

GENERATING A MODELICA COMPILER FROM NATURAL SEMANTICS SPECIFICATIONS

David Kågedal and Peter Fritzson

PELAB, Dept. of Computer and Information Science
Linköping University, S-58183, Linköping, Sweden

davidk@lysator.liu.se
petfr@ida.liu.se

KEYWORDS

Modelica, RML, formal semantics, simulation

ABSTRACT

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

Modelica is an object-oriented language for modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages.

A Modelica model is defined in terms of classes containing equations and definitions. The semantics, i.e. the meaning of such a model is defined via translation of classes, instances and connections into a flat set of constants, variables and equations.

This paper describes and defines a formal semantics for Modelica expressed in a high-level specification notation called natural semantics.

A compiler generation system called RML produces a Modelica translator from such a language specification. The generated translator is produced in ANSI C and has comparable performance as hand-written translators.

The RML tool has also been used to produce compilers for Java, Pascal and few other languages.

MODELICA OVERVIEW

Modelica (Modelica 1998) is an object-oriented language for modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages.

Compared with the widespread simulation languages available today this language offers three important advances: 1) non-causal modeling based on differential and algebraic equations; 2) multidomain modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic etc. model components within the same application model; 3) a general type system that unifies object-orientation, multiple inheritance, and templates within a single class construct.

A Modelica model is defined in terms of classes containing equations and definitions. The semantics, i.e. the meaning of such a model is defined via translation of classes, instances, connections and functions into a flat set of constants, variables and equations. Equations are sorted and converted to assignment statements when possible. Strongly connected sets of equations are solved by calling a symbolic and/or numeric solver.

Modelica View of Object-orientation

Traditional object-oriented languages like C++, Java and Simula support programming with operations on state. The state of the program includes variable values and object data, and the number of objects may change dynamically. The Modelica approach is different. The Modelica language emphasizes *structured* mathematical modeling and uses the structural benefits of object orientation. A Modelica model is primarily a declarative mathematical description, which allows analysis and equational reasoning. For these reasons, dynamic object creation at runtime is usually not interesting from a mathematical modeling point of view, and is currently not supported by the Modelica language.

For other reasons, and to compensate this missing feature *arrays* are provided by Modelica. An array is an indexed set of objects of equal type. The size of the set is determined once at runtime. This construct for example can be used to represent a set of similar rollers in a bearing, or a set of electrons around an atomic nucleus.

Object-Oriented Mathematical Modeling

Mathematical models used for analysis in scientific computing are inherently complex in the same way as other software. One way to handle this complexity is to use object-oriented techniques.

However, there are some fundamental differences between object-oriented *programming* and object-oriented *mathematical modeling*, where a class description may consist of a set of equations, which implicitly define the behavior of some class of physical objects or the relationships between objects. Functions should be side-effect free and are regarded as mathematical functions rather than operations on objects. Explicit operations on state may be completely absent, but may also be present. Also, causality, i.e. which variables are regarded as input, and which are regarded as output, is usually not defined by such an equation-based model.

There are usually many choices of causality, but one must be selected prior to solving a system of equations. If a system of such equations is solved symbolically, the equations are transformed into a form where some (state) variables are explicitly defined in terms of other (state) variables. If the solution process is numeric, it will compute new state variables from old variable values, and thus operate on the state variables.

Modelica Fundamentals

Modelica models are built from *classes*. Like in other object-oriented languages, a class contains variables, i.e. class attributes representing data. The main difference compared to traditional object-oriented languages is that instead of functions (methods) we use *equations* to specify behavior. Equations can be written explicitly, like $a=b$, or be inherited from other classes. Equations can also be specified by the **connect** statement. The statement **connect** ($v1$, $v2$) expresses coupling between variables $v1$ and $v2$. These variables are called *connectors* and belong to the connected objects. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using Modelica.

In the following sections we introduce some basic and distinctive syntactical and semantic features of Modelica, such as connectors, encapsulation of equations, inheritance, declaration of parameters and constants, powerful parametrization capabilities.

The Modelica Notion of Subtypes

The notion of subtyping in Modelica is influenced by the type theory of Abadi and Cardelli (Abadi and Cardelli 1996). The notion of inheritance in Modelica is separated from the

notion of subtyping. According to the definition, a class A is a *subtype* of class B if class A contains all the public variables declared in the class B, and types of these variables are subtypes of the types of corresponding variables in B. The main benefit of this definition is additional flexibility in the composition of types. For instance, the class `TempResistor` is a subtype of `Resistor`.

```
class Resistor
  extends TwoPin
  parameter Real R;
equation
  v=R*i;
end Resistor;

class TempResistor
  extends TwoPin
  parameter Real R, RT, Tref ;
  Real T;
equation
  v=i*(R+RT*(T-Tref));
end TempResistor
```

Subtyping is used for example in class *instantiation*, *re-declarations* and function calls. If variable a is of type A, and A is a subtype of B, then a can be initialized by a variable of type B.

Note that `TempResistor` does not inherit the `Resistor` class. There are different equations for evaluation of v . If equations would be inherited from `Resistor` then the set of equations will become inconsistent in `TempResistor`, since Modelica currently does not support named equations and replacement of equations. For example, the specialized equation below from `TempResistor`:

$$v=i*(R+RT*(T-Tref))$$

and the general equation from class `Resistor`

$$v=R*i$$

are inconsistent.

Class Parametrization

A distinctive feature of object-oriented programming languages and environments is the ability to fetch classes from standard libraries to be reused for particular needs. Such reuse should be done without modification of the library codes. The two main mechanisms for reuse are:

- *Inheritance*. This is essentially “copying” class definition and adding additional elements (variables, equations and functions).

- *Class parametrization.* (Also called generic classes or types) This replaces a generic type identifier in the whole class definition by an actual type.

Modelica contains a way to control class parametrization. Assume that a library class is defined as

```
class SimpleCircuit
  Resistor R1(R=100), R2(R=200), R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

Assume that in our particular application we would like to reuse the definition of `SimpleCircuit`: we want to use the parameter values given for `R1.R` and `R2.R` and the circuit topology, but exchange `Resistor` with the temperature-dependent resistor model, `TempResistor`, discussed above.

This can be accomplished by redeclaring `R1` and `R2` as follows.

```
class RefinedSimpleCircuit = SimpleCircuit(
  redeclare TempResistor R1,
  redeclare TempResistor R2);
```

Since `TempResistor` is a subtype of `Resistor`, it is possible to replace the ideal resistor model. Values of the additional parameters of `TempResistor` can be added in the redeclaration:

```
redeclare TempResistor R1(RT=0.1, Tref=20.0)
```

This is a major modification, however it should be noted that all equations that could be defined in `SimpleCircuit` are still valid.

FORMAL SEMANTICS

Language Specification

Any new programming language needs to be described in one way or another. The syntax of the language is conveniently described by a grammar in a common format, such as BNF. 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.

Most 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 implementors 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.

The formalism used here to describe the Modelica semantics is called Natural semantics, and belongs to the class of operational semantic formalisms.

A natural semantics specification consists of a number of rules, similar to inference rules in formal logic. A rule has the following structure¹:

$$\frac{p_1 \Rightarrow q_1 \wedge \dots \wedge p_n \Rightarrow q_n}{p \Rightarrow q}$$

Each clause $p \Rightarrow q$ is called a *sequent*, and the sequents above the line are *premises*, and the sequent below the line is the *consequence*. To prove the consequence, all the premises need to be proved. A small example of how this typically looks is shown in figure 1, where the interpretive semantics of if-expressions in a normal functional programming language is described. If the condition c evaluates to

$$\frac{c \Rightarrow TRUE \wedge t \Rightarrow x}{IF(c, t, f) \Rightarrow x}$$

$$\frac{c \Rightarrow FALSE \wedge f \Rightarrow x}{IF(c, t, f) \Rightarrow x}$$

Fig. 1. Semantics of an if-statement

true, and the true-branch evaluates to something (here represented by variable x), then the whole expression evaluates to the value of x . If the condition does not evaluate to true, then it is not possible to prove all the premises of the first rule. On the other hand, if the condition evaluates to false, and the false branch evaluates to something, then the whole expression evaluates to that.

Compiler Generation

Writing a good compiler is not easy, and making sure that it

1. The description of natural semantics here is simplified compared to many other presentations.

does the right thing in all possible cases is even harder. This makes automatic compiler generation from the language specification very interesting. If the 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 2, 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 into an equation sequence. Combining the natural semantics with other formalisms for describing the syntax, means that all the parts of the compiler can be specified in a formal manner, with tools to generate the C code, or whatever, to build the actual compiler. At all stages, there is a tool to convert the formalism into a computer program.

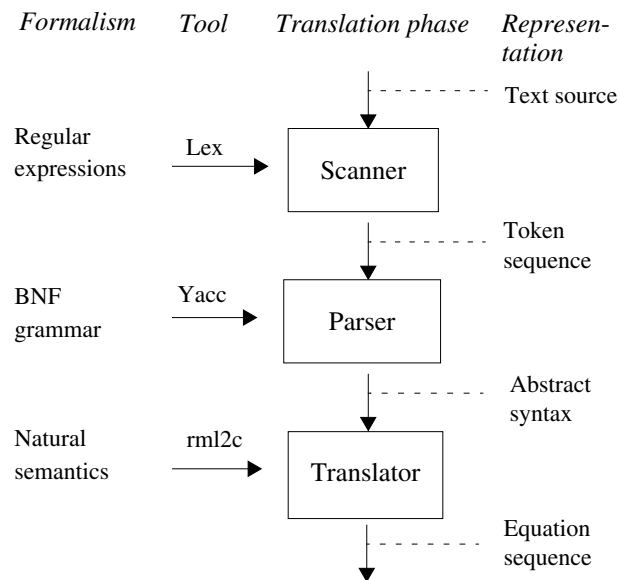


Fig. 2. Translation phases and corresponding formalisms.

RML

The language that our Modelica semantics is written in is a language called RML (Pettersson 1995), developed by Mikael Pettersson. The RML language is based on natural semantics, and uses features from languages like SML to allow for strong typing and datatype constructors. The RML source, e.g. a Modelica specifications, is compiled by an RML compiler 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.

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 $3 \cdot (4 + 5)$ 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* $x \Rightarrow y$ always holds if y is the value resulting from the evaluation of x .

```
relation eval =
  axiom eval (NUMBER (x) ) => x
  rule  eval (x) => x' & eval (y) => y' &
        real_add (x' , y' ) => sum
        -----
        eval (ADD (x , y) ) => sum
  rule  eval (x) => x' & eval (y) => y' &
        real_mul (x' , y' ) => 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

```
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 x' , y evaluates to y' , and the sum of x' , and y' is sum , then the result of evaluating $ADD(x, y)$ is sum .” The relations *real_add* and *real_mul* are predefined in RML.

When a RML specification is executed, a relation named *main* is evaluated at the top level. If our program had a *main* relation that looked 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 backtracking 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 consequent.

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.

OVERVIEW OF THE MODELICA SEMANTICS

The purpose of the semantic specification of Modelica is to describe how the object oriented structuring of models and their equations work. It is not intended to describe the equation solving process, nor the actual simulation.

Static semantics

The part of the language we describe is what is usually called the static semantics of the language. However, it should be noted that writing a semantic specification for an equation-based language such as Modelica differs from describing many other languages, as the Modelica code is not a program in any usual sense. Instead Modelica is a modelling language used to specify the relations between different objects in a modeled system. The program that will be run by the user is not only based on the Modelica model but rather a simulation package that uses the Modelica description of the model.

Dynamic Semantics

The dynamic semantics, which in the case of Modelica means the simulation-time behaviour, is not covered by this specification. The language specification still needs to specify the dynamic semantics, but a formal specification will be

a little tricky to do, as the algorithmic parts of a Modelica model is run from within a simulation environment which the formal semantics cannot describe, except in general terms. Such specification is possible, but is further work.

Translation

The simulation of a model involves some preparatory stages, and the diagram in figure 3 shows how a typical simulation

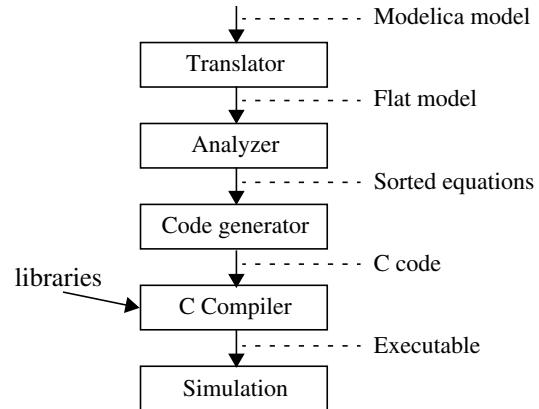


Fig. 3. Simulation preparation.

environment works. It is the first stage we are interested in, as our semantic specification describes the static properties of the model.

The semantic specification of Modelica is expressed as a translator from a Modelica source file to a flat list of variables and equations. The algorithms in the Modelica source are also included, but are treated separately, and are not really handled by the semantic specification as they are part of the runtime (simulation-time) semantics.

As an example, the Modelica model in figure 4 is

```

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;

```

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

Fig. 4. A Modelica model

Fig. 5. Resulting Equations

translated to the equations in figure 5. The structure is mostly

lost, but the variable names in the output give an hint about their origin.

What the translator actually does is that it translates from the Modelica source code to an internal representation of the resulting set of equations. This internal representation is then emitted in textual form. The exact format of the target language is thus simple to change to accommodate for different needs. The current implementation exports something that looks like one huge Modelica model, but with some syntactic modifications.

Connections

Connections between objects in a Modelica model are typically introduced by a `connect` statement. A class in a Modelica model can specify how it can be connected to other objects by providing interface components, called *connectors* in Modelica.

A connection is not necessarily point-to-point. Several connectors can be connected together by simply using the same connector in several connect statements.

The semantics of a connect statement is defined by the equations that it generates. The process of generating equations from connect statements begins with grouping the connectors into connected clusters. If connector A is connected to connector B and connector B is connected to connector C, the connectors A, B and C form such a cluster. A connector is an instance of a class, like any other object in Modelica. The class specifies what component variables are in the connectors, and also how they should be used in the generated connection equations. For normal variables, the translator simply generates equations that equates the corresponding components of all the connected components in a cluster.

For connector components declared with the `flow` type prefix a sum-to-zero equation $\sum v_i = 0$ where the values of that component in all the connectors sum to zero, as in Kirchhoff's law. A typical example is electrical circuits, where the electrical components have pins which are connected. Connecting two electrical components affects the voltage and current of the pin, and the pin connector is declared in the following fashion:

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

The convention is accepted that positive current always flows into the electrical component, which means that if a number of pins are connected, the sum of the currents in all the pins equals zero, as the positive (inbound) current must be equal to the negative (outbound) current. The voltage, on

the other hand, is equal in all the connected pins.

Parameterization

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

But if the parameter N is modified to be 5 when the model is instantiated, 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, this means that the value of the parameters need to be determined 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.

THE SPECIFICATION

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 cover 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, so that it is easier on human eyes.

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

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

Parsing and Abstract Syntax

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

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 components are described separately.* In the source and in the AST several components 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 instantiation is to do a more or less full instantiation. 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 instantiated as-is, but since it is possible to modify much of the class while instantiating it that analysis would not be of much use.

Instantiation

The central part of the translation is the instantiation of the model. The convention is that the last model in the source file is instantiated, which means that the equations in that model declaration, and all its subcomponents, are calculated and collected.

The instantiation of a class is done by looking at the class definition, instantiating all subcomponents and collecting all equations. 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, instantiating the model `M` means instantiating its subcomponent `f`, which is of type `Foo`. While instantiating `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.

```

relation inst_class =

  rule Env.open_scope(env) => env' &
    inst_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
    -----
    inst_class(env, mod, pre, csets,
              c as SCode.CLASS(n, _, r, _))
    => (dae, [], ty)

end

```

Fig. 6. The `inst_class` relation

The relation `inst_class` (figure 6) instantiates a class. It takes five arguments, the environment (`env`), the set of modifications (`mod`), the prefix which is used to build a globally unique name of the component in a hierarchical fashion (`pre`), 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 `inst_class_in` to do most of the work. Finally it generates equations from the connection sets collected while instantiating this class. The “result” of the relation are the equations and some information about what was in the class.

One of the most important relations is `inst_element` (see figure 7), that instantiates an element of a class. An element can be a class definition, a variable declaration or an extends clause. Below is shown only the rule for instantiating variable declarations. The comments to the right of the code try to explain what is happening and why.

Output

The equations and variables found during instantiation 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 this list is printed to the output file in a simple format.

```

relation inst_element =

  rule Prefix.prefix_cref(pre, Exp.CREF_IDENT(n, []))
    => vn &
    Lookup.lookup_class(env, t)                               Find the class definition
    => (cl, classmod) &
    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
    inst_class(env, mod', pre', csets, cl)                    Instantiate 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' &
    inst_mod_equation(env, pre, n, mod')                      Fetch the equation, if supplied
    => dae2 &
    list_append(dae1, dae2) => dae                            Concatenate the equation lists
    -----
    inst_element(env, mods, pre, csets,
                SCode.COMPONENT(n, final,
                                prot, attr, t, m))
    => ((*DAE.VAR(vn, DAE.LOCAL)::*) dae,
        env', csets', [(n, attr, ty)])

end

```

Fig. 7. The `inst_element` relation

CONCLUSIONS

The Modelica language is very young, and is still under development. No complete implementations are, at the time of this writing, available. This means that the exact semantics of the language is still a somewhat open issue, which our specification intends to close. The semantic specification covers all the aspects of the static semantics of Modelica.

There are several goals with the specification. One is to produce a complete implementation, which forces us to find all problems with the current design of the language and resolve all unresolved issues, which has proven to be a great help in the language design process.

Another goal is to assist current and future implementors by providing a semantic reference, as a kind of reference implementation. To accomplish this, the specification should be possible to read and understand by a human reader. This is of course not an easy task, but using RML instead of a conventional programming language helps immensely. Reading the RML source is not something anybody will be able to do, but fortunately only language implementors, who should already be familiar with the concepts of the RML specification, need to read it.

A third goal is to produce a usable and efficient translator. This is only partly done yet. The current translator treats all parameters as constants, and it does not communicate its result with any equation solver. This could easily be resolved with more work. No benchmarking of the translator has been done yet, but experience from previous RML specification makes us confident that the performance will be at least reasonable.

ACKNOWLEDGMENTS

This work had been supported by the Wallenberg Foundation in the WITAS project.

REFERENCES

- Abadi, M., and Cardelli, L., *A Theory of Objects*. Springer Verlag, ISBN 0-387-94775-2, 1996.
- Barton, P. I., and Pantelides, C.C., *Modeling of combined discrete/continuous processes*. *AIChE J.*, 40, pp. 966--979, 1994.
- Elmqvist, H., Brück, D., and Otter, M., *Dymola --- User's Manual*. Dynasim AB, Research Park Ideon, Lund, Sweden, 1996.
- Fritzson, P., Viklund, L., Fritzson D., Herber, J., High-Level Mathematical Modelling and Programming, *IEEE Software*, 12(4):77-87, July 1995.

ObjectMath Home Page, <http://www.ida.liu.se/labs/pelab/omath>

Otter, M., Schlegel, C., and Elmqvist, H., Modeling and Real-time Simulation of an Automatic Gearbox using Modelica. In *Proceedings of ESS'97 - European Simulation Symposium*, Passau, Oct. 19-23, 1997.

Modelica Home Page <http://www.Dynasim.se/Modelica>

Elmqvist, H., Mattsson, S. E., "Modelica - The Next Generation Modeling Language - An International Design Effort". In *Proceedings of First World Congress of System Simulation*, Singapore, September 1-3 1997.

Petersson, M., *Compiling Natural Semantics*, Linköping Studies in Science and Technology. Dissertation No. 413, 1995.

PCCTS home page: <http://www.ANTLR.org/>

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