2nd International Workshop on Equation-Based Object-Oriented Languages and Tools at ECOOP 2008, July 8, Paphos, Cyprus

A Static Aspect Language for Modelica Models

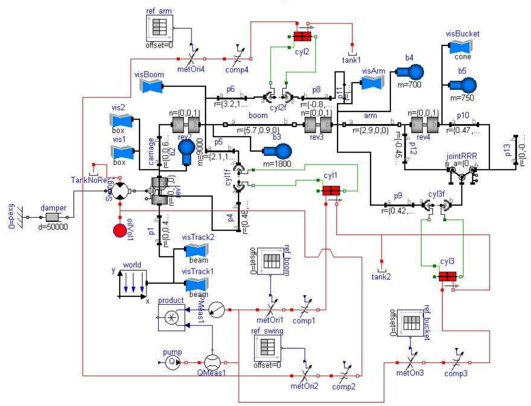
Malte Lochau (lochau@ips.cs.tu-bs.de)
Henning Günther (h.guenther@tu-bs.de)

Contents

Contents

- **Introduction**
  - Modelica and Quality Requirements
  - Principles of Aspect Orientation
- **Static Aspect Language**
  - Rule Syntax
  - Evaluation Semantics
  - Variables and Type System
- **Implementation Framework**
- **Conclusion**

- Modelica
  - Multi-discipline mathematical modeling and simulation of complex physical systems
  - Object-oriented
  - Equation-based (declarative)
  - …



- Modelica 3: „balanced models“ concept
  - Restrictions / design rules for increased model quality
  - E.g. balanced connector property:

„… the number of flow variables in a connector must be identical to the number of non-causal non-flow variables …“

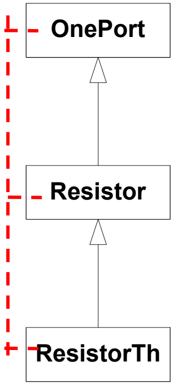
- Generalizing: Quality Requirements
  - Modeling restrictions, design rules, conventions, policies, …
  - Domain specific, non-functional, …

„… flow variables shall be named with a `_flow` postfix …“

„… inheritance hierarchies deeper than 4 are to be avoided …“

➢ Exceed expressiveness of Modelica language capabilites

➢ Requirements superpose or *crosscut* model components and hierarchies




```
partial model OnePort
...
end OnePort;

model Resistor extends OnePort
...
end Resistor;

model ResistorTh extends Resistor
...
end ResistorTh;
```

Objectives:

- Formalism for concise specification and automated evaluation of quality requirements
- Querying Modelica models, matching point(s) meeting certain criteria
- Rule checking by negation:

  `<forbidden property> => <error message>`
- Model manipulation / transformation
- Modelica specific approach
- …

- Aspect Orientation [Kizcales et. al]:

  „Modularization and integration of crosscutting concerns in existing systems"

  „… is *Obliviousness* and *Quantification*"

  „… applied to *procedural-like* programming languages"

- Aspect Orientation for EOO languages?
  - *dynamic* vs. *static* aspects
  - ➢ Structural properties of Modelica models are largely stated at compile-time

➢ Static Aspects for Modelica models:

„In a model M, wherever condition C arises, perform action A"

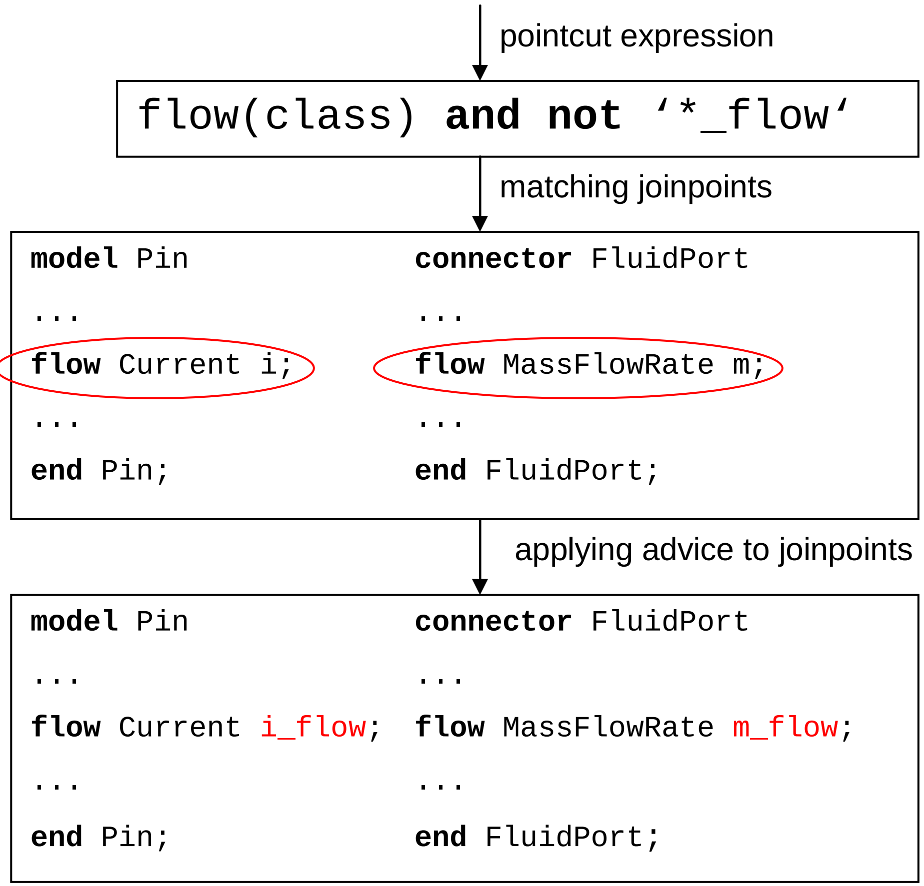
Aspects Terminology

- Aspects: Encapsulation of crosscutting concerns

```
<Pointcut> => <Advice>;
```

- *Pointcut*: Expression matching specific elements (joinpoints) of a model
- *Joinpoints*: Model entities considered in an aspect
- *Advice*: Action(s) to be applied to joinpoints

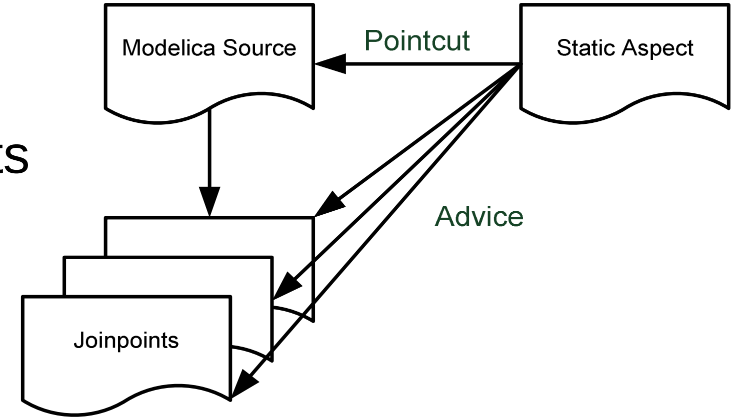
➢ *Weaving*: „Injecting“ advices of an aspect to the original model



- **Introduction**
  - Modelica and Quality Requirements
  - Principles of Aspect Orientation
- **Static Aspect Language**
  - Rule Syntax
  - Evaluation Semantics
  - Variables and Type System
- **Implementation Framework**
- **Conclusion**

Syntactic Structure of Static Aspects

```
<Pointcut> => <Advice>;
```



- Pointcut language:
  - Expression terms matching joinpoints
  - Primitives: predicates, relations (Modelica-specific)
  - Operators for term compositions (crosscutting entities)

- Advice language:
  - Actions applied to each joinpoint matching the pointcut
  - E.g. high-level programming language code

Syntax of the Pointcut Language

- Expressions applied to the set of all relevant joinpoints in a model
- Step-wise refinements by *unary* predicates, *binary* relations, and *operators*

p ::=	u	operators
|	b(p)	
|	p **and** p	
|	p **or** p	
|	**not** p	
|	**exists** b : p	
|	**forall** b : p	
|	p **equals** p	
|	p **subset** p	
|	p **less** p	
|	**<relop>** n b	
…		

u ::=	<id>	unary
|	'<pattern>'	

b ::=	<id>	binary
|	b**+**	
|	b**+**<n>	
|	p **product** p	
|	p **product-d** p	

p:	pointcut expression
u:	unary pointcut expression
b:	binary pointcut relation
n:	natural number
relop:	on of <,<=, =, !=, >, >=
id:	identifier
pattern:	name pattern expression

Primitives of the Pointcut Language

- Predefined Modelica primitives matching subsets / pairs of Modelica entities
  - *Unary* primitives: High-level structural entities for model organization, i.e. *Class types*
    - `class, model, connector, block, …`
    - `partialType, finalType, localType, …`

  - *Binary* relations: Inspecting properties of model elements
    - Class type *members*
      - `member(p), publicMember(p), replMember(p), …`
      - `flow(p), parameter(p), modifier(p), …`
    - Class type *inheritance*
      - `derivedType(p), baseType(p), subType(p), …`
    - Class type *behavior*
      - `equation(p), connectEquation(p), unknown(p), …`
    - …

- *Crosscutting* model entities: Correlation / combination of pointcut expressions
- Operators working on joinpoint sets
  - Logical *combination* of joinpoint sets
    `and, or, not, less, …`
  - Naming pattern for accessing elements by their *names*
    `‘*_flow‘`
  - Cardinalities: *Number* of joinpoints matching pointcuts
    `<relop> n b`
  - Quantification: Conditions on a *range* of values
    `forall, exists, …`
  - *Transitive closure* of binary relations
    `derivedType+`

Examples: Pointcut Expression

- Are there partial types that are never derived?

```
partialType and not baseType(class)
```

- Are there package declarations with less than 5 members?

```
package and ( < 5 componentMember )
```

- Are there blocks only having output members?

```
forall primitiveMember : output(block)
```

- Evaluation of pointcuts: $P : J_M \rightarrow \mathcal{P}(J_M)$
  - Pointcut expression P
  - Modelica model specification M
  - Set of *all* joinpoints J_M present in M

- Element-wise reasoning of joinpoint sets by considering the stated conditions of P
  - Evaluation preceeds from inwards to outwards
  - Stepwise refinement of the resulting joinpoint set via unary and binary pointcut evaluation

Evaluation Rules

operators	
$P[\![p]\!] = U[\![u]\!]$	
$P[\![b(p)]\!] = \{ j_1 \mid (j_1, j_2) \in B[\![b]\!], j_\in \in P[\![p]\!] \}$	
…	
$P[\![p_1$ **and** $p_2]\!] = P[\![p_1]\!] \cap P[\![p_2]\!]$	
$P[\![p_1$ **or** $p_2]\!] = P[\![p_1]\!] \cup P[\![p_2]\!]$	
$P[\![$ **not** $p_1]\!] = \{ j \mid j \notin P[\![p_1]\!] \}$	
$P[\![p_1$ **less** $p_2]\!] = P[\![p_1]\!] \setminus P[\![p_2]\!]$	
…	
$P[\![$ **forall** $b : p]\!] = \{ j_2 \mid \forall (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!] \}$	
$P[\![$ **exists** $b : p]\!] = \{ j_2 \mid \exists (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!] \}$	
$P[\![p_1$ **product** $p_2]\!] = \{ (j_1, j_2) \in P[\![p_1]\!] \times P[\![p_2]\!] \}$	
…	
$P[\![b+]\!] = \{ (j_1, j_2) \mid \exists (j_1, j_2), \dots, (j_{k-1}, j_k) \in B[\![b]\!] \}$	

unary

U : unary pointcut $\to \mathcal{P}(J_m)$

$U[\![id]\!] = \{ j \mid j \in J_M$ matching id $\}$

$U[\![$'pattern'$]\!] = \{ j \mid j \in J_M$ matching 'pattern'$\}$

binary

B : binary relation $\to \mathcal{P}(J_M \times J_M)$

$B[\![id]\!] = \{ (j_1, j_2) \mid j_1, j_2 \in J_M$ related pair w.r.t. id$\}$

Advices

- Advices are executed for each joinpoint of the result set of a pointcut evaluation

  - Error reports for rule checking by negation:

```
<pointcut> => “violated naming convention“
```

  - Syntax for iterating joinpoints from the result set:

```
<pointcut> => “violated naming convention in “ +
              ResultSet.nextItem().getName();
```

  - … arbitrary pieces of program code, e.g. subsequent model manipulations by referencing *AST* nodes of joinpoints …

- Parameterizing pointcut expressions by a set of variable deklarations Φ being bound to joinpoint sets:

$$[\ \Phi\ ]p : \text{Pointcut}_{\Phi}, \text{ where } \Phi = \{\ v_1 := p_1, \ldots, v_n := p_n\ \}$$

  - Enhanced evaluation semantics: (nested) “*for-each*“ loops over the set of joinpoint combinations in the variables of *p*
  - Scoping: Bindings for *p* are adopted to all subterms of *p,* e.g.:

$$P[\![\ p_1\ \textbf{and}\ p_2\ ]\!]_{\Phi} = P[\![\ p_1\ ]\!]_{\Phi} \cap P[\![\ p_2\ ]\!]_{\Phi}$$

  - Further application: Passing variables to the *advice* part…

Balanced connector property:

„… the number of flow variables in a connector must be identical to the number of non-causal non-flow variables …“

- How to compare cardinalities within the *same* entity?
  - Parameterized pointcut expressions: *names* for joinpoint sets

```
[v := connector](!= flow(v)
                   (primitiveMember(v)
                    less (flow(v) or input(v) or output(v)
                          or parameter(v) or constant(v)
                   )
                );
=> „Balanced connector property violated in“ + v.getName();
```

Type System

- Pointcut type system
  - Types according to related Modelica elements, e.g. class types, member, equation, …
  - Ensuring „soundness" of pointcut expressions
  - Determining types of joinpoints matching pointcut expressions using typing rules, e.g.:

$$\frac{p_1 : \sigma_1 \qquad p_2 : \sigma_2}{p_1 \ \mathbf{or} \ p_2 \ : \ \sigma_1 \cup \sigma_2}$$

- Further applications:
  - Enforcing „reasonable" parameter types for binary primitives:

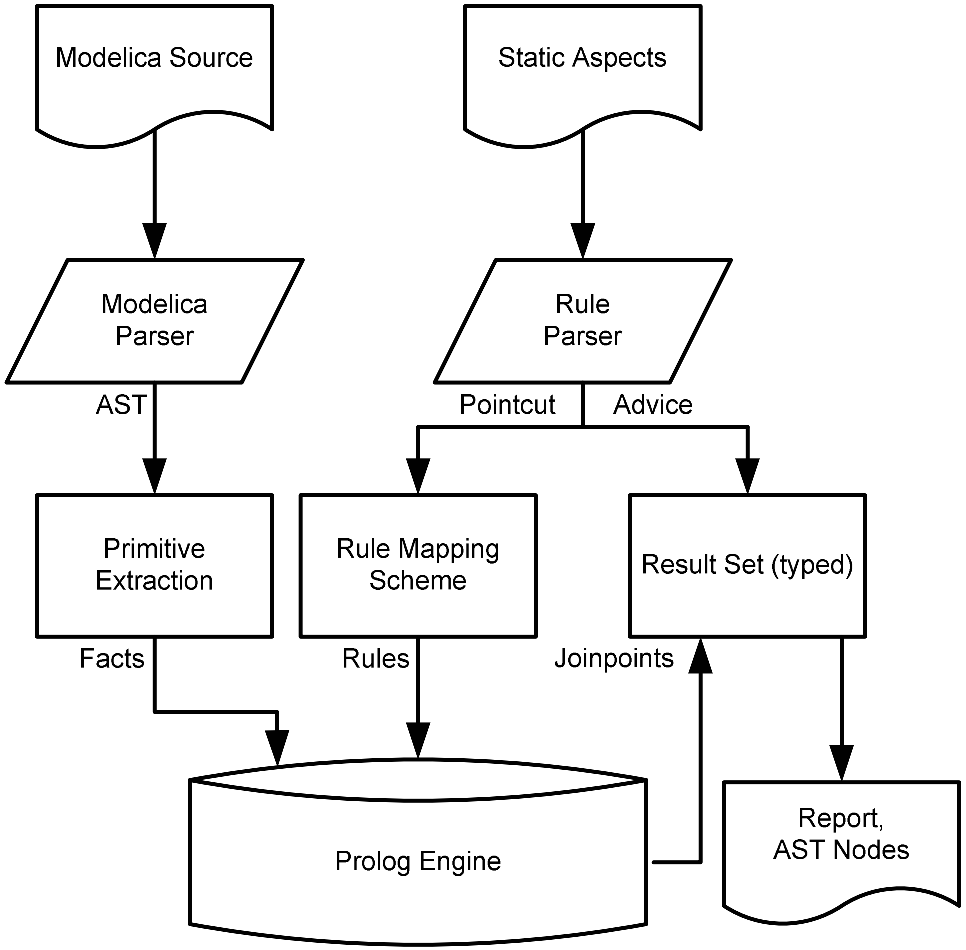
```
equation(connector) // wrong parameter type
```

  - …

Contents

- **Introduction**
  - Modelica and Quality Requirements
  - Principles of Aspect Orientation
- **Static Aspect Language**
  - Rule Syntax
  - Evaluation Semantics
  - Variables and Type System
- **Implementation Framework**
- **Conclusion**

Framework Architecture



- Logic Meta Programming:
  - Strong relationship between AOP and logic programming
  - *User language* for concise rule specification *(problem oriented)*: static aspect language
  - *Implementation language* for efficient rule evaluation: logic programming language, e.g. Prolog

- Example: subclass relation
  - Primitives: Facts extracted from source models

```
model(m1,'OnePort').
model(m2,'Resistor').
derive(m2,m1).
```

  - Pointcuts: Rules, e.g. for transitive closure calculation

```
derivedType(Sub,Sup)      :- derive(Sub,Sup).
derivedType(Sub,X)        :- derive(Sub,X),
                             derivedType(X,Sup).
```

- Introduction
  - Modelica and Quality Requirements
  - Principles of Aspect Orientation
- Static Aspect Language
  - Rule Syntax
  - Evaluation Semantics
  - Variables and Type System
- Implementation Framework
- **Conclusion**

Conclusion and Future Work

- Summary:
  - Static aspect language for Modelica models: formal syntax and evaluation semantics for pointcuts
  - Variable concept and type system
  - Implementation framework based on the logic meta programming approach
- Future Work:
  - Finishing the implementation
  - Evaluation of
    - the expressiveness of the aspect language
    - the efficiency of rule evaluations
  - AOSD for Modelica?

Thank you for your attention.

Questions?