DSVL for a subset of Modelica

Daniel Rieglehaupt

Universiteit Antwerpen (UA),Campus Middelheim, Middelheimlaan 1, B-2020 Antwerpen

Abstract

This paper gives a very small introduction to Modelica and explains the need for a DSVL. We do so by building a small DSVL for a subset of pieces from the library Modelica.Electric.Analog After detailing the building and experience of using the DSVL we try to form appropriate conclusions

Keywords: Domain Specific Visual Language, modeling, Modelica, Atom3, electrical circuit

1. Introduction

Modelica is a widely used modeling language for which there are many visual editors available. There are commercial ones like for example Dymola(a limited free demo is available) and free ones like OMEdit from OpenModelica. The constraints for the domains Modelica can handle are all build in as mathematical formulas as part of the Modelica standard library.Therefore those editors have limited constraints and return mathematical errors in the style of “can’t solve this singular matrix”. In this paper we are going to build a small Domain Specific Visual language for a subset of the Modelica.Electric.Analog standard library. The purpose is make the life of the average user easier by providing domain specific errors and constraints instead of mathematical ones.

The content of the paper is as follows: In section 2 we give a small introduction to Modelica, the reader familiar with Modelica may safely skip this. In section 3 we explain the reasons as to the need for a DSVL as stated in this introduction again; this time with an example. We also design

Email address: `Daniel.Riegelhaupt@student.ua.ac.be` (Daniel Rieglehaupt)

and build one for a small subset of pieces of the Modelica standard library: Modelica.Electric.Analog.
The next section, section 4, describes our observations after using the DSVL and we give our conclusions in section 5

2. A Small introduction to Modelica^{TM}

Modelica is a freely available, dynamic (notion of time) declarative (mathematical equations) OO language for multi-domain modeling. [1] Examples of domains are: mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power... [2] Although the basic structure in Modelica is a class the use of OO in Modelica is for hierarchical and inheritance purposes, not for message sending and such.

In order to give a small introduction to the language itself we use the following example, taken from [2]

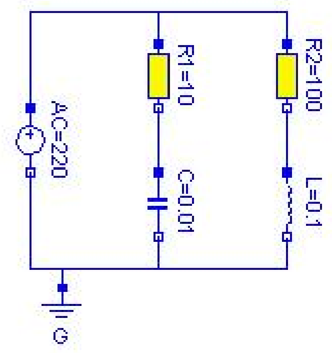

Figure 1: An example circuit

The circuit shown in Figure 1 can be modeled as follows:

```
model circuit
        Resistor R1(R=10);
        Capacitor C(C=0.01);
        Resistor R2(R=100);
        Inductor L(L=0.1);
        VsourceAC AC;
        Ground G;
equation
        connect (AC.p, R1.p); // Capacitor circuit
        connect (R1.n, C.p);
        connect (C.n, AC.n);
        connect (R1.p, R2.p); // Inductor circuit
        connect (R2.n, L.p);
        connect (L.n, C.n);
        connect (AC.n, G.p); // Ground
end circuit;
```

for clarity the imports and graphical annotations have been left out. Graphical annotations denote the graphical the circuit and is used by tools only not by the Modelica translator note that we will not consider those annotations when building our DSVL.
Let us explain this model. First of all a **model** is a class. The statements directly below are declarations. For example the first one Resistor R1(R=10) declares a component R1 of class Resistor and sets the value of its resistance to 10 Ohm. **Connect** is not an function but rather an operator that generates equations taking into account what kind of quantities that are involved. Let us explain this a bit. We start with a pin which is a connector (to elements of a circuits are connected by pins)

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

Were Voltage an Current are defined as :

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");
```

a type is a class and real is a predefined variable type with attributes such as unit of measure , initial value, min and max values . . . (here the unit of measure are SI units) A connection *connect (Pin1, Pin2)*, with Pin1 and Pin2 of connector class Pin, connects the two pins such that they form one node. This implies two equations, namely $Pin1.v = Pin2.v$ and $Pin1.i + Pin2.i =$

0. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current 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. Note that the use of an equation *Pin1.i + Pin2.i = 0* instead of something like *Pin1.i = - Pin2.i* which most programmers might expect. We can now build an interface for elements that have to pins p and n (positive and negative) like for example a source or a resistance…

```
partial model OnePort
"Superclass of elements with two electrical pins"
        Pin p, n;
        Voltage v;
        Current i;
equation
        v = p.v - n.v;
        0 = p.i + n.i;
        i = p.i;
end OnePort;
```

We can now create a resistor by using the one port and adding the equation of Ohm's law.

```
model Resistor "Ideal electrical resistor"
        extends OnePort;
        parameter Real R(unit="Ohm") "Resistance";
equation
        R*i = v;
end Resistor;
```

Note that when taking into account time equation and the 'flow' this resembles causal block diagram. Modelica itself is not causal but a Modelica model can indeed be translated into a causal block diagram this is however beyond this paper's goal

The above was just a very short introduction in order to explain the principles behind Modelica, there is however much more to Modelica then just this. For more information on Modelica see:
`https://www.modelica.org/`
`https://www.modelica.org/documents/ModelicaSpec32.pdf`

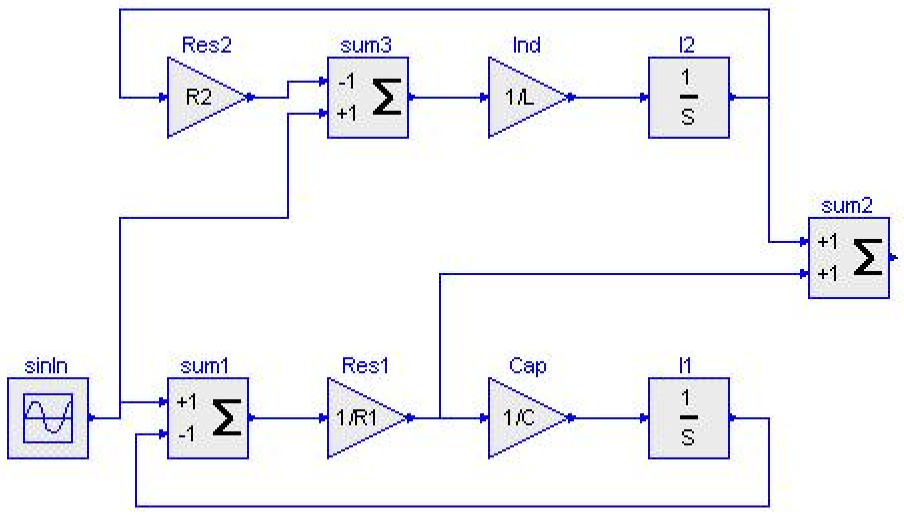

Figure 2: An example of a causal block diagram

3. A DSVL for Modelica

3.1. Why ?

Modelica is a multi-domain mathematical language, therefore the constraints are of a mathematical nature instead of domains specific. When making a simple error in a model this can lead to very confusing and unclear errors. Consider the very simple model of a resistor connected to source shown in Figure 3. When simulated on the Dymola7 simulator it will give the following errors:

```
Translation of Unnamed:
DAE having 12 scalar unknowns and 12 scalar equations.
Error: The equations
  equation
  constantVoltage.p.i+resistor.p.i = 0;

  which was derived from
  constantVoltage.p.i+resistor.p.i = 0;

  0 = constantVoltage.p.i+constantVoltage.n.i;

  constantVoltage.i = constantVoltage.p.i;
```

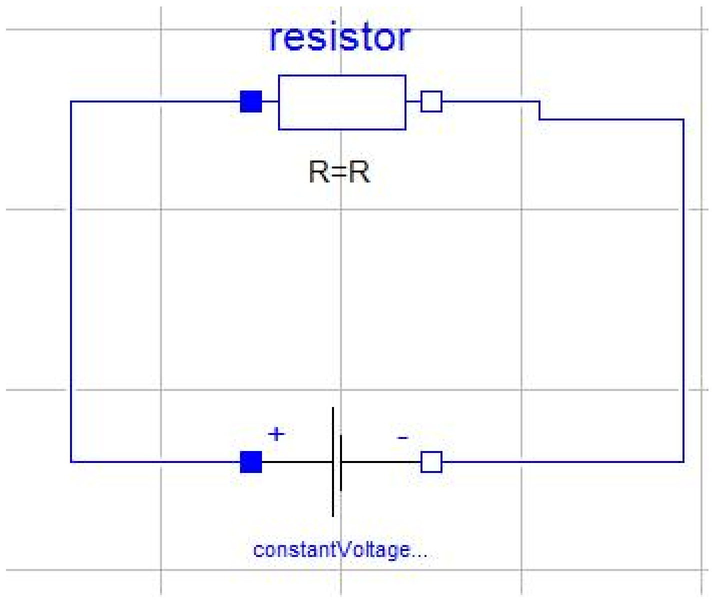

Figure 3: A simple circuit

```
  0 = resistor.p.i+resistor.n.i;

  resistor.i = resistor.p.i;

  constantVoltage.n.i+resistor.n.i = 0;

mean circular equalities for
constantVoltage.p.i, constantVoltage.n.i,
constantVoltage.i, resistor.p.i,
resistor.n.i, resistor.i

Translation aborted.

Translation aborted.

Translation aborted.
ERROR: 1 error was found
```

This is not very user friendly, especially considering that all this means is : an electrical circuit must contain exactly one ground. It is now easy to see that a domain specific visual language would be very useful in this context. It is much clearer to have errors and constraints specific to a domain then general errors about a set of equations.

3.2. A case study: a small DSVL for Modelica

We are now going to create a small DSVL for the following pieces of the Modelica standard library. The following pieces from the Modelica.Electric. Analog.Basic library will be used :

- Ground

```
model Ground "Ground node"
  Modelica.Electrical.Analog.Interfaces.Pin p;
equation
  p.v = 0;
end Ground;
```

- Resistor

```
model Resistor "Ideal linear electrical resistor"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter SIunits.Resistance R=1 "Resistance";
equation
  R*i = v;
end Resistor;
```

- Capicitor

```
model Capacitor "Ideal linear electrical capacitor"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter SIunits.Capacitance C=1 "Capacitance";
equation
  i = C*der(v);
end Capacitor;
```

- Inductor

```
model Inductor "Ideal linear electrical inductor"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter SIunits.Inductance L=1 "Inductance";
equation
  L*der(i) = v;
end Inductor;
```

from Modelica.Electric.Analog.Sources we use:

- Constant Voltage

  ```
  model ConstantVoltage "Source for constant voltage"
    parameter Modelica.SIunits.Voltage V=1
          "Value of constant voltage";
    extends Modelica.Electrical.Analog.Interfaces.OnePort;
  equation
    v = V;
  end ConstantVoltage;
  ```

- Step Voltage

  ```
  model StepVoltage "Step voltage source"
    parameter Modelica.SIunits.Voltage V=1 "Height of step";
    extends Modelica.Electrical.Analog.Interfaces.
          VoltageSource
    (redeclare Modelica.Blocks.Sources.Step
          signalSource(height={V}));
  end StepVoltage;
  ```

- Sine Voltage

  ```
  model SineVoltage "Sine voltage source"
    parameter Modelica.SIunits.Voltage V=1
          "Amplitude of sine wave";
    parameter Modelica.SIunits.Angle phase=0
          "Phase of sine wave";
    parameter Modelica.SIunits.Frequency freqHz=1
          "Frequency of sine wave";
  ```

```
    extends Modelica.Electrical.Analog.Interfaces.
          VoltageSource (redeclare Modelica.Blocks.
                  Sources.Sine signalSource(
                          amplitude={V},
                          freqHz={freqHz},
                          phase={phase})
          );
  end SineVoltage;
```

note: we can't directly see this in the definition but both the step voltage source and the sine voltage source also have the parameters offset (in V) and startTime (in s). The definitions can be found in the Modelica files themselves but a nice overview(including inherited parameters) can be found here: http://www.ida.liu.se/ pelab/realsim/library/Modelica/docu/Modelica_Electrical_Analog.html#Modelica.Electrical.Analog

We will now build the language using atom3(A Tool for Multi-formalism and Meta-Modeling)in the class diagram formalism. First we need input and output gates for the elements of the circuit. We take the positive pin as the input and the negative pin as the output gate. (note that this is purely for the editor, we will not declare the pins ourselves, they are already declared in the models we are going to use) The connections to elements have the following constraints (cardinalities) each element can have at most 1 input and 1 output gate (a ground only has an input). When connecting gate g1 to gate g2 in the editor the following must hold:

- g1 $\neq$ g2
- there is not already a g1 $\rightarrow$ g2 connection
- there is not already a g1 $\leftarrow$ g2 connection

All possible connections are allowed (positive to positive , negative to negative, positive to negative and negative to positive)

Now that we have this we add a class for each of the elements and give them the exact same class name and attributes, including default values, as the ones they have in Modelica. A key attribute name is also added. In order to avoid having to add gates manually every time we add an element we simply add an action on create which will do this automatically. That action is handy when creating a model but not so much when loading a saved one. New gates will be created when loading along with the ones that were already there. This will cause a constraint violation because the cardinalities are not respected. We solve this by cheating. First we raise the maximum allowed to 2 - so that no constraint violation will pop up when loading - then we add a cheat action to remove the gates too many. There is no danger of the user manually adding a second gate simply because we have removed the create new gates buttons. There is no need for them because the gates are created automatically after all. A valid question here is "why not simply check if the gates are already there and only create if they aren't ?".The response is that the action is done on creation of the piece, it isn't connected to anything yet so we can't check. There are slightly different way of doing the same thing but it all comes down to the same conclusion. If we don't want to disturb the user with a raised constrain we must first allow the connection then remove it (you can't check otherwise).

We now have an editor (or to be more prcised a visual meta language for simple electircal languages that can be used in atom3), the next step is to check for language constraints and build a Modelica codegenerator. This will happen in python.

We will check for 3 constraints in that order:

- Every circuit must be closed
- Every circuit must have at least one source
- Every circuit must have exactly one ground

To check if a circuit is closed we simply iterate over every element and say that they must be connected through all their gates (meaning for every gate either there is at least one outgoing connection to another gate or one incoming connection from another gate).

To see if every circuit has a source we make a recursive procedure that checks elements reachable form other pieces. It will stop when every times he checks for next and previous elements they are all reachable already. We also assign each source an index and keep a list of sources that reach a certain ground. Once that is done and because we know that the circuits are all closed we simply iterate over every element to see if its reachable status is true. If there is at least one that can't be reached from a source it means the circuit doesn't contain one.

Now that we know that all circuits are closed and they all contains sources we can now check for grounds. We do this by looking at the list of sources that reach a ground that we kept for every ground. Each of those lists must be mutuality disjunctive to every other list or else that would mean that a circuit has more than one ground. Once we know that each list of sources contains unique sources we can check if every circuit has a ground by taking the sum of the length of those lists and seeing if it is equal to the number of sources there are.

Generating the code is much more straightforward. Iterate over every element and declare it. Then iterate over every element and add connections for the elements it can reach. The connection requires us to give ports - as in connect(elem.p , elem2.n) - therefore the adding connections is done in 4 parts: Check for connections going from his pos gate to the pos gate of another element , check for connections going from his pos gate to the negative gate of another element...
Congratulations we now have a simple Modelica generator!

4. Observations

After playing around a bit with his newly made created 'toy' the author of this paper discovered by chance that Modelica (neither Dymola nor OpenModelica) can simulate two sources in parallel. This is perfectly possible in reality. On the condition, of course, that they have the same voltage or else the most powerful one of the two would also flow into the other one causing heating. Another surprise a simple model of a source, a ground and a capacitor won't simulate in OpenModelica but will simulate in Dymola (evaluating to 0). Note that it is normal that it doesn't work well; the capacitor works blocking but the difference in reaction is strange. It should also be pointed out that the author didn't know that the capacitor would work blocking, herby illustrating the need for a domain expert when working on a project like this in a bigger scale.

What we have been building constraint wise is a DSVL for the domain of electrical circuits (or at least a subset of it) and not for Modelica; we simply translated it to Modelica. (To clarify a DSVL for Modelica should for example also have the constraint of no parallel sources.) This is actually a good thing, the same DSVL can be used to translate to other modeling languages; but the fact remains that the purpose of this assignment/paper was to create a DSVL for Modelica. This would appear to be not completely realizable based only on the domain constraints. In other words in order to build a DSVL for Modelica specifically we need not only the specifications of the domain we're building for (a domain expert would be very usefull) but we also need the Modelica version of the specification for that particular domain.

Modelica however is not documented as such. It is clearly specified as a language, but the restrictions of the domain specific environment are formulated in mathematical equations considered part of the Modelica language rather than part of the domain. This leads to strange results that seem to vary depending on which simulator is being used. In order to build a correct translator for Modelica specifically first specifications have to be made as to the exact capabilities of Modelica in a particular domain.

5. Conclusion

The entire conclusion can be given in one sentence. While a DSVL is certainly useful (and probably very needed), in order for it to be fully compatible with Modelica specifications are needed of the capabilities and limitations of Modelica in the particular domain the DSVL is for.

6. Future work

It is interesting to note that Modelica icons are annotations within the pieces. So when doing a large DSVL it would be interesting to be able to extract those and transform them into icons usable by our editor (in our case Atom3). This could be done with for example ANTLR. One would simply need the EBNF of these annotation to create a quick parser with ANTLR, parse the file and then use ANTLR string templates to print out an ATOM3 graph.py. While there is some difficulty in the fact that no EBNF is given in the specification, some annotations may be hierarchical or vendor specific; it is still an interesting exercise to do.

7. References

[1] Fritzson, P., 2006. Introduction to object-oriented modeling and simulation with openmodelica.
URL http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/Documents/ModelicaTutorialFritzson.pdf

[2] Modelica-Association, December 2000. Modelica^{TM}- a unified object-oriented language for physical systems modeling tutorial. Version 1.4.
URL https://www.modelica.org/documents/ModelicaTutorial14.pdf