Meta Programming and Function Overloading in OpenModelica

Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus and Kaj Nyström
{petar,petfr,levsa,petbu,kajny}@ida.liu.se
Programming Environments Laboratory (PELAB)
Department of Computer and Information Science Linköping University, Sweden

Abstract

The OpenModelica framework is an Open Source effort for building a complete compiler for Modelica started at the programming environments laboratory at Linköping university. It is written in a language called RML [10], Relational Meta Language, based on natural semantics. Natural semantics is a popular formalism for describing the semantics (i.e. the *meaning* of language constructs) for compilers. By using the RML language this formalism is combined with efficient compilation into optimized C code.

The OpenModelica framework is used to experiment with new language features and language design for the ongoing development of the Modelica language. The design of the Modelica language is performed in the Modelica Design Group (by the Modelica Association) - an open group of Modelica users, resarchers, vendors, etc., where the the Modelica language is evolved through intensive discussions in threedays workshops, three or four times a year.

Recently, support for Meta-programming and function overloading (including an external interface to LAPACK) have been implemented in the OpenModelica compiler. This paper present the design and implementation of these language constructs in the OpenModelica framework and illustrates how to utilize this framework for research in e.g. language design, meta-programming and modeling and simulation methodology.

1 Introduction

The OpenModelica[6, 9] environment consist of a compiler that translates Modelica [3, 5] code into flat Modelica, which basically is the set of equations, algorithms and variables needed to simulate the compiled Modelica model. The environment also includes a shell, i.e. an interactive command and expression interpreter, similar to a Matlab prompt, where models can be entered, computations can be performed and functions can be called. In this environment it is also possible to execute Modelica scripts, i.e. Modelica functions or expressions executed interactively or a set of algorithm statements defined in a text file.

An example of a session in the OpenModelica shell is given below:

```
> ./mosh.exe
  Open Modelica 1.0
  Copyright 2003, PELAB,
  Linkoping University
  >>> loadModel(Modelica)
  true
  >>> model A=Modelica.Electrical
  .Analog.Interfaces.OnePort;
  Ok
  >>> translateModel(A)
  record
    flatClass = "fclass A
       Real v;
       Real i;
       Real p.v;
       Real p.i;
       Real n.v;
       Real n.i;
         equation
       v = p.v - n.v;
       0.0 = p.i + n.i;
       i = p.i;
    end A;",
    exefile = ""
  end record
  >>>
```

The OpenModelica compiler has been developed at the programming environments laboratory (PELAB) at the department of Computer and Information science at Linköping University. It is used to conduct research on Modelica and tools for modeling and simulation. Current research activities at PELAB involve automatic paralleization [1], support for Partial Differential equations in Modelica [11] and debugging techniques for Modelica [2]. The OpenModelica framework is also used as a testbench for new language

constructs, discussed at the Modelica design meetings held by the Modelica Association [7] approximately four times per year. Many of the ideas presented in this paper have originated from these meetings and some have been elaborated and refined.

The rest of the paper is organized as follows. The next section present the design of Meta-programming in Modelica and how it is used in the OpenModelica compiler. This is followed by a section on operator overloading in Modelica. The paper ends with conclusions and future work.

2 Meta-programming

Meta-programming is to write programs having other programs (so called object-programs) as data [12]. 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-program language are the same, like for instance in LISP or Java reflection. This is also the approach we have taken for Modelica. A language needs some way of representing the object-program as data. A simple and naive approach is to use strings for this. For example as follows:

```
String equationCode =
 "equation v = L*der(i);"
```

However, with this naive approach there is no internal structure of the object. We cannot even guarantee syntactic correctness, e.g. that the program inside the string corresponds to a valid (from a grammatical point of view) piece of code. This is a major disadvantage, and therefore most Meta-programming languages do not use this approach.

Another solution is to encode the object-program using data types of the meta-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 *datatype* for a Java class. The class has methods to query a Java class for its methods, members, iterfaces, etc.

Our current design uses built-in Modelica types to handle Modelica code, like for instance `TypeName` for a Modelica type name or `VariableName` for a Modelica variable name.

To create data values of the object program in the meta-program a *quoting* mechanism is needed. This approach is used in several different programming languages, such as Tick-C [4], LISP, MetaML [13] and Mathematica [14]. A quote is used to distinguish the object-program from the meta-program. For instance LISP use the quote character as quotation mechanism.

```
´(plus 1 3)
```

Furthermore to allow insertion of values into quoted expressions an anti-quote mechanism is needed. An anti-quote will lift the following expression up to the meta-program level, and it will thus be *evaluated* and replaced by a piece of object code. For example in LISP the anti-quote is the comma character.

```
(let x '(plus z 3))
`(foo ,x 1)
```

will result in

```
`(foo (plus z 3) 1)
```

2.1 Design Requirements

The requirements for meta-programming support in Modelica are the following:

- **Ease of use** Meta-programming should be easy to learn and use. This means that e.g. the syntax should be similar to what a Modelica programmer is used to. It should be possible to write small pieces of code and insert them with a singe command. For instance, adding an equation to an existing model should be a short one-line command.
- **Advanced** At the same time, it should be advanced enough to make it possible to perform the tasks needed by an advanced user who for instance wants to use meta-programming to write diagnosis applications, system identification, applications or other technically advanced tasks where a high level of automation is needed.
- **Backwards compatibility** The design of new language constructs and semantic behavior should be compatible with the current Modelica language [8]. This means that old Modelica code will work with these new extensions.

Considering these requirements, the proposed design is given in the next section.

2.2 Design for Meta-programming

For quoting Modelica code we propose the keyword `Code` together with parentheses and for anti-quoting we propose the keyword `Eval` also with parentheses.

A reasonable limitation for `Eval` is to only allow it in the same context as expressions, i.e. the Eval keyword including parenthesis can only be used where expressions can be used. This simplifies the parsing, without limiting the practical usage of the anti-quote mechanism.

The use of these two keywords are given in the following example:

```
myExpr := Code(der(x) + x);
```

Then the code expression

```
Code(equation x=Eval(myExpr))
```

will evaluate to

```
Code(equation x = der(x) + x;)
```

We also introduce built-in types for `Code` expressions. A type for any piece of Modelica code is in our proposal of the type `Code`. Then we have subtypes (subclasses) of this type for specific Modelica code pieces, like `VariableName` for code representing a variable name, `TypeName` for code representing a type name. Table 1 gives the type names for the proposed kinds of Modelica code that can be constructed using the `Code` quote.

Note that some of the cases are overlapping. For instance a variable reference (component_reference) is also an expression. In such cases, the most specialized type will be used. For instance, in this case the expression `Code(a.b.c)` will have the type `VariableName` which is a subtype of `TypeName` which is a subtype of `Expression`. This can be inconvenient in some cases when for instance we want to create a piece of code for a type name. To partly remedy this lack we introduce an optional extra "argument" to `Code` giving the type name of the `Code` piece. For example to create a type name we can write[1]

```
Code(Modelica.SIunits, TypeName)
```

Also, to fulfill the ease-of-use requirement to a greater extent, and allow for easy use of type and variable names as arguments to functions, we also propose an *automatic quoting* mechanism. The rule is quite simple and solves our problem mentioned above:

- *When the expected type (formal parameters and in operator arguments) of an expression is a subtype of Code (i.e. any of the types presented in Table 1), if the type of the argument expression is not a subtype of Code, the expression is automatically quoted, i.e. becomes a Code literal with the same type as the expected type.*

[1] This also helps in implementing a parser for Code constructs, since ambiguities can then be resolved by inspecting the second argument to Code.

For example, if we have a function `foo` taking a `TypeName` as an argument and we call it with

```
foo(a.b.c)
```

this will be automatically translated (by the automatic quoting mechanism) into

```
foo(Code(a.b.c,TypeName))
```

With these constructs at hand it is possible to start using Meta-programming by a set of built-in functions for updating Modelica code such as models and functions. Such functions are already partly available in the OpenModelica research compiler and will not be presented in further detail. Instead we will give an example on how to use Meta-programming and scripting functionality to achieve a parameter sweep on a Modelica model. The function is presented in Figure 2 and can be used as follows: We call the `paramSweep` function in the interactive environment and store the result in the variable r:

```
>> r:=paramSweep(test,R1.R,1:10);
>>
```

Then we call the function typeOf which returns a string representation of the type of a variable:

```
>> typeOf(res)
"SimulationResult[10]"
>>
```

which results in `SimulationResult[10]`, i.e. a vector of 10 elements with the element type being a record of information about a simulation execution.

2.3 Implementation in OpenModelica

In this section we will describe how the Meta-programming support has been implemented in OpenModelica. The support for the quoting mechanism `Code` and `Eval` is added to the internal representation (the abstract syntax tree or AST) using the following data types (in RML code):

```
datatype Code =
 TYPENAME of Path |
 VARIABLENAME of Component_ref |
 EQUATIONSECTION of bool *
    EquationItem list |
 ALGORITHMSECTION of bool
    * AlgorithmItem list |
 ELEMENT of Element |
 EXPRESSION of Exp |
 MODIFICATION of Modification
```

The datatype declarations in RML are similar to those in the Standard ML language. The vertical bar (pipe character) indicates alternatives, the capital letter words are names of node type constructors. For instance, a data object of type Code is:

Type	*Non-terminal in grammar*	*Description*
`TypeName`	name	The name of a type, e.g. `Modelica.SIunits`
`VariableName`	component_reference	The name of a variable, e.g. `a[3].b.c`
`EquationSection`	equation_clause	An equation section, e.g. `equation x=y; z=1;`
`AlgorithmSection`	algorithm_clause	An algorithm section, e.g. `algorithm x:=sin(y);`
`Element`	element	A class definition, components, import statements and extends clauses declared in a class.
`Expression`	expression	A Modelica expression, e.g. `foo(1:3,a+1)+PI`.
`Modification`	modification	A modification of a component declaration, extends clause, etc. for instance ”=1.5” or ”(R=10)”.

Figure 1: Types for Code expressions.

```
function paramSweep
  "A function for performing a parameter sweep of a model"
  input TypeName modelName;
  input VariableName variable;
  input Real values[:];
  input Real startTime=0.;
  input Real stopTime=1.;
  output SimulationResult result[size(values,1)];
protected
  Boolean flag;
  SimulationObject simObj;
algorithm
  (flag,simObj) := translateClass(modelName);
  assert(flag,"Error translating class.");
  for i in values loop
result[i]:=simulate(
startTime=startTime,
        stopTime=stopTime,
        params = SimulateParams(
    {Code(Eval(variable) = Eval(values[i]))},{})
);
// If variable is R1.R and values[i] is 5.6
// then parameters is Code(R1.R=5.6)
  end for;
end paramSweep;
```

Figure 2: A parameter sweep function using Meta-programming.

```
TYPENAME(
   QUALIFIED("A",
   IDENT("B")
   )
)
```

which represents the AST for a typename `A.B`. The new AST type Code contain several different AST nodes, such as a `Path` node for representing type names, etc. The boolean values for equation and algorithm sections indicate whether the section has prefix initial, e.g. if it is an initial equation or initial algorithm. The `Eval` construct does not need additional AST types since it is limited to be used as an expression and can thus be expressed as a built-in operator (e.g. a function call in the grammar).

The semantic parts needed for `Code` and `Eval` are straightforward to implement. For instance resolving types for `Code` expressions can be done immediately by using the built-in types presented in Table 1. The semantic rules for `Eval` must ensure that the evaluated Modelica code has the correct type for insertion in the abstract syntax of its context, i.e. that the result from a eval expression is indeed an expression.

3 Function Overloading

What does it mean to have overloading of operators and functions in a language, and why do we need it? The main reason to have this mechanism in a programming language is economy of notation — overloading allows us to reuse well-known notation and names for more than one kind of data structure. This is convenient and gives more concise and readable code. The concept of overloading can be defined roughly as follows:

- A function or operator is *overloaded* if it has several definitions, each with a different type signature.

The concept of Modelica function type signature is the same as the Modelica class type of the function, and can be defined roughly as follows:

- A Modelica function type signature consists of the set of input formal parameter and result types together with the names of the formal parameters and results, in the order they are defined. To avoid certain lookup and type resolution difficulties, overloading is defined based on the input formal parameters only.

In fact, overloading already exists to a limited extent for certain operators in the standard Modelica 2.1. For example, the plus (+) operator for addition has several different definitions depending on the data type:

- $1+2$ – means integer addition of two integer constants giving an integer result, here 3.
- $1.0+2.0$ – means floating point number addition of two Real constants giving a floating-point number result, here 3.0.
- ”ab”+”2” – means string concatenation of two string constants giving a string result, here ”ab2”.
- $\{1,2\}+\{3,4\}$ – means integer vector addition of two integer constant vectors giving a vector result, here $\{4,6\}$.

Overloading of certain built-in functions also exists. For example, the `size` function is defined for arrays of different functionality and occurs in two variants: with one (e.g. `size(A)`) or two arguments (e.g. `size(A,1)`). Scalar functions of one or more arguments are implicitly defined also for arrays. However, the above examples are just special cases. It is very desirable for the user to be able to define the standard operators as overloaded for user-defined data structures of choice, and to define different overloaded variants of functions with the same name.

To handle function overloading a new short class definition construct is defined, similar to the enumeration defintion. It introduces the new keyword `overload` and has the following grammar rule (added to the class specifier rule):

```
'=' overload '(' name_list ')'
```

where `name_list` is a list of type names. It can only be used to define functions, like for instance:

```
function solve =
    overload(denseSolve,
             sparseSolve,
             bandSolve);
```

The description of user-defined overloaded operators and functions in Modelica presented here is based on design proposals that have been discussed at several Modelica design meetings by the Modelica Association. The presentation here is roughly the outcome of those discussions, with a few small details added. This design has reached the stage of being approved for test implementation, but not yet made it into the Modelica language specification. Thus, there might be some changes in the final version.

Operator	*Operator Example*	*Function*	*Function Example*
+	a+b	plus	plus(a,b)
+	+a	unaryPlus	unaryPlus(a)
-	a-b	minus	minus(a,b)
-	-a	unaryMinus	unaryMinus(a)
*	a*b	times	times(a,b)
/	a/b	divide	divide(a,b)
^	a^b+	power	power(a,b)
=	a=b	equal	equal(a,b)
:=	a:=b	assign	assign(a,b)
<	a<b	less	less(a,b)
<=	a<=b	lessEqual	lessEqual(a,b)
==	a==b	equalEqual	equalEqual(a,b)
>=	a>=b	greaterEqual	greaterEqual(a,b)
>	a>b	greater	greater(a,b)
<>	a<>b	notEqual	notEqual(a,b)
[ ]	a[b,c,...]	index	index(a,{b,c,...})
[ ] :=	a[b,c,...] := v	indexedAssign	indexedAssign(a,{b,c,...},v)

Figure 3: Overloaded operators together with their associated built-in function names

3.1 Operator Overloading

The mechanism for overloading operators is only defined for the standard operators mentioned in Table 3. Adding arbitrary new operators is not possible. Each operator is associated with the name of a built-in function, as defined in Table 3. Note that equality = and assignment := are not expression operators since they are allowed only in equations and in assignment statements respectively. All binary expression operators are left associative. When an operator is applied to some arguments, e.g. a+b, this is interpreted as an application of the corresponding built-in function, e.g. plus(a,b). The usual lookup of the function definition of plus is performed. If a user-defined function plus with matching type signature is found, this function is used, otherwise the standard built-in operator/function +/plus implicitly available in the top-level scope is found if it is defined for the argument data types in question. For example, two addition functions named plus are defined within the same scope, where each definition can be distinguished by the nonequivalence of the second formal parameter types:

```
function plus
  input  Real      x;
  input  Real[2]   y;
  output Real      z;
  ...
end plus;
function plus
  input  Real      x;
  input  MyRecord  y;
  output Real      z;
  ...
end plus;
```

A user-defined record class DiagonalMatrix, shown in figure 4, defines the + (plus) and the [] (index) operators for diagonal matrices that are internally represented compactly as vectors using the DiagonalMatrix data type.

3.1.1 Lookup Rules

Lookup of function definitions (or operators represented by their corresponding built-in functions) will follow the usual Modelica lookup rules[8], with the following additions:

- Both the function name and the input formal parameter types of the called function are used during the lookup process to distinguish matching definitions. The matching criterion for lookup of functions is identity of function names and equivalence of input formal parameter types. In such a match, if the function names are identical and some argument types are not equivalent to corresponding formal parameter types, assignment conversion of argument types, e.g. from Integer to Real, is performed when applicable, and then equivalence of types is checked once more for failure or success of the match.

```
package DiagonalMatrices
  record DiagonalMatrix  "Diagonal matrix stored compactly as a vector"
    Real v[:]      "Compact vector representation";
  end DiagonalMatrix;
  function plus  "Addition of diagonal matrices"
    input  DiagonalMatrix  a;
    input  DiagonalMatrix  b;
    output DiagonalMatrix  c;
  algorithm
    // Insert matrix size checks here
     c.v := a.v + b.v;  // Use builtin array assignment
  end plus;
  function index  "Indexing of a diagonal matrix"
    input  DiagonalMatrix  a;
    input  Integer         b[2]; // Exactly two indices are allowed
    output Real            c;
  algorithm
    c := if (b[1] == b[2]) then v[b[1]] else 0;
  end index;
end DiagonalMatrices;
```

Figure 4: The DiagonalMatrix example, using operator overloading for addition.

- There is an implicit ”import” of the packages containing the function argument type definitions, where the desired operator or function definition also might be found. If there is a package scope containing the first argument type definition, this scope is searched first during lookup. If this fails, the package scope containing the second argument type is searched, etc., until this procedure has been repeated for all arguments having a user-defined type. This is the Koening lookup rule originally used for lookup of overloaded definitions in C++.

The second rule might sound strange, but makes the lookup more specific, and avoids the need to write many import statements specifically for importing function definitions. It is enough to refer to the argument type. For example:

```
  Matrices.Symmetric.Matrix  A4;
equation
  solve(A4,v2) + func2(5+5,v2) = 0;
```

Here the first argument to solve is the variable `A4`. Its type is `Symmetric.Matrix` defined within the package `Matrices.Symmetric`. Therefore the scope of this package is searched first during the lookup, and the function `solve` is found. However, `func2` is searched in the usual way since the type of `v2` is not defined within any package scope.

3.2 Implementation in OpenModelica

Since operator overloading already exist in Modelica today, the design and implementation of operator and function overloading can be performed at a low effort. The largest change is to introduce Koening lookup mechanism. For this purpose we add the fully qualified type name to a type, giving a new definition of a type as a tuple

```
type Type = (TType * Absyn.path option)
```

Thus, a type now have the fully qualified class name of its definition, making it possible to search for function definitions from the scope where the type is defined, i.e. the Koening C++ lookup rule.

The same class name can also be used for function types when deoverloading.

This is the major change needed to the type system. The rest of the implementation is concerned with adding the rules to the lookup mechanism and the actual deoverloading mechanism, when the overloaded names are looked up and replaced with the correct function name, according to the types of the input arguments of the function call.

The deoverloading of function calls is performed by traversing a list of function types until a match is found. The only addition needed in this case is to add the function type candidates through the koening lookup rule. For this purpose we add the relation[2]:

[2] A RML relation can be seen as a function call, taking arguments as input and producing outputs

```
relation get_koening_function_types:
    (Env.Env,
    Absyn.Path,
    Absyn.Exp list,
    Absyn.NamedArg list)
      => Types.Type list
```

Its arguments are

- `Env.Env` The environment for lookup of types, classes,etc.
- `Absyn.Path` The function name, e.g. `A.foo` or `solve`.
- `Absym.Exp list` A list of expressions for the positional arguments.
- `Absyn.NamedArg list` A list of named arguments (a pair of identifiers and expressions)

The result from this relation is a list of function types, to be added to the rest of the function type candidates for the deoverloading process. The relation checks each expression in order to find its type. If the type is user defined, it will look in the types scope to find potential function types.

Below follows a short example run in OpenModelica, using Complex numbers and operator overloading. First we present a short Complex number package:

```
encapsulated package ComplexNumbers
  record Complex
    Real re;
    Real im(start=0);
  end Complex;

  function foo = overload(
        complexFoo,realFoo);

  function complexFoo
    input Complex x;
    input Complex y;
    output Complex res(
       im=x.im + y.im,
       re=x.re + 2 * y.re);
  end complexFoo;

  function realFoo
    input Real x;
    input Complex y;
    output Complex res(
       im=y.im,
       re=x + y.re);
  end realFoo;

  function plus
    "Overloaded user-defined
     complex number addition"
    input Complex   x;
    input Complex   y;
    output Complex  res(
       re = x.re + y.re,
       im = x.im + y.im);
  end plus;

  function times
    "Overloaded user-defined
     complex number multiplication"
    input Complex x;
    input Complex y;
    output Complex res(
       re = x.re * y.re
            - x.im * y.im,
       im = x.re * y.im
            + x.im * y.re);
  end times;

  function unaryMinus
    "Overloaded user-defined
     complex number unary minus"
    input Complex x;
    output Complex res(re = -x.re,
                       im = x.im);
  end unaryMinus;
end ComplexNumbers;
```

The package also contain an overloaded function `foo`, for illustration of the overload operator. Then we define a test class that uses operator and function overloading:

```
model test
  import ComplexNumbers.Complex;
  import Vectors.Q4Position;
  Complex c1,c2,c3;
  Q4Position p1,p2,p3;
equation
  c1=c2+c3; // ComplexNumbers.plus
  c2=c1*c3; // ComplexNumbers.times
  c3=-c2;  // ComplexNumbers.unaryMinus
  c2=foo(c1,c2);
       // ComplexNumbers.complexFoo
  c3=foo(1.0,c1*c3);
       // ComplexNumbers.realFoo
  p1=foo(p2,p3);  // Vectors.foo
  p1=p2+p3;  // Vectors.plus
end test;
```

We translate the model in the OpenModelica environment:

```
>>> translateClass(test)
record
flatClass = "fclass test
Real c1.re;
Real c1.im;
Real c2.re;
Real c2.im;
Real c3.re;
Real c3.im;
Real p1[1];
Real p1[2];
Real p1[3];
Real p1[4];
```

```
Real p2[1];
Real p2[2];
Real p2[3];
Real p2[4];
Real p3[1];
Real p3[2];
Real p3[3];
Real p3[4];
equation
  TMP0 = ComplexNumbers.plus(c2,c3);
  c1.re = TMP0.re;
  c1.im = TMP0.im;
  TMP1 = ComplexNumbers.times(c1,c3);
  c2.re = TMP1.re;
  c2.im = TMP1.im;
  TMP2 = ComplexNumbers.unaryMinus(c2);
  c3.re = TMP2.re;
  c3.im = TMP2.im;
  TMP3 = ComplexNumbers.complexFoo(c1,
   c2);
  c2.re = TMP3.re;
  c2.im = TMP3.im;
  TMP4 = ComplexNumbers.realFoo(1.0,
      ComplexNumbers.times(c1,c3));
  c3.re = TMP4.re;
  c3.im = TMP4.im;
  p1 = Vectors.foo(p2,p3);
  p1 = p2 + p3;
end test;
",
    exefile = ""

 end record
```

4 Conclusions

In this paper we have presented two new areas of interest for the design of the Modelica modeling language, Meta-programming and function overloading. A design of these two language features have been presented and a test implementation has been made in the OpenModelica environment. The effort for implementing these two features have been low, especially for function overloading since most of the required mechanisms were already in place.

The OpenModelica research compiler for Modelica also has some rudimentary support for simulation of Modelica models. This makes it at the same time an interesting tool and/or for Modelica beginners, wanting to learn the language or use Modelica as a computational language, a free replacement of e.g. Matlab or Mathematica.

Function and operator overloading are two modern language mechanisms that make it easier for a user to write programs. Thus, these two new potential additions to the Modelica language will strengthen the Modelica language as a computational programming language, allowing users to write sophisticated numerical computation code, which also allow fast execution due to the Modelica type system. This aspect will also be aided by the Meta-programming mechanisms, which will allow users to *program* models using scripts, to be used in e.g. design optimization, system diagnosis, and adapting models in a more flexible way for large and complex system modeling.

Future work on the compiler includes implementing full support for Modelica version 2.1. Also, better support for simulation of models must be added. There is also a great need of an updated Modelica test suite, to be able to test modelica compilers against the specification.

References

[1] P. Aronsson. Licentiate thesis: *Automatic Parallelization of Simulation Code from Equation Based Simulation Languages*. Department of Computer and Information Science, Linköpings universitet, Sweden, 2002.

[2] P. Bunus. Licentiate thesis: *Debugging and Structural Analysis of Declarative Equation-Based Languages*. Department of Computer and Information Science, Linköpings universitet, Sweden, 2002.

[3] P. Fritzson, V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European conference on Object-Oriented Programming, LNCS*. Springer Verlag, 1998.

[4] Dawson R. Engler and Massimiliano Poletto. A 'c tutorial.

[5] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation*. Wiley - IEEE Press, 2003. ISBN 0-471-471631.

[6] P. Fritzson, P. Aronsson, P. Bunus, V. Engelson, L. Saldamli, H. Johansson, and A. Karström. The open source modelica project. In *Proceedings of the 2nd International Modelica Conference, Germany*, March 2002.

[7] The modelica association. http://www.modelica.org.

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

[9] P.Aronsson, P. Fritzson, L. Saldamli, and P. Bunus. Incremental declaration handling in open source modelica. In *SIMS - 43rd Conference on Simulation and Modeling on September 26-27*, Oulu, Finland, 2002.

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

[11] L. Saldamli. Licentiate thesis: *PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations*. Department of Computer and Information Science, Linköpings universitet, Sweden, 2002.

[12] Tim Sheard. Accomplishments and research challenges in meta-programming. *Lecture Notes in Computer Science*, 2196:2–??, 2001.

[13] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[14] S. Wolfram. *The Mathematica Book, 5th Ed.* Wolfram Media, Inc, 2003.