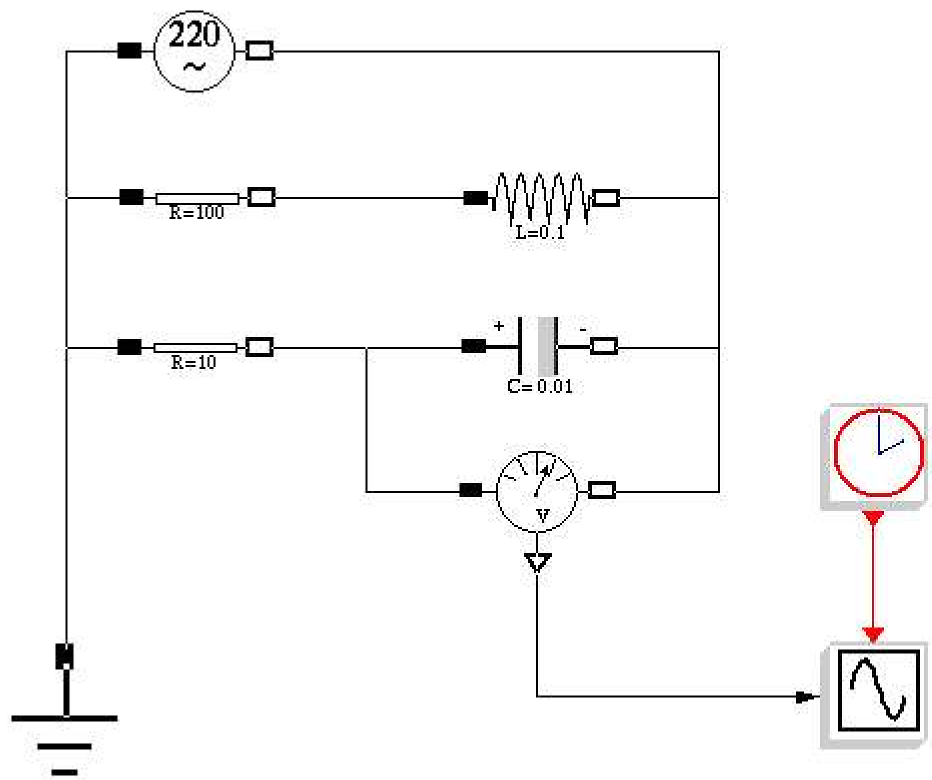
- Basic types (Boolean, Integer, Real and String)
- Enumerations
- Compound classes
- Arrays
- Equations and/or algorithms
- Connections
- Functions

Data abstraction

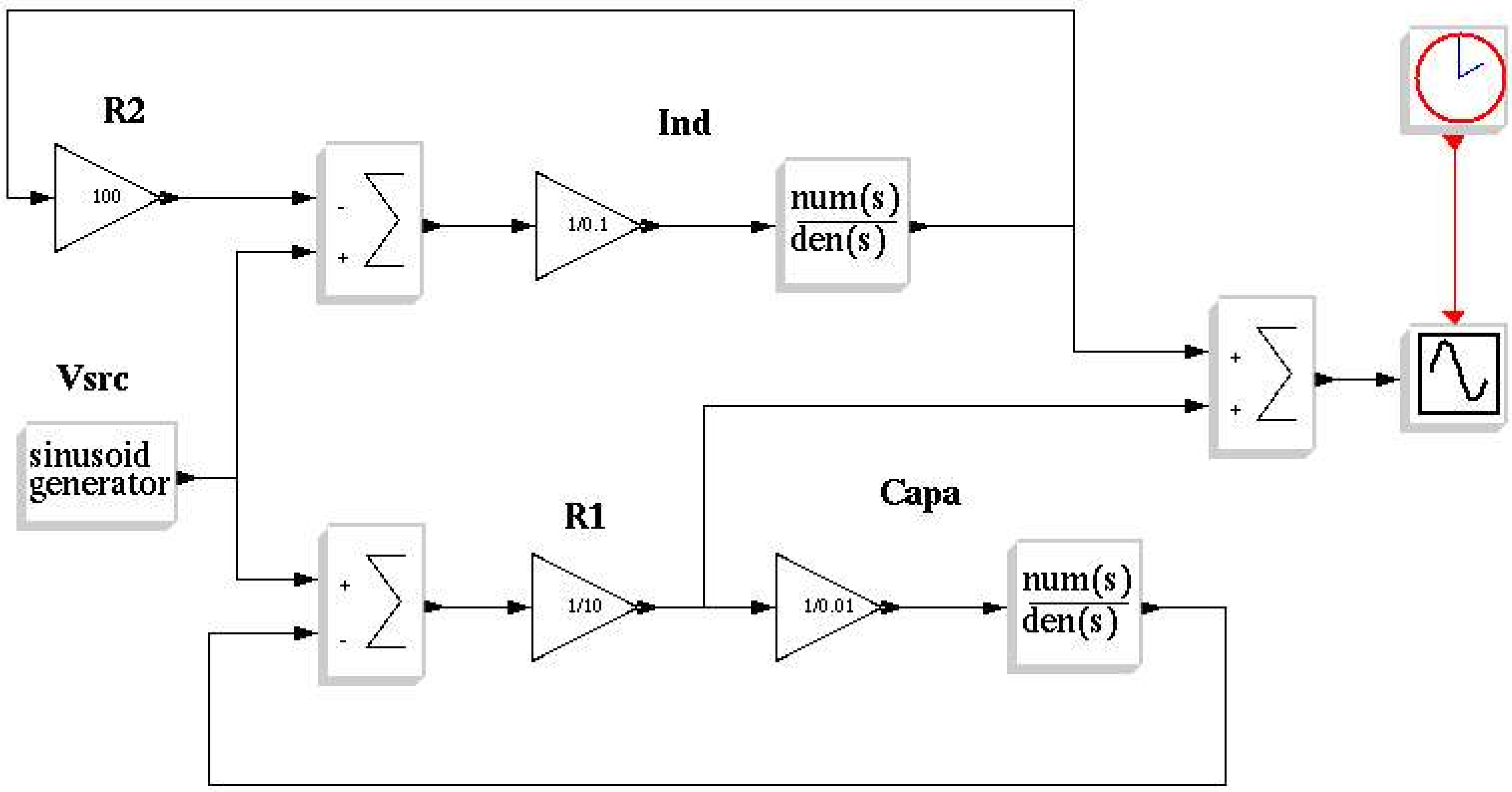
- Packages, models, functions etc. are all described using classes
  - Classes are the only way to build abstractions in Modelica
  - Classes enable structured modelling
  - Classes offer an elegant way of classifying manipulated entities that share common properties (nested sets)

Example of a simple model

```
class imppart_circuit
  Ground          Grnd;
  VsourceAC       VSrc(VA=220, f=50);
  Resistor        R1(R=100);
  Resistor        R2(R=10);
  Inductor        Ind(L=0.1);
  Capacitor       Capa(C=0.01);
  VoltageSensor   Vsnsr;
  OutPutPort      Out;
equation
  connect (Ind.n,VSrc.n);
  connect (Capa.n,VSrc.n);
  connect (Vsnsr.n,VSrc.n);
  connect (Capa.p,R2.n);
  connect (Vsnsr.p,R2.n);
  connect (R1.p,VSrc.p);
  connect (R2.p,VSrc.p);
  connect (Grnd.p,VSrc.p);
  connect (Ind.p,R1.n);
  Vsnsr.v = Out.vi;
end imppart_circuit;
```



Example of a complicated model



Class description

- A class is composed of three kinds of sections:
  - Element declaration sections
  - Equation and/or algorithm clause sections
  - External function call sections

Restricted classes

- Restricted classes can be defined in Modelica by replacing the keyword “class” by one the following ones: “record”, “connector”, “model”, “block”, “type”, “package”, “function”
- Restricted classes allow library designers to enforce the intended use of a given class

Element declaration section

- Elements include:
  - Local components and/or local classes (named elements)
  - Imports (to allow components and local classes of another class to be in scope)
  - “Extends clauses” (Modelica's inheritance mechanism)

Visibility modifiers

- By default, a declared named element is public (i.e., it can be accessed from the outside using dot notation)
- Protected named elements can be declared using the prefix “protected”

Scoping rules

- Local classes declarations form an ordered set of lexically enclosing parents
- Apart from its own named elements, a class may only access names of constants and local class definitions in the enclosing set of parents
- By default, name lookup is static

Static lookup of simple names

- Simple names (no dots) are lookup as follows:
  - In the sequence of control variable names of enclosing “for” constructs
  - In the locally defined components and classes (including inherited ones)
  - In the import statements (qualified ones first, then unqualified ones)
  - In the sequence of enclosing parents until the current class is encapsulated
  - In the unnamed toplevel class

Static lookup of composite names

- Composite names (of the form A.B, A.B.C, etc.) are looked up as follows:
  - A is looked up as any other simple name
  - B, C, etc. are looked up among the public declared named elements of the denoted element (including inherited ones). If an element denotes a class, that class is temporarily instantiated and lookup is performed in the temporary instance

Static name lookup example

```
class Foo

  constant Real pi=3.1416;

  Real x;

  Bar b;

  class Bar
    Real y=cos(2*pi*time);
  end Bar;

  class Baz
    constant Real e=2.71828;
  end Baz;

  import Modelica.Math.*;

equation
  Baz.e*x = b.y;

end Foo;
```

Dynamic lookup of names

- A named element declared with the prefix “outer” references an element of the enclosing set of instances that has the same name and is declared with the prefix “inner”

Dynamic name lookup example

```
class BarType
  Real y;
end BarType;

class Foo
  inner Real pi=3.1416;
  inner class Bar
    Real y;
  end Bar;
  Baz b;
end Foo;

class Baz
  outer Real pi;
  outer class Bar = BarType;
  Bar b;
equation
  Modelica.Math.cos(2*pi*time) = b.y
end Baz;
```

Order of declarations

- The order of declaration of elements does not matter (i.e., it is possible to use a variable before declaring it, provided a declaration exists in the scope)
  - Modelica was designed with ease of code generation in mind (a graphical tool is not supposed to sort elements before generating code)

Component declaration specification

- A component declaration is composed of:
  - An optional type prefix
  - A type specifier
  - An optional array dimension specification
  - An identifier
  - An optional set of modifications
  - An optional comment

Examples of component declarations

- To declare a constant:

```
constant Real pi=3.141592654;
```

- To declare an array of 10 Resistors, each internal R=100 Ohms:

```
Resistor[10] Rs ”my array of resistors”;

Resistor Rs[10];
```

- To declare an input vector of flow Real (i.e., floating point) numbers:

```
flow input Real[:] Is;
```

Type prefix

- Three kinds of type prefixes:
  - “flow” prefix (indicating a flow variable when set and a potential variable otherwise)
  - Variability prefix (one of “constant”, “parameter” or “discrete” in the case of a non-continuous variable)
  - Causality prefix (“input” or “output”, to force causality, for instance in case of a function formal parameter)

Component modifications

- Two kinds of modifications:
  - Value modifications (mainly used to give values to parameters)
  - Structural (type) modifications (used to refine an existing class definition, either by restricting a type or by replacing some named elements)

Initial values of variables

- Variables of predefined types can be given initial values using modifications:

```
Real x(start=0.0); /* just a guess */

Real x(start=0.0, fixed=true); /* we want
  x to start at 0.0 */
```

- Another way to initialize variables is to use “initial equations”

Class inheritance

- Introduced by the “extends” keyword
- Inheritance is used to:
  - Create new classes by extending several existing ones (i.e., merging contents of several classes) before eventually adding new sections
  - Modifying an existing class using class modifications

Class inheritance example

```
class Bar
  Real x=1;
end Bar;

class Baz
  Real y;
end Baz;

class Foo
  extends Bar;
  Real z=3;
  extends Baz(y=2);
end Foo;

Foo my_foo;
/* my_foo has 3 internal variables: x, y and z
   whose values are 1, 2 and 3 respectively    */
```

Replaceable elements

- Named elements may be declared as “replaceable”:
  - These elements may be replaced by new ones in structural modifications, provided type compatibility constraints to be verified
  - Allow a flexible model parametrization (parametric polymorphism)

Example of element replacement

```
class ElectricalMotor
  replaceable IdealResistor R(R=100);
...
end ElectricalMotor;

class Circuit
  ElectricalMotor m(redeclare MyResistorModel R);
...
end Circuit;
```

Partial classes

- Some classes are said to be "partial" if they are declared under the heading "partial"
- A partial class can not be instantiated
- Partial classes are used to provide a framework to develop models according to a given interface

Example of a partial class

```
partial class TwoPin
  Pin p, n;
  Real v, i;
equation
  i = p.i;
  i = -n.i;
  v = p.v - n.v;
end TwoPin;

class Resistor
  extends TwoPin;
  parameter Real R;
equation
  v = R * i;
end Resistor;
```

Equation clauses

- Equation clauses are used to describe the set of constraints that apply to a model
- Constraints can apply either at initialization time (initial equations) or at simulation time (ordinary equations)

Examples of equation clauses

```
class Resistor
  parameter Real R;
  Pin p, n;
  Real i;
  Real v;
equation
  p.v – n.v = v;
  p.i = i;
  n.i = -p.i;
  v = R * i;
end Resistor;

class Circuit
  Resistor R(R=100);
  VsourceAC Src;
  ...
initial equation
  Src.v = 0;
equation
  connect(R.p, Src.p);
  ...
end Circuit;
```

Comments about equation clauses

- Equation clauses are not sequences of statements! (in particular, there is no notion of assignment, nor evaluation order)
- It is however possible to describe how to compute a result by means of sequences of assignments, loops, etc. in Modelica, but not using equations!

Different kinds of equations (1)

- Equality between two expressions:

```
v = R * i;
```

- Conditional equation:

```
if mode == Modes.basic then
  x = basicControl.c;
else
  x = complexControl.c;
end if;
```

- “For” equation:

```
for k in 1 : n loop
  v[k] = R[k] * i[k];
end for;
```

Different kinds of equations (2)

- “Connect” equation:

```
connect(R.p, Src.p);
```

- “When” equation:

```
when x <= 0.0 then
  reinit(a, -a);
  reinit(v, 0);
end when;
```

- “Function call”:

```
assert(n > 0, “Model is not valid”);
```

Expressions

- Modelica provides the necessary functionalities to express:
  - The usual “mathematical” functions (sin(), cos(), exp(), etc.)
  - The derivative of a variable
  - Conditional expressions
  - “Event-free” expressions
  - Multi-dimensional arrays and associated operations

Variability of expresions

- Varibility modifiers in declarations:
  - “constant”
  - “parameter”
  - “discrete”
- Discrete variables and ordinary variables only may change their values during simulation time (discrete variables are only modified inside “when” equations)

Examples of equations

- Algebraic equation:

```
v = R * i;
i = v / R; // a “less general” formulation
```

- Differential equation:

```
a = -g;
der(v) = a;
der(x) = v; // der(der(x)) = a is illegal!
```

- Conditional expression in equation:

```
y = if x > x0 then exp(x0) else exp(x);
y = if noEvent(x > x0) then exp(x0) else
  exp(x); // the correct version
```

Algorithm clauses

- Algorithm clauses are sequences of assignments and control structures statements
- Algorithm clauses are used to describe <u>how</u> a quantity has to be computed
- Like equation clauses, algorithm clauses may apply either at initialization time or at simulation time
-

Different kinds of statements(1)

- Assignment:

```
y := 2 * x;
```

- “If” statement:

```
if x >= 0.0 then
  y := x;
else
  y := -x;
end if;
```

- “For” statement:

```
for i in 1 : n loop
  y[i] := 2 * x[i];
end for;
```

Different kinds of statements(2)

- "While" statement:

```
while abs(x – y) > eps loop
  x := y;
  y := x – f(x) / fdot(x);
end while;
```

- "When" statement:

```
when x == 0.0 then
  y := 0;
end when;
```

- Continuation statements:

```
return;
break;
```

Examples of an algorithm clause

```
block Fib
  input Integer n;
  protected Integer p, q:=1;
  public output Integer f:=1;
algorithm
  assert(n > 0, "Argument must be strictly positive");
  for i in 1 : n loop
    f := p + q;
    p := q;
    q := f;
  end for;
end Fib;
```

External function calls

- Modelica allows the user to call external functions written in a foreign language (only C and FORTRAN are currently supported)
- Modelica provides the necessary framework to handle formal parameters and multiple return values

Restrictions over external functions

- External functions must be “pure”, in the sense that they should not attempt to alter any variable that is not declared as “output” in the calling Modelica code
- Also, external functions must return the same values given the same arguments (referential transparency property)
-

Exemple of external function

```
function Foo
  input Real x[:];
  input Real y[size(x,1),:];
  input Integer i;
  output Real u1[size(y,1)];
  output Integer u2[size(y,2)];
external "FORTRAN 77"
  myfoo(x, y, size(x,1), size(y,2), u1, i, u2);
end foo;
```

References

- Modelica's official WEB site:
  - http://www.modelica.org
- Books:
  - “Introduction to Physical Modeling with Modelica”, by M. Tiller
  - “Principles of Object-Oriented Modeling and Simulation with Modelica 2.1”, by P. Fritzson

Overview of the Modelica language

Building models using Modelica

Notion of package

- A package is a hierarchical set of Modelica classes and constant components
- Packages may be stored:
  - As nested modelica classes, in a single file
  - In the host file system, as a tree of directories and files

Contents of a package

- Packages are generally divided into subpackages corresponding to a discipline (library)
- The default Modelica package contains the definition of:
  - physical quantities and constants
  - Useful connectors, blocks and models (electrical domain, mechanical domain, etc.)
  - Many more...

Overview of a Modelica library

- A library usually provides several subpackages containing:
  - The public types used in the library
  - Eventually, some useful functions
  - The connectors used to build classes
  - Interfaces of classes
  - Instantiable classes
  - Some test models

Example of a Modelica library

```
package MyElectricalLibrary

  package Types
    type Voltage = Real(unit="v");
    type Current = flow Real(unit="A");
  end Types;

  package Connectors
    connector Pin
      Voltage v;
      Current i;
    end Pin;
  end Connectors;

  package Interfaces
    partial model TwoPin
      ...
    end TwoPin;
    ...
  end Interfaces;

  ...

end MyElectricalLibrary;
```

Building models

- To build models, one has to proceed the following steps:
  - Define the types attached to the discipline
  - Define connectors
  - Build library models
  - Build “main” models (i.e., models that can be simulated)

Model building example

```
type Voltage = Real(unit=”v”);
type Current = flow Real(unit=”A”);

connector Pin
  Voltage v;
  Current i;
end Pin;

model Resistor ... end Resistor;

model Capacitor ... end Capacitor;

...

model Circuit
  Resistor R1(R=100);
  Capacitor C1(C=0.001);
  ...
equation
  connect(R1.p, C1.n);
  ...
end Circuit;
```

Modelicac, a Modelica compiler

Overview

History

- 1998~2001: SimLab project (EDF R&D and TNI)
  - Causality analysis problem
  - Symbolic manipulation of mathematical expressions
- 2001~2004: SimPA Project (TNI, INRIA, EDF, IFP and Cril Technology)
  - Compilation of Modelica models
  - Automatic event handling, DAE solvers
  - Ehancements of Scicos's editor

Inside Modelicac

- Modelicac is composed of three main modules:
  - Modelica parsing and compilation
  - Symbolic manipulation
  - Code generation

Compilation modes

- Modelicac can be used for two different purposes:
  - Compiling connectors and library models as “object files”
  - Generating code (usually C code) for the target simulation environment (for instance, Scicos)

The compiled modelica subset: class definitions

- Only two kinds of classes are currently supported
  - “class”: to describe connectors and models
  - “function”: to describe external functions
- “encapsulated”, “final” and “partial” are not supported

The compiled modelica subset: definition of elements

- An element can be either an instance of another class or an instance of the primitive type “Real”
- Only “value” modifications are supported
- Local classes, imports and extensions are not currently supported

The compiled modelica subset: equations and algorithms

- Initial equations are not supported (use modifications instead)
- Equations defined by equality, “for” equations and “when” equations are supported
- Currently, it is possible to use neither “if” equations nor algorithm clauses

The compiled modelica subset: external functions

- Only “Real” scalars can currently be passed as arguments to functions
- Functions only return one “Real” scalar result
- The foreign language is supposed to be C

Model transformation

- Before generating code for the target, Modelicac performs the following tasks:
  - Building an internal flat model representing the model to simulate
  - Performing some symbolic simplifications (elimation of the linearities, inversions of some bijective functions)
  - Eventually, computing the analytic jacobian matrix of the system

Modelica library files

- Modelicac requires a file to contain exactly one Modelica class definition
- The name of the file is the same as the name of the defined class, followed by the suffix ".mo"

Compiling library models

- Modelicac has to be invoked with the “-c” option:
  - Modelicac -c <model.mo>
- Modelicac generates a file named “<model>.moc” that contains binary object code
- No link is done at that stage, names of external classes are only looked up when flattening a complete model

Writing a “main” model

- The difference between a “main” model and a library model is that “main” models are required to be well constrained
- Usually, writing main models is not done by the user: graphical simulation environments (like Scicos) can do it automatically

Compiling a “main” model

- By default, Modelicac considers the file passed as argument to contain a “main” model:

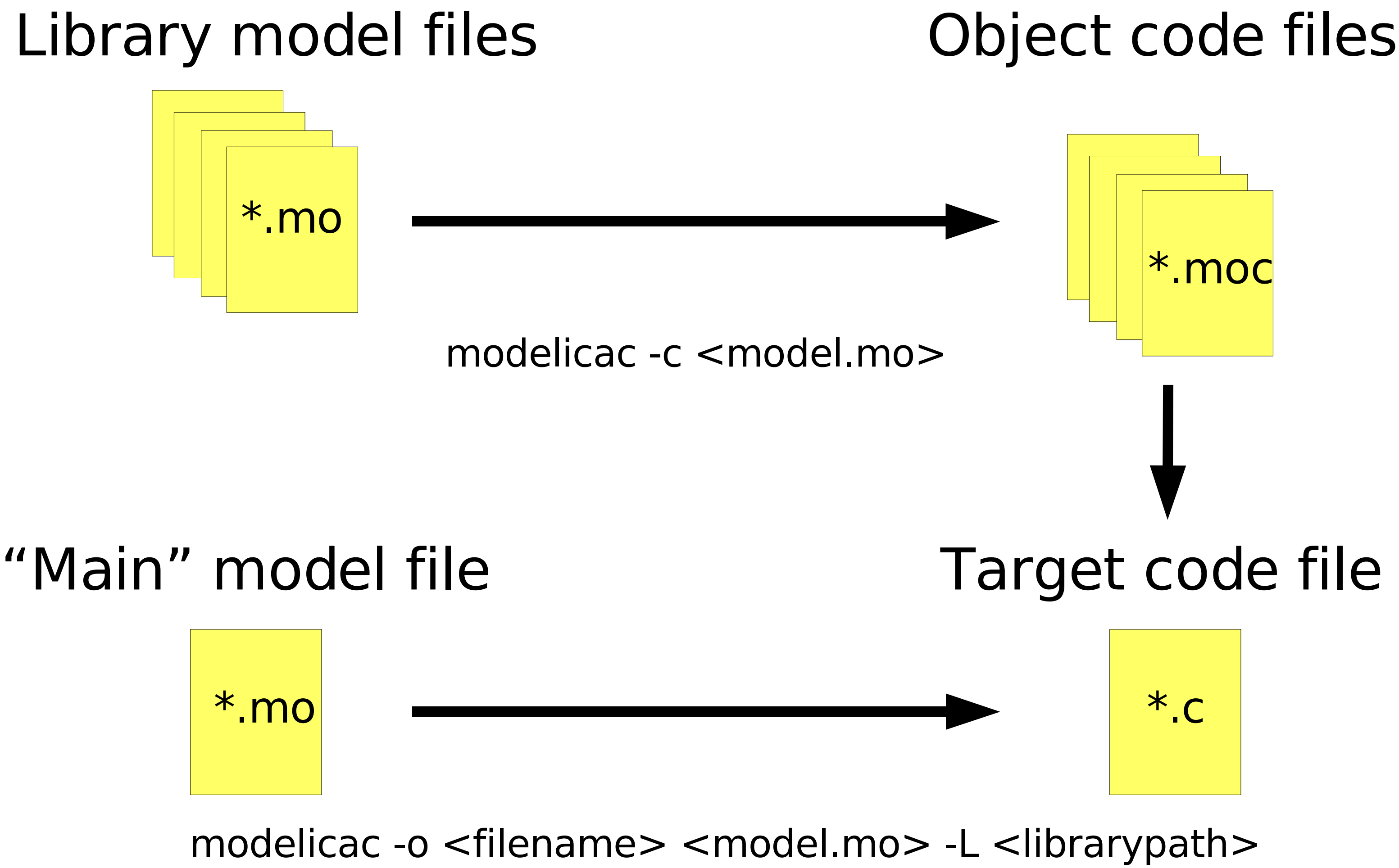
  modelicac <model.mo>

- Additional command line arguments can be passed to Modelicac, for instance:
  - -o <filename>: to indicate the name of the file to be generated
  - -L <library_path>: to indicate where to find object files to link to the model

The C code generated by Modelicac for the Scicos target

- Modelicac generates a file containing a C function that is compiled and linked against Scilab before simulation takes place
- The generated C code is in fact the code of an ordinary external Scicos block

Compilation process



Modelicac, a Modelica compiler

Generating C code from a Modelica specification using Modelicac

Building an electrical library(1)

- Defining connectors

```
class Pin
  Real v;
  flow Real i;
end Pin;
```

Building an electrical library(2)

- Defining a class of resistors

```
class Resistor
  Pin p, n;
  Real v, i;
  parameter Real R “Resistance”;
equation
  v = p.v – n.v;
  i = p.i;
  i = -n.i;
  v = R * i;
end Resistor;
```

Building an electrical library(3)

- Defining a class of capacitors

```
class Capacitor
  Pin p, n;
  Real v;
  parameter Real C “Capacitance”;
equation
  v = p.v – n.v;
  0 = p.i + n.i;
  C * der(v) = p.i;
end Capacitor;
```

Building an electrical library(4)

- Defining a class of inductors

```
class Inductor
  Pin p, n;
  Real i;
  parameter Real L "Inductance";
equation
  L * der(i) = p.v – n.v;
  0 = p.i + n.i;
  i = p.i;
end Inductor;
```

Building an electrical library(5)

- Defining a class of AC voltage sources

```
class VsourceAC
  Pin p, n;
  parameter Real VA = 220 "Amplitude";
  parameter Real f = 50 "Frequency"
equation
  VA*Modelica.Math.sin(2*3.14159*f*time) = p.v – n.v;
  0 = p.i + n.i;
end VsourceAC;
```

Building an electrical library(6)

- Defining a class for the ground

```
class Ground
  Pin p;
equation
  p.v = 0;
end Ground;
```

Writing a “main” model

```
class Circuit
  Resistor R1(R=100), R2(R=10);
  Capacitor C(C=0.01);
  Inductor I(L=0.1);
  VsourceAC S(V0=220.0, f=50);
  Ground G;
  output Real v;
equation
  connect(R1.p, S.p);
  connect(R1.n, I.p);
  connect(I.n, S.n);
  connect(R2.p, S.p);
  connect(R2.n, C.p);
  connect(C.n, S.n);
  connect(G.p, S.p);
  v = C.p.v - C.n.v;
end Circuit;
```

Invoking Modelicac (1)

- Compiling the library models is done by entering the following commands:

```
modelicac -c Pin.mo
modelicac -c VsourceAC.mo
modelicac -c Ground.mo
modelicac -c Resistor.mo
modelicac -c Capacitor.mo
modelicac -c Inductor.mo
```

Invoking Modelicac (2)

- Finaly, to compile the “main” model, enter:

  modelicac -o Circuit.c Circuit.mo

Writing an external function(1)

- The prototype of the external function is an ordinary C “header file”:

```
#include <math.h>

float Sine(float);
```

Writing an external function(2)

- The C code of the external function:

```
#include "Sine.h"

float Sine(float u)
{
  float y;
  y = sin(u);
  return y;
}
```

Writing an external function(3)

- The Modelica code of the external function:

```
function Sine
  input Real u;
  output Real y;
external;
end Sine;
```

Compiling an external function

- External functions are compiled like any ordinary library model:

  modelicac -c <functionname.mo>

- By default, Modelicac assumes a C header file (with the same base name) to be present in the compilation directory
- Additional paths can be indicated using the “-hpath” option

Calling an external function from a Modelica model

- The VsourceAC model, rewritten to call an external version of the sine function:

```
class VsourceAC
  Pin p, n;
  Real v;
  parameter Real V0 "Amplitude";
  parameter Real f "Frequency";
  parameter Real phi "Phase angle";
equation
  V0 * Sine(6.2832 * f * time + phi) = v;
  v = p.v - n.v;
  0 = p.i + n.i;
end VsourceAC;
```

Generated C code

```
...
if (flag == 0) {
    v0 = sin(314.16*get_scicos_time());
    res[0] = 0.01*xd[0]+0.1*x[0]-22.0*v0;
    res[1] = 0.1*xd[1]+100.0*x[1]-220.0*v0;
}
...
```