

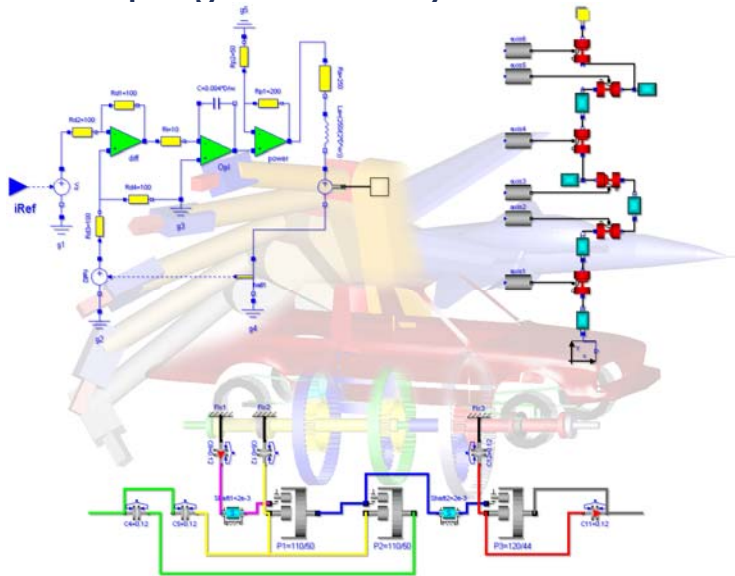
Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages

Adrian Pop

Programming Environment Laboratory

Department of Computer and Information Science

Linköping University



A Servo Mechanism Model
- a micro example of a full system

1. Introduction

2. DDC Model

3. Simulation Results

4. Conclusions

$\tau_2 = \frac{1}{k_2} \tau_1$

$e = \omega_{ref} - \omega_{out}$

$u = K \left(e + \frac{1}{T_I} \int_0^t e \, dt \right)$

$v = u \quad u_R = R i \quad u_{emf} = k_1 \omega_{emf}$

$v = u$

$\theta_2 = k_2 \theta_1$

$u_L = L \frac{di}{dt}$

$u = K \left(e + \frac{1}{T_I} \int_0^t e \, dt \right)$

$e = \omega_{ref} - \omega_{out}$

$v - u_R - u_L - u_{emf} = 0$

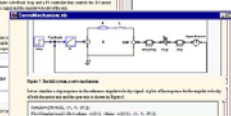
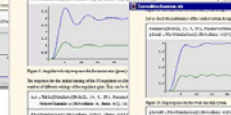
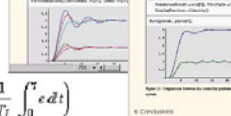
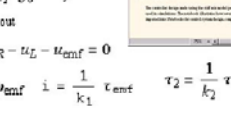
$u_{emf} = k_1 \omega_{emf} \quad i = \frac{1}{k_1} \tau_{emf} \quad \tau_2 = \frac{1}{k_2} \tau_1$

$\frac{J_1 - J_2 k_2^2}{k_2} \frac{d^2 \theta_2}{dt^2} = \tau_{emf} - k_2 \tau_3$

$J_1 \frac{d^2 \theta_1}{dt^2} = \tau_{emf} + \tau_1$

$J_2 \frac{d^2 \theta_2}{dt^2} = \tau_2 + \tau_3$

$J_3 \frac{d^2 \theta_3}{dt^2} = -\tau_4 - \tau_{load}$

2008-06-05

- Introduction
- Equation-Based Object-Oriented Languages
- The MetaModelica Language
 - Idea, Language constructs, Compiler Prototype, OpenModelica Bootstrapping
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

Current state-of-the art EOO languages are supported by tools that have fixed features and are hard to extend

The existing tools do not satisfy different user requirements

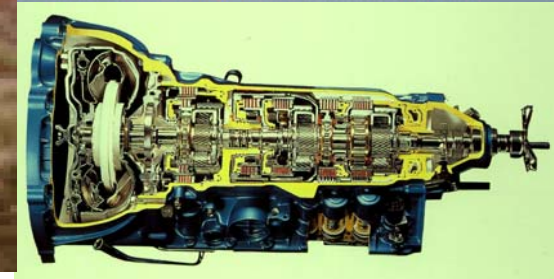
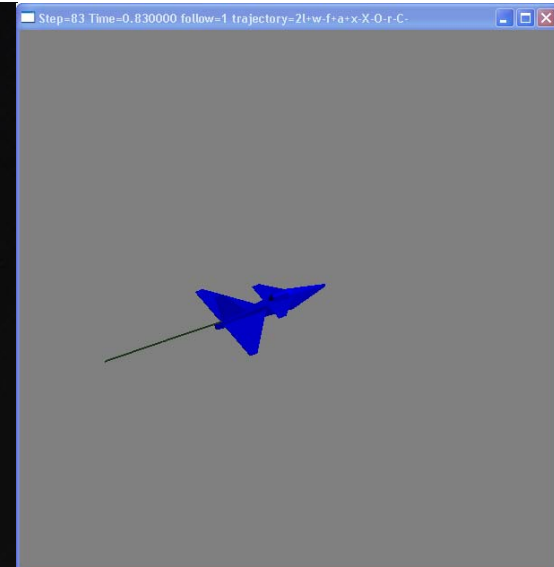
- **Management of models:** creation, query, manipulation, composition.
- **Query of model equations for:** optimization purposes, parallelization, model checking, simulation with different solvers, etc.
- **Model configuration** for simulation purposes
- **Simulation features:** running a simulation and displaying a result, running more simulations in parallel, possibility to handle simulation failures and continue the simulation on a different path, possibility to generate only specific data within a simulation, possibility to manipulate simulation data for export to another tool.
- **Model transformation and refactoring:** export to a different tool, improve the current model or library but retain the semantics, model composition and invasive model composition.

- Can we deliver a new language that allows people to build their own solution to their problems without having to go via tool vendors?
- What is expected from such a language?
- What properties should the language have based on the requirements for it? This includes language primitives, type system, semantics, etc.
- Can such a language combined with a general tool be better than a special-purpose tool?
- What are the steps to design and develop such a language?
- What methods and tools should support debugging of the new language?
- How can we construct advanced interactive development environments that support such a language?

- Introduction
- Equation-Based Object-Oriented Languages
 - The MetaModelica Language
 - Idea, Language constructs, Compiler Prototype, OpenModelica Bootstrapping
 - Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
 - Integrated Environments for Equation-Based Object-Oriented Languages
 - ModelicaML - A UML/SysML profile for Modelica
 - Conclusions and Future Work
 - Thesis Contributions

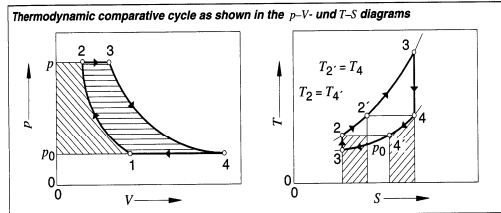
Examples of Complex Systems

- Robotics
- Automotive
- Aircrafts
- Satellites
- Biomechanics
- Power plants
- Hardware-in-the-loop, real-time simulation



Model knowledge is stored in books and human minds which computers cannot access

Internal-combustion engines 417



from T_2 to T_2' , supplied by the heat exchanger is coupled with a thermal discharge ($4 \rightarrow 4'$). If heat is completely exchanged, the quantity of heat to be added per unit of gas is reduced to

$$q_{in} = c_p \cdot (T_3 - T_2) = c_p \cdot (T_3 - T_4)$$

and the quantity of heat to be removed is

$$q_{out} = c_p \cdot (T_4 - T_1) = c_p \cdot (T_2 - T_1).$$

The maximum thermal efficiency for the gas turbine with heat exchanger is:

$$\eta_{th} = 1 - Q_{out}/Q_{in} = 1 - (T_2 - T_1)/(T_3 - T_4)$$

Where $p_2/p_1 = (T_2/T_1)^{\frac{\gamma}{\gamma-1}} = (T_3/T_4)^{\frac{\gamma}{\gamma-1}}$
and $T_4 = T_3 \cdot (T_1/T_2)^{\frac{\gamma-1}{\gamma}}$ thus

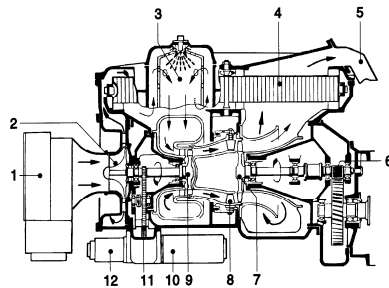
$$\eta_{th} = 1 - (T_2/T_3)$$

Current gas-turbine powerplants achieve thermal efficiencies of up to 35%.

Advantages of the gas turbine: clean exhaust without supplementary emissions-control devices; extremely smooth running; multifuel capability; good static torque curve; extended maintenance intervals.

Disadvantages: manufacturing costs still high; poor transitional response; higher fuel consumption; less suitable for low-power applications.

Gas turbine 1 Filter and silencer, 2 Radial-flow compressor, 3 Burner, 4 Heat exchanger, 5 Exhaust port, 6 Reduction gearset, 7 Power turbine, 8 Adjustable guide vanes, 9 Compressor turbine, 10 Starter, 11 Auxiliary equipment drive, 12 Lubricating oil pump.

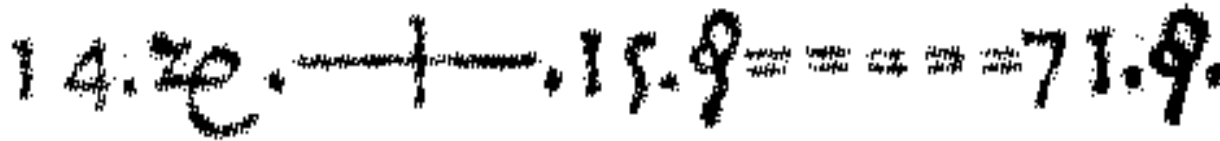


“The change of motion is proportional to the motive force impressed”
– Newton

Lex. II.

Mutationem motus proportionalem esse vi motrici impressae, & fieri secundum lineam rectam qua vis illa imprimitur.

- Equations were used in the third millennium B.C.
- Equality sign was introduced by Robert Recorde in 1557



Newton still wrote text (Principia, vol. 1, 1686)

“The change of motion is proportional to the motive force impressed”

CSSL (1967) introduced a special form of “equation”:

variable = expression

$v = \text{INTEG}(F)/m$

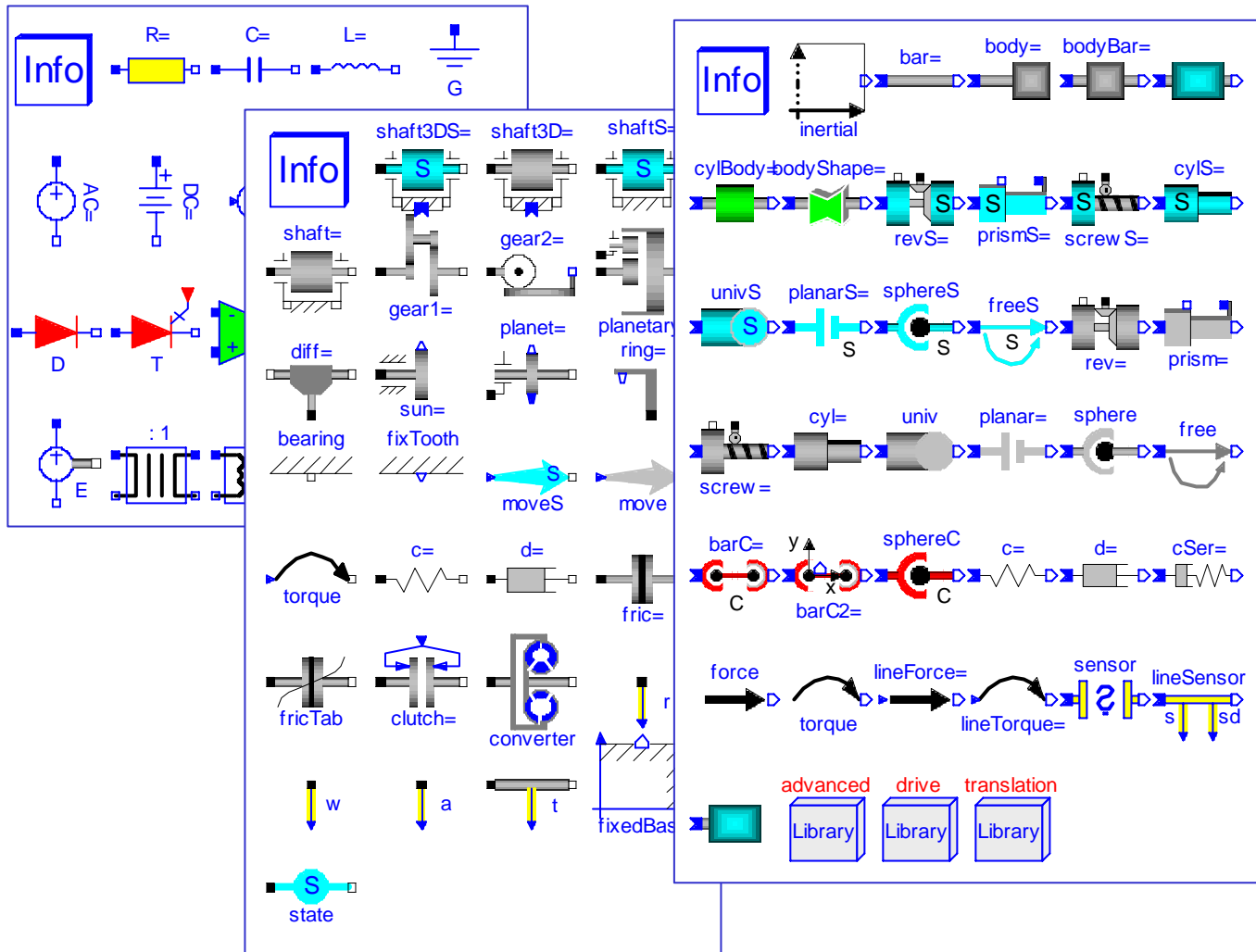
Programming languages usually do not allow equations!

- **Declarative language**
 - Equations and mathematical functions allow acausal modeling, high level specification, increased correctness
- **Multi-domain modeling**
 - Combine electrical, mechanical, thermodynamic, hydraulic, biological, control, event, real-time, etc...
- **Everything is a class**
 - Strongly typed object-oriented language with a general class concept, Java & Matlab like syntax
- **Visual component programming**
 - Hierarchical system architecture capabilities
- **Efficient, nonproprietary**
 - Efficiency comparable to C; advanced equation compilation, e.g. 300 000 equations

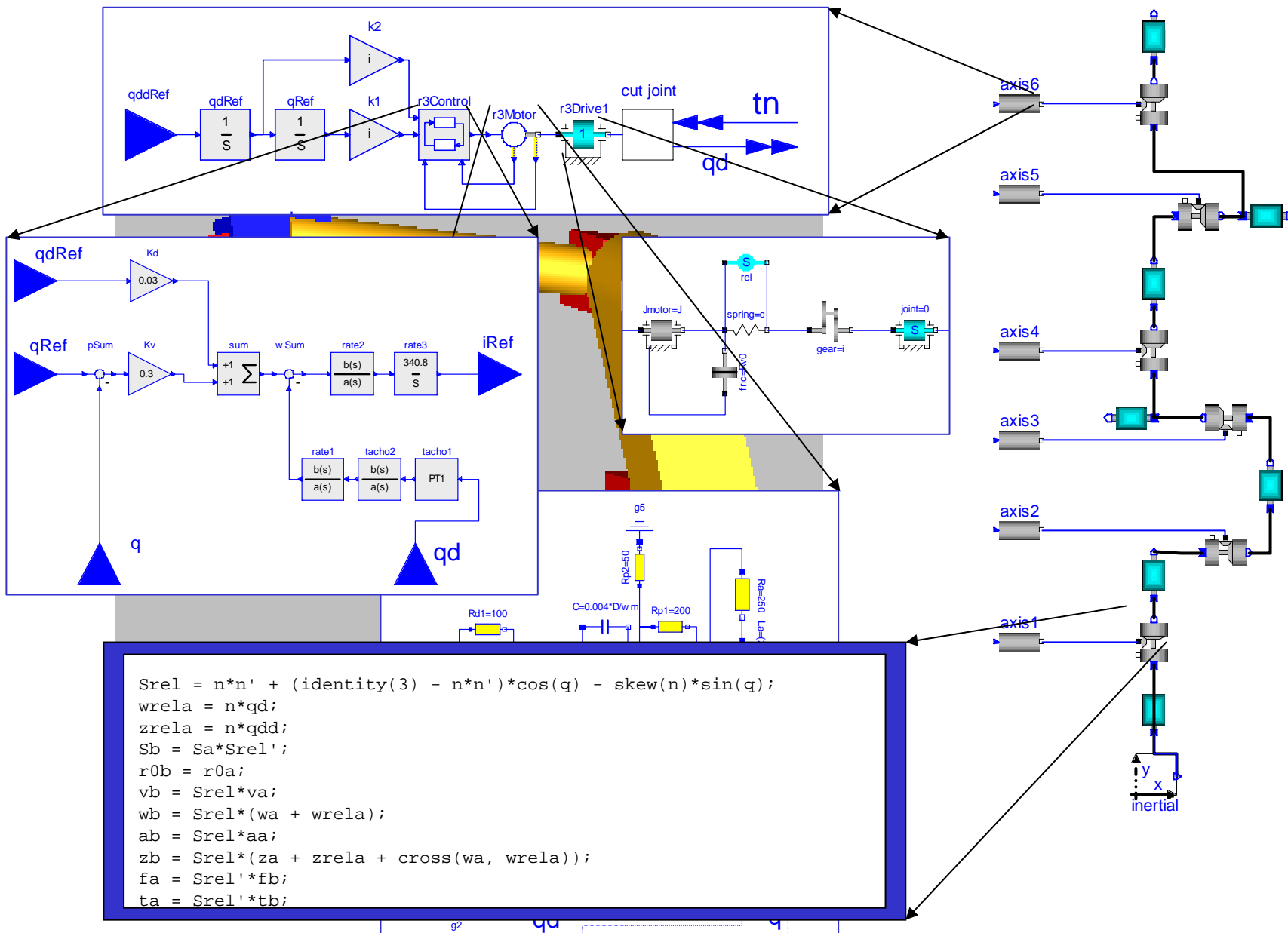
- What is *acausal* modeling/design?
- Why does it increase *reuse*?

The acausality makes Modelica library classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.
- Example: a resistor *equation*:
 $R \cdot i = v;$
- can be used in three ways:
 $i := v/R;$
 $v := R \cdot i;$
 $R := v/i;$

Modelica - Reusable Class Libraries



Hierarchical Composition Diagram



```

Srel = n*n' + (identity(3) - n*n')*cos(q) - skew(n)*sin(q);
wrela = n*qd;
zrela = n*qdd;
Sb = Sa*Srel';
r0b = r0a;
vb = Srel*va;
wb = Srel*(wa + wrela);
ab = Srel*aa;
zb = Srel*(za + zrela + cross(wa, wrela));
fa = Srel'*fb;
ta = Srel'*tb;
    
```

Multi-Domain Modelica Model - DCMotor

- A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

model DCMotor

```
Resistor R(R=100);
```

```
Inductor L(L=100);
```

```
VsourceDC DC(f=10);
```

```
Ground G;
```

```
ElectroMechanicalElement EM(k=10,J=10, b=2);
```

```
Inertia load;
```

equation

```
connect(DC.p,R.n);
```

```
connect(R.p,L.n);
```

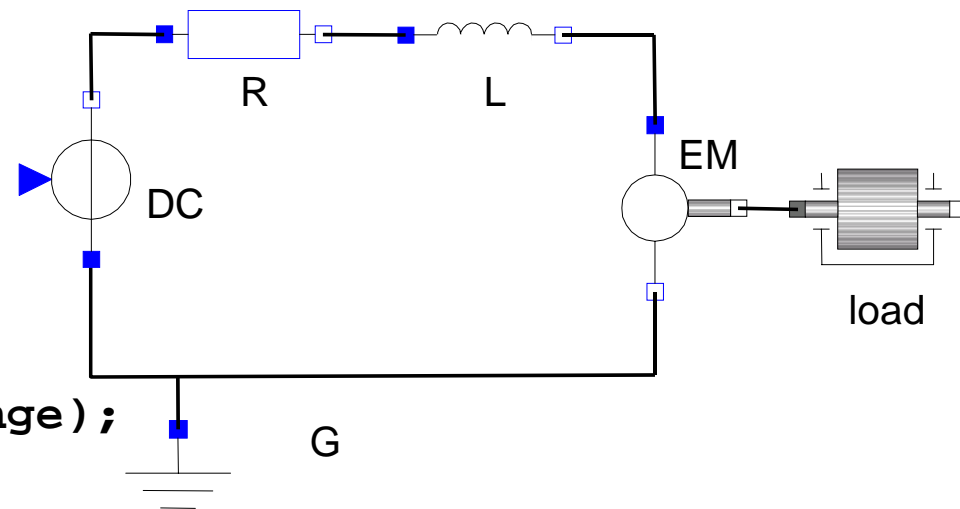
```
connect(L.p, EM.n);
```

```
connect(EM.p, DC.n);
```

```
connect(DC.n,G.p);
```

```
connect(EM.flange,load.flange);
```

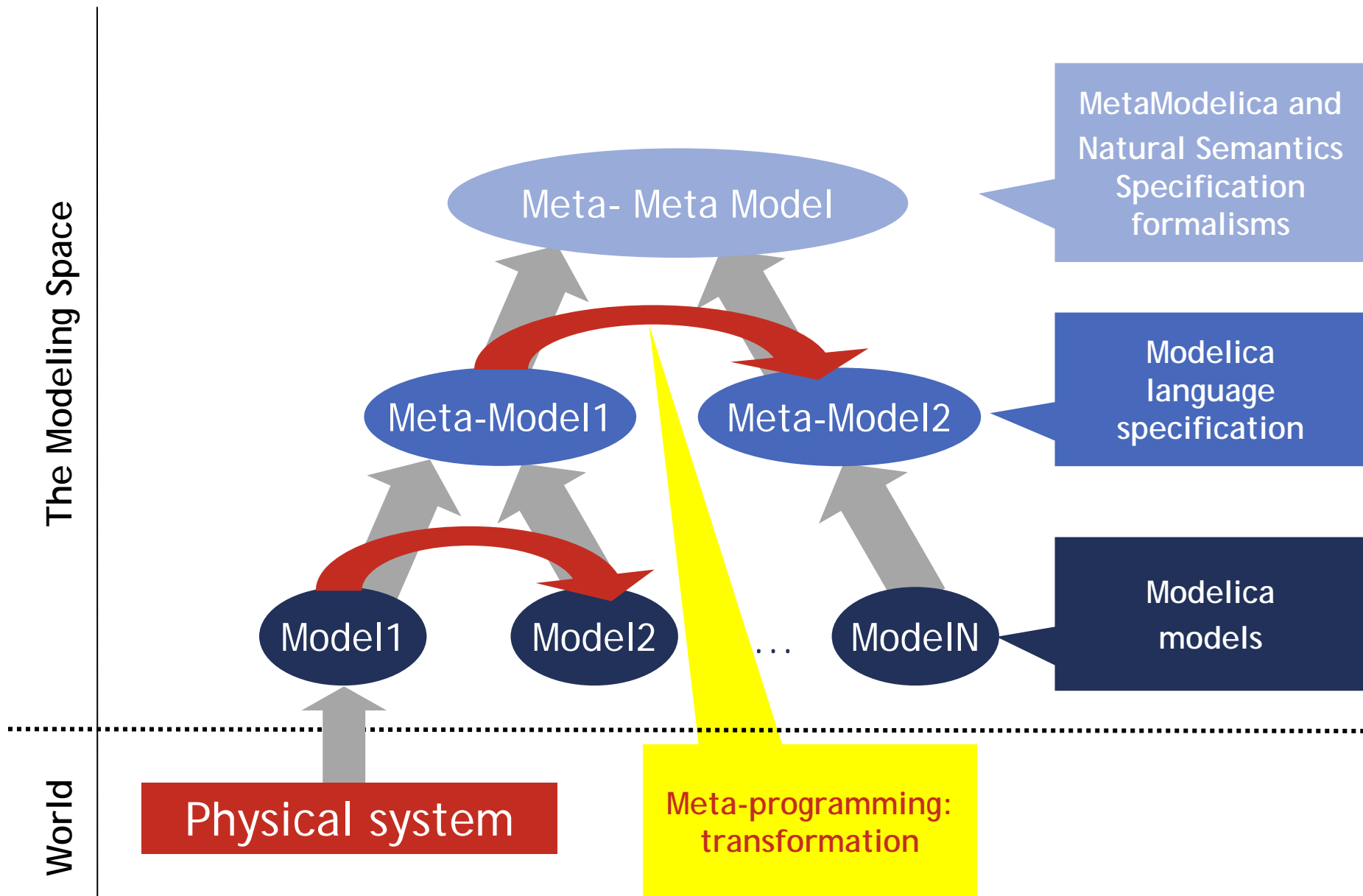
end DCMotor



- Introduction
- Equation-Based Object-Oriented Languages
- MetaModelica
 - Idea, Language constructs, Compiler Prototype, OpenModelica Bootstrapping
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

- Research Question
 - Can we deliver a new language that allows users to build their own solutions to their problems?
- Our idea - extend Modelica with support for
 - Meta-Modeling - represent models as data
 - Meta-Programming - transform or query models
- The new language - **MetaModelica**

Meta-Modeling and Meta-Programming



- Syntax - there are many efficient parser generator tools
 - lex (flex), yacc (bison), ANTLR, Coco, etc.
- *Semantics:*
 - *there are no standard efficient and easy to use compiler-compiler tools*

- Can we adapt the Modelica equation-based style to define semantics of programming languages?
 - *Answer: Yes!*
- MetaModelica Language
 - executable language specification based on
 - a model (abstract syntax tree)
 - semantic functions over the model
 - elaboration and typechecking
 - translation, transformation, querying
 - etc.

- We started from
 - The Relational Meta-Language (RML)
 - A system for building executable natural semantics specifications
 - Used to specify Java, Pascal-subset, C-subset, Mini-ML, etc.
 - The OpenModelica compiler for Modelica specified in RML
- Idea: *integrate the RML meta-modeling and meta-programming facilities within the Modelica language. The notion of equation is used as the unifying feature*

- Modelica
 - classes, models, records, functions, packages
 - behavior is defined by equations or/and functions
 - equations
 - differential equations
 - algebraic equations
 - difference equations
 - conditional equations
- MetaModelica extensions
 - local equations
 - pattern equations
 - match expressions
 - high-level data structures: lists, tuples, option and uniontypes

- pattern equations
 - unbound variables get their value by unification

```
Env.BOOLVAL(x,y) = eval_something(env, e);
```

- match expressions
 - pattern matching
 - case rules

```
pattern = match expression optional-local-declarations  
  case pattern-expression opt-local-declarations  
    optional-local-equations then value-expression;  
case ...  
...  
else optional-local-declarations  
  optional-local-equations then value-expression;  
end match;
```

```
package ExpressionEvaluator

// abstract syntax declarations
...
// semantic functions
...

end ExpressionEvaluator;
```

MetaModelica – Example (II)

```
package ExpressionEvaluator
```

```
// abstract syntax declarations
```

```
uniontype Exp
```

```
  record RCONST Real x1; end RCONST; 12
```

```
  record PLUS   Exp x1; Exp x2; end PLUS;
```

```
  record SUB    Exp x1; Exp x2; end SUB;
```

```
  record MUL    Exp x1; Exp x2; end MUL;
```

```
  record DIV    Exp x1; Exp x2; end DIV;
```

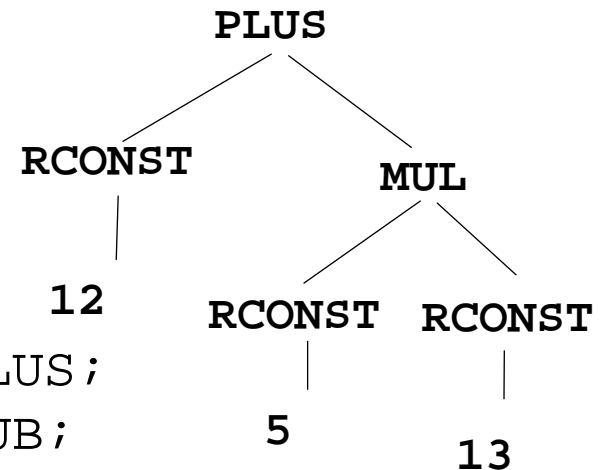
```
  record NEG    Exp x1;          end NEG;
```

```
end Exp;
```

```
// semantic functions
```

```
...
```

```
end ExpressionEvaluator;
```



Expression: 12+5*13

Representation:

```
PLUS(
  RCONST(12),
  MUL(
    RCONST(5),
    RCONST(13)
  )
)
```

MetaModelica – Example (III)

```
package ExpressionEvaluator
// abstract syntax declarations
...

// semantic functions
function eval
  input  Exp  in_exp;
  output Real out_real;
algorithm
  out_real := match in_exp
    local Real v1,v2,v3;  Exp e1,e2;
    case RCONST(v1) then v1;
    case ADD(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 + v2;  then v3;
    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 - v2;  then v3;
    case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 * v2;  then v3;
    case DIV(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 / v2;  then v3;
    case NEG(e1) equation
      v1 = eval(e1); v2 = -v1;  then v2;
  end match;
end eval;

end ExpressionEvaluator;
```


- Based on the RML compiler with a new front-end
- Can handle large specifications
- Supports debugging, mutable arrays
- Supports only a subset of MetaModelica

- To support the full MetaModelica language
 - Integrate the meta-modeling and meta-programming facilities in the OpenModelica compiler
- New features in OpenModelica targeting the MetaModelica Language
 - Pattern matching
 - High-level data structures (list, option, union types, tuples)
 - Exception handling

- Introduction
- Equation-Based Object-Oriented Languages
- MetaModelica
 - Idea, Language constructs, Compiler Prototype, OpenModelica Bootstrapping
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

■ Static aspect

- *Overconstrained system*: the number of variables is smaller than the number of equations
- *Underconstrained system*: the number of variables is larger than the number of equations
- Solved partially by Modelica 3.0 that requires models to be balanced

■ Dynamic (run-time) aspect

- Handles errors due to:
 - *model configuration*: when parameters values for the model simulation are incorrect.
 - *model specification*: when the equations that specify the model behavior are incorrect.
 - *algorithmic code*: when the functions (either native or external) called from equations return incorrect results.

Portable Debugging of EOO Meta-Programs

Why we need debugging

- To debug large meta-programs
- The OpenModelica Compiler Specification
 - 4,65 MB of MetaModelica sources, ~140 000 LOC
 - 52 Packages, 5422 Functions

Debugging strategy: Code Instrumentation

- Early instrumentation
 - Debugging instrumentation at the AST level
 - *Slow compilation and execution time*
- Late instrumentation
 - Debugging instrumentation at the C code level
 - *Acceptable compilation and execution time*

Early Instrumentation – AST level

```
function bubbleSort
  input Real [:] unordElem;
  output Real [size(unordElem, 1)] ordElem;
  protected
    Real tempVal;
    Boolean isOver = false;
  algorithm
    ordElem := unordElem;
    while not isOver loop
      isOver := true;
      for i in 1:size(ordElem, 1)-1 loop
        if ordElem[i] > ordElem[i+1]
          then
            tempVal      := ordElem[i];
            ordElem[i]   := ordElem[i+1];
            ordElem[i+1] := tempVal;
            isOver := false;
          end if;
        end for;
      end while;
    end bubbleSort;
```

```
function bubbleSort
  input Real [:] unordElem;
  output Real [size(unordElem, 1)] ordElem;
  protected
    Real tempVal;
    Boolean isOver = false;
  algorithm
    Debug.register_in("unordElem", unordElem);
    Debug.step(...);
    ordElem := unordElem;
    Debug.register_out("ordElem", ordElem);
    Debug.register_in("isOver", isOver);
    Debug.step(...);
    while not isOver loop
      isOver := true;
      Debug.register_out("isOver", isOver);
      Debug.register_in("ordElem", ordElem);
      Debug.step(...);
      for i in 1:size(ordElem, 1)-1 loop
        Debug.register_out("i", i);
        Debug.register_in("i", i);
        Debug.register_in("ordElem[i]",
                          ordElem[i]);
        Debug.register_in("ordElem[i+1]",
                          ordElem[i+1]);
        Debug.step(...);
        ...
      end bubbleSort;
```

Late Instrumentation – C level

```
function bubbleSort
  input Real [:] unordElem;
  output Real [size(unordElem, 1)] ordElem;
  protected
    Real tempVal;
    Boolean isOver = false;
  algorithm
    ordElem := unordElem;
    while not isOver loop
      isOver := true;
      for i in 1:size(ordElem, 1)-1 loop
        if ordElem[i] > ordElem[i+1]
          then
            tempVal      := ordElem[i];
            ordElem[i]   := ordElem[i+1];
            ordElem[i+1] := tempVal;
            isOver := false;
          end if;
        end for;
      end while;
    end bubbleSort;
```

```
bubbleSort_retype _bubbleSort(real_array unordElem)
{
  size_t tmp2;
  bubbleSort_retype tmp1;
  real_array ordElem; /* [:] */
  modelica_boolean isOver;
  ...
  Debug.register_in("unordElem", unordElem);
  Debug.step(...);
  copy_real_array_data(&unordElem, &ordElem);
  Debug.register_out("ordElem", ordElem);
  Debug.register_in("isOver", isOver);
  Debug.step(...);
  while ...
}
```

The test case

- Meta-Program: *The OpenModelica Compiler*
 - 4,65 MB of MetaModelica sources, ~140 000 lines of code
 - 52 Packages, 5422 Functions
- Compilation times (seconds)

	Generated C Code	Compilation time
No debugging	37 (MB)	269.86 (s)
Early instrumentation	130+ (MB)	850.35 (s)
Late instrumentation	103 (MB)	610.61 (s)

The test case

- RRLargeModel2.mo - *model with 1659 equations/variables*
- Execution time for the OpenModelica Compiler while checking RRLargeModel2.mo

No debugging	223.01 (s)
Early instrumentation	5395.47 (s)
Late instrumentation	864.36 (s)

Eclipse Debugging Environment

- Type information for all variables
- Browsing of complex data structures

The screenshot shows the Eclipse IDE with the following components:

- Breakpoints**: Shows no active breakpoints.
- Variables**: A table showing the current state of variables in the program. The root variable is `p` of type `Absyn.Program`. It contains a `[record]` of type `Absyn.PROGRAM[2]`, which is a list of `Absyn.CLASS` objects. The first element in the list is `[0]`, which is a `Absyn.CLASS[7]` object. This object has several fields: `name` (string "Bla"), `partial_` (bool false), `final_` (bool false), `encapsulated_` (bool false), `restriction` (enum `Absyn.R_MODEL`), `body` (list of `Absyn.PARTS`), `classParts` (list of `Absyn.CLASSPART`), `contents` (list of `Absyn.ELEMENTITEM`), `comment` (string option `NONE`), and `info` (enum `Absyn.INFO`). The `body` list contains one element `[0]` of type `Absyn.PUBLIC` with value `[1]`. The `contents` list contains one element `[0]` of type `Absyn.ELEMENTITEM` with value `[1]`. The `comment` field is `NONE`. The `info` field is `Absyn.INFO[6]`. The `within_` field is `Absyn.TOP` with value `[0]`. The `f` field is a string `"Bla.mo"`. The `->` field is a string `"Bla.mo"`.
- Console**: Shows the output of the program, including the message `Parsed program`.
- Outline**: Shows the structure of the program, including the `translateFile` function.
- Code Editor**: Shows the source code of the `Bla.mo` file, including the `model Bla` block and the `translateFile` function.

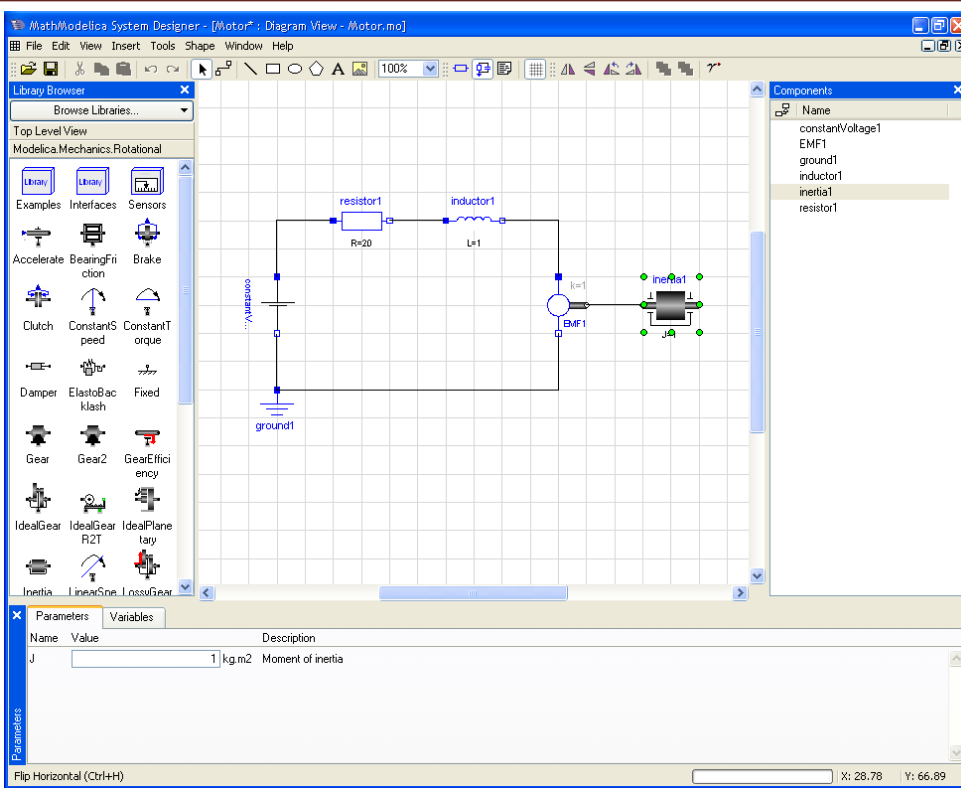
- No type information for variables

The screenshot displays the SML.NET Debugger interface. The main window shows the source code for `Server.sml` with a breakpoint set at line 83, character 54. The code defines a function `alive` that filters a list of neighbours based on a `length` constraint and a `filter` function. The `survivors` variable is assigned the result of `filter` applied to `alive` and `neighbours`. The `newborn` variable is assigned the result of `occurs3` applied to `newbornlist`. The `main` function calls `alive` and `newborn` with the `survivors` and `newborn` variables.

The Call Stack window shows the current call stack, including the `Server.dll` and `Client.exe` processes. The `Command Window - Immediate` shows the current state of the `survivors` variable, which is a `System.Object` of type `System.Tuple2_c` with values `a: 7` and `b: 3`. The `Locals` window shows the current state of the `survivors` variable, which is a `System.Object` of type `System.Tuple2_b` with values `a: 7` and `b: 3`.

```
let val living = alive gen
    val isalive = member living
    val liveneighbours = length o filter isalive o neighbours
    fun twotothree n = n=2 orelse n=3
    val survivors = filter (twotothree o liveneighbours) living
    val newbornlist = collect (filter (not o isalive) o neighbours) living
    val newborn = occurs3 newbornlist
in
  main (survivors @ newborn)
end
```

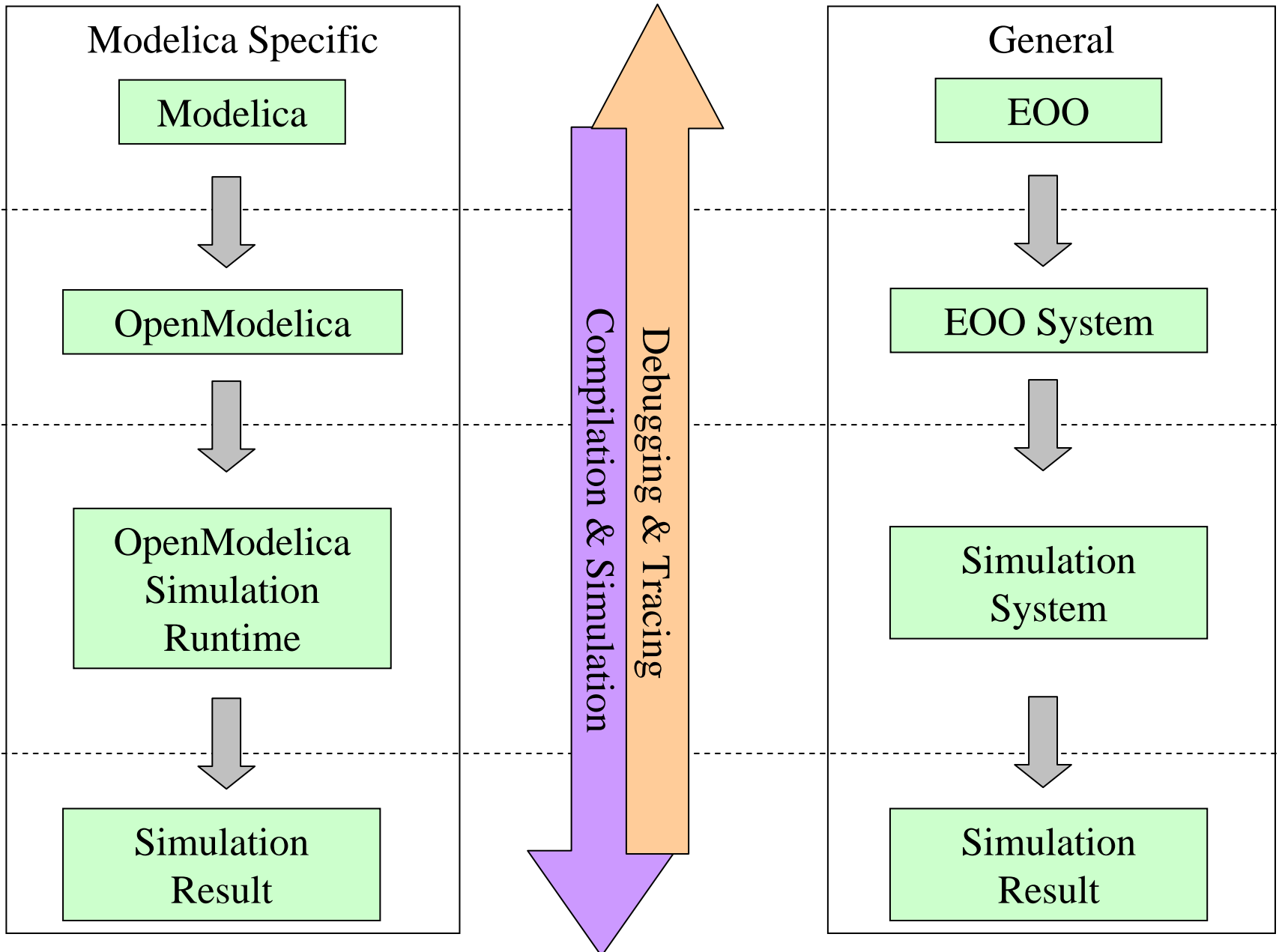
Why do we need Equation-based debugging?



- Easy to build large systems
 - Drag and Drop composition
 - Hierarchical Modeling
- Model behavior depends on data from various sources (xml, databases, files, etc)
- Models could be external (Hardware in the loop, co-simulation, etc)

- You build your model by connecting components together
- You simulate (hopefully there are no compilation errors)
- **The result you get back is wrong!**
 - Why is the result wrong?
 - Where is the error?
 - How can I pin-point the error?

Translation process



Existing Debugging Strategies Do Not Suffice

Modelica Specific

Modelica

OpenModelica

OpenModelica
Simulation
Runtime

Simulation
Files

Compilation & Simulation
Debugging & Tracing

Error

```
model Apollo
  equation
    gravity = ...;
end Apollo;
```

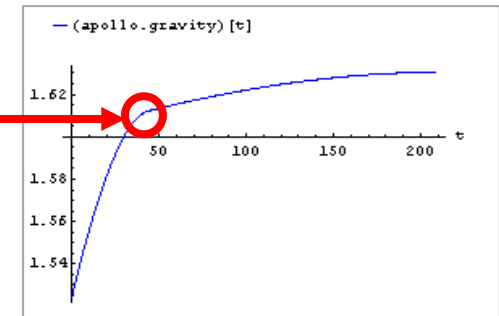
?

Where is the actual code that caused this error?
How do we go back?
How can we automate the round trip?

?

Error Discovered

How do we fix it?
Where is the actual code that caused this error?



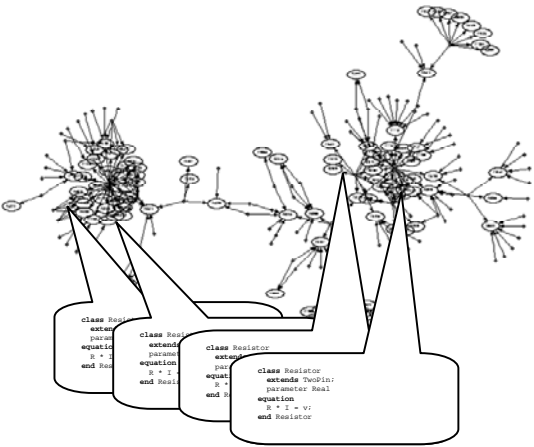
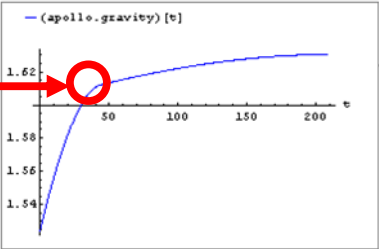
Debugging method

Error Discovered
What now?
Where is the equation or code that generated this error?

Build graph

Interactive Dependency Graph
These equations contributed to the result

Code viewer
Show which model or function the equation node belongs to



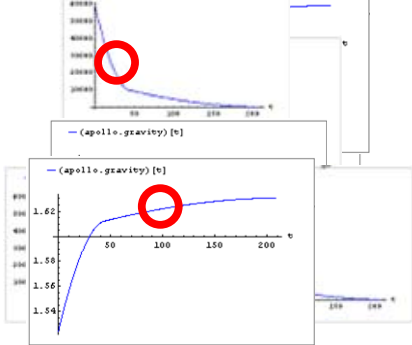
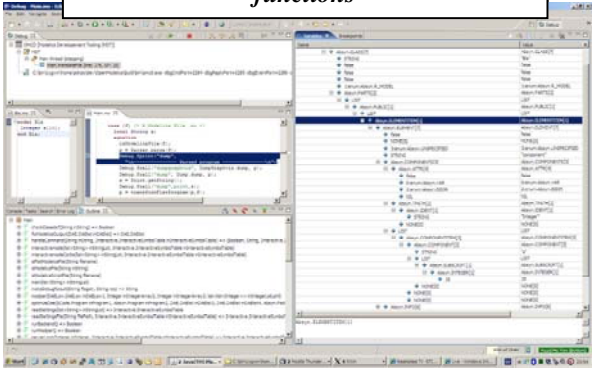
Follow if error is in a function

Follow if error is in an equation

Algorithmic Code Debugging
Normal execution point debugging of functions

Simulation Results
These are the intermediate simulation results that contributed to the result

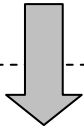
- Mark the error
- Build an interactive graph containing the evaluation
- Walk the graph interactively to find the error



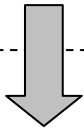
Debugging Strategy: Compiling With Debugging In Mind

Modelica Specific

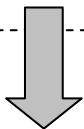
Modelica



OpenModelica



OpenModelica
Simulation
Runtime



Simulation
Files

Compilation & Simulation
Debugging & Tracing

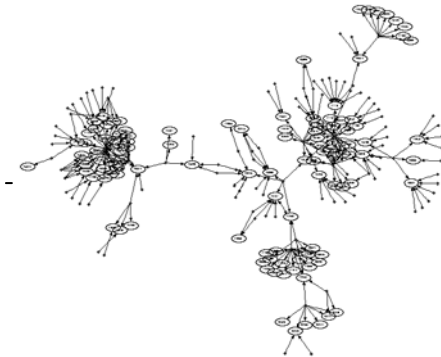
```
model Apollo
```

```
  equation
```

Error

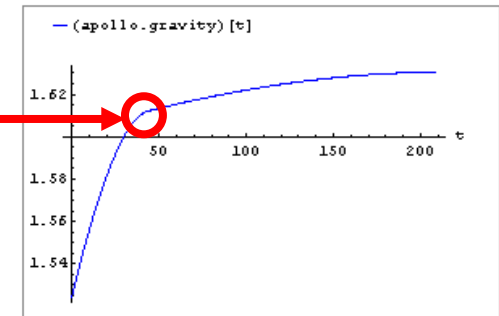
```
    gravity = ...;
```

```
end Apollo;
```



Error Discovered

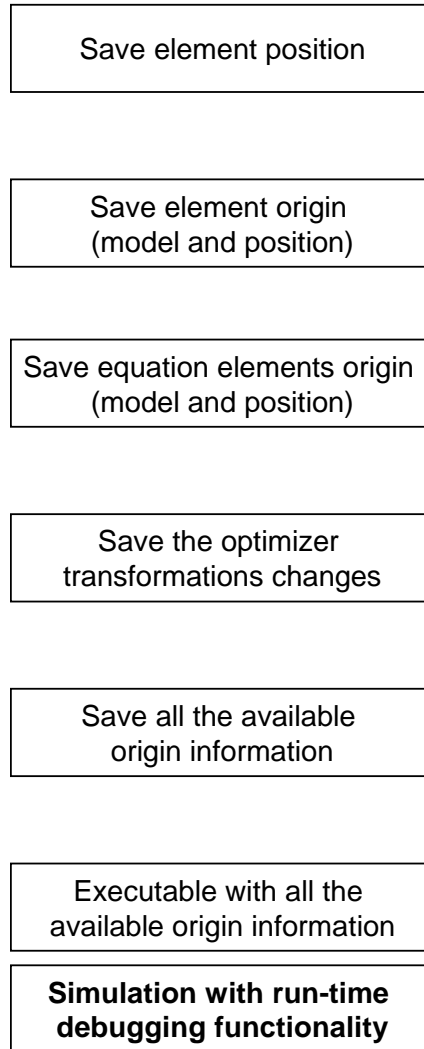
How do we fix it?
Where is the actual
code that caused this
error?



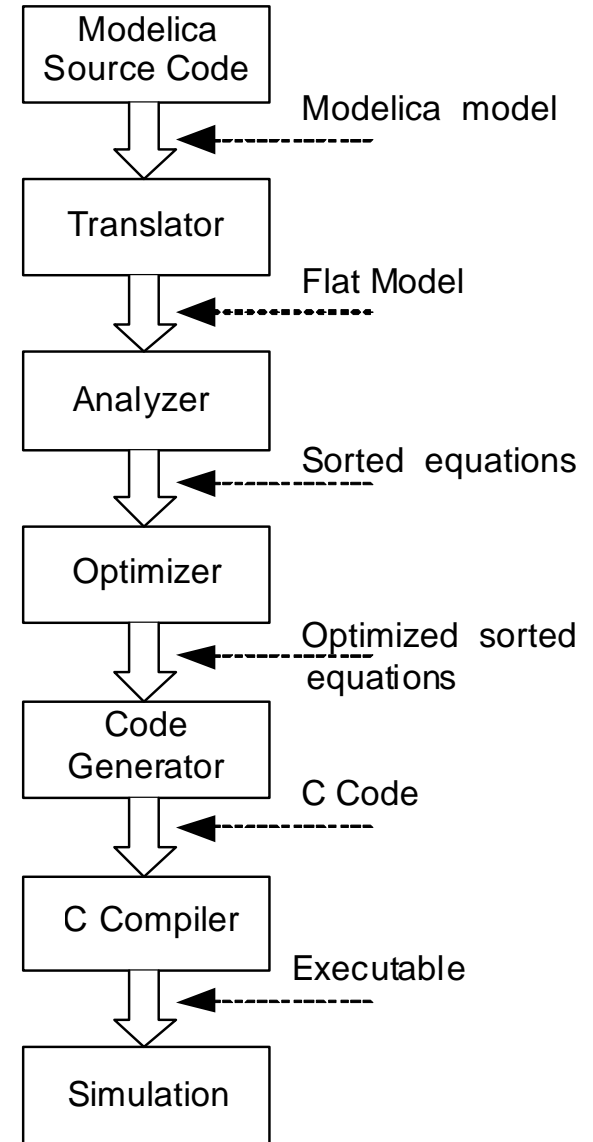
Translation Phases with Debugging

- Include debugging support within the translation process

Debugging Translation Process Additional Steps



Normal Translation Process



- Introduction
- Equation-Based Object-Oriented Languages
- MetaModelica
 - Idea, Language constructs, Compiler Prototype
- OpenModelica Bootstrapping
 - High Level Data Structures, Pattern Matching, Exception Handling
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

- Advanced Interactive Modelica compiler (OMC)
 - Supports most of the Modelica Language
- Basic environments for creating models
 - OMShell - an interactive command handler
 - OMNotebook - a literate programming notebook
 - MDT - an advanced textual environment in Eclipse

OMSHELL - OpenModelica Shell

```

OpenModelica 1.4.3
Copyright 2002-2006, PELAB, Linköping University

To get help on using OMShell and OpenModelica, type "help()" and
press enter.

>> loadModel(Modelica)
true

>> loadFile("C:/OpenModelica1.4.3/testmodels/BouncingBall.mo")
true

>> simulate(BouncingBall, stopTime=3)
record
  resultFile = "BouncingBall_res.plt"
end record

>> plot(h)
true

>>
    
```

tmpPlot.plt

Plot by OpenModelica

DrModelica Modelica Edition

Version 2007-06-20

Copyright: (c) Linköping University, PELAB, 2003-2007, Wiley-IEEE Press, Modelica Association.
 Contact: OpenModelica@ida.lnu.se; OpenModelica Project web site: www.ida.lnu.se/p...
 Book web page: Peter.Fritzon@...

DrModelica Auth: Sandelin, Peter Fritzon
 DrModelica Auth: Peter Fritzon
 This DrModelica language as we simulation. It is Peter Fritzon: Modelica" (200... examples and e... Most of the text...

Detailed Copy

1 Getting Started

IMPORTANT: If you end a con returned in an ou change the direc the `cd()` comman

Van der Pol Model

This example describes a Van der Pol oscillator. Notice that here the keyword `model` is used instead of `class` with the same meaning. This example contains declarations of two state variables `x` and `y`, both of type `Real` and a parameter `lambda`, which is a so-called simulation parameter. The keyword `parameter` specifies that the variable is constant during a simulation run, but can have its value modified before a run, or between runs. Finally, there is an equation section starting with the keyword `equation`, containing two mutually dependent equations that define the dynamics of the model.

```

model VanDerPol "Van der Pol oscillator model"
  Real x(start = 1);
  Real y(start = 1);
  parameter Real lambda = 0.3;
equation
  der(x) = y;
  der(y) = -x + lambda*(1 - x*x)*y;
end VanDerPol;
    
```

1 Simulation of Van der Pol

To illustrate the behavior of the model, we give a command to simulate the Van der Pol oscillator during 25 seconds starting at time 0.

```

simulate(VanDerPol, startTime=0, stopTime=25);
    
```

Perform a parametric plot:

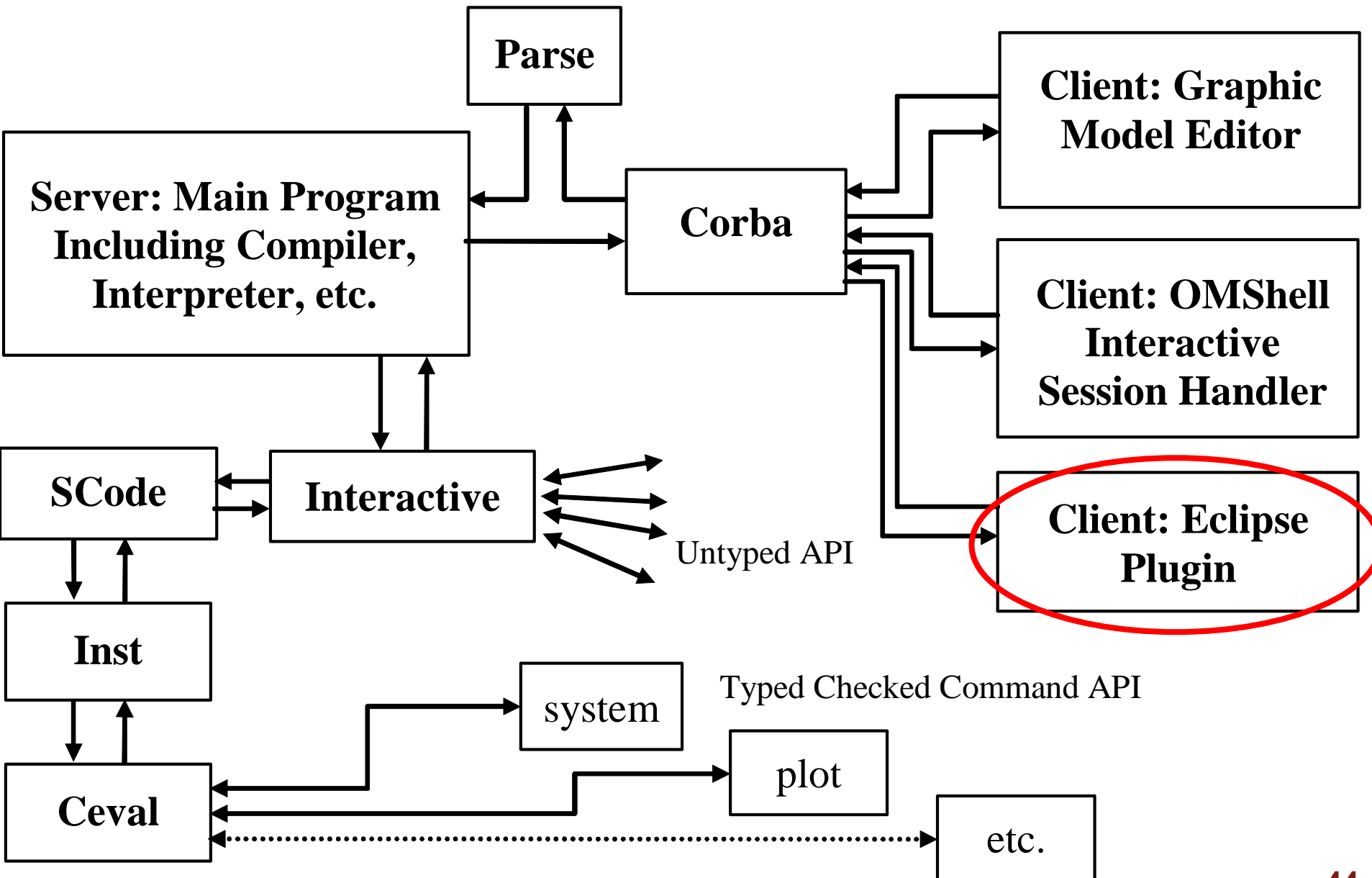
```

plotParametric(x, y);
    
```

Plot by OpenModelica

The image shows two overlapping windows from the Modelica IDE. The top window displays the source code of a model, and the bottom window shows the simulation results, including a plot of the model's behavior over time.

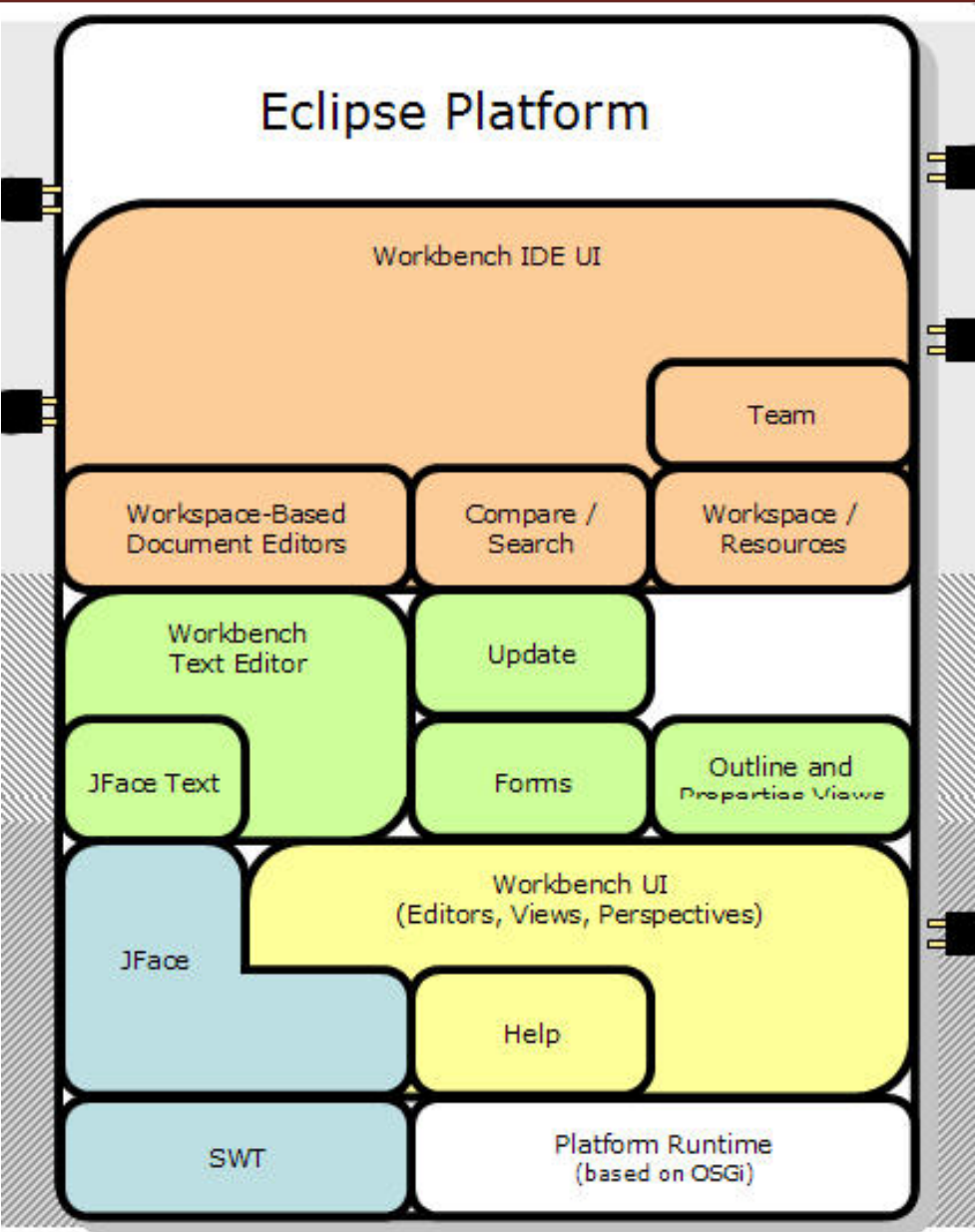
OpenModelica Context



Modelica Development Tooling (MDT)

- Supports textual editing of Modelica/MetaModelica code
- Was created to ease the development of the OpenModelica development (~140 000 lines of code) and to support advanced Modelica library development
- It has most of the functionality expected from a Development Environment
 - code browsing, assistance, indentation, highlighting
 - error detection and debugging
 - automated build of Modelica/MetaModelica projects

The MDT Eclipse Environment (I)



Modelica Browser

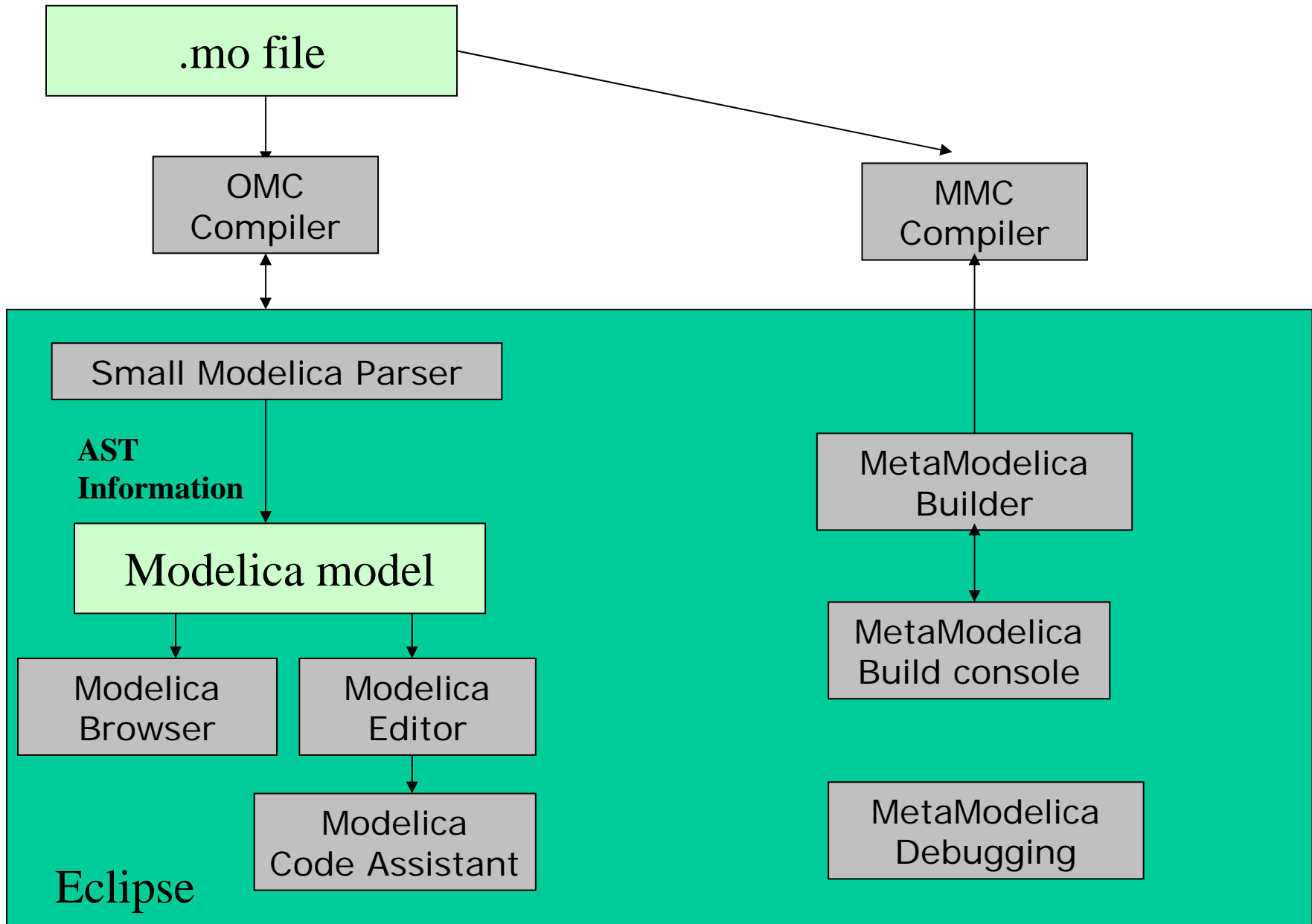
Modelica Editor

Modelica Code Assistant

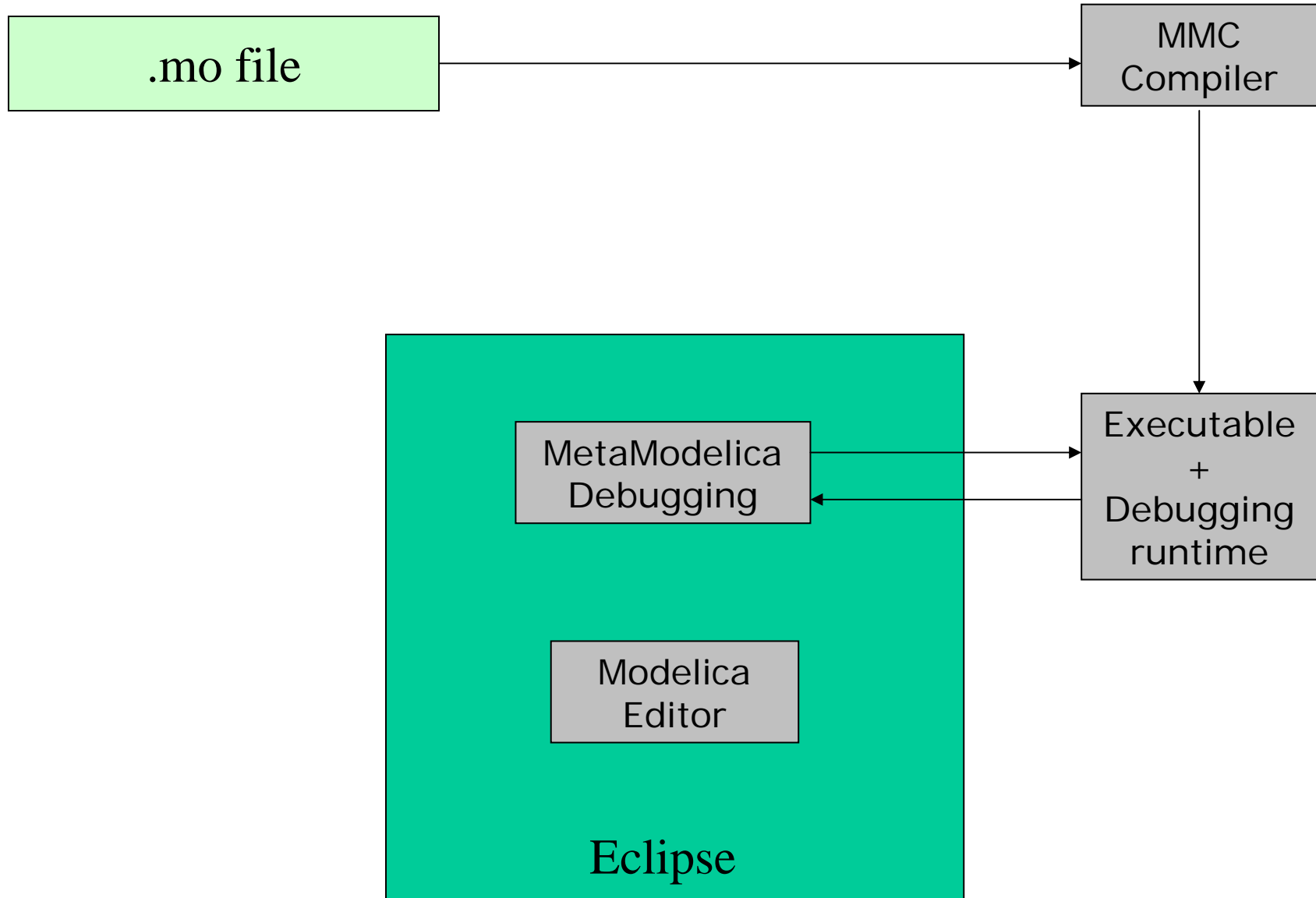
MetaModelica Debugging

Modelica Perspective

The MDT Eclipse Environment (II)



The MDT Eclipse Environment (III)



Creating Modelica projects (I)

Modelica - Eclipse SDK

File Edit Refactor Navigate Search Project Run Window Help

New Alt+Shift+N Project...

Open File... Ctrl+F4

Close Ctrl+Shift+F4

Save Ctrl+S

Save As... Ctrl+Shift+S

Save All

Revert

Move...

Rename... F2

Refresh F5

Convert Line Delimiters To

Print... Ctrl+P

Switch Workspace...

Import

New Project

Select a wizard

Create a new Modelica project.

Wizards:

- Plug-in Project
- C
- C++
- CVS
- Eclipse Modeling Framework
- EJB
- Functional Programming
- J2EE
- Java
- Modelica
- Modelica Project**
- Plug-in Development
- Simple
- Web
- Examples

New Modelica Project

Create a Modelica project

Create a Modelica project in the workspace.

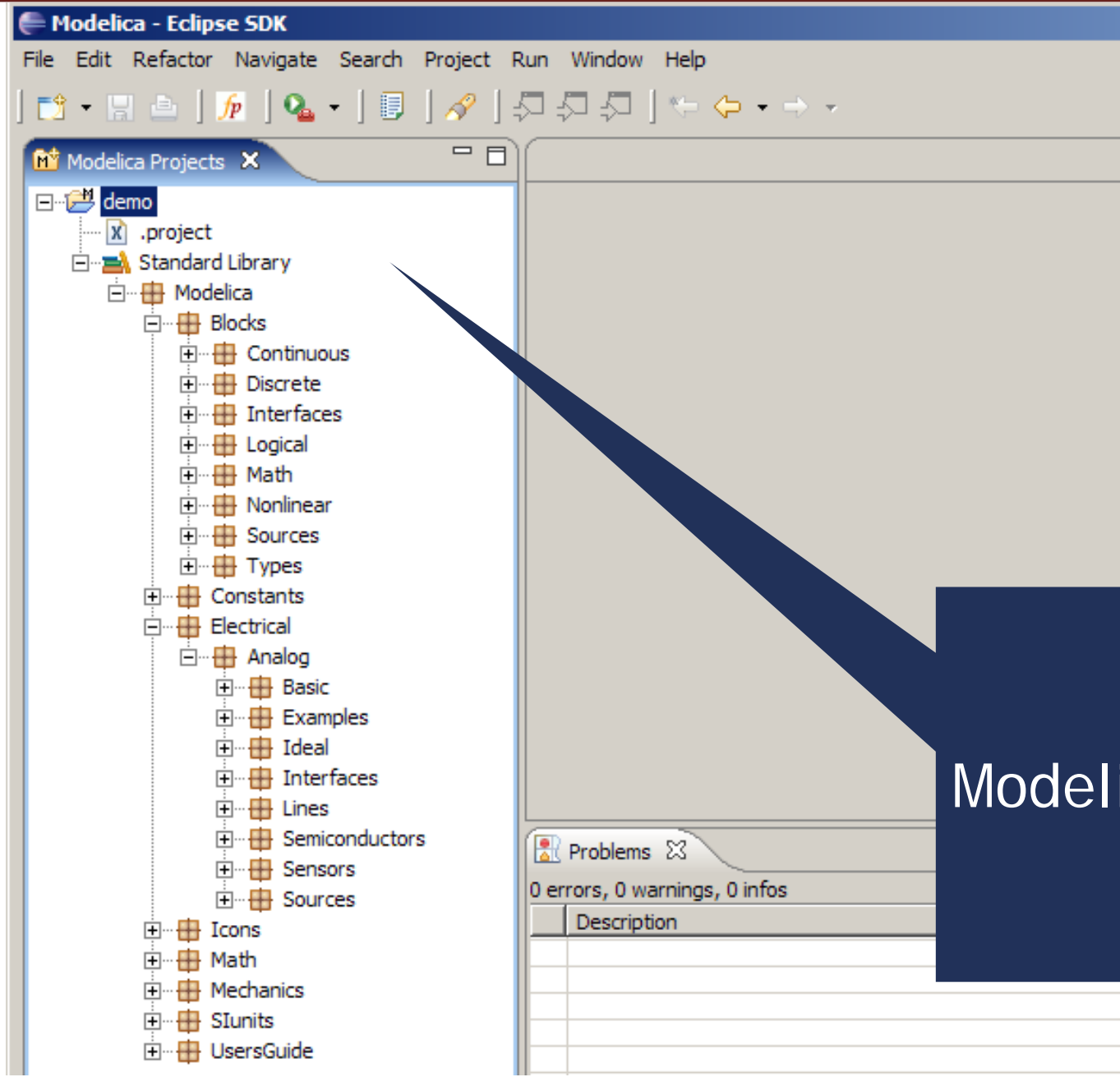
Project name: demo

< Back Next >

< Back Next > Finish Cancel

Creation of Modelica projects using wizards

Creating Modelica projects (II)



Modelica project

Creating Modelica packages

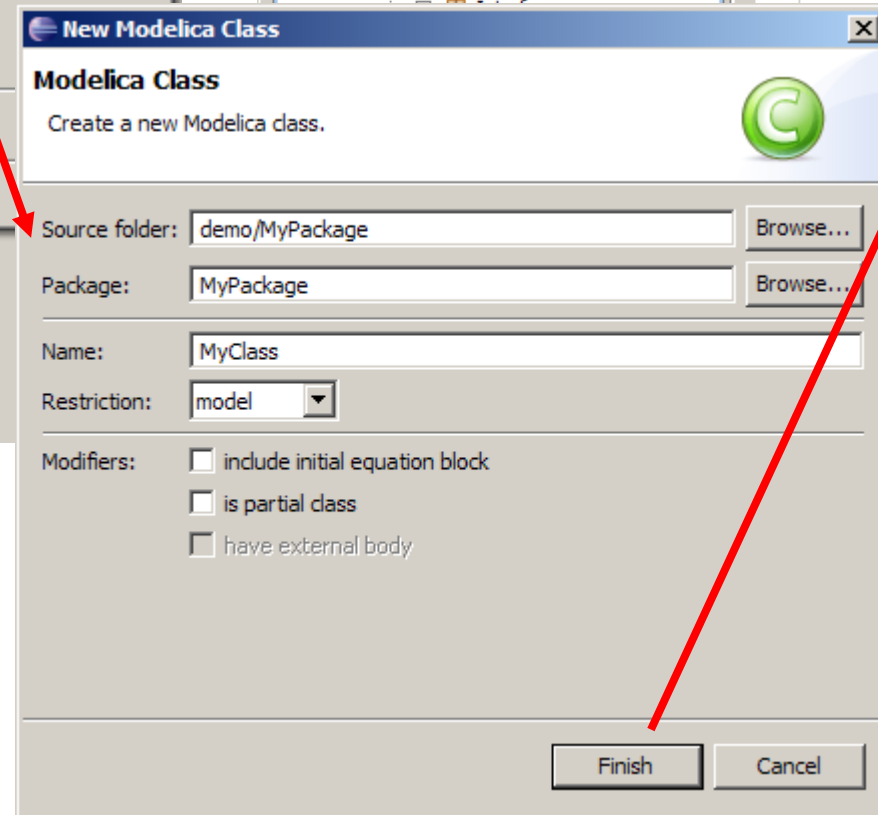
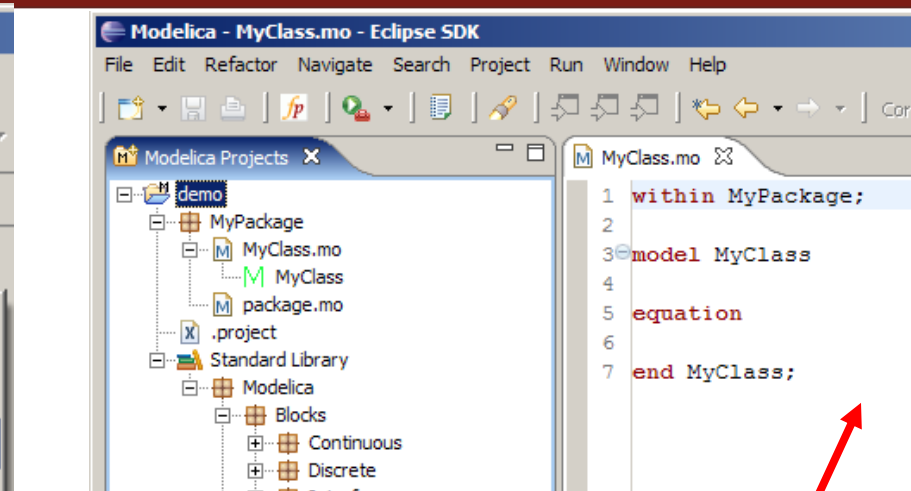
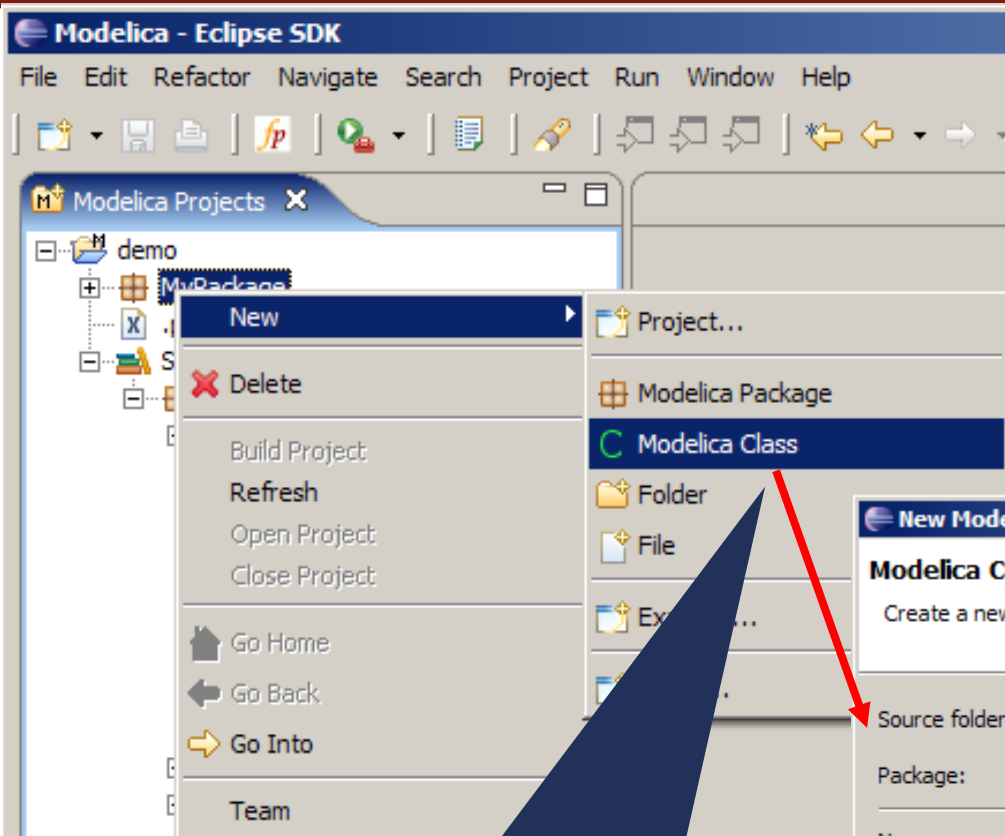
The image shows the Eclipse IDE interface for creating a new Modelica package. The 'New' menu is open, and the 'Modelica Package' option is selected. The 'New Modelica Package' wizard is displayed, showing the following fields:

- Source folder: demo
- Package: (empty)
- Name: MyPackage
- Description: A Modelica Package
- is encapsulated package

The 'Finish' button is highlighted with a red arrow. The 'Modelica Projects' view shows the project structure, including the 'demo' folder and the 'MyPackage' package.

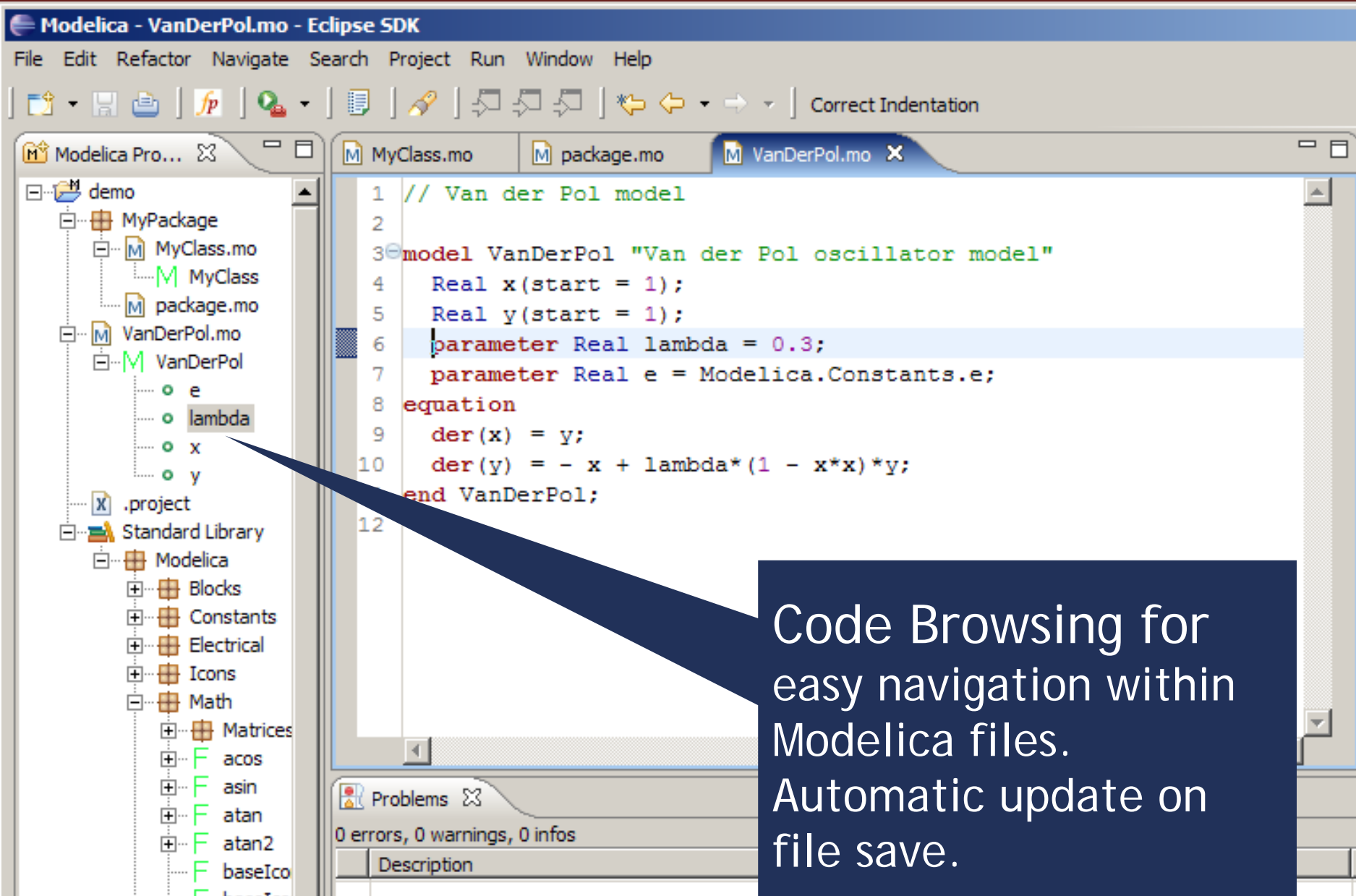
Creation of Modelica packages using wizards

Creating Modelica classes



Creation of Modelica classes, models, etc, using wizards

Code browsing



The screenshot shows the Eclipse IDE interface with the following components:

- Project Explorer (Left):** Displays a project named 'demo' with a package 'VanDerPol' containing a parameter 'lambda'. A blue callout arrow points from the 'lambda' parameter in the tree to the corresponding line in the code editor.
- Code Editor (Center):** Shows the source code for 'VanDerPol.mo'. The line `parameter Real lambda = 0.3;` is highlighted in blue, matching the selection in the Project Explorer.
- Problems Window (Bottom):** Shows '0 errors, 0 warnings, 0 infos'.

```
1 // Van der Pol model
2
3 model VanDerPol "Van der Pol oscillator model"
4   Real x(start = 1);
5   Real y(start = 1);
6   parameter Real lambda = 0.3;
7   parameter Real e = Modelica.Constants.e;
8 equation
9   der(x) = y;
10  der(y) = - x + lambda*(1 - x*x)*y;
11 end VanDerPol;
12
```

Code Browsing for easy navigation within Modelica files. Automatic update on file save.

Error detection (I)

The screenshot shows the Eclipse IDE with the following components:

- Project Explorer:** Shows a project named 'demo' containing a package 'MyPackage' with files 'MyClass.mo', 'package.mo', and 'VanDerPol.mo'. The 'VanDerPol' model is expanded, showing parameters 'e', 'x', and 'y'.
- Code Editor:** Displays the content of 'VanDerPol.mo'. The code is as follows:

```
1 // Van der Pol model
2
3 model VanDerPol "Van der Pol oscillator model"
4   Real x(start = 1);
5   Real y(start = 1);
6   parameter Real lambda = 0.3;
7   parameter Real e = Modelica.Constants.e;
8 equation
9   der(x) = y;
10  der(y) = - x + lambda*(1 - x*x)*y;
11 end VanDerPol;
12
```

Line 6 is highlighted in blue, and a red 'X' icon is visible in the left margin next to it.
- Problems View:** Shows a table with one error entry:

Description	Resource	In Folder	Location
unexpected token: lambda, parsing resumed at token ';' on line 6, column 29	VanDerPol.mo	demo	line 6

A blue callout bubble with white text points to the error message in the Problems view, containing the text: "Parse error detection on file save".

Parse error
detection on
file save

Error detection (II)

The screenshot shows the Eclipse IDE interface for the Modelica project. The left sidebar displays the project structure, including folders like 'Compiler', 'absyn_builder', and 'winruntime', and files like 'Absyn.mo'. The main editor window shows the source code for 'Absyn.mo', with lines 69 to 79. Line 77 is highlighted in blue, indicating an error. The error message in the console is: `Absyn.mo:77.5-77.9 Error: unbound type constructor Withi`. A blue callout box points to this error message with the text 'Semantic error detection on compilation'.

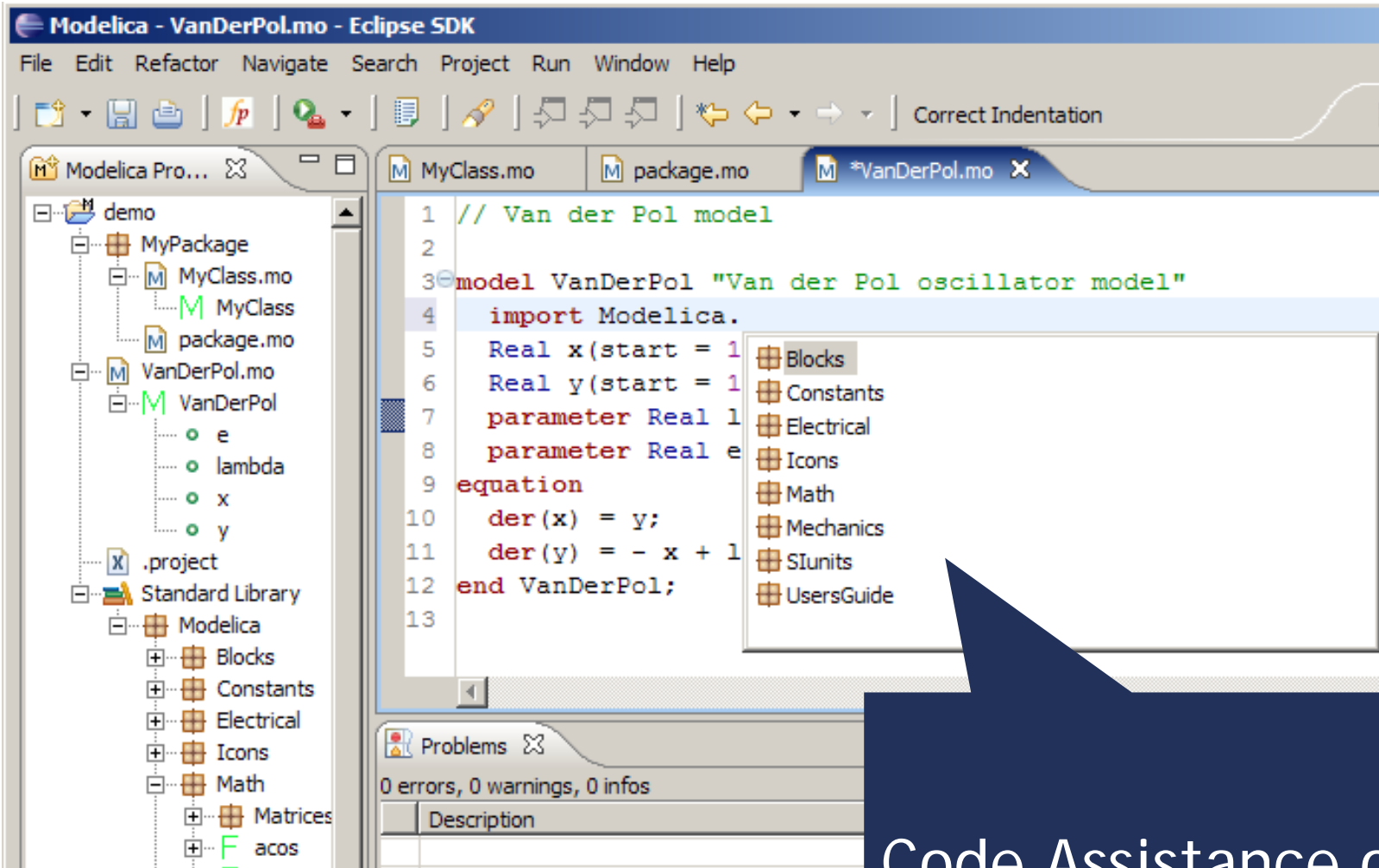
```
69 public
70 uniontype Program "- Programs, the top level construct
71   A program is simply a list of class definitions declared at top
72   level in the source file, combined with a within statement that
73   indicates the hieractical position of the program.
74 "
75   record PROGRAM
76     list<Class> classes "classes ; List of classes" ;
77     Withi within_ "within ; Within statement" ;
78   end PROGRAM;
79
```

Problems Error Log Console

```
<terminated> OMDev-MINGW-OpenModelicaBuilder [Program] c:\OMDev\tools\msys\bin\make.exe
cp -p ../Static.mo Static.mo
cp -p ../SimCodegen.mo SimCodegen.mo
cp -p ../Values.mo Values.mo
cp -p ../System.mo System.mo
/c/OMDev//tools/rml/bin/rmlc -v -Wc,-O3 -c Absyn.mo
"/c/OMDev//tools/rml/bin/rml" -Eplain Absyn.mo
Absyn.mo:77.5-77.9 Error: unbound type constructor Withi
Error: StaticElaborationError
make[2]: Leaving directory `~/c/bin/mingw/home/...
make[1]: Leaving directory `~/c/bin/cy/...
make[2]: *** [Absyn.h] Error 1
make[1]: *** [omc_release] Error 2
make: *** [omc] Error 2
```

Semantic error
detection on
compilation

Code assistance (I)



The screenshot shows the Eclipse IDE interface with the following components:

- Project Explorer (Left):** Shows a project named 'demo' containing a package 'MyPackage' with files 'MyClass.mo', 'package.mo', and 'VanDerPol.mo'. The 'VanDerPol.mo' file is expanded to show a class 'VanDerPol' with parameters 'e', 'lambda', 'x', and 'y'. A 'Standard Library' is also visible with categories like 'Blocks', 'Constants', 'Electrical', 'Icons', 'Math', 'Matrices', and 'acos'.
- Editor (Center):** Displays the code for 'VanDerPol.mo'. The code is as follows:

```
1 // Van der Pol model
2
3 model VanDerPol "Van der Pol oscillator model"
4   import Modelica.
5   Real x(start = 1
6   Real y(start = 1
7   parameter Real l
8   parameter Real e
9   equation
10  der(x) = y;
11  der(y) = - x + 1
12 end VanDerPol;
13
```
- Code Assistance (Right):** A pop-up window is shown over the 'import Modelica.' line, displaying a list of categories: 'Blocks', 'Constants', 'Electrical', 'Icons', 'Math', 'Mechanics', 'SIunits', and 'UsersGuide'.
- Problems (Bottom):** Shows '0 errors, 0 warnings, 0 infos'.

Code Assistance on imports

Code assistance (II)

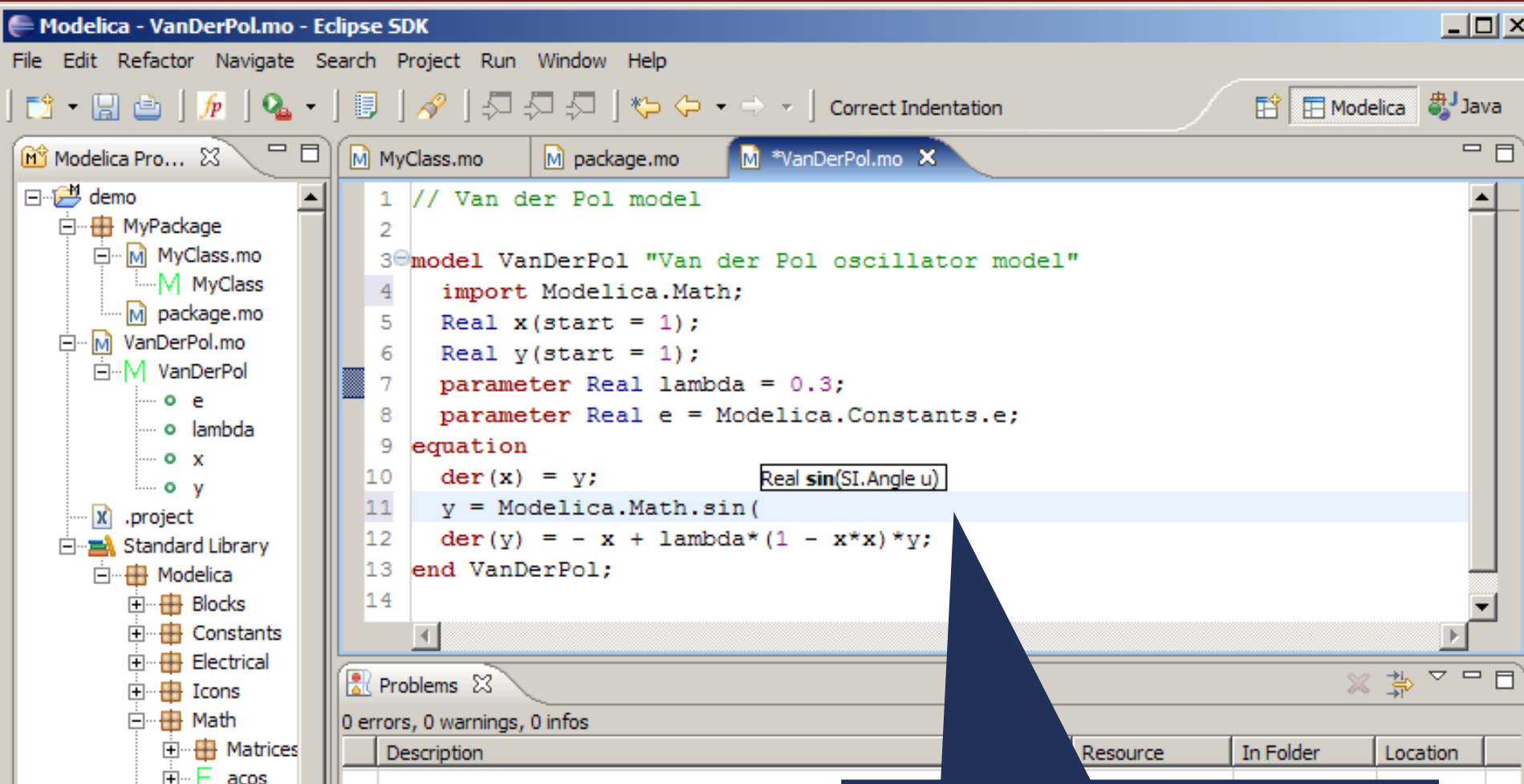
The screenshot displays the Eclipse IDE interface for a Modelica project. The left-hand side shows a project explorer with a tree view containing a 'demo' package, 'MyPackage', 'MyClass', 'package.mo', 'VanDerPol.mo', and 'Standard Library'. The main editor window shows the file '*VanDerPol.mo' with the following code:

```
1 // Van der Pol model
2
3 model VanDerPol "Van der Pol oscillator model"
4   import Modelica.Math;
5   Real x(start = 1);
6   Real y(start = 1);
7   parameter Real lambda = 0.3;
8   parameter Real e = Modelica.Constants.
9 equation
10  der(x) = y;
11  der(y) = - x + lambda*(1 - x*x)*y;
12 end VanDerPol;
13
```

Line 8 is selected, and a code completion popup is visible on the right, listing constants from the 'Modelica.Constants' package. The list includes 'c', 'D2R', 'e', 'eps', 'epsilon_0', 'G', 'g_n', 'h', and 'inf'. The 'e' constant is highlighted. Below the editor, the 'Problems' window shows '0 errors, 0 warnings, 0 infos'.

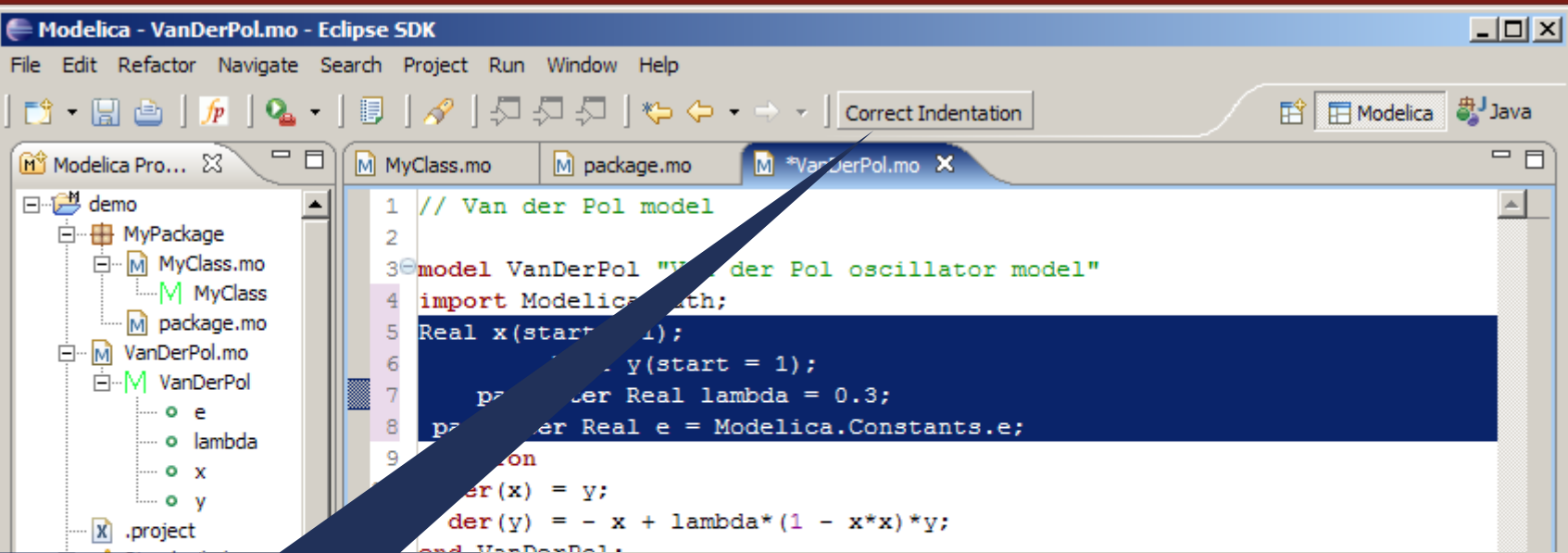
Code Assistance on assignments

Code assistance (III)



Code Assistance on
function calls

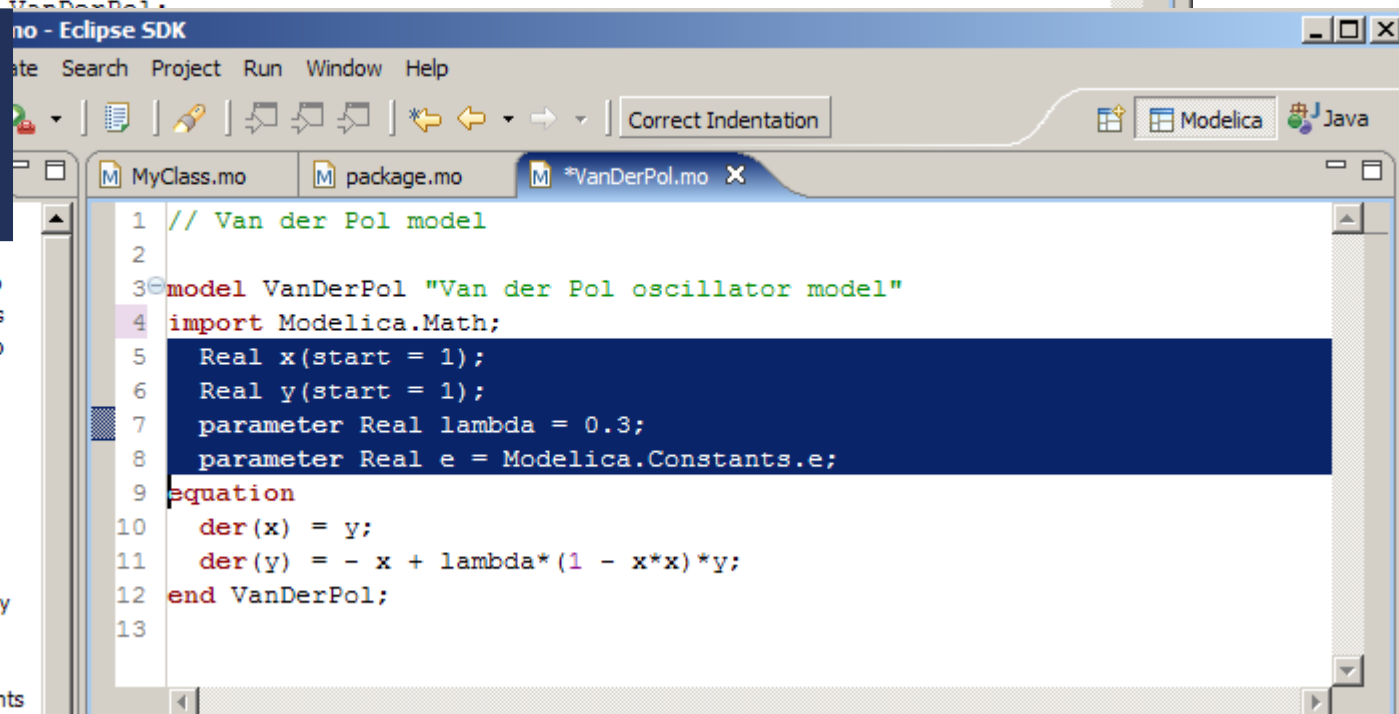
Code indentation



Modelica - VanDerPol.mo - Eclipse SDK

```
1 // Van der Pol model
2
3 model VanDerPol "Van der Pol oscillator model"
4 import Modelica.Math;
5 Real x(start = 1);
6 Real y(start = 1);
7 parameter Real lambda = 0.3;
8 parameter Real e = Modelica.Constants.e;
9
10 equation
11   der(x) = y;
12   der(y) = - x + lambda*(1 - x*x)*y;
13 end VanDerPol;
```

Code
Indentation



Modelica - Eclipse SDK

```
1 // Van der Pol model
2
3 model VanDerPol "Van der Pol oscillator model"
4 import Modelica.Math;
5   Real x(start = 1);
6   Real y(start = 1);
7   parameter Real lambda = 0.3;
8   parameter Real e = Modelica.Constants.e;
9
10 equation
11   der(x) = y;
12   der(y) = - x + lambda*(1 - x*x)*y;
13 end VanDerPol;
```

Code Outline and Hovering Info

The screenshot displays the Eclipse IDE interface for the Modelica project. The top-left pane shows the 'Modelica Projects' tree with 'Absyn.mo' selected. The main editor shows the source code of 'Absyn.mo' with a hover tooltip over the function 'getCrefFromExp'. The bottom-left pane shows the 'Outline' view for 'Absyn' with a tree of algorithm items. The bottom-right pane shows the 'Problems' view with a list of errors.

```
case (MATRIX(matrix = exp11))
  local list<list<list<ComponentRef>>> res1;
  equation
    res1 = Util.listListMap(exp11, getCrefFromExp);
    res2 = Util.listFlatten(res1);
    res = Util.listFlatten(res2);
  then
    res;
case (RANGE(start = e1, step = SOME(e3), stop = e2))
  equation
    l1 = getCrefFromExp(e1);
    l2 =
      function getCrefFromExp "function: getCrefFromExp
        Returns a flattened list of the
        component references in an expression"
        input Exp inExp;
        then
          output list<ComponentRef> outComponentRefLst;
        algorithm
          outComponentRefLst:=matchcontinue inExp
          local
            l1 =
              ComponentRef cr;
```

Identifier Info on Hovering

Code Outline for easy navigation within Modelica files

113 errors, 0 warnings, 0 infos

Description	File
The identifier at start and end are different	OpenModelica/tools/rml2mod
The identifier at start and end are different	OpenModelica/tools/rml2mod
The identifier at start and end are different, par	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod
' on line	OpenModelica/tools/rml2mod

64M of 254M

Ctrl Contrib (Bottom)

Go to definition

The screenshot shows the Eclipse IDE with the 'Absyn.mo' file open. The 'Go to Definition' feature is demonstrated by hovering over the identifier 'getCrefFromExp' in the code. A callout box provides the following information:

```
function getCrefFromExp "function: getCrefFromExp"
  Returns a flattened list of the
  component references in an expression"
input Exp inExp;
output list<ComponentRef> outComponentRefList;
algorithm
  outComponentRefList:=matchcontinue inExp
  local
    ComponentRef cr;
```

The IDE interface includes a 'Modelica Projects' view on the left, an 'Outline' view at the bottom left, and a 'Console' view at the bottom right. The status bar at the bottom indicates '64M of 254M' and 'Ctrl Contrib (Bottom)'.

CTRL+Click on
identifier goes to
definition

Identifier Info on
Hovering

- Introduction
- Equation-Based Object-Oriented Languages
- MetaModelica
 - Idea, Language constructs, Compiler Prototype
- OpenModelica Bootstrapping
 - High Level Data Structures, Pattern Matching, Exception Handling
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

System Modeling Language (SysML™)

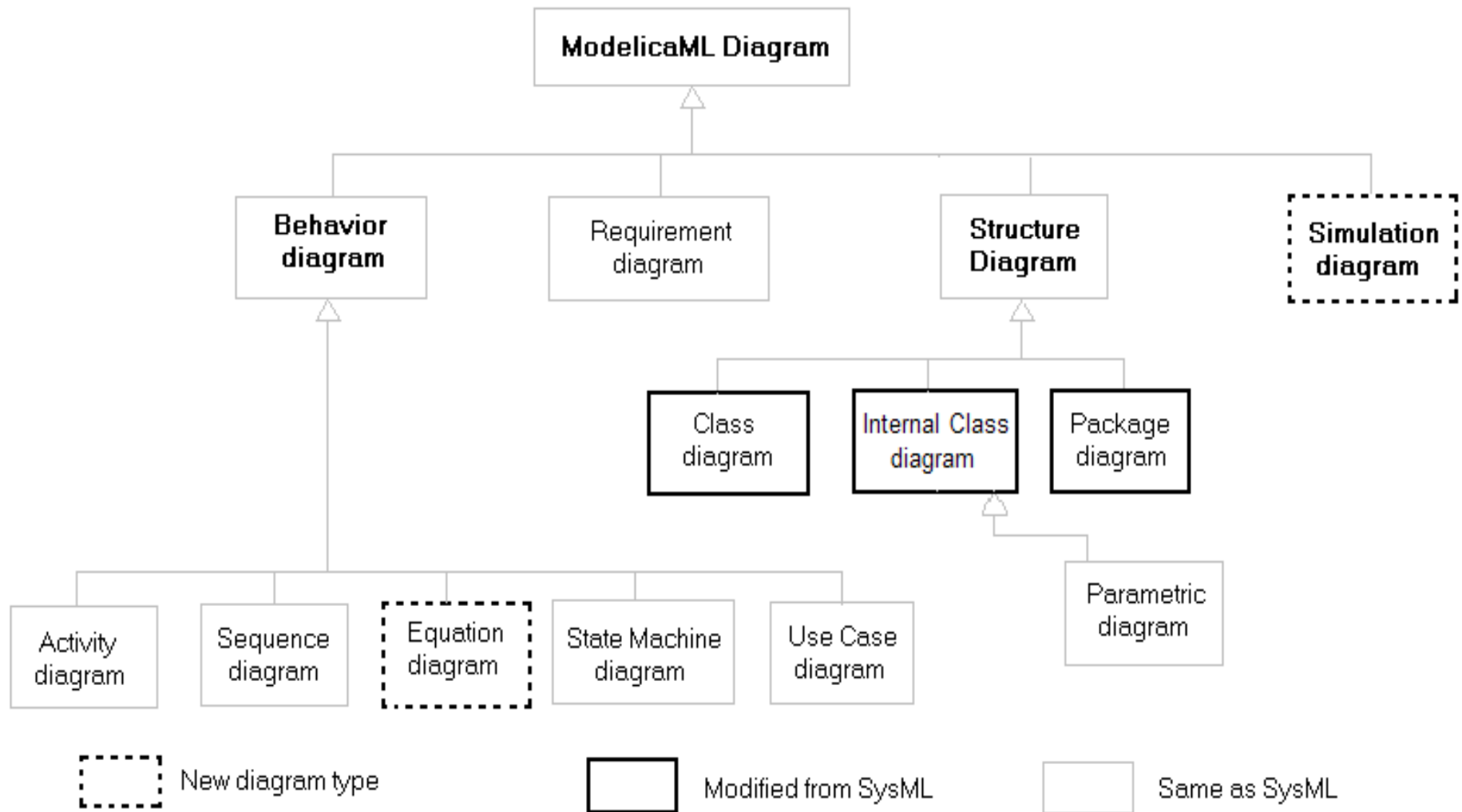
- Graphical modeling language for Systems Engineering constructed as a UML2 Profile
- Designed to provide simple but powerful constructs for modeling a wide range of systems engineering problems
- Effective in specifying requirements, structure, behavior, allocations, and constraints on system properties to support engineering analysis
- Intended to support multiple processes and methods such as structured, object-oriented, etc.

ModelicaML - a UML profile for Modelica

- Supports modeling with all Modelica constructs i.e. restricted classes, equations, generics, discrete variables, etc.
- Multiple aspects of a system being designed are supported
 - system development process phases such as requirements analysis, design, implementation, verification, validation and integration.
- Supports mathematical modeling with equations (to specify system behavior). Algorithm sections are also supported.
- Simulation diagrams are introduced to configure, model and document simulation parameters and results in a consistent and usable way.
- The ModelicaML meta-model is consistent with SysML in order to provide SysML-to-ModelicaML conversion and back.

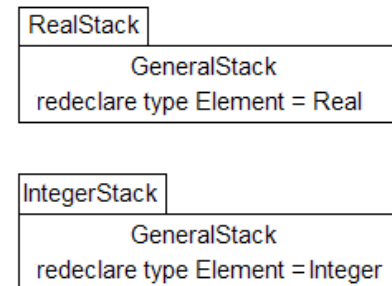
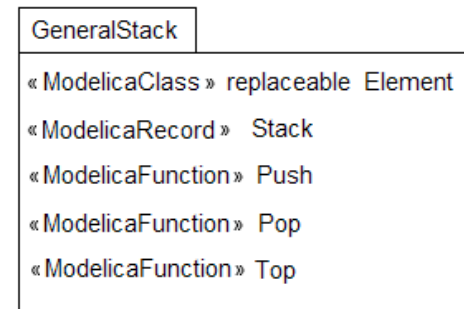
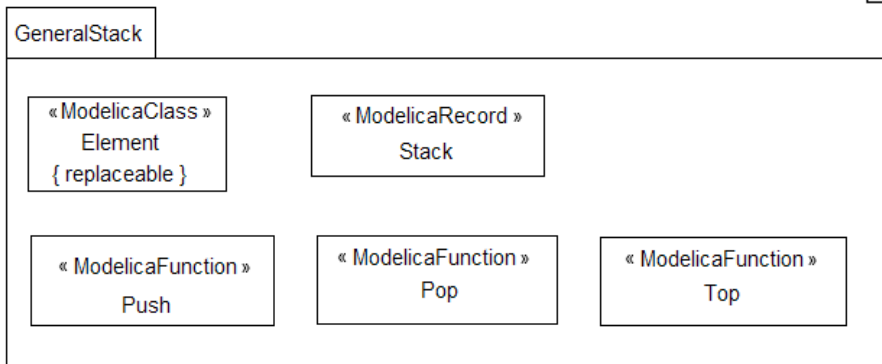
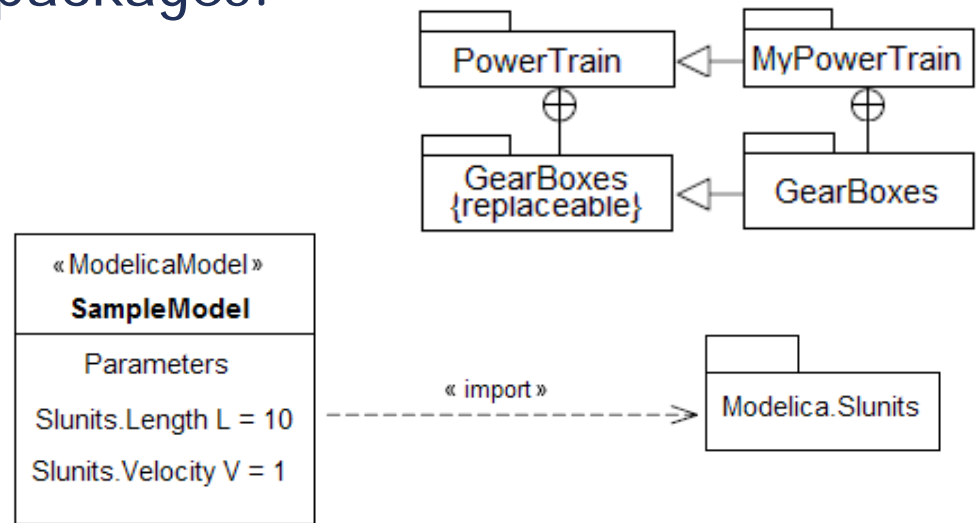
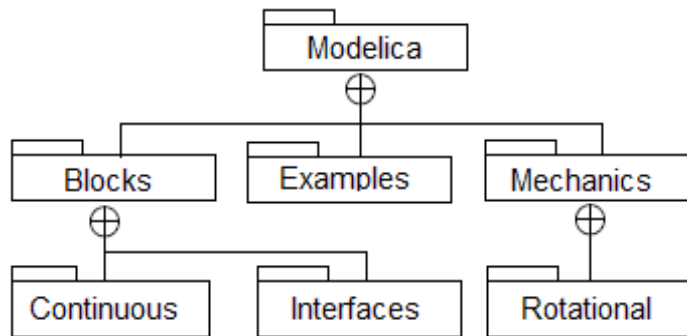
- *Targeted to Modelica and SysML users*
- *Provide a SysML/UML view of Modelica for*
 - *Documentation purposes*
 - *Language understanding*
- *To extend Modelica with additional design capabilities (requirements modeling, inheritance diagrams, etc)*
- *To support translation between Modelica and SysML models via XMI*

ModelicaML - Overview



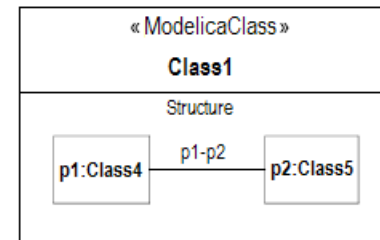
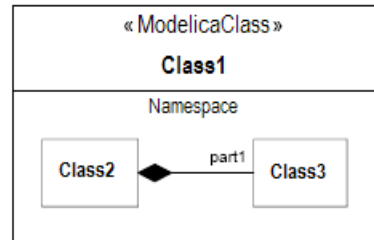
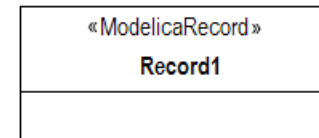
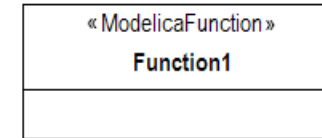
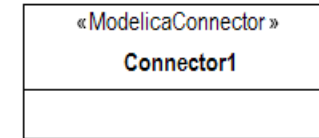
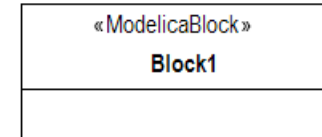
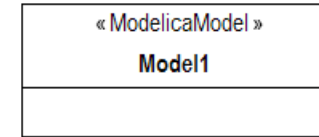
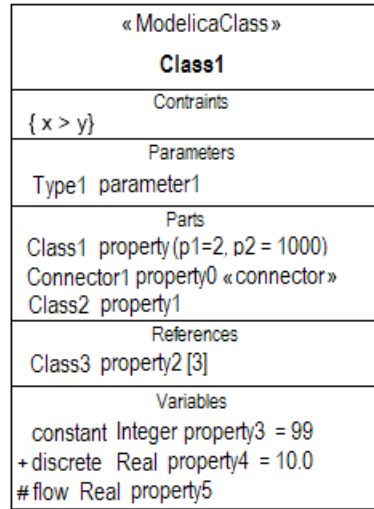
ModelicaML – Package Diagram

- The Package Diagram groups logically connected user defined elements into packages.
- The primarily purpose of this diagram is to support the specifics of the Modelica packages.



ModelicaML – Class Diagram

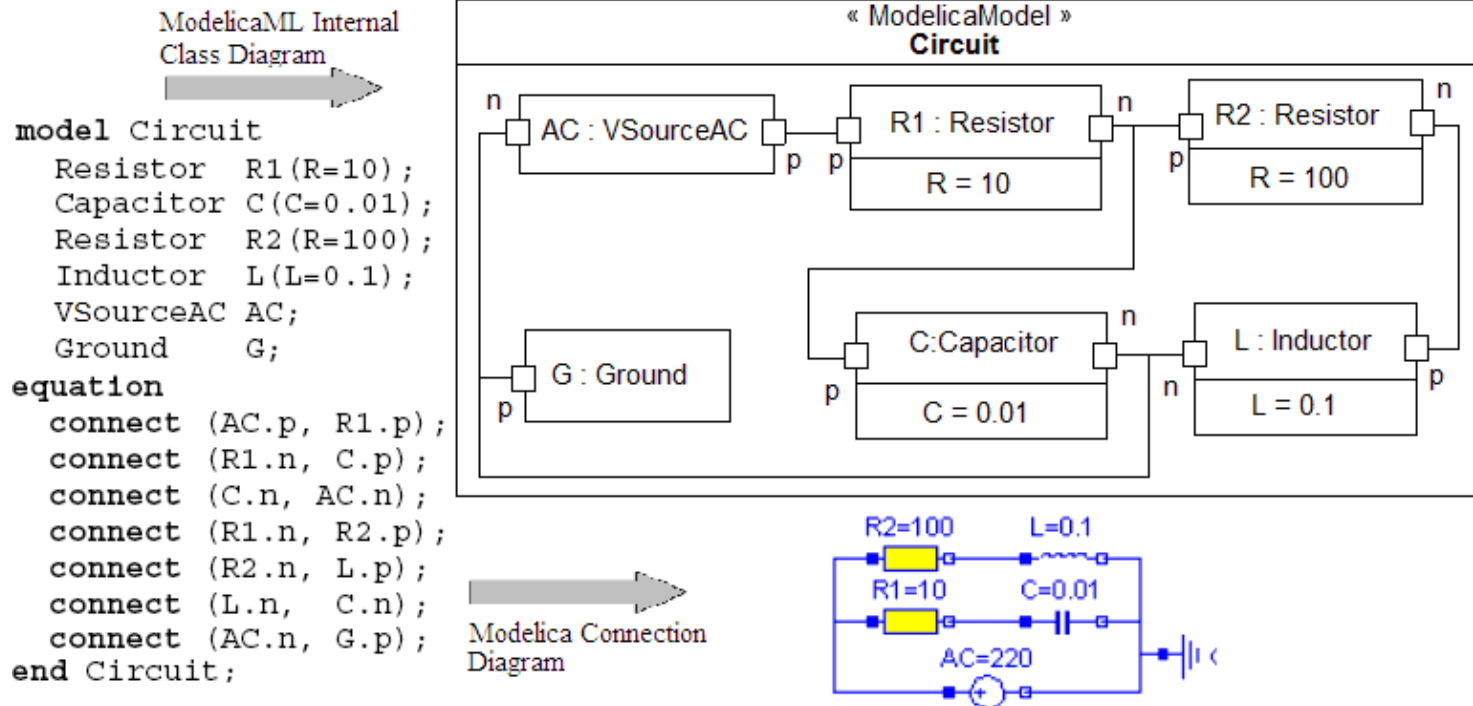
- ModelicaML provides extensions to SysML in order to support the full set of Modelica constructs.
- ModelicaML defines unique class definition types ModelicaClass, ModelicaModel, ModelicaBlock, ModelicaConnector, ModelicaFunction and ModelicaRecord that correspond to class, model, block, connector, function and record restricted Modelica classes.
- Modelica specific restricted classes are included because a modeling tool needs to impose their semantic restrictions (for example a record cannot have equations, etc).



Class Diagram defines Modelica classes and relationships between classes, like generalizations, association and dependencies

ModelicaML - Internal Class Diagram

- Internal Class Diagram shows the internal structure of a class in terms of parts and connections

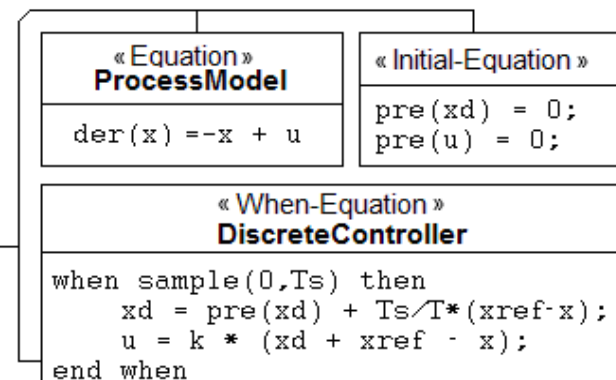
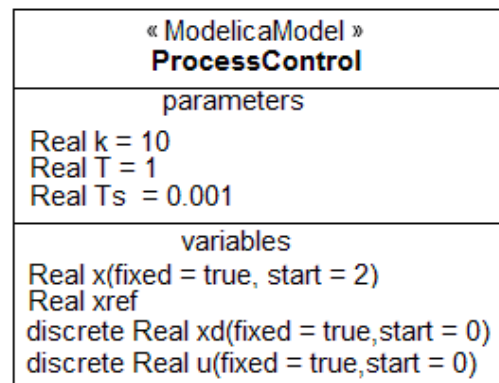
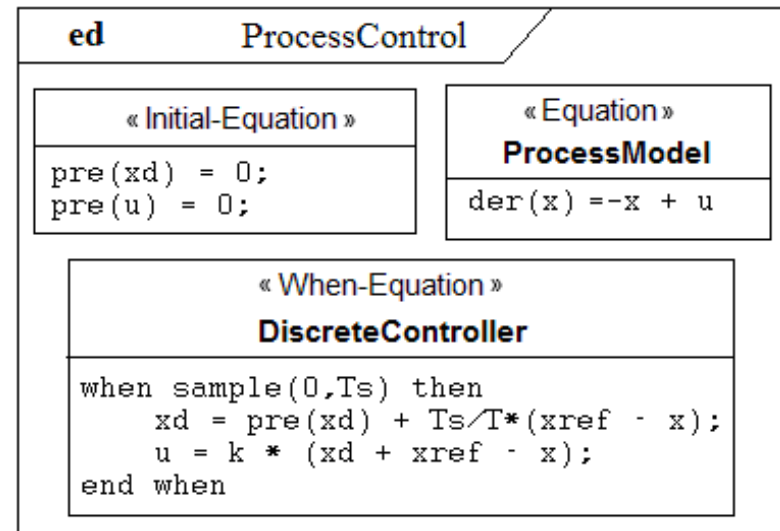


ModelicaML – Equation Diagram

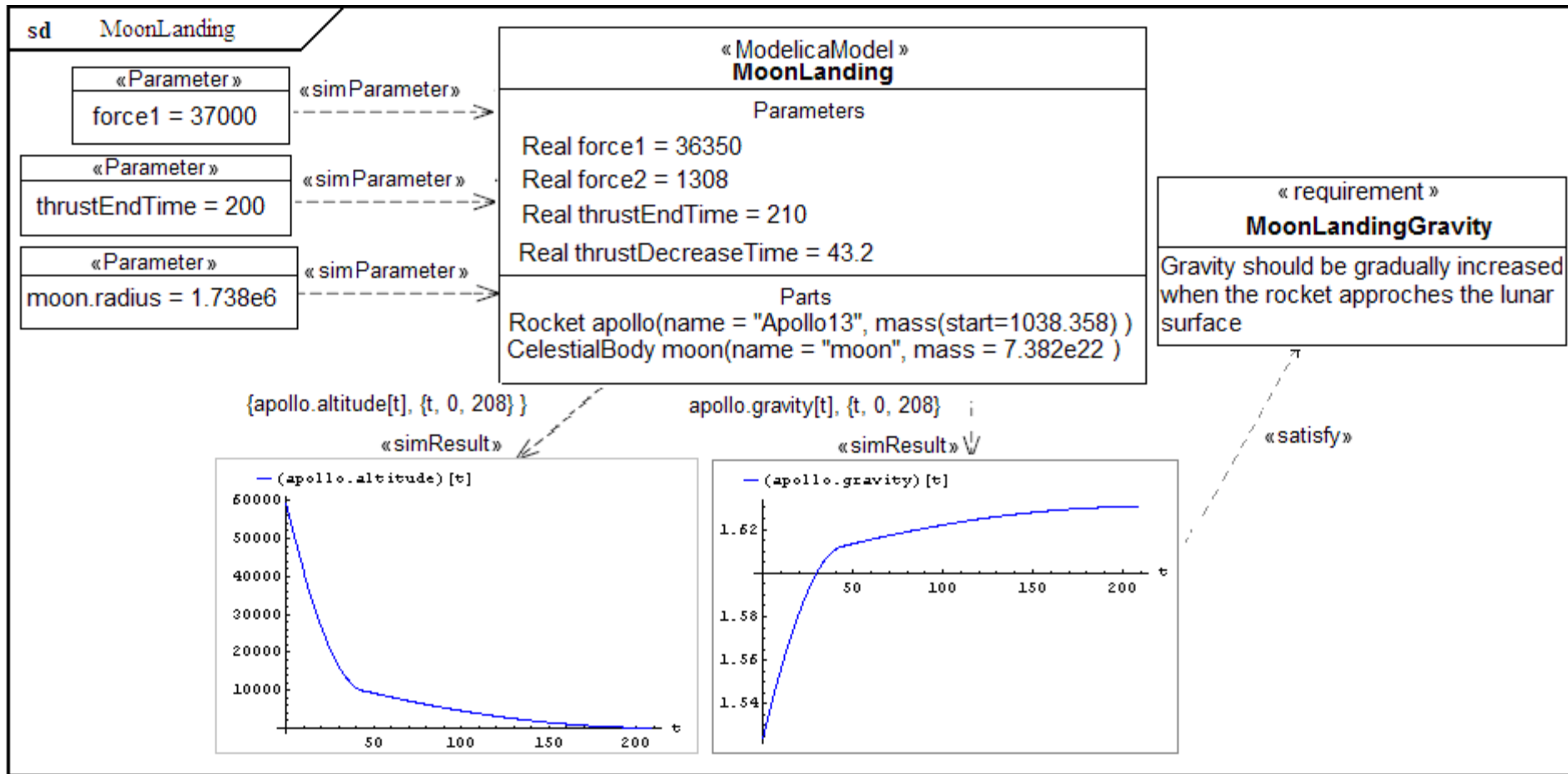
- behavior is specified using Equation Diagrams
- all Modelica equations have their specific diagram:
 - initial, when, for, if equations

```

model ProcessControl
  parameter Real k=10,T=1;
  parameter Real Ts=0.001;
  Real x(fixed=true,start=2);
  Real xref;
  discrete Real xd(fixed=true,start=0);
  discrete Real u(fixed=true,start=0);
equation
  der(x) = -x + u; // Process model
  // Discrete PI Controller
  when sample(0,Ts) then
    xd=pre(xd)+Ts/T*(xref-x);
    u=k*(xd + xref - x);
  end when;
initial equation
  pre(xd) = 0; pre(u) = 0;
end ProcessControl;
    
```



ModelicaML – Simulation Diagram



- Used to model, configure and document simulation parameters and results
- Simulation diagrams can be integrated with any Modelica modeling and simulation environment (OpenModelica)

Eclipse environment for ModelicaML

The screenshot shows the Eclipse IDE with the ModelicaML environment. The main window displays a UML class diagram with the following classes and relationships:

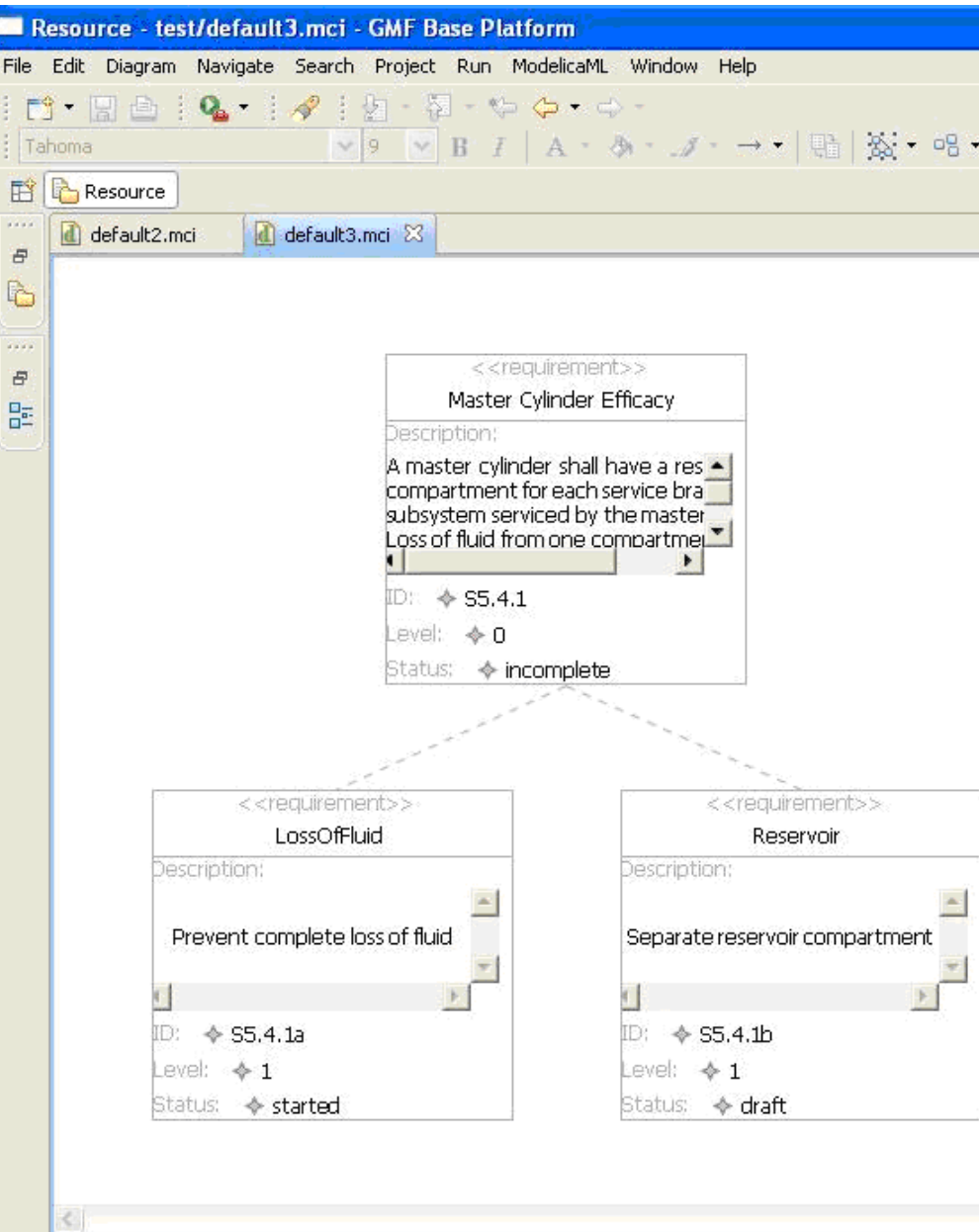
- TwoPin** (class): Parameters: + p, + n.
- Resistor** (class): Parameters: Real R (unit = 'Ohm').
- Inductor** (class): Parameters: Real L ((unit = "H")).

Relationships:

- TwoPin is a generalization of Resistor and Inductor.
- Resistor is associated with Inductor via the parameter R2(R = 100).
- Inductor is associated with Resistor via the parameter I(L=0.1).
- Resistor is associated with TwoPin via the parameter R1(R = 10).

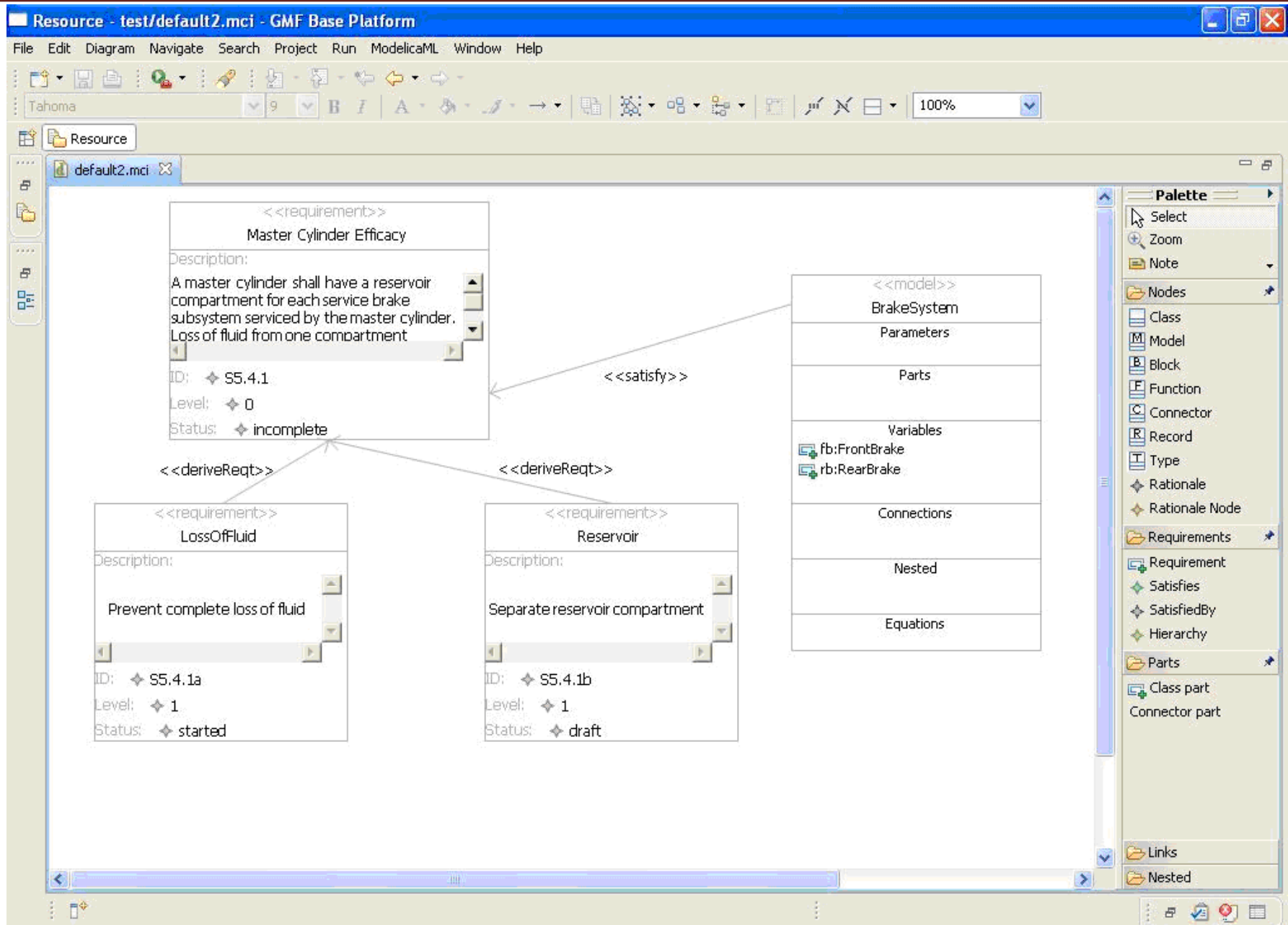
The Properties view for the selected Resistor class shows the following properties:

Property	Value
UML	
Access	public
Array Dimension	
Default	unit = 'Ohm'
Direction	inout
Is Flow	false
Name	R



- Requirements
 - can be *modeled hierarchically*
 - can be *traced*
 - can be *linked* with other ModelicaML models
 - can be *queried* with respect of their attributes and links (coverage)

Requirements Modeling in Eclipse



- Introduction
- Equation-Based Object-Oriented Languages
- MetaModelica
 - Idea, Language constructs, Compiler Prototype
- OpenModelica Bootstrapping
 - High Level Data Structures, Pattern Matching, Exception Handling
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

- EOO languages can be successfully generalized to also support software modeling, thus addressing the whole product modeling process.
- Integrated environments that support such a generalized EOO language can be created and effectively used on real-sized applications.

- Conclude the OpenModelica bootstrapping
- Further develop the EOO debugging framework
- Modularity and scalability of MetaModelica language

- Introduction
- Equation-Based Object-Oriented Languages
- MetaModelica
 - Idea, Language constructs, Compiler Prototype
- OpenModelica Bootstrapping
 - High Level Data Structures, Pattern Matching, Exception Handling
- Debugging of Equation-Based Object-Oriented Languages
 - Debugging of EOO Meta-Programs (Late vs. Early instrumentation)
 - Runtime debugging
- Integrated Environments for Equation-Based Object-Oriented Languages
- ModelicaML - A UML/SysML profile for Modelica
- Conclusions and Future Work
- Thesis Contributions

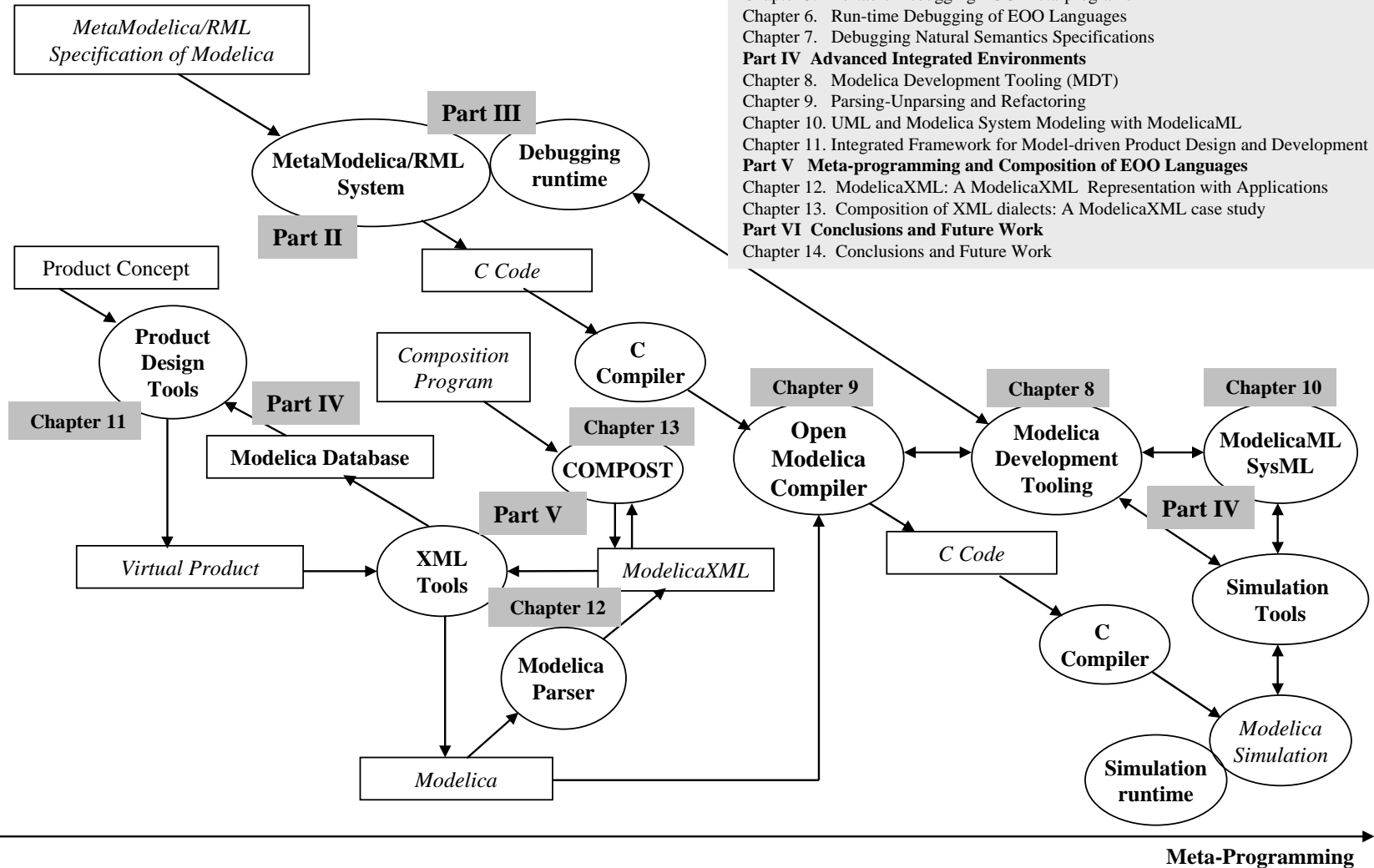
- The design, implementation, and evaluation of
 - a new, general, executable mathematical modeling and semantics meta-modeling language called MetaModelica. The MetaModelica language extends the existing Modelica language with support for meta-modeling, meta-programming, and exception handling
 - advanced portable debugging methods and frameworks for runtime debugging of MetaModelica and semantic specifications
 - several integrated model-driven environments supporting creation, development, refactoring, debugging, management, composition, serialization, and graphical representation of models in EOO languages. Additionally, an integrated model-driven product design and development environment based on EOO languages is also contributed
- Alternative representation of Modelica EOO models based on XML and UML/SysML are investigated and evaluated
- Transformation and invasive composition of EOO models has also been investigated

Thank you!
Questions?

<http://www.OpenModelica.org>

Thesis Structure

Meta-Modeling



Meta-Programming