MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language*

Adrian Pop and Peter Fritzson

Programming Environment Laboratory
Department of Computer and Information Science
Linköping University, 58183 Linköping, Sweden
{adrpo, petfr}@ida.liu.se

Abstract. For a long time, one of the major research goals in the computer science research community has been to raise the level of abstraction power of specification languages/programming languages. Many specification languages and formalisms have been invented, but unfortunately very few of those are practically useful, due to limited computer support of these languages and/or inefficient implementations. Thus, one important goal is executable specification languages of high abstraction power and with high performance, good enough for practical usage and comparable in execution speed to hand implementations of applications in low-level languages such as C or C++. In this paper we briefly describe our work in creating efficient executable specification languages for two application domains. The first area is formal specification of programming language semantics, whereas the second is formal specification of complex systems for which we have developed an object-oriented mathematical modeling language called Modelica, including architectural support for components and connectors. Based on these efforts, we are currently working on a unified equation-based mathematical modeling language that can handle modeling of items as diverse as programming languages, computer algebra, event-driven systems, and continuous-time physical systems. The key unifying feature is the notion of equation. In this paper we describe the design and implementation of the unified language. A compiler implementation is already up and running, and used for substantial applications.

1 Introduction

For a long time, one of the major research goals in the computer science research community has been to raise the level of abstraction power of specification languages/programming languages. Many specification languages and formalisms have been invented, but unfortunately very few of those are practically useful, due to limited computer support of these languages and/or inefficient implementations.

In this paper we briefly describe our existing work in creating efficient executable specification languages for two application domains and propose an integration of this work within a unified language for mathematical and semantical modeling.

* This work was supported by the SSF RISE project, the Vinnova SWEBPROD project, and by the CUGS graduate school.

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 211 – 229, 2006.

Thus, the main goal of this work is the design and development of a *general executable mathematical modeling and semantics meta-modeling language*. This language should have a clean semantics as in the case of Modelica and Natural Semantics (RML), and should be compiled to code of high performance. This language will allow expressing mathematical models but also meta-models and meta-programs that specify composition of models, transformation of models, model constraints, etc. This language is based on Modelica extended with several new language constructs that allows program language specification. The unified language is called MetaModelica.

The paper is structured as follows: In the next section we present the starting background for the development of the MetaModelica unified language. Section 3 presents the proposed mathematical/semantical unified modeling language. In Section 4 we present the implementation of the MetaModelica compiler for the unified language. Section 5 presents performance evaluation of our generated code. Section 6 presents future work. Conclusions and Future work are presented in Section 7.

2 Background

About sixteen years ago, our research group has selected two application domains for research on high-level specification languages:

- Specification languages for programming language semantics. Much work has been done in that area, but there is still no standard class of compiler-compiler tools around, as successful as parser generators based on grammars in BNF form like lex (flex), yacc (bison), ANTLR, etc.
- Equation-based specification languages for mathematical modeling of complex (physical) systems.

In the following sections we briefly describe the main achievements of this work.

2.1 Natural Semantics and the Relational Meta-Language (RML)

Concerning *specification languages for programming language semantics*, compiler generators based on denotational semantics (Pettersson and Fritzson 1992 [24]) (Ringström et al. 1994 [29]), were investigated and developed with some success. However this formalism has certain usage problems, and Operational Semantics/Natural Semantics started to become the dominant formalism in common literature. Therefore a meta-language and compiler generator called RML (Relational Meta Language) (Fritzson 1998 [8], PELAB 1994-2005 [21], Pettersson 1995 [25], 1999 [26]) for Natural Semantics was developed, which we have used extensively for full-scale specifications of languages like Java (object oriented), C subset with pointer arithmetic, functional, and equation-based languages (Modelica). Generated implementations are comparable in performance to hand implementations. However, it turned out that development environment support is needed also for specification languages. Recent developments include a debugger for Natural Semantics specifications (Pop and Fritzson 2005 [28]).

Natural Semantics (Kahn 1988 [16]) is a specification formalism that is used to specify the semantics of programming languages, i.e., type systems, dynamic semantics,

translational semantics, static semantics (Despeyroux 1984 [4], Glesner and Zimmermann 2004 [14]), etc. Natural Semantics is an operational semantics derived from the Plotkin (Plotkin 1981 [27]) structural operational semantics combined with the sequent calculus for natural deduction. There are few systems implemented that compile or interpret Natural Semantics.

One of these systems is Centaur (Borras et al. 1988 [1]) with its implementation of Natural Semantics called Typol (Despeyroux 1984 [4], 1988 [5]). This system is translating the Natural Semantics inference rules to Prolog.

The Relational Meta-Language (RML) is a much more efficient implementation of Natural Semantics, with a performance of the generated code that is several orders of magnitude better than Typol. The RML language is compiled to highly efficient C code by the rml2c compiler. In this way large parts of a compiler can be automatically generated from their Natural Semantics specifications. RML is successfully used for specifying and generating practically usable compilers from Natural Semantics for Java, Modelica, MiniML (Clément et al. 1986 [3]), Mini-Freja (Pettersson 1995 [25]) and other languages.

2.1.1 An Example of Natural Semantics and RML

As a simple example of using Natural Semantics and the Relational Meta-Language (RML) we present a trivial expression (Exp1) language and its specification in Natural Semantics and RML. A specification in Natural Semantics has two parts:

- Declarations of syntactic and semantic objects involved.
- Groups of inference rules which can be grouped together into relations.

In our example language we have expressions built from numbers. The abstract syntax of this language is declared in the following way:

integers:

$v \in Int$

expressions (abstract syntax):

$e \in Exp ::= v \mid e1 + e2 \mid e1 - e2 \mid e1 * e2 \mid e1 / e2 \mid -e$

The inference rules for our language are bundled together in a judgment $e \Rightarrow v$ in the following way (we do not present here the similar rules for the other operators.):

(1) $v \Rightarrow v$

(2) $$\frac{e1 \Rightarrow v1 \;\; e2 \Rightarrow v2 \;\; v1+v2 \Rightarrow v3}{e1+e2 \Rightarrow v3}$$

RML modules have two parts, an interface comprising datatype declarations (abstract syntax) and signatures of relations (judgments) that operate on such datatypes, followed by the declarations of the actual relations which group together rules and axioms. In RML, the Natural Semantics specification shown above is represented as follows:

```
module Exp1:

  (* Abstract syntax of the language Exp1 *)
  datatype Exp =  RCONST of real
               |  ADD    of Exp * Exp
               |  SUB    of Exp * Exp
               |  MUL    of Exp * Exp
               |  DIV    of Exp * Exp
               |  NEG    of Exp
  relation eval: Exp => real
end
(* Evaluation semantics of Exp1 *)
relation eval: Exp => real  =

 (* Evaluation of a real node is the real number itself *)
 axiom eval(RCONST(rval)) => rval

 (* Evaluation of an addition node ADD is v3, if v3 is the result of
    adding the evaluated results of its children e1 and e2. *)
 rule  eval(e1) => v1 & eval(e2) => v2 &  v1 + v2 => v3
       ------------------------------------------------
       eval( ADD(e1, e2) ) => v3

 rule  eval(e1) => v1 & eval(e2) => v2 &  v1 - v2 => v3
       ------------------------------------------------
       eval( SUB(e1, e2) ) => v3

 rule  eval(e1) => v1 & eval(e2) => v2 &  v1 * v2 => v3
       ------------------------------------------------
       eval( MUL(e1, e2) ) => v3

 rule  eval(e1) => v1 & eval(e2) => v2 &  v1 / v2 => v3
       ------------------------------------------------
       eval( DIV(e1, e2) ) => v3

 rule  eval(e) => v & -v => vneg
       -------------------------
       eval( NEG(e) ) => vneg

end (* eval *)
```

A proof-theoretic interpretation can be assigned to this specification. We interpret inference rules as recipes for constructing proofs. We wish to prove that there is a value v such that $1+2 \Rightarrow v$ holds for this specification. To prove this proposition we need an inference rule that has a conclusion, which can be instantiated (matched) to the proposition. The only proposition that matches is the second proposition (2), which is instantiated as follows:

$$\frac{1 \Rightarrow v1 \;\; 2 \Rightarrow v2 \;\; v1+v2 \Rightarrow v}{1+2 \Rightarrow v}$$

To continue the proof, we need to apply the first proposition (axiom) several times, and we soon reach the conclusion. One can observe that debugging of Natural Semantics comprise proof-tree understanding.

2.1.2 Specification of Syntax

Regarding the specification of lexical and syntatic rules for a new language, we use external tools such as Lex, Yacc, Flex, Bison, etc., to generate those modules. The parser builds abstract syntax by calling RML-defined constructors. The abstract syntax is then passed from the parser to the RML-generated modules. We currently use the same approach for languages defined using MetaModelica.

2.2 Modelica – An Object-Oriented Equation-Based Component Language

Starting 1989, we developed an equation-based specification language for mathematical modeling called ObjectMath (Viklund et al. 1992 [36]), using Mathematica as a basis and a frontend, but adding object orientation and efficient code generation was developed. Following this path our group joined effort with several other groups in object oriented mathematical modeling to start a design-group for developing an internationally viable declarative mathematical modeling language. The language resulted from this effort is called *Modelica.* Modelica (Elmqvist et al. 1999 [7], Fritzson 2004 [13], Fritzson and Engelson 1998 [9], Modelica-Association 1996-2005 [18], Tiller 2001 [35]) is an object-oriented modeling language for declarative equation-based mathematical modeling of large and heterogeneous physical systems. For modeling with Modelica, commercial software products such as MathModelica (MathCore [17]) or Dymola (Dynasim 2005 [6]) have been developed. Also open-source implementations like the OpenModelica system (Fritzson et al. 2002 [10], PELAB 2002-2005 [22]) are available.

The Modelica language has been designed to allow tools to generate efficient simulation code automatically with the main objective of facilitating exchange of models, model libraries and simulation specifications. The definition of simulation models is expressed in a declarative manner, modularly and hierarchically. Various formalisms can be combined with the more general Modelica formalism. In this respect Modelica has a multi-domain modeling capability which gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. Compared to most other modeling languages available today, Modelica offers several important advantages from the simulation practitioner's point of view:

- Acausal modeling based on ordinary differential equations (ODE) and differential algebraic equations and discrete equations (DAE). There is also ongoing research to include partial differential equations (PDE) in the language syntax and semantics (Saldamli et al. 2002 [31]), (Saldamli 2002 [30], Saldamli et al. 2005 [32]).
- Multi-domain modeling capability, which gives the user the possibility to combine electrical, mechanical, thermodynamic, hydraulic etc., model components within the same application model.
- A general type system that unifies object-orientation, multiple inheritance, and generics templates within a single class construct. This facilitates reuse of components and evolution of models.
- A strong software component model, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.

- Visual drag & drop and connect composition of models from components present in different libraries targeted to different domains (electrical, mechanical, etc).

The language is strongly typed and declarative. See (Modelica-Association 1996-2005 [18]), (Modelica-Association 2005 [19]), (Tiller 2001 [35]), and (Fritzson 2004 [13]) for a complete description of the language and its functionality from the perspective of the motivations and design goals of the researchers who developed it. Shorter overviews of the language are available in (Elmqvist et al. 1999 [7]), (Fritzson and Engelson 1998 [9]), and (Fritzson and Bunus 2002 [12]).

2.2.1 An Example Modelica Model

The following is an example Lotka Volterra Modelica model containing two differential equations relating the sizes of rabbit and fox populations which are represented by the variables `rabbits` and `foxes`: The rabbits multiply; the foxes eat rabbits. Eventually there are enough foxes eating rabbits causing a decrease in the rabbit population, etc., causing cyclic population sizes. The model is simulated and the sizes of the rabbit and fox populations as a function of time are plotted in Fig. 1.

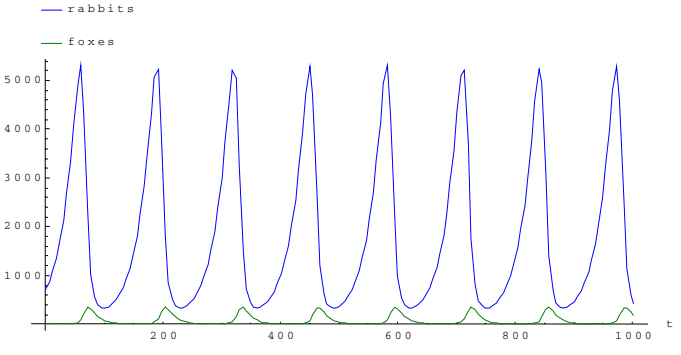

Fig. 1. Number of rabbits – prey animals, and foxes - predators, as a function of time simulated from the PredatorPrey model

The notation `der(rabbits)` means time derivative of the `rabbits` (population) variable.

```
class LotkaVolterra
  parameter Real g_r =0.04    "Natural growth rate for rabbits";
  parameter Real d_rf=0.0005  "Death rate of rabbits due to foxes";
  parameter Real d_f =0.09    "Natural deathrate for foxes";
  parameter Real g_fr=0.1     "Efficency in growing foxes from
  rabbits";
   Real     rabbits(start=700) "Rabbits,(R) with start population
  700";
   Real     foxes(start=10)    "Foxes,(F) with start population 10";
equation
  der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
  der(foxes)   = g_fr*d_rf*rabbits*foxes - d_f*foxes;
end LotkaVolterra;
```

2.2.2 Modelica as a Component Language

Modelica offers quite a powerful software component model that is on par with hardware component systems in flexibility and potential for reuse. The key to this increased flexibility is the fact that Modelica classes are based on equations, i.e., acausal connections for which the direction of data flow across the connection is not fixed. Components are connected via the connection mechanism, which can be visualized in connection diagrams. The component framework realizes components and connections, and ensures that communication works and constraints are maintained over the connections. For systems composed of acausal components the direction of data flow, i.e., the causality is automatically deduced by the compiler at composition time.

Two types of coupling can be established by connections depending on whether the variables in the connected connectors are nonflow (default), or declared using the `flow` prefix:

1. Equality coupling, for nonflow variables, according to Kirchhoff's first law.
2. Sum-to-zero coupling, for flow variables, according to Kirchhoff's current law.

For example, the keyword `flow` for the variable `i` of type `Current` in the `Pin` connector class indicates that all currents in connected pins are summed to zero, according to Kirchhoff's current law.

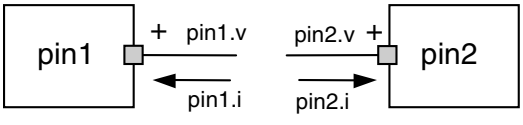

Fig. 2. Connecting two components that have electrical pins

Connection equations are used to connect instances of connection classes. A connection equation `connect (pin1,pin2)`, with `pin1` and `pin2` of connector class `Pin`, connects the two pins (Fig. 2) so that they form one node. This produces two equations, namely:

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's second law, saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flows in piping networks and to forces and torques in mechanical systems.

3 MetaModelica – A Unified Equation-Based Modeling Language

The idea to define a unified equation-based mathematical and semantical modeling language started from the development of the OpenModelica compiler (Fritzson et al.

2002 [11]). The entire compiler is generated from a Natural Semantics specification written in RML. The open source OpenModelica compiler has its users in the Modelica community which have detailed knowledge of Modelica but very little knowledge of RML and Natural Semantics. In order to allow people from the Modelica community to contribute to the OpenModelica compiler we retargeted the development language from RML to MetaModelica, which is based on the Modelica language with several extensions. We already translated the OpenModelica compiler from RML to the MetaModelica using an automated translator (Carlsson 2005 [2]) implemented in RML. We also developed a compiler which can handle the entire OpenModelica compiler specification (~105000 lines of code) defined in MetaModelica. An evaluation of the performance of the generated code is presented in section 6.

The basic idea behind the unified language is to use equations as the unifying feature. Most declarative formalisms, including functional languages, support some kind of limited equations even though people often do not regard these as equations, e.g. single-assignment equations.

Using the meta-programming facilities, usual tasks like generation, composition and querying of Modelica models can be automated.

The MetaModelica language inherits all the strong component capabilities Modelica. Components can be reused in different contexts because the causality is not fixed in equations and is up to the compiler to decide it.

3.1 The Types of Equations in the Unified Language

In the following we present the current types of equations already present in Modelica and detail the addition of the equations that support the definition of semantic specifications.

3.1.1 Mathematical Equations

Mathematical models almost always contain equations. There are basically four main kinds of mathematical equations in Modelica which we detail below.

Differential equations contain time derivatives such as $\frac{dx}{dt}$, usually denoted $\dot{x}$:

$$\dot{x} = a \cdot x + 3 \tag{1}$$

Algebraic equations do not include any differentiated variables:

$$x^2 + y^2 = L^2 \tag{2}$$

Partial differential equations also contain derivatives with respect to other variables than time:

$$\frac{\partial a}{\partial t} = \frac{\partial^2 a}{\partial z^2} \tag{3}$$

Difference equations express relations between variables, e.g. at different points in time:

$$x(t+1) = 3x(t) + 2 \tag{4}$$

3.1.2 Conditional Equations and Events

Behavior can develop continuously over time or as discrete changes at certain points in time, usually called events. It is possible to express events and discrete behavior solely based on conditional equations. An event in Modelica is something that happens that has the following four properties:

- A point in time that is instantaneous, i.e., has zero duration.
- An event condition that switches from false to true for the event to happen.
- A set of variables that are associated with the event, i.e., are referenced or explicitly changed by equations associated with the event.
- Some behavior associated with the event, expressed as conditional equations that become active or are deactivated at the event. Instantaneous equations are a special case of conditional equations that are active only at events.

Modelica has several constructs to express conditional equations, e.g. if-then-else equations for conditional equations that are active during certain time durations, or when-equations for instantaneous equations.

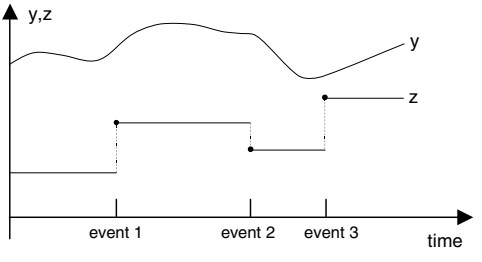

Fig. 3. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y may change both between and at events

3.1.3 Single-Assignment Equations

A single-assignment equation is quite close to an assignment, e.g.:

```
x = eval_expr(env, e);
```

but with the difference that the unbound variable (here `x`) which obtains a value by solving the equation, only gets its value once, whereas a variable in an assignment may obtain its value several times, e.g.:

```
x := eval_expr(env, e); x := eval_expr2(env, x);
```

3.1.4 Pattern Equations in Match Expressions

In this section we present our addition to the Modelica language which allows definitions of semantic specifications. The new language features are pattern equations, match expressions and union datatypes.

Pattern equations are a more general case than single-assignment equations, e.g.:

```
Env.BOOLVAL(x,y) = eval_something(env, e);
```

Unbound variables get their values by using pattern-matching (i.e., unification) to solve for the unbound variables in the pattern equation. For example, `x` and `e` might be unbound and solved for in the equations, whereas `y` and `env` could be bound and just supply values.

The following extension to Modelica is essential for specifying semantics of language constructs represented as abstract syntax trees:

- Match expressions with pattern-matching case rules, local declarations, and local equations.

It has the following general structure:

```
match expression  optional-local-declarations
  case pattern-expression opt-local-declarations
    optional-local-equations then value-expression;
  ...
  else optional-local-declarations
    optional-local-equations then value-expression;
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.

A match expression is closely related to pattern matching in functional languages, but is also related to switch statements in C or Java. It has two important advantages over traditional switch statements:

- A match expression can appear in any of the three Modelica contexts: expressions, statements, or in equations.
- The selection in the case branches is based on pattern matching, which reduces to equality testing in simple cases, but is unification in the general case.

Local equations in match expressions have the following properties:

- Only algebraic equations are allowed as local equations, no differential equations.
- Only locally declared variables (local unknowns) declared by local declarations within the case expression are solved for, or may appear as pattern variables.
- Equations are solved in the order they are declared (this restriction may be removed in the future, allowing more general local algebraic systems of equations).
- If an equation or an expression in a case-branch of a match-expression fails, all local variables become unbound, and matching continues with the next branch.

We also need to introduce the possibility to declare recursive tree data structures in Modelica, e.g.:

```
uniontype Exp
  record RCONST Real x1; end RCONST;
  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;
```

A small expression tree, of the expression `12+5*13`, is depicted in Fig. 4. Using the record constructors `PLUS`, `MUL`, `RCONST`, this tree can be constructed by the expression `PLUS(RCONST(12), MUL( RCONST(5), RCONST(13)))`

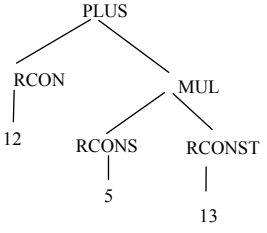

Fig. 4. Abstract syntax tree of the expression `12+5*13`

The `uniontype` construct has the following properties:

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

This is a preliminary union type design, which however is very close (just different syntax) to similar datatype constructs in declarative languages such as Haskell, Standard ML, OCaml, and RML. The uniontypes can model any abstract syntax tree while the match expressions are used to model the semantics, composition or transformation of the specified language.

3.2 Solution of Equations

The process of solving systems of equations is central for the execution of equation-based languages. For example:

- Differential equations are solved by numeric differential equation solvers.
- Differential-algebraic equations are solved by numeric DAE solvers.
- Algebraic equations are solved by symbolic manipulation and/or numeric solution
- Single-assignment equations are solved by performing an assignment.
- Pattern equations are solved by the process of unification which assigns values to unbound variables in the patterns.

The first three solution procedures are used in current Modelica. By the addition of local equations (Section 3.1.4) in match expressions to be solved at run-time, we generalize the allowable kinds of equations in Modelica.

3.3 Evaluator for the Exp1 Language in the Unified Language

As an example of the meta-modeling and meta-programming capabilities of the MetaModelica we give a very simple example. The semantics of the operations in the small expression language Exp1 follows below, expressed as an interpretative language specification in Modelica in a style close to Natural and/or Operational Semantics, see Exp1 specified in RML in Section 2.1.1. Such specifications typically consist of a number of functions, each of which contains a match expression with one or more cases, also called rules. In this simple example there is only one function, here called `eval`, since we specify an expression evaluator.

```
function eval
  input  Exp  in_exp;
  output Real out_real;
algorithm
 out_real :=
  match in_exp
    local Real v1,v2,v3;  Exp e1,e2;
    case RCONST(v1) then v1;
    case ADD(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 + v2;  then v3;
    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 - v2;  then v3;
    case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 * v2;  then v3;
    case DIV(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = v1 / v2;  then v3;
    case NEG(e1) equation
      v1 = eval(e1); v2 =  -v1;  then v2;
   end match;
end eval;
```

As usual in Modelica the equations are not directional, e.g. the two equations `v1 = eval(e1)` and `eval(e1) = v1` are equivalent. The compiler will select one of the forms based on input/output parameters and data dependencies.

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, statements, expressions) 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. 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 the `then` keyword before the returned value? The code becomes easier to read if there is a keyword before the returned value-expression. Note that most functional languages use the `in` keyword instead in this context, which is less

intuitive, and would conflict with the set or array element membership meaning the Modelica in keyword.

4 Details of the Compiler Implementation

The current compiler for the MetaModelica language is based on OpenModelica compiler which was extended with code from the RML compiler (for meta-modeling/meta-programming facilities like pattern matching, unification, higher order functions, optimizations, etc). In the current version the meta-programming code can appear only in functions which can be called by Modelica code in the way an external function is called.

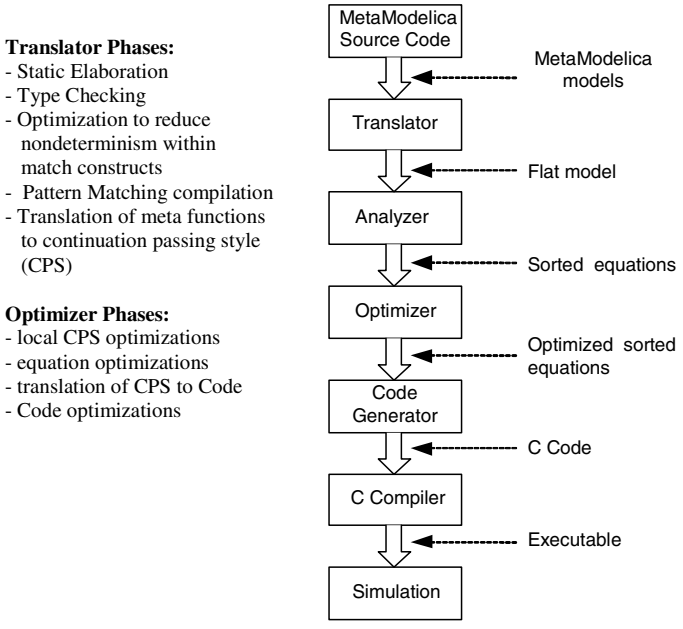

Fig. 5. The stages of translation and execution of a MetaModelica model

All variable values are boxed to be distinguished by the garbage collector. Every boxed value has a small integer as a header. Composite values are boxed structures. The structure header contains a small integer tag which is used for pattern matching. Logical variables are represented as boxed references. A different header is used to represent unbounded or bounded logical variables.

The MetaModelica source code is first translated into a so-called "*flat model*". This phase includes type checking, performing all object-oriented operations such as inheritance, modifications, compilation of pattern matching, translation of meta functions to continuation passing style. The flat model includes a set of equations declarations, functions and meta functions, with all object-oriented structure removed,

apart from the dot notation within the names. This process is called the "*partial instantiation*" of the model.

The next step is to solve the system of equations. First the equations need to be transformed into a suitable form for the numerical solvers. This is done by the symbolic and the numerical module of the compiler. The simulation code generator takes as input the flattened form of the equations. The equations are mapped into an internal data structure that permits simple symbolic manipulations such as: common subexpresions elimination, algebraic simplifications, constant folding, etc. These symbolic operations decrease substantially the complexity of the system of equations. After this stage the Block Lower Triagular form of the system of equations is computed.

Finally, in the last phase, the procedural code (in our implementation C code), is generated based on the previously computed BLT blocks when each block is linked to a numerical solver and the runtime for the meta functions. Within the C code the meta functions are called like normal functions.

5 Performance Evaluation of the MetaModelica Compiler

We are not aware of any language that is similar with the MetaModelica language. However, the meta-modeling and meta-programming parts of the MetaModelica language are close to a logic/functional language. Backtracking is used within the match construct to select the correct case and the specifications can contain logical variables. The uniontypes are similar with the SML datatype definitions, however MetaModelica functions have multiple inputs and outputs not just one argument like in SML. Also, because a reordering phase is applied to the MetaModelica code there is no need to explicitly declare mutually recursive types and functions.

All the information, the test code and the files needed to reproduce our results are available online at: **http://www.ida.liu.se/~adrpo/jmlc2006**. Please contact the authors for any additional information regarding the performance evaluation tests.

We have compared the execution speed of our generated code with SWI-Prolog 5.6.9 (SWI-Prolog [34]), SICStus Prolog 3.11.2 (Science [33]), Maude MSOS Tool (MMT) on top of Maude 2.1.1 (Illinois [15]). The Maude MSOS Tool (MMT) is an execution environment for Modular Structural Operational Semantics (MSOS) (Mosses 2004 [20]) specifications that brings the power of analysis available in the Maude system to MSOS specifications. The Maude MSOS Mini-Freja translation was implemented by Fabricio Chalub and Christiano Braga and is available as a case study together with sources from http://maude-msos-tool.sourceforge.net/. SWI-Prolog is a widely known open source implementation of Prolog. SICStus Prolog is a commercial Prolog implementation.

The closest match to the meta-modeling and meta-programming facilities of the MetaModelica compiler is the Maude MSOS Tool.

The test case is based on an executable specification of the Mini-Freja language (Pettersson 1999 [26]) running a test program based on the sieve of Eratosthenes. Mini-Freja is a call-by-name pure functional language. The test program calculates prime numbers.

The Prolog translation (`mf.pl`) was implemented by Mikael Pettersson and this author corrected a minor mistake.

Table 1. Execution time in seconds. The – sign represents out of memory.

	MetaModelica	SICStus	SWI	Maude MSOS Tool
8	0.00	0.05	0.00	2.92
10	0.00	0.10	0.03	5.60
30	0.02	1.42	1.79	226.77
40	0.06	3.48	3.879	-
50	0.13	-	11.339	-
100	1.25	-	-	-
200	16.32	-	-	-

The comparison was performed on a Fedora Core4 Linux machine with two AMD Athlon(TM) XP 1800+ processors at 1500 MHz and 1.5GB of memory.

The memory consumption was at peak 9Mb for MetaModelica and the others consumed the entire 1.5Gb of memory and aborted at arround 40 prime numbers. With this test we stressed only the meta-programming and meta-modeling part of the compiler. The Modelica part of the compiler was already able to handle huge models with thousands of equations.

6 Related Work

As related work we can consider the Unified Modeling Language (UML). Modeling in the UML sense has more emphasis on graphical notation for modeling rather than precise mathematical model definitions as in the modeling languages mentioned in the previous section. Initially, execution support was lacking, but during recent years code generators from UML2 has appeared. Also, during recent years, there has been an increased interest in model-driven developments and the OMG has launched model-driven architectures, primarily based on UML models. The idea of meta-modeling has attracted increased interest: a meta-model describes the structure of models at the next lower abstraction level. Meta-modeling and meta-programming allows transformations and composition of models and programs, which is becoming increasingly relevant in order to specify and manage complex industrial software and system applications. However, UML has developed into a rather heterogeneous collection of modeling notations. Also, precise mathematically defined semantics is not always available for these graphical notations. By contrast, MetaModelica is defined exclusively based on equations, functions and meta functions. Similar meta-programming facilities are present in functional languages like SML, Haskell and OCaml but the execution strategy is different in these languages as they do not support backtracking to select cases.

In the area of mathematical modeling the most important general de-facto standards for different dynamic simulation modes are:

- Continuous: Matlab/Simulink, MatrixX/SystemBuild, Scilab/Scicos for general systems, SPICE and its derivates for electrical circuits, ADAMS, DADS/Motion, SimPack for multi-body mechanical systems.
- Discrete: general-purpose simulators based on the discrete-event GPSS line, VHDL- and Verilog simulators in digital electronics, etc.
- Hybrid (discrete + continuous): Modelica/Dymola, AnyLogic, VHDL-AMS and Verilog-AMS simulators (not only for electronics but also for multi-physics problems).

The insufficient power and generality of the former modeling languages stimulated the development of Modelica (as a true object-oriented, multi-physics language) and VHDL-AMS/Verilog-AMS (multi-physics but strongly influenced by electronics).

The rapid increase in new requirements to handle the dynamics of highly complex, heterogeneous systems requires enhanced efforts in developing new language features (based on existing languages!). Especially the efficient simulation of hardware-software systems and model structural dynamics are yet unsolved problems. In electronics and telecommunications, therefore, the development of SystemC-AMS has been launched but these attempts are far from the multi-physics and multi-domain applications which are addressed by Modelica.

7 Conclusions and Future Work

We have presented two executable specification languages: RML for Natural Semantics specifications of programming languages, and Modelica for equation-based semantics and mathematical modeling of complex systems. We have also described MetaModelica as a unified mathematical and semantical modeling language by generalizing the concept of equation and introducing local equations and match expressions in the Modelica language. This gives interesting perspectives for the future regarding meta-modeling, model transformations and compositions during simulation, etc.

The current status of this work is that the OpenModelica compiler has been ported to the new unified Modelica modeling language, resulting in ~105000 lines of code expressed in the unified language. A compiler for MetaModelica has been completed at the time of this writing. We have also developed an integrated development environment based on Eclipse which facilitates writing and debugging of MetaModelica code (PELAB 2006 [23]). The MetaModelica language can be used to write semantic specifications for a broad spectrum of languages ranging from functional to imperative languages. We have also translated all our RML specification examples to MetaModelica in order to provide teaching material for the new language. The current specifications include imperative, functional, equation-based, and object-oriented languages.

The unified MetaModelica language gives new perspectives for a broad range of items, from programming and modeling languages to physical systems, but also including model transformations and composition. Apart from language specification to generate interpreters and compilers, symbolic differentiation rules for differentiating expressions and equations have been specified in MetaModelica and is in use.

Our near future plans are to extend MetaModelica with exceptions and reflection. The long term goal for MetaModelica is to achieve the generation of compilers for any language by drag and drop semantic components from libraries and connect them together in a similar way the physical systems are modeled today in Modelica.

References

[1] Patrik Borras, Dominique Clement, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. *CENTAUR: The System*. ed. P. Henderson, Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, February, 1988, vol. 24 of SIGPLAN, p.: 14-24

[2] Emil Carlsson, *Translating Natural Semantics to MetaModelica*, Department of Computer and Information Science. 2005, Linköping University, Linköping, Master's Thesis.

[3] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. *A Simple Applicative Language: Mini-ML*, Proceedings of the ACM Conference on Lisp and Functional Programming, August, 1986. also available as research report RR-529, INRIA, Sophia-Antipolis, May 1986.

[4] Thierry Despeyroux. *Executable Specification of Static Semantics*. ed. Gilles Kahn, Proceedings of Semantics of Data Types, 1984. Berlin, Germany, Springer-Verlag, Lecture Notes in Computer Science, vol. 173, p.: 215-233

[5] Thierry Despeyroux, *TYPOL: A Formalism to Implement Natural Semantics*, INRIA, Sofia-Antipolis, Report: RR 94, 1988, www: http://www.inria.fr/rrrt/rt-0094.html.

[6] Dynasim, *Dymola*, Last Accessed: 2005, www: http://www.dynasim.se/.

[7] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. *Modelica - A Language for Physical System Modeling, Visualization and Interaction*, Proceedings of IEEE Symposium on Computer-Aided Control System Design, August 22-27, 1999. Hawaii, USA

[8] Peter Fritzson, *Efficient Language Implementation by Natural Semantics*. 1998, http://www.ida.liu.se/~pelab/rml.

[9] Peter Fritzson and Vadim Engelson. *Modelica, a general Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation*, Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98), July 20-24, 1998. Brussels, Belgium

[10] Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, and Andreas Karström. *The Open Source Modelica Project*, Proceedings of Proceedings of The 2th International Modelica Conference, March 18-19, 2002. Munich, Germany

[11] Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, and Andreas Karström. *The Open Source Modelica Project*, Proceedings of the 2nd International Modelica Conference, March 18-19, 2002. Munich, Germany, Modelica Association, www: http://www.modelica.org/events/Conference2002/, Open Modelica System: http://www.ida.liu.se/~pelab/modelica/

[12] Peter Fritzson and Peter Bunus. *Modelica, a General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation*, Proceedings of 35th Annual Simulation Symposium, April 14-18, 2002. San Diego, California

[13] Peter Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. 2004, Wiley-IEEE Press. 940 pages, ISBN:0-471-471631, Book home page: http://www.mathcore.com/drmodelica.

[14] Sabine Glesner and Wolf Zimmermann, *Natural semantics as a static program analysis framework.* ACM Transactions on Programming Languages and Systems (TOPLAS), 2004. vol: 26(3), p.: 510-577.
[15] University of Illinois, *The Maude System Website*, Last Accessed, www: http://maude.cs.uiuc.edu/.
[16] Gilles Kahn, *Natural Semantics*, in Programming of Future Generation Computers, ed. Niva M. 1988, Elsevier Science Publishers, North Holland. p. 237-258.
[17] MathCore, *MathModelica*, Last Accessed: 2005, MathCore, www: http://www.mathcore.se/.
[18] Modelica-Association, *Modelica: A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification 2.2*, Last Accessed: 2005, www: http://www.modelica.org/.
[19] Modelica-Association, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 2.0*, Last Accessed: 2005, www: http://www.modelica.org/.
[20] Peter D. Mosses, *Modular structural operational semantics.* Journal of Functional Programming and Algebraic Programming. Special issue on SOS., 2004. vol: 60-61, p.: 195-228.
[21] PELAB, *Relational Meta-Language (RML) Environment*, Last Accessed: 2005, Programming Environments Laboratory (PELAB), www: http://www.ida.liu.se/~pelab/rml.
[22] PELAB, *Open Modelica System*, Last Accessed: 2005, Programming Environments Laboratory, www: http://www.ida.liu.se/~pelab/modelica.
[23] PELAB, *Modelica Development Tooling (MDT)*, Last Accessed: April, 2006, PELAB, www: http://www.ida.liu.se/~pelab/modelica/OpenModelica/MDT/.
[24] Mikael Pettersson and Peter Fritzson. *DML - A Meta-language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications*, Proceedings of the 1992 International Conference on Computer Languages, April 20-23, 1992. Oakland, California
[25] Mikael Pettersson, *Compiling Natural Semantics*, Department of Computer and Information Science. 1995, Linköping University, Linköping, PhD. Thesis.
[26] Mikael Pettersson, *Compiling Natural Semantics*. Lecture Notes in Computer Science (LNCS). Vol. 1549. 1999, Springer-Verlag.
[27] Gordon Plotkin, *A structural approach to operational semantics*, Århus University, Report: DAIMI FN-19, 1981
[28] Adrian Pop and Peter Fritzson. *Debugging Natural Semantics Specifications*, Proceedings of Sixth International Symposium on Automated and Analysis-Driven Debugging, September 19-21, 2005. Monterey, California
[29] Johan Ringström, Peter Fritzson, and Mikael Pettersson. *Generating an Efficient Compiler for a Data Parallel Language from Denotational Specifications*, Proceedings of Int. Conf. of Compiler Construction, April, 1994. Edinburgh, Springer Verlag, vol. LNCS 786
[30] Levon Saldamli, *PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations*, Department of Computer and Information Science. 2002, Linköping University, Linköping, Licenciate Thesis.
[31] Levon Saldamli, Peter Fritzson, and Bernhard Bachmann. *Extending Modelica for Partial Differential Equations*, Proceedings of 2nd International Modelica Conference, March. 18-29, 2002. Munich, Gernany
[32] Levon Saldamli, Bernhard Bachmann, Peter Fritzson, and Hansjürg Wiesmann. *A Framework for Describing and Solving PDE Models in Modelica*. ed. Gerhard Schmitz, Proceedings of 4th International Modelica Conference, 2005. Hamburg-Harburg, Modelica Association, www: http://www.modelica.org/events/Conference2005/

[33] SICS - Swedish Institute of Computer Science, *SICStus Prolog Website*, Last Accessed: April, 2006, www: http://www.sics.se/sicstus/.

[34] SWI-Prolog, *SWI-Prolog Website*, Last Accessed: April, 2006, University of Amsterdam, www: http://www.swi-prolog.org/.

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

[36] Lars Viklund, Johan Herber, and Peter Fritzson. *The implementation of ObjectMath - a hight-level programming enviornment for scientific computing*. ed. Uwe Kastens and Peter Pfahler, Proceedings of Compiler Construction - 4th International Conference (CC'92), 1992, Springer-Verlag, Lecture Notes in Computer Science (LNCS), vol. 641, p.: 312-318