Fritzson P., Aronsson P., Bunus P., Engelson V., Saldamli L., Johansson H., Karström A.:

The Open Source Modelica Project

2nd International Modelica Conference, Proceedings, pp. 297-306

Paper presented at the 2nd International Modelica Conference, March 18-19, 2002,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Oberpfaffenhofen, Germany.

All papers of this workshop can be downloaded from
http://www.Modelica.org/Conference2002/papers.shtml

Program Committee:

- Martin Otter, Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Robotik und Mechatronik, Oberpfaffenhofen, Germany (chairman of the program committee).
- Hilding Elmqvist, Dynasim AB, Lund, Sweden.
- Peter Fritzson, PELAB, Department of Computer and Information Science, Linköping University, Sweden.

Local organizers:
Martin Otter, Astrid Jaschinski, Christian Schweiger, Erika Woeller, Johann Bals,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Robotik und Mechatronik, Oberpfaffenhofen, Germany

The Open Source Modelica Project

Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson[1], Andreas Karström[1]

PELAB, Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, SE-581 83, Linköping, Sweden
{petbu,petfr}@ida.liu.se

[1]MathCore AB
Wallenbergs gata 4
SE-583 35 Linköping, Sweden
{henrik,andreas}@mathcore.com

Abstract

The open source software movement has received enormous attention in recent years. It is often characterized as a fundamentally new way to develop software. This paper describes an effort to develop an open source Modelica environment to a large extent based on a formal specification of Modelica, coordinated by PELAB, Department of Computer and Information Science, Linköping University, Sweden. The current version of the system provides an efficient interactive computational environment for most of the expression, algorithm, and function parts of the Modelica language as well as an almost complete static semantics for Modelica 2.0.

The longer-term goal is to provide reasonable simulation execution support, at least for less complex models, also for the equation part of Modelica which is the real essence of the language. People are invited to contribute to this open source project, e.g. to provide implementations of numerical algorithms as Modelica functions, add-on tools to the environment, or contributions to compiler itself. The source code of the tool components of the open source Modelica environment is available under the Gnu Public License, GPL. The library components are available under the same conditions as the standard Modelica library. The system currently runs under Microsoft Windows, Linux, and Sun Sparc Solaris. A benchmark example of running a simplex algorithm shows that the performance of the current system is close to the performance of handwritten C code for the same algorithm.

1 Introduction and Project Goals

The open source Modelica effort described in this paper has both short-term and long-term goals:

- The short-term goal is to develop an efficient interactive computational environment for most of the expression, algorithm, and function parts of the Modelica language, as well as a complete formal static semantic specification of the language. It turns out that with support of appropriate tools and libraries, Modelica is very well suited as a computational language for development and execution of both low level and high level numerical algorithms, e.g. for control system design, solving nonlinear equation systems, or to develop optimization algorithms that are applied to complex applications.
- The long-term goal is to have a rather complete implementation of the Modelica language, including simulation of equation based models and additional facilities in the programming environment, as well as convenient facilities for research and experimentation in language design or other research activities. However, our goal is not to reach the level of performance and quality provided by current commercial Modelica environments that can handle large models requiring advanced analysis and optimization by the Modelica compiler.

The long-term *research* related goals and issues of the open source implementation of a Modelica environment include but are not limited to the following:

- Development of a *complete formal specification* of Modelica, including both static and dynamic semantics. Such a specification can be used to assist current and future Modelica implementers by providing a semantic reference, as a kind of reference implementation.
- *Language design*, e.g. to further *extend the scope* of the language, e.g. for use in diagnosis, structural analysis, system identification, etc., as well as modeling problems that require partial differential equations.
- *Language design* to *improve abstract properties* such as expressiveness, orthogonality, declarativity, reuse, configurability, architectural properties, etc.
- *Improved implementation techniques*, e.g. to enhance the performance of compiled Modelica code by generating code for parallel hardware.

- *Improved debugging* support for equation based languages such as Modelica, to make them even easier to use.
- *Easy-to-use* specialized high-level (graphical) *user interfaces* for certain application domains.
- *Visualization* and animation techniques for interpretation and presentation of results.

The complete formal specification for Modelica is developed in *Natural Semantics*, which is currently the most popular and widely used semantics specification formalism. This specification is used as input for automatic generation of Modelica translator implementations being part of the open source Modelica environment, using the RML compiler generation tool developed at PELAB. The availability of a formal specification facilitates language design research on new language constructs to widen the scope of the language, as well as improving its abstract properties.

The open source Modelica environment thus provides a test bench for language design ideas that, if successful, can be submitted to the Modelica Association for consideration regarding possible inclusion in the official Modelica standard.

The current version of the open source Modelica environment allows most of the expression, algorithm, and function parts of Modelica to be executed interactively, and Modelica functions to be compiled into efficient C code. The generated C code is combined with a library of utility functions and a run-time library. An external function library interfacing a LAPACK subset and other basic algorithms is also under development.

2 The Open Source Modelica Environment

The interactive open source Modelica environment currently consists of the following components:

- *An interactive session handler*, that parses and interprets commands and Modelica expressions for evaluation. The session handler also contains simple history facilities, and completion of file names and certain identifiers in commands. It is based on the ReadLine library which is available in most Linux and Unix distributions.
- *A Modelica compiler*, translating Modelica to C code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries.
- *An execution and run-time module*. This module currently executes compiled binary code from translated expressions and functions. In the future it will also support simulation of equation based models, requiring numerical solvers as well as event handling facilities for the discrete and hybrid parts of the Modelica language.
- *A textual model editor*. Any text editor can be used. We have so far primarily employed Gnu Emacs, which has the advantage of being programmable for future extensions. A Gnu Emacs mode for Modelica and a Modelica syntax highlighting for the UltraEdit text editor has previously been developed. There is also a special Modelica editor that recognizes the syntax to some extent, and marks keywords and comments using different colors [Modelica]. Both the Emacs mode and the special editor hides Modelica graphical annotations during editing, which otherwise clutters the code and makes it hard to read.
- *A graphical model editor*. This is a graphical connection editor, for component based model design by connecting instances of Modelica classes. This part of the system is not yet implemented. A Java based prototype is however under development.

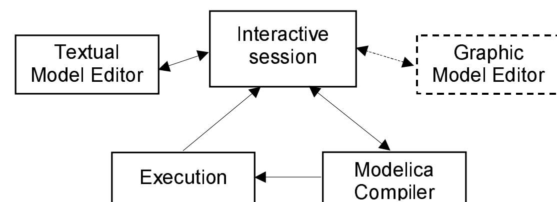

Figure 1. The architecture of the open source Modelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating expressions that are translated and executed. The graphic model editor is not yet implemented.

2.1 Interactive Session with Examples

The following is an interactive session with the open source Modelica environment including some commands and examples. First we start the system, which responds with a header line:

```
Open Source Modelica 0.1
```

We enter an assignment of a vector expression, created by the range construction expression 1:12, to be stored in the variable x. The type and the value of the expression is returned.

```
>> x := 1:12
  Integer[12]: {1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12}
```

The function `bubblesort` is called to sort this vector in descending order. The sorted result is returned together with its type. Note that the result vector is of type `Real[:]`, instantiated as `Real[12]`, since this is the declared type of the function result. The input `Integer` vector was automatically converted to a `Real` vector according to the Modelica type coercion rules.

```
>> bubblesort(x)
```

```
  Real[12]: {12, 11, 10, 9, 8, 7, 6, 5,
4, 3, 2, 1}
```

Now we want to try another small application, a simplex algorithm for optimization. First read in a small matrix containing coefficients that define a simplex problem to be solved:

```
>> a := read("simplex_in.txt")
  Real[6, 9]:
  {{-1,-1,-1, 0, 0, 0, 0, 0, 0},
   {-1, 1, 0, 1, 0, 0, 0, 0, 5},
   { 1, 4, 0, 0, 1, 0, 0, 0, 45},
   { 2, 1, 0, 0, 0, 1, 0, 0, 27},
   { 3,-4, 0, 0, 0, 0, 1, 0, 24},
   { 0, 0, 1, 0, 0, 0, 0, 1, 4}}
```

Then call the simplex algorithm implemented as the Modelica function `simplex1`. This function returns four results, which are represented as a tuple of four return values:

```
>> simplex1(a)
  Tuple: 4
  Real[8]: {9, 9, 4, 5, 0, 0, 33, 0}
  Real: 22
  Integer: 9
  Integer: 3
```

It is possible to compute an expression, e.g. `12:-1:1`, and store the result in a file using the `write` command:

```
>> write(12:-1:1,"test.dat")
```

We can read back the stored result from the file into a variable `y`:

```
>> y := read("test.dat")
  Integer[12]: {12, 11, 10, 9, 8, 7, 6,
5, 4, 3, 2, 1}
```

It is also possible to give operating system commands via the `system` utility function. A command is provided as a string argument. The example below shows `system` applied to the UNIX command `cat`, which here outputs the contents of the file `bubblesort.mo` to the output stream.

```
>> system("cat bubblesort.mo")
function bubblesort
  input Real[:] x;
  output Real[size(x,1)] y;
protected
  Real t;
algorithm
  y := x;
  for i in 1:size(x,1) loop
    for j in 1:size(x,1) loop
      if y[i] > y[j] then
            t := y[i];
            y[i] := y[j];
            y[j] := t;
      end if;
    end for;
  end for;
end bubblesort;
```

Another built-in command is `cd`, the *change current directory* command. The resulting current directory is returned as a string.

```
>> cd("..")
String: "/home/petfr/modelica"
```

3 The Modelica Translation Process

The Modelica translation process is depicted in Figure 2 below. The Modelica source code is first translated to a so-called flat model. This phase includes type checking, performing all object-oriented operations such as inheritance, modifications etc., and fixing package inclusion and lookup as well as import statements. The flat model includes a set of equations declarations and functions, with all object-oriented structure removed apart from dot notation within names. This process is a *partial instantiation* of the model, called *elaboration* in subsequent sections.

The next two phases, the equation analyzer and equation optimizer, are necessary for compiling models containing equations. These phases are currently missing, but will be supplied in some future version of the open source Modelica system. Finally, C code is generated which is fed through a C compiler to produce executable code. In the current preliminary version of the system this code cannot be any equation based simulation code, only code for computing expressions, algorithms, and functions.

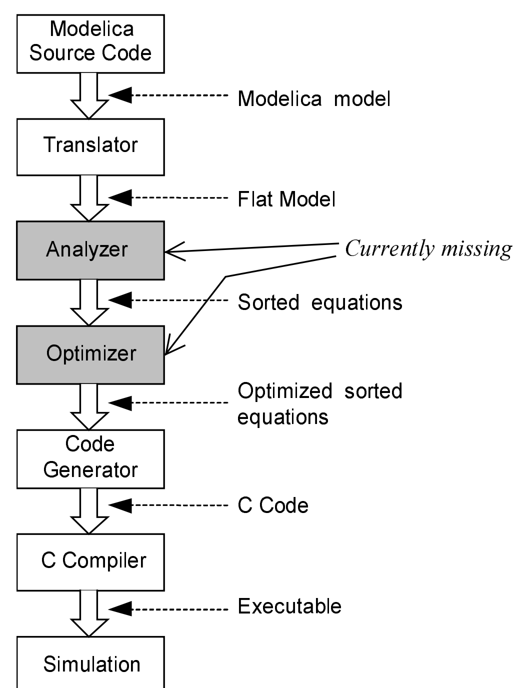

Figure 2. Translation stages from Modelica code to executing simulation. The current version of the open source Modelica compiler generates executable code only for functions and expressions since the equation analyzer and optimizer are still missing.

4 Modelica Static and Dynamic Semantics

The complete semantics, i.e. meaning, of the constructs in any programming language, also including the Modelica language, is usually divided into two major parts:

- static semantics
- dynamic semantics

The *static semantics* specifies compile time issues such as type checking, equivalence between different representations of the program such as a form with inheritance operations present, and the corresponding form with inheritance and modifications expanded, elaboration of the object-oriented model representation, conversion between different levels of the intermediate form, etc. Such a static semantics is currently given by our formal specification of Modelica.

It should be noted that the semantics specification for an equation based language such as Modelica differs from semantics specifications of ordinary programming languages, since Modelica equation based models are not programs in the usual sense. Instead Modelica is a modeling language used to specify relations between different objects in the modeled system.

The *dynamic semantics* specifies the run-time behavior including the equation solving process during the simulation, execution of algorithmic code, and additional run-time aspects of the execution process. We do not currently have a formal specification of the Modelica dynamic semantics, but intend to develop such a specification in the future. The current dynamic run-time behavior is implemented by hand in the C programming language

4.1 An Example Translation of Models into Equations.

As an example, the Modelica model `B` below is translated into equations.

```
model A
  Real x,y;
equation
  x = 2 * y;
end A;

model B
  A a;
  Real x[10];
equation
  x[5] = a.y;
end B;
```

The resulting equations appear as follows:

```
B.a.x = 2 * B.a.y
B.x[5] = B.a.y
```

The object-oriented structure is mostly lost which is why we talk about a flat set of equations, but the variable names in the equations still give a hint about their origin.

What the translator actually does is translating from the Modelica source code to an internal representation of the resulting set of equations. This internal representation is then further translated to C code. For functions, declarations, and interactively entered expressions, the C code can be generated fairly directly, whereas for equations several optimization stages must be applied before generating the C code.

4.2 Model Parameterization and Structural Parameters

In many cases model parameters in a Modelica model are unproblematic, and the translator can simply emit a flat model with the parameters still available as parameters that can be given different values during equation solution. But in some cases this is not so. We differentiate between two types of parameters, structural parameters, which affect the number and contents of the equations, and value parameters, that do not.

One simple example of a structural parameter is when a parameter is used in the size of an array. Consider the following model:

```
model ArrayEx
  parameter Real N = 3;
  Real a[N];
equation
  a[1] = 1;
  for i in 2:N loop
    a[i] = a[i-1] * 2;
  end for;
end ArrayEx;
```

This will, with the parameter `N` unmodified, produce the following equations:

```
a[1] = 1
a[2] = 2
a[3] = 4
```

However, if the parameter N is modified to be 5 when the model is elaborated, i.e. symbolically expanded, the set of equations will be different:

```
a[1] = 1
a[2] = 2
a[3] = 4
a[4] = 8
a[5] = 16
```

As the semantic specification specifies the semantics in terms of the generated equations, regarding algorithms and functions as special cases of equations, this means that the values of the parameters need to be determined at compile time to make it possible for the translator to do the translation.

Another, even more serious, complication with parameters is the combined use of parameters and connect statements. If a model contains e.g. a connect statements that looks like `connect(a[N],c)`, where a is an array of connectors, and `N` is a parameter, then the generated equations may look very different depending

on the value of N. Not only the number of equations may change, but the equations themselves can be altered.

5 Automatic Generation and Formal Specification of Translators

The implementation of compilers and interpreters for non-trivial languages is a complex and error prone process, if done by hand. Therefore, formalisms and generator tools have been developed that allow automatic generation of compilers and interpreters from formal specifications. This offers two major advantages:

- *High level descriptions* of language properties, rather than detailed programming of the translation process
- *High degree of correctness* of generated implementations. The high level specifications are more concise and easier to read than a detailed implementation in some programming language

The *syntax* of a programming language is conveniently described by a *grammar* in a common format, such as BNF, and the structure of the lexical entities, usually called *tokens*, are easily described by *regular expressions*. However, when it comes to the semantics of a programming language, things are not so straightforward.

The *semantics* of the language describes what any particular program written in the language "means", i.e. what should happen when the program is executed, or evaluated in some form.

Many programming language semantic definitions are given by a standard text written in English which tries to give a complete and unambiguous specification of the language. Unfortunately, this is often not enough. Natural language is inherently not very exact, and it is hard to make sure that all special cases are covered. One of the most important requirements of a language specification is that two different language implementers should be able to read the specification and make implementations that interpret the specification in the same way. This implies that the specification must be exact and unambiguous.

For this reason formalisms for writing formal specifications of languages have been invented (Pagan 1981). These formalisms allow for a mathematical semantic description, which as a consequence is exact, unambiguous and easier to verify. One of the most widely used formalisms for specifying semantics is called Natural Semantics (Kahn, 1987). A computer processable Natural Semantic specification language called RML, for Relational Meta Language, with a compiler generation tool rml2c (Pettersson, 1995), has previously been developed at PELAB, and is used for the formal specification of Modelica in the open source Modelica project.

5.1 Compiler Generation

Writing a good compiler is not easy. This makes automatic compiler generation from language specifications very interesting. If a language specification is written in a formal manner, all the information needed to build the compiler is already available in a machine-understandable form. Writing the formal semantics for a language can even be regarded as the high-level programming of a compiler.

In Figure 3 below, the different phases of the compilation process are shown. More specifically, it shows the translation process of a Modelica translator, which compiles, or rather translates, the Modelica source file. All the parts of the compiler can be specified in a formal manner, using different formalisms such as *regular expressions*, *BNF grammars*, and *Natural Semantics* specifications. At all stages, there is a tool to convert the formalism into an executable compiler module.

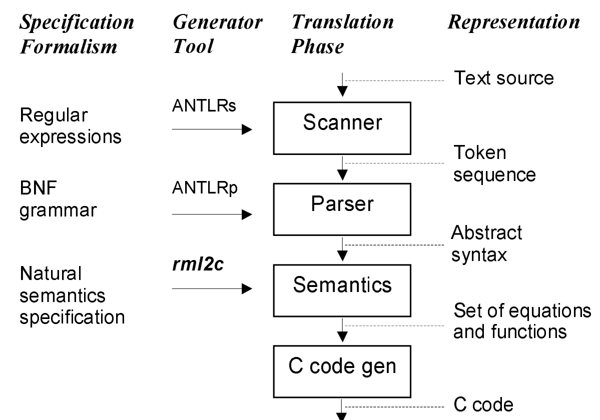

Figure 3. Phases in generating Modelica compiler modules from different kinds of specifications. The semantics module performs elaboration of the models including type checking and expansion of class structures, resulting in a set of equations, algorithms and functions.

6 Natural Semantics and RML

We have previously mentioned that a compiler generation system called RML is used to produce the Modelica translator in the open source project, from a Natural Semantics language specification. The generated translator is produced in ANSI C with a performance comparable to hand-written translators.

Below we give a short overview of the Natural Semantics formalism, and the corresponding RML specification language for expressing Natural Semantics in a computer processable way.

6.1 Natural Semantics

A Natural Semantics specification consists of a number of rules, similar to inference rules in formal logic. A rule has a structure consisting of clauses above a horizontal line followed by a clause under the line. A clause such as $a \Rightarrow b$ means a gives b.

$$\frac{c \Rightarrow b \wedge e \Rightarrow f}{a \Rightarrow b}$$

All clauses above the line are *premises*, and the clause below the line is the *consequence*. To prove the consequence, all the premises need to be proved. A small example of how rules typically appear is shown below, where a rule stating that the addition of a *Real* expression with an *Integer* expression gives a *Real* result.

$$\frac{\text{typeof}(e1) \Rightarrow \mathit{Real} \wedge \text{typeof}(e2) \Rightarrow \mathit{Integer}}{\text{elaborate}(e1 + e2) \Rightarrow \mathit{Real}}$$

6.2 RML

Modelica semantics is specified using the RML specification language (Pettersson 1995, 1999). The RML language is based on Natural Semantics, and uses features from languages like SML (Miller et. al. 1991) to allow for strong typing and datatype constructors. The RML source, e.g. a Modelica specifications, is compiled by an RML compiler (rml2c) to produce a translator for the described language. The RML compiler generates an efficient ANSI C program that is subsequently compiled by an ordinary C compiler, e.g. producing an executable Modelica translator. The RML tool has also been used to produce compilers for Java, Pascal and a few other languages.

6.2.1 Correspondence between Natural Semantics and RML.

The correspondence between Natural Semantics and RML is perhaps easiest to understand using an example. The following three Natural Semantics rules specifies the types of expressions containing addition operators and `Boolean` constants:

$$\frac{\text{typeof}(e1) \Rightarrow \mathit{Real} \wedge \text{typeof}(e2) \Rightarrow \mathit{Real}}{\text{typeof}(e1 + e2) \Rightarrow \mathit{Real}}$$

$$\frac{\text{typeof}(e1) \Rightarrow \mathit{Integerl} \wedge \text{typeof}(e2) \Rightarrow \mathit{Integer}}{\text{typeof}(e1 + e2) \Rightarrow \mathit{Integer}}$$

$$\frac{}{\text{typeof}(\text{FALSE}) \Rightarrow \mathit{Boolean}}$$

The corresponding three RML rules are collected in a relation called `typeof`, which maps expressions to types:

```
relation typeof :: Exp => Type =
  ...
  rule  typeof(e1) => Real & typeof(e2) => Real
        ------------------------------------
        typeof(ADD(e1,e2)) => Real

  rule  typeof(e1) => Integer & typeof(e2) => Integer
        -----------------------------------------
        typeof(ADD(e1,e2)) => Integer

        axiom   typeof(FALSE) => Boolean
end
```

6.2.2 A Simple Interpretive Semantics

As a simple example of the RML syntax, we show a small expression evaluator, which "translates" a mathematical expression to the value of the expression. First, an expression data type is declared:

```
datatype Exp = NUMBER of real
             | ADD of Exp * Exp
             | MUL of Exp * Exp
```

An expression is either a number, a sum of two other expressions, or a product of two other expressions. As an example, the expression is represented as the value.

```
MUL(NUMBER(3),ADD(NUMBER(4),NUMBER(5)))
```

Then we describe the relation `eval`. The `eval` relation relates expressions to their evaluated values, so that `eval(x)=>y` always holds if `y` is the value resulting from the evaluation of `x`.

```
relation eval =
  axiom eval(NUMBER(x)) => x

  rule  eval(x) => x2 & eval(y) => y2 &
        real_add(x2,y2) => sum
        ---------------------------
        eval(ADD(x,y)) => sum

  rule  eval(x) => x2 & eval(y) => y2 &
        real_mul(x2,y2) => prod
        ---------------------------
        eval(MUL(x,y)) => prod
end
```

The first rule is an axiom, which means that the set of premises for the rule is empty. It could also have been written as a rule with nothing above the line:

```
rule -------------------
       eval(NUMBER(x)) => x
```

The second rule tells how to evaluate sums of two expressions. What the rule says is "if `x` evaluates to `x2`, `y` evaluates to `y2`, and the sum of `x2` and `y2` is `sum`, then the result of evaluating `ADD(x,y)` is `sum`." The relations `real_add` and `real_mul` are predefined in RML.

When an RML specification is executed, a relation named `main` is evaluated at the top level. If our program has a `main` relation that looks like the following, the program would simply print the number 27 and exit.

```
relation main =
  rule  eval(MUL(NUMBER(3),
               ADD(NUMBER(4),
                  NUMBER(5)))) => x &
        real_string(x) => xs &
        print(cs)
        ----------------------
        main(_)
end
```

The RML language has some obvious similarities with functional programming languages and logic programming languages. While resolving, or “proving” the premises in a rule, a simple resolution process is carried out which tries to find a rule in the relation which matches the arguments, and if it fails, a simple retry mechanism is used to find other possible solutions.

If none of the rules in a relation is possible to prove, the relation fails, but there is also the possibility to introduce a rule that explicitly fails, by using the keyword `fail` in the corresponding clause.

There are a few other syntactic features of RML that should be known to the reader. If an argument is not used in a rule, it can be written as an underscore, which means that it matches anything, but is not used. If a relation has the right-hand side (result) type `()`, it can be omitted together with the `=>` symbol. The `main` relation above shows an example of both of these features.

7 The Formal Specification of Modelica

The specification is separated into a number of modules, to separate different stages of the translation, and to make it more manageable. This section will briefly cover some of the most important parts of the specification. In all, the specification contains several thousand lines of RML, but it should be kept in mind that the RML code is rather sparse, with many empty or short lines.

The top level relation in the semantics is called `main`, and appears as follows:

```
relation main =

  rule  Parser.parse f => p &
        SCode.elaborate(p) => p2 &
        Inst.elaborate(p2) => d &
        DAE.dump d &
        ----------------
        main([f])
end
```

7.1 Parsing and Abstract Syntax

The relation `Parser.parse` is actually written in C, and calls the parser generated from a grammar by the ANTLR parser generator tool (ANTLR 1998). This parser builds an abstract syntax tree (AST) from the source file, using the AST data types in a RML module called `Absyn`. The parsing stage is not really part of the semantic description, but is of course necessary to build a real translator.

7.2 Rewriting the AST

The AST closely corresponds to the parse tree and keeps the structure of the source file. This has several disadvantages when it comes to translating the program, and especially if the translation rules should be easy to read for a human. For this reason a preparatory translation pass is introduced which translates the AST into an intermediate form, called `SCode`. Besides some minor simplifications the `SCode` structure differs from the AST in the following respects:

- All variables are described separately. In the source and in the AST several variables in a class definition can be declared at once, as in `Real x, y[17];`. In the `SCode` this is represented as two unrelated declarations, as if it had been written `Real x; Real y[17];`.
- Class declaration sections. In a Modelica class declaration the public, protected, equation and algorithm sections may be included in any number and in any order, with an implicit public section first. In the `SCode` these sections are collected so that all public and protected sections are combined into one section, while keeping the order of the elements. The information about which elements were in a protected section is stored with the element itself.

One might have thought that more work could be done at this stage, like analyzing expression types and resolving names. But due to the nature of the Modelica language, the only way to know anything about how the names will be resolved during elaboration is to do a more or less full elaboration. It is possible to analyze a class declaration and find out what the parts of the declaration would mean if the class was to be elaborated as-is, but since it is possible to modify much of the class while elaborating it that analysis would not be of much use.

7.3 Elaboration and Instantiation

- To be executed, classes in a model need to be instantiated, i.e. data objects are created according to the class declaration. There are two phases of instantiation:
- The symbolic, or compile time, phase of instantiation is usually called *elaboration*. No data objects are created during this phase. Instead the

symbolic internal representation of the model to be executed/simulated is transformed, by performing inheritance operations, modification operations, aggregation operations, etc.

- The creation of the data object, usually called *instantiation* in ordinary object-oriented terminology. This can be done either at compile time or at run-time depending on the circumstances and choice of implementation.

The central part of the translation is the *elaboration* of the model. The convention is that the last model in the source file is elaborated, which means that the equations in that model declaration, and all its subcomponents, are computed and collected.

The elaboration of a class is done by looking at the class definition, elaborating all subcomponents and collecting all equations, functions, and algorithms. To accomplish this, the translator needs to keep track of the class context. The context includes the lexical scope of the class definition. This constitutes the *environment* which includes the variables and classes declared previously in the same scope as the current class, and its parent scope, and all enclosing scopes. The other part of the context is the current set of modifiers which modify things like parameter values or redeclare subcomponents.

```
model M
  constant Real c = 5;
  model Foo
    parameter Real p = 3;
    Real x;
  equation
    x = p * sin(time) + c;
  end Foo;

  Foo f(p = 17);
end M;
```

In the example above, elaborating the model `M` means elaborating its subcomponent f, which is of type `Foo`. While elaborating `f` the current environment is the parent environment, which includes the constant `c`. The current set of modifications is (`p = 17`), which means that the parameter `p` in the component `f` will be 17 rather than 3.

There are many semantic rules that takes care of this, but only a few are shown below. They are also somewhat simplified to focus on the central aspects.

7.4 The elab_class and elab_element Relations

The relation `elab_class` elaborates a class. It takes five arguments, the environment `env`, the set of modifications `mod`, the prefix `pre` which is used to build a globally unique name of the component in a hierarchical fashion, a collection of connection sets `csets`, and the class definition `c`. It opens a new scope in the environment where all the names in this class will be stored, and then uses a relation called `elab_class_in` to do most of the work. Finally it generates equations from the connection sets collected while elaborating this class. The “result” of the relation are the *elaborated* equations and some information about what was in the class. In the case of a function, regarded as a restricted class, the result is an algorithm section.

One of the most important relations is `elab_element`, that elaborates an element of a class. An element can typically be a class definition, a variable or constant declaration, or an extends clause. Below is shown *only* the rule for elaborating variable declarations

```
relation elab_class: (Env, Mod, Prefix, Connect.Sets, Scode.Class) =>
          (DAE.Element list, Connect.Sets, Types.Type) =

                  rule Env.open_scope(env) => env’ &
  elab_class_in(env’,mod,pre,csets,c) => (dae1,_,csets’,ci_state’, tys) &
    Connect.equations csets’ => dae2 & list_append(dae1, dae2) => dae &
                    mktype(ci_state’,tys) => ty
   ------------------------------------------------------------------
   elab_class(env,mod,pre,csets,c as SCode.CLASS(n,_,r,_)) => (dae, [], ty)
                              end
```

```
relation elab_element: (Env, Mod, Prefix, Connect.Sets, Scode.Element) =>
(DAE.Element list, Env, Connect.Sets, Types.Var list) =
...
rule Prefix.prefix_cref(pre,Exp.CREF_IDENT(n,[]))  => vn &
     Lookup.lookup_class(env,t) => (cl,classmod) &  Find the class definition
     Mod.lookup_modification(mods,n)=>mm &
     Mod.merge(classmod,mm) => mod &                 Merge the modifications
     Mod.merge(mod,m) => mod' &
     Prefix.prefix_add(n,[],pre) => pre' &           Extend the prefix
     elab_class(env,mod',pre',csets,cl)              Elaborate the variable
          => (dae1,csets',ty,st) &
     Mod.mod_equation mod' => eq &     If the variable is declared with a default equation,
     make_binding (env,attr,eq,cl)    add it to the environment with the variable.
          => binding &
     Env.extend_frame_v(env,          Add the variable binding to the environment
          Env.FRAMEVAR(n,attr,ty,binding))
          => env' &
     elab_mod_equation(env,pre,n,mod')               Fetch the equation, if supplied
          => dae2 &
     list_append(dae1, dae2) => dae                  Concatenate the equation lists
     -------------------------
     elab_element(env,mods,pre,csets,
                       SCode.COMPONENT(n,final,prot,attr,t,m))
          => (dae, env',csets',[(n,attr,ty)])
...
end
```

7.5 Output

The equations, functions, and variables found during elaboration are collected in a list of objects of type `DAEcomp`:

```
datatype DAEcomp = VAR of
Exp.ComponentRef * VarKind
                | EQUATION of Exp * Exp
                  ...
```

As the final stage of translation, in the current version of the translator, functions and expressions in this list are converted to C code.

8 A Small Benchmark

We have evaluated the current implementation of the open source Modelica compiler on a small benchmark consisting of solving a simplex optimization problem consisting of 25 variables and 5 conditions. The measured execution time was averaged over 100 executions on a Sun UltraSparcstation 10. The Modelica code of the simplex algorithm is available in Appendix A. We found that the execution time for the Modelica code was a factor of 1.54 slower than handwritten C code for the same algorithm. This is a preliminary result obtained at the time of writing this paper, and we expect to be able to get even closer to C code execution performance by tuning the implementation.

9 Conclusions

We have developed the first version of an open source Modelica environment, to a large extent based on a Modelica compiler automatically generated from a formal Natural Semantics specification of Modelica. This formal specification is intended to become a reference specification for research purposes and for future Modelica implementers.

An important short-term goal for this open source project is to provide an interactive and efficient computational environment for using Modelica as a high level strongly typed programming language for computational applications.

In the longer-term perspective we expect to extend the system to also provide simulation of equation based models in Modelica, however with lower performance and handling models of less complexity than what is currently managed by commercial implementation such as Dymola and MathModelica.

Appendix – The Simplex Algorithm

```
function pivot1
  input Real b[:,:];
  input Integer p;
  input Integer q;
  output Real a[size(b,1),size(b,2)];
protected
  Integer M;
  Integer N;
algorithm
  a := b;
  N := size(a,1)-1;
  M := size(a,2)-1;
  for j in 0:N loop
    for k in 0:M loop
      if j<>p and k<>q then
       a[j+1,k+1] := a[j+1,k+1]-
a[p+1,k+1]*a[j+1,q+1]/a[p+1,q+1];
      end if;
    end for;
  end for;
  for j in 0:N loop
    if j<>p then
      a[j+1,q+1] := 0;
    end if;
  end for;
  for k in 0:M loop
    if k<>q then

a[p+1,k+1]:=a[p+1,k+1]/a[p+1,q+1];
    end if;
  end for;
  a[p+1,q+1] := 1;
end pivot1;

function simplex1
  input Real matr[:,:];
  output Real x[size(matr,2)-1];
  output Real z;
  output  Integer q;
  output  Integer p;
protected
  Real a[size(matr,1),size(matr,2)];
  Integer M;
  Integer N;
algorithm
  N := size(a,1)-1;
  M := size(a,2)-1;
  a := matr;
  p:=0; q:=0;
  while not (q==(M+1) or p==(N+1)) loop
    q := 0;
    while not (q == (M+1) or
a[0+1,q+1]<0) loop
      q:=q+1;
    end while;
    p := 0;
    while not (p == (N+1) or
a[p+1,q+1]>0) loop
      p:=p+1;
    end while;
    for i in p+1:N loop
      if a[i+1,q+1] > 0 then
        if (a[i+1,M+1]/a[i+1,q+1]) <
(a[p+1,M+1]/a[p+1,q+1]) then
            p := i;
        end if;
      end if;
    end for;
        if (q < M+1) and (p < N+1) then
      a := pivot1(a,p,q);
    end if;
  end while;
  for i in 1:M loop
    x[i] := -1;
    for j in 1:N+1 loop
      if (x[i] < 0) and ((a[j,i] >=
1.0) and (a[j,i] <= 1.0)) then
          x[i] := a[j,M+1];
      elseif ((a[j,i] < 0) or (a[j,i] >
0)) then
          x[i] := 0;
      end if;
    end for;
  end for;
  z := a[1,M+1];
end simplex1;
```

References

Abadi Martin and Cardelli Luca. A Theory of Objects. Springer Verlag, ISBN 0-387-94775-2, 1996.

Elmqvist Hilding, Dag Brück, and Martin Otter, Dymola - User's Manual. Dynasim AB, Research Park Ideon, Lund, Sweden, 1996

Kågedal, D., Fritzson, P. Generating a Modelica Compiler from Natural Semantics Specifications. Summer Computer Simulation Conference '98, Reno, Nevada, USA, July 19-22, 1998.

Modelica Home Page http://www.Modelica.org

Dymola Home Page: http://www.Dynasim.se

MathModelica Home Page: http://www.MathCore.com

Mikael Pettersson. Compiling Natural Semantics, Linköping Studies in Science and Technology. Dissertation No. 413, 1995.

Mikael Pettersson. Compiling Natural Semantics. LNCS 1549, Springer-Verlag, 1999.

Miller Robin, Tofte Mads, Harper Robert. *Commentary on Standard ML*. The MIT Press, 1991.

ANTLR home page: http://www.ANTLR.org/

Frank Pagan. *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, ISBN 0-13-329052-2, 1981.

Gilles Kahn. Natural Semantics. In *Proc. of the Symposium on Theoretical Aspects on Computer Science, STACS'87*, LNCS 247, pp 22-39. Springer-Verlag, 1987.