Towards the generation of graphical modelling environments aided by patterns

Antonio Garmendia, Ana Pescador, Esther Guerra, and Juan de Lara

Modelling and Software Engineering research group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Abstract. Model-Driven Engineering (MDE) promotes the use of models to conduct all phases of software development in an automated way. Such models are described using Domain Specific Modelling Languages (DSMLs). While the definition of DSMLs and their supporting environments are recurring activities in MDE, they are mostly developed ad-hoc from scratch. This paper proposes the use of patterns to describe the abstract and concrete graphical syntax of DSMLs, and to automate the generation of a graphical modelling environment for them.

Keywords: Model-Driven Engineering, Domain Specific Languages, Patterns, Graphical Modelling Environments

1 Introduction

Model-Driven Engineering (MDE) promotes a model-centric approach for software development, where models are used to specify, design, test, simulate and generate code for applications. While models can be described using general-purpose modelling languages, like UML, it is frequent the use of Domain Specific Modelling Languages (DSMLs) focussed on the particularities of a domain [8].

Hence, the creation of DSMLs is recurrent in MDE, for which one needs to describe their abstract and concrete syntax, their semantics, and developing a suitable modelling environment for them. Although there are software frameworks to ease the development of textual and graphical environments [8, 10, 12], the creation of DSMLs is mostly an ad-hoc process lacking the ability to build on existing knowledge coming from the construction of similar DSMLs.

To simplify the creation of DSMLs, we propose their assisted construction by means of patterns. In particular, *domain patterns* describe recurring concepts common to a domain, and *concrete syntax patterns* gather standard representation options for DSMLs and enable the synthesis of modelling environments. As a proof of concept, we show a prototype implementation for Eclipse.

The remaining of the paper is organized as follows. First, Section 2 introduces our approach. Then, Section 3 explains how to build graphical DSMLs with patterns. Next, Section 4 reviews related works, and finally, Section 5 concludes.

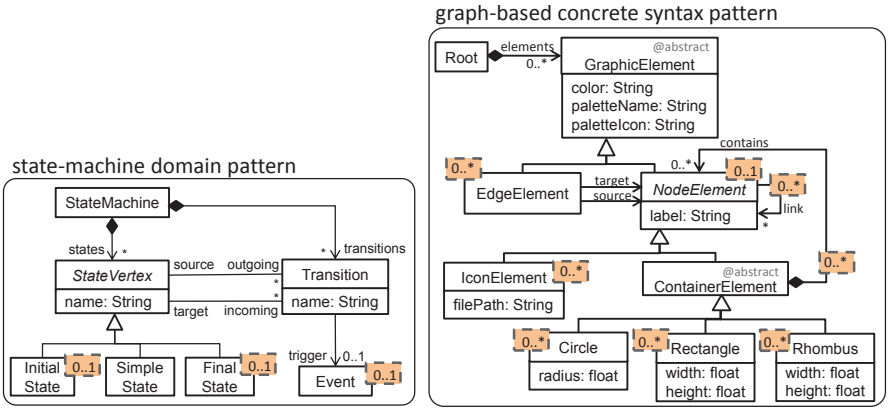

Fig. 1: Domain pattern (left). Graph-based concrete syntax pattern (right).

2 Overview

The design of a DSML encompasses several aspects, including abstract syntax, concrete syntax, and semantics. In addition, editing DSML models is usually performed using a dedicated environment providing services like model persistence, conformance checking, and others more advanced. We propose the use of patterns to address all these aspects, in order to facilitate and speed up their definition. By lack of space, we focus on patterns dealing with the abstract and concrete syntax, as well as the generation of modelling environments from them.

To deal with the abstract syntax, we propose *domain patterns*, gathering typical requirements of similar languages within a domain, and documenting their variability. Here, there may be patterns for workflow languages, arithmetical or logical expressions, variants of state machines, query languages, and component-based architectural languages, among others. A DSML may use several domain patterns, customized for a given need, and extended with other domain-specific concepts. These patterns may help to build a DSML more quickly and trustworthily, in a constructive way. As an example, Fig. 1 (left) shows a simplified domain pattern for state machines. Pattern elements have a cardinality, which governs how many times they can be instantiated (1 if no cardinality is specified). For instance, any application of the state-machine pattern should have one SimpleState, while it may lack InitialState and FinalState.

On the other side, *concrete syntax patterns* characterize families of similar representations [2], like textual, graphical, tabular or form-based. In the case of a graphical syntax, aspects like layouting or zooming may be configured. Moreover, concrete syntax patterns can be used to automate the generation of editors supporting the defined syntax (which otherwise should be implemented by hand), and can be attached to domain patterns in order to define different default visualization options for them. As an example, Fig. 1 (right) shows a simplified pattern for graph-based representation. This pattern permits assigning graphical elements (Circles, Rectangles, etc.) to elements in the DSML meta-model.

Altogether, in order to define DSMLs, we propose a reutilization-based, pattern-centric approach, which we have implemented in our prototype tool DSL-tao (http://jdelara.github.io/DSL-tao/). DSL-tao enables the construction of meta-models, where some meta-model parts can be defined through the application of existing patterns in a repository. Basic pattern application is performed in three steps. First, a pattern is selected and a wizard guides the designer in its application (see window 1 in Fig. 2 for the wizard of the state-machine pattern). In this step, variants and attached patterns can also be selected (see next section). Then, the designer can bind meta-model elements to pattern elements. Finally, the unbound pattern elements are automatically created new in the meta-model, annotated with their participant role in the pattern (window 2 in Fig. 2).

The next section presents two ways to describe and generate graphical environments for DSMLs using patterns.

3 Defining graphical DSMLs through patterns

We propose two ways to describe the graphical syntax of a DSML. In the first one, domain patterns have attached a default visualization, which the DSML designer just reuses. This option profits from commonly agreed means to represent domain patterns (e.g., state machines, or component-based systems). In the second option, a dedicated wizard is used to apply a graphical pattern over the elements of the DSML meta-model. This approach is to be used when the DSML needs a non-predefined, or special syntax. Next, we present these two possibilities, as well as the environments generated through their use.

3.1 Using the visual syntax attached to domain patterns

Domain patterns may have attached concrete syntaxes, accounting for typical representations of the domain concepts. For instance, Fig. 2 shows the application of the state-machine pattern. The pattern has three concrete syntax patterns attached: one for the standard graph-based representation, another for its representation as tables, and another using forms. Designers can select one of them. In this way, when the domain pattern is applied, the concrete syntax pattern will be automatically instantiated as well. Thus, this approach permits predefining a set of concrete visualizations, which can be reused "as is" by DSML designers.

3.2 Using the dedicated custom wizard

Sometimes, the designer requires a fine grained control of the concrete syntax for the DSML, or he has not used domain patterns with attached concrete syntax. In such cases, the designer can still use a concrete syntax pattern to automate the generation of a modelling environment, for which he needs to map meta-model elements to concrete representations in the selected concrete syntax pattern. Since the application of concrete syntax patterns has many specificities

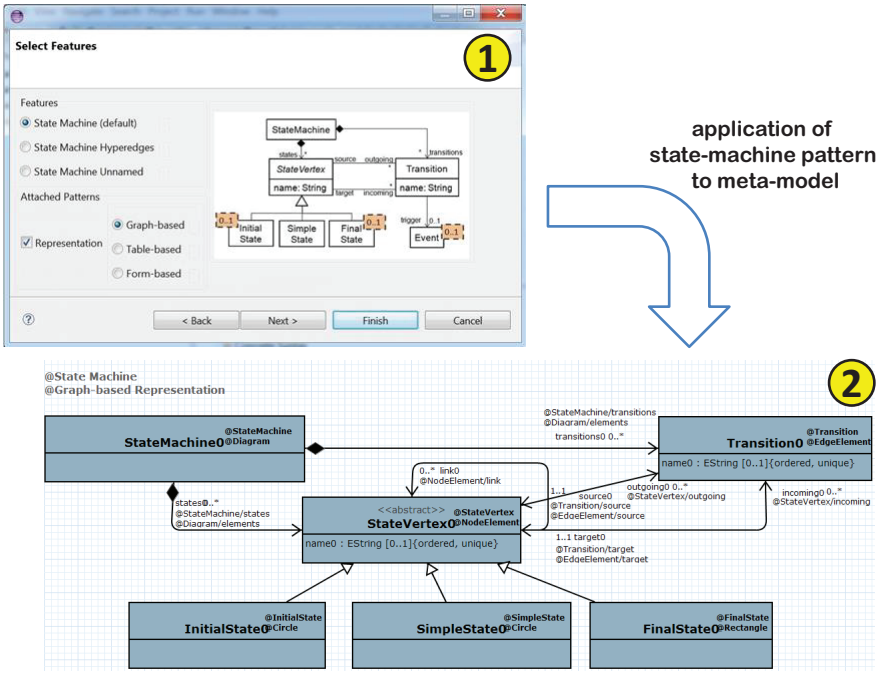

Fig. 2: The wizard for pattern application (1). Applied pattern (2).

(like selecting figures for nodes and decorations for edges), patterns may provide dedicated wizards for their application. For instance, the graph-based concrete syntax pattern has a customized wizard that implements heuristics to decide which classes will be represented as nodes, which ones as edges, the attributes to display, and the nodes that are containers of other nodes. Then, the designer can refine the inferred concrete syntax and fine-tune the visual representation for nodes and edges.

Fig. 3 shows the wizard to customize the following heuristics:

- *Root strategies*: These are alternatives to select the root class to be used in diagrams. The root class is usually a class that contains all elements of the model, directly or indirectly. The strategy *Contains more classes* counts how many classes contain each class, and selects the one that contains more. The strategy *Class with no parents* suggests classes that are not contained in other classes. Both strategies are based on the tree of containment references defined in the meta-model. The last strategy (*Modularity pattern*) selects as root the meta-model classes annotated as Unit by a modularity pattern [5] (not shown in this paper) that allows organizing models in a modular way.
- *Label selection*: These heuristics are used to decide the data that node-like classes will display close to the node representation. The strategy *First string attribute* displays the first string attribute of the class, and *Identifier of the class* its identifier. The strategy *Parameter string attribute* receives several input strings, and selects the attribute whose name contains some of them.

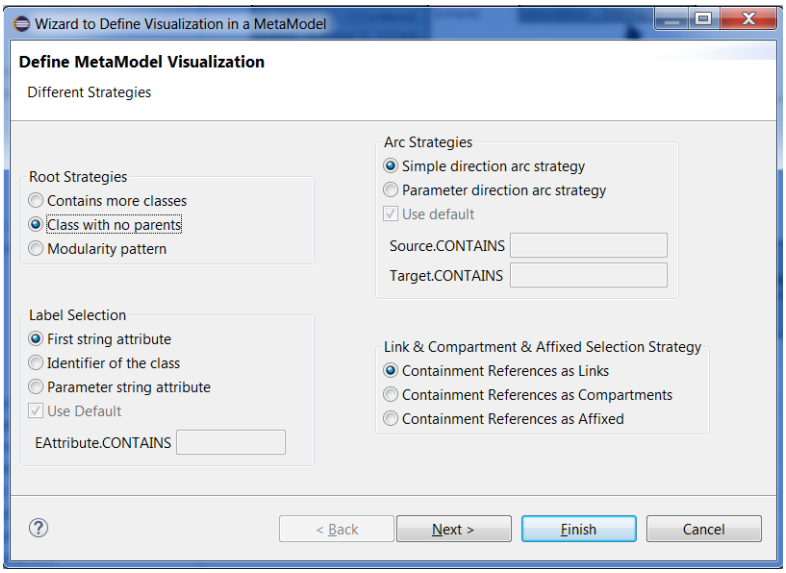

Fig. 3: Dedicated wizard for assigning a graph-based concrete syntax. Step 1: selection of heuristics.

- *Arc strategies*: They are used to select edge-like classes. In this case, we select the classes that define two non-containment references with lower bound `0` or `1`, and upper bound `1`. These two references will be mapped to the source and target of the edge representation for the class. While the first strategy (*Simple direction arc strategy*) selects the source and target references randomly, the second one (*Parameter direction arc strategy*) takes into account possible naming conventions (e.g., `source` or `src` for the source reference).
- *Link & compartment & affixed selection strategies*: These strategies identify the references that will be displayed graphically as edges, compartments or affixes. If the strategy *Containment references as links* is selected, all containment references will be represented as links. If the selected strategy is *Containment references as compartments*, they will be displayed as containers for the objects conformant to the type of the reference. Finally, if the chosen strategy is *Containment references as affixed*, the nodes will be placed on the border of another element.

The wizard uses the heuristics to infer the optimal concrete representation of meta-model elements, which are proposed to the designer in a second step (see Fig. 4). The, the designer is allowed to modify the inferred syntax, as well as fine-tune the concrete visualization for nodes and edges to customize the decorations for the start and end of edges, the types of figures for nodes, their size and colour. This last step is shown in Fig. 5.

Finally, although we have presented the wizard for the graph-based concrete syntax pattern, the same idea could be used to implement further strategies

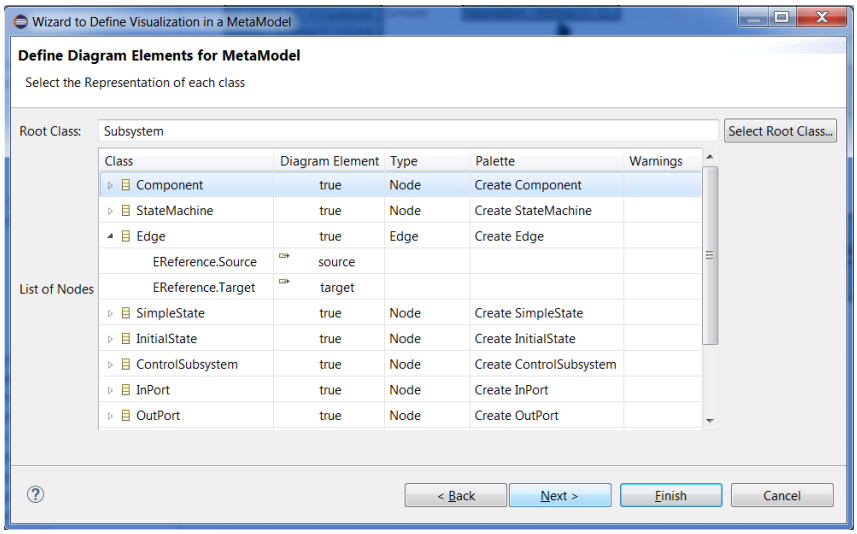

Fig. 4: Dedicated wizard for assigning a graph-based concrete syntax. Step 2: customization of inferred concrete syntax.

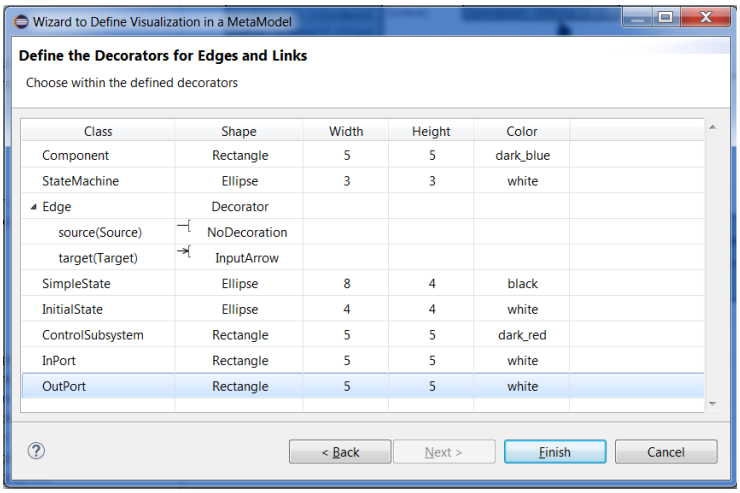

Fig. 5: Dedicated wizard for assigning a graph-based concrete syntax. Step 3: customization of appearance of nodes and edges.

for this or other concrete syntax patterns. Currently, we support tabular and form-based representations, in addition to graph-based ones.

3.3 The generated graphical environment

The modelling environment for a DSML can be synthesized from its meta-model. For this purpose, DSL-tao invokes the code generators of the services associated

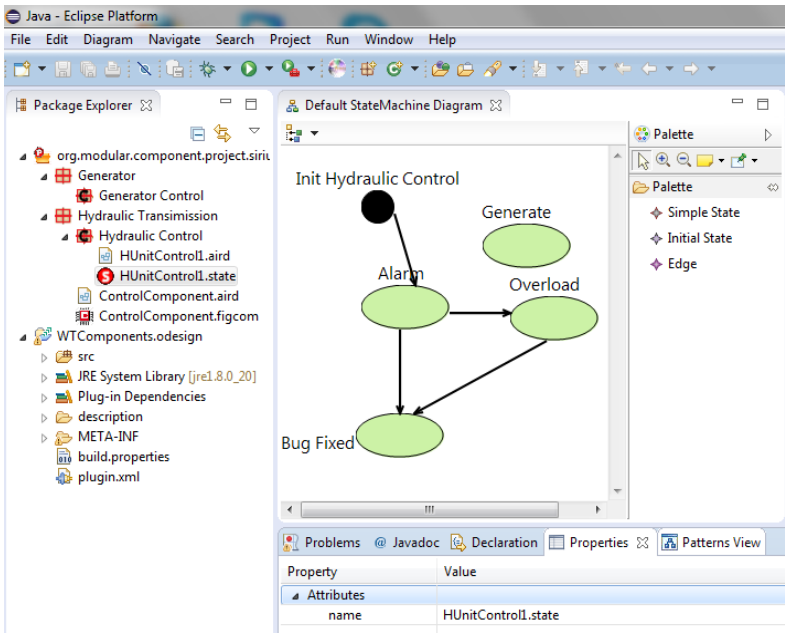

Fig. 6: Generated graphical modelling environment.

to the applied patterns. For graphical concrete syntax patterns, the generator creates an Eclipse plugin that uses the Sirius graphical framework [10] as back-end. Thus, once the meta-model is annotated with the concrete syntax pattern, a Sirius *.odesign* model is generated. This model describes the shapes for nodes, the style for edges, the mappings of graphical elements to meta-model elements, the elements in the palette, and the actions to be performed when palette elements are invoked. Technically, this model is created using a model transformation. Then, the Sirius model is packaged in a plugin, which is contributed to the modelling environment of the DSML.

Fig. 6 shows the generated graphical environment for the meta-model shown at the bottom of Fig. 2, which was created by instantiating the default concrete syntax pattern attached to the domain pattern for state machines.

4 Related work

There are many tools to develop graphical modelling environments for different applications, like meta-CASE tools [8], diagram sketching [3] or multi-formalism modelling and simulation [4]. The advent of Eclipse has promoted frameworks to construct visual editors as plugins, like Tiger [1], GMF [6], Eugenia [9], Spray [11], Graphiti [7], or Sirius [10]. All these tools are model-based, except Graphiti which provides a Java API for coding. Some generate artefacts for other lower-level approaches, like Eugenia which is built atop GMF, and Spray atop Graphiti. In our case, DSL-tao produces graphical editors based on Sirius. All

frameworks use code generation except Sirius, which is interpreted. The way of specifying the concrete syntax varies: Eugenia requires annotating the metamodel elements, Spray uses a textual DSL, GMF and Sirius require building models that describe the concrete syntax, and Graphiti requires programming. Our approach is closer to Eugenia, as our pattern applications result in metamodel annotations. However, our domain patterns can attach concrete syntax styles, which speeds up the generation of graphical environments. This feature is unique among the mentioned tools.

5 Conclusions and future work

We have presented a pattern-based approach to the development of graphical DSMLs. The approach is supported by a tool that permits applying patterns from a repository and the automatic generation of a modelling environment. We are currently working on defining new patterns, and developing further services for graphical environments like support for layers and abstractions.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitivity (TIN2011-24139 and TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

References

1. E. Biermann, K. Ehrig, C. Ermel, and G. Taentzer. Generating Eclipse editor plug-ins using Tiger. In *AGTIVE*, volume 5088 of *LNCS*, pages 583–584. Springer, 2007.
2. P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In *VL/HCC*, pages 83–90, 2004.
3. F. Brieler and M. Minas. A model-based recognition engine for sketched diagrams. *J. Vis. Lang. Comput.*, 21(2):81–97, 2010.
4. J. de Lara and H. Vangheluwe. Atom^3: A tool for multi-formalism and metamodelling. In *FASE*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.
5. A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara. EMF splitter: A structured approach to EMF modularity. In *XM@MoDELS*, volume 1239 of *CEUR*, pages 22–31. CEUR-WS.org, 2014.
6. GMF. `https://wiki.eclipse.org/Graphical_Modeling_Framework`.
7. Graphiti. `http://eclipse.org/graphiti/`.
8. S. Kelly and J. Tolvanen. *Domain-specific modeling - enabling full code generation*. Wiley, 2008.
9. D. S. Kolovos, L. M. Rose, S. bin Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In *MODELS*, volume 6394 of *LNCS*, pages 211–225. Springer, 2010.
10. Sirius. `https://eclipse.org/sirius/`.
11. Spray. `https://code.google.com/a/eclipselabs.org/p/spray/`.
12. Xtext. `http://www.eclipse.org/Xtext/`.