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
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

“The change of motion is proportional to the motive force impressed“
– Newton
Equations were used in the third millennium B.C.
- Equality sign was introduced by Robert Recorde in 1557

\[
14.7 \times 1 - 1.15 \times 9 = -71.9
\]

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”:
\[
\text{variable} = \text{expression}
\]
\[
v = \text{INTEGRAL}(F)/m
\]

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*:

\[ R \times i = v; \]

can be used in three ways:

\[ i := v/R; \]
\[ v := R \times i; \]
\[ R := v/i; \]
Hierarchical Composition Diagram

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

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```

Multi-Domain Modelica Model - 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

The Modeling Space

Meta- Meta Model

Meta-Model1

Meta-Model2

Model1

Model2

ModelN

Physical system

Meta-programming: transformation

MetaModelica and Natural Semantics Specification formalisms

Modelica language specification

Modelica models
- **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.
MetaModelica - Idea

- 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
  
  \[
  \text{Env.BOOLVAL}(x,y) = \text{eval}_{\text{something}}(\text{env}, e);
  \]

- match expressions
  - pattern matching
  - case rules

  \[
  \text{pattern} = \text{match} \\text{expression} \\text{optional-local-declarations}
  \]
  \[
  \text{case} \\text{pattern-expression} \\text{opt-local-declarations}
  \]
  \[
  \text{optional-local-equations then value-expression;}
  \]

  \[
  \text{case} \ldots
  \]

  \[
  \ldots
  \]

  \[
  \text{else} \\text{optional-local-declarations}
  \]
  \[
  \text{optional-local-equations then value-expression;}
  \]

  end match;
package ExpressionEvaluator

// abstract syntax declarations
...

// semantic functions
...

end ExpressionEvaluator;
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;
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
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*
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
  ordElem := unordElem;
  while not isOver loop
    isOver := true;
    Debug.register_out("isOver", isOver);
    Debug.register_in("ordElem",ordElem);
    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.register_in("ordElem[i]",
                       ordElem[i+1]);
      Debug.step(...);
    end for;
  end while;
end bubbleSort;
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)

<table>
<thead>
<tr>
<th></th>
<th>Generated C Code</th>
<th>Compilation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No debugging</td>
<td>37 (MB)</td>
<td>269.86 (s)</td>
</tr>
<tr>
<td>Early instrumentation</td>
<td>130+ (MB)</td>
<td>850.35 (s)</td>
</tr>
<tr>
<td>Late instrumentation</td>
<td>103 (MB)</td>
<td>610.61 (s)</td>
</tr>
</tbody>
</table>
The test case

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

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

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No debugging</td>
<td>223.01 (s)</td>
</tr>
<tr>
<td>Early instrumentation</td>
<td>5395.47 (s)</td>
</tr>
<tr>
<td>Late instrumentation</td>
<td>864.36 (s)</td>
</tr>
</tbody>
</table>
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?
Existing Debugging Strategies Do Not Suffice

Modelica Specific
- Modelica
- OpenModelica
- OpenModelica Simulation Runtime
- Simulation Files

Compilation & Simulation

Debugging & Tracing

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

Model Apollo
... equation
... gravity = ...;
end Apollo;

Error

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

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

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
Modelica Specific Debugging Strategy: Compiling With Debugging In Mind

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

Error Discovered

Error

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

1. Modelica Source Code
2. Translator
3. Analyzer
4. Optimizer
5. Code Generator
6. C Compiler
7. Executable
8. Simulation

Include debugging support within the 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
OpenModelica

- 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

Server: Main Program Including Compiler, Interpreter, etc.

Parse

Corba

Client: Graphic Model Editor

Client: OMShell Interactive Session Handler

Client: Eclipse Plugin

SCode

Interactive

Inst

Ceval

system

plot

e tc.

SCode

Interactive

Inst

Ceval

system

plot

e tc.

SCode

Interactive

Inst

Ceval

system

plot

e tc.
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 (1)

Modelica Browser
Modelica Editor
Modelica Code Assistant
MetaModelica Debugging
Modelica Perspective
The MDT Eclipse Environment (II)

- .mo file
- OMC Compiler
- MMC Compiler
- Small Modelica Parser
- AST Information
- Modelica model
- Modelica Browser
- Modelica Editor
- Modelica Code Assistant
- Eclipse
- MetaModelica Builder
- MetaModelica Build console
- MetaModelica Debugging
The MDT Eclipse Environment (III)

- .mo file
- MetaModelica
- Debugging
- Modelica Editor
- Eclipse
- MMC Compiler
- Executable + Debugging runtime
Creating Modelica projects (1)

Creation of Modelica projects using wizards
Creating Modelica packages

Creation of Modelica packages using wizards
Creating Modelica classes

Creation of Modelica classes, models, etc, using wizards
Error detection (1)

Parse error detection on file save
Semantic error detection on compilation
Code Assistance (1)

Code Assistance on imports
Code assistance (II)

Code Assistance on assignments
Code Assistance on function calls
Code indentation
Code Outline and Hovering Info

Code Outline for easy navigation within Modelica files

Identifier Info on Hovering
Go to definition

CTRL+Click on identifier goes to definition

Identifier Info on Hovering
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
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 Diagram

Behavior diagram

Requirement diagram

Structure Diagram

Simulation diagram

Class diagram

Internal Class diagram

Package diagram

Activity diagram

Sequence diagram

Equation diagram

State Machine diagram

Use Case diagram

Parametric diagram

New diagram type

Modified from SysML

Same as SysML
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 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.
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 Modeling

- 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

Resource - test/default2.mci - GMF Base Platform

Diagram showing requirements related to Master Cylinder Efficacy and Loss of Fluid.
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
Part I  Motivation, Introduction, Background and Related Work
Chapter 1.  Introduction
Chapter 2.  Background and Related Work
Part II  Extending EOO Languages for Safe Symbolic Processing
Chapter 3.  Extending Equation-based Object-oriented Languages
Chapter 4.  Efficient Implementation of Meta-Programming EOO Languages
Part III  Debugging of Equation-based Object Oriented Languages
Chapter 5.  Portable Debugging EOO Meta-programs
Chapter 6.  Run-time Debugging of EOO Languages
Chapter 7.  Debugging Natural Semantics Specifications
Part IV  Advanced Integrated Environments
Chapter 8.  Modelica Development Tooling (MDT)
Chapter 9.  Parsing-Unparsing and Refactoring
Chapter 10.  UML and Modelica System Modeling with ModelicaML
Chapter 11.  Integrated Framework for Model-driven Product Design and Development
Part V  Meta-programming and Composition of EOO Languages
Chapter 12.  ModelicaXML: A ModelicaXML Representation with Applications
Chapter 13.  Composition of XML dialects: A ModelicaXML case study
Part VI  Conclusions and Future Work
Chapter 14.  Conclusions and Future Work