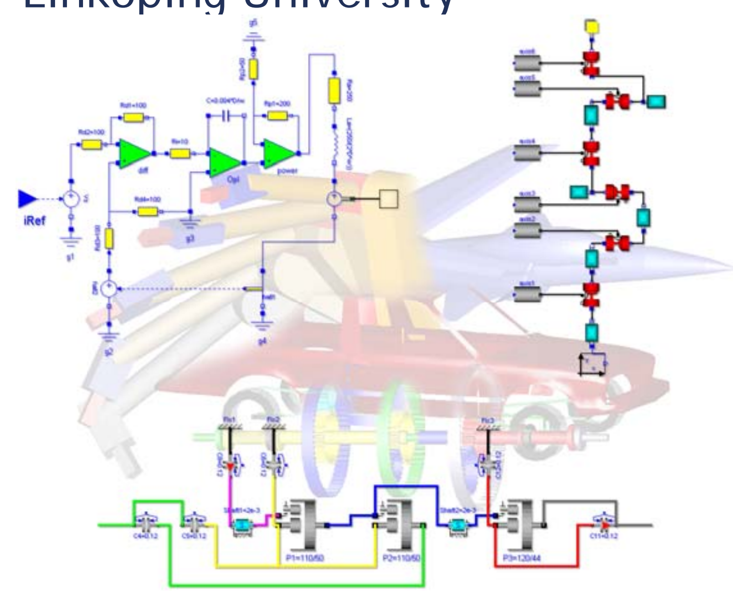
Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages

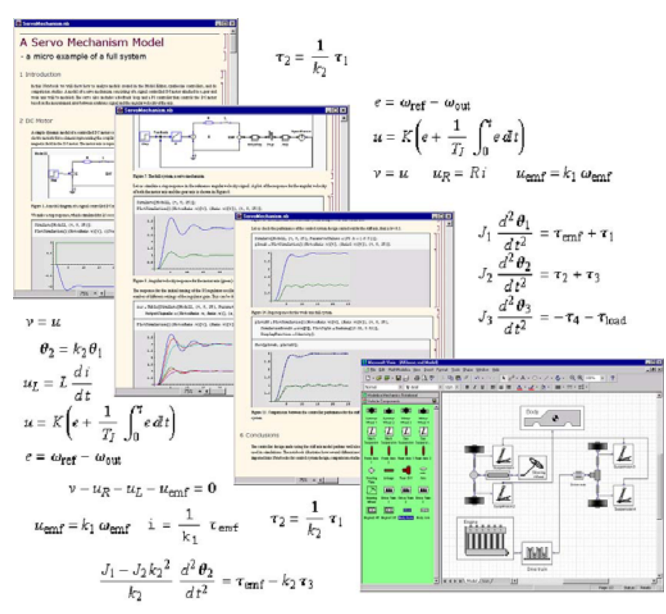
Adrian Pop

Programming Environment Laboratory

Department of Computer and Information Science

Linköping University





2008-06-05

Outline

- 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

Thesis Motivation

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.

Research Questions

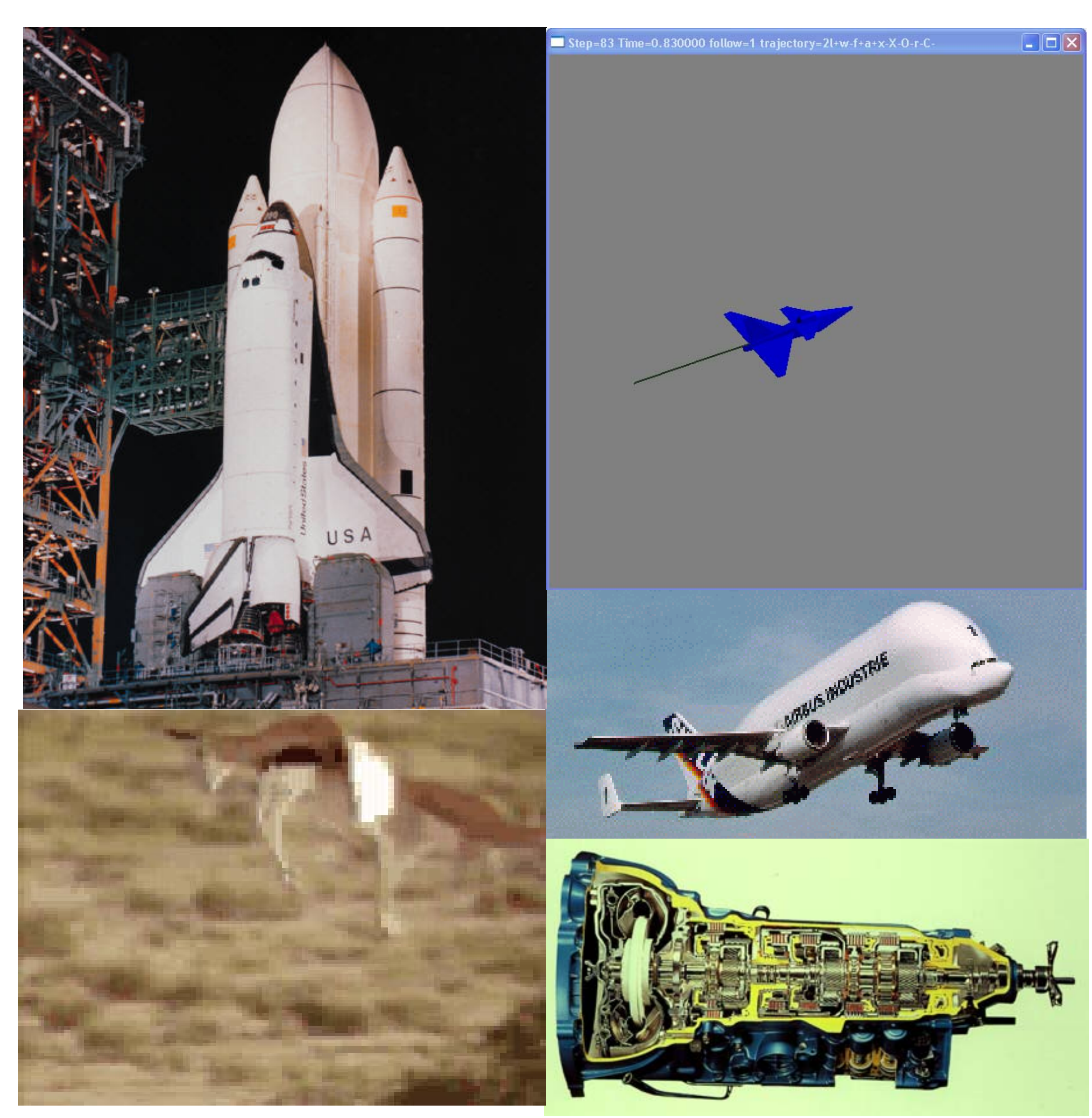
- Can we deliver a new language that allows people to build their own solution to their problems without having to go via tool vendors?
- What is expected from such a language?
- What properties should the language have based on the requirements for it? This includes language primitives, type system, 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?

Outline

- 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

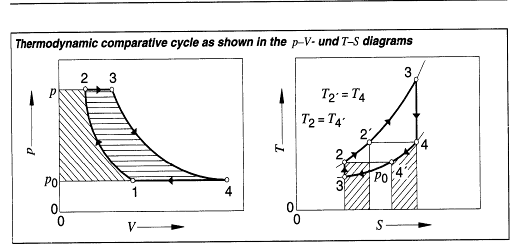
Examples of Complex Systems

- 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



Thermodynamic comparative cycle as shown in the p–V- und T–S diagrams

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:

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

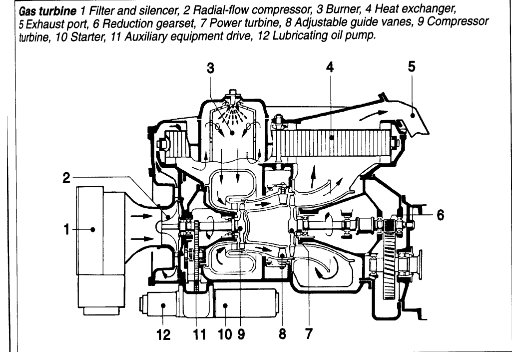
Where $p_2/p_1 = (T_2/T_1)^{\frac{\chi}{\chi-1}} = (T_3/T_4)^{\frac{\chi}{\chi-1}}$ and $T_4 = T_3 \cdot (T_1/T_2)$ thus

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

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 impressæ, & fieri secundum lineam rectam qua vis illa imprimitur.

The Form – Equations

- 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 = INTEG(F)/m
```

Programming languages usually do not allow equations!

Modelica

- 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*?

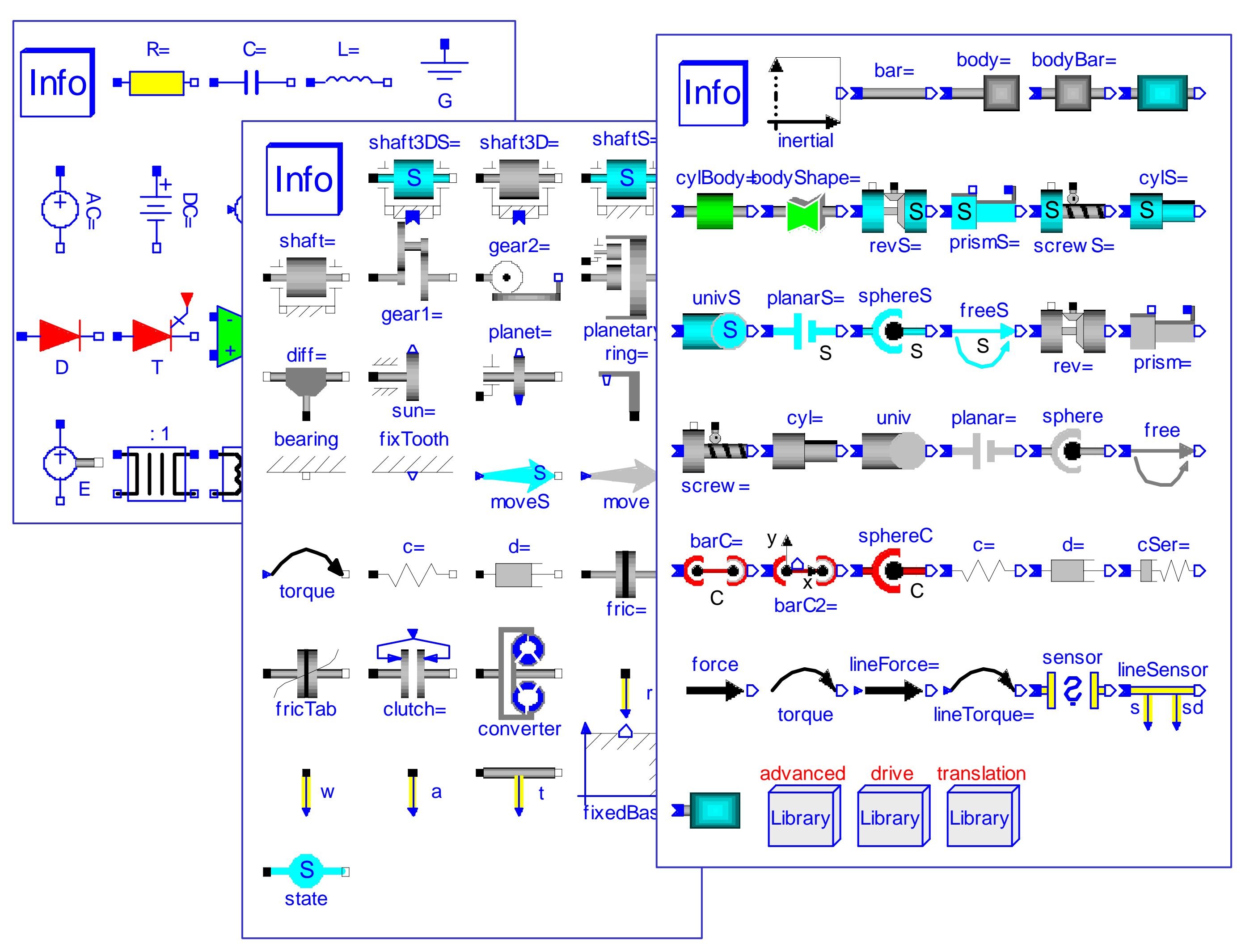
  The acausality makes Modelica library classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.

- Example: a resistor *equation*:

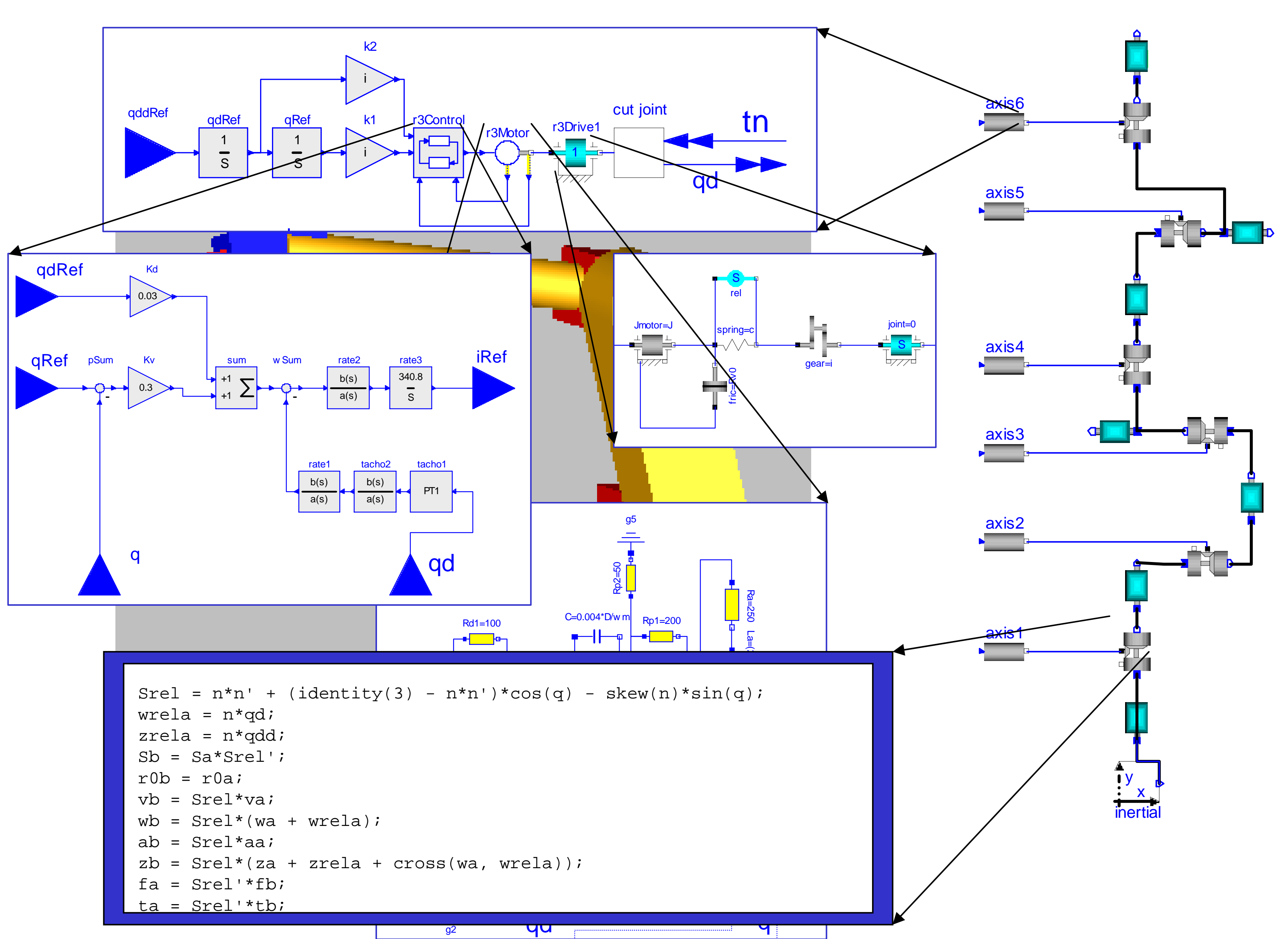
  R*i = v;
- can be used in three ways:

  i := v/R;
  v := R*i;
  R := v/i;

Modelica - Reusable Class Libraries



Hierarchical Composition Diagram



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
```

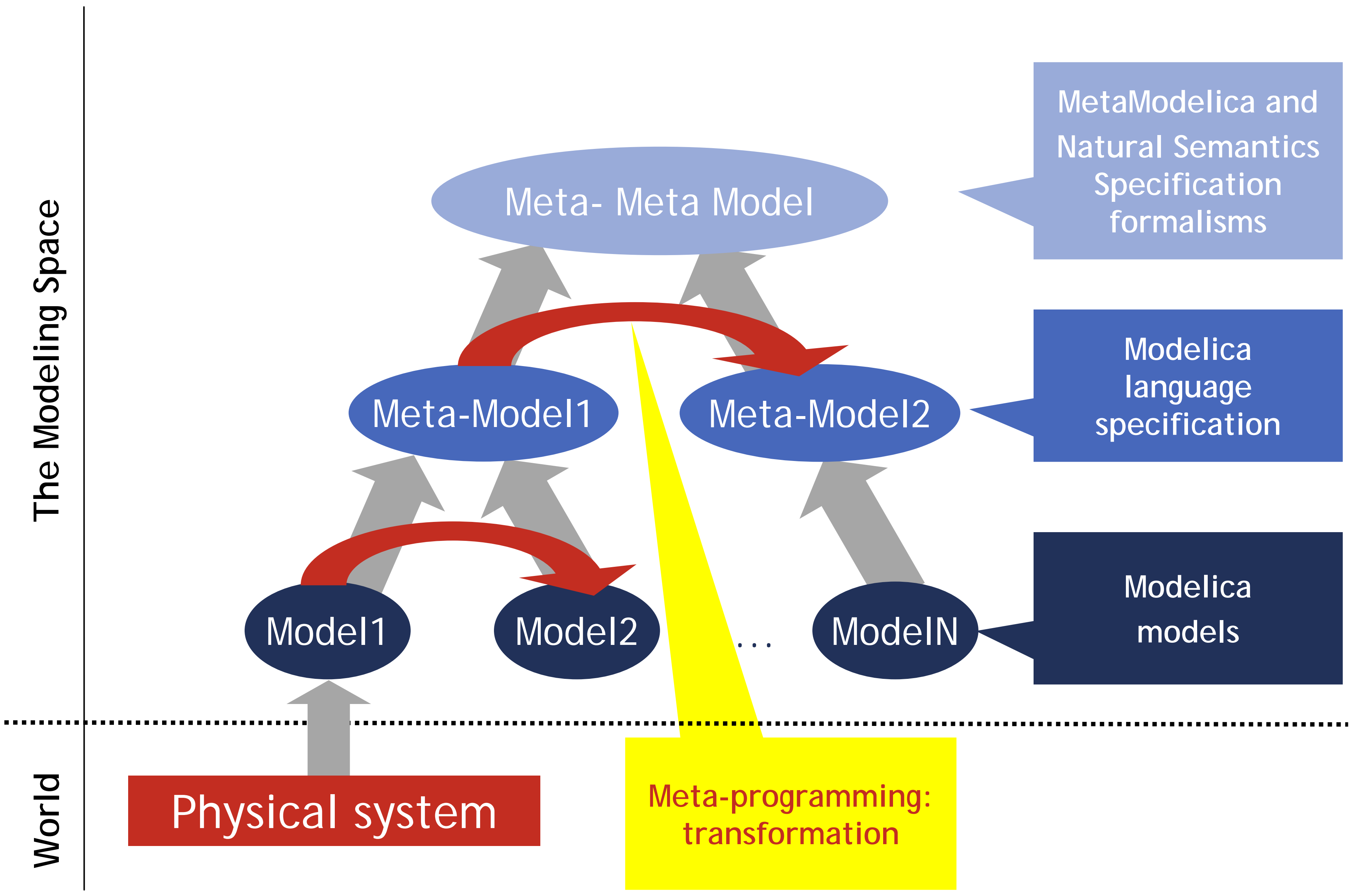
R
L
EM
DC
load
G

Outline

- 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



MetaModelica - Context

- 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*

MetaModelica - Motivation

- 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*

MetaModelica extensions to Modelica (I)

- 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

MetaModelica extensions to Modelica (II)

- 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;
  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;
```

PLUS
RCONST
MUL
12
RCONST
RCONST
5
13

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;
```

MetaModelica Compiler Prototype

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

OpenModelica Bootstrapping

- 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

Outline

- 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

Debugging EOO Languages

- 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_rettype _bubbleSort(real_array unordElem)
{
  size_t tmp2;
  bubbleSort_rettype 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)

Debugging - Performance Evaluation (II)

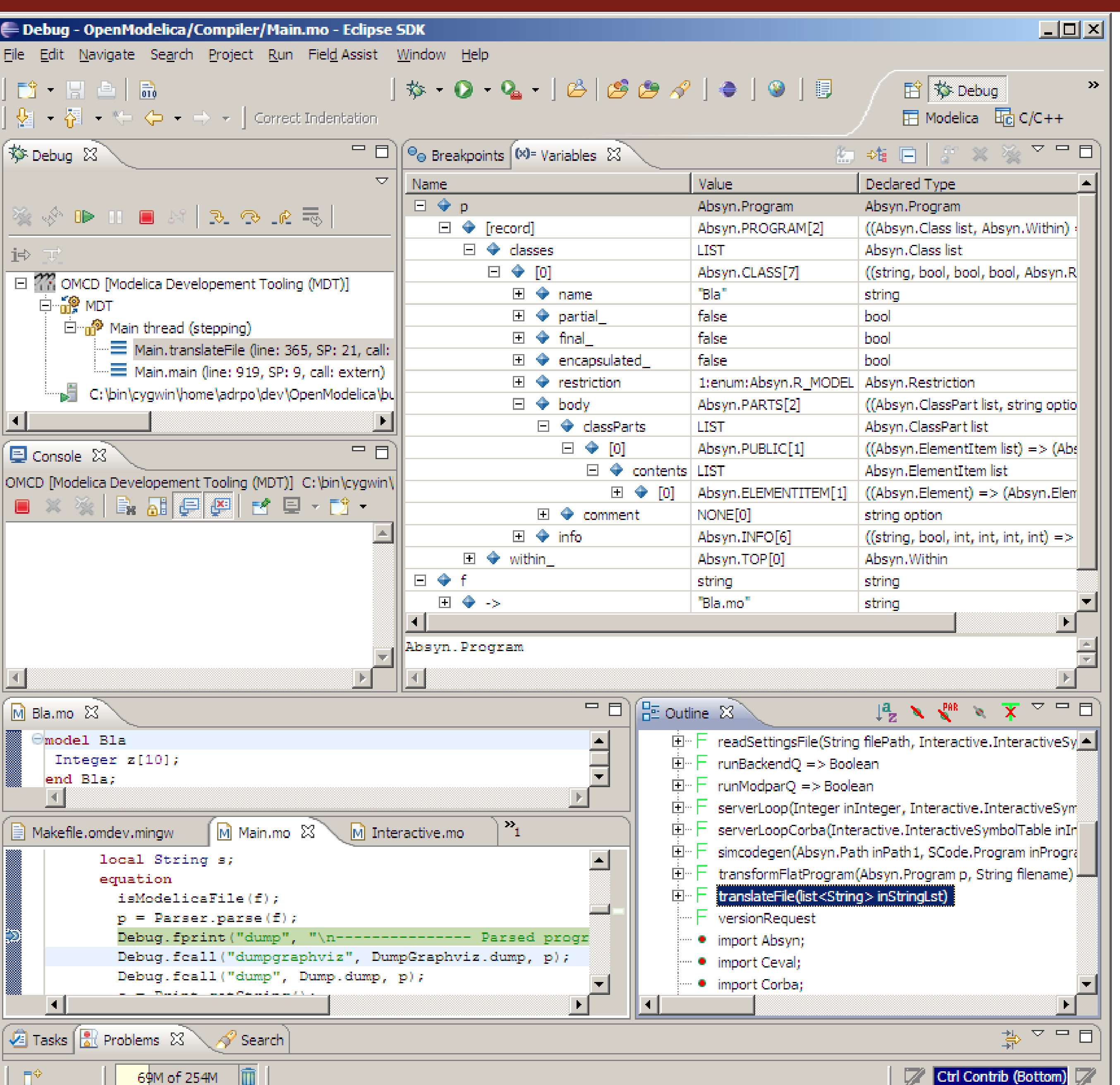
The test case

- RRLargeModel2.mo - *model with 1659 equations/variables*

- Execution time for the OpenModelica Compiler while checking RRLargeModel2.mo

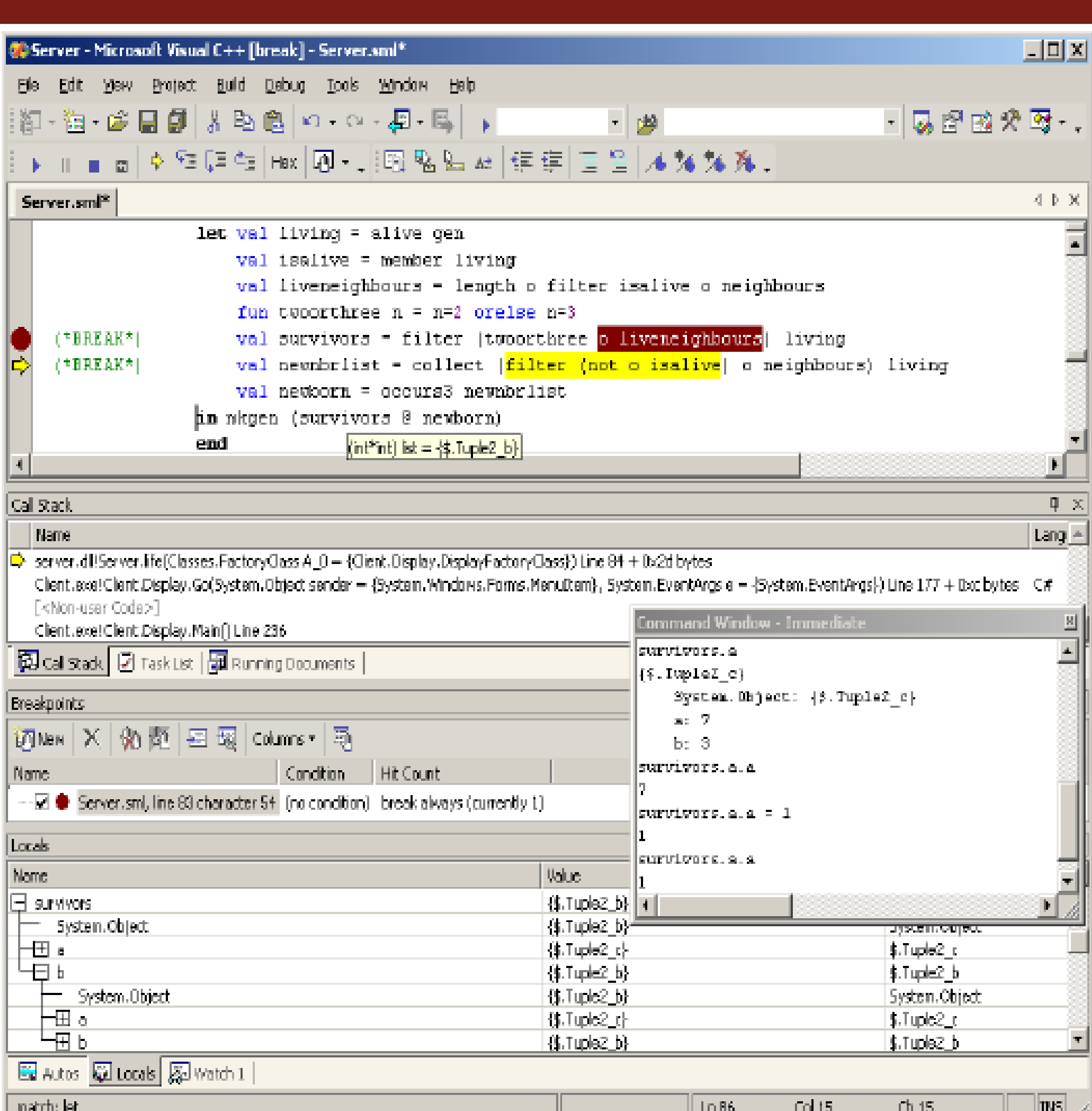
No debugging	223.01 (s)
Early instrumentation	5395.47 (s)
Late instrumentation	864.36 (s)

Eclipse Debugging Environment



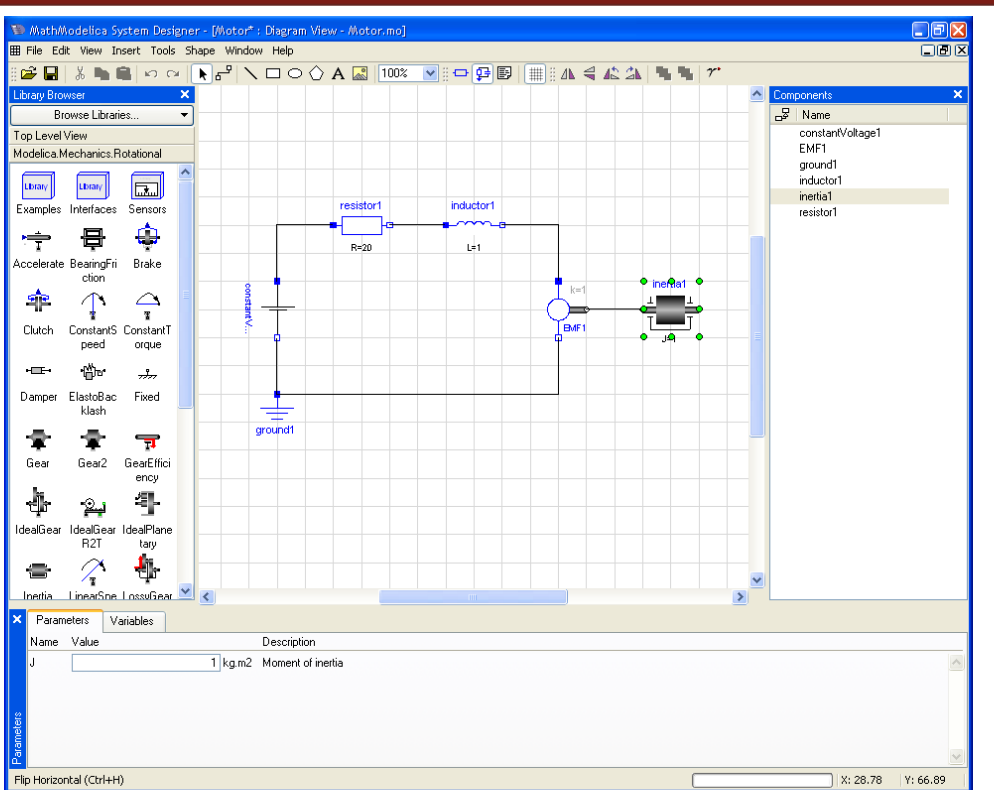
- Type information for all variables
- Browsing of complex data structures

SML.NET Debugger



- No type information for variables

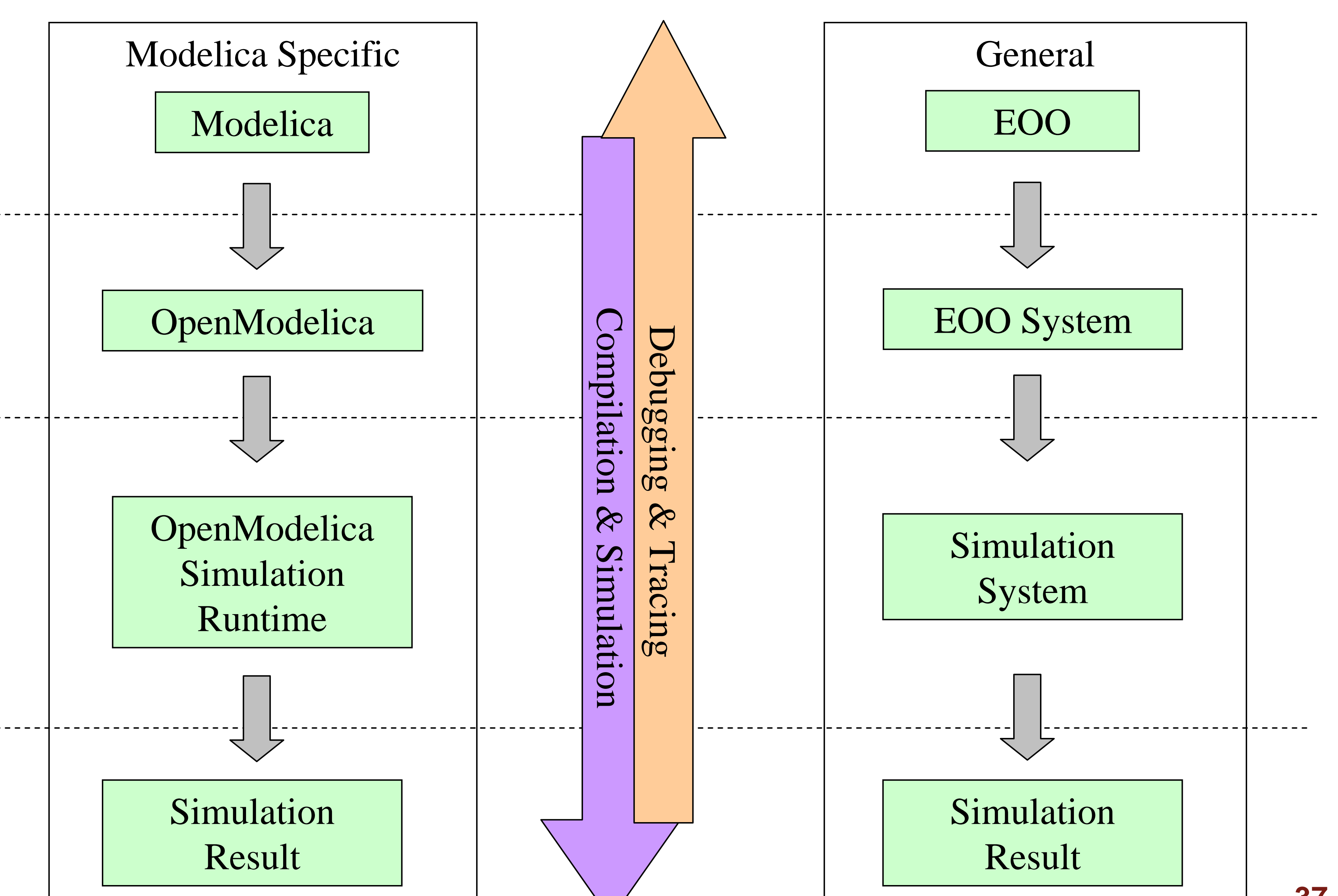
Why do we need Equation-based debugging?



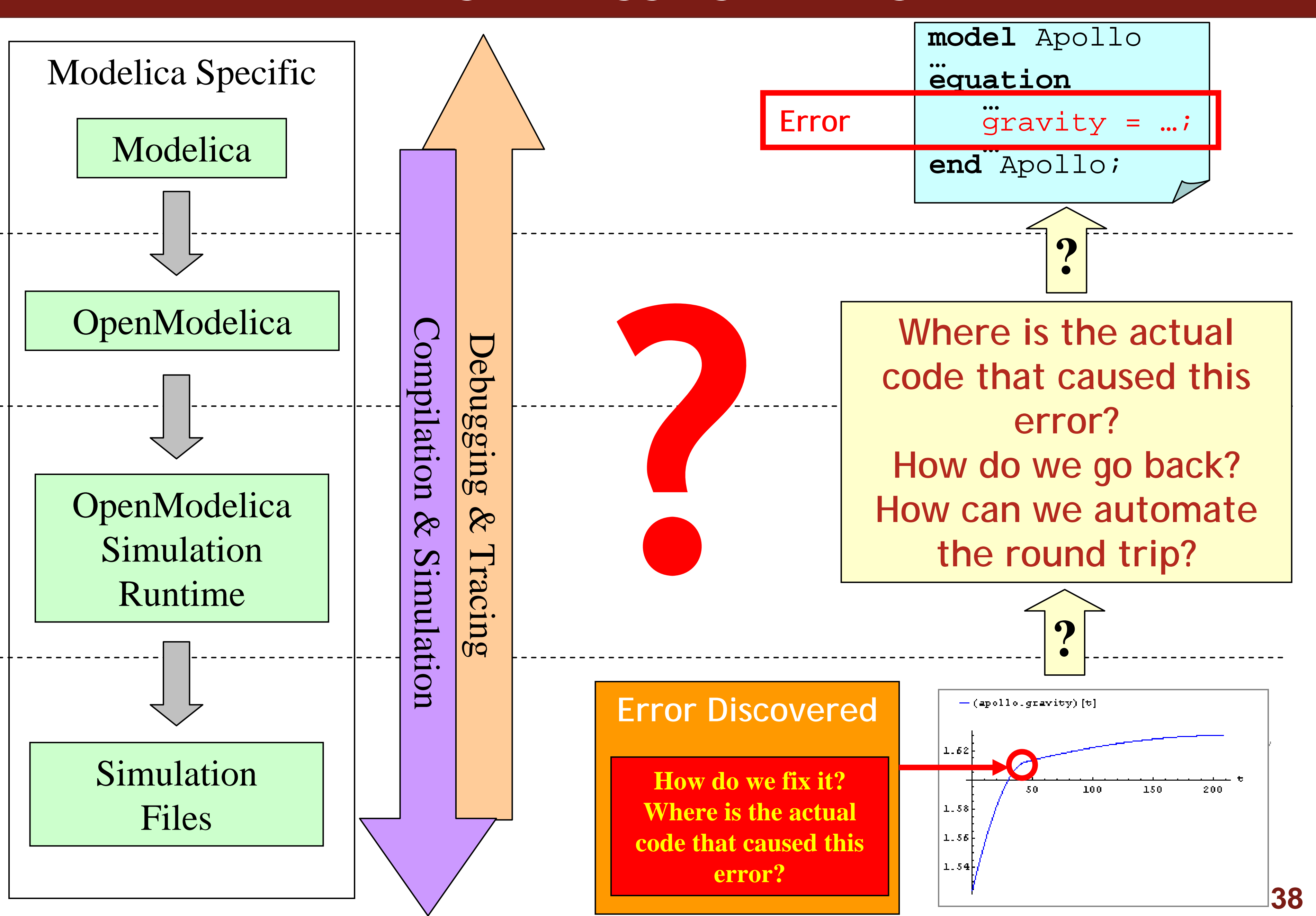
- Easy to build large systems
  - Drag and Drop composition
  - Hierarchical Modeling
- Model behavior depends on data from various sources (xml, databases, files, etc)
- Models could be external (Hardware in the loop, co-simulation, etc)

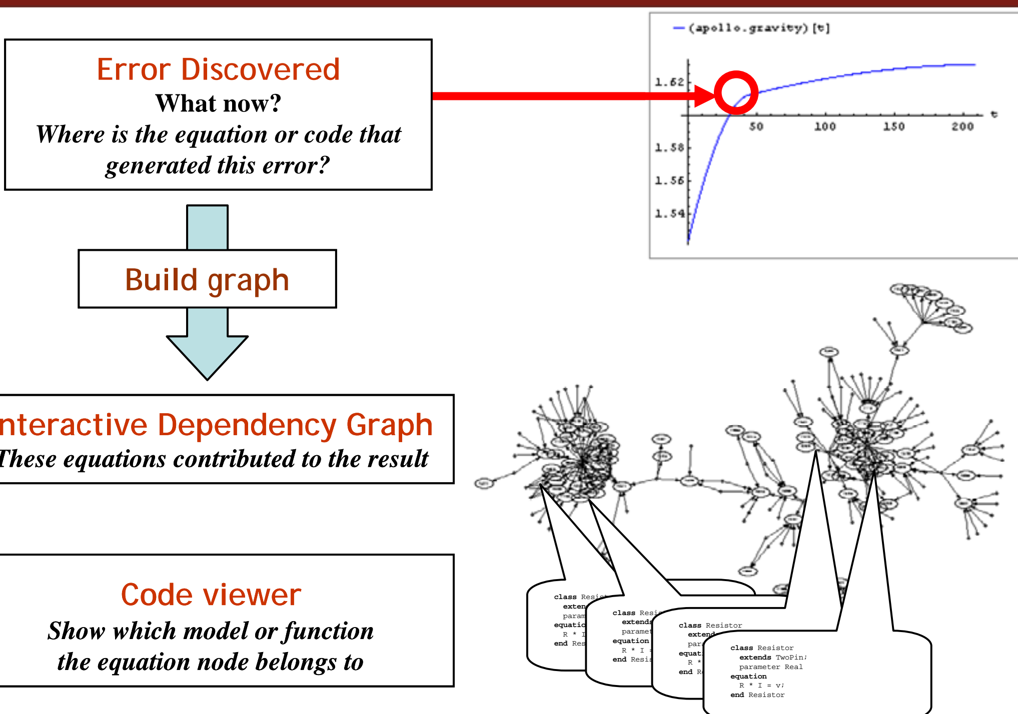
- You build your model by connecting components together
- You simulate (hopefully there are no compilation errors)
- The result you get back is wrong!
  - Why is the result wrong?
  - Where is the error?
  - How can I pin-point the error?

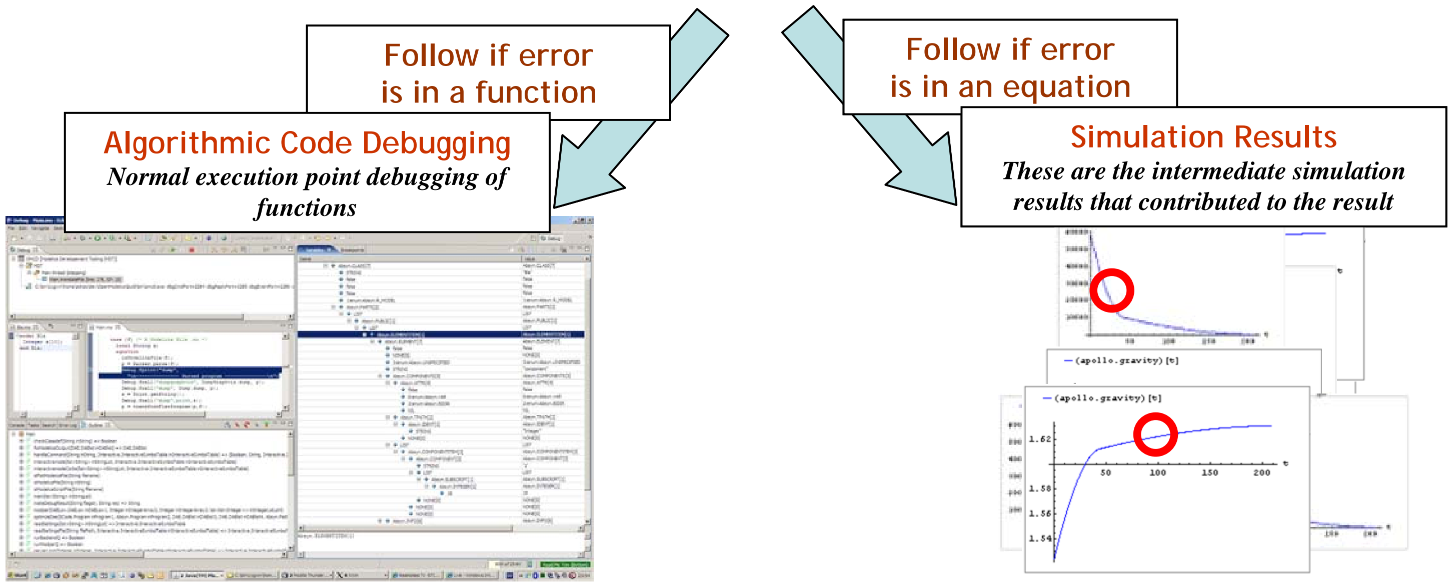
Translation process



Existing Debugging Strategies Do Not Suffice

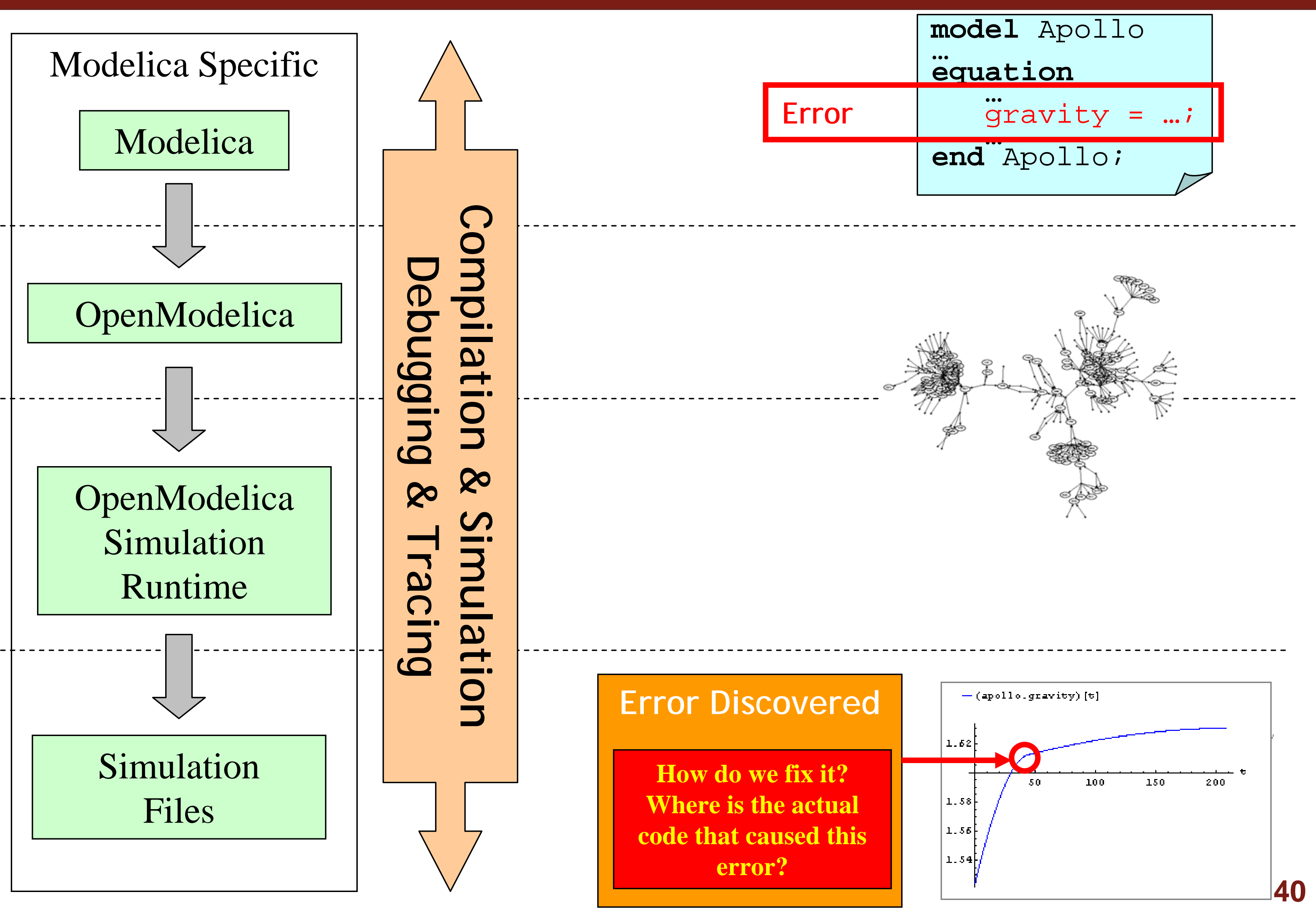






- Mark the error
- Build an interactive graph containing the evaluation
- Walk the graph interactively to find the error

Debugging Strategy: Compiling With Debugging In Mind



Translation Phases with Debugging

- Include debugging support within the translation process

Debugging Translation Process Additional Steps

Save element position

Save element origin (model and position)

Save equation elements origin (model and position)

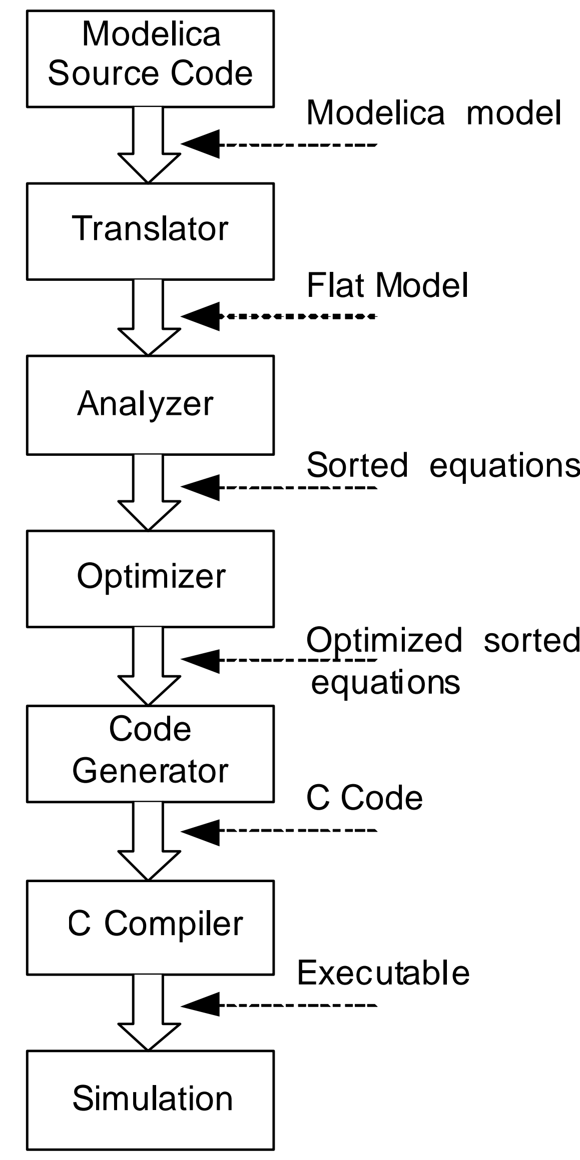
Save the optimizer transformations changes

Save all the available origin information

Executable with all the available origin information

Simulation with run-time debugging functionality

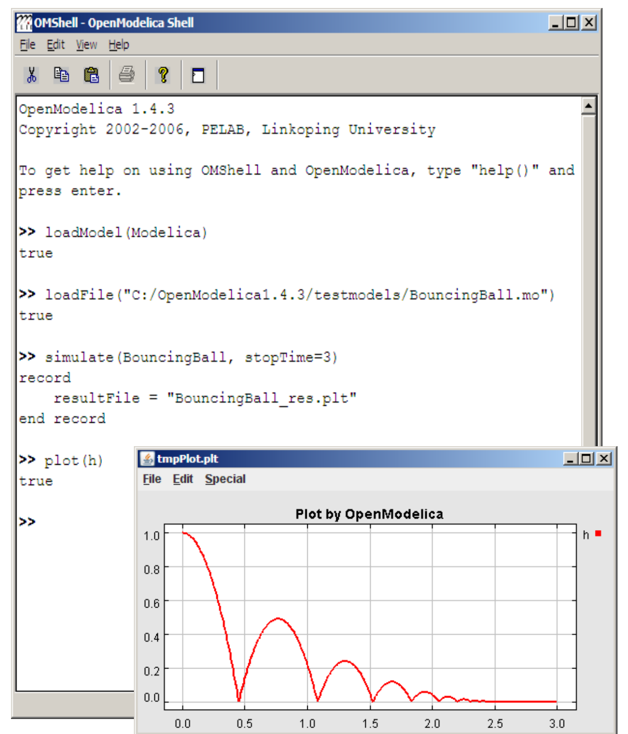
Normal Translation Process

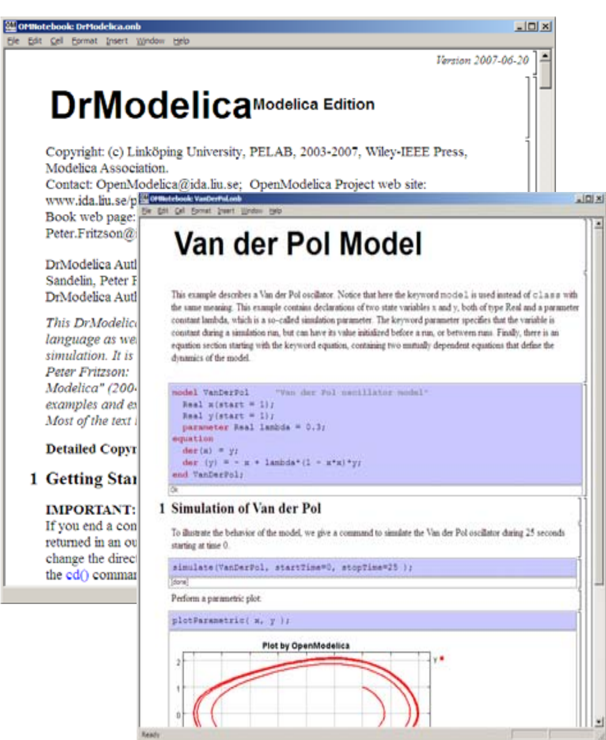


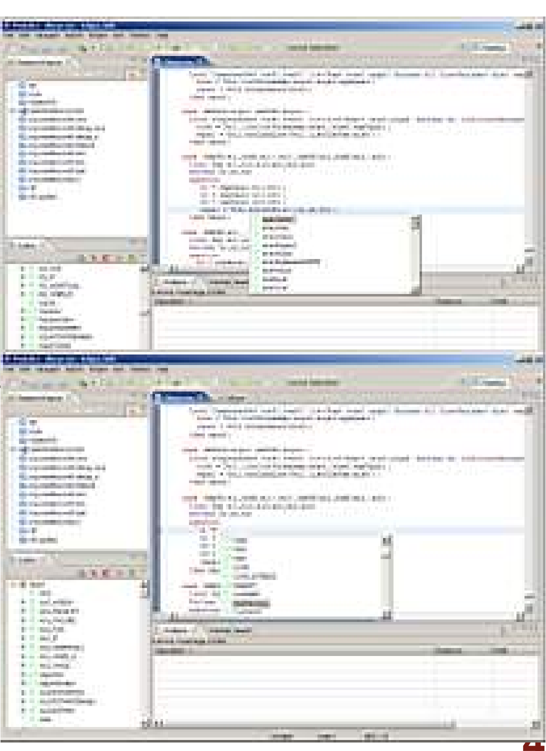
Outline

- 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

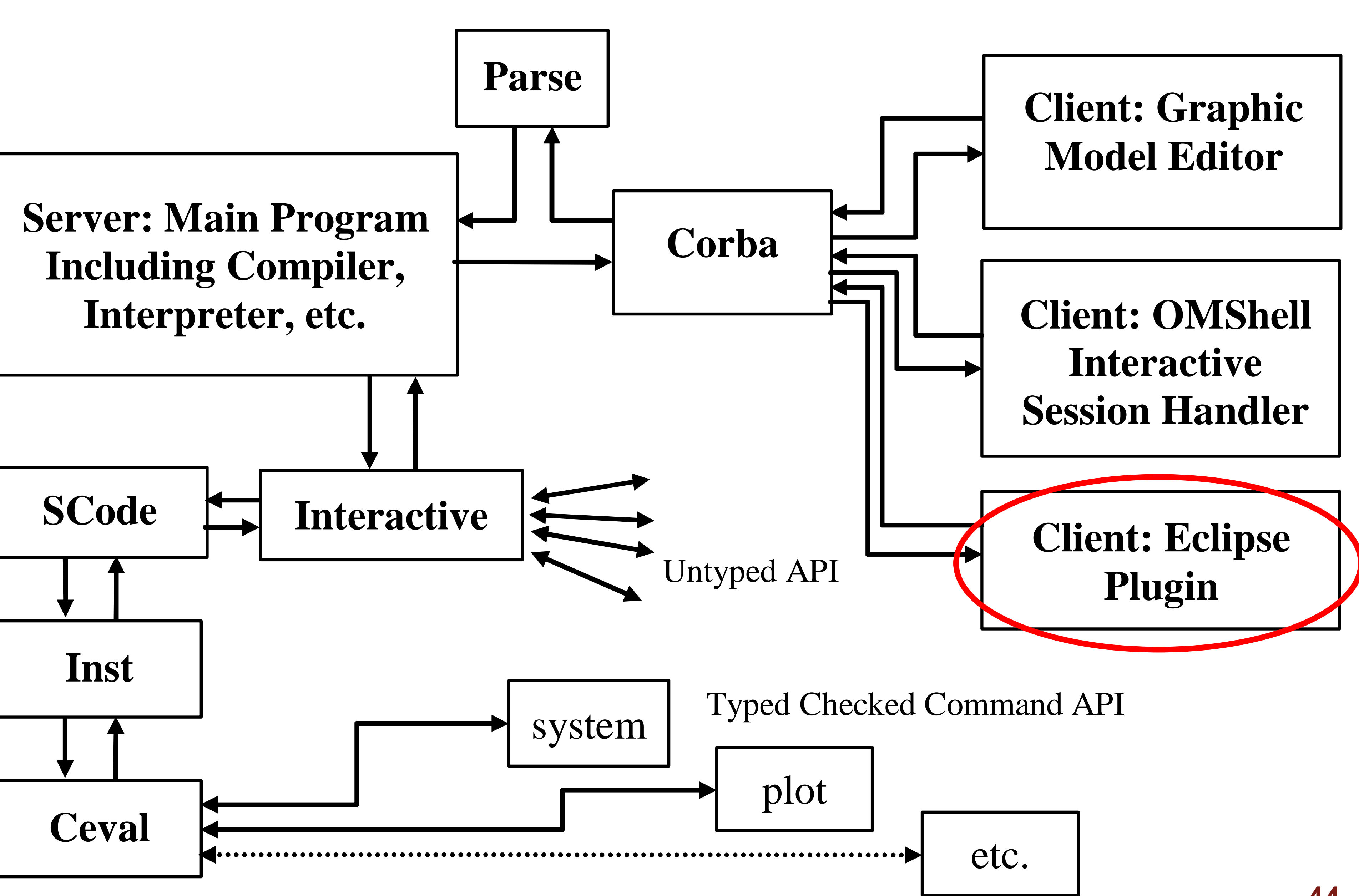
- Advanced Interactive Modelica compiler (OMC)
  - Supports most of the Modelica Language
- Basic environments for creating models
  - OMShell – an interactive command handler
  - OMNotebook – a literate programming notebook
  - MDT – an advanced textual environment in Eclipse







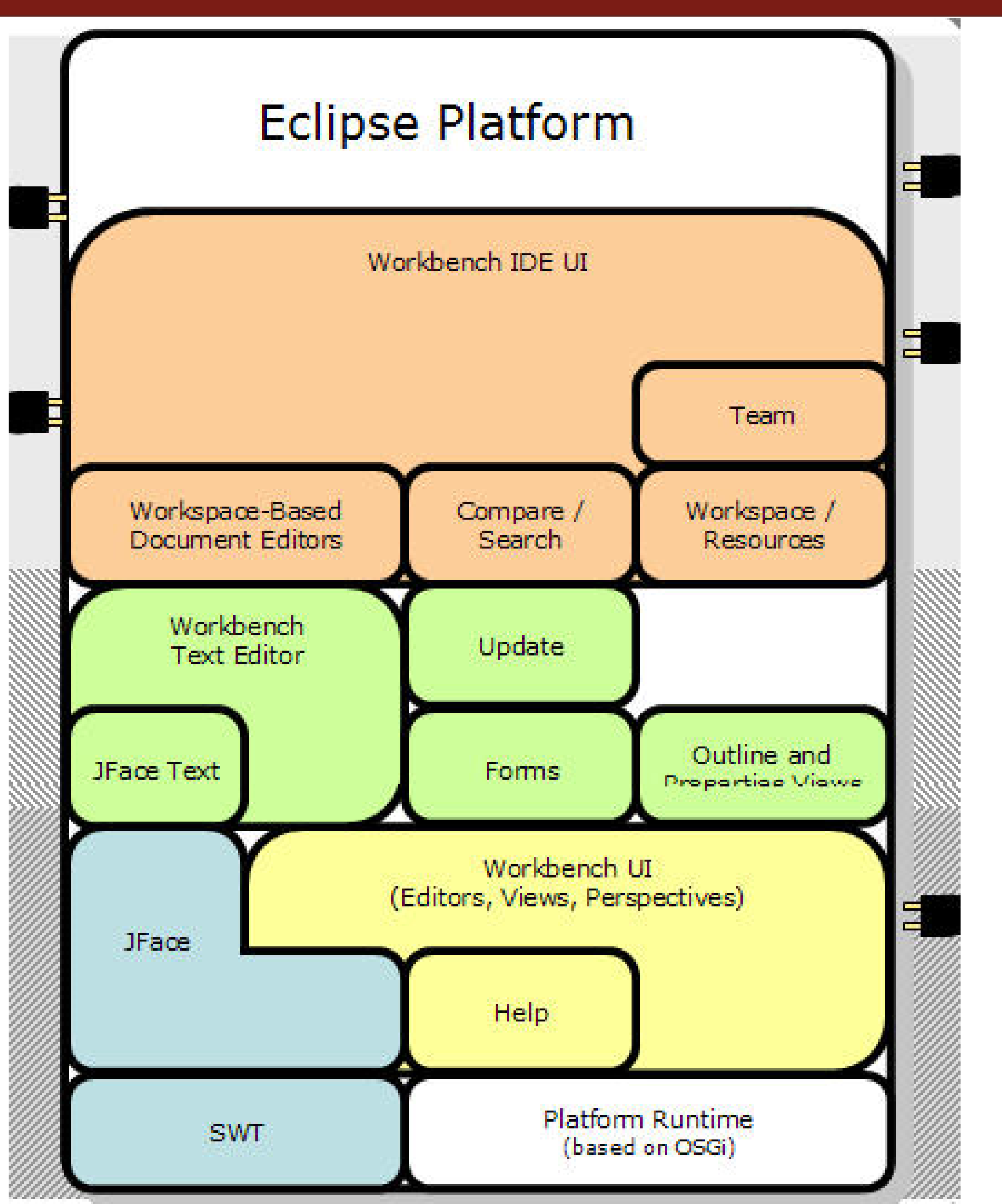
OpenModelica Context



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

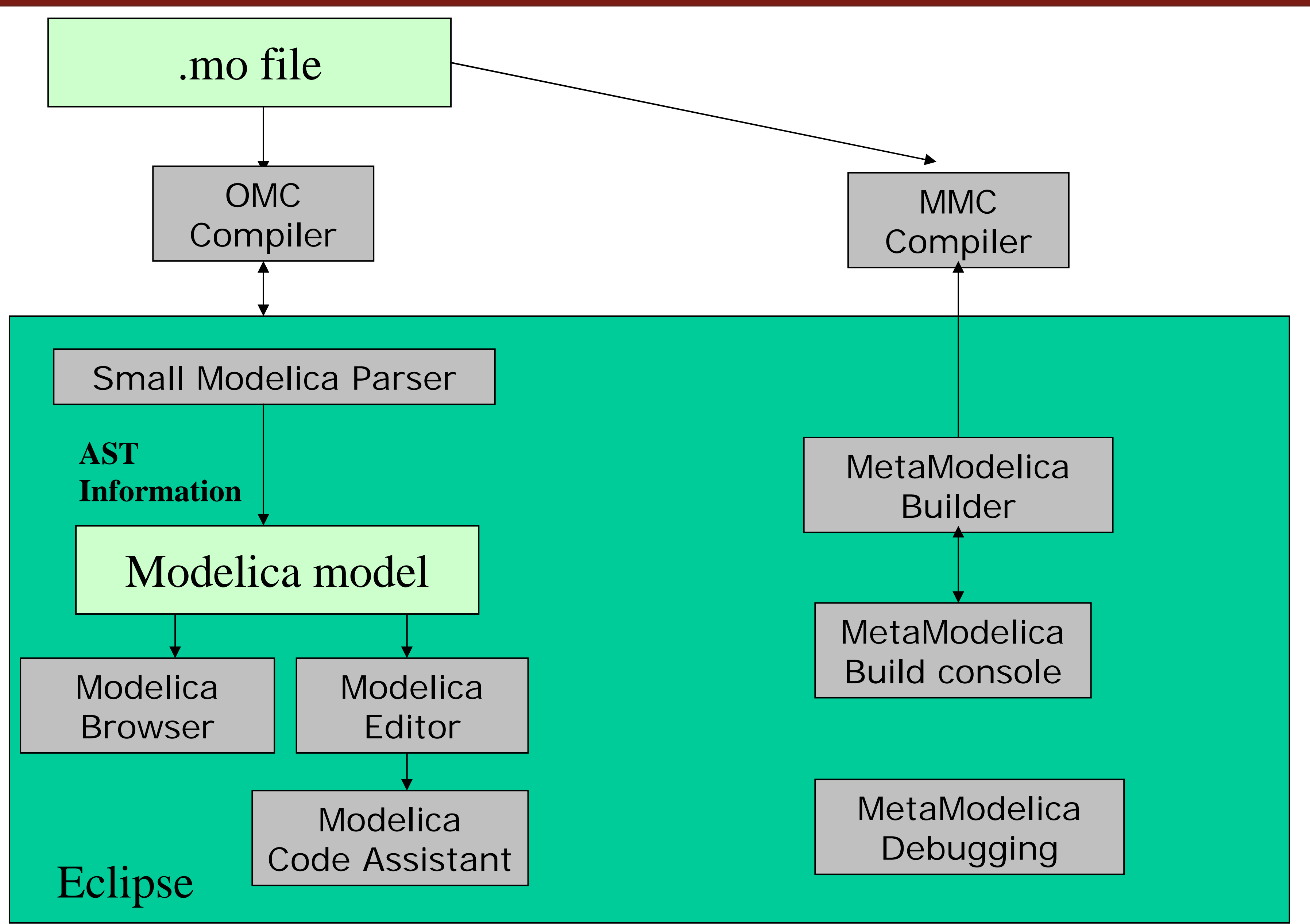
Modelica Editor

Modelica Code Assistant

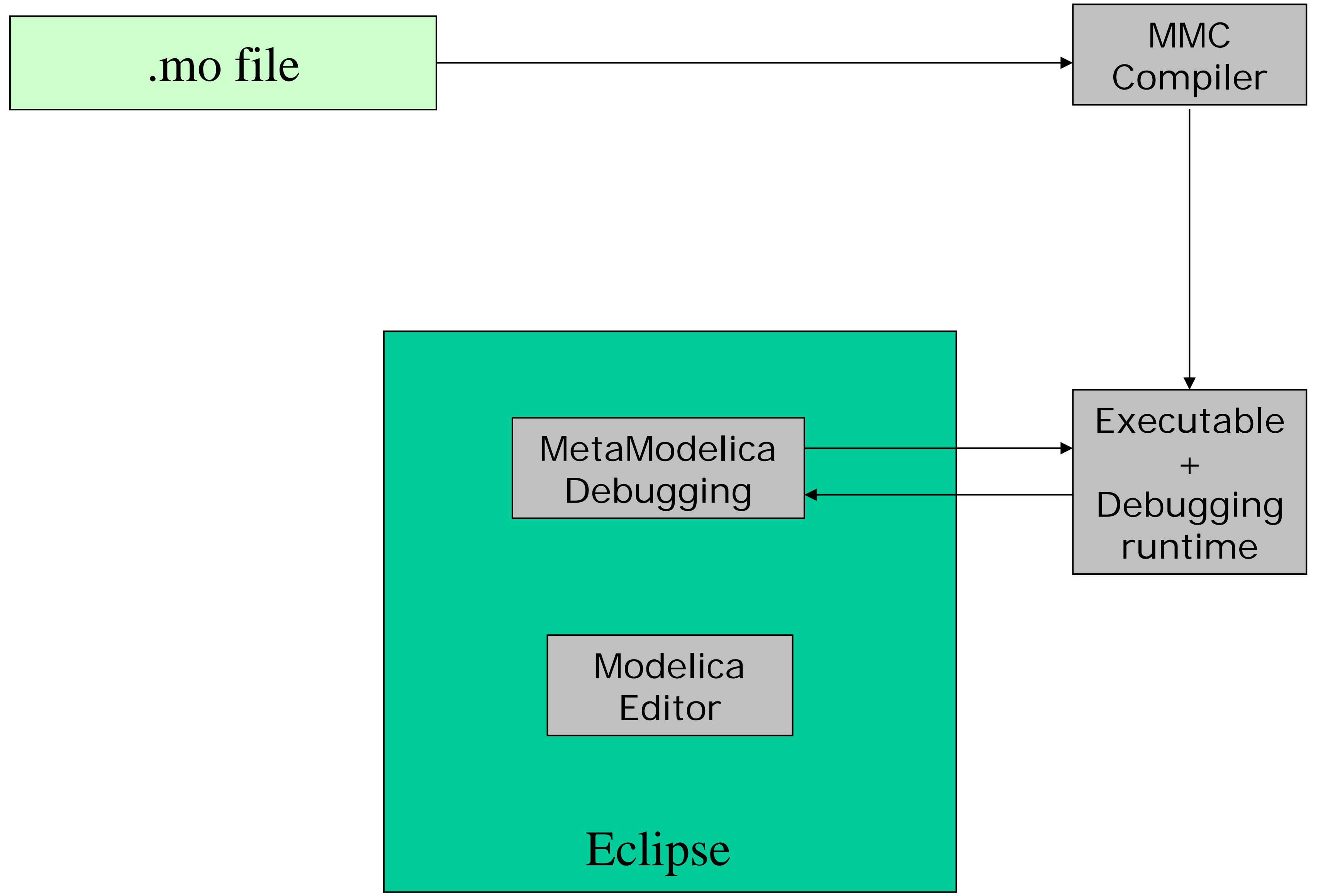
MetaModelica Debugging

Modelica Perspective

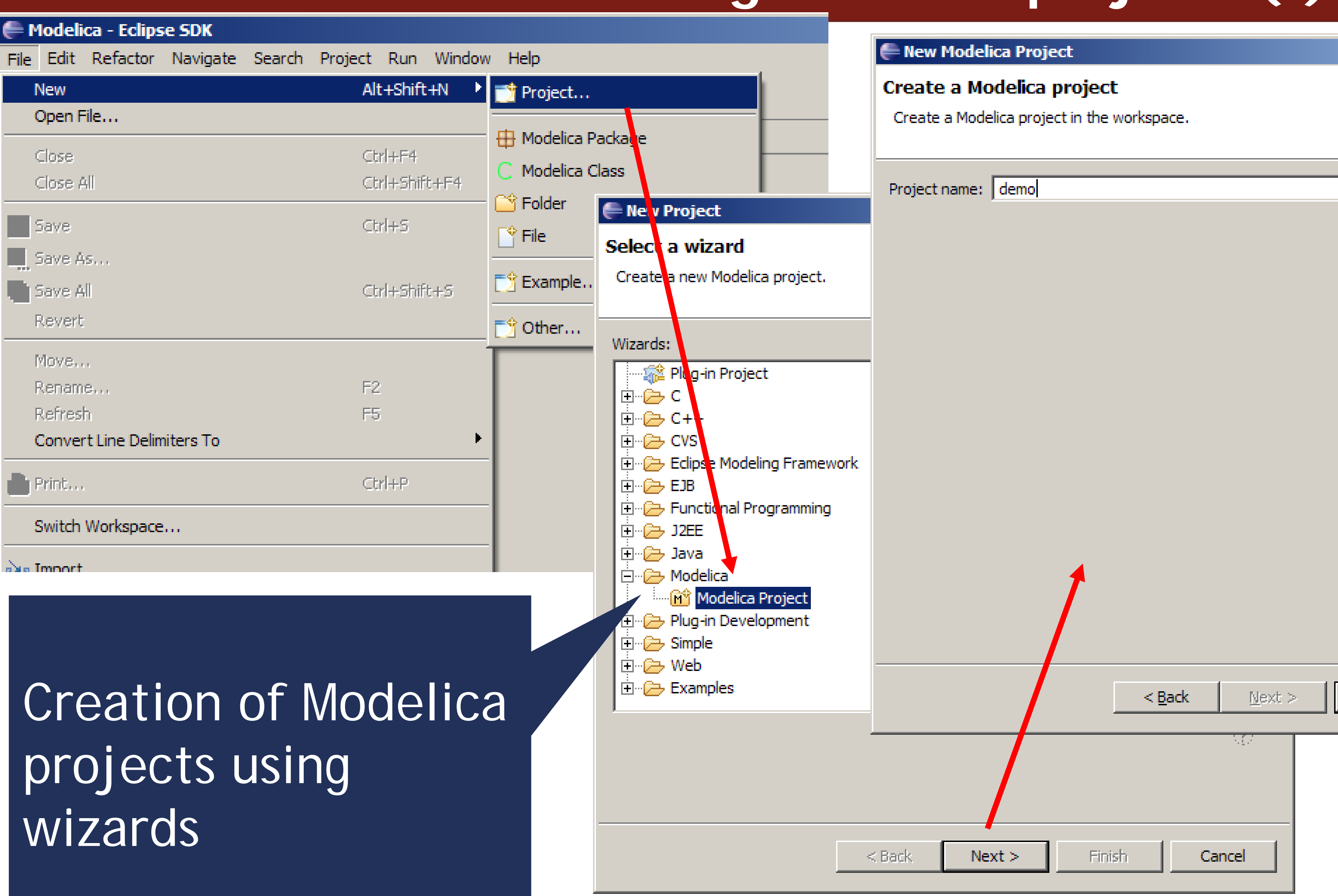
The MDT Eclipse Environment (II)



The MDT Eclipse Environment (III)

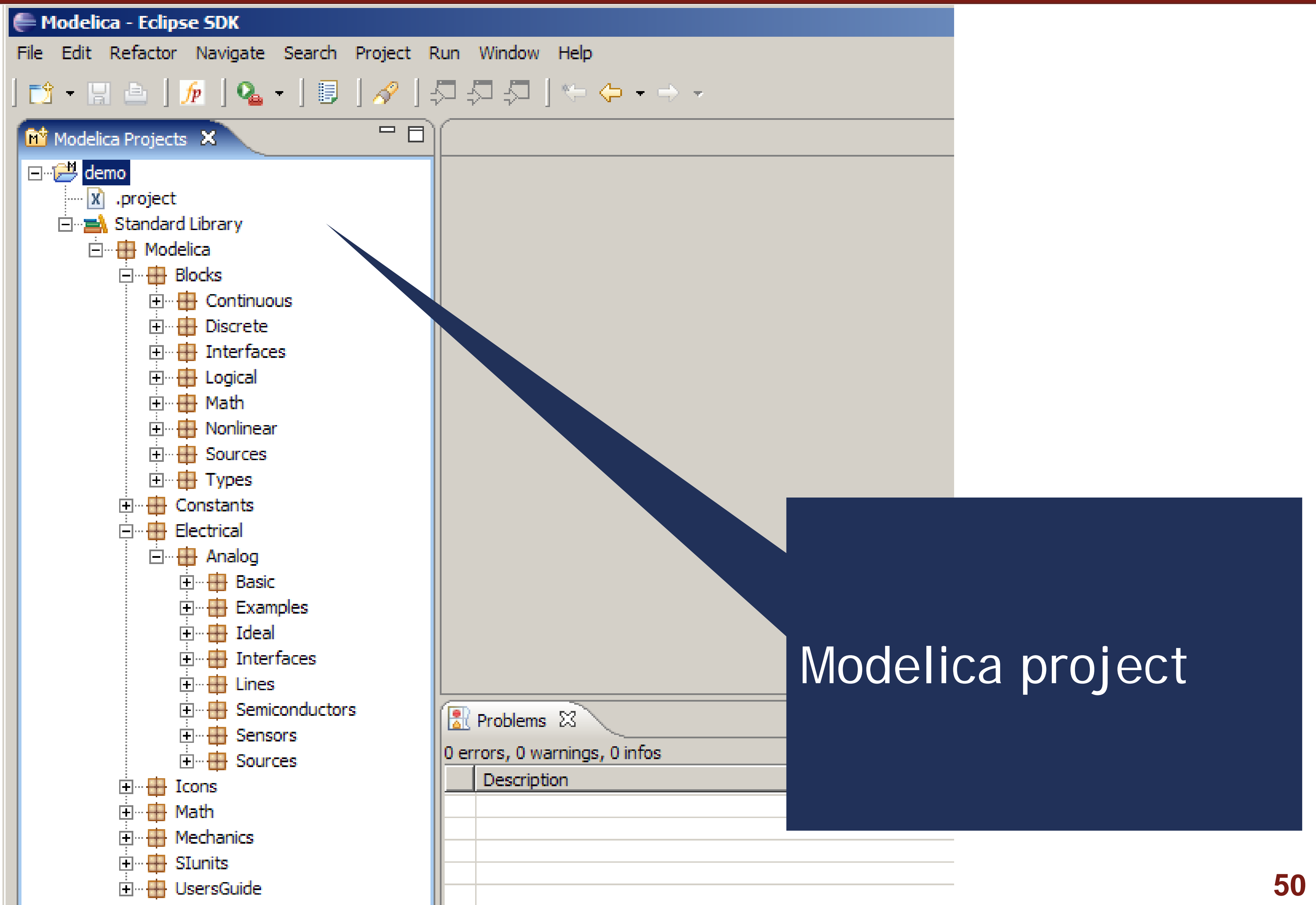


Creating Modelica projects (I)

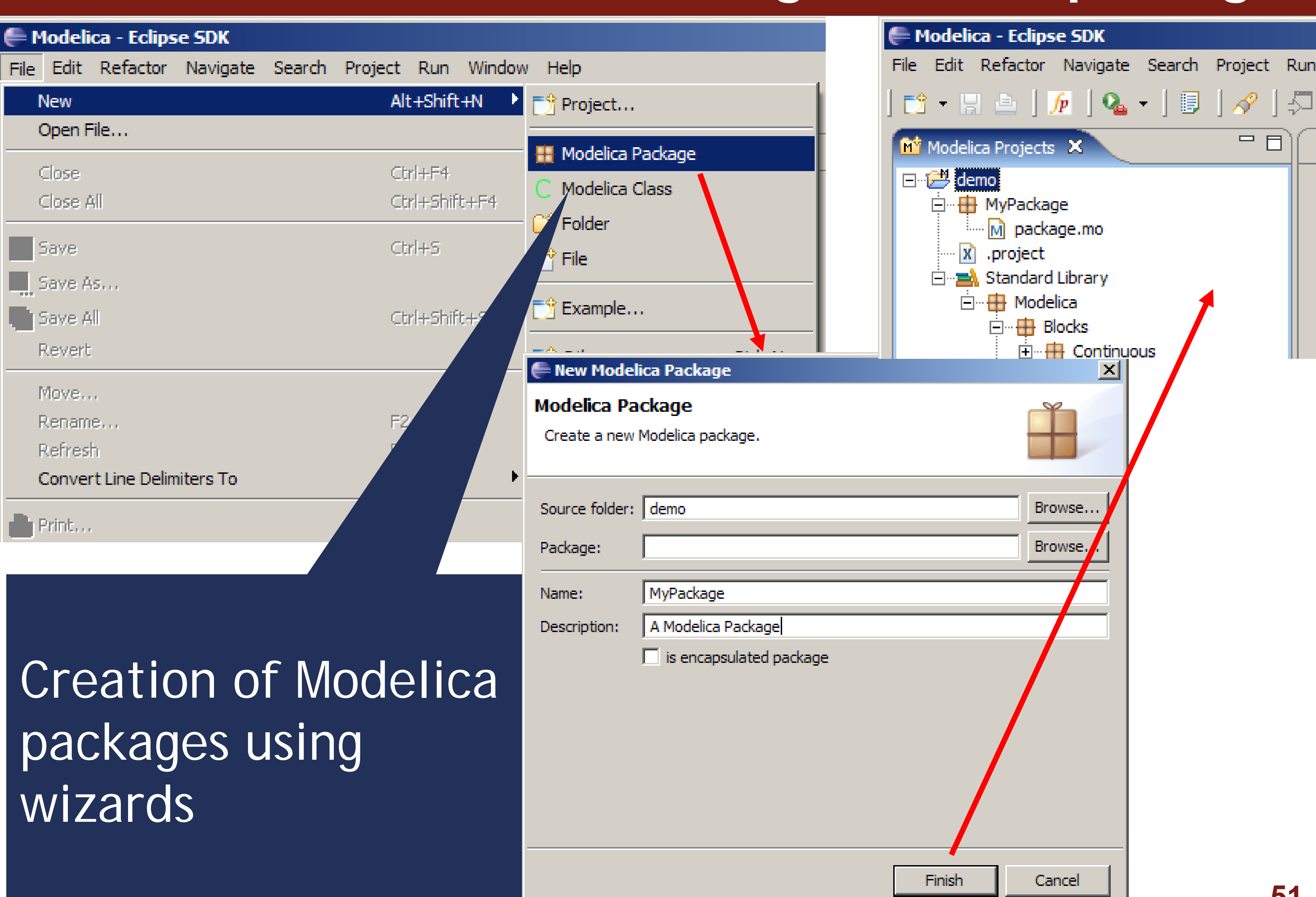


Creation of Modelica projects using wizards

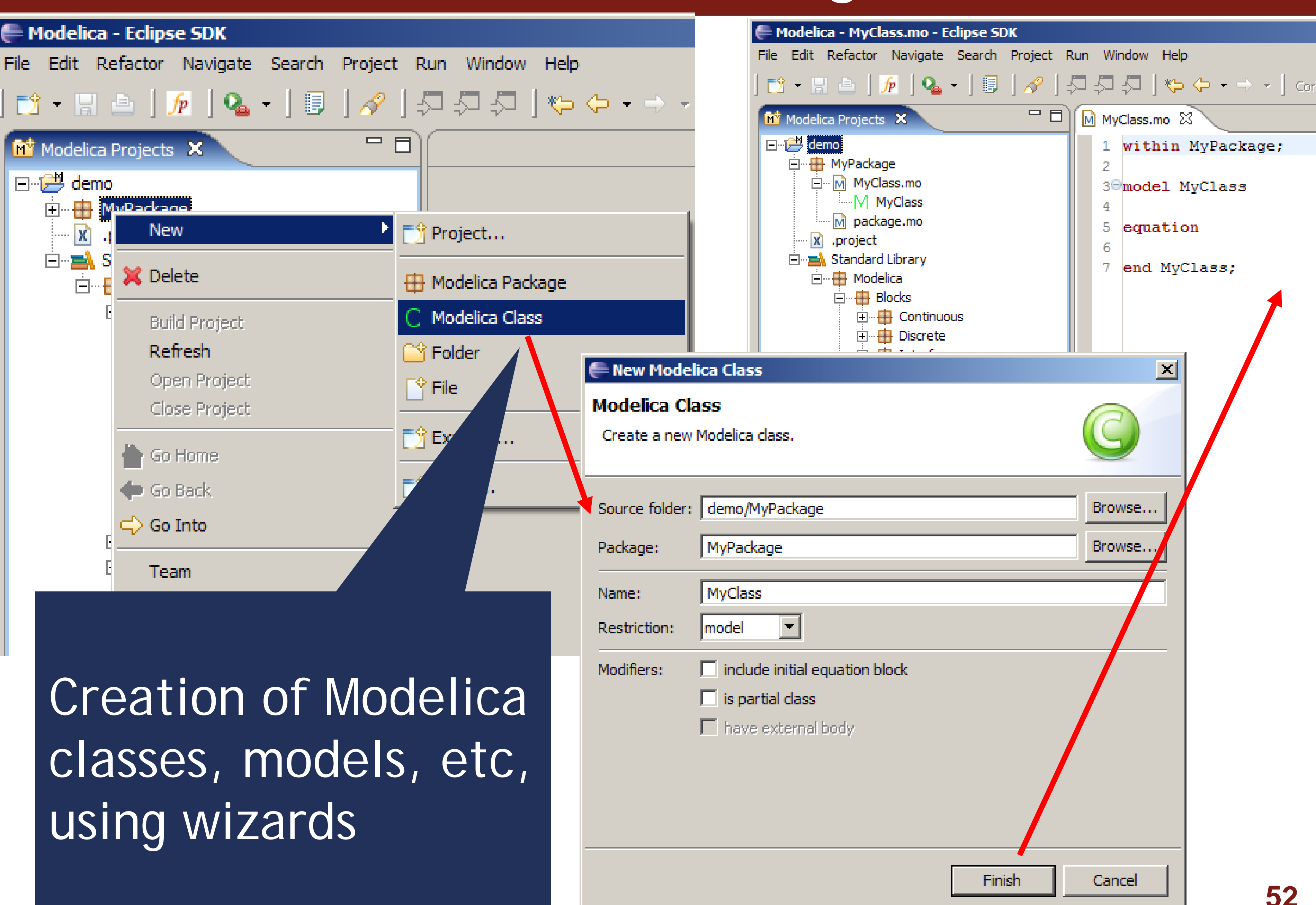
Creating Modelica projects (II)

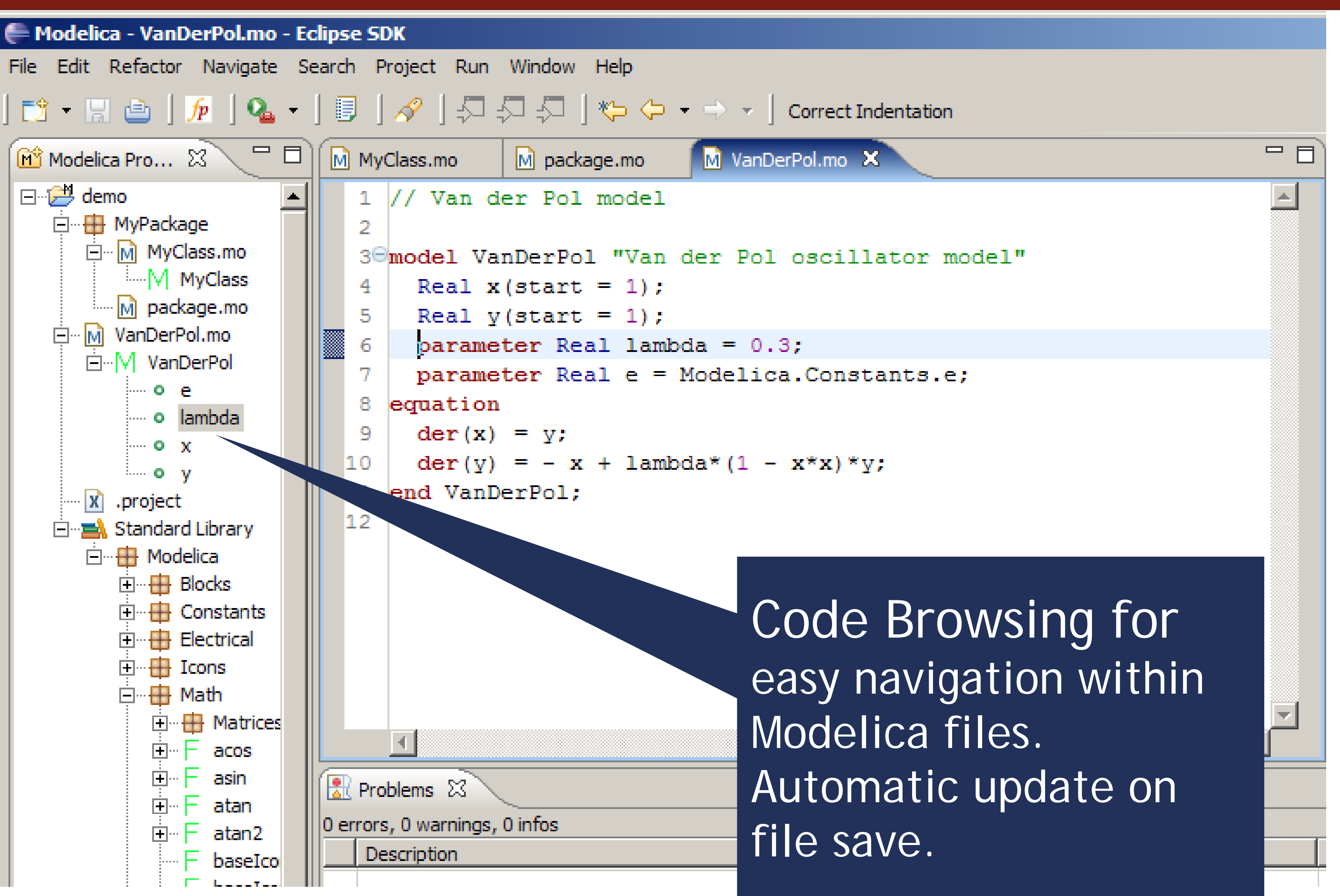


Creating Modelica packages

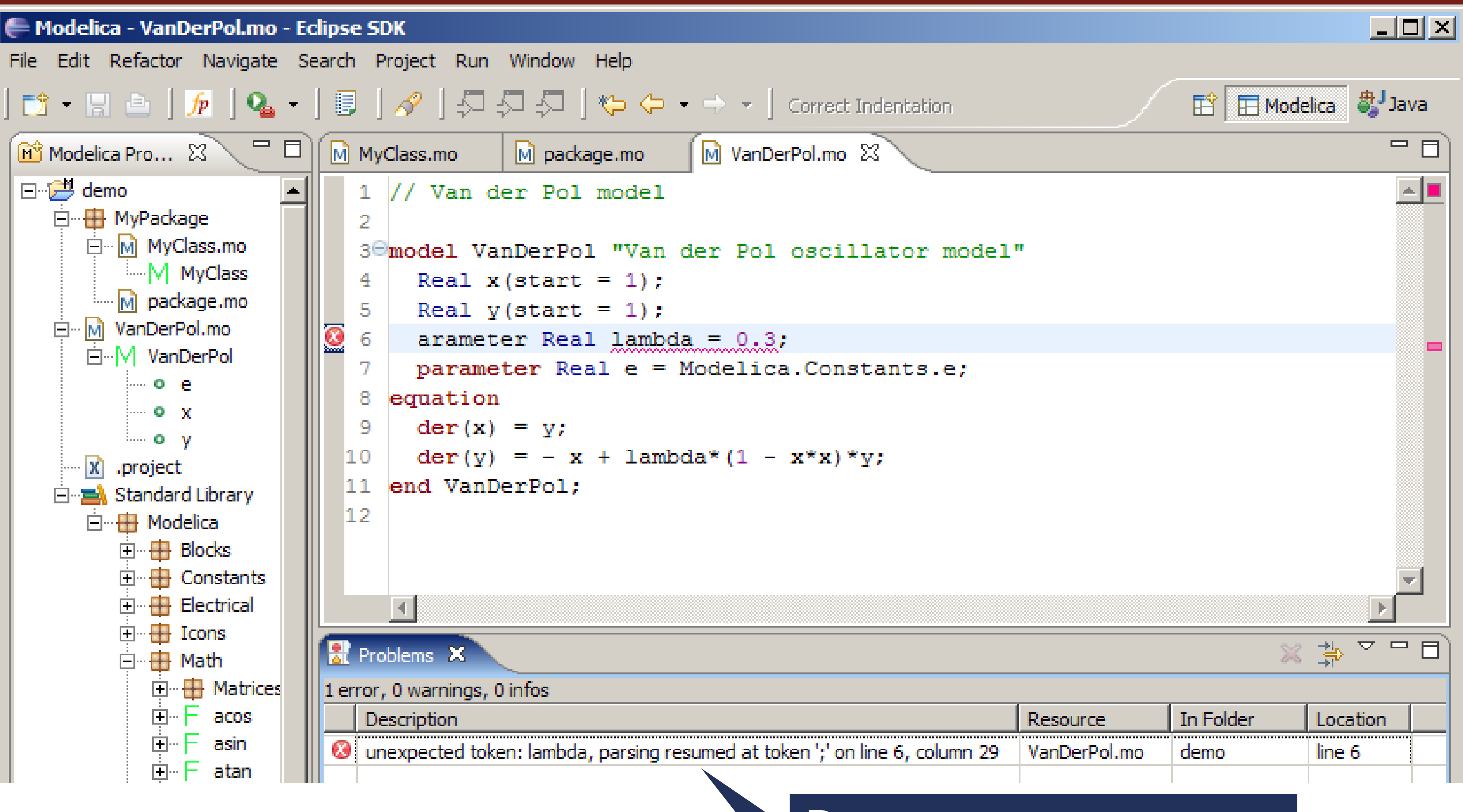


Creating Modelica classes



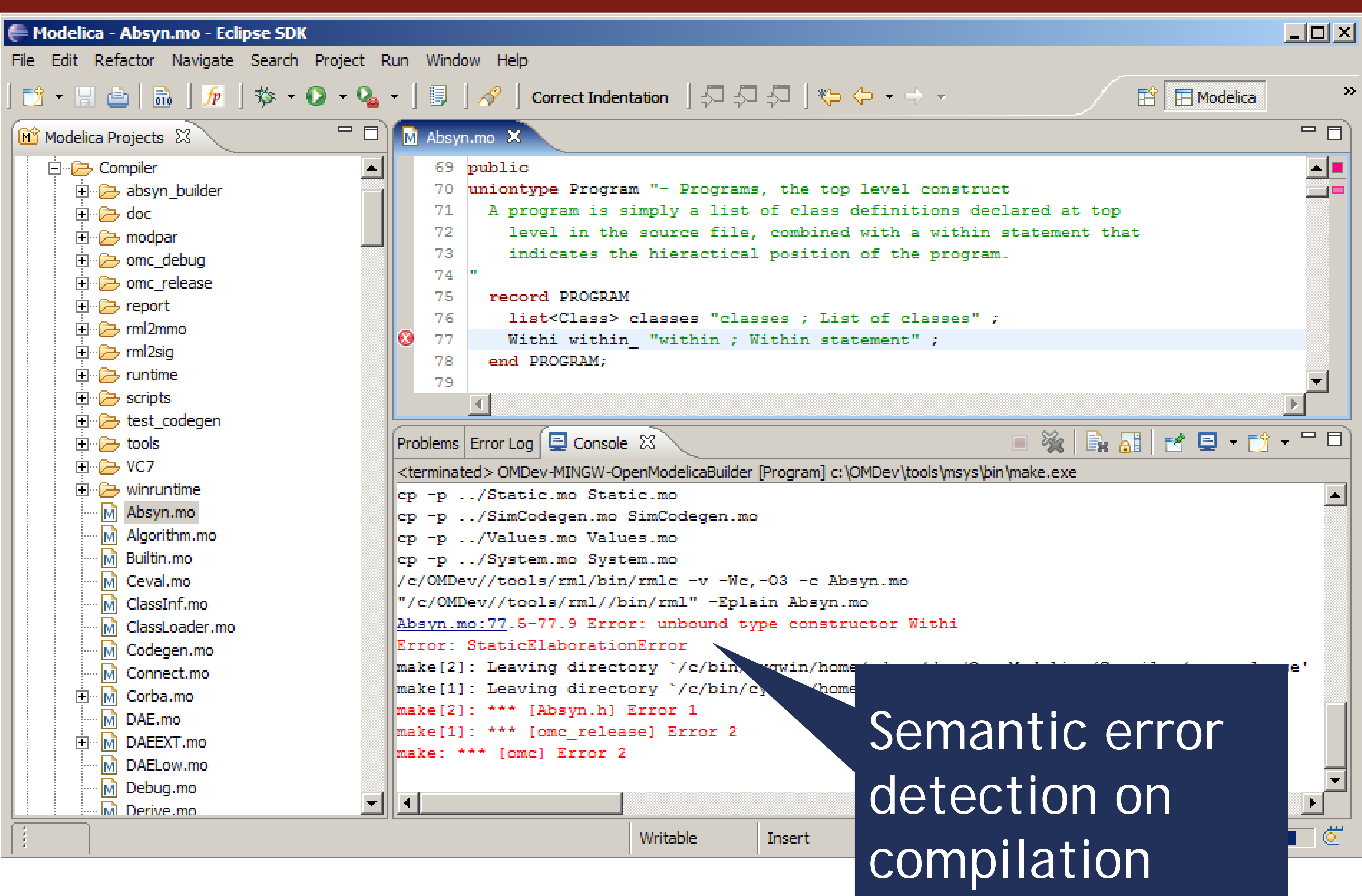

Modelica - VanDerPol.mo - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
Correct Indentation
Modelica Pro...
MyClass.mo
package.mo
VanDerPol.mo
demo
MyPackage
MyClass.mo
MyClass
package.mo
VanDerPol.mo
VanDerPol
e
lambda
x
y
.project
Standard Library
Modelica
Blocks
Constants
Electrical
Icons
Math
Matrices
acos
asin
atan
atan2
baseIco
// Van der Pol model
model VanDerPol "Van der Pol oscillator model"
Real x(start = 1);
Real y(start = 1);
parameter Real lambda = 0.3;
parameter Real e = Modelica.Constants.e;
equation
der(x) = y;
der(y) = - x + lambda*(1 - x*x)*y;
end VanDerPol;
Problems
0 errors, 0 warnings, 0 infos
Description
Code Browsing for easy navigation within Modelica files. Automatic update on file save.

Error detection (I)

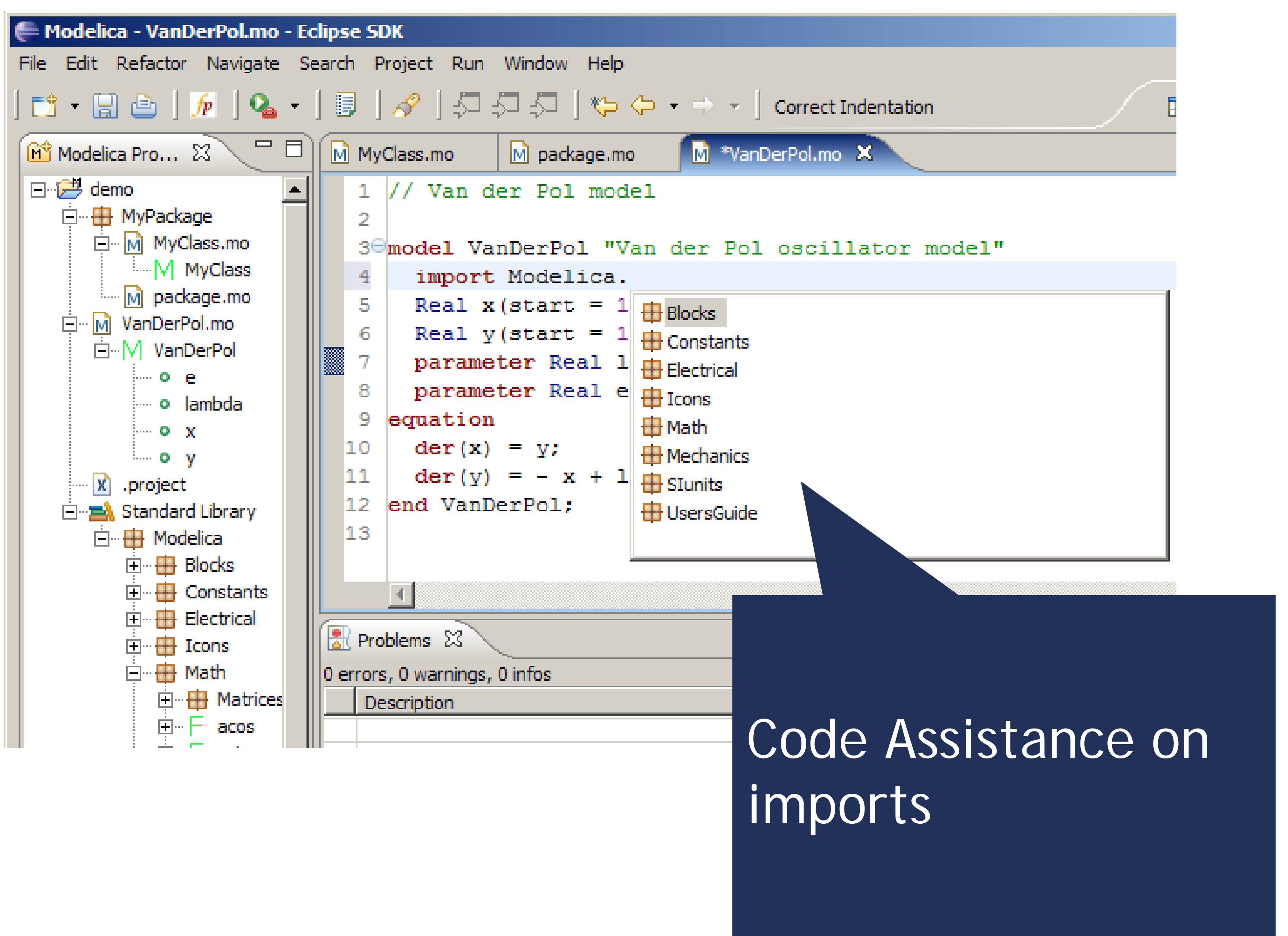


Parse error detection on file save

Error detection (II)

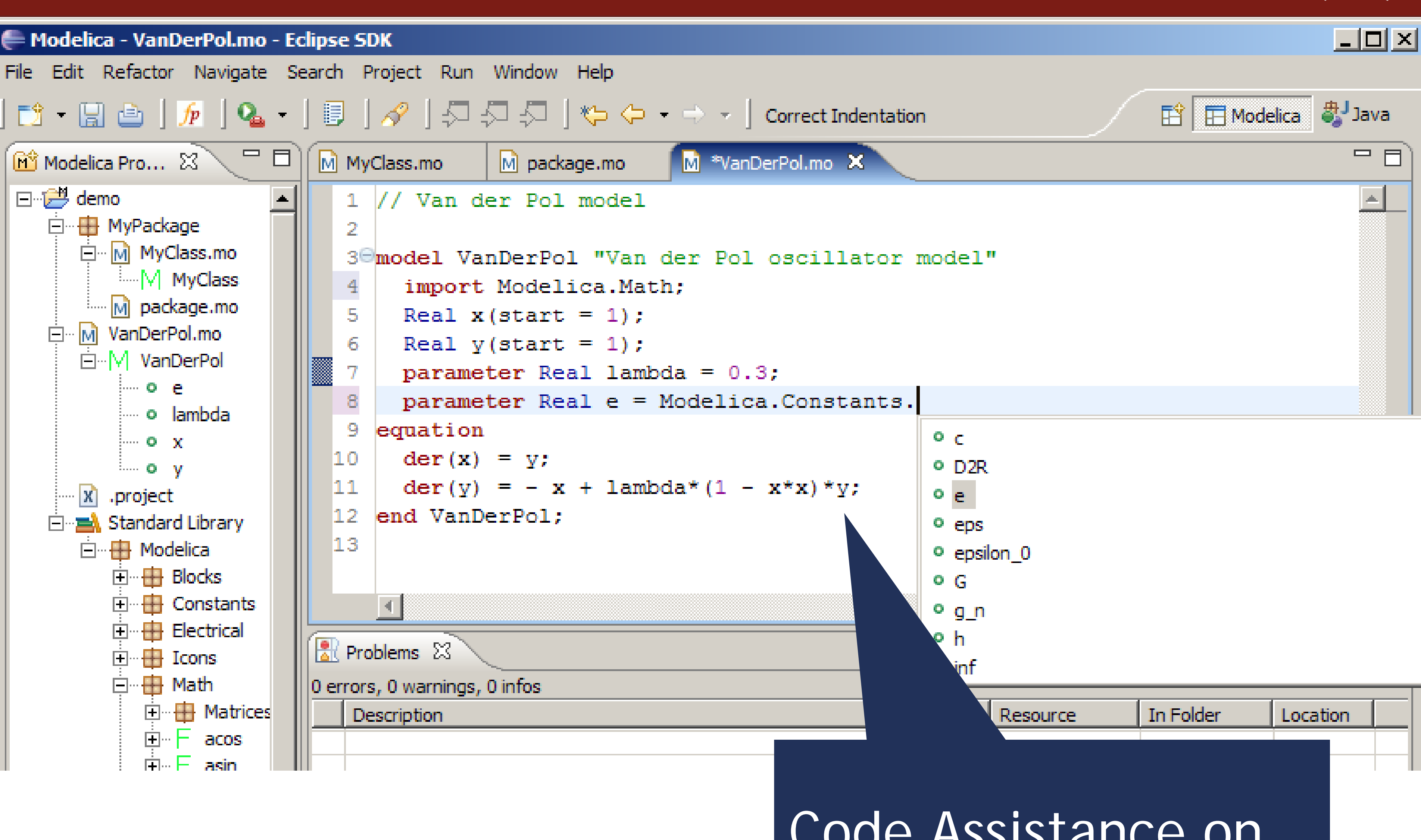


Code assistance (I)



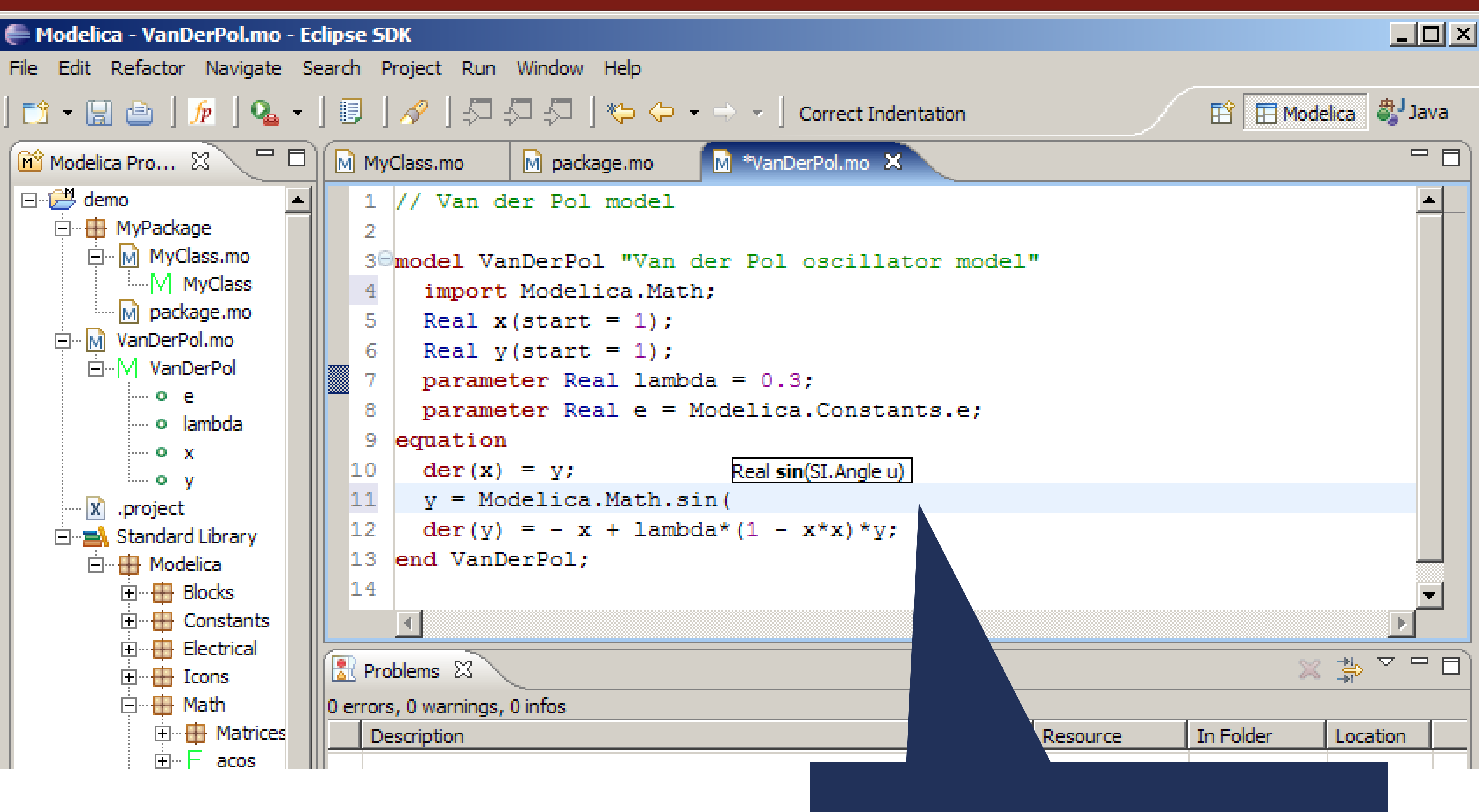
Code Assistance on imports

Code assistance (II)



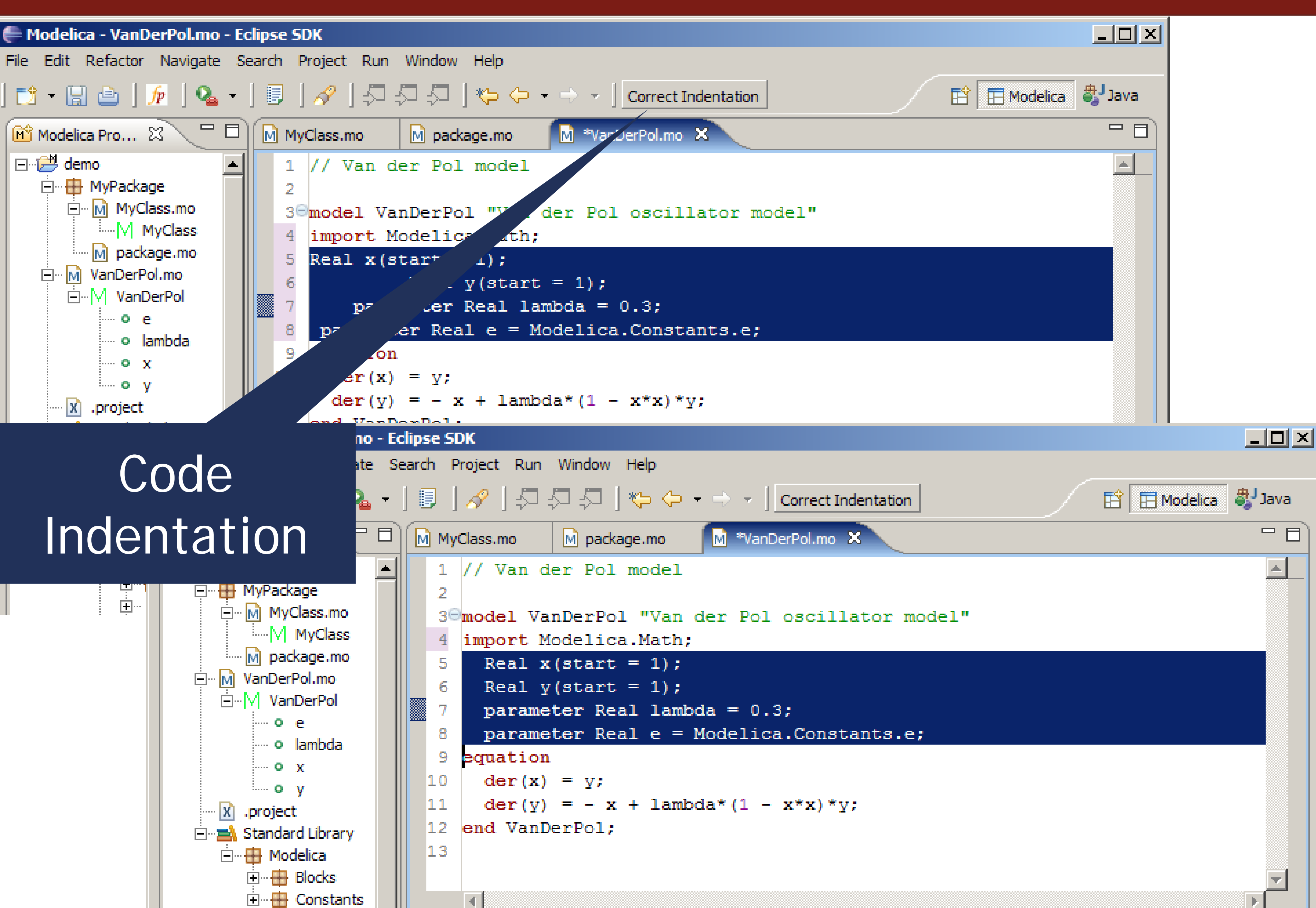
Code Assistance on assignments

Code assistance (III)

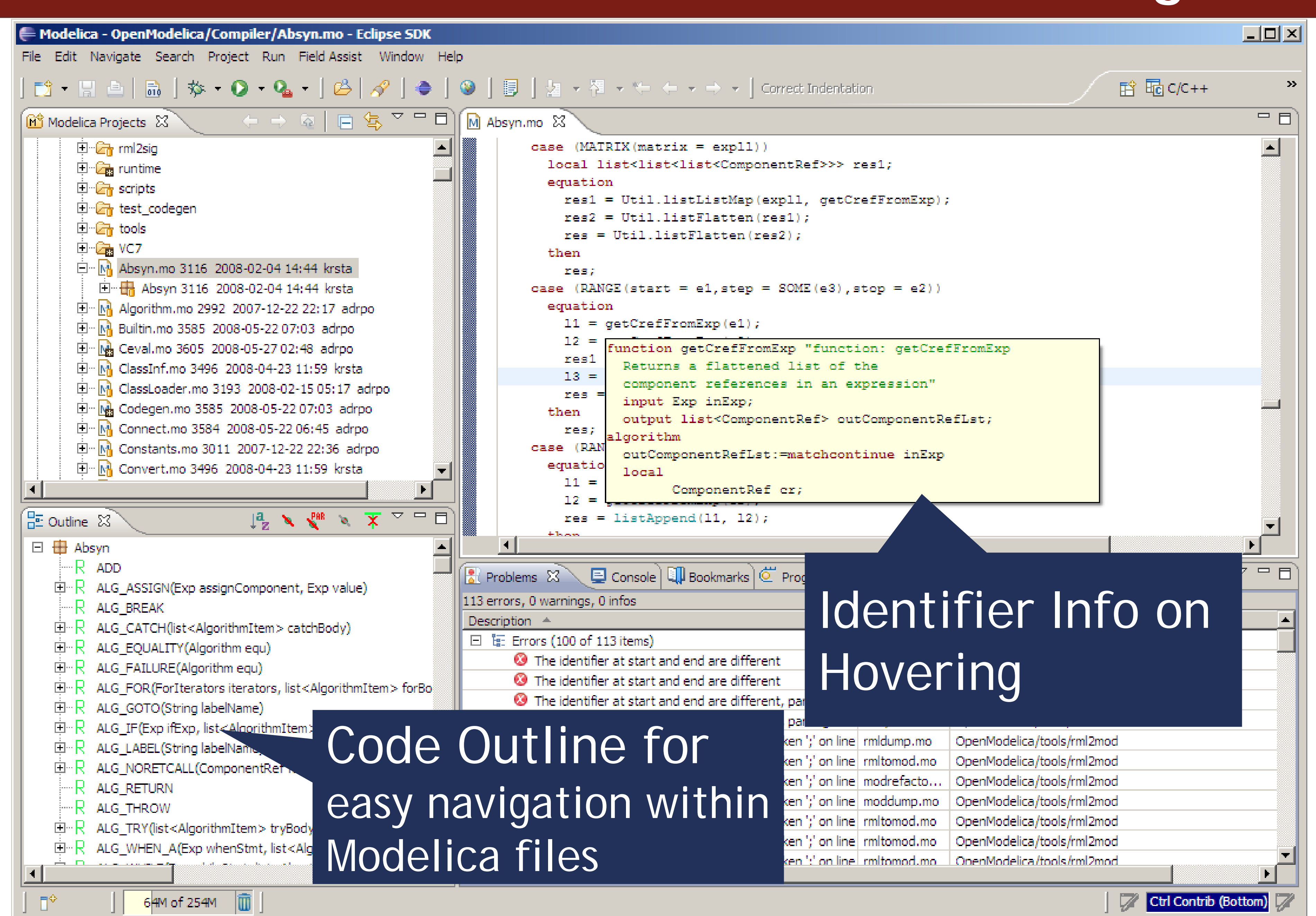


Code Assistance on function calls

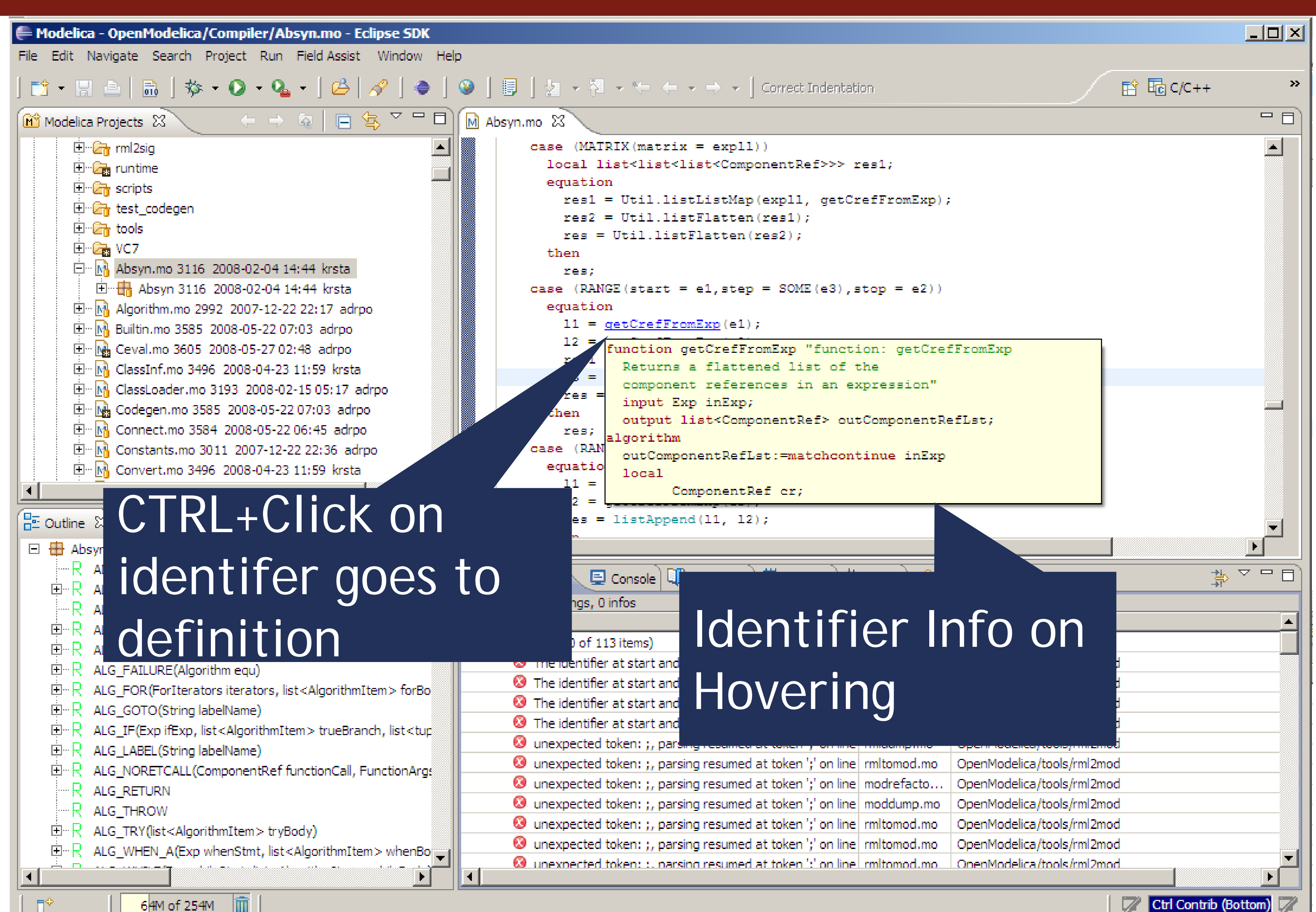
Code indentation



Code Outline and Hovering Info



Go to definition



Outline

- 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.

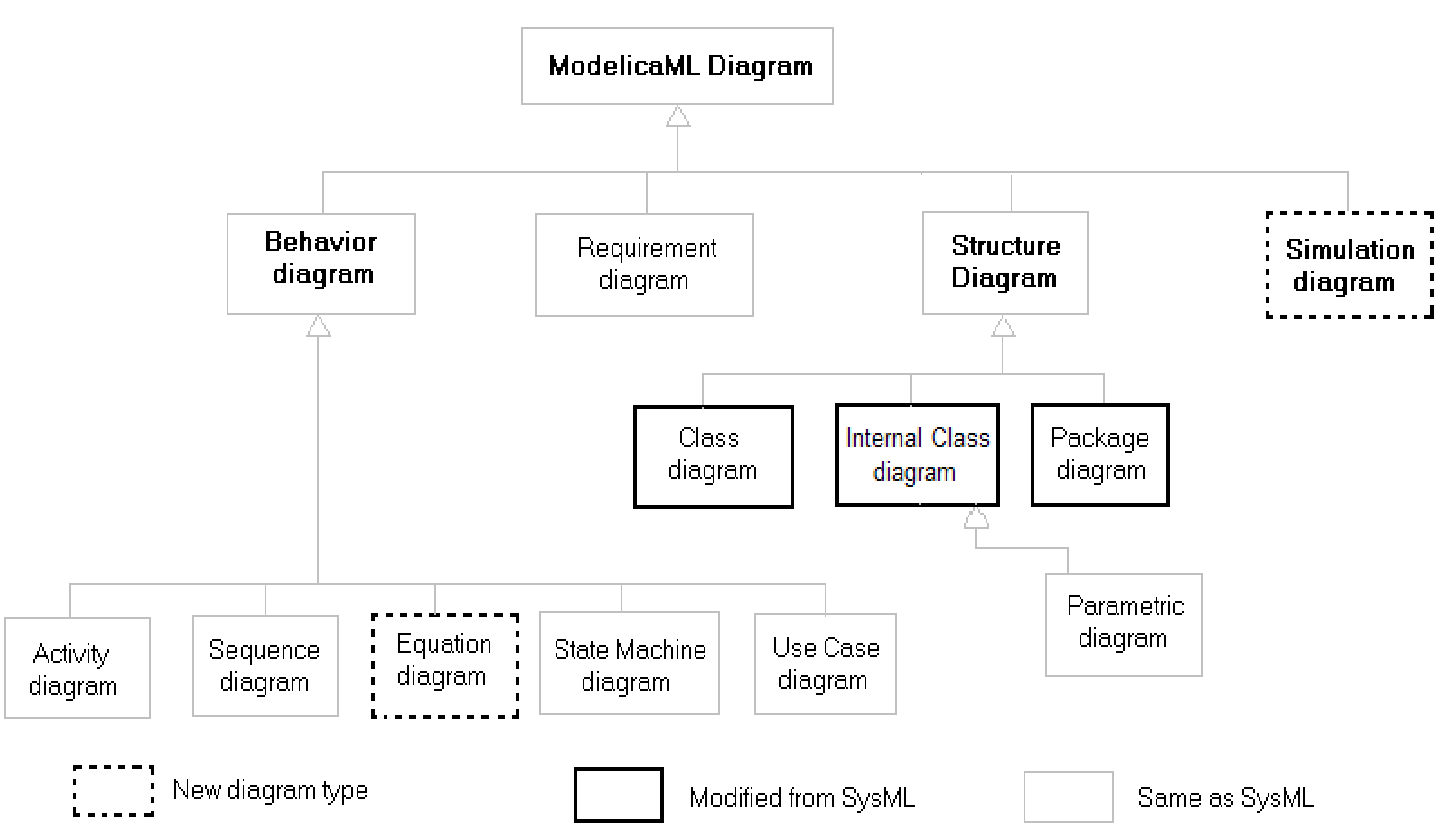
ModelicaML - a UML profile for Modelica

- 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.

ModelicaML - Purpose

- *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.

PowerTrain
MyPowerTrain
GearBoxes {replaceable}
GearBoxes

Modelica
Blocks
Examples
Mechanics
Continuous
Interfaces
Rotational

« ModelicaModel »
SampleModel
Parameters
SIunits.Length L = 10
SIunits.Velocity V = 1

« import »
Modelica.SIunits

GeneralStack
« ModelicaClass »
Element
{ replaceable }
« ModelicaRecord »
Stack
« ModelicaFunction »
Push
« ModelicaFunction »
Pop
« ModelicaFunction »
Top

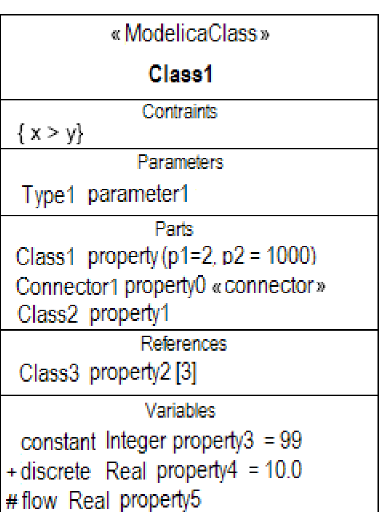
GeneralStack
« ModelicaClass » replaceable Element
« ModelicaRecord » Stack
« ModelicaFunction » Push
« ModelicaFunction » Pop
« ModelicaFunction » Top

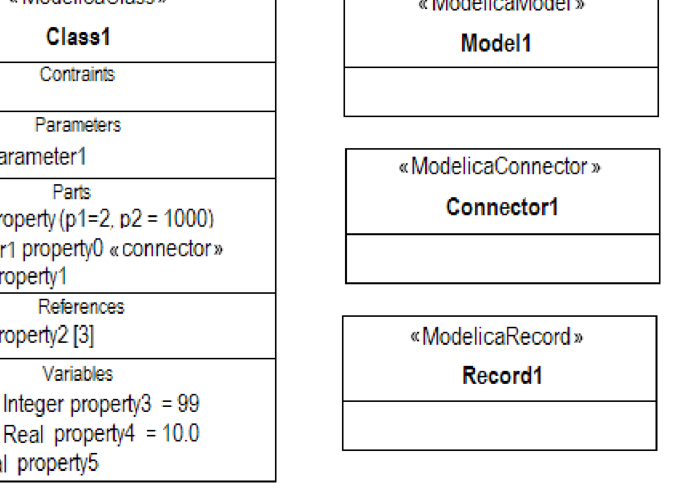
RealStack
GeneralStack
redeclare type Element = Real

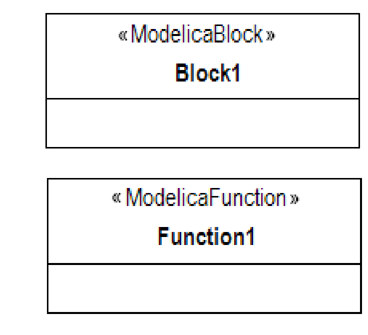
IntegerStack
GeneralStack
redeclare type Element = Integer

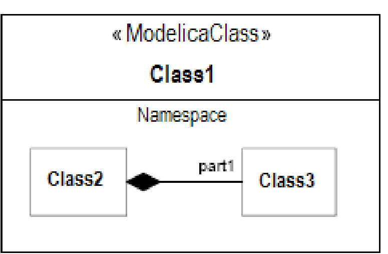
ModelicaML – Class Diagram

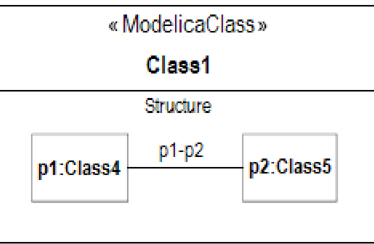
- ModelicaML provides extensions to SysML in order to support the full set of Modelica constructs.
- ModelicaML defines unique class definition types ModelicaClass, ModelicaModel, ModelicaBlock, ModelicaConnector, ModelicaFunction and ModelicaRecord that correspond to `class`, `model`, `block`, `connector`, `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).





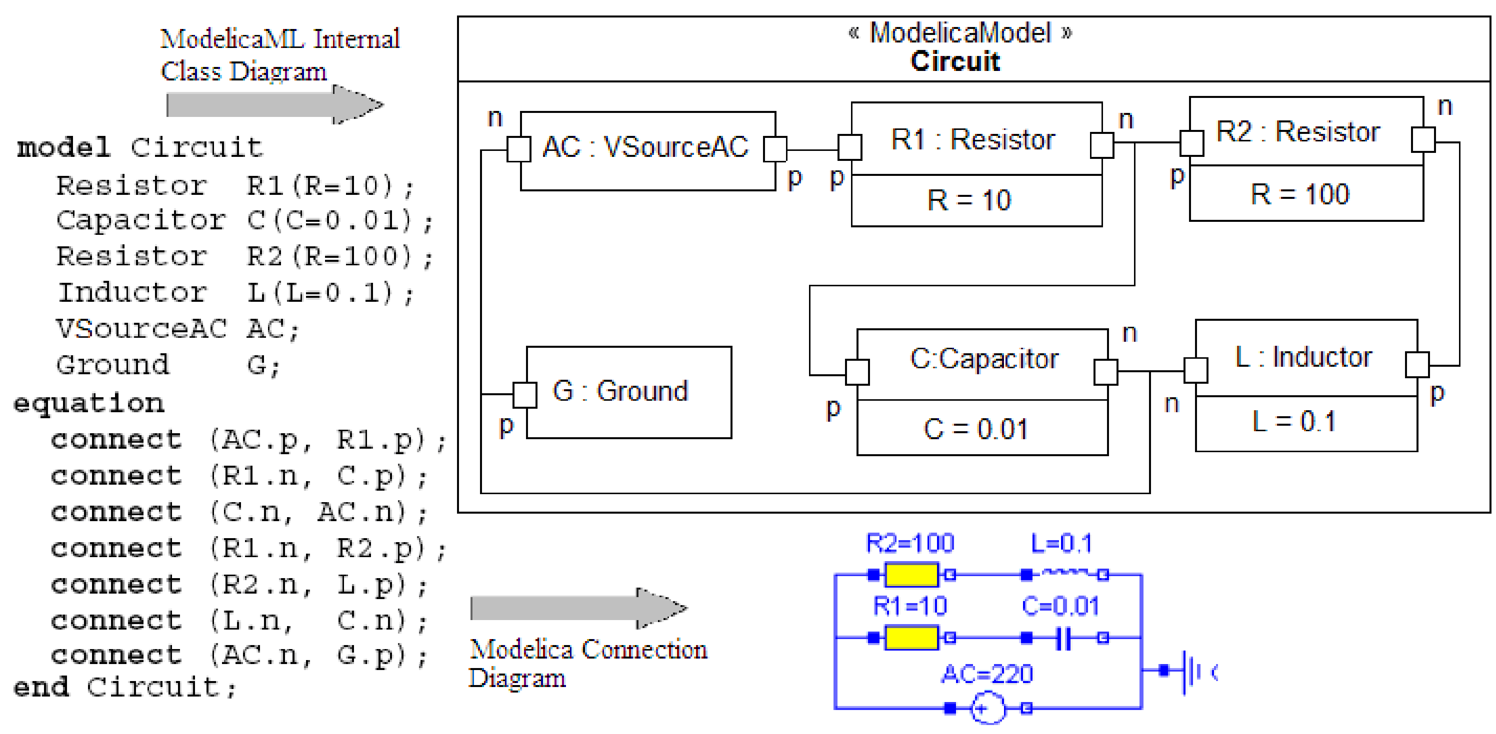






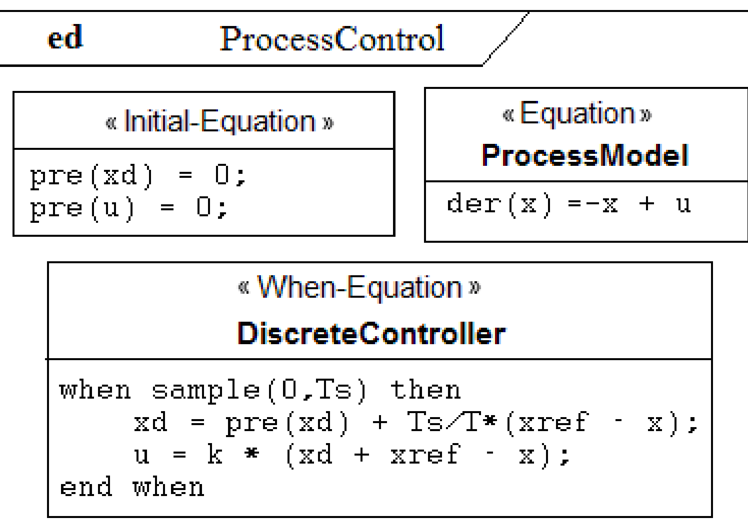
Class Diagram defines Modelica classes and relationships between classes, like generalizations, association and dependencies

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

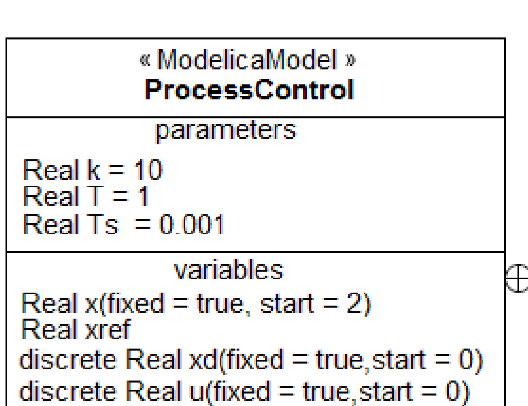


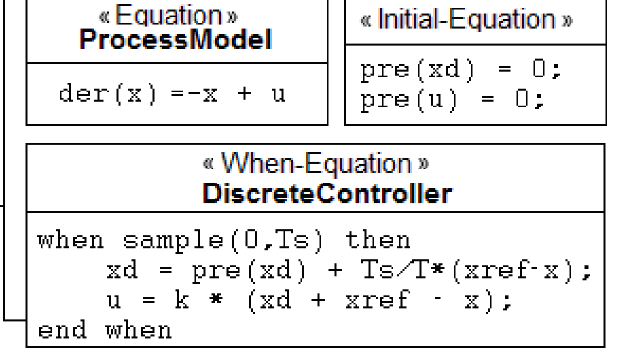
ModelicaML – Equation Diagram

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

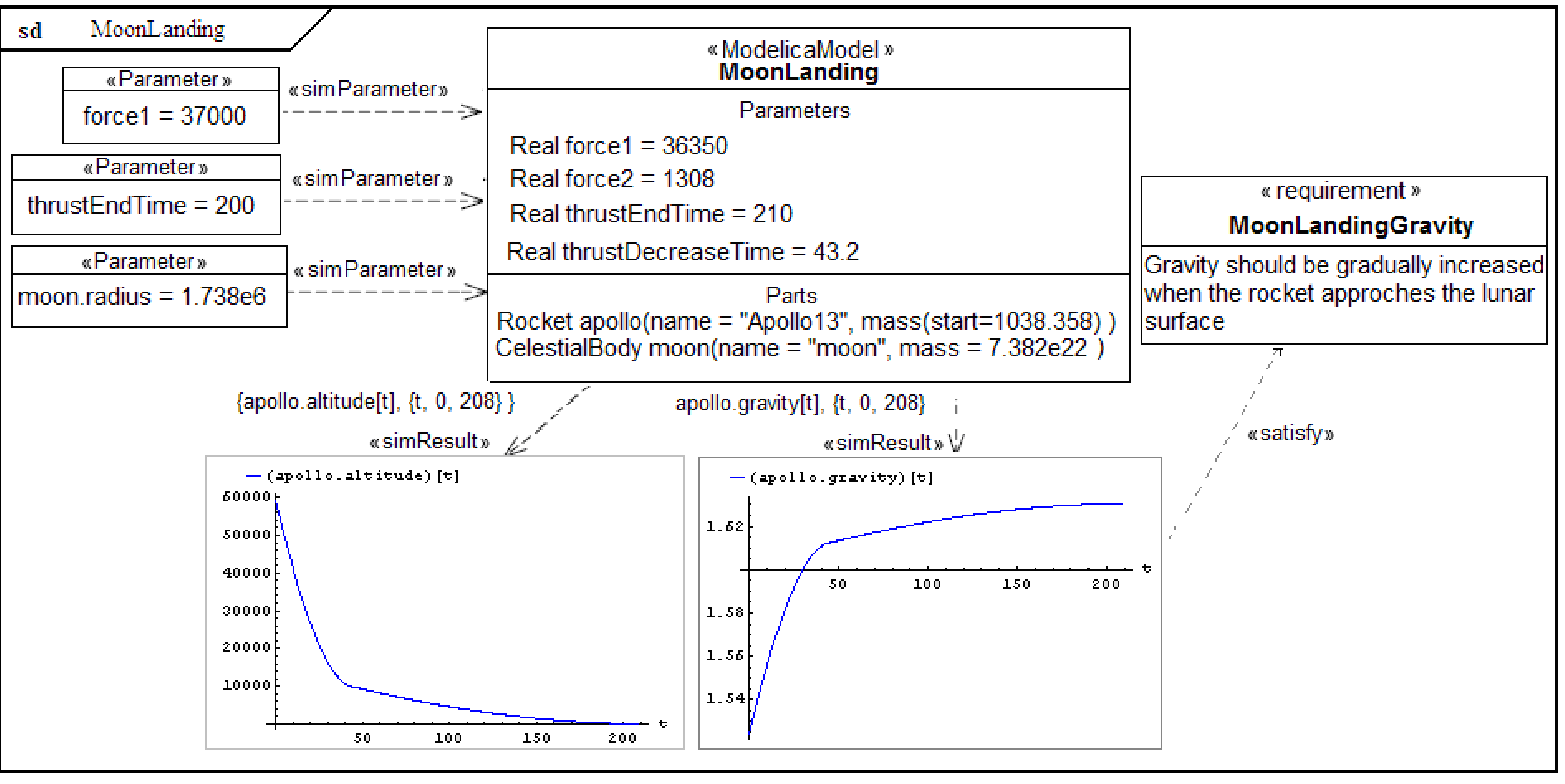



```
model ProcessControl
  parameter Real k=10,T=1;
  parameter Real Ts=0.001;
  Real x(fixed=true,start=2);
  Real xref;
  discrete Real xd(fixed=true,start=0);
  discrete Real u(fixed=true,start=0);
equation
  der(x) = =-x + u; // Process model
  // Discrete PI Controller
  when sample(0,Ts) then
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



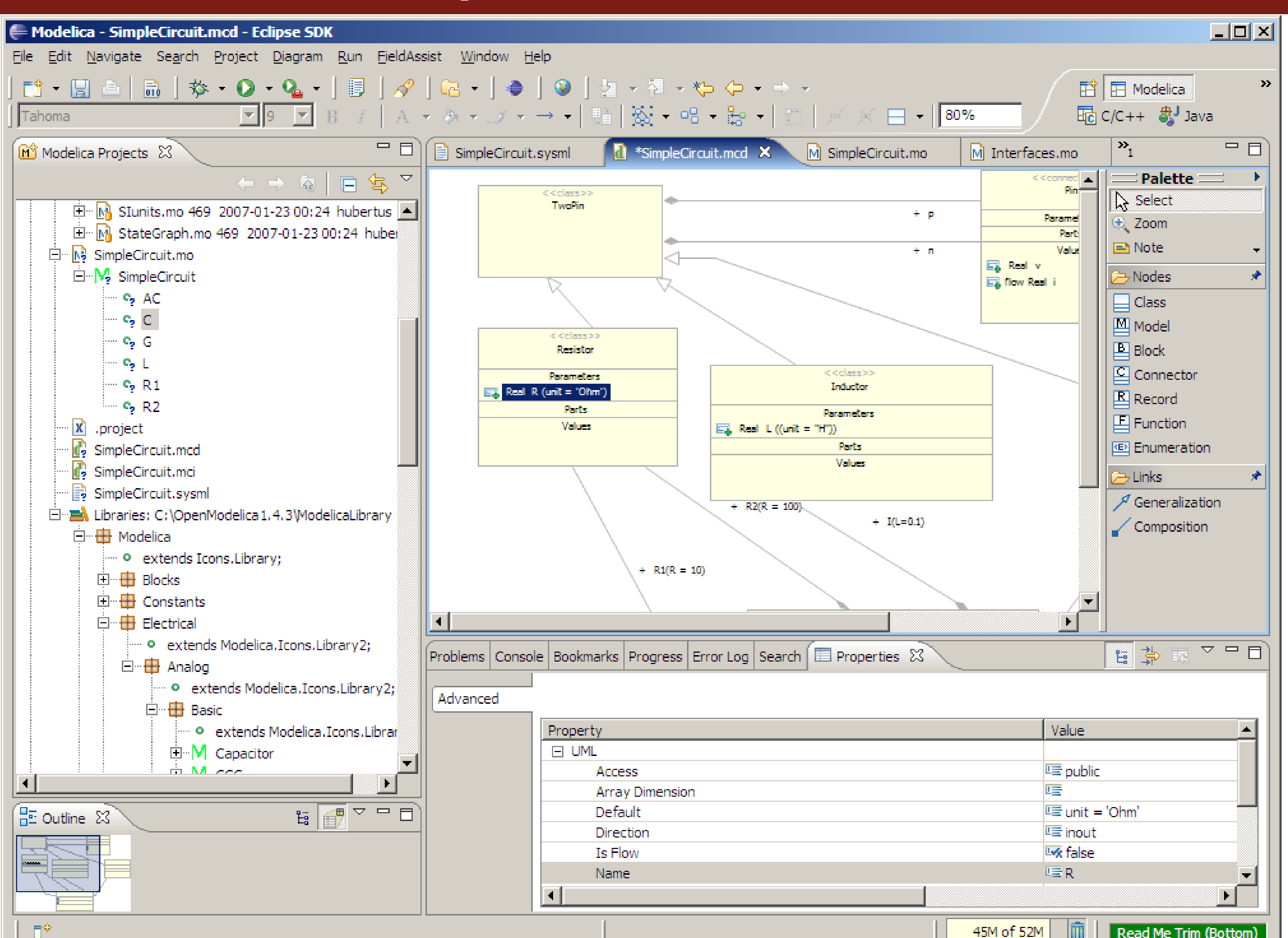


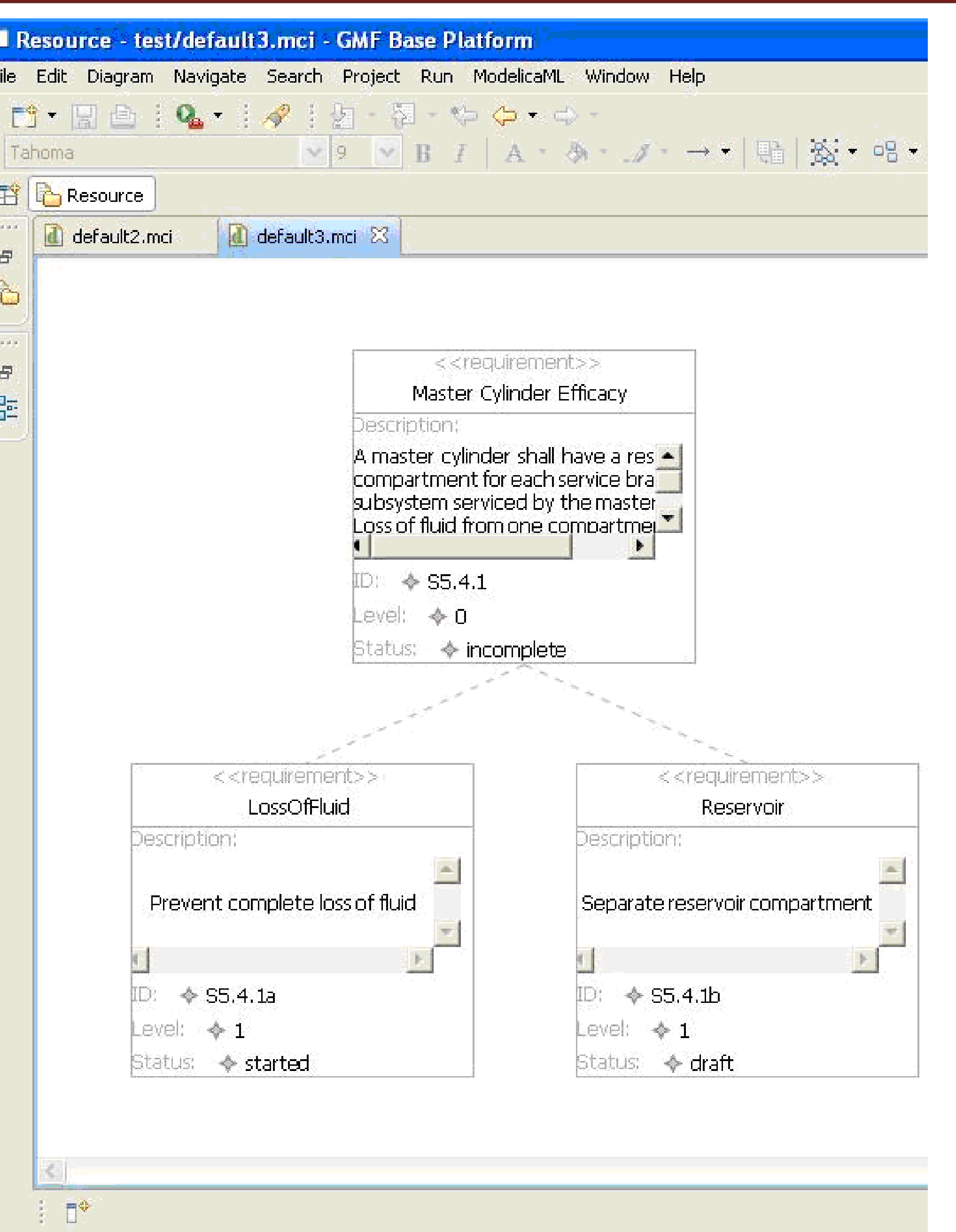
ModelicaML – Simulation Diagram



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

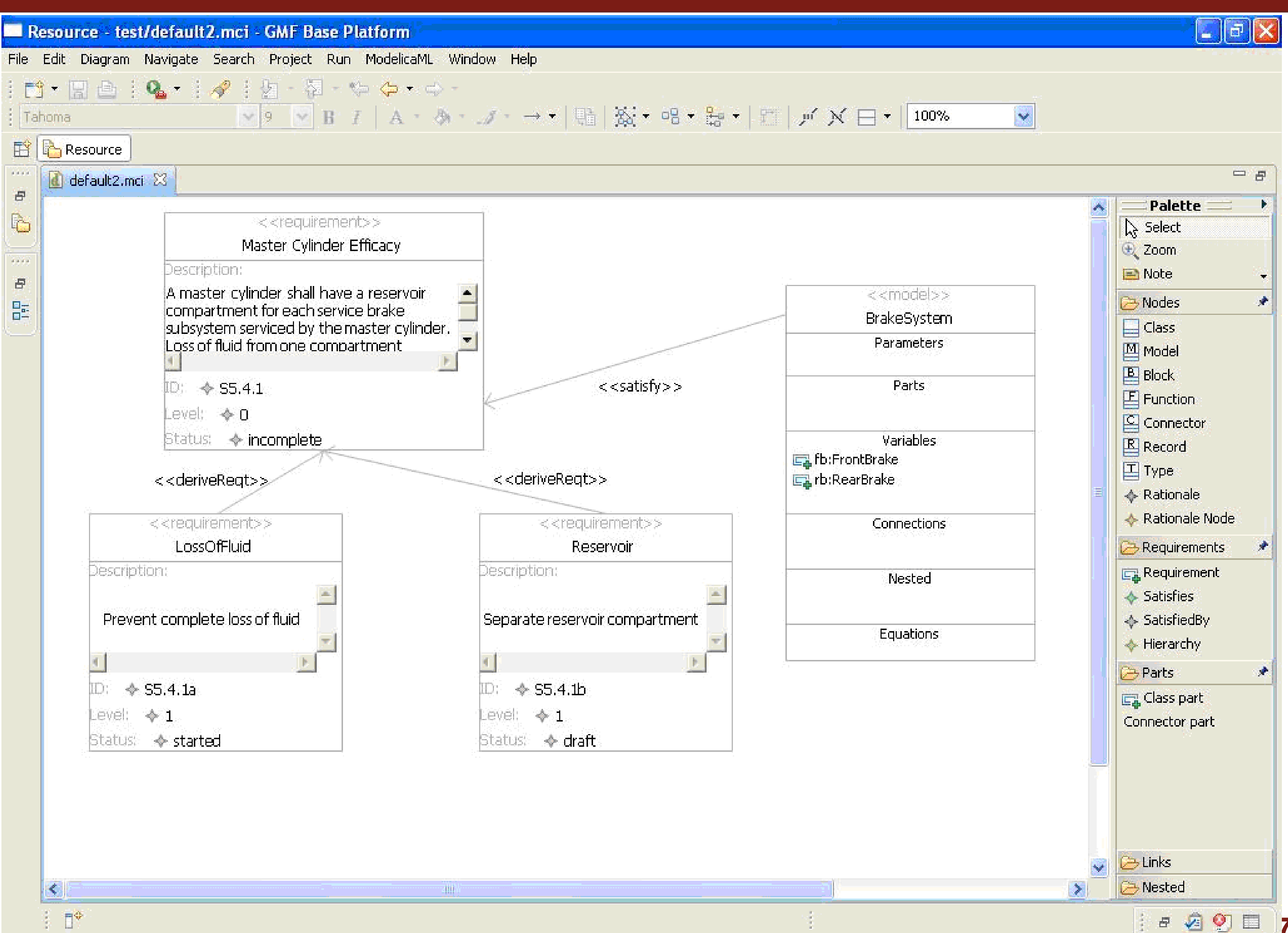
Eclipse environment for ModelicaML





- 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



Outline

- 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

Conclusions

- 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.

Future Work

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

Outline

- 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

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