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

Gearbox

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**
  - Declarative language
  - Multi-domain modeling
  - Everything is a class
  - Visual component programming

- **Modelica Association**
  - [http://www.modelica.org](http://www.modelica.org)
Meta-Programming

Meta-Modelica 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
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
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
Applications of ModelicaXML Representation

- Interoperability and transformation
- Easy access from any programming language
- Query facilities
- Documentation generation
- Validation of models using the meta-model
- 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
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
Defines a set of templates for each language structure
<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;

Example Box Hierarchy
Example: Hooks

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

parameter Integer
    cylinders = 4;

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

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

ModelicaParameterHook
- name
- value

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

<definition ident="Engine" restriction="class">
    <extract>
        <component>..</component> ...
    </extract>
</definition>
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
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
Simulation Components for an Aircraft Product
- **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
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

- Product Concept Design Database
  - Selection and Configuration Tool
  - Automatic Model Generator Tool
    - Modelica Model Database
    - ModelicaXML Generated Models
  - Operation Cases
    - Means Evaluations
    - Simulation Evaluation Optimisation
  - Refrence Links
    - Product Concept Design Tool (FMDESIGN)
      - F1
        - F1a.1
        - F1a.2
        - F1a.3
      - M1a
      - M1b
      - M1c

F = Function
M = Means

- Requirements Database

Means
Operation
Cases
Simulation
Evaluation
Optimisation
Modelica
Simulation
Source code
ModelicaXML
Generated
Models
- 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 formalism

integers:

\[ v \in Int \]

expressions (abstract syntax):

\[ e \in Exp ::= \]

\[ v \]

\[ | e1 + e2 \]

\[ | e1 - e2 \]

\[ | e1 \cdot e2 \]

\[ | e1 / e2 \]

\[ | -e \]

(1) \[ v \Rightarrow v \]

(2) \[ \frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad v1+v2 \Rightarrow v3}{e1+e2 \Rightarrow v3} \]

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

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

rule eval(e1) => v1 &
    eval(e2) => v2 &
    v1 + v2 => v3
------
end (* eval *)
Debugger Functionality (1)

Breakpoints

Stepping and Running
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
Variable value inspection

Current Execution Point
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
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

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