Contributions to Meta-Modeling Tools and Methods

Adrian Pop

Programming Environments Laboratory

Outline

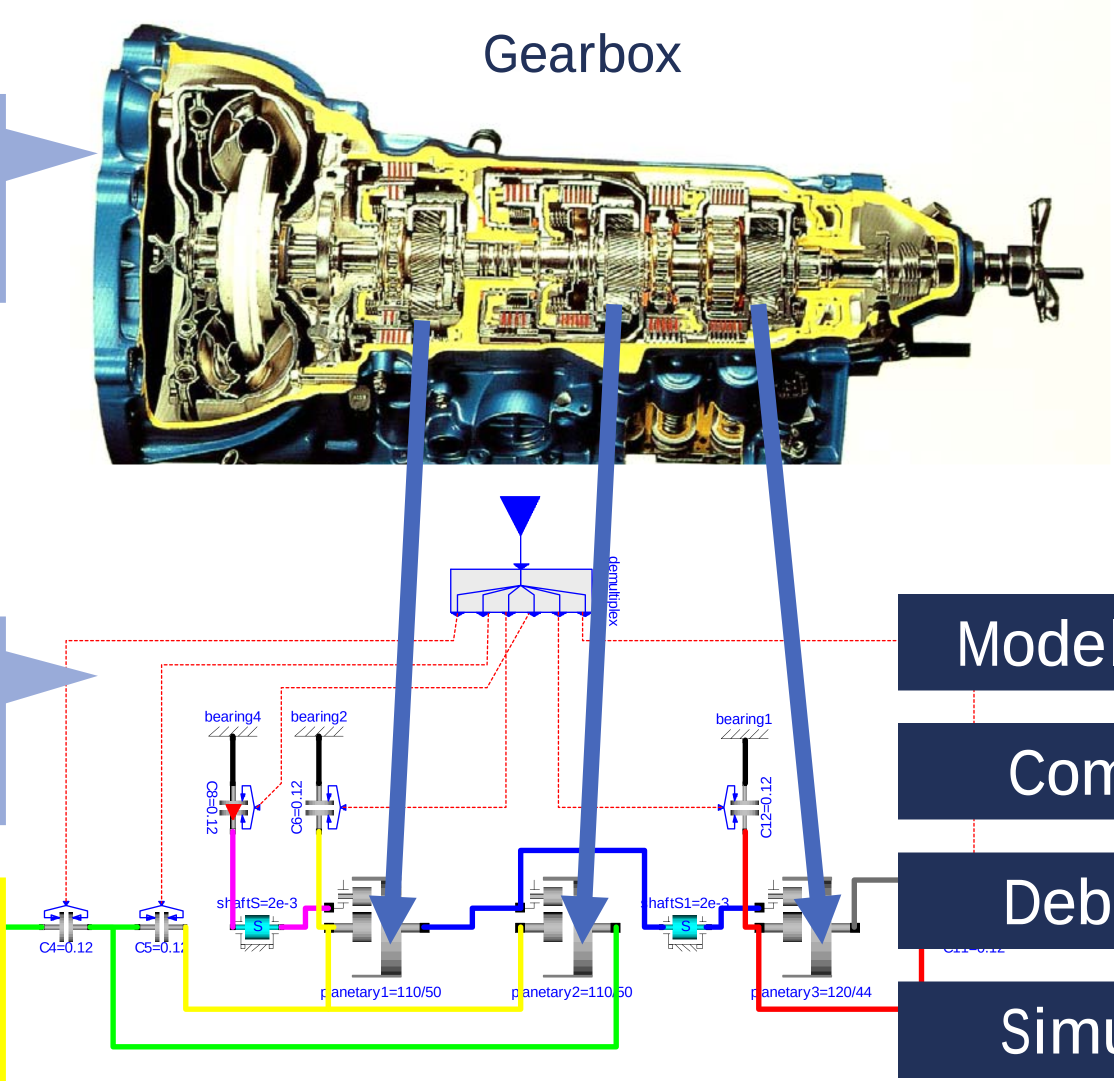
- Product Design Environments

- Meta-Modeling
  - Modelica Meta-Model
  - Invasive Composition of Modelica
  - Model-driven Product Design using Modelica

- Meta-Programming
  - Debugging of Natural Semantics Specifications

- Conclusions and Future Work

Domain Specific Environments



Complex physical products

Design environments

Difficult manual process

Model editors

Compilers

Debuggers

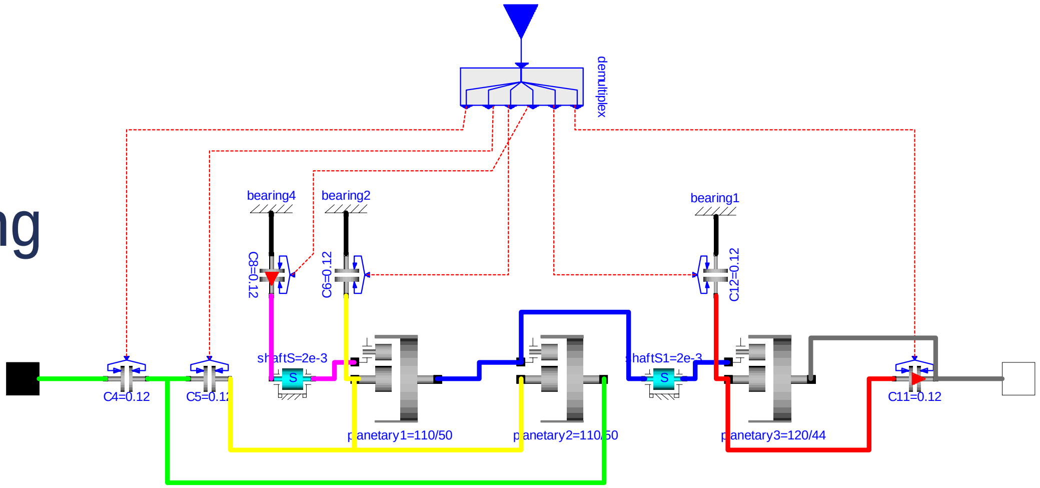
Simulators

Research Objectives

- Context
  - Model-driven product design environments
    - Modeling and simulation
  - Modelica Framework

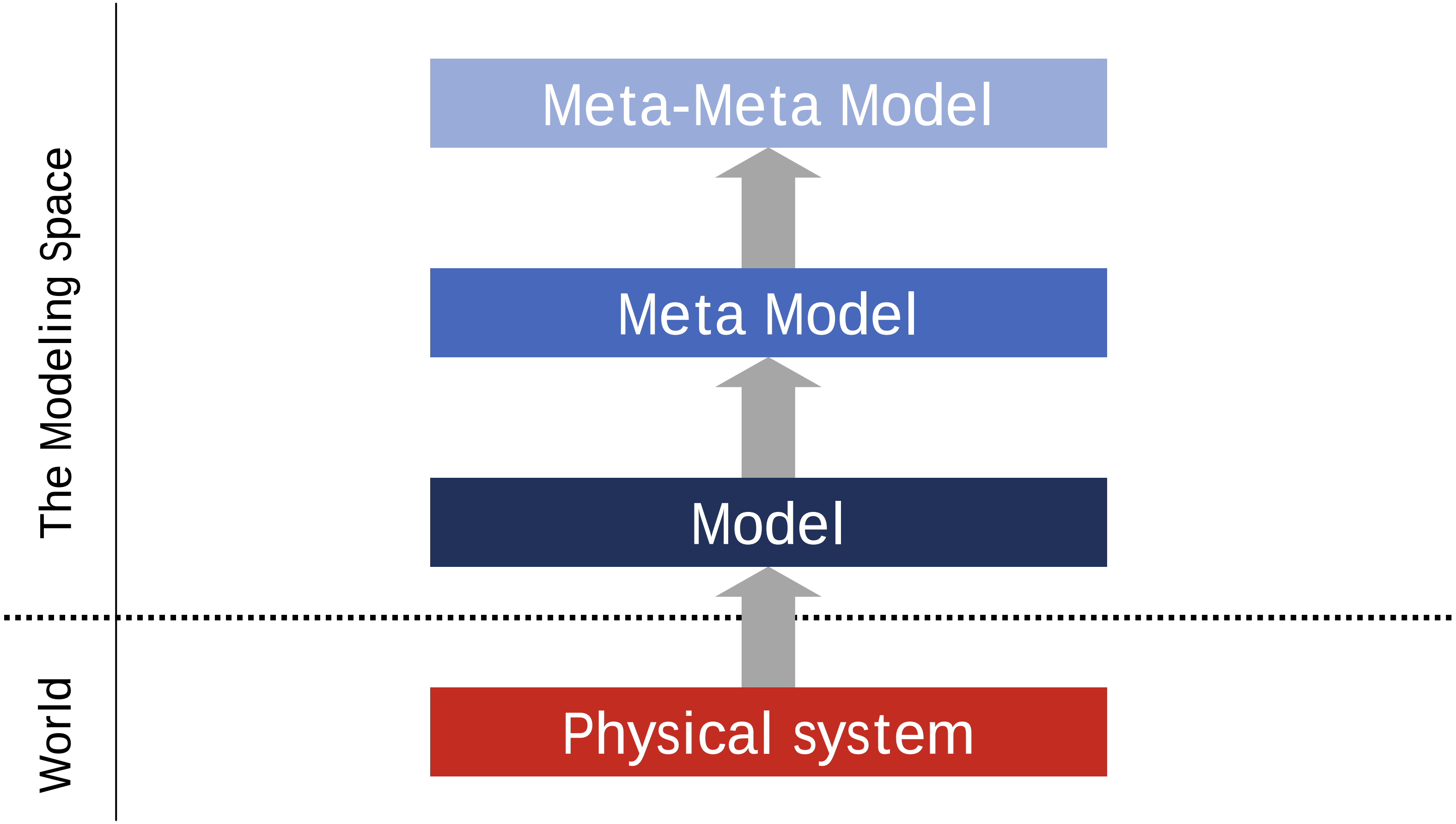
- Objective
  - Efficient development of such environments
    - Meta-modeling and meta-programming tools and methods

Modelica



- Modelica
  - Declarative language
  - Multi-domain modeling
  - Everything is a class
  - Visual component programming
- Modelica Association
  - http://www.modelica.org

Meta-Modeling



The Modeling Space

World

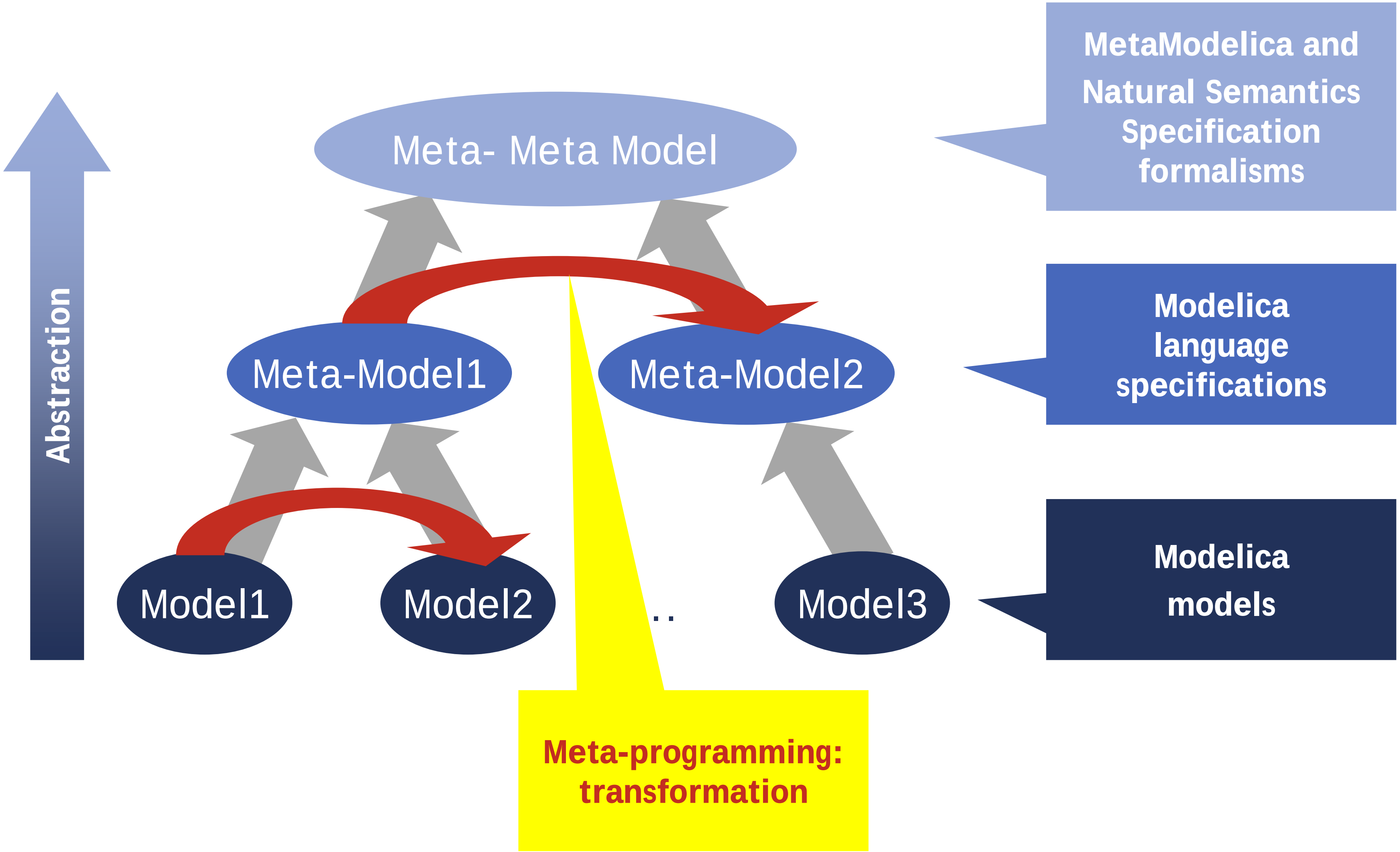
Meta-Meta Model

Meta Model

Model

Physical system

Meta-Programming



Abstraction

Meta- Meta Model

Meta-Model1

Meta-Model2

Model1

Model2

..

Model3

MetaModelica and Natural Semantics Specification formalisms

Modelica language specifications

Modelica models

Meta-programming: transformation

Outline

- Product Design Environments

- Meta-Modeling
  - Modelica Meta-Model
    - Purpose
    - Definition and Applications
    - Problems
  - Invasive Composition of Modelica
  - Model-driven Product Design using Modelica

- Meta-Programming
  - Debugging of Natural Semantics Specifications

- Conclusions and Future Work

Modelica Community

- Fast growing model base

- Needs flexible stand-alone tools for:
  - analysis of models (checkers and validators)
  - pretty printing (un-parsing)
  - interchange with other modeling languages
  - query and transformation of models
  - imposing code style guidelines
  - documentation generation (Html, SVG, MathML, etc)

Need of better support:

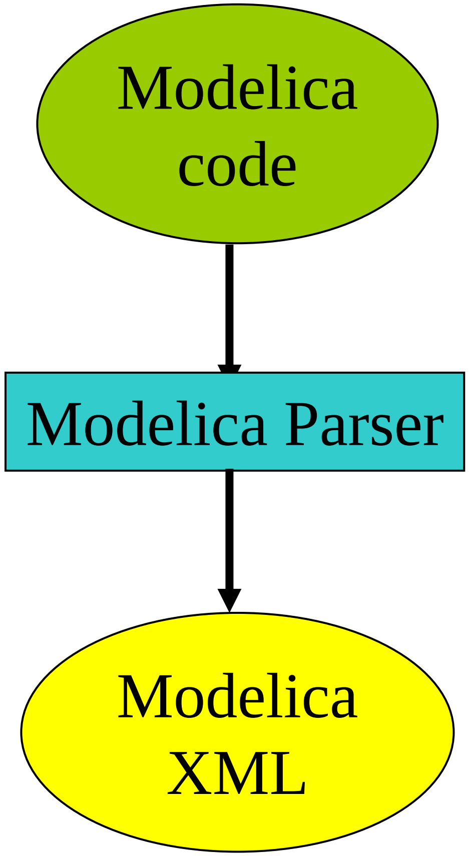
- **easy access to the language structure**
- **interoperability, flexibility**

Modelica Meta-Model

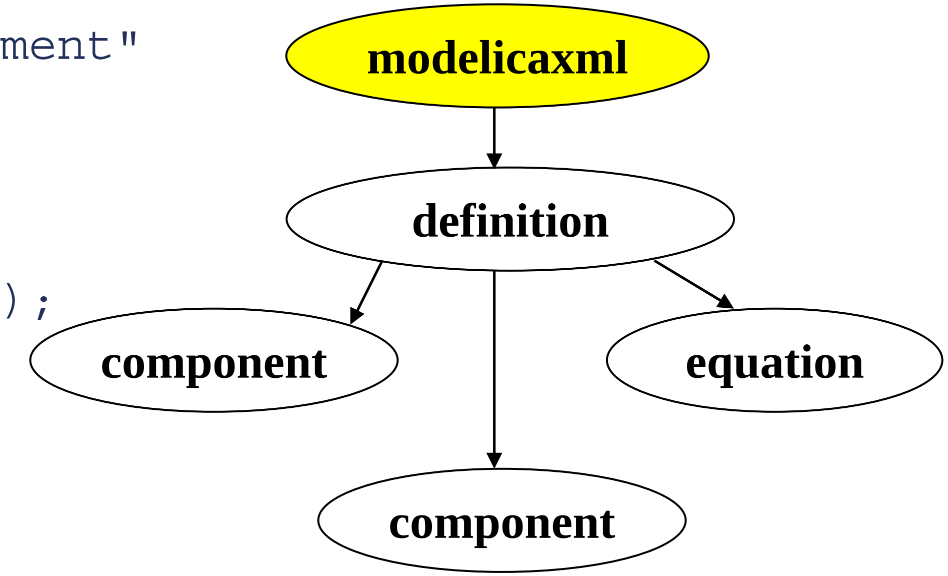
- Store the structure (Abstract Syntax) of the Modelica language using an *alternative representation*
- Create tools that use this alternative representation
- The alternative representation should
  - be easy accessible from any programming language
  - be easy to transform, query and manipulate
  - Support validation through a *meta-model*

XML has all these properties

ModelicaXML Representation



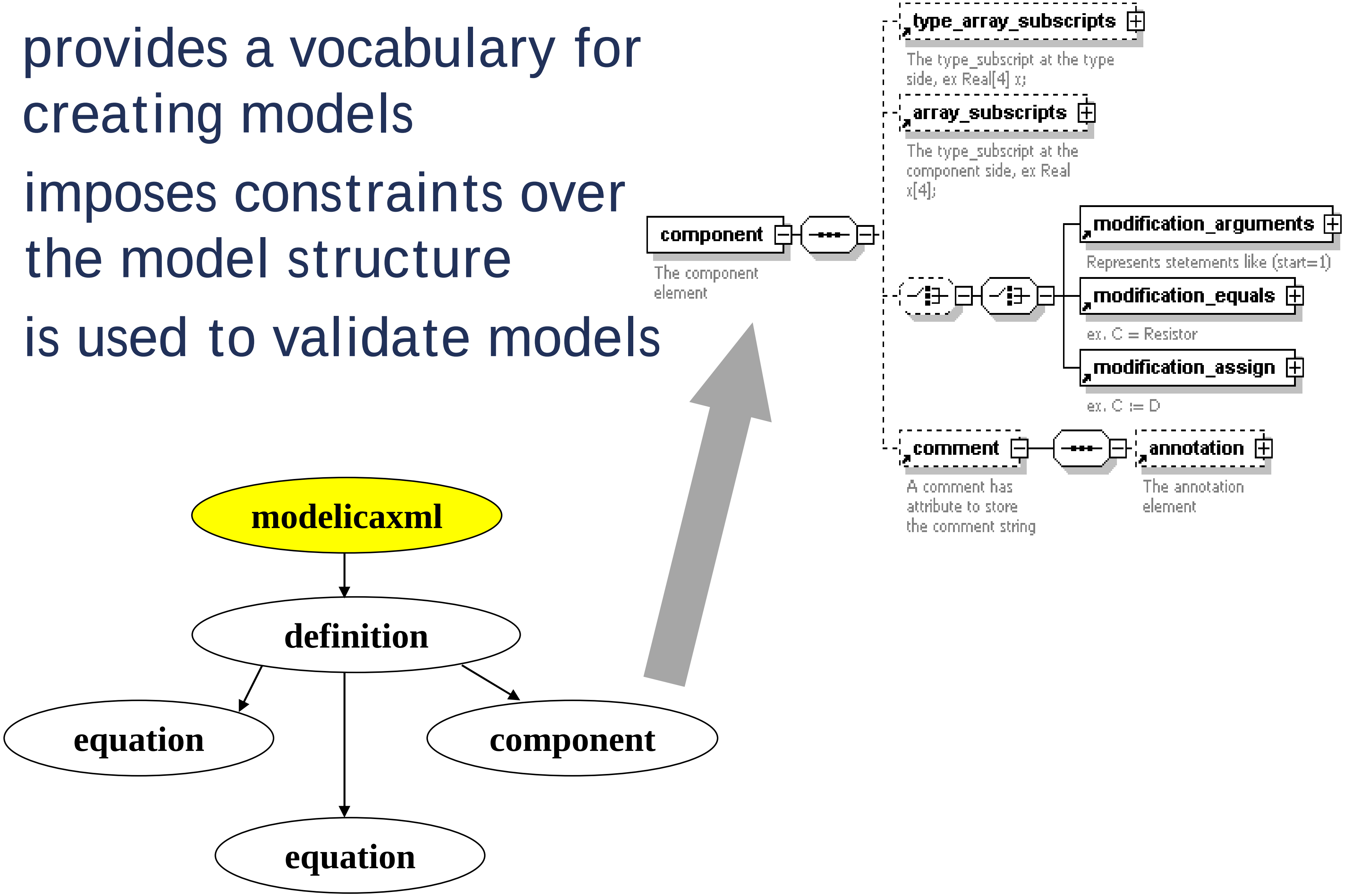
```
class Test "comment"
   Real x;
   Real xdot;
 equation
   xdot = der(x);
end Test;
```



```
<modelicaxml>
  <definition ident= "Test"
              comment="comment">
     <component ident="x" type="Real"
                visibility="public" />
     <component ident="xdot" type="Real"
                visibility="public" />
     <equation>...</equation>
  </definition>
</modelicaxml>
```

Validation using Modelica Meta-Model

- provides a vocabulary for creating models
- imposes constraints over the model structure
- is used to validate models



ModelicaXML Representation - Applications

- Applications of ModelicaXML Representation
  - Interoperability and transformation
  - Easy access from any programming language
  - Query facilities
  - Documentation generation
  - Validation of models using the meta-model

Outline

- Product Design Environments

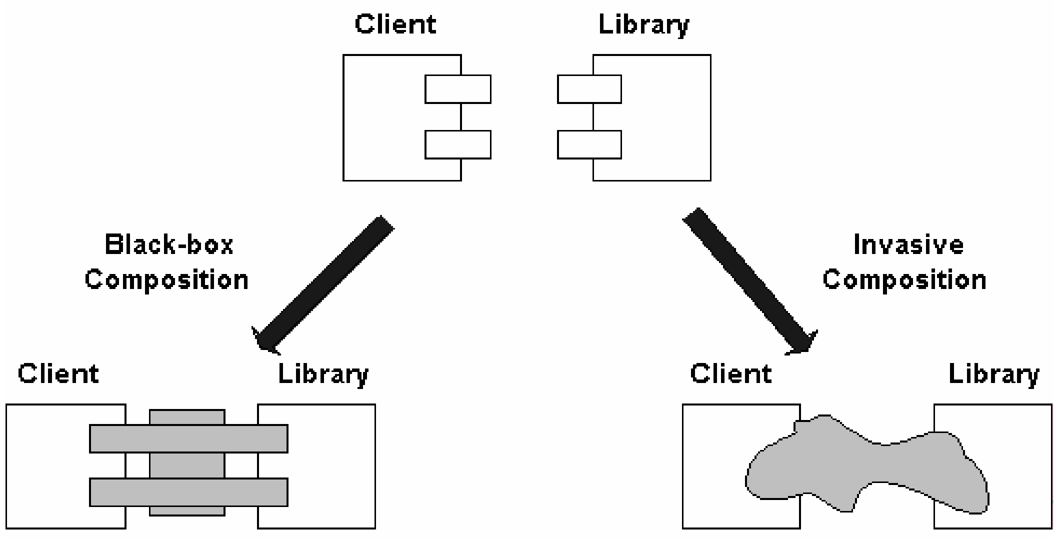
- Meta-Modeling
  - Modelica Meta-Model
  - Invasive Composition of Modelica
    - Invasive Software Composition
    - Modelica Composition
    - Applications
  - Model-driven Product Design using Modelica

- Meta-Programming
  - Debugging of Natural Semantics Specifications

- Conclusions and Future Work

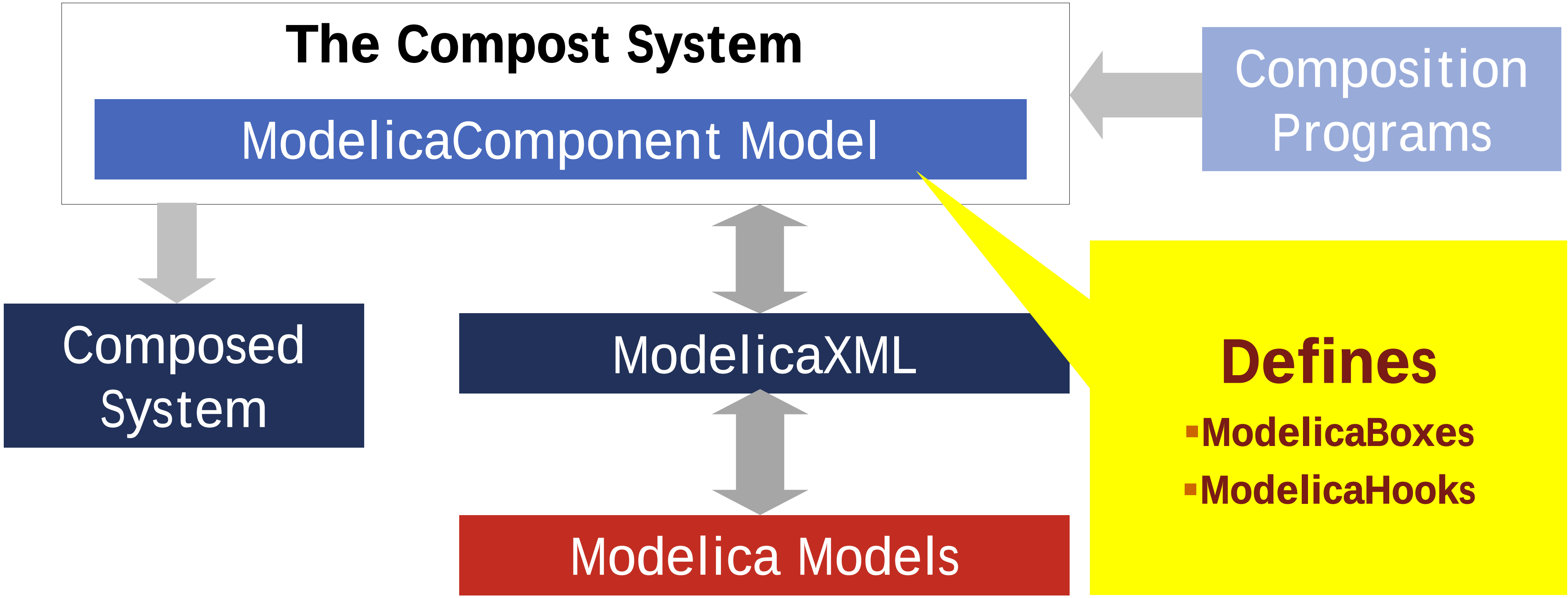
Invasive Software Composition

- Composition of black box components
  - Hard to adapt components to context
  - Generates possibly inefficient systems



- Invasive Software Composition
  - Composition system can *see inside the components*
  - *Components are hidden behind a composition interface* Components are composed using a *composition language*
  - Components can be configured by changing their actual code at variation points (boxes and hooks) defined by the *component model*

Invasive Composition for Modelica



The Compost System

ModelicaComponent Model

Composition Programs

Composed System

ModelicaXML

Modelica Models

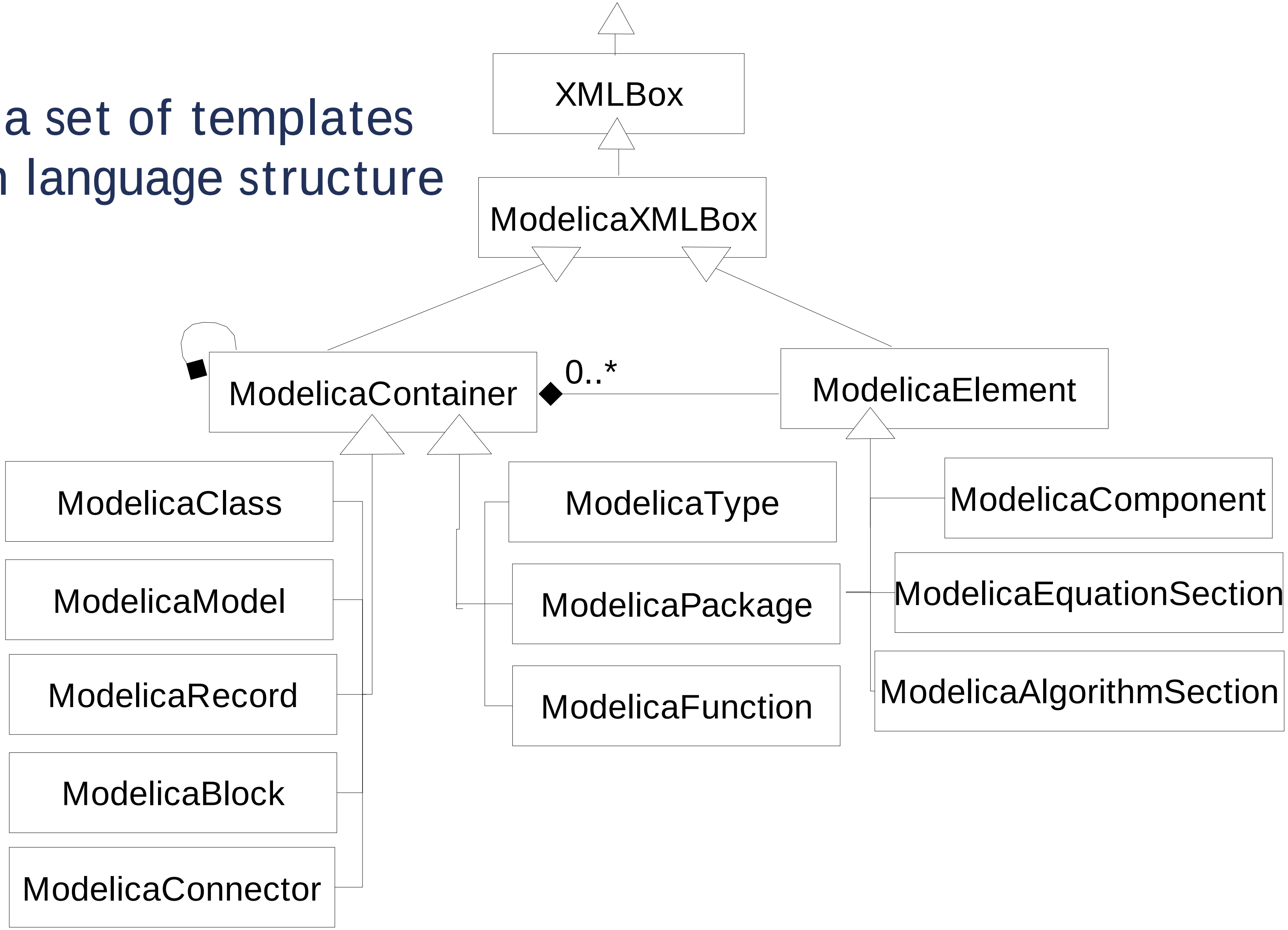
Defines

- **ModelicaBoxes**
- **ModelicaHooks**

- The benefit of Invasive Modelica Composition
  - Generation of different version of models from product specifications
  - Automatic configuration of models using external sources
  - Fine grain support for library developers
  - Refactoring, reverse engineering, etc

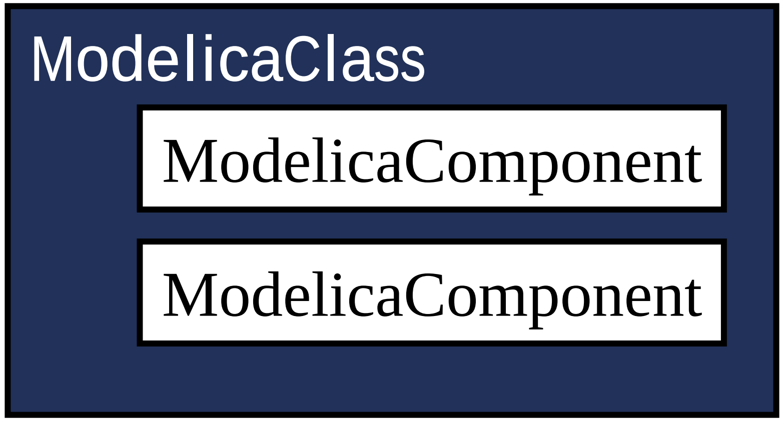
Modelica Component Model – Boxes

Defines a set of templates for each language structure



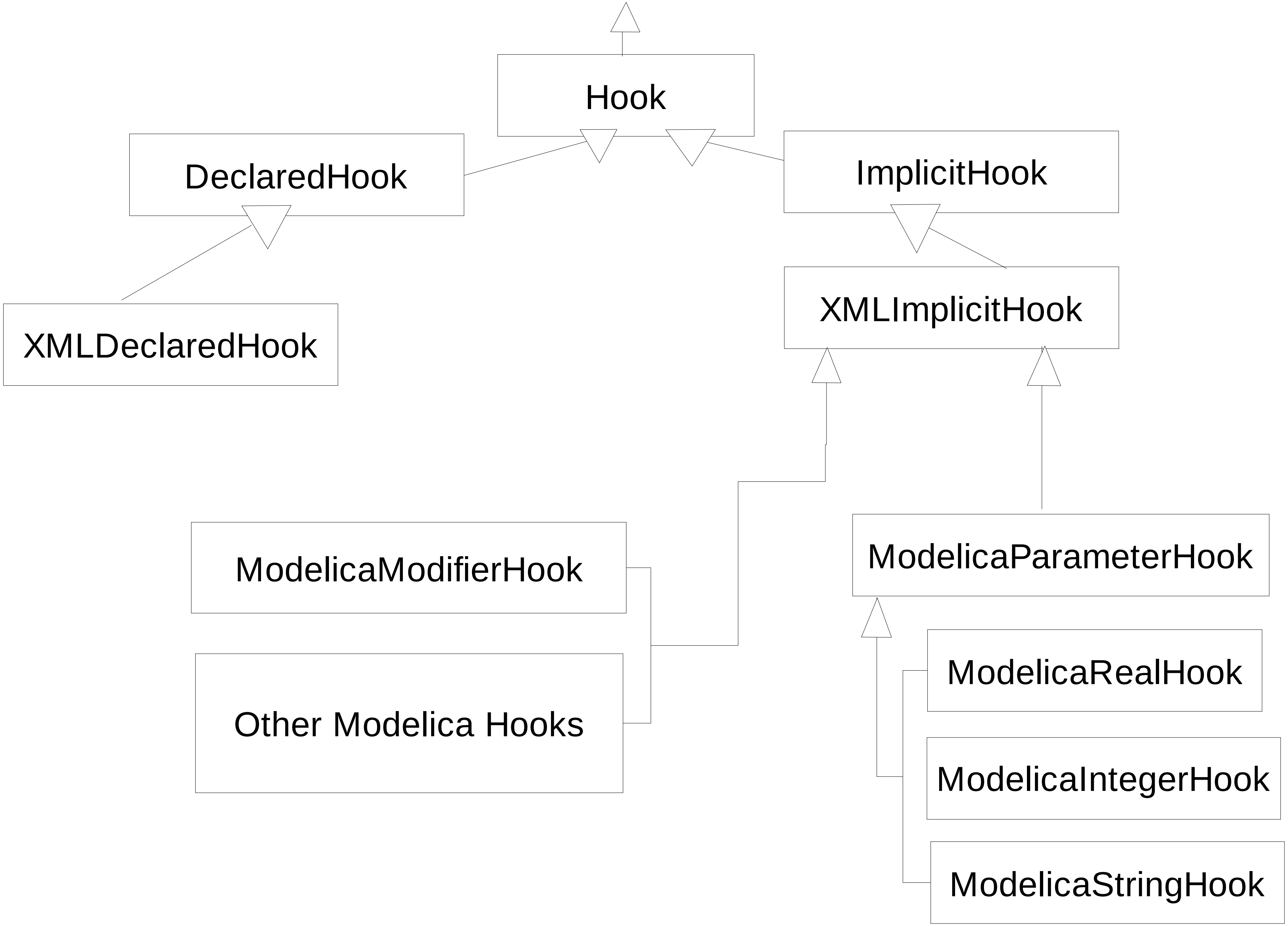
Example Box Hierarchy

```
<definition ident="Engine" restriction="class">
   <component visibility="public” variability="parameter"
              type="Integer" ident="cylinders">
      <modification_equals>
         <integer_literal value="4"/>
      </modification_equals>
   </component>
   <component visibility="public" type="Cylinder" ident="c">
      <array_subscripts>
         <component_reference ident="cylinders"/>
      </array_subscripts>
   </component>
</definition>
```



```
class Engine
   parameter Integer
             cylinders = 4;
   Cylinder c[cylinders];
end Engine;
```

Modelica Component Model – Hooks



Hook

DeclaredHook

ImplicitHook

XMLDeclaredHook

XMLImplicitHook

ModelicaModifierHook

Other Modelica Hooks

ModelicaParameterHook

ModelicaRealHook

ModelicaIntegerHook

ModelicaStringHook

Example: Hooks

```
<component visibility="public” variability="parameter"
           type="Integer" ident="cylinders">
   <modification_equals>
       <integer_literal value="4"/>
   </modification_equals>
</component>
```

ModelicaParameterHook
name
value

```
parameter Integer
            cylinders = 4;
```

```
<definition ident=”NewEngine" restriction="class">
   <extends type=”Engine”>
   ....
</definition>
```

```
class NewEngine
  extends Engine;
  ....
end NewEngine;
```

```
<definition ident=”Engine” restriction="class">
  <extract>
    <component>..</component> ...
  </extract>
</definition>
```

Composition Programs: Mixin

```
ModelicaCompositionSystem cs =
     new ModelicaCompositionSystem();
ModelicaClass resultBox =
     cs.createModelicaClass(”Result.mo.xml”);
ModelicaClass firstMixin =
     cs.createModelicaClass(”Class1.mo.xml”);
ModelicaClass secondBox =
     cs.createModelicaClass(”Class2.mo.xml”);
   resultBox.mixin(firstMixin);
   resultBox.mixin(secondMixin);
   resultBox.print();
```

Outline

- Product Design Environments

- Meta-Modeling
    - Modelica Meta-Model
    - Invasive Composition of Modelica
    - Model-driven Product Design using Modelica
        - Product Design based on Function-Means decomposition
        - Integration with Modelica for Early Design Validation

- Meta-Programming
    - Debugging of Natural Semantics Specifications

- Conclusions and Future Work

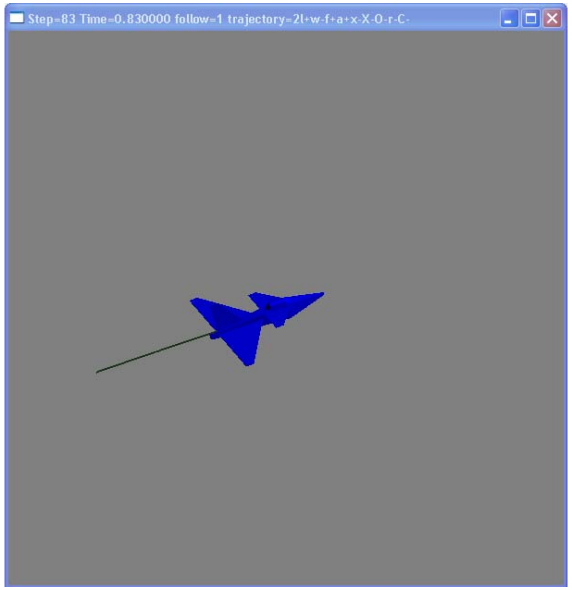
Model-Driven Product Design

- Product design
  - product concept modeling and evaluation
  - physical modeling and simulation

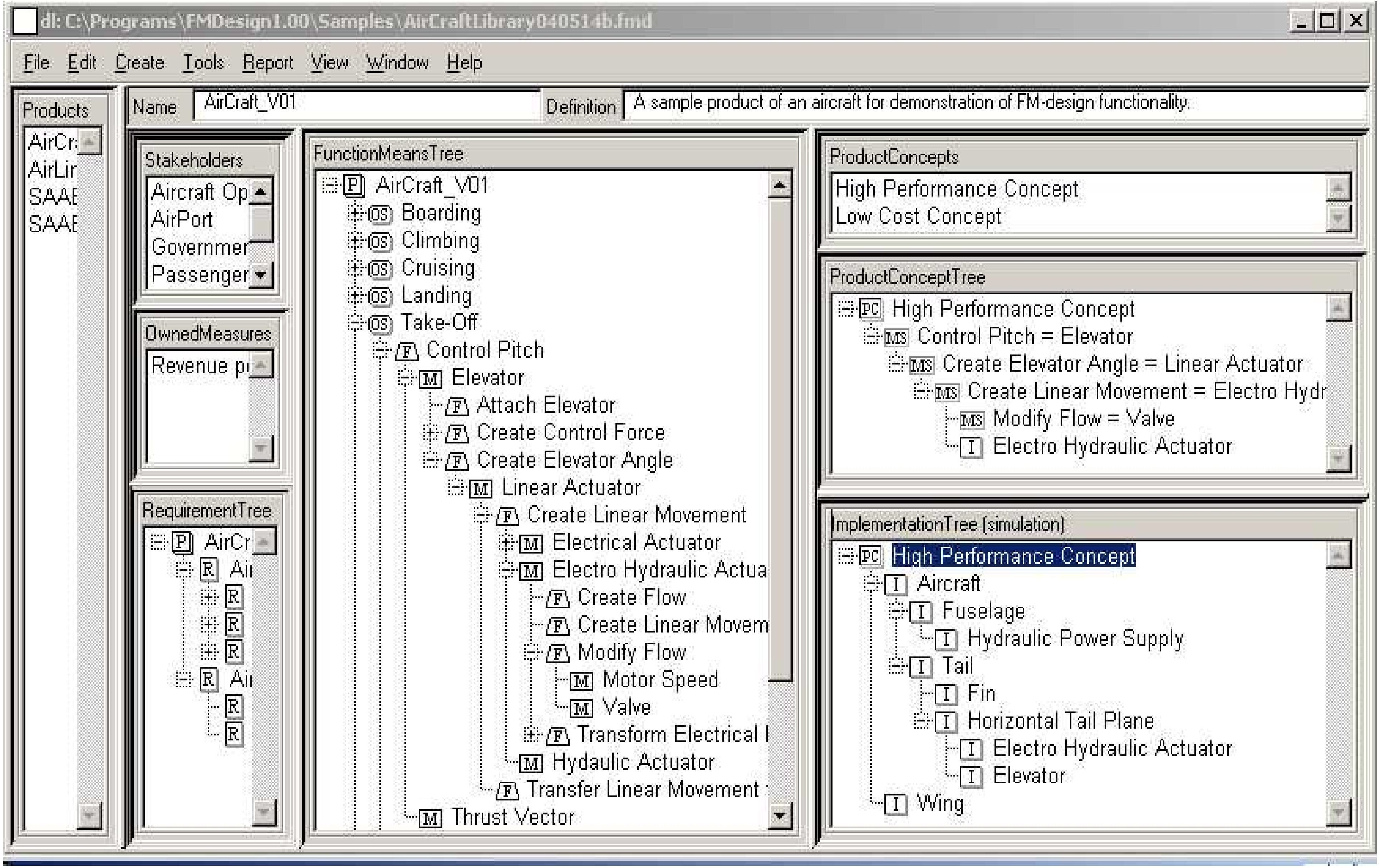
- Need for integration of
  - conceptual modeling tools and
  - modeling and simulation tools

Example: design phases of an Aircraft Product

- Aircraft conceptual model in FMDesign
  - decomposition of the aircraft into functions and means
  - mapping between means and Modelica simulation components
  - simulation of various design choices
  - choosing the best design choice using the simulation results



FMDesign – Conceptual Product Design



Name: AirCraft_V01

Definition: A sample product of an aircraft for demonstration of FM-design functionality.

Products:
- AirCr
- AirLir
- SAAE
- SAAE

Stakeholders:
- Aircraft Op
- AirPort
- Governmer
- Passenger

OwnedMeasures:
- Revenue p

RequirementTree:
- P AirCr
  - R Ai
    - R
    - R
    - R
  - R Ai
    - R
    - R

FunctionMeansTree:
- P AirCraft_V01
  - OS Boarding
  - OS Climbing
  - OS Cruising
  - OS Landing
  - OS Take-Off
    - F Control Pitch
      - M Elevator
        - F Attach Elevator
        - F Create Control Force
        - F Create Elevator Angle
          - M Linear Actuator
            - F Create Linear Movement
              - M Electrical Actuator
              - M Electro Hydraulic Actua
                - F Create Flow
                - F Create Linear Movem
                - F Modify Flow
                  - M Motor Speed
                  - M Valve
                - F Transform Electrical I
              - M Hydaulic Actuator
            - F Transfer Linear Movement
      - M Thrust Vector

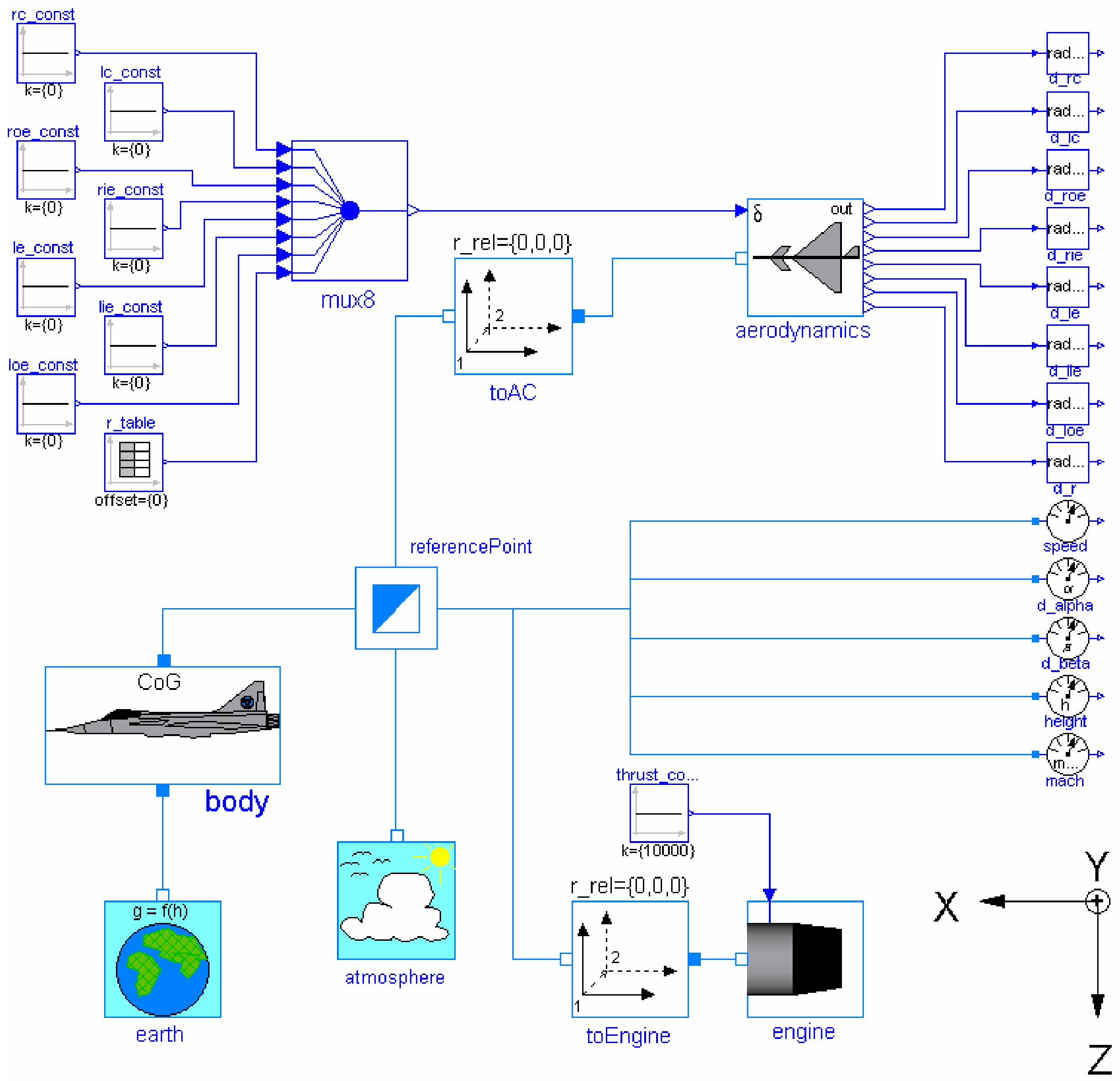
ProductConcepts:
- High Performance Concept
- Low Cost Concept

ProductConceptTree:
- PC High Performance Concept
  - MS Control Pitch = Elevator
    - MS Create Elevator Angle = Linear Actuator
      - MS Create Linear Movement = Electro Hydr
        - MS Modify Flow = Valve
        - I Electro Hydraulic Actuator

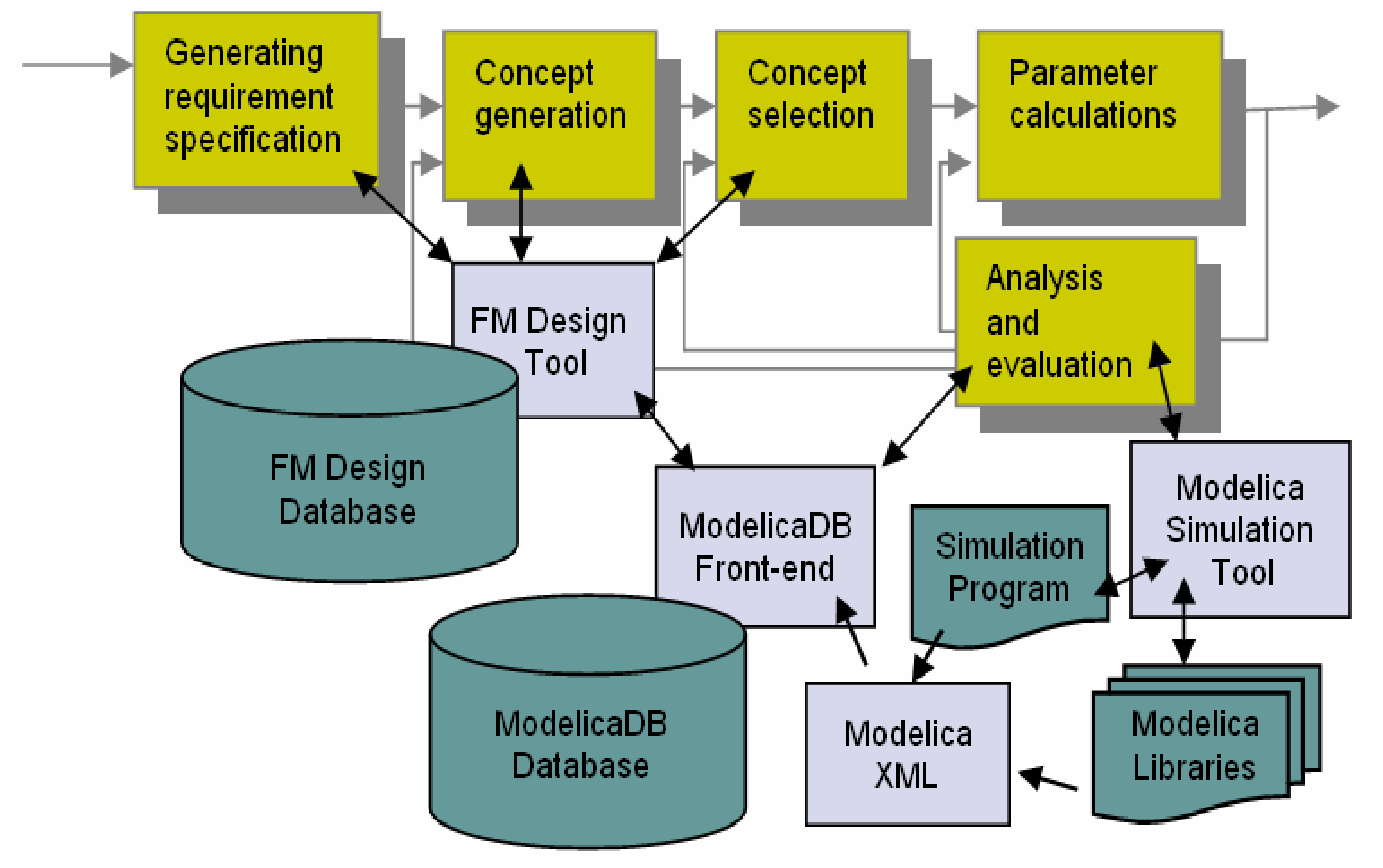
ImplementationTree (simulation):
- PC High Performance Concept
  - I Aircraft
    - I Fuselage
      - I Hydraulic Power Supply
    - I Tail
      - I Fin
      - I Horizontal Tail Plane
        - I Electro Hydraulic Actuator
        - I Elevator
    - I Wing

Courtesy of Olof Johansson. Developed in cooperation with Peter Krus, IKP

Simulation Components for an Aircraft Product



A Framework for Product Design



Framework Integration Tools

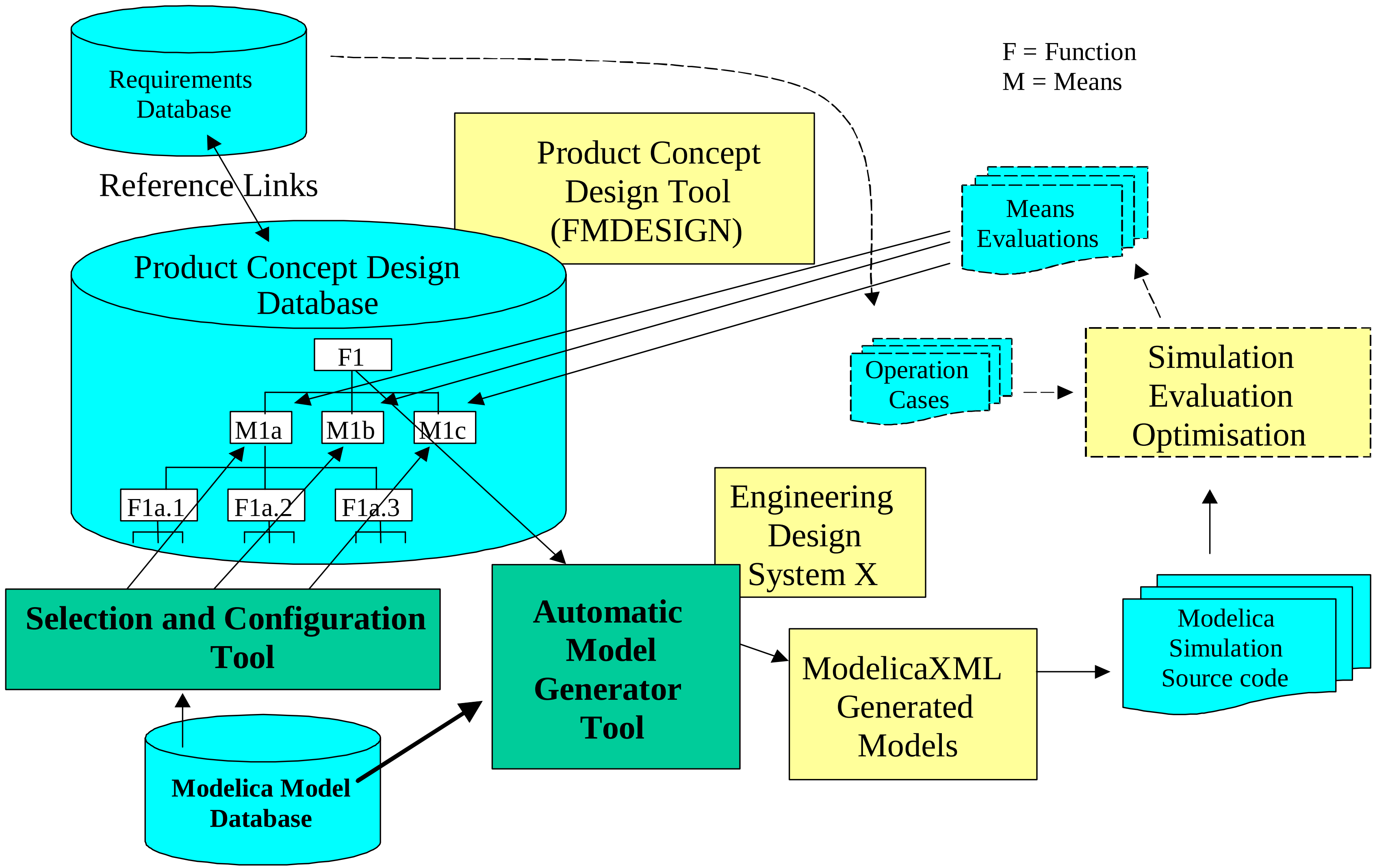
- ModelicaDB – Modelica Model Database
  - is populated with simulation models by importing their ModelicaXML representation
  - is a simulation models repository
  - provides search and organizational features
  - flexibility, scalability and collaborative development

Framework Integration Tools (cont)

- The Selection and Configuration Tool
  - searches ModelicaDB for simulation models
  - calls modeling tools for creating/editing simulation models
  - configuration dialogs for selected simulation models for specific means implementation

- The Automatic Model Generator Tool
  - generates Modelica models of the product
  - calls external simulation tools for simulation
  - feeds the simulation results back to the designer to help him/her choose the best design choice

Architecture Overview



Outline

- Product Design Environments

- Meta-Modeling
  - Modelica Meta-Model
  - Invasive Composition of Modelica
  - Model-driven Product Design using Modelica

- Meta-Programming
  - Debugging of Natural Semantics Specifications
    - Natural Semantics and Relational Meta-Language
    - Debugging framework

- Conclusions and Future Work

ModelicaXML Representation - Problems

- Problems
  - XML can only express *syntax*
  - No easy way to automatically handle *semantics*
- Possible solutions when expressing semantics
  - use markup languages developed by Semantic Web to express some of the Modelica semantics
  - use other formalisms like Natural Semantics

Meta-Programming

- Meta-Programs
  - programs that manipulate other programs

- Natural Semantics, a formalism widely used for specification of programming language aspects
  - type systems
  - static, dynamic and translational semantics
  - few implementations in real systems

- Relational Meta-Language (RML)
  - a system for generating efficient executable code from Natural Semantics specifications
  - fast learning curve, used in teaching and specification of languages such as: Java, Modelica, MiniML, etc.
  - previously no support for debugging

Natural Semantics vs. Relational Meta-Language

Natural Semantics formalism

integers:

$$v \in Int$$

expressions (abstract syntax):

$$\begin{aligned} e \in Exp ::= {} & v \\ & \mid e1 + e2 \\ & \mid e1 - e2 \\ & \mid e1 * e2 \\ & \mid e1 / e2 \\ & \mid -e \end{aligned}$$

$$(1) \quad v \Rightarrow v$$

$$(2) \quad \frac{e1 \Rightarrow v1 \;\; e2 \Rightarrow v2 \;\; v1 + v2 \Rightarrow v3}{e1 + e2 \Rightarrow v3}$$

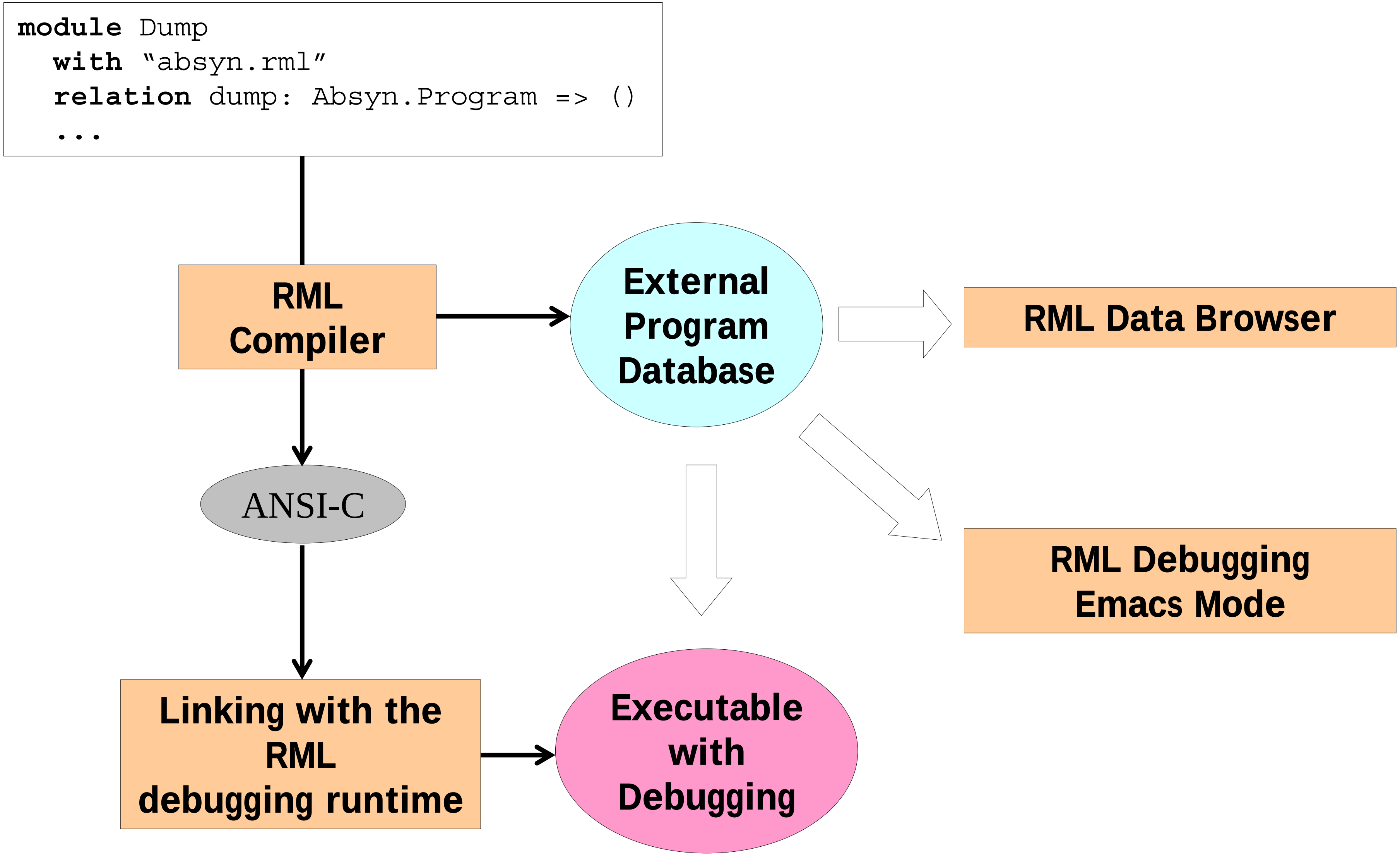
Relational Meta-Language

```
module exp1:
  (* Abstract syntax of language Exp1 *)
  datatype Exp =  INTconst of int
               |  ADDop    of Exp * Exp
               |  SUBop    of Exp * Exp
               |  MULop    of Exp * Exp
               |  DIVop    of Exp * Exp
               |  NEGop    of Exp
  relation eval: Exp => int
end

relation eval: Exp => int  =

axiom eval(INTconst(ival)) => ival

rule  eval(e1) => v1 & eval(e2) => v2 & v1 + v2 => v3
      ------------------------------------------------
      eval( ADDop(e1, e2) ) => v3
 rule  eval(e1) => v1 & eval(e2) => v2 & v1 - v2 => v3
      ------------------------------------------------
      eval( SUBop(e1, e2) ) => v3
…
end (* eval *)
```

The Need for RML Debugging

- Facilitate language learning
  - run, stop and inspect features
- Large specifications are hard to debug
  - Example: OpenModelica compiler
    - **43** packages
    - **57083** lines of code and counting
    - **4054** functions
    - **132** data structures

Debugger Implementation - Overview



```
module Dump
  with “absyn.rml”
  relation dump: Absyn.Program => ()
  ...
```

RML Compiler

External Program Database

RML Data Browser

RML Debugging Emacs Mode

ANSI-C

Linking with the RML debugging runtime

Executable with Debugging

Debugger Implementation - Instrumentation

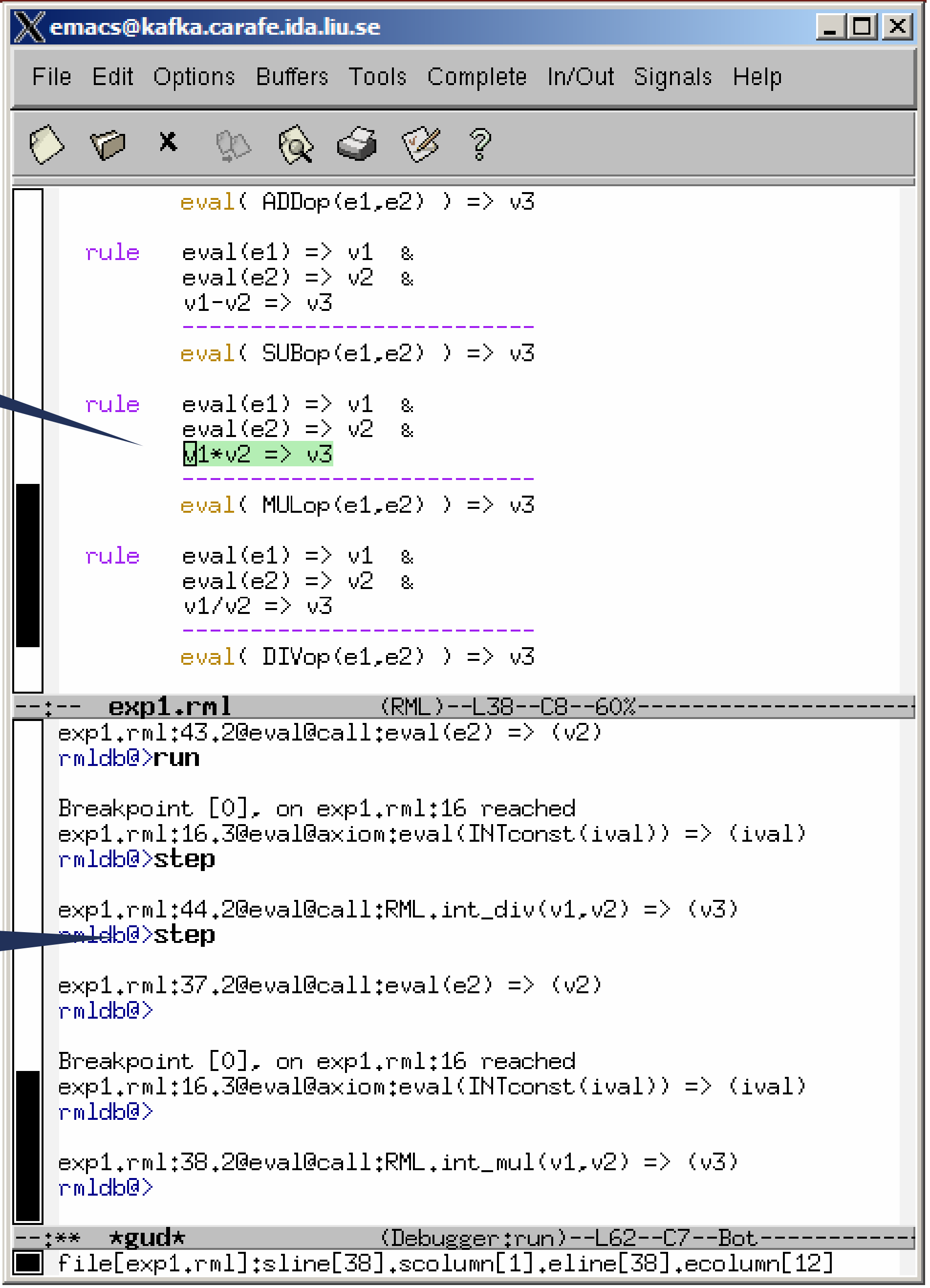
```
(* Evaluation semantics of Exp1 *)
relation eval: Exp => int  =
axiom eval(INTconst(ival)) => ival

rule  eval(e1) => v1 &
      eval(e2) => v2 &
      v1 + v2 => v3
      ---------------------------
      eval( ADDop(e1, e2) ) => v3
 ...
end (* eval *)
```

```
(* Evaluation semantics of Exp1 *)
relation eval: Exp => int  =
axiom eval(INTconst(ival)) => ival

rule     RML.debug_push_in01("e1",e1) &
         RML.debug(...) &
         eval(e1) => (v1) &
         RML.debug_push_out01("v1",v1) &
         RML.debug_push_in01("e2",e2) &
         RML.debug(...) => () &
         eval(e2) => (v2) &
         RML.debug_push_out01("v2",v2) &
         RML.debug_push_in02("v1",v1,"v2",v2
   ) &           RML.debug(...) &
         RML.int_add(v1,v2) => (v3)

         -----------------------------------
   ----
         eval(ADDop(e1,e2)) => (v3)
 ...
end (* eval *)
```

Debugger Functionality (1)

Breakpoints

Stepping and Running



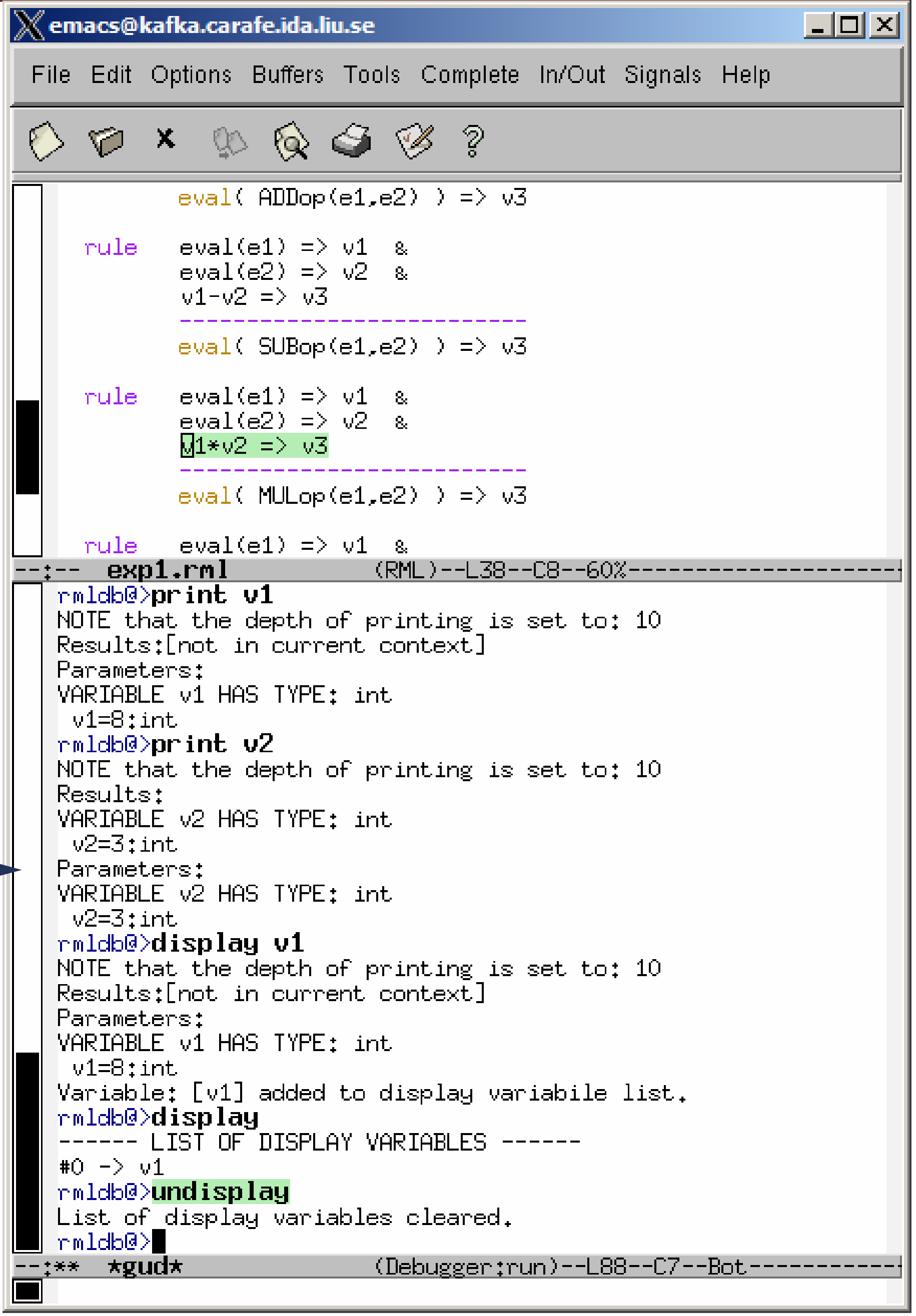
```
        eval( ADDop(e1,e2) ) => v3

  rule  eval(e1) => v1  &
        eval(e2) => v2  &
        v1-v2 => v3
        ---------------------------
        eval( SUBop(e1,e2) ) => v3

  rule  eval(e1) => v1  &
        eval(e2) => v2  &
        v1*v2 => v3
        ---------------------------
        eval( MULop(e1,e2) ) => v3

  rule  eval(e1) => v1  &
        eval(e2) => v2  &
        v1/v2 => v3
        ---------------------------
        eval( DIVop(e1,e2) ) => v3
--:--  exp1.rml          (RML)--L38--C8--60%--------------------
exp1.rml:43.2@eval@call:eval(e2) => (v2)
rmldb@>run

Breakpoint [0], on exp1.rml:16 reached
exp1.rml:16.3@eval@axiom:eval(INTconst(ival)) => (ival)
rmldb@>step

exp1.rml:44.2@eval@call:RML.int_div(v1,v2) => (v3)
rmldb@>step

exp1.rml:37.2@eval@call:eval(e2) => (v2)
rmldb@>

Breakpoint [0], on exp1.rml:16 reached
exp1.rml:16.3@eval@axiom:eval(INTconst(ival)) => (ival)
rmldb@>

exp1.rml:38.2@eval@call:RML.int_mul(v1,v2) => (v3)
rmldb@>
--:**  *gud*          (Debugger:run)--L62--C7--Bot-----------
file[exp1.rml]:sline[38],scolumn[1],eline[38],ecolumn[12]
```

Debugger Functionality (2)

- Additional functionality
  - viewing status information
  - printing backtrace information (stack trace)
  - printing call chain
  - setting debugger defaults
  - getting help

Examining data

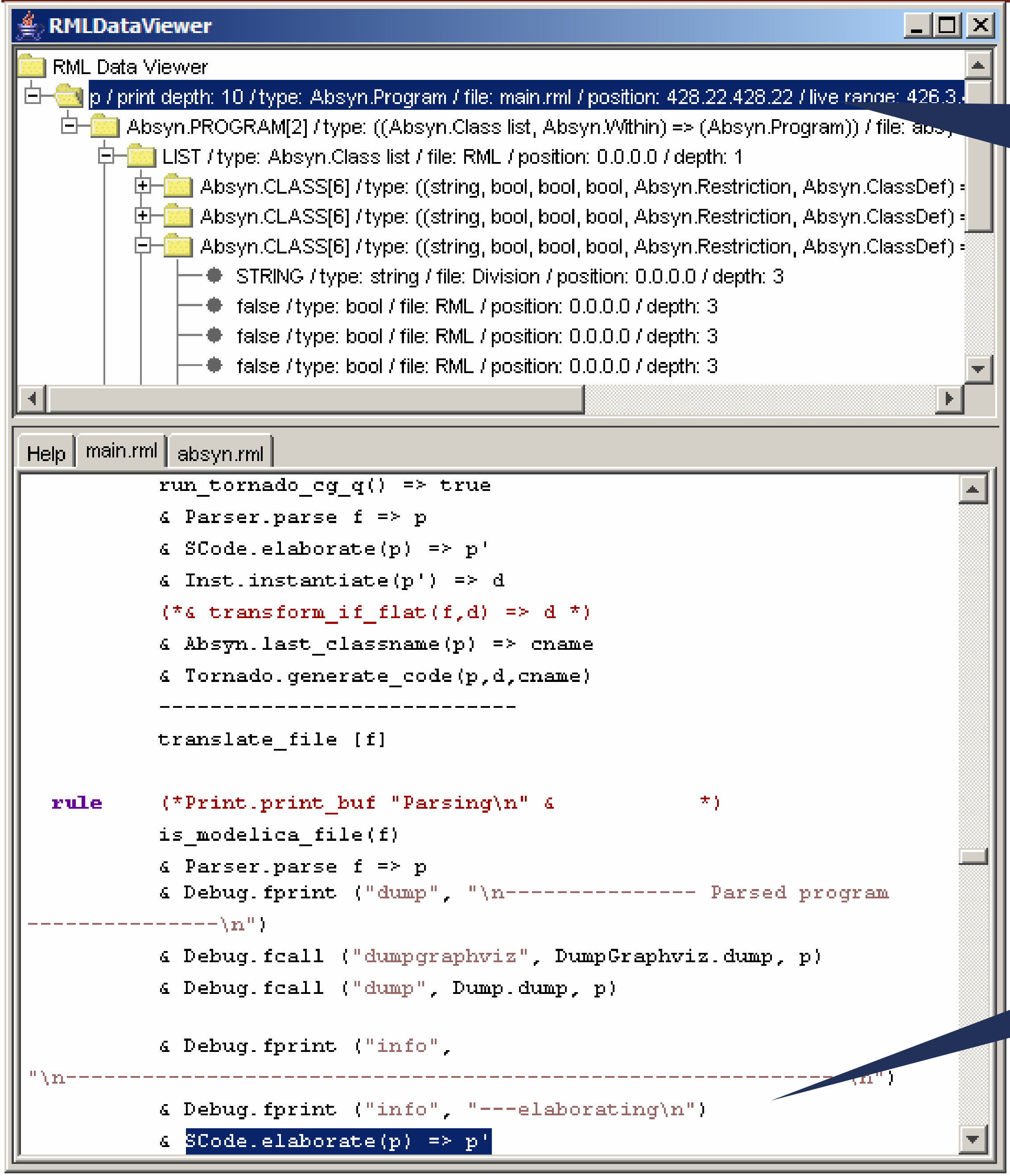
- printing variables
- sending variables to an external browser



```
          eval( ADDop(e1,e2) ) => v3

  rule    eval(e1) => v1  &
          eval(e2) => v2  &
          v1-v2 => v3
          ---------------------------
          eval( SUBop(e1,e2) ) => v3

  rule    eval(e1) => v1  &
          eval(e2) => v2  &
          v1*v2 => v3
          ---------------------------
          eval( MULop(e1,e2) ) => v3

  rule    eval(e1) => v1  &
--:--  exp1.rml           (RML)--L38--C8--60%--------------------
```

```
rmldb@>print v1
NOTE that the depth of printing is set to: 10
Results:[not in current context]
Parameters:
VARIABLE v1 HAS TYPE: int
 v1=8:int
rmldb@>print v2
NOTE that the depth of printing is set to: 10
Results:
VARIABLE v2 HAS TYPE: int
 v2=3:int
Parameters:
VARIABLE v2 HAS TYPE: int
 v2=3:int
rmldb@>display v1
NOTE that the depth of printing is set to: 10
Results:[not in current context]
Parameters:
VARIABLE v1 HAS TYPE: int
 v1=8:int
Variable: [v1] added to display variabile list.
rmldb@>display
------ LIST OF DISPLAY VARIABLES ------
#0 -> v1
rmldb@>undisplay
List of display variables cleared.
rmldb@>
--:**  *gud*          (Debugger:run)--L88--C7--Bot-----------
```

Browser for RML Data Structures (1)



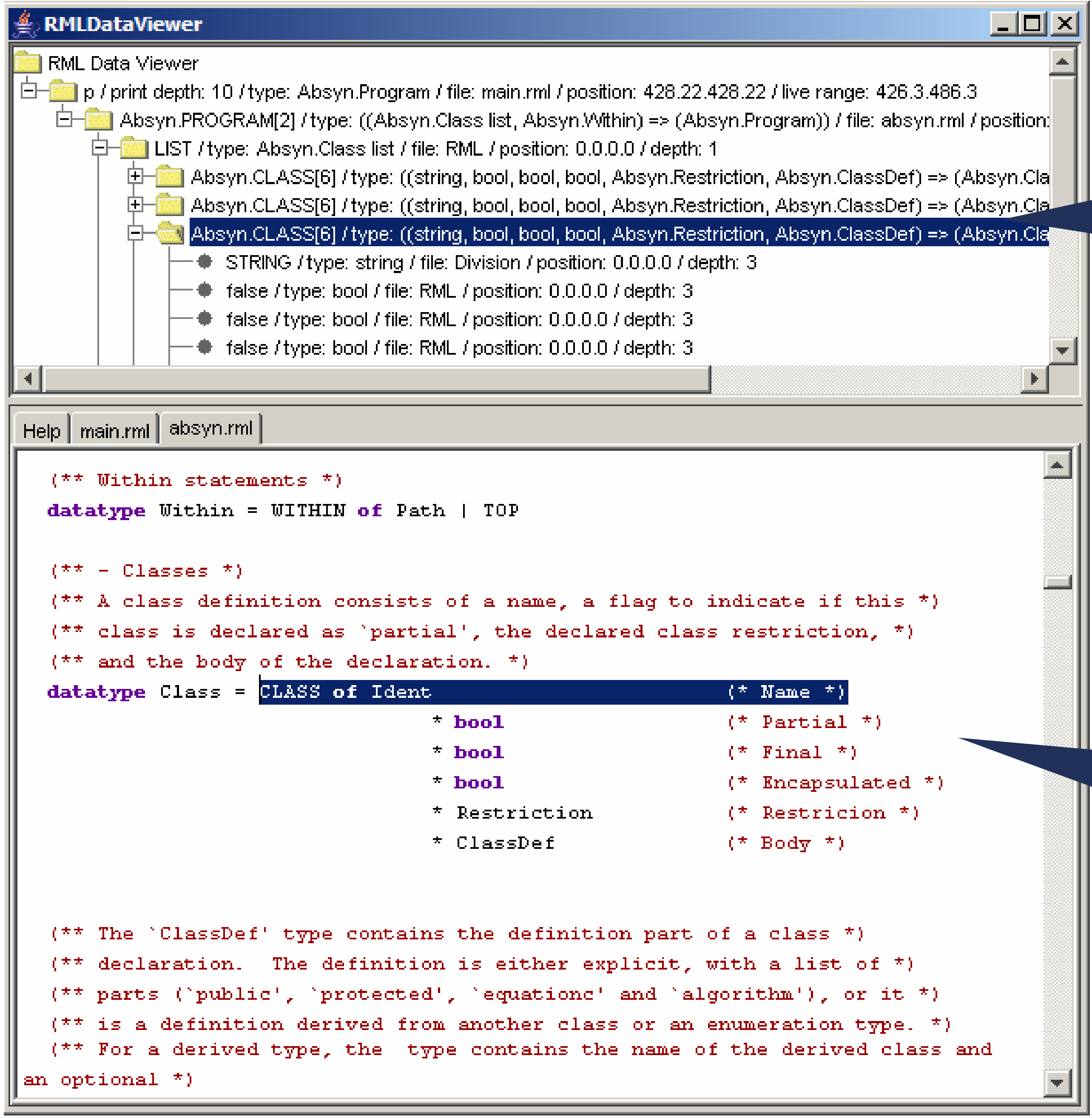
```
RML Data Viewer
p / print depth: 10 / type: Absyn.Program / file: main.rml / position: 428.22.428.22 / live range: 426.3.
  Absyn.PROGRAM[2] / type: ((Absyn.Class list, Absyn.Within) => (Absyn.Program)) / file: abs
    LIST / type: Absyn.Class list / file: RML / position: 0.0.0.0 / depth: 1
      Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) =
      Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) =
      Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) =
        STRING / type: string / file: Division / position: 0.0.0.0 / depth: 3
        false / type: bool / file: RML / position: 0.0.0.0 / depth: 3
        false / type: bool / file: RML / position: 0.0.0.0 / depth: 3
        false / type: bool / file: RML / position: 0.0.0.0 / depth: 3
```

```
          run_tornado_cg_q() => true
          & Parser.parse f => p
          & SCode.elaborate(p) => p'
          & Inst.instantiate(p') => d
          (*& transform_if_flat(f,d) => d *)
          & Absyn.last_classname(p) => cname
          & Tornado.generate_code(p,d,cname)
          ----------------------------
          translate_file [f]

  rule    (*Print.print_buf "Parsing\n" &           *)
          is_modelica_file(f)
          & Parser.parse f => p
          & Debug.fprint ("dump", "\n--------------- Parsed program
---------------\n")
          & Debug.fcall ("dumpgraphviz", DumpGraphviz.dump, p)
          & Debug.fcall ("dump", Dump.dump, p)

          & Debug.fprint ("info",
"\n-------------------------------------------------------------\n")
          & Debug.fprint ("info", "---elaborating\n")
          & SCode.elaborate(p) => p'
```

Variable value inspection

Current Execution Point

Browser for RML Data Structures (2)



RML Data Viewer
p / print depth: 10 / type: Absyn.Program / file: main.rml / position: 428.22.428.22 / live range: 426.3.486.3
Absyn.PROGRAM[2] / type: ((Absyn.Class list, Absyn.Within) => (Absyn.Program)) / file: absyn.rml / position:
LIST / type: Absyn.Class list / file: RML / position: 0.0.0.0 / depth: 1
Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) => (Absyn.Cla
Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) => (Absyn.Cla
Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) => (Absyn.Cla
STRING / type: string / file: Division / position: 0.0.0.0 / depth: 3
false / type: bool / file: RML / position: 0.0.0.0 / depth: 3
false / type: bool / file: RML / position: 0.0.0.0 / depth: 3
false / type: bool / file: RML / position: 0.0.0.0 / depth: 3

```
  (** Within statements *)
  datatype Within = WITHIN of Path | TOP

  (** - Classes *)
  (** A class definition consists of a name, a flag to indicate if this *)
  (** class is declared as `partial', the declared class restriction, *)
  (** and the body of the declaration. *)
  datatype Class = CLASS of Ident                        (* Name *)
                                 * bool                  (* Partial *)
                                 * bool                  (* Final *)
                                 * bool                  (* Encapsulated *)
                                 * Restriction           (* Restricion *)
                                 * ClassDef              (* Body *)


  (** The `ClassDef' type contains the definition part of a class *)
  (** declaration.  The definition is either explicit, with a list of *)
  (** parts (`public', `protected', `equationc' and `algorithm'), or it *)
  (** is a definition derived from another class or an enumeration type. *)
  (** For a derived type, the  type contains the name of the derived class and
an optional *)
```

Data structure browsing

Data structure definition

Conclusions

- Meta-Modeling
  - Alternative Modelica Representation (ModelicaXML)
    - Conform to a Meta-Model for Modelica
  - Invasive Composition of Modelica
    - Model configuration and adaptation
    - Based on ModelicaXML and a Component Model for Modelica
  - Model-driven Product Design using Modelica
    - Integration of conceptual product modeling with modeling and simulation tools
    - Flexibility, scalability
    - Uses ModelicaXML as a middleware

- Meta-Programming
  - Debugging of Natural Semantics Specifications
    - large specification debugging (OpenModelica Compiler)
  - Debugging of MetaModelica models

Future Work

- Meta-Modelica Compiler
    - Unified equation-based meta-modeling and meta-programming specification language for both:
        - language models and physical system models
    - Work in progress, first version based on RML
    - Compilation of a Modelica extended with features such as:
        - pattern matching, tree structures, lists, tuples, etc.
    - More Meta-Modeling capabilities (constraints on models, etc)

- Improvement of the debugging framework
- Experimenting with an Eclipse-IDE that integrates these tools

- Add new features to the Relational Meta-Language system

End

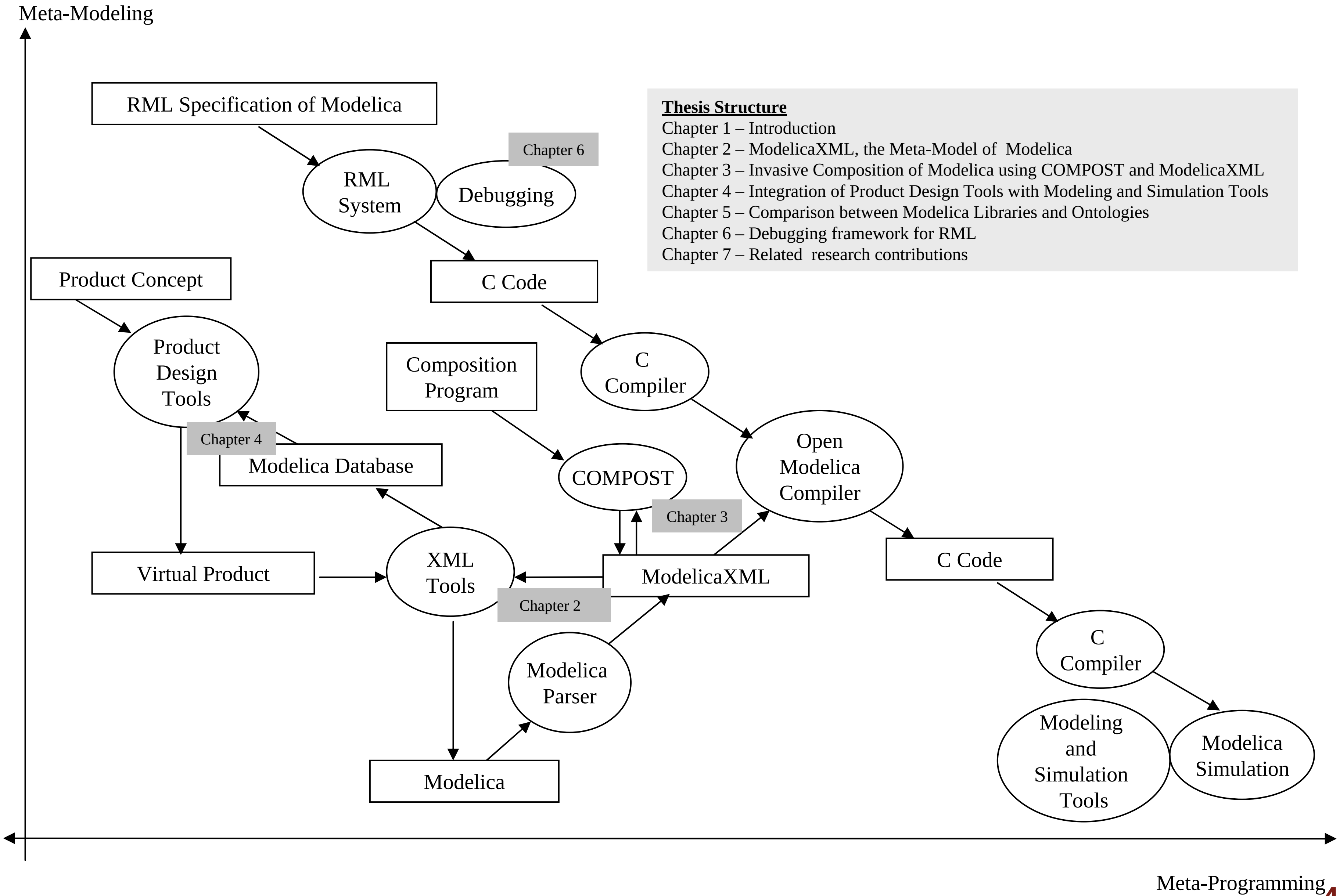
Thank you!

Questions?

Resources

- ModelicaXML and ModelicaOWL
  - http://www.ida.liu.se/~adrpo/modelica/xml
  - http://www.ida.liu.se/~adrpo/modelica/owl
- The Invasive Composition System Compost
  - http://www.the-compost-system.org/
- Relational Meta-Language (RML)
  - http://www.ida.liu.se/~pelab/rml
- MetaModelica Compiler (MMC)
  - http://www.ida.liu.se/~adrpo/mmc
- Licentiate Thesis
  - http://www.ida.liu.se/~adrpo/lic

Thesis Structure



Meta-Modeling

RML Specification of Modelica
RML System
Debugging
Chapter 6
C Code
C Compiler
Open Modelica Compiler
C Code
C Compiler
Modelica Simulation
Modeling and Simulation Tools
Product Concept
Product Design Tools
Chapter 4
Modelica Database
Composition Program
COMPOST
Chapter 3
Virtual Product
XML Tools
ModelicaXML
Chapter 2
Modelica Parser
Modelica

Meta-Programming

Thesis Structure
Chapter 1 – Introduction
Chapter 2 – ModelicaXML, the Meta-Model of Modelica
Chapter 3 – Invasive Composition of Modelica using COMPOST and ModelicaXML
Chapter 4 – Integration of Product Design Tools with Modeling and Simulation Tools
Chapter 5 – Comparison between Modelica Libraries and Ontologies
Chapter 6 – Debugging framework for RML
Chapter 7 – Related research contributions