A Static Aspect Language for Modelica Models

Malte Lochau Henning Günther

Institute for Programming and Reactive Systems, TU Braunschweig, Germany,
{m.lochau,h.guenther}@tu-bs.de

Abstract

With the introduction of the new Modelica major version 3, innovations mainly consist of further model restrictions for increased model quality. In addition, developers often want to ensure the compliance to further requirements early in the development cycle. Mostly emerging as domain specific conventions that often crosscut model structures, according checking mechanisms are required that are detached from the core language. In this paper, a declarative language is presented for specifying and evaluating quantified rules for static model properties. Based on aspect-oriented programming, the language allows for concise and expressive model inspections and a variable and typing concept facilitate subsequent model manipulations. A nascent implementation framework is proposed, based on the logic meta programming paradigm, thus leading to efficient and scalable aspect processing applicable as model query engine for an AOP Modelica Compiler.

Keywords Early Checking, Aspect Orientation, Modelica Model Inspection

1. Introduction

The Modelica description standard [2] proposes a modern multi-discipline language for component-based mathematical modeling and simulation of complex physical systems (see e.g. [12, 21]). Its equation-based, object-oriented, and declarative nature allows for hierarchical specification of system structure and behavior and smooth integration, evolution and reuse of developed components. The innovations of the version 3 of the Modelica Specification [2] mainly constitute further restrictions, e.g. the locally balanced model property [17], hence aiming at design rules for increased model quality.

Moreover, modeling follows established practices according to a substantial set of rules and properties. Generalizing, a developer wants to be able to specify, maintain, and analyze arbitrary structural model properties accompanying all development phases. A multitude of requirements, especially non-behavioral quality properties and properties not evident in testing cannot be adequately expressed solely using language constructs of Modelica as they exceed the expressiveness of object-oriented principles. As an example, the locations within a model to be considered for checking design policies such as

"For clarity reasons, inheritance hierarchies deeper than 4 are to be avoided"

are scattered throughout the code, i.e. the formulation of such a demand crosses model composition hierarchies. Also the balanced model concept of Modelica 3 [17] consists of a set of entangled demands, e.g.

"The number of flow variables in a connector must be identical to the number of non-causal non-flow variables"

which correlates properties of connectors.

Besides, design criteria considering chains of several models/connectors might be of interest, as well as further coding conventions such as correct library usage, domain specific patterns, and simple naming conventions, e.g.

"Flow variables shall be named with a flow postfix"

Therefore, an overall mechanism is desirable for expressing, modularizing, and finally ensuring compliance with especially application-domain specific design rules and patterns. Allowing for arbitrary model inspection with respect to those requirements, certain kinds of errors can be avoided from the start. For this purpose, aspect-oriented programming (AOP) offers a promising approach: Superposing the object-oriented paradigm and being oblivious with respect to the underlying core language, aspects allow for declarative quantification and modularization of concerns that crosscut the component structure and functionality of models.

As the multitude of properties of Modelica models are already fixed at definition/compile time – especially almost all structural characteristics – *static aspects* at source code level seem to provide a sufficient approach for handling a wide range of aspect-oriented concerns for an early and efficient checking process inside the development loop. Therefore, a domain-specific static language is required for specifying aspect rules that refer to static model entities constituted by fundamental Modelica language units. According to the aspect-oriented paradigm, this language must allow for intuitive and quantified formulation and integration (*weaving*) of aspects without explicitly affecting the underlying "host" language and existing models under consideration. Instead, a smooth description, isolation,

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

Copyright is held by the author/owner(s). The proceedings are published by Linköping University Electronic Press. Proceedings available at:
http://www.ep.liu.se/ecp/029/

EOOLT 2008 website:
http://www.eoolt.org/2008/

composition, and reuse of aspects is made possible for a collection of (connected) Modelica models, their inner components and behavioral descriptions, therefore specifying the properties emerging from the demanded requirements.

In this paper, syntax and semantics of a rule language for static aspects is presented in terms of expressions (point cuts) matching specific locations (join points) in Modelica models. Reflecting basic entities of such models, the language enables model queries at source code level, e.g. inspection and correlation of classes, connectors, and equations. The syntactical structure is based on predefined Modelica language primitives serving as "atoms" and operators for complex term construction. The design of the language aims at expressive and declarative encoding of a concise and succinct set of static design rules. The semantics includes proposals for the usage of variable bindings and point cut types for accessing and manipulating model elements. Although this paper focuses on a comprise definition of the static aspect language rather than an extensive case study, nevertheless some examples as well as a general discussion of its application is provided. An implementation framework for the aspect language is proposed that is under construction. Herein, logic programming principles are used for efficient rule processing by an integrated evaluation engine.

This paper is organized as follows: In Section 2, a brief overview on aspect-orientation and further related work is given. In Section 3, a complete and formalized specification of syntax and semantics is presented for a domain specific static aspect language for Modelica models, and an extension for a variable concept and type system is mentioned. The implementation framework of the language is proposed in Section 4, and some proposals for sample applications are depicted in Section 5. Section 6 concludes.

2. Aspect-Orientation and Related Work

Aspect-oriented programming (AOP) [10, 15] is motivated by the observation, that nowadays abstraction perceptions for logical system structuring and design mainly refer to the notions of hierarchy and (de-) composition. Accordingly, current programming and modeling language paradigms such as the object-oriented principles of Modelica are designed from this perspective. Nevertheless, in many cases there are concerns that crosscut a composition structure, so-called *aspects*, which are therefore contradicting the means of expression of such languages.

Concepts of aspect-oriented programming aim at capturing such crosscutting concerns by appropriate modularization constructs called *point cuts*. Point cuts are expressions matching well-defined combinations of correlated points within programs/models of the underlying component-based language, so-called *join points*. In the *advice* part of an aspect definition, actions/manipulations can be defined to be applied to the matching join points. Depending on the point in time at which aspects are considered, hence *weaved* to the host program/model, two categories can be distinguished: static and dynamic aspects, thus either at definition/compile time or run time. For instance, an implementation of dynamic aspects for Java is provided by AspectJ [14], whereas the JTL approach [7] allows for the specification of static aspects for Java using *query by example*.

The static aspect language for Modelica proposed in this paper is mainly inspired by the PDL described in [16] which is based on principles of description logics [3, 4]. The adoption of the approach to the Modelica language refers to the syntactical and semantic structure of Modelica version 3 [2]. As a major enhancement, a concept for variable bindings is proposed.

Previous efforts for checking properties of Modelica models have been made: In [22], mainly the technical issues are discussed for analyzing Modelica model properties such as naming conventions and inheritance complexity. However, no integrated approach for expressing such properties is mentioned. Furthermore, analysis techniques for specific facets of Modelica models can be found, e.g. in [5] and [6], where the determination of under and over constrained systems of equations is presented.

The implementation structure presented in this paper is based on the logic meta programming approach described in [23], where the intimate correlation between aspect orientation and logic programming is outlined. As already proposed by the OpenModelica Project [11] and the MetaModelica Language [20], an ANTLR parser is used to create an Abstract Syntax Tree (AST) for examining Modelica models under consideration. Thereupon, the RML/MetaModelica approach [19] can also be seen as a kind of strongly typed logic programming language for investigating Modelica models, but it is not linked to aspect oriented principles. Finally, a first attempt of an AOP Compiler focusing on merging AspectJ-like *inter-type declarations* into the AST of Modelica models can be found in [1].

3. Static Aspect Language for Modelica

In this Section, a domain-specific static aspect language is defined by giving a complete formalized syntax and related semantics for obtaining join points in Modelica models via quantified point cut expressions. Furthermore, variable capabilities are proposed for adequate binding of (typed) join points according to related Modelica language entities.

3.1 Applying Static Aspects to Modelica

First, the requirements for a static aspect language for Modelica is given to motivate the language design decisions. As an object-oriented language, Modelica benefits from component-based and inheritance principles fitting well with real structures from the problem domain thus being seamlessly transferable to the solution domain. The equation-based specifications of components' inertia allow for declarative and encapsulated behavioral descriptions detached from the system context. As a consequence, Modelica breaks a system down into smaller units of structure and behavior which is supported by according language constructs.

When aiming at a language for investigating static aspects of Modelica models, one starts with these units stating the *primitives* (atoms, terminals, etc.) as well-known starting points. A distinction between *unary* and *binary* primitives of Modelica models can be made, where the first category comprises high level units such as packages, classes, connectors, etc., therefore stating single, isolated entities. Instead, binary primitives are relations that group two kinds of units by a certain (semantic) criterion, e.g. all components of a model or all unknowns of an equation. On this basis, the aspect language must allow for expressing and modularization of combined, unit-spanning primitives by appropriate operators. Hereby, arbitrary complex relations (the static aspects) can be iteratively derived from simpler ones, always starting with the aforementioned primitives. Remind again the balanced model properties of Modelica 3: To ensure these properties, the inner structure of models and their connectors are to be considered on each hierarchy level.

To demonstrate the application of the syntactical constructs of the static aspect language introduced in the following, a simple Modelica model is provided as a running example consisting of a simple `Pin` for an electrical circuit as

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

For electrical components with two pins, a corresponding port "interface" model given as

```
partial model OnePort
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

defines the fundamental relations between quantities of electrical circuits. Being `partial`, the port model is to be derived by a concrete electrical component, e.g. an ideal resistor such as

```
model Resistor
  extends OnePort;
  parameter Real R(unit="Ohm");
equation
  R*i = v;
end Resistor;
```

characterized by a resistance parameter `R` and behavior according to Ohm's law. A simple `Circuit` model

```
model Circuit
  Resistor R1(R=100), R2(R=200);
equation
  R1.n = R2.p;
end Circuit;
```

consists of two resistors connected in series.

3.2 Syntax

The following language is specifically tailored to extend Modelica by static aspects and consists of two sublanguages for modularizing aspect definitions:

- A language for specifying a set of *join points*, hence static elements of interest within the model under consideration. The set of join points to be obtained is defined by a *point cut* expression that can be composed out of Modelica primitives (fundamental types of language units) and appropriate operators for combination.
- An action language for defining *advices*, therefore stating what to be done with the join points previously calculated as result of the point cut. The actual content of an advice might vary from the simple output of an error message in case of using the language for checking design rules, to arbitrary complex modifications of the join points. The latter entails static aspect weaving capabilities affecting the inspected model, e.g. by refactorings.

Accordingly, the language allows for specifying a series of rules of the form:

```
<Point cut> => <Advice>;
```

constituting *point cut – advice*-pairs with well-defined syntactical structure as will be given in this Section, thereby focusing on the first part.

Point cuts describe specific classes of elements in Modelica models referring to the language's basic constructs such as class definitions (types), components (members of classes), and equations. According to static aspects, point cuts constitute definite points within the model code to be considered in the aspect advice. The *join points* that match to a point cut are those complying with the predicates the point cut is composed of by combinations of logical and quantified expressions. Through this declarative approach of describing "requirements" for model elements under investigation, these point cut expressions are decoupled from a specific model and allow for a quantified model inspection that can be applied to arbitrary Modelica models without knowing inner details.

Due to the obvious relation of the concept to the logic paradigm, a point cut expression can be construed as a set of predicate definitions to be evaluated on a Modelica source of interest (set of model definitions). The therein contained set of model-specific join points is adjusted stepwise by iteratively processing the point cut subterms, and finally resulting in a set of join points fulfilling the overall point cut claims. The complete syntax for the point cut language is depicted in Figure 1. The syntactical structure is inspired by the PDL in [16], but modified in some parts, especially concerning parameterized point cuts which will be introduced in detail below. Meaning and application of the different constituents will be described in the following Sections.

$$
\begin{array}{rcl}
p & ::= & u \\
 & | & b(p) \\
 & | & p \textbf{ and } p \\
 & | & p \textbf{ or } p \\
 & | & \textbf{not } p \\
 & | & p \textbf{ equals } p \\
 & | & p \textbf{ less } p \\
 & | & p \textbf{ subset } p \\
 & | & \textbf{exists } b : p \\
 & | & \textbf{forall } b : p \\
 & | & \textit{relop}\ \ n\ b \\
 & | & [v := p]\textit{relop}\ b\ b \\
u & ::= & \textit{id} \\
 & | & '\textit{pattern}' \\
b & ::= & \textit{id} \\
 & | & b+ \\
 & | & b+n \\
 & | & p \textbf{ product } p \\
 & | & p \textbf{ product-d } p \\
p & : & \textit{Point Cut} \\
u & : & \textit{Unary Point Cut} \\
b & : & \textit{Binary Relation} \\
id & : & \textit{Identifier} \\
n & : & \textit{Natural Number} \\
relop & : & \textit{Relational Operator} \\
pattern & : & \textit{Name Pattern}
\end{array}
$$

Figure 1. Syntax of the Point Cut Language

3.2.1 Primitives

Starting the inspection of a Modelica input source by considering the set of all join points present, the point cut language provides some fundamentals *primitives* for a first limitation of the join point set. These predefined primitive terms constitute subsets of the overall join point set to those of a certain kind and/or within a certain context with respect to elementary structuring constructs of Modelica.

As mentioned in [16], primitives might either be classified as *unary* or *binary* which depends on the number of join points to be considered for a match. Binary primitives express some property of join points. The choice of appropriate unary primitives focuses on major entities in which models are organized: packages and class types. Such individuals represent first class members, hence key paradigm concepts of Modelica. Tables 1 and 2 list sets of unary primitives which are simply accessed by their names. The most general primitive `class` includes all specialized class type definitions, namely `model`, `connector`, `package`, `block`, `type`, `record` and `function` [2]. All of whom are Modelica primitives on their part, there-

Primitive	Matches
`class`	all class types defined in the source
`model`	model types defined in the source
`connector`	connector types defined in the source
`package`	package types defined in the source
`block`	block types defined in the source
`type`	data types defined in the source
`record`	record types defined in the source
`function`	function types defined in the source

Table 1. Unary Modelica Primitives for basic Class Types

fore partitioning the set of class types into disjoint subsets. Applied to the `Circuit` example, `class` matches to `Pin`, `OnePort`, `Resistor`, and `Circuit`, whereas `connector` only selects the `Pin`. The (sub-) set of par-

Primitive	Matches
`partialType`	partial types defined in the source
`finalType`	final types defined in the source
`localType`	local types defined within a type

Table 2. Unary Modelica Primitives for specialized Types

tial class type definitions can be obtained by the primitive `partialType`, and the final types correspondingly. Hence, `partialType` matches to `OnePort`, whereas the predicate `finalType` is matched nowhere in the sample model. The `localType` primitive matches local class instances nested within a model, e.g. consider an additional local model declaration

```
replaceable model Res = Resistor;
```

within `Circuit` for parameterized typing of circuit elements. Primitive `localType` matches to the type of `Res` which actually refers to the global type `Resistor` in this example.

Binary primitives match pairs of join points, therefore expressing binary relations. They can be used for incrementally deriving interrelations, therefore inspecting further properties of model elements, e.g. relating a member variable or nested component and its surrounding class type. As unary Modelica primitives were defined on the high-level type structure, now binary primitives allow for a detailed inspection of the internal properties of a class, namely the parts for structure (variables/component members, inheritance, accessibility etc.) and behavior (equations). Again, binary primitives are given by a name followed by a parameter `p` stating the kind of join point, it is related to. Tables 3 to 7 contain structural, and Table 8 behavioral unary primitives for Modelica models. The primitives in Table 3 allow for structural insights of a given type `p` by accessing its members. These might either be `primitiveMember` or components, thus class-typed variables. For the circuit example, `primitiveMember(model)` results in the resistance variable `R` from model `Resistor`[1]. Further partitioning of members is done by their dimension (vectors,

[1] Note: The variables v and i from Pin and OnePort are typed by an according (non-primitive) Type declaration, thus stating component members.

Primitive	Matches
`member(p)`	all members of a type that matches p
`primitiveMember(p)`	primitive members of a type that matches p
`componentMember(p)`	components of a type that matches p
`vectorMember(p)`	vector members of a type that matches p
`matrixMember(p)`	matrix members of a type that matches p
`publicMember(p)`	public members of a type that matches p
`protectedMember(p)`	protected members of a type that matches p
`replMember(p)`	polymorphic members of a type that matches p
`replType(p)`	local class member of a type that matches p
`constrType(p)`	upper bound type of a replaceable type that matches p

Table 3. Binary Modelica Primitives for Type Members

matrices), and visibility. Note that members not explicitly stated as public or protected in a model are assumed to be public, e.g. `public(connector)` results in `v,i`. A Modelica specific concept is that of *replaceable* members, explicit polymorphic members, and replaceable local types for type parameterization. The `replMember(model)` primitive for example would match components such as:

```
replaceable OnePort op;
```

within `Circuit`. This component can for instance be replaced by `Resistor` via a modifier on instantiation. For `replType` consider again the replaceable resistor model example and the aforementioned `localType` primitive: In contrast, `replType(model)` matches the member variable `Res` itself instead of its referenced type which *is* actually the type of components within `Resistor` typed as `Res`. Finally, `constrType(model)` matches types constraining replaceable types, i.e. for

```
replaceable model Res
          = Resistor extends OnePort;
```

the point cut expression

```
constrType(replType(model))
```

results in `OnePort`. The `primModifier(p)` primitive in Table 4 matches modifier values for parameter members `p` applied when its surrounding class is instantiated, e.g.

```
primModifier(primitiveMember(model))
```

matches `100,200` as `Resistor` is instantiated twice within `Circuit` with corresponding modifier values for the parameter `R`. The primitive `compModifier(p)` matches types used for redeclaration of replaceable components `p` when its surrounding class type is being instantiated, e.g. in

Primitive	Matches
`primModifier(p)`	modifier of a primitive parameter member that matches p
`compModifier(p)`	modifier of a replaceable component that matches p
`typeModifier(p)`	redeclaration type of a replaceable type

Table 4. Binary Modelica Primitives for Modification and Redeclaration

```
Circuit circuit(
        redeclare OnePort Resistor);
```

the modifier `Resistor` for the replaceable `OnePort` matches this predicate. In case of redeclaration of local class types `p` such as

```
Circuit circuit(
        redeclare Model Res =
        Capacitor);
```

the new type parameter assigned to `Res` can be obtained by

```
typeModifier(replType(model))
```

thus resulting in `Capacitor`. Table 5 lists primitives

Primitive	Matches
`derivedType(p)`	types that are derived from a type that matches p
`baseType(p)`	types that are derived by a type that matches p
`subType(p)`	types that are subtypes of a type that matches p
`varDerivedType(p)`	types that are variably derived from a local type that matches p

Table 5. Binary Modelica Primitives for Inheritance Hierarchies

for obtaining inheritance relations between types. The result of `derivedType(partial)` is `Resistor` (direct subclass relation), and `baseType(model)` inversely matches `OnePort` as the direct super class of `Resistor`. In Modelica, a distinction is made between explicit subclasses and implicit subtypes within the type hierarchy of a system model. The primitive `subType` matches all types with public interfaces being compatible with that of type `p`. Note that this primitive is quite powerful as it is not directly extractable from a model source, but rather requires additional computational efforts. A further advanced construct of Modelica is variable inheritance. Consider the modified `OnePort` model:

```
model OnePort
  replaceable model Res = Resistor;
  ...
protected
  extends Res;
```

```
  ...
end OnePort;
```

Here, the local class type `Res` can be used to redeclare the base class of `OnePort` which is initially stated as `Resistor`. Thus, the result of the expression

```
varDerivedType(localType)
```

where `localType` matches `Resistor`, is `OnePort`. The primitives depicted in Table 6 allow for inspection of

Primitive	Matches
`flow(p)`	flow members of a type that matches p
`input(p)`	input members of a type that matches p
`output(p)`	output members of a type that matches p
`constant(p)`	constant members of a type that matches p
`parameter(p)`	parameter members of a type that matches p
`inner(p)`	inner members of a type that matches p
`outer(p)`	outer members of a type that matches p
`final(p)`	final members of a type that matches p
`discrete(p)`	discrete members of a type that matches p

Table 6. Binary Modelica Primitives for Member Properties

further member properties. Being mostly self-explanatory, a detailed description shall be omitted at this point.

Primitive	Matches
`startValue(p)`	start value of a member that matches p
`fixedStartValue(p)`	fixed value of a member that matches p

Table 7. Binary Modelica Primitives for Member Initialization

Modelica allows for the propagation of start values and initialization values for primitive member variables. The primitives in Table 7 can be used to obtain such values. The primitives listed in Table 8 can be used to delve into model bodies and explore the behavioral specifications in equations[2]. As `equation(p)` matches all kinds of equations defined for type `p`, e.g. `R*i = v;` for `Resistor`, further primitives for partitioning the set of equations are given, i.e. initial equations, equations for connecting two connector components, equations containing `if` or `when` clauses (potentially causing events), and `for` loops. After having selected a certain equation, the `unknown` primitive can be used to get the set of variables appearing in an equation, hence

[2] Note: Algorithm parts can also constitute model behavior, but are no further considered at this point.

Primitive	Matches
`equation(p)`	equations defined in a type that matches p
`initEquation(p)`	initial equations in a type that matches p
`connectEquation(p)`	connect equations in a type that matches p
`ifEquation(p)`	equations containing `if` in a type that matches p
`whenEquation(p)`	eq. containing `when` in a type that matches p
`forEquation(p)`	equations containing `for` in a type that matches p
`unknown(p)`	unknown variables in an equation p
`derivated(p)`	unknown variables derivated in an equation p

Table 8. Binary Modelica Primitives for Model Behavior

```
unknown(equation(p))
```

for `p` matching the `Resistor` results in `R,i,v`. Variables being derivated within an equation match the primitive `derivated`. On this basis, even more arbitrary complex primitives concerning equation details might be useful, e.g. inspecting operators, but shall be omitted at this point.

3.3 Operators

For the definition of extensive aspects, thus concerns that crosscut the entities of models, operators for complex term construction are provided for correlating Modelica primitives of initially separated model units. Being closed under the set of join points of the overall model, such operators allow for iterative combinations of point cut expressions permitting rules of any complexity. Some criteria to take into account when choosing appropriate operators:

- The resulting language's expressiveness must be sufficient for capturing a wide range of possibly occurring requirements.
- The operators must allow for an "atomic" conversion and efficient evaluation.
- The operators usage must be concise and intuitive.

As the operators are applied to *sets* of join points, they are mainly of set oriented nature inspired by logic programming and query languages. For operator precedences, the usage of appropriate parentheses is recommended as usual.

3.3.1 Logical Operators

For simple interrelations of join point sets, logical connectors are provided. For instance, classes, that are both subclasses derived from other classes *and* subtype of another type, can be searched via

```
derivedType(class) and subType(class)
```

As another example, the expression

```
input(class) or output(class)
```

matches all directed member variables. As one of the most powerful operators, the logical negation allows for expressions matching all join points *not* being contained in a given set of join points[3]. For instance

```
partial and not baseType(class)
```

matches all partial class types not being derived by any other class type. Moreover, the equality of two sets of join points can be stated, e.g.

```
class equals (model or connector)
```

requires systems only consisting of models and connectors. Two more operators for more intuitive descriptions of stepwise refinements for intermediate point cuts are provided:

```
model less partial
```

matches all model types *but* partial ones, whereas

```
partial subset model
```

matches *only those* model types that are also partial, which can actually also be expressed by the `and` operator.

3.3.2 Pattern

In order to examine join points that refer to named model elements, a pattern operator is provided. Using arbitrary pattern expressions, the set of join points can be reduced to the ones whose names match the given pattern. By

```
model and 'Resistor'
```

the resistor model can be obtained. Moreover, patterns can be used to check naming conventions, e.g. the demand *"Flow variables shall be named with a flow postfix"* can be expressed by

```
(flow(class) less '*_flow')
```

matching those flow members whose names do not have the required postfix. Besides wild cards `*` matching arbitrary sequences of symbols, further operators inspired e.g. by *regular expressions* can be used such as ranges `[a,b,c]` demanding one of the listed symbols. As there are primitives referring to unnamed join point types, e.g. equations, the pattern operator is only allowed at innermost position of expressions and no explicit comparison operator is provided.

3.3.3 Quantification

Point cut quantification can be used to check some condition on a *range* of values in a binary relation of join points. Therefore, point cut expressions can be constructed whose matching join points are based on properties of related join points. The expression

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

matches block types, whose primitive members are *all* output variables, i.e. sources. In this example, the property `output` (of a block) is postulated for all primitive members of that block (stated as a binary primitive relation). In contrast, the expression

```
exists primitiveMember : output(block)
```

matches blocks with *at least* one output variable.

3.3.4 Cardinality

Operators dealing with the cardinality of join point sets allow for evaluation of model metrics. Style conventions for structuring Modelica libraries such as *"The number of models defined in an own package must be at least 5"* can be expressed by

```
package and (> 5 componentMember)
```

resulting in "malformed" packages containing less than 5 type declarations. A cardinality expression can be parameterized by an optional point cut `[v:=p]`, hence a set of join points. Note that this concept will be generalized in Section 3.5 for application to all point cut expressions. In this way, comparisons of cardinalities concerning further properties between join points can be enforced. The complex balanced model demand *"The number of flow variables in a connector must be identical to the number of non-causal non-flow variables"* [17] can be stated as

```
[v:=connector](= flow(v)
(primitiveMember(v)
less (flow(v) or input(v) or output(v)
or parameter(v) or constant(v)))
```

constituting an iterative "foreach" loop over the set of connectors.

3.3.5 Composition

New binary relations b can be created by composing two point cuts by building the Cartesian product, hence combining all join points matching these point cuts. For instance, the expression

```
[v:=class](derivedType(v) product
derivedType(v))
```

relates types having the same (direct) super class. Here, again the parameterization syntax is used to relate both subclasses to the same super class. Note that the `product` operator also relates identical join points, thus creating reflexive relations. For avoiding such self-references, the `product-d` (disjoint) operator can be used. As a further example, the expression

```
[v:=connectEquation(model)]
(unknown(v) product-d unknown(v))
```

relates components being connected within models.

3.3.6 Transitive Closures

The deduction of (anti-symmetric) transitive closures of a binary relations can be obtained by

```
derivedType+
```

for instance, resulting in the subclass relation, hence relating two classes (indirectly) associated via arbitrary chains of derivations. The "bounded" transitive closure operator creates chains limited to at most n links, e.g.

```
derivedType+4
```

[3] Depending on the type system applied, this complemented set could be further limited to only those join points of the same type as the given ones.

can be used to check whether inheritance hierarchies deeper than 4 are present in the model by comparing the result to that of the unbounded case. The closure operator can also be used for examining connector traces of Modelica models.

3.4 Semantics

Partly taken from [16], the point cut evaluation is reduced to element-wise reasoning of join point sets considering the stipulated conditions. The evaluation of a point cut expression $p \in P$ from a set of rules P with respect to a Modelica model specification $\mathcal{M}$ (a collection of conjugated class type definitions) is stated as:

$$p : J_{\mathcal{M}} \rightarrow \mathcal{P}(J_{\mathcal{M}})$$

resulting in a (sub-) set of join points of the model under consideration, where $J_{\mathcal{M}}$ is assumed to be

$$J_{\mathcal{M}} = \{\text{sets of all types of join points present in } \mathcal{M}\}$$

The evaluation of a rule expression precedes from inwards to outwards, therefore calculating temporary join point sets as intermediate stages that are gradually refined toward the overall result set. Being composites of unary and binary expressions, point cuts are calculated through sequences of mappings

$$U : \textit{Unary Point Cut} \rightarrow \mathcal{P}(J_{\mathcal{M}})$$

for unary primitives u, and

$$B : \textit{Binary Relation} \rightarrow \mathcal{P}(J_{\mathcal{M}} \times J_{\mathcal{M}})$$

for binary primitives $b(p)$, respectively. The evaluation of u in a point cut expression can be simply stated as $P[\![u]\!] = U[\![u]\!]$, where u might be either a unary primitive denoted by *id* and will therefore be replaced by those join points $j \in J_{\mathcal{M}}$ matching *id*, or it is some kind of regular expression, thus

$$U[\![' pattern']\!] = \{j \mid j \textit{ matches 'pattern'}\}$$

is to be applied. For the evaluation of binary primitives $b(p)$, the given parameter p is to be taken into account:

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

The relation set b between the parameter join point p and the binary primitive is constructed by trying out all possible combinations and keeping those fulfilling b. The result, again is a set of single join points "fitting" to the parameter p. Unary and binary primitives can be nested (composed) at will, e.g. $b(b(p))$. The operators for logical combinations of join point sets can be reduced to according set operators:

$$\begin{aligned} P[\![p_1 \textbf{ and } p_2]\!] &= P[\![p_1]\!] \cap P[\![p_2]\!] \\ P[\![p_1 \textbf{ or } p_2]\!] &= P[\![p_1]\!] \cup P[\![p_2]\!] \\ P[\![p_1 \textbf{ less } p_2]\!] &= P[\![p_1]\!] \setminus P[\![p_2]\!] \\ P[\![\textbf{not } p_1]\!] &= \{j \mid j \notin P[\![p_1]\!]\} \end{aligned}$$

The operators for comparing sets of join points (equality and subsets) are evaluated as follows:

$$\begin{aligned} P[\![p_1 \textbf{ subset } p_2]\!] &= \{j \mid \forall j \in p_1 : j \in p_2\} \\ P[\![p_1 \textbf{ equals } p_2]\!] &= (p_1 \textbf{ subset } p_2) \textbf{ and } (p_2 \textbf{ subset } p_1) \end{aligned}$$

Hence, `subset` results in the join point p_1, iff $p_1 \subseteq p_2$ and in the empty set, otherwise. The equality of two join point sets can then be ensured via `equals` by checking the result set not being empty. For the quantification operators, the semantics are given as

$$\begin{aligned} P[\![\textbf{forall } b : p]\!] &= \{j_2 \mid \forall (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!]\} \\ P[\![\textbf{exists } b : p]\!] &= \{j_2 \mid \exists (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!]\} \end{aligned}$$

Binary relations constitute relationships between two join points and can be obtained by according primitives *id*:

$$B[\![id]\!] = \textit{<primitive>}$$

Further relations can be constructed as Cartesian products:

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

For the `product-d` operator, the requirement $j_1 \neq j_2$ must be satisfied. The semantics for transitive closure is defined to be

$$P[\![b+]\!] = \{(j_1, j_k) \mid \exists (j_1, j_2), \ldots, (j_{k-1}, j_k) \in B[\![b]\!]\}$$

where $k \leq n$ must hold in the bounded case for $P[\![p + n]\!]$.

3.5 Variable Binding and Type System

As already used for the cardinality comparison operator, for further examinations of elements within other subterms, variable bindings are introduced. They allow for binding of join points to variables which can be used as parameters in subsequent terms. This concept shall now be enhanced to all kinds of point cut expressions. Generally, a point cut expression can be parameterized by an arbitrary set of point cut variables ϕ that are then visible within the point cut:

$$[\phi]p : \textit{Point cut}_{\phi}, \text{ where } \phi = \{v_1 := p_1, \ldots, v_n := p_n\}$$

is a set of n point cut variables v_i, whose content is again defined by point cut expressions p_i. In the modified semantics, these parameters are passed to all subterms of p, e.g.

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

These enhanced evaluation semantics constitutes a (nested) "for-each" loop over the set of join point combinations depicted by the parameter point cut(s) to be likewise adopted to all point cut subterms.

As proposed in [16], the adoption of *types* for point cuts allows for sound expressions with respect to the types expected for the matching join points. Such types reflect the basic kinds of Modelica constructs the join points refer to, namely types, members (primitive, components), scalar values, and equations. The integration of a type system for point cuts into the aforementioned semantics allows for

exact determination of the join point types in the result set. Therefore, the matching join points are restricted to those, that are in the static type calculated for the point cut. Due to lack of space, a formalization of this approach, especially considering types of parameter variables for point cuts, is deferred to future work.

3.6 Advices

The *advice* part of a rule shall only be discussed informally at this point. Generally, it is considered to be "executable" and it is applied for each join point matching the point cut part of a static aspect. For the simplest case, e.g. rule checking, a report string can be put out as an error description:

```
<point cut> =>
"Error: violated naming convention";
```

For providing a more expressive report, access to the resulting join point set should be made possible, e.g. by the following syntax for iterating the result set:

```
<point cut> =>
"Error: violated naming convention in "
+ ResultSet.nextItem().getName();
```

returning the name of the join point within the result set currently iterated. Note that the join points within `ResultSet` must be of a type that refers to a named element in Modelica. As a next step, arbitrary code as well as access to additional variables defined in the point cut could be permitted, e.g. for obtaining the corresponding AST Nodes for manipulation within a compiler environment.

4. Implementation Framework

A sample implementation framework of the static aspect language that is currently under construction is proposed in the following. It is based on the principles of *logic meta programming* [23]. The application of two languages is proposed for processing static aspects on given Modelica models: (1) A *User* language for specifying static aspects, such as the previously defined point cut language which serves as an inspection API for Modelica models, and (2) an efficient *Implementation* language for processing of the aspects, therefore stating a point cut evaluation engine. Due to the similarities of aspect principles and the logic paradigm [23], the first order logic programming language Prolog [9] can serve as an evaluation engine. Prolog allows for seamless conceptual representation and efficient evaluation of aspect queries on Modelica models in terms of logical facts and rules. A structural overview of the resulting implementation framework architecture is depicted in Figure 2. The front end for user input parsing and transformation consists of two interfaces:

1. A Modelica model input interface realized as a conventional ANTLR [11, 18] Modelica Parser that constructs the Abstract Syntax Tree (AST) representation of the model under consideration and extracts primitives in terms of Prolog facts. A similar approach is taken in MetaModelica [20].

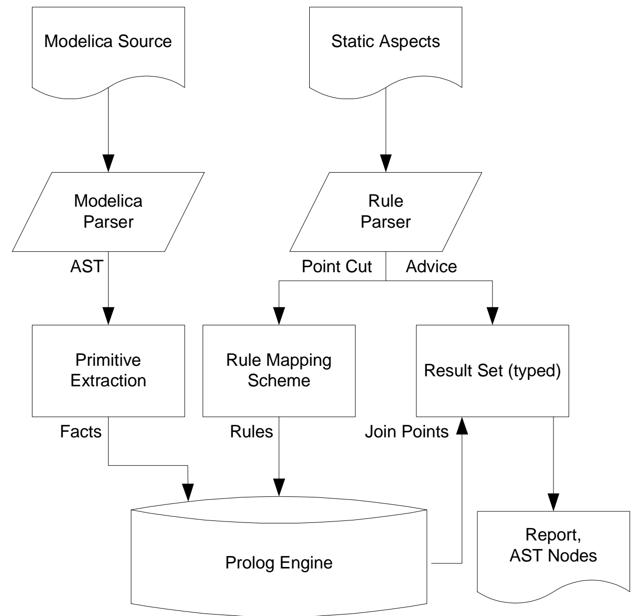

Figure 2. Architecture of the Static Aspects Framework for Modelica Models

2. An aspect rule input interface accepting syntactically well-formed static aspects expressions according to the aforementioned point cut language grammar. The point cut expression parts are then transformed to suitable Prolog rules according to a mapping scheme for iteratively composing primitives and operators.

The middle end forms the aspect processing engine in terms of a Prolog interpreter implemented on top of the Modelica source code parser. Applying the Prolog rules generated for point cut expressions to the fact basis containing the model primitives, the engine integrates both, the input models and the related aspect rules for join point result set processing. The back end interface conducts advice processing with respect to the resulting join point set, i.e.

- Error reports for simple design rule evaluation,
- References to the originating AST nodes of the resulting join points, thus allowing for arbitrary post-processing of complex aspect advices detached from the framework, e.g. as described in [8].

A simple example for mapping model structures to corresponding Prolog rules shall be given. Consider the aforementioned `Resistor` model inheriting from the partial `OnePort` model. First, the "existence" of both models can be expressed by appropriate Prolog facts:

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

Next, the interrelation of both models with respect to the implied inheritance hierarchy can be stated as:

```
derive(m2,m1).
```

According to these facts, implications can be derived by appropriate rules, e.g.:

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

for calculation of the transitive closure of the inheritance hierarchy.

The decoupling of the *User* and *Implementation* language aims at a high grade of extensibility and adaptability. Being Turing complete, the Prolog engine allows for integrating point cut language constructs of any complexity, and does not dictate the concrete representation of the aspect language implementation. The implementation can either be used as a "stand-alone" rule checker for systematic model inspection, or it can be integrated to an ambient Modelica compiler/development environment accomplishing static aspect *weaving* e.g. AST transformations.

5. Application

On the basis of the adaptable and scalable framework implementation and the flexibility of the proposed static aspect language for Modelica, various possible areas of application are conceivable:

1. Rule checking "by negation": Describing point cuts matching join points that are not desired to appear in the models as proposed in [16]. In case of non empty result sets, the rule is violated by the join points calculated and corresponding error reports can be generated in the advice part. By expressing new restrictions of the Modelica 3 specification as static aspects, the compatibility of legacy code such as libraries can be examined automatically.
2. Model inspection: Searching for model elements or patterns matching criteria of interest, either "off-line" (e.g. metrics calculations), or "on-line" as a model inspection tool within a Modelica IDE. Further applications in the realm of the object oriented paradigm might be that of *concepts* [13], thus checking whether a given set of types are applicable as parameters for a generic construct (upper bound resolution for constraining types).
3. Join point manipulations within the advice part, e.g. renaming of certain elements with respect to naming conventions or model maintenance.
4. Arbitrary model restructuring by join points referencing nodes of the AST. Therefore, static aspect weaving can be done by graph transformation, e.g. context aware refactorings.

6. Conclusion

The formal syntax and semantics definition and sample implementation framework of a static aspect language for Modelica was presented. The language design aims at sufficient expressiveness and extensibility, but yet still provides intuitive usage. Language extensions for variable bindings of (typed) join points were mentioned for enhanced rule precision.

The framework proposed can serve as a foundation for a wide range of applications, e.g. simple rule checking up to source code manipulations. An integration into existing environments is aimed at, either as a basis for a point cut evaluation engine for a Modelica AOP Compiler, or as a programmer's on-line assistance tool for code inspection queries, e.g. providing an interactive search engine with Eclipse IDE integration.

In future work, after having finished the implementation, the application of the language in various case studies can indicate, whether the language boundaries defined up to this point are sufficient. For this purpose, the formulation of balanced model requirements of Modelica version 3 in terms of static aspects is assumed to be a convenient case study. Hereby, the performance of the implementation can be investigated concerning the number and complexity of rules and models under investigation. Optimizations can lead to increased efficiency, e.g. by dynamical and cached AST access on demand during rule evaluation. Moreover, a detailed survey of the advice part is aimed at, especially in view of conflicts analysis between different aspects. On this basis, studies of applying dynamic aspect approaches to Modelica can be promising, although Modelica-like language are not *procedural* ones as demanded e.g. in [15].

Acknowledgments

The authors' thank goes to Dr. Michaela Huhn and TLK Thermo for helpful discussions supporting this paper.

References

[1] Johan Akesson. *Languages and Tools for Optimization of Large-Scale Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, November 2007.

[2] The Modelica Association. *The Modelica Language Specification 3.0.* http://www.modelica.org, September 2007. http://www.modelica.org.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuiness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[4] Alexander Borgida. Description Logics in Data Management. In *IEEE Transactions on Knowledge and Data Engineering*, volume 7, pages 671–682, 1995.

[5] David Broman, Kaj Nyström, and Peter Fritzson. Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 151–160, Portland, Oregon, USA, 2006. ACM Press.

[6] Peter Bunus and Peter Fritzson. Automated Static Analysis of Equation-Based Components. *SIMULATION*, 80(7-8):321–345, July-August 2004.

[7] Tal Cohen, Joseph Gil, and Italy Maman. JTL - The Java Tools Language. In *OOPSLA'06: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[8] Roger F. Crew. ASTLOG: A Language for Examinging Abstract Syntax Trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.

[9] P. Deransart, A. Ed-Dbali, and L. Ceravoni. *Prolog: The Standard*. Springer-Verlag, NewYork, 1996.

[10] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness, 2000.

[11] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In *IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1588–1595, October 2006.

[12] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. IEEE Press, 2004.

[13] J. Järvi, J. Willcock, and A. Lumsdaine. Associated Types and Constrain Propagation for Mainstream Object-Oriented Generics. In *Proceedings of the 20th OOPSLA*, pages 327–355. Springer Verlag, June 2001.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2027:327–355, 2001.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*. Springer-Verlag, June 1997.

[16] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A Static Aspect Language for Checking Design Rules. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 63–72, New York, NY, USA, 2007. ACM.

[17] Hans Olsson, Martin Otter, Sven Erik Mattsson, and Hilding Elmqvist. Balanced Models in Modelica 3.0 for Increased Model Quality. In Prof. Dr. B. Bachmann, editor, *Proceedings of the 6th International Modelica Conference*, volume 1, University of Applied Sciences Germany, Bielefeld, 2008. The Modelica Association.

[18] Terence Parr. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Programmers, 2007.

[19] Adrian Pop and Peter Fritzson. Debugging Natural Semantics Specifications. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 77–82, New York, NY, USA, 2005. ACM.

[20] Adrian Pop and Peter Fritzson. MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In *Joint Modular Languages Conference (JMLC2006)*, Jesus College, Oxford, England, September 2006.

[21] Michael Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[22] Michael Tiller. Parsing and Semantics Analysis of Modelica Code for Non-Simulation Applications. In Peter Fritzson, editor, *Proceedings of the 3rd International Modelica Conference*, volume 1, pages 411–418, Linköping, November 2003.

[23] Kris De Volder and Theo D'Hondt. Aspect-Oriented Logic Meta Programming. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 250–272, London, UK, 1999. Springer-Verlag.