Design-Space Exploration using Alloy

Ing. Ken Vanherpen
University of Antwerp, Belgium
ken.vanherpen@uantwerpen.be

Abstract

In the growing world of MDE many tools are offered to meta-model (a part of) a system, constrain it, and create instances of the meta-model. Those tools can be used to describe and solve problems in different domains. An example of such a tool is Alloy, which is a declarative constraint language for describing structures and a tool (Alloy Analyzer) for exploring them. However, these tools are limited in that they can not simulate the created instances and check their correctness. This is where tools such as Modelica comes in handy. Modelica is an equation based language to model complex physical systems from different domains. In order to combine the best of two worlds, this work describes a method to meta-model a domain specific using Alloy, where after the instances are translated to a Modelica model to simulate them.

Keywords:
MDE, Alloy, DSL, DSE, Modelica

1. Introduction

Nowadays, a domain-specific problem is mostly solved by using a set of background information and experience of the domain expert himself. For example, given a problem in the electrical domain, the domain expert will use its "toolbox" of components (resistors, capacitors,...) and analytical equations in order to find a combination of components which satisfy the specifications. The Model Driven Engineering (MDE) approach allows a domain expert to meta-model his background information of a particular domain-specific problem, where after instances (i.e. possible solutions) of that model are generated. The domain expert has the flexibility to what

extent the meta-model is constrained. Off coarse, a less constrained meta-model will generate more instances. Anyway, all of the generated instances are possible solutions, which needs to be verified on their correctness related to the specific domain. Therefore, a transformation to another formalism is needed in order to simulate and evaluate the correctness of those instances.

An example of this approach is shown in [Sen and Vangheluwe, 2006]. A multi-domain problem is meta-modeled using Unified Modeling Language Class Diagram (UML CD), which is constrained using OCL or Python. An instance of this meta-model is ultimately transformed to an object-oriented textual formalism called Modelica [Modelica, 2014] by using Graph Grammar rules. This is achieved by an intermediate transformation to the Bond Graph formalism. In [Sen, Baudry, and Vangheluwe, 2008] a methodology is represented for automatic model completion. A domain-specific modeling language (DSML) is created by a domain-expert, consisting out of a meta-model constrained by Alloy [MIT, 2014] facts. Alloy is a declarative constraint language for describing a meta-model of a domain-specific language, and create instances of it. In [Sen et al., 2008] it is used to textually constraint the meta-model, which is build in AToM^3. By transforming the DSML to a model editor, a domain specific design space model is created. This is used by designers, which are not a domain-experts, to draw a partial model of particular problem. A third step of the process transforms the meta-model, Alloy constraints (i.e. facts) and the partial model to an Alloy model. Alloy will solve this model, create instances (i.e. possible solutions) and return them to the model editor. These recommendations are used by the designer to complete the model.

This paper will present a method to solve a domain-specific problem using Alloy. Therefore, the meta-model of the domain-specific language (DSL) will be described using Alloy its declarative constraint language. Alloy will generate instances of this meta-model which are possible solutions to our domain-specific problem. In order to verify these instances, we will transform them to the object-oriented textual formalism of Modelica where they can be simulated. After evaluating the performance of each of those instances, one can determine the optimal solution. This work flow is represented in Figure 1. As a running example, our domain-specific problem is situated in the electrical domain. By using Alloy, we will try to find a Low-Pass Filter (LPF) which passes frequency signals until 50 Hz. When this problem is solved analytically, one can calculate that a resistor of 100 Ω and a capacitor of 33 μF is needed. This schematic is represented in Figure 2, where one can

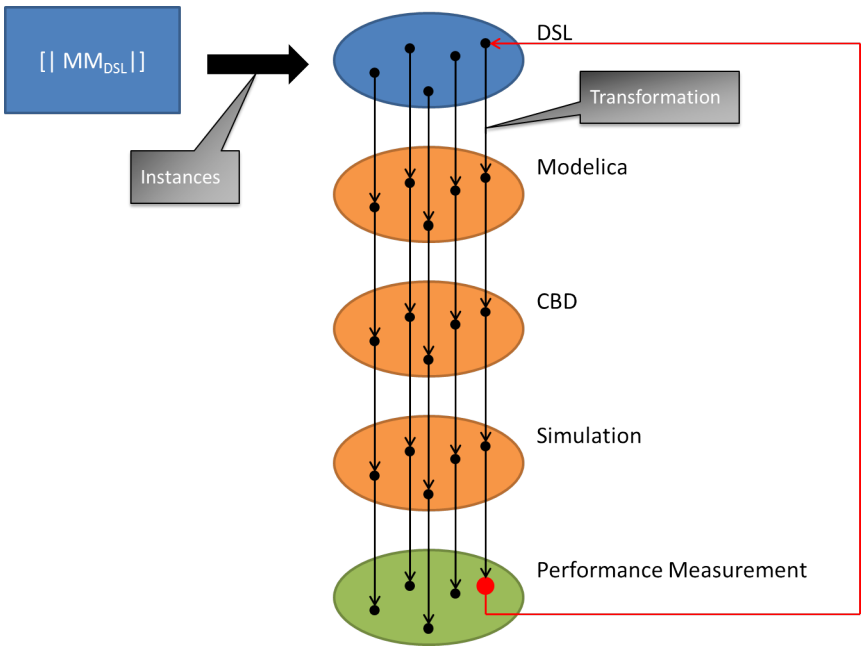


Figure 1: Work flow

notice that the blue part will be meta-model by using Alloy while to orange part is provided by Modelica. This paper is structured as follows. Section 2 elaborates how a filter is meta-modeled using Alloy. Section 3 shows how instances of this meta-model are created. Transforming those instances to the object-oriented textual formalism of Modelica is shown in section 4, this by using our running example. In section 5 our running example will be simulated and validated. Finally, section 6 concludes our work and suggests future work.

2. Meta-modeling using Alloy

As presented in Figure 1, the first step in the process is defining the meta-model using Alloy. The meta-model contains the knowledge of the domain expert, in this case an electrical engineer for example. It's up to the domain expert to determine in which extend the meta-model is constrained, which will influence the number of generated instances. For example, when we constrain the number of components to only one resistor of $100\,\Omega$ and one capacitor of $33\,\mu$F, the number of generated instances will be two. When

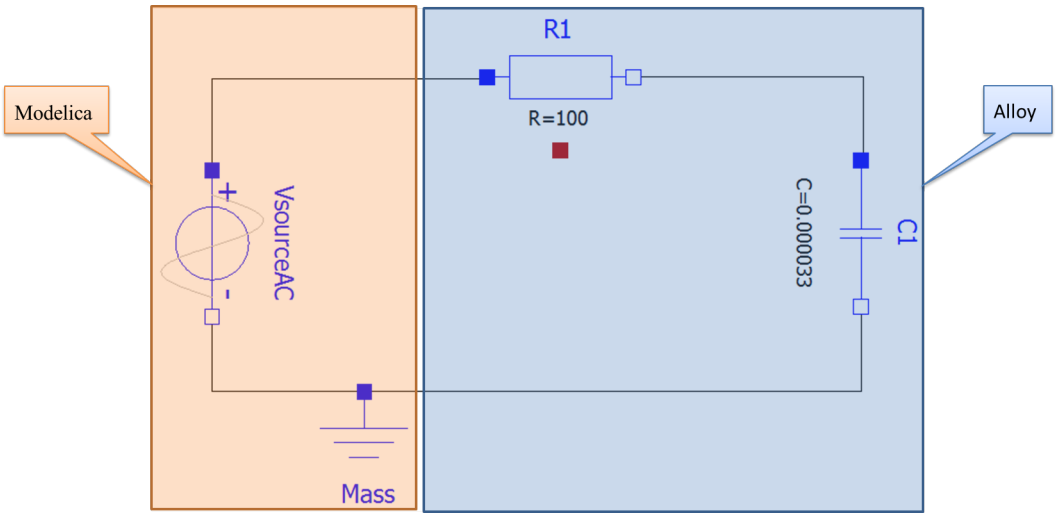


Figure 2: Schematic of the Low-Pass Filter example

we add one resistor of 100 Ω, the number of generated instances increases to six. The meta-model of our running example -a (low-pass) filter- is shown in Figure 3. The following subsections will describe the meta-model in detail, showing how this is modeled using the Alloy syntax.

2.1. Filter

A filter always contains a set of some components in order to give the filter its appropriate functionality. In our case, those components can be resistors and/or capacitors. As can be seen in Figure 3, all of the components are derived from a "super class" called 'Component' in our meta-model. This makes it possible to define a circuit which can hold a set of components. This is defined in the body of the signature 'Filter', shown in Code block 1. Since the objective is to translate the instances to the object-oriented textual formalism of Modelica, there is chosen to define some connection nodes. This will make the transformation straightforward. The multiplicity keyword 'one' in front of the signature constraints the amount of filters in a generated instance to only one.

2.2. Component

As already mentioned in the previous subsection, all of the components are derived from the "super class" 'Component'. In Alloy, this is done by

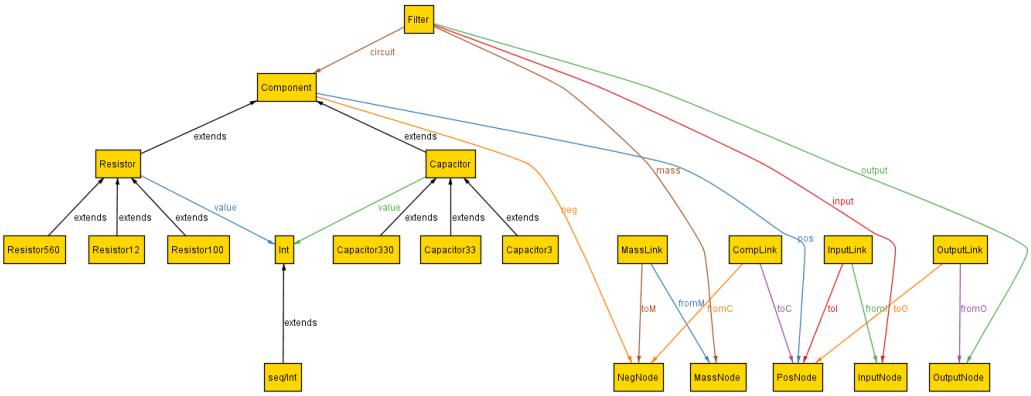


Figure 3: Meta-model of (LPF-)filter

```
one sig Filter{
  input : InputNode,
  output : OutputNode,
  mass : MassNode,
  circuit : set Component,
}
```

Code block 1: Alloy - signature: Filter

```
abstract sig Component {
  pos : one PosNode,
  neg : one NegNode
}
```

Code block 2: Alloy - signature: Component

```
abstract sig Resistor, Capacitor extends Component {
  value : Int
}

//Resistor of 12 Ohm
sig Resistor12 extends Resistor {
}
{ //Appended fact => constraint: value of the resistor
  value = 12
}
```

Code block 3: Alloy - signature: Resistor and Capacitor

writing the keyword 'abstract' in front of a signature. Since each component needs to be connected to a node of the filter or another component, this signature defines some general attributes in its body. They describe the two possible connections a component contains, positive and negative connection. This can be seen in Code Block 2. Again, the multiplicity keyword 'one' constraints how many nodes a component can contain.

2.3. Resistor and Capacitor

The signature 'Component' is extended by the signatures 'Resistor' and type 'Capacitor'. There is chosen to keep those signatures abstract, in order to extend them with a specific resistor or capacitor signature defining a specific value. This is done by defining a relation 'value' in the body of the abstract signatures 'Resistor' and 'Capacitor'. By using an appended fact, this relation can be specified in the extended signatures. This can be observed in Code Block 3, where a resistor with a value of $12\,\Omega$ is modeled.

2.4. Links

Nodes, either from the filter or a component, needs to be connected to each other. Therefore, some signatures are defined:

```
1 //Link
2 some sig CompLink {
3   fromC : NegNode,
4   toC : PosNode
5 }
```

Code block 4: Alloy - signature: CompLink

- CompLink: describes that a negative node of a component needs to connect to a positive node of a component.
- InputLink: describes that the input node of the filter needs to connect to a positive node of a component.
- OutputLink: describes that the output node of the filter needs to connect to a positive node of a component.
- MassLink: describes that the mass node of the filter needs to connect to a negative node of a component.

An example is shown in Code Block 5, where the signature 'CompLink' is described. Notice the multiplicity keyword 'some', indicating there exist one or more connections between components. Furthermore, the domain expert can specify how many links from a component-node to a filter-node can/may occur. In our example, we constrained these link with a multiplicity of one. This because we know what kind of instance we are looking for, namely an instance which represents the well known Low-Pass Filter (LPF).

2.5. Constraints

The previous subsections described how a filter, components, nodes and links are modeled. Their multiplicities were constrained by using keywords such as 'one', 'some', etc. However, Alloy will create instances of such a meta-model which are far from valid. The following list describes those invalid instances:

- An instance containing components which are not part of the filter.
- An instance where a node is not related to a component.
- An instance in which a component its positive node is connected to its negative node.

- An instance where duplicate links occur.
- An instance where nodes are not linked to other nodes.

To suppress those invalid instances, constraints needs to be specified which is done by defining facts in Alloy. These are shown in Code Block **??**.

3. Creating instances of the meta-model

When creating a run-command in Alloy, instances (i.e. solutions) of the meta-model can be generated. When a run-command without further restrictions is given for the meta-model described in the previous section, hundreds of possible solutions are generated. Since we want to find an instance which is a representation of a Low-Pass Filter, we can further restrict the meta-model by the following run-command:

run for exactly 2 Component, 2 PosNode, 2 NegNode, 1 CompLink, 11 int

This allows us to further restrict the signatures with a multiplicity keyword 'some' to the one specified in the run-command. Not surprisingly, one of the generated instances is the one as shown in Figure 4.

4. Exporting the generated instances

In order to simulate a generated instance, it needs to be transformed to the object-oriented textual formalism of Modelica. Unfortunately, Alloy doesn't support such a transformation. On the other hand, Alloy supports the ability to export an instance to a XML-file. We decided to develop an exporter in python, which translates such a XML-file to an appropriate Modelica-file. This section will briefly describe this exporter, which consists out of two python-files.

4.1. Parser

A first step in the transformation is the interpretation of the generated XML-file. This is done by a python file called 'AlloyParser'. As the file name indicates, it parses the XML-file in order to retrieve the structure of the instance generated by Alloy (Figure 4). This is obtained by executing the following steps:

```
1
2  fact {
3    all fil : Filter, comp : Component | comp in fil.
       circuit            //Make sure each component is an
        element of the circuit
4  }
5
6  fact {    //All nodes (Positive and Negative) needs to
     be connect to a component => No floating instances
7    all n : NegNode | one c : Component | n in c.neg
8    all p : PosNode | one c : Component | p in c.pos
9  }
10
11 fact {    //No self-links
12   all cl : CompLink | all c : Component | all p :
       PosNode, n : NegNode {
13     ( p in c.pos && n in c.neg ) => n in cl.fromC => p !
         in cl.toC
14   }
15 }
16
17 fact {    //No links which connects the same nodes to
     each other
18   all cl1, cl2 : CompLink | all c1, c2 : Component | all
        p : PosNode | all n: NegNode {
19     ( cl1 != cl2 && c1 != c2 ) => ( p in c2.pos && n in
         c1.neg ) => ( n in cl1.fromC && n in cl2.fromC )
         => p in cl1.toC => p !in cl2.toC
20   }
21 }
22
23 fact {    //All nodes have an appropriate link
24   all p : PosNode | all n : NegNode | one cl : CompLink
       | one il : InputLink | one ol : OutputLink | one ml
        : MassLink {
25     (p in cl.toC || p in il.toI || p in ol.toO) && (n
         in cl.fromC || n in ml.toM)
26   }
27 }
```

Code block 5: Alloy - signature: CompLink

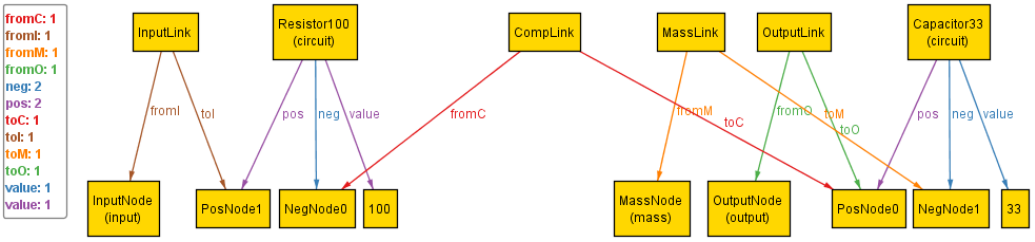


Figure 4: Instance representing a Low-Pass Filter

1. Retrieve all of the filters (in this case, just one) and their relevant information. This information includes the filter its ID, name of its input and output node, the XML-ID and name of its components. This is stored in an array.
2. For each filter, retrieve its components. This includes their name, value, name of the positive and negative node. This information is stored in an array.
3. For each filter, retrieve the from-node and the to-node for each type of link (CompLink, InputLink, etc.).

We have chosen to store all of this retrieved information in multi-dimensional arrays. This way, the parsing is universal in such a way that it can be used when multiple filters are present in one and the same instance. For example, when cascading filters.

4.2. Creating the Modelica-file

A Modelica-file consists out of classes, types, connectors and models in order to describe a certain domain specific problem. Most of these are static and thus needs to be modeled just once for each domain specific problem. Others are dynamic, and thus will depend on the used components and their connections. This can be explained by our running example -situated in the electrical domain- where a Modelica-file must consist out of:

- A class which describes the circuit: its components and their connections.
- A type for describing the voltage and current units.
- A description of a connector its voltage and current.

- A model of a node describing the laws of Kirchhoff.
- A model of a resistor.
- A model of a capacitor.
- A model of a voltage source.
- A model of a ground.

It is clear that only the class which describes the circuit will depend on the instances generated by Alloy. Therefore, the information stored in the multi-dimensional arrays described in subsection 4.1 are used. Again, all this is done by executing some steps in a certain order:

1. A look-up table is made to translate XML-names to appropriate Modelica-names. For example, the array [Resistor100$0', '100', 'PosNode$1', 'NegNode$0'] retrieved from the XML-file, needs to be read as ['R1', '100', 'R1.p', 'R1.n'].
2. The class which describes the circuit its components and their connections is written to a new Modelica-file. Therefore, the multi-dimensional arrays of the parser are used in combination with the look-up table.
3. All of the static elements are written to the Modelica-file.

The python file 'AlloytoMO' can be used for the creation of the Modelica-file. This can be done by executing the following commands:

exporter = XMLtoMO()
alloyFile = 'LPF.xml'
modelicaFile = 'LPF.mo'
exporter.setVoltage(220)
exporter.setFrequency(50)
exporter.createFile(alloyFile, modelicaFile)

Automatically the parser described in subsection 4.1 will be called. As an option, one can set the voltage and frequency of the voltage source, which comes in handy when simulating the filter in section 5. As a result, a schematic as shown in Figure 2 will be obtained as an object-oriented textual Modelica formalism.

5. Simulate the transformed instances

The simulation of the created Modelica-file is done by using the tool JModelica [**?**]. Jmodelica is a Modelica-based open source tool which can be used to analyze and simulate complex dynamic systems. Using the JModelica Python interpreter, one can compile the created Modelica-file where after it can be simulated. The simulation results are stored in array, which can be accessed in order to plot them or perform some additional calculations.

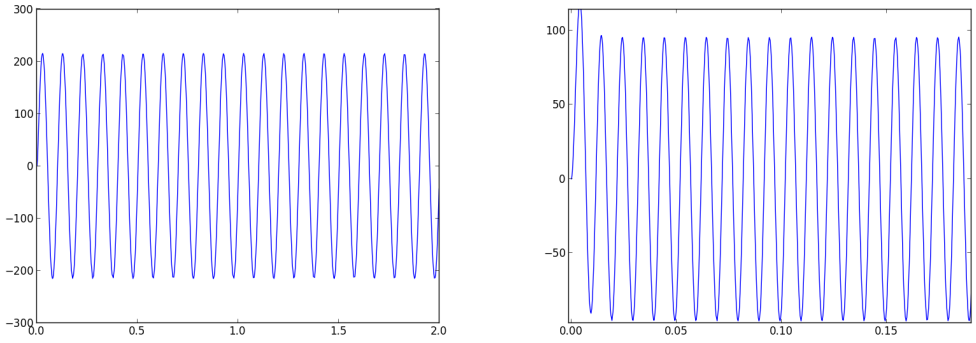
To simulate our running example, in order to verify the requirements of our Low-Pass Filter, the JModelica Python interpreter executes the python file 'simulateAlloyModel'. It was our intend to automatically parse the generated Alloy instances, transform them to an appropriate Modelica-file where after it is simulated by executing one single python file. Unfortunately, the JModelica Python interpreter doesn't support the importation of "unknown" python files.

As an example, Figure 5 shows the simulation result of our running example shown in Figure 2, where the voltage across capacitor C1 is displayed. A sin-wave voltage with an amplitude of 220 V is applied with a frequency of respectively 10 Hz and 100 Hz. As one can notice, the voltage across capacitor C1 when applying a frequency of 10 Hz to the circuit is identical to the one which is applied. On the contrary, when applying a frequency of 100 Hz the voltage across C1 isn't zero. However, it is attenuated to approximately 97 V. This is due to the gain-magnitude frequency response of a first-order Low-Pass Filter, which attenuates with a slope of -20 dB/decade after crossing the cutoff frequency. Therefore, one can conclude Alloy generated an instance which is conform to our analytical solution.

6. Conclusion and future work

This paper described a method to meta-model a domain-specific problem in Alloy, where after it's instances (i.e. possible solutions) are transformed to the object-oriented textual formalism of Modelica. By using Modelica, instances can be simulated and evaluated to their correctness. More over, instances which conforms the given requirements can be compared in order to determine the optimum valid instance.

However, the described method still needs some human interaction, due to some bottlenecks, which constraints the practical use when hundreds of instances are generated. A first bottleneck is the generation of instances by



(a) 10 Hz (b) 100 Hz

Figure 5: Simulation results of the Alloy LPF-instance

Alloy. These are done one by one, and thus they need to be exported one by one. One possible solution to overcome this bottleneck is using the Alloy API within a programming environment instead of the provided executable. A second bottleneck is the limitation of importing "unknown" python files when using the JModelica Python interpreter. A quite straightforward method to overcome this problem is to simulate all the generated Alloy instances in some kind of batch method. Obviously, it is our intention to solve these bottlenecks to obtain a more practical to use method. This way, some valid instances out of a set of hundreds may be compared to each other to determine the optimum solution.

References

MIT, December 2014. Alloy.
URL `http://alloy.mit.edu/alloy/index.html`

Modelica, 2014. Modelica.
URL `https://www.modelica.org/`

Sen, S., Baudry, B., Vangheluwe, H., 2008. Towards domain-specific model editors with automatic model completion.

Sen, S., Vangheluwe, H., 2006. Multi-domain system modeling and control based on meta-modeling and graph rewriting.