A Static Aspect Language for Modelica Models

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

• Introduction
  • Modelica and Quality Requirements
  • Principles of Aspect Orientation

• Static Aspect Language
  • Rule Syntax
  • Evaluation Semantics
  • Variables and Type System

• Implementation Framework

• Conclusion
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 …“
Quality Requirements for Modelica Models

- 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 capabilities

- Requirements superpose or crosscut model components and hierarchies

```model
OnePort

Resistor

ResistorTh
```

```partial model
OnePort

model
Resistor

extends
OnePort

end

model
ResistorTh

extends
Resistor

end
```
Objectives:

- Formalism for concise specification and automated evaluation of quality requirements
- Querying Modelica models, matching point(s) meeting certain criteria
- Rule checking by negation:
  \[ \text{<forbidden property>} \Rightarrow \text{<error message>} \]
- 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

\[
\text{<Pointcut>} \Rightarrow \text{<Advice>};
\]

- **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
Contents

• Introduction
  • Modelica and Quality Requirements
  • Principles of Aspect Orientation

• Static Aspect Language
  • Rule Syntax
  • Evaluation Semantics
  • Variables and Type System

• Implementation Framework

• Conclusion
<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
Expressions applied to the set of all relevant joinpoints in a model

Step-wise refinements by *unary* predicates, *binary* relations, and *operators*

<table>
<thead>
<tr>
<th>p ::= u operators</th>
<th>u ::= &lt;id&gt; unary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>p ::= b(p)</td>
<td>b ::= &lt;id&gt; binary</td>
</tr>
<tr>
<td>p ::= p and p</td>
<td></td>
</tr>
<tr>
<td>p ::= p or p</td>
<td></td>
</tr>
<tr>
<td>p ::= not p</td>
<td></td>
</tr>
<tr>
<td>p ::= exists b : p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>p ::= forall b : p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>p ::= p equals p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>p ::= p subset p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>p ::= p less p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;relop&gt; n b</td>
<td></td>
</tr>
</tbody>
</table>

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?

\[
\text{partialType and not baseType(class)}
\]

- Are there package declarations with less than 5 members?

\[
\text{package and ( < 5 componentMember )}
\]

- Are there blocks only having output members?

\[
\text{forall primitiveMember : output(block)}
\]
Semantics of the Pointcut Language

Introduction

Static Aspect Language

Implementation

Conclusion

- 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 proceeds from inwards to outwards
  - Stepwise refinement of the resulting joinpoint set via unary and binary pointcut evaluation
Evaluation Rules

U : unary pointcut → \mathcal{P}(J_m)

U[ id ] = \{ j | j \in J_M \text{ matching id} \}

U[’pattern’] = \{ j | j \in J_M \text{ matching ’pattern’} \}

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

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

The operators are:

\[ P[p] = U[u] \]

\[ P[b(p)] = \{ j_1 | (j_1, j_2) \in B[b], j_2 \in P[p] \} \]

\[ \ldots \]

\[ P[p_1 \text{ and } p_2] = P[p_1] \cap P[p_2] \]

\[ P[p_1 \text{ or } p_2] = P[p_1] \cup P[p_2] \]

\[ P[\text{not } p_1] = \{ j | j \notin P[p_1] \} \]

\[ P[p_1 \text{ less } p_2] = P[p_1] \setminus P[p_2] \]

\[ \ldots \]

\[ P[\text{forall } b : p] = \{ j_2 | \forall (j_1, j_2) \in B[b], j_1 \in P[p] \} \]

\[ P[\text{exists } b : p] = \{ j_2 | \exists (j_1, j_2) \in B[b], j_1 \in P[p] \} \]

\[ P[p_1 \text{ product } p_2] = \{ (j_1, j_2) \in P[p_1] \times P[p_2] \} \]

\[ \ldots \]

\[ P[b+] = \{ (j_1, j_2) | \exists (j_1, j_2), \ldots, (j_{k-1}, j_k) \in B[b] \} \]
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 …
Parameterized Pointcuts

• Parameterizing pointcut expressions by a set of variable declarations $\Phi$ being bound to joinpoint sets:

$$[\Phi]p : \text{Pointcut}_\Phi,$$

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.:

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

• Further application: Passing variables to the advice part...
Example: Rule Checking by Negation

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();
```
• 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.:
    \[ p_1 : \sigma_1 \quad p_2 : \sigma_2 \]
    \[ p_1 \text{ or } p_2 : \sigma_1 \cup \sigma_2 \]

• Further applications:
  • Enforcing „reasonable“ parameter types for binary primitives:
    \[ \text{equation}(\text{connector}) \quad // \quad \text{wrong parameter type} \]
  • …
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

Introduction

Static Aspect Language

Implementation

Conclusion

Modelica Source

Modelica Parser

AST

Primitive Extraction

Facts

Rule Mapping Scheme

Rules

Report, AST Nodes

Prolog Engine

Result Set (typed)

Joinpoints

Rule Parser

Static Aspects

Pointcut

Advice
• 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).
Contents

- 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?