ENABLING DESIGN-SPACE EXPLORATION FOR DOMAIN-SPECIFIC MODELLING

Bart Meyers

Flanders Make vzw, Belgium
firstname.lastname@flandersmake.be

Joachim Denil, Ken Vanherpen, Hans Vangheluwe

University of Antwerp, Belgium
Flanders Make vzw, Belgium
Vangheluwe: McGill University, Canada
firstname.lastname@uantwerpen.be

ABSTRACT

Design-Space Exploration (DSE) looks for a suitable candidate solution to a problem, with respect to a set of criteria, by searching through a space of possible solution designs. Domain-Specific Modelling (DSM) allows language engineers to create Domain-Specific Languages (DSLs) for a particular domain, allowing non-technical domain experts to use the DSL to model a system, analyse, optimise or transform the model, generate code or documentation, etc. This paper presents a framework to enable DSE for DSM, so that non-technical domain experts can define DSE input using DSL syntax, and obtain DSL instances as a result of execution the DSE. The contribution of our framework is twofold: (1) automatic generation of a family of related DSLs (to describe structural constraints as well as constraints on simulation results) for modelling a DSE problem at the DSL level from a given DSL definition, and (2) generic support for executing a DSE algorithm, which searches the design space and generates suitable DSL instances. The framework can be applied to any explicitly defined DSL with an explicitly defined semantic domain. We evaluate this claim by applying our framework to a user-defined Simulink library. The approach is explained using a DSL for modelling electronic filters.

Keywords: Design-Space Exploration, Domain-Specific Modelling

1 INTRODUCTION

Design-Space Exploration (DSE) and optimization look for a suitable candidate solution, with respect to a set of quality criteria, by searching through a design space. Different approaches to design-space exploration are currently in common use in different engineering disciplines. Examples include mathematical optimization techniques such as Mixed Integer Linear Programming (Zeng and Natale 2010), Constraint-Satisfaction techniques (Sen, Baudry, and Vangheluwe 2010, Jackson, Kang, Dahlweid, Seifert, and Santen 2010) and Search-Based Optimization techniques (Burton, Paige, Rose, Kolovos, Poulding, and Smith 2012, Williams, Poulding, Rose, Paige, and Polack 2011). These techniques require complex constraints, algorithms, goal-functions, search methods, etc. as input.

Model-Driven Engineering (MDE) (Schmidt 2006) uses abstraction to bridge the cognitive gap between the problem space and the solution space in complex system problems in general and in software engineering problems in particular. To bridge this gap, MDE uses models to describe complex systems at multiple levels of abstraction, using appropriate modelling formalisms. Model transformations (Sendall and Kozaczynski 2003) can be defined to translate models, optimise, or analyse them, keep models consistent, generate code or documentation, etc. In MDE, Domain-Specific Modelling (DSM) (Kelly and Tolvanen 2008) allows a language engineer to create their own Domain-Specific Modelling Languages (DSLs) according to the needs of domain experts, who often lack programming skills. Such a DSL represents the set of valid instance models, i.e., the design space, of its domain. A DSL allows non-technical domain experts to precisely express systems in their domain, which can be optimised, analysed, used for code generation, etc.

Spring Simulation Multi-Conference (SpringSim'18), April 15-18, Baltimore, MD, USA

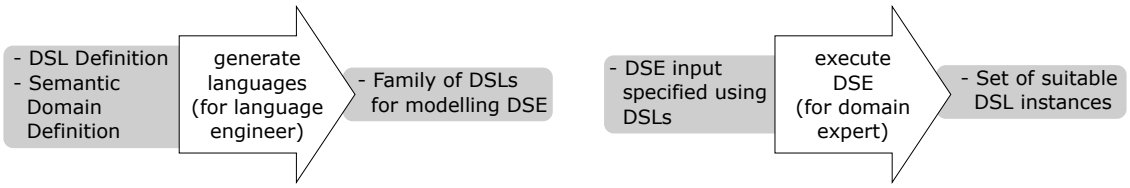

Figure 1: Overview of the two automated steps of the approach, represented as arrows.

Because DSLs represent a design space, a natural step is to search this design space according to some given quality criteria. The goal is, from a given set of domain-specific concepts as specified in the DSL, to generate a model that is deemed suitable according to the DSE input (i.e., constraints, goal function, etc.). Following the principles of DSM, domain experts should be able to express the necessary DSE input (i.e., constraints) in a domain-specific way, such that DSE becomes part of the tool kit of non-technical users. This paper presents a generic framework that provides DSM with DSE support. The contribution of this paper is twofold, as illustrated in Figure 1:

- we provide a means to automatically derive a set of DSLs from a given DSL, which can be used to model all constraints to a DSE problem for that DSL. This includes static or structural constraints, as well as constraints on the behaviour, i.e., simulation results of the model candidates;
- we provide an automatic mapping to and from a DSE backbone, so that suitable instance models can be generated.

We first explain our generic approach using a running example of a DSL for modelling electronic filters (called *EF*), for which we will try to automatically find models that are low-pass filters. Our framework is applicable to any DSL that is explicitly defined, and has an explicitly defined semantic domain. We illustrate the validity of this claim by applying the framework to Simulink. In this paper, we provide an elegant solution for the subclass of DSLs that have a simulation trace as semantic domain (in our example, a Bode magnitude plot).

2 RUNNING EXAMPLE

A DSL is defined by a language engineer, and describes the set of valid instance models, i.e., the design space, of its domain. A DSL definition consists of an abstract syntax model (representing the structure of its instance models, in the form of a metamodel), concrete syntax model (representing the visual representation of the instance models, in the form of a concrete syntax model), and model of the semantics or semantic mapping (representing a translation to a well-known semantic domain, analogous to a compiler, in the form of a model transformation).

2.1 Electronic Filters

The running example is a visual DSL called *EF*, representing an electronic filter composed of resistors, inductors and capacitors. An example is shown in Figure 2 on the left. The surrounding rountangle represents the filter, and has three ports located left (the input port), right (the output port) and bottom (the mass port). The filter connects the input port to the output port and mass port through a topology of resistors, capacitors and inductors. It is assumed that the input connects an electric source to the system, but it may as well be an output signal of an entire circuit. The mass port leads directly to a ground. The filter behaviour is measured by inspecting the potential difference at the output port and mass port.

The goal of *EF* is to model filters, intended to diminish or enhance certain frequencies from a signal. To inspect the filter behaviour of a given filter topology, a Bode magnitude plot shows the gain (in decibels) of each frequency (in Hertz), measured between the output port and mass port. The Bode magnitude plot of the example is shown on the right of Figure 2. It shows how the example filter filters frequencies around 500 Hz. For our DSL, the Bode magnitude plot is the semantic domain.

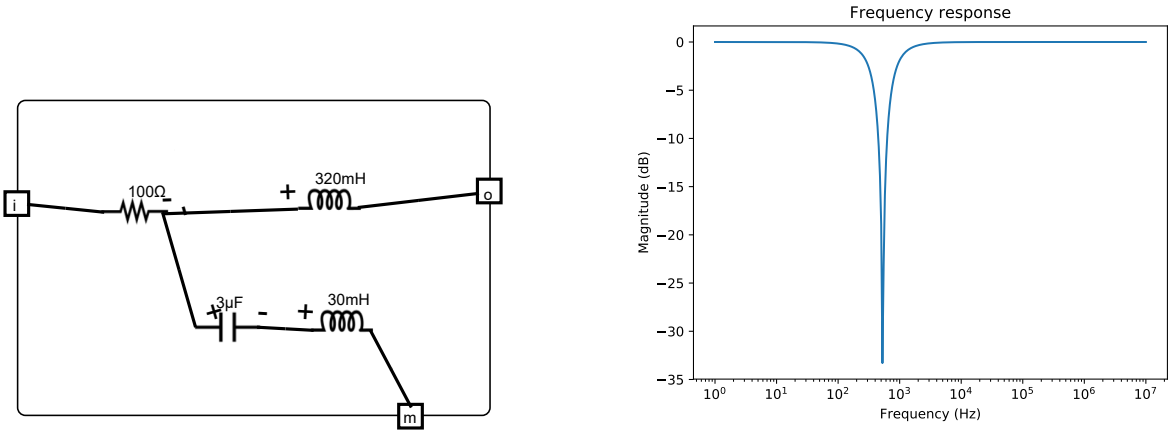

Figure 2: Instance of *EF* (left) and the Bode magnitude plot (right).

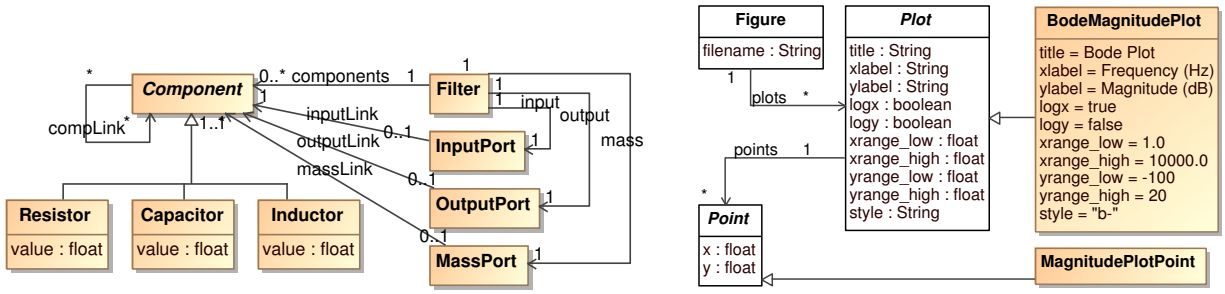

Figure 3: Metamodel of *EF* (left), and of its semantic domain BodeMag using the Plot template (right).

2.2 The *EF* DSL definition

The abstract syntax of *EF* is modelled as a metamodel consisting of classes, associations, attributes and inheritance links, as shown in Figure 3 on the left. A *Filter* can have multiple *Components*, which can be a *Resistor* (with *value* attribute for its resistance in ohm), *Capacitor* (with *value* attribute for its capacitance in farad) or *Inductor* (with *value* attribute for its inductance in henry). The *Component* class cannot be instantiated as it is an abstract class, denoted by italic font. A filter has exactly one *InputPort*, *OutputPort*, *MassPort*. Only one *Component* can be connected to an *InputPort* and *OutputPort*, but more than one *Component*s can be connected to the mass. The instance model of Figure 2 is a valid *EF* instance as it conforms to the metamodel.

The concrete syntax is modelled as a concrete syntax model, in which an icon or link type is defined for each language construct in the metamodel. The concrete syntax model is not shown due to space constraints. As previously stated, and as can be seen in the instance model of Figure 2, a *Filter* is displayed as a rountangle. *Resistor*s, *Capacitor*s and *Inductor*s are displayed using the ANSI standard notation, and their value is displayed above. Ports are displayed as boxes with an associated letter inside. *compLink* instances are displayed as lines, with a "-" on one side, and a "+" on the other side, to denote direction of the electric flow. *inputLink*, *outputLink* and *massLink* instances are plain lines. *components*, *input*, *output* and *mass* instances are based on the position of icons: if a *Component* is inside a *Filter*, it is part of the *Filter components*. Similarly, an *InputPort*, *OutputPort*, *MassPort* is connected to a *Filter* if it is on the edge of the filter icon.

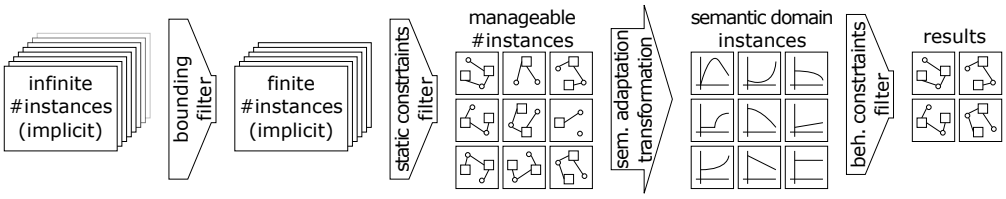

Figure 4: The conceptual filter and transformation phases of the DSE process.

As previously stated, the semantics of *EF* is a mapping to a Bode magnitude plot. The language of Bode magnitude plots can also be considered a DSL, called BodeMag, with its own abstract syntax, concrete syntax and semantics. In this case, we consider the semantics irrelevant, as we are interested in displaying instances of BodeMag. As plotting is a quite common semantic domain, we define BodeMag by inheriting from the Plot template, which includes visualisations on several platforms. The abstract syntax of BodeMag, including the Plot template, is shown in Figure 3 on the right. The Plot template is a generic metamodel for representing plots. It consists of a *Figure* (with a filename) that can have multiple abstract *Plot*s (with their plot parameters defined), that in turn has multiple plot *Point*s with *x* and *y* values. The domain-specific part of the BodeMag metamodel is shaded, and only includes a *Plot* subclass *BodeMagnitudePlot* that predefines the plot parameters, and a *Point* subclass *MagnitudePlotPoint*. The concrete syntax of BodeMag is its plot representation, which is defined for the Plot template for reusability. An instance of BodeMag can be seen in Figure 2 on the right, which displays all points on the plot and uses interpolation to form a line. The Plot template supports a data file representation for storing the exact data points. Theoretically, the semantic mapping is a transformation that transforms an instance of the *EF* DSL to an instance of the BodeMag DSL, as shown in Figure 2. In practice, the mapping is realised by transforming the *EF* instance to a SPICE net list, and running SPICE (Meares and Hymowitz 1988). This yields a file containing (frequency, magnitude) pairs which can be easily transformed to a BodeMag instance.

This description of *EF* can be seen as a typical way of using DSM, and serves as the starting point of our approach. Specifically, our approach is applicable to any DSL definition with an explicitly modelled semantic domain. Note that it is not mandatory to use the Plot template in the semantic domain of a DSL. However, additional support for the Plot template is provided by our approach.

The running example is modelled in AToMPM (Syriani, Vangheluwe, Mannadiar, Hansen, Mierlo, and Ergin 2013), a tool that supports DSM, i.e., metamodelling and model transformation. AToMPM can be considered a generic DSM tool, and we intend our approach to be applicable for other DSM tools as well.

3 APPROACH

In this section, the framework is explained. First, we show how the framework is used for the running example. This involves the use of a family of derived DSLs for modelling DSE constraints, and executing the DSE to obtain suitable models. Next, we explain how this family of derived DSLs can be generated (left part of Figure 1). Finally, we explain how a DSE tool is plugged into our framework, and how it can be used (right part of Figure 1).

3.1 Design-Space Exploration in Domain-Specific Modelling

The goal is to use DSE to generate "suitable" instance models of *EF*. These are models that are (a) valid and useful filter models, and (b) are a low-pass filter. The DSL definition itself, more specifically the DSL metamodel, represents the set of valid models, so in principle instances can be generated. Typically, the set of valid models that conform to the metamodel is infinite. However, since we use SAT (Boolean satisfiability) as DSE method, the design space needs to be bounded, i.e., finite. This subsection explains the DSE process, as shown in Figure 4.

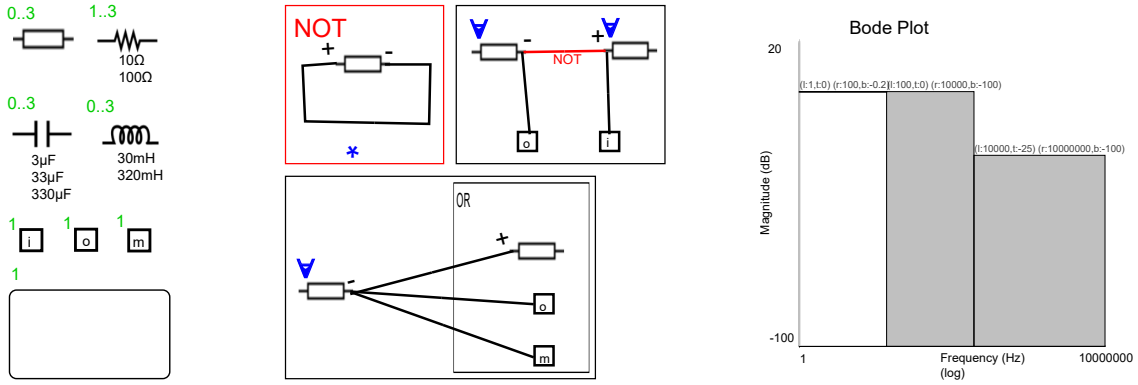

Figure 5: A bounding model in EF_{Bnd} (left), three static constraints in DSL_{SC} (centre), and a behavioural constraints model in DSL_{BC} (right).

Bounding model. Specifying the boundaries of the design space is the first step, in order to apply the bounding phase as shown in Figure 4. Our approach allows users to specify a maximum number of instances for each class in the DSL's metamodel. For *EF*, a bounding model is shown in Figure 5 on the left, specified in the bounding DSL EF_{Bnd} which will be defined in Section 3.2. For each class in the metamodel, the bounding model has exactly one instance, that states what the minimum and maximum number of instances of that class can be (stated in the icon's top left corner). In this example, we are looking for models of a single filter, with zero to three components, of which at least one resistor. Note the generic *component* representation, which is an abstract class in the metamodel. Furthermore, a list of possible values must be assigned to attribute value. In this case, we assume that we can make use of resistors of 10 Ω and 100 Ω, capacitors of 3 μF, 33 μF or 330 μF, and inductors of 30 mH or 320 mH.

Note that these bounding constraints could have been created using a programming language or a constraint language such as OCL. Writing such constraints often requires good knowledge of programming or logic, which contradicts the intent of DSM. In contrast, note how in our framework the bounding model reuses the DSL syntax. This is key to our approach, as this allows non-technical domain experts to create such a bounding model.

Now, it is technically possible to generate all possible instances using DSE, as the design space is finite. Then, the semantic mapping to SPICE can be executed and the results plotted for each generated instance, and a domain expert can select suitable models based on the plots. However, the number of instances grows quickly due to combinatorial explosion, so this typically leads to an infeasibly long total execution time of the semantic mapping and evaluation effort by the domain expert. During execution of the DSE process, our framework allows the domain expert to inspect how many DSL instances have been generated, and how many still have to be transformed to the semantic domain (a process that takes significantly longer than generating instances using DSE). Based on this, the domain expert can choose to abort the process and decide that more constraints should be defined. As shown in Figure 4, the design space remains implicit until static constraints are defined, which is the next step.

Static constraints model. Our framework allows the user to further constrain the design space using static (i.e., structural) constraints as shown in Figure 4. Structural constraints specify constraints on the structure of the models, i.e., how they should and should not be connected, that are more elaborate than the constraints imposed by the metamodel. The DSE process is typically executed (i.e., the instances become explicitly generated) after static constraints are defined. In our example, we look for static constraints that make sure only sensible filter models are valid:

- there should be no loops in the filter;

- if the outgoing signal of a component is connected to the output port, then the outgoing signal of the component cannot be connected to a component that is also connected to an input port;
- there should be no "dangling components": the output signal of a component should be connected, i.e., to another component, the output port or the mass port;
- analogously, the input signal of a component should be connected, i.e., to another component or the input port;
- a component cannot be connected to the mass port and the output port, as this will result in a potential difference of 0;
- a capacitor cannot be directly connected to the output port.

The first static constraint is shown at the top left of the three static constraints in the centre Figure 5, and is an instance of the static constraints DSL EF_{SC} which will be defined in Section 3.2. A constraint contains a pattern describing what should be or should not be matched in the model. In this case, the pattern is a *NACPattern* (negative application condition) meaning that a well-formed model should not contain a pattern instance. In particular, no component should be transitively (annotated with "*") connected to itself, meaning there should be no loops. Note how, once again, the familiar DSL syntax is used, enhanced with some new language constructs. The second constraint is shown at the top right of the three static constraints of Figure 5. It contains a pattern that *should* occur in the model for each pair of components (annotated with the "∀" symbol at the component icons) that is connected to an output port, respectively an input port: they should *not* be connected. Note that this pattern can be expressed, maybe more intuitively, as a *NACPattern*. The third constraint is shown at the bottom centre of Figure 5. It contains a pattern that *should* occur in the model for each component: it must be connected to another component, *or* to the output port, *or* to the mass port.

Such static constraints dramatically decrease the design space. Taking the static constraints into account, it should be feasible to generate all possible instances, simulate them and plot the results, and let a domain expert select which plots (and consequently which model) he or she prefers. Nevertheless, our framework supports behavioural constraints that enable automation of this selection process.

Behavioural constraints model. The final constraint we need to specify, is that we intend to find low-pass filters. This constraint will be specified as a behavioural constraint. Behavioural constraints are constraints that are not specified on the model structure, but on the model behaviour. In DSM, this behaviour is captured by the semantic mapping to a semantic domain. Hence, behavioural constraints are specified on the semantic mapping, as shown in Figure 4. This means that in our framework, evaluating behavioural constraints requires the model to be explicitly generated. The semantic mapping will be applied to each generated model and the behavioural constraints will be checked. Note that all models can be processed in parallel.

The general approach of our framework allows the user to specify behavioural constraints in a similar way as static constraints. For the *EF* DSL, one is able to specify a pattern that states that each *Point* with x value between 1 and 100 should have a y value between -0.2 and 0, and so on. In case the semantic domain derives from the Plot template however, the framework allows the domain expert to re-use the specific concrete syntax of the Plot template as shown in Figure 5 on the right. This model is an instance of EF_{BC}, which will be defined in Section 3.2. The "low-pass filter" constraint is specified on a plot, by adding rectangular regions that denote the valid regions for the plot. In particular, three rectangular regions are specified, that cover the entire frequency domain. For each region, the exact boundaries are displayed on top. Again, notice the intentional syntactic similarity to the plot in Figure 2. The behavioural constraints present the final filtering phase on the design space. All models that are found, are indeed low-pass filters. Note that the example of Figure 2 is not a low-pass filter according to the behavioural constraints presented here.

***EF* experiment**. In our experiment for finding a low-pass filter, we found 2198 instances after applying the bounding constraints and static constraints. Out of these 2198 instances, 63 were valid low-pass filters according to the behavioural constraint of Figure 5.

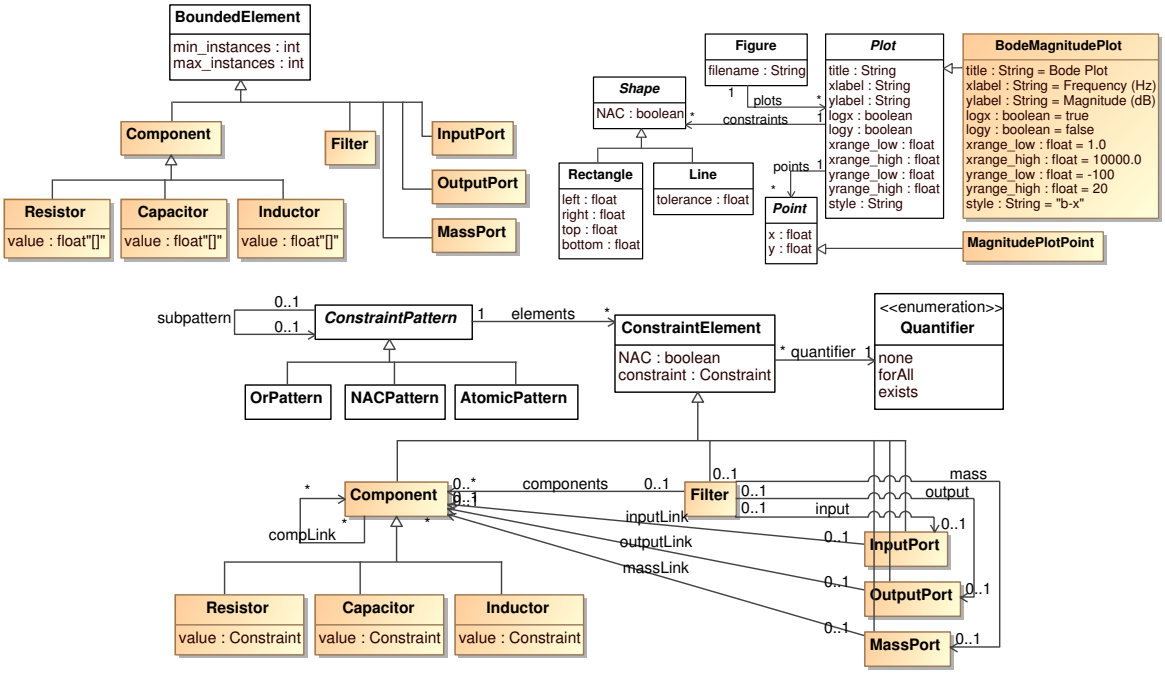

Figure 6: Generated metamodels of the family of DSLs: EF_{Bnd} (left), EF_{SC} (bottom), and EF_{BC} (top).

3.2 Language Generation

In this subsection it is explained how the three constraint DSLs are generated automatically from a given DSL definition, as introduced by the left part of Figure 1. Abstract syntax, concrete syntax and semantics must be generated for each DSL. Abstract and concrete syntax will be explained in this subsection whereas semantics, in the form of a semantic mapping to a DSE backbone, will be explained in the next subsection. The language generation approach is based on the ProMoBox approach (Meyers, Deshayes, Lucio, Syriani, Vangheluwe, and Wimmer 2014, Meyers, Wimmer, Vangheluwe, and Denil 2013, Deshayes, Meyers, Mens, and Vangheluwe 2014), a framework for generating property languages from a given DSL definition.

For the running example, the metamodel of each derived DSL is shown in Figure 6. The generated metamodel of each derived DSL consists of a DSL-independent template (the unshaded constructs), and DSL-dependent constructs (shaded) that have been changed. Because metamodels and concrete syntax models are models too, the generation is implemented as a model transformation in our framework. The remainder of this section presents the details of the generation process.

Bounding Language (DSL_{Bnd}). The following steps describe the automatic generation of the DSL_{Bnd}[1] metamodel, starting from a copy of the given DSL metamodel (Figure 3 on the left):

1. all classes subclass a new abstract *BoundedElement* class, that has *min_instances* and *max_instances* attributes;
2. all abstract classes become concrete;
3. all classes become singletons;
4. all associations are removed;
5. all attribute values with a type that represents an infinite domain (e.g., float, string, but not Boolean) become lists.

[1] With DSL_{Bnd} we refer to the bounding DSL for an unspecified given DSL; EF_{Bnd} is one example of a DSL_{Bnd}

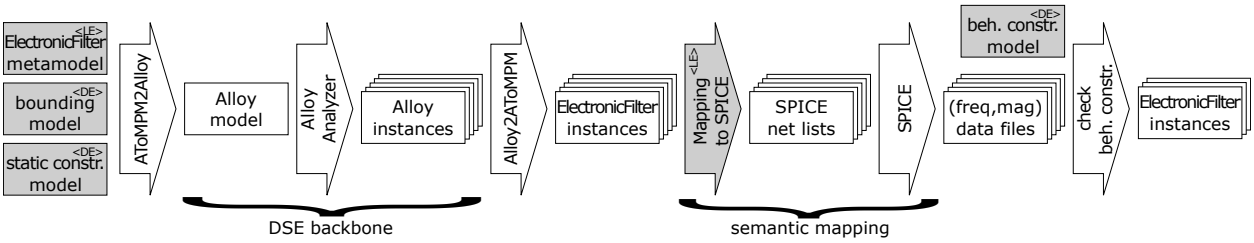

Figure 7: The DSE transformation chain for *EF*.

Using these steps, the EF_{Bnd} metamodel (top left of Figure 6) is generated from the *EF* metamodel (left of Figure 3).

The concrete syntax of EF_{Bnd} is generated by augmenting the representation of attribute values with a green field to denote the *min_instances* and *max_instances* values. Note how the bounding model shown in Figure 5 is an instance of EF_{Bnd}.

Structural Constraint Language (DSL_{SC}). To automatically generate the DSL_{SC} metamodel, starting from a copy of the given DSL metamodel, these steps must be followed:

1. all abstract classes become concrete. This allows e.g., Component to be used in a constraint;
2. all lower multiplicities of associations become 0. This allows for partial patterns;
3. all attribute types become conditions, i.e., expressions returning *true* or *false*;
4. all transitions are enriched with a *NAC* (which should be set to *true* if a transition should *not* occur) and *transitive* Boolean attribute;
5. the SCTemplate is added. This template consists of a number of pattern types that can be nested, and that can contain elements;
6. all DSL classes become subclasses of *ConstraintElement*.

Applying these steps to the *EF* metamodel (left of Figure 3) results in the EF_{SC} metamodel (bottom of Figure 6).

Similar to the DSL_{Bnd}, concrete syntax is altered so that icons for patterns are added, and a visualisation of *NAC* ("NOT" annotation for links and a cross for objects), *transitive* (by a "*" annotation) and a *quantifier* (by a "$\forall$" or "$\exists$" annotation) is added. Note how the model shown in Figure 5 on the left is an instance of EF_{SC}.

Behavioural Constraint Language (DSL_{BC}). As stated before, we leverage the use of the Plot template for defining the DSL_{BC}. Our framework has a dedicated *BCPlotTemplate* for the Plot template, which replaces the Plot template in the DSL_{BC} metamodel. The *BCPlotTemplate* includes a *Shape* class representing *constraints* on the *Plot*, which can be *Rectangle* or *Line*. Replacing the template of the BodeMag metamodel (right of Figure 3) results in the EF_{BC} metamodel (top right of Figure 6).

It is important to note that a language engineer can develop its own templates for the derived DSLs, effectively changing the expressiveness of newly generated languages. A hard condition for developing custom templates is that they can be mapped to the DSE backbone, which is explained in the next section.

3.3 Generic DSE Backbone

The complete transformation chain for executing DSE for the *EF* case is shown in Figure 7. This represents the process for the domain expert (right of Figure 1). Shaded artefacts are manually created, unshaded are provided by the framework or by another tool. The *EF* metamodel and semantic mapping must be created by the language engineer (annotated with "LE"), as explained in Section 2. While the languages DSL_{Bnd}, DSL_{SC} and DSL_{BC} are generated automatically (see Section 3.2), instances of DSL_{Bnd}, DSL_{SC} and DSL_{BC} are created by the domain expert (annotated with "DE"). Our approach maximises automation, as the remainder

```
1  one sig Filter { input : one InputPort, components : set Component, output : one OutputPort, mass : one MassPort }
2  some abstract sig Resistor extends Component {}
3    ...
4  fact srcMult_input { all tgt : InputPort | one src : Filter { tgt in src.input } }
5    ...
6  sig Resistor_10_, Resistor_100_ extends Resistor {}
7  sig Capacitor_3_, Capacitor_33_, Capacitor_330_ extends Capacitor {}
8  sig Inductor_30_, Inductor_320_ extends Inductor {}
9  fact nACPattern153 { all component152 : Component { component152 !in component152.^compLink } }
10 fact atomicPattern195 { all component199 : Component | all component198 : Component {
11     (component198 != component199) && (one inputPort203 : InputPort | one outputPort201 : OutputPort {
12         ((component198 in outputPort201.outputLink) && (component199 in inputPort203.inputLink))
13     } ) => (component199 !in component198.compLink) } }
14   ...
15 run {} for 3 Component, 4 Capacitor, 4 Inductor, 4 Resistor
```

Listing 1: The generated Alloy model for the *EF* example.

of the process is automatic. Also, all artefacts that need to be created are at the most appropriate level of abstraction.

We use Alloy (Jackson, Schechter, and Shlyakhter 2000) as a DSE backbone, a model finder (*Alloy Analyzer* in Figure 7) based on SAT. Alloy is suitable for our approach, as it can be used to generate structures that satisfy the constraints specified in the Alloy model. It implements a relational logic, inspired by Z and modelling languages. To use Alloy as DSE backbone, we implemented two transformations: *AToMPM2Alloy* and *Alloy2AToMPM*.

AToMPM2Alloy. AToMPM2Alloy is a transformation from metamodel, DSL_{Bnd} model and DSL_{SC} model to an Alloy model. We illustrate AToMPM2Alloy by showing its result for the running example in Listing 1. The *EF* metamodel is transformed to signatures (`sig` constructs) as shown on line 1-2, and attributes and associations are transformed to relations in the `sig` body, with the appropriate multiplicity. As shown on line 4, source multiplicities of associations are transformed to constraints, i.e., (`fact`s). The DSL_{Bnd} model is transformed to (1) multiplicity markings where possible (e.g., `one sig` on line 1, `some sig` on line 2), (2) upper limits in the `run` statement at line 15, or (3) for attribute value bounds, signatures extending metamodel class signatures to model instances with each possible attribute value as shown on line 6-8. The DSL_{SC} model is transformed to `fact` constructs (line 9-13), which reflect the hierarchical structure of the static constraint. The first static constraint of Figure 5 is shown on line 9 (notice the transitivity operator `^`). The second static constraint of Figure 5 is shown on line 10-13, which represents a precondition (line 11-12), and the actual constraint following `=>` on line 13. Although we think Alloy has an intuitive syntax, we feel that it is not easy for a non-technical expert to use the Alloy language, and that it is preferable to specify constraints in a language that is familiar to them (i.e., the Static Constraints DSL).

Alloy2AToMPM. The instances that are generated by the Alloy Analyzer are represented as xml-files. These files contain all information of the Alloy model that represents the metamodel (i.e., which signatures exist and which relations they contain), which can be easily parsed and transformed back to AToMPM in a metamodel-generic way. Note that the resulting DSL instance in AToMPM does not contain information about icon location on the canvas, so all icons will have the same canvas coordinate. Layout algorithms can assist the domain expert in arranging the icons.

Behavioural constraint checking. As shown in Figure 7, the framework implements a behavioural constraint checking transformation that takes the DSL_{BC} model into account. In the general case, i.e., if the Plot template is not used, behavioural constraint checking is done by checking for matches in the generated semantic domain model of the patterns that are specified in the DSL_{BC} model. For example, all points in the *BodeMagnitudePlot* with x between 1 and 100 must have a y between 0 and -0.2. Pattern matching is a built-in feature of AToMPM. If the Plot template is used, then the constraints are retrieved from the *Rectan-*

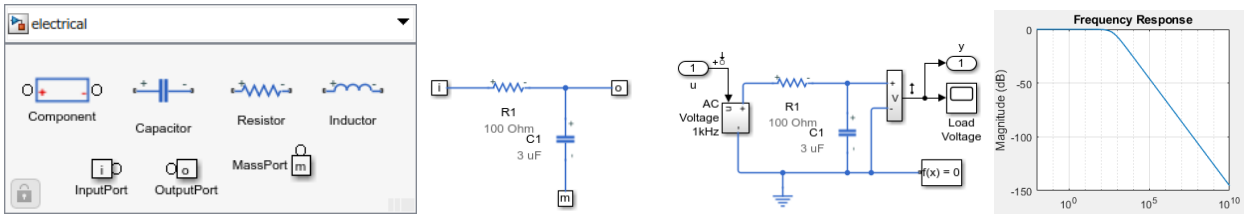

Figure 8: The *EF* in Simulink, with the metamodel as library (left), a filter instance (centre left), the filter transformed to a full circuit (centre right) and the Bode plot for that circuit (right).

*gle*s and *Line*s on the *Plot*. Then, a Python script reads the date from the data files and checks each parsed point against these constraints.

4 EVALUATION: APPLICATION TO SIMULINK

Our framework is applicable to any DSL that is explicitly defined, and has an explicitly defined semantic domain. In this section, we illustrate the validity of this claim by applying the framework to Simulink.

Simulink DSL. Similar to our approach in AToMPM, we use a DSL as starting point. In the context of Simulink, the DSL takes the form of (a) a Simulink library (i.e., the abstract and concrete syntax), and (b) two simulation scripts to obtain the Bode magnitude plot points from a model using this library (i.e., the semantic mapping). Figure 8 shows the custom made Simulink library representing the DSL on the left. Masks are specified for blocks, to obtain the correct interface (i.e., metamodel attributes) and concrete syntax. In particular, *Component*, *InputPort*, *OutputPort* and *MassPort* are created by masking *Subsystem* blocks. An instance using this library is shown at the centre left. The semantic mapping involves simulating this model. In order to do so, it needs to be transformed to a model that includes a source, ground, voltage centre etc. Transformation can be achieved with a Matlab script using commands `add_block()`, `delete_block()`, `add_line()`, `delete_line()`. The result of this transformation (which can be expressed using a Matlab script) is shown at the centre right. A second simulation script simulates the transformed model, and plots the results. If a Simulink DSL is specified in this way, DSE support can be automatically generated using our framework. This includes support for modelling a bounding model, static constraints model, and behavioural constraints model, and DSE execution support resulting in a set of Simulink models.

Language generation. The library for DSL_{Bnd} and DSL_{SC} is generated by our framework, and it turns out that for the Simulink case, support for the DSL_{BC} can be reused instead of generated. Our approach includes a generic transformation from Simulink libraries to AToMPM (meta)models called *Simulink2AToMPM*. The derived DSLs are generated in AToMPM as explained in Section 3.2 and are transformed back to Simulink libraries using a generic transformation *AToMPM2Simulink*. With these generic transformation, no effort is required from the language engineer to support DSE for any Simulink library. *Simulink2AToMPM* and *AToMPM2Simulink* make use of the Simulink library (e.g., `new_system(libname, 'Library')` for creating a new library), and automatically reads/edits the masks using the `Simulink.Mask` class, according to Section 3.2[2]. The result is shown in Figure 9. Ports have disappeared in the bounding DSL. The added attributes *min_instances* and *max_instances* in the DSL_{Bnd}, and *NAC* and *quantifier* in the DSL_{SC} are visualised as block annotations. The top row of the DSL_{SC} represents the generic SCTemplate. The derived DSLs consist of classes only, as Simulink does not provide support for defining new signals (i.e., associations) or new connectivity rules. Nevertheless, the *nac* and *transitive* attribute for all associations in the DSL_{SC} are modelled using new blocks (top right) that can be used to annotate line segments. The DSL_{BC} is generic for all Simulink DSLs that have a semantic mapping to a plot. A DSL_{BC} instance can be specified by adding rectangles to an empty plot, either by using the command *rectangle*, or by graphically drawing

[2]Note that the generation of derived DSLs can be re-implemented for Simulink as a Matlab script, bypassing AToMPM.

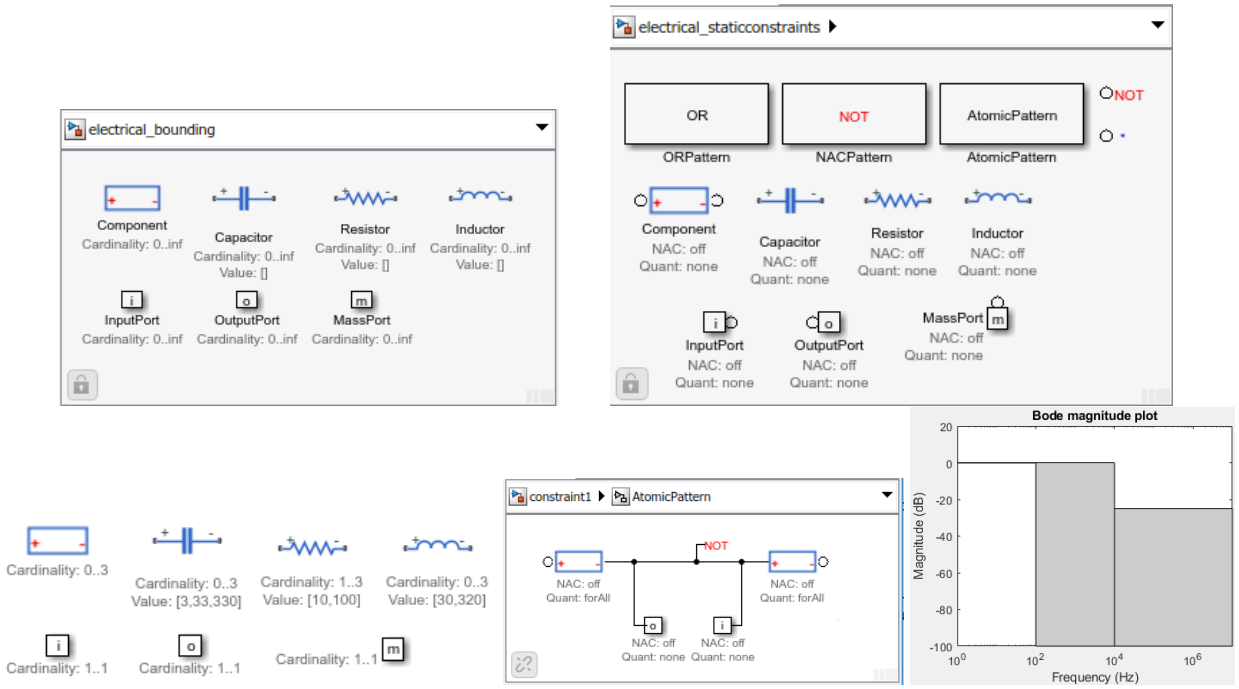

Figure 9: The generated Simulink libraries EF_{Bnd} (top left) and DSL_{SC} (top right), and the EF_{Bnd} instance (bottom left), a DSL_{SC} instance (bottom centre), and a DSL_{BC} instance (bottom right).

rectangles on the plot. In the latter case, a coordinate transformation must be executed to find the actual plot coordinates of the rectangle. An instance model of each of these languages is shown in Figure 9.

5 RELATED WORK

Related work can be found in Search-Based Software Engineering (SBSE) and in Model-Driven DSE. SBSE solves software engineering problems using Search-Based optimisation (SBO). An example of the use of models and search can be found in (Kessentini, Wimmer, Sahraoui, and Boukadoum 2010). The authors search for a model transformation to translate a sequence diagram into a coloured Petri net. Simulated annealing as well as Particle Swarm Optimizations are used to search in the large design-space of such a problem. The authors use this experience in (Kessentini, Langer, and Wimmer 2013) to create a framework for using genetic algorithms with models. A generic encoding metamodel is proposed as well as the use of model transformations for encoding and decoding the domain specific models. Another approach is proposed in (Burton, Paige, Rose, Kolovos, Poulding, and Smith 2012). The authors introduce a MDE solution to solving acquisition problems. Model transformations are used to create an initial population for a genetic algorithm and to evaluate candidate solutions. Finally, evolutionary algorithms have been used before to search for optimized models (Williams, Poulding, Rose, Paige, and Polack 2011). Our approach does not require the user to create a transformation to another representation, as this is provided by our framework.

6 CONCLUSION AND FUTURE WORK

We introduced a framework to automatically enable DSE for a any given DSL, effectively pulling up DSE to the DSL level. We have implemented the framework in the DSM tool AToMPM. We have illustrated the genericity of our approach by applying the framework to Simulink, thus allowing exploration of models in Simulink with a minimal effort. This clearly complements the existing DSE in Simulink in the form of parameter estimation support, as block topologies can be explored with our approach.

Interesting future work includes firstly the support of an optimisation function to find the "best" candidate among found models. For instance, in our example, such a soft constraint could be to minimise the number of components used in a filter. Secondly assessing and improving performance. This is however largely dependent of our DSE backbone, i.e., Alloy. Therefore, different DSE backbones need to be explored and compared. In this respect, we have experimented with a mapping to Gecode[3], and with a model transformation approach to control the design space exploration. Thirdly, although we have limited support for assessing a lower bound on the remaining execution time, it would be interesting to further elaborate on this, e.g., by estimating an upper bound. This would be an enabler for an iterative, phased DSE design process, where the user gradually strengthens the constraints for the DSE depending on the feedback of the execution time: maybe a bounding model and static constraints model is sufficient, or maybe some static constraints do not need to be modelled to obtain a manageable number of suitable candidates? Fourthly, it is interesting to investigate whether we can map the DSE to a smooth design space. This way, local optima can be exploited, thus greatly improving performance compared to Alloy's brute force search technique. In our opinion, it is required to include search operations, e.g., by using transformation rules, and include domain knowledge that entails a search plan. Related to this, interesting future work involves the use of behavioural constraints during this controlled DSE.

REFERENCES

Burton, F. R., R. F. Paige, L. M. Rose, D. S. Kolovos, S. M. Poulding, and S. Smith. 2012. "Solving Acquisition Problems Using Model-Driven Engineering". In *ECMFA 2012*, Volume 7349 of *Lecture Notes in Computer Science*, pp. 428–443.

Deshayes, R., B. Meyers, T. Mens, and H. Vangheluwe. 2014. "ProMoBox in Practice : A Case Study on the GISMO Domain-Specific Modelling Language". In *MPM@MODELS 2014*, Volume 1237 of *CEUR Workshop Proceedings*, pp. 21–30.

Jackson, D., I. Schechter, and I. Shlyakhter. 2000. "Alcoa: the Alloy constraint analyzer". In *ICSE 2000*, pp. 730–733, ACM.

Jackson, E. K., E. Kang, M. Dahlweid, D. Seifert, and T. Santen. 2010. "Components, platforms and possibilities: towards generic automation for MDA". In *EMSOFT 2010*, pp. 39–48, ACM.

Kelly, S., and J. Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley.

Kessentini, M., P. Langer, and M. Wimmer. 2013. "Searching models, modeling search: On the synergies of SBSE and MDE". In *CMSBSE@ICSE 2013*, pp. 51–54, IEEE Computer Society.

Kessentini, M., M. Wimmer, H. A. Sahraoui, and M. Boukadoum. 2010. "Generating transformation rules from examples for behavioral models". In *BM-MDA@ECMFA 2010*, pp. 2, ACM.

Meares, L. G., and C. E. Hymowitz. 1988. *Simulating with SPICE (Simulation Program with Integrated Circuit Emphasis)*. San Pedro, CA, Intusoft.

Meyers, B., R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer. 2014. "ProMoBox: A Framework for Generating Domain-Specific Property Languages". In *SLE 2014*, Volume 8706 of *LNCS*, pp. 1–20, Springer.

Meyers, B., M. Wimmer, H. Vangheluwe, and J. Denil. 2013. "Towards domain-specific property languages: the ProMoBox approach". In *DSM@SPLASH 2013*, pp. 39–44, ACM.

Schmidt, D. C. 2006. "Guest Editor's Introduction: Model-Driven Engineering". *IEEE Computer* vol. 39 (2), pp. 25–31.

Sen, S., B. Baudry, and H. Vangheluwe. 2010. "Towards Domain-specific Model Editors with Automatic Model Completion". *Simulation* vol. 86 (2), pp. 109–126.

Sendall, S., and W. Kozaczynski. 2003. "Model Transformation: The Heart and Soul of Model-Driven Software Development". *IEEE Software* vol. 20 (5), pp. 42–45.

Syriani, E., H. Vangheluwe, R. Mannadiar, C. Hansen, S. V. Mierlo, and H. Ergin. 2013. "AToMPM: A Web-based Modeling Environment". In *Demonstrations@MODELS 2013*, Volume 1115 of *CEUR Workshop Proceedings*, pp. 21–25, CEUR-WS.org.

Williams, J. R., S. M. Poulding, L. M. Rose, R. F. Paige, and F. A. C. Polack. 2011. "Identifying Desirable Game Character Behaviours through the Application of Evolutionary Algorithms to Model-Driven Engineering Metamodels". In *SSBSE 2011*, Volume 6956 of *Lecture Notes in Computer Science*, pp. 112–126, Springer.

Zeng, H., and M. D. Natale. 2010. "Improving Real-Time Feasibility Analysis for Use in Linear Optimization Methods". In *ECRTS 2010*, pp. 279–290, IEEE Computer Society.

[3] http://www.gecode.org/