Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

P. Fritzson, A. Pop, P. Aronsson
Linköping University, Sweden

Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica

pp. 519-525

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Program Committee

- Prof. Gerhard Schmitz, Hamburg University of Technology, Germany (Program chair).
- Prof. Bernhard Bachmann, University of Applied Sciences Bielefeld, Germany.
- Dr. Francesco Casella, Politecnico di Milano, Italy.
- Dr. Hilding Elmqvist, Dynasim AB, Sweden.
- Prof. Peter Fritzson, University of Linkping, Sweden
- Prof. Martin Otter, DLR, Germany
- Dr. Michael Tiller, Ford Motor Company, USA
- Dr. Hubertus Tummescheit, Scynamics HB, Sweden

Local Organization: Gerhard Schmitz, Katrin Prölß, Wilson Casas, Henning Knigge, Jens Vasel, Stefan Wischhusen, TuTech Innovation GmbH

Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica

Peter Fritzson, Adrian Pop, Peter Aronsson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{petfr,adrpo,petar}@ida.liu.se

Abstract

The need for integrating system modeling with tool capabilities is becoming increasingly pronounced. For example, a set of simulation experiments may give rise to new data that are used to systematically construct a series of new models, e.g. for further simulation and design optimization. Using models to construct other models is called meta-modeling or meta-programming.

In this paper we present extensions to the Modelica language for comprehensive meta-programming, involving transformations of abstract syntax tree representations of models and programs. The extensions have been implemented and used in several applications, are and currently being integrated into the OpenModelica environment.

1 Introduction

Meta-programming (meta-modeling) is writing programs (models) having other programs (so called object-programs) as inputs or results. A program can for instance take another program as input data, perform computations on the program by traversing its internal structure (the abstract syntax of the program) and return a modified program as output data.

Often, the object program language and the meta-programming language are the same, like for instance in LISP, in Mathematica, or in the Java reflection mechanism. This is also the approach we have taken for Modelica. Thus, a language needs some way of representing the object program as data.

A simple approach is to use text strings as program representation. However, this has the disadvantage that not even simple structural (syntactic) correctness can be guaranteed. Another problem is low performance. Thus, this approach is only suitable for simple and less demanding tasks.

Another solution is to encode the object program using structured data types of the meta-programming language. This basically means that data types for the abstract syntax are defined in the language itself. This approach has the benefit of ensuring correct syntax of object programs. It is used in for instance Java reflection where the class `java.lang.Class` is the data type for a Java class. The class has methods to query a Java class for its methods, members, interfaces, etc.

In a previous paper (Aronsson *et.al.*, 2003) we presented an approach of quoted Modelica code combined with built-in predefined Modelica types to handle certain syntax classes, like for instance `TypeName` for a Modelica type name or `VariableName` for a Modelica variable name. However, this does not give full flexibility and meta-programming power, since the abstract syntax tree representation cannot be fully manipulated in the meta-programming language itself. That work should be seen as a precursor and initial stage for the work presented in this paper.

2 Tree Data Structures

What are then the needs for data structures and operations for full meta-programming capabilities? One of the most common examples of programs that manipulate and produce other programs are compilers, which translate programs in some language into the same or another language.

The most common data type representation for programs in compilers are tree structures, and typical operations are transformations of such trees into trees during the translation process. Lists are a special case of tree data types, but are typically given special support in many symbolic programming languages..

Tree data types have two interesting properties:

- Union type – a tree data type is typically the union of a number of node types, each representing a tree node.
- Recursive type – the children of a tree node may a type which is the tree data type itself.

A small expression tree, of the expression 12+5*13, is depicted in Figure 1. Using the record constructors `PLUS`, `MUL`, `RCONST`, this tree can be constructed by the

expression `PLUS(RCONST(12), MUL( RCONST(5), RCONST(13)))`

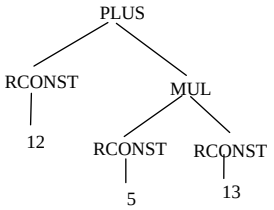

Figure 1. Abstract syntax tree of the expression 12+5*13.

Union types and recursive types are currently missing from the Modelica language, which so far has been a conscious decision in order to avoid heap-allocated objects.

However, with the increased relevance of meta-modeling, the time may now be ripe for a possible extension such as the introduction of the `uniontype` restricted class construct. The example below declares a small expression tree type `Exp` containing 6 different node types represented as ordinary Modelica record types.

```
uniontype Exp
  record RCONST Real x1; end INT;
  record PLUS  Exp x1; Exp x2; end PLUS;
  record SUB   Exp x1; Exp x2; end SUB;
  record MUL   Exp x1; Exp x2; end MUL;
  record DIV   Exp x1; Exp x2; end DIV;
  record NEG   Exp x1;         end NEG;
end Exp;
```

The `uniontype` restricted class construct currently has the following properties:

- Union types can be *recursive*, i.e., reference themselves. The is the case in the above `Exp` example, where `Exp` is referenced inside its member record types.
- Union types are currently restricted to contain only *record* types. This restriction may be removed in the future.
- Record declarations declared within a union type are automatically inherited *into the enclosing* scope of the union type declaration.
- A record type may only belong to *one* union type. This restriction may be removed in the future.

This is a preliminary union type design, which however is very close to (just different syntax) similar constructs in functional languages such as Haskell, Standard ML, OCaml, and RML.

3 Tree Transformation Operations

Regarding operations on trees, most languages supporting tree transformations provide a kind of pattern matching and transformation construct. Therefore we propose the introduction of match-expressions in the Modelica language. A trivial example of match-expression is presented below:

```
  String str;
  Real   x;
algorithm
  x :=
    match str
      case "one"   then 1;
      case "two"   then 2;
      case "three" then 3;
      else   0;
    end match;
```

The string variable `str` is matched against the constant patterns `"one"`, `"two"`, etc., returning the corresponding value from each branch in the match-expression. A default value can be returned from the optional else-branch if no other branch matches.

The general form of the proposed match-expression is as follows:

```
match <expr>  <opt-local-decl>
  case <pat-expr> <opt-local-decl>
    <opt-local-equations>
    then <value-expr>;
  case <pat-expr> <opt-local-decl>
    <opt-local-equations>
    then <value-expr>;
  ...
  else <opt-local-decl>
    <opt-local-equations>
   then <value-expr>;
end match;
```

The **`then`** keyword precedes the value to be returned in each branch.. The local declarations started by the **`local`** keyword, as well as the equations started by the **`equation`** keyword are optional.There should be at least one **`case`**`...`**`then`** branch, but the **`else`**-branch is optional.

The match-expression introduces several new concepts in Modelica:

- Pattern expressions, `<pat-expr>`, which may reference unbound local pattern variables declared within the match-expression.
- Optional local variable declarations, `<opt-local-decl>`. These variables are local and have a scope within the match-expression or within a specific branch of the match-expression if they are declared within such a branch.
- Optional local equations, `<opt-local-equations>`, which are solved locally within the match-expression, and where the unbound unknowns to be solved for have been declared in local variable declarations.

An example of a match-expression within the function `eval` shows its usage in a simple expression tree evaluator. The local variables `v1`,`v2`,`e1`,`e2` have scope throughout the whole match-expression. Pattern variables such as e1 and e2 are belong to pattern expressions that are matched against tree expressions. For example, `PLUS(e1,e2)` is matched against `PLUS(RCONST(12), MUL( RCONST(5), RCONST(13)))` depicted in Figure 1, thereby binding `e1` and `e2` to the children of the `PLUS` node, in this match `e1` to `RCONST(12)` and `e2` to `MUL( RCONST(5), RCONST(13))`.

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
  match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
    case RCONST(v1) then v1;

    case PLUS(e1,e2) equation
      v1 = eval(e1;  eval(e2) = v2;
      then v1+v2;

    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2;
      then v1-v2;

    case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;

    case DIV(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1/v2;

    case NEG(e1) equation
      v1 = eval(e1);
      then -v1;
   end match;
end eval;
```

Note that the match-expression just like other expressions can be used in three contexts: inside equations, inside algorithm sections, and inside functions.

As usual in Modelica the equations are not directional, e.g. the two equations `v1 = eval(e1)` and `eval(e1) = v1` are equivalent.

There are some design considerations behind the above match-expression construct that may need some motivation.

- Why do we have *local variable declarations* within the match-expression? The main reason is clear and understandable semantics. In all three usage contexts (equations, algorithm sections, functions) it should be easy to understand for the user and for the compiler which variables are unknowns (i.e., unbound local variables) in pattern expressions or in local equations.

  Other variables that are bound to values might have been declared in some class, or be protected variables in a function. Without the simple rule that local unknowns must be declared locally, it would be hard to discover the difference between variables that are unknowns and still can receive values, and other variables which already have values.

  Another reason for declaring the types of local variables is better documentation of the code – the modeler/programmer is relieved of the burden of doing manual type-inference to understand the code.
- Why *local equations* instead of assignment statements? The match-expression is an expression construct that can be used in the three contexts, including expressions in equations which are declarative. Having non-local assignments inside expressions would make the expressions nondeclarative.
- Why match-expressions and not match-statements? The match-expression is more important since it can be used in all three contexts, and therefore has been designed first. An analogous match-statement without local equations can be designed at a later stage.
- Why the keywords **`match`** ... **`case`** instead of **`switch`** ... **`case`** as in Java? The current choice of keywords is inspired by the languages Modelica, Java, and Mathematica, and is just a matter of taste – it is easy to change to other keywords. However, it is probably good style to indicate the increase power of the match-expression compared to the switch-statement by a different keyword.
- Why the **`then`** keyword before the returned value? We have experimented with various syntax designs, and the code becomes easier to read if there is a keyword before the returned value-expression, especially when it is preceded by local equations. The keyword cannot be **`return`** since that means return from a function. The **`then`** keyword is used in a similar way in Modelica **`if-then-else`** expressions. Note that most functional languages use the `in` keyword instead in this context, which is less intuitive. However, the `in` keyword has more of a set or array element membership meaning in Modelica.

Local equations in match-expressions have the following semantics:

- Only algebraic equations are allowed, no differential equations
- Only locally declared variables (local unknowns) declared by local declarations within the match-expression are solved for.

- Equations are solved one by one in the order they are declared. (This restriction may be removed in the future).
- If an equation or an expression in a case-branch fails, all local variables become unbound, and the next branch is tried. (There is some discussion whether the semantics of trying the next case-branch after a fail should be kept).

3.1 Example of Symbolic Differentiation

To make the following example of symbolic differentiation more realistic, we add a few expression nodes to the `Exp` data type, including a function call node `CALL` whose argument list has the type **`list`**`<Exp>`, see Section 4.1.

```
record IDENT String name; end IDENT;
record CALL  Exp id; list<Exp> args;
  end CALL;
record AND   Exp x1; Exp x2; end AND;
record OR    Exp x1; Exp x2; end OR;
record LESS  Exp x1; Exp x2; end LESS;
record GREATER Exp x1; Exp x2;
  end GREATER;
```

An example function `difft` performs symbolic differentiation of the expression `expr` with respect to the variable `time`, returning a differentiated expression. In the patterns, _ underscore is a reserved word that can be used as a placeholder instead of a pattern variable when the particular value in that place is not needed later as a variable value. The **`as`**-construct: `id` **`as`** `IDENT(_)` in the third of-branch is used to bind the additional identifier `id` to the relevant expression. Both tuples with syntax `(expr1,expr2,....)`, see Section 4.2, and lists are used in the example.

We can recognize the following well-known derivative rules represented in the match-expression code:

- The time-derivative of a constant (`RCONST()`) is zero.
- The time-derivative of the `time` variable is one.
- The time-derivative of a time dependent variable `id` is `der(id)`, but is zero if the variable is not time dependent, i.e., not in the list `tvars`/`timevars`.
- The time-derivative of the sum (`add(e1,e2)`) of two expressions is the sum of the expression derivatives.
- The time-derivative of `sin(x)` is `cos(x)*x'` if `x` is a function of time.
- etc...

We have excluded some operators in the `difft` example because of limitations of space in this paper.

```
function difft "Symbolic differentiation
    of expression with respect to time"
  input  Exp expr;
  input  list <IDENT> timevars;
  output Exp diffexpr;
algorithm
 diffexpr :=
  match (expr, timevars)
    local Exp e1prim,e2prim,tvars;
          Exp e1,e2,id;
// der of constant
    case(RCONST(_), _) then RCONST(0.0);
// der of time variable
    case(IDENT("time"), _) then
      RCONST(1.0);
// der of any variable id
    case difft(id as IDENT(_), tvars) then
      if list_member(id,tvars) then
        CALL(IDENT("der"),list(id))
      else
        RCONST(0.0);
// (e1+e2)' => e1'+e2'
    case (ADD(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then ADD(e1prim,e2prim);
// (e1-e2)' => e1'-e2'
    case (SUB(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then SUB(e1prim,e2prim);
// (e1*e2)' => e1'*e2 + e1*e2'
    case (MUL(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then PLUS(MUL(e1prim,e2),
              MUL(e1,e2prim));
// (e1/e2)' => (e1'*e2 - e1*e2')/e2*e2
    case (DIV(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then DIV(SUB(MUL(e1prim,e2),
                MUL(e1,e2prim)),
              MUL(e2,e2));
// (-e1)' => -e1'
    case (NEG(e1),tvars)  equation
      e1prim = difft(e1,tvars);
      then NEG(e1prim);
// sin(e1)' => cos(e1)*e1'
   case CALL(IDENT("sin"),list(e1)),tvars)
      equation e1prim = difft(e1,tvars);
     then MUL(CALL(IDENT("cos"),list(e1)),
              e1prim);
// (e1 and e2)' => e1'and e2'
    case (AND(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then AND(e1prim,e2prim);
// (e1 or e2)' => e1' or e2'
    case (OR(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then OR(e1prim,e2prim);
// (e1<e2)' => e1'<e2'
    case (LESS(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
      then LESS(e1prim,e2prim);
// (e1>e2)' => e1'>e2'
    case (GREATER(e1,e2),tvars) equation
      e1prim = difft(e1,tvars);
```

```
      e2prim = difft(e2,tvars);
      then GREATER(e1prim,e2prim);
// etc...
   end match;

 end difft;
```

4 Lists and Tuples

List and tuple data types are common in many languages used for meta-programming and symbolic programming.

4.1 Lists

The following operations allows creation of lists and addition of new elements in front of lists in a declarative way. Extracting elements is done through pattern-matching in match-expressions shown earlier.

- **list** – **list**(el1,el2,el3, ...) creates a list of elements of identical type. Examples: **list**()– the empty list, **list**(2,3,4) – a list of integers.
- **nil** – denotes an empty reference to a list or tree.
- cons – the call cons(element, lst) adds an element in front of the list lst and returns the resulting list. Also available as a new built-in operator :: (coloncolon), e.g. used as in: element::lst.

Types of lists and list variables can be specified as follows:

- **list** – **list**<type-expr> is also a list type constructor, e.g. :
  type RealList = **list**<Real>;
- Direct declaration of a variable rlist that denotes a list of real numbers:
  list<Real> rlist;

4.2 Tuples

Tuples can be viewed as instances of anonymous records. The syntax is a parenthesized list. The same syntax is used in extended Modelica presented here, and is in fact already present in standard Modelica as a receiver of values for functions returning multiple results.

- An example of a tuple literal:
  (a, b, "cc")
- A tuple with a single element has a comma in order to have different syntax compared to a parenthesized expression: (a,)
- A tuple can be seen as being returned from a function with multiple results in standard Modelica:
  (a,b,c) := foo(x, 2, 3, 5);
- Access of field values in tuples is achieved via dot-notation, tupvalue.fieldnr, analogous to recvalue.fieldname for ordinary record values. For example, accessing the second value in tup:
  tup.2

The main reason to introduce tuples is for convenience of notation. You can use them directly without explicit declaration. Tuples using this syntax are already present in the major functional programming languages.

A tuple will of course also have a type. When tuple variable types are needed, they can for example be declared using the following notation:

```
type VarBND  = record<Ident, Integer>;
```

or directly in a declaration of a variable bnd:

```
record<Ident, Integer>   bnd;
```

The tuple type used in the match-expression of the previous simple eval function is **record**<Exp,Exp>.

5 Positional Type Parameters

Class definitions in Modelica allow type parameters, declared as replaceable local types, e.g.:

```
class C2 = C(redeclare class
                ColoredClass = BlueClass);
```

Using a shorter angle-bracket syntax for positional type parameters similar to what is used in other object-oriented languages such as C++ or Java, this can be expressed as:

```
class C2 = C<BlueClass>;
```

We have used this syntax in several places throughout this paper, including a call to a polymorphic function in Section 7.

6 Expression Evaluator with Environments

The previous small expression evaluator presented in Section 3 could only handle constant expressions. The following example can handle expressions with variables. It demonstrates a different representation of expression trees, with BINARY nodes that are parameterized in terms of the operator, and thereby can handle several binary operators in a single of-branch in the match-expression. First we give the type declarations:

```
type Ident   = String;

uniontype Exp
  record RCONST Real x1; end RCONST;
  record IDENT  Ident x1; end IDENT;
  record BINARY Exp x1; BinOp op; Exp x2;
    end BINARY;
  record UNARY  UnOp x1; end UNARY;
  record ASSIGN Ident x1;  Exp x2;
```

```
    end ASSIGN;
end Exp;

uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

uniontype UnOp
 record NEG end NEG;
end UnOp;

uniontype Value
  record REALval Real x1;  end REALval;
end Value;
```

The following `eval` function can handle evaluation of expressions with variable references. It calls the `lookup` function for access of variable references, and `apply_binop` for evaluation of binary operators.

```
type Ident   = String;

function eval
         // Evaluation of expression exp
         // in an environment env
  input  Env   env_1;
  input  Exp   exp_1;
  output Value value_1;
algorithm
 value_1 :=
  match (env_1,exp_1)
    local Real v,v1,v2;
         String   id;
         Env      env;
         Exp      e1,e2;
         Boolean  v3;
         BinOp    relop;
 // Real constant
    case (_,REALval(v)) then REALval(v);

 // variable identifier id
    case (env,IDENT(id)) equation
       v = lookup(env,id);
      then REALval(v);

// If id not declared, give an error
// message and fail through error
    case (env,IDENT(id)) equation
      v = not lookup(env,id);
      print("Error - undef variable: ");
      print(id);  print("\n");
      then fail()

 // expr1 binop expr2
    case (env, BINARY(e1,binop,e2))
      equation
        eval(env,e1) = REALval(v1);
        eval(env,e2) = REALval(v2);
        v3 = apply_binop(env,binop,v1,v2);
      then REALval(v3);

  end match;
end eval;
```

7 Function Parameters

A common and rather useful language feature not yet present in standard Modelica is the ability to pass function parameters. For example, passing the `add1` function to a mapping function that applies it to each element:

```
arr2 := arr_map_int(add1, {2,3,5,8})
```

returns:

```
{2,4,6,9}
```

We propose the following style of declaring a function that accepts a function formal parameter, exemplified through an example. The only syntax extension is to allow the declaration of a function without body, here `Functype`, which allows us to declare the type signature of the function formal parameter `func`.

```
function arr_map_int
  "Map over an array of integers"
   function Functype
     input Integer x1; output Integer x2;
   end FuncType;
  input replaceable function func
                          extends FuncType;
  input  Integer[:] inarr;
  output Integer[size(inarr,1)] outarr;
algorithm
  for i in 1:size(inarr,1) loop
    outarr[i] := func(inarr[i]);
  end for;
end arr_map_int;
```

The next example is polymorphic since the array element type `Type_a` is not fixed. It is a replaceable type, which makes it possible to apply `arr_map` to arrays of any element type. For example, applied to an array of strings, with the `addA` function that adds `"A"` to the end of a string:

```
arr3 :=
  arr_map<String>(addA, {"foo","fie"})
```

returns:

```
{"fooA","fieA"}
```

The definition of the `arr_map` function:

```
function arr_map
"Map over an array of elements of Type_a"
    replaceable type Type_a;
    function Functype
      input Type_a x1; output Type_a x2;
   end Functype;
  input replaceable function func
                          extends FuncType;
  input  Type_a[:]  inarr;
  output Type_a[size(inarr,1)] outarr;
algorithm
  for i in 1:size(inarr,1) loop
    outarr[i] := func(inarr[i]);
  end for;
```

```
end arr_map;
```

The semantics of function parameters include the following:

- Functions can be passed as actual arguments at function calls.
- Type checking done on the function formal parameter type signature, not including the actual names of inputs and outputs to the passed function.

8 Exception Handling

The design of exception handling capabilities in Modelica is currently in a preliminary phase. The following constructs are being discussed:

- A **try...catch** statement or expression.
- A **raise(...)** call for raising exceptions.

The statement variant has approximately the following syntax:

```
try
  <statements>
  ...
catch <x> then
  <statements>
  ...
end try;
```

The syntax of the expression variant is as follows:

```
try
  <expression>
catch <x> then
     <expression>
end try;
```

This design is still very preliminary, several issues need to be determined, and no implementation has yet been produced.

9 Conclusions

It has been demonstrated how Modelica can be extended with data structures and operations that are typically needed for comprehensive meta-programming and symbolic transformations. The extensions are declarative and preserve the declarative and equation-based style of Modelica. Recursive data types, lists, and tree pattern matching in match-expressions with local equations can be naturally integrated into the current Modelica 2.1 language. A implementation of most of this functionality has been tested on a number of examples, including those in this paper, and is currently being integrated into the OpenModelica compiler.

We believe that the combination of the modeling power and numeric capabilities of the current Modelica language, combined with symbolic transformation capabilities of the new extensions, will make Modelica into a very powerful meta-modeling and meta-programming language for the future.

10 Acknowledgements

This work has been supported by Swedish Foundation for Strategic Research (SSF), in the RISE project, and by Vinnova in the SWEBPROD project.

References

[1] Peter Aronsson, Peter Fritzson, Levon Saldamli, and Peter Bunus. Incremental declaration handling in Open Source Modelica. In SIMS - 43rd Conference on Simulation and Modeling on September 26-27, Oulu, Finland, 2002.

[2] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In proceedings of the 3rd International Modelica Conference, Linköping, Sweden, Nov 2003.

[3] Peter Fritzson, et al. The Open Source Modelica Project. In Proceedings of The 2nd International Modelica Conference, 18-19 March, 2002. Munich, Germany http://www.ida.liu.se/ ~pelab/modelica/ OpenModelica.html.

[4] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica*. 940 pp. Wiley-IEEE Press, 2004.

[5] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[6] The Modelica Association. The Modelica Language Specification Version 2.1, June 2003. http://www.modelica.org.

[7] Mikael Pettersson. Compiling Natural Semantics. PhD thesis, Linköping Studies in Science and Technology, 1995.

[8] Peter van Roy and Seif Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

[9] Tim Sheard. Accomplishments and Research Challenges in Meta-Programming. Lecture Notes in Computer Science, 2196:2–.., 2001