Xdiagram: A Declarative Textual DSL for Describing Diagram Editors (Tool Demo [*])

André L. Santos Eduardo Gomes

Instituto Universitário de Lisboa (ISCTE–IUL), ISTAR–IUL
Av. Das Forças Armadas, Edifício II,1649-026 Lisboa, PORTUGAL

andre.santos@iscte.pt
santana.eduardo@gmail.com

Abstract

When compared to the realm of textual syntax, developing graphical syntax for a domain-specific modeling language (DSML) is still challenging. Xdiagram is a research prototype that consists of a textual domain-specific language (DSL) for specifying diagrammatic representations against abstract syntax defined in the Eclipse Modeling Framework (EMF). Specifications are written in Xdiagram against such models in a declarative fashion in order to obtain a diagram editor. We explain the main primitives of our DSL using conceptual modeling as an example domain for illustration.

Categories and Subject Descriptors Software notations and tools [*Development frameworks and environments*]: Integrated and visual development environments; Software notations and tools [*Context specific languages*]: Visual languages

General Terms Design, Languages

Keywords DSLs, graphical syntax, diagrams, language workbenches, EMF

1. Introduction

A domain-specific modeling language (DSML) (Kelly and Tolvanen 2008) is typically engineered with resort to a metamodel that defines the abstract syntax of the language, and one or more forms of concrete syntax that materialize the concepts described in the metamodel. Our work focuses on graphical syntax, where DSMLs are manipulated through diagrams with specific editors. The goal of our research is to explore the design space of declarative DSLs based on textual primitives for specifying graphical syntax, aiming at achieving a suitable abstraction to address the problem of translating notational requirements into working diagram editors.

We developed a research prototype that we refer to as Xdiagram, comprised of a textual DSL and an interpreter to provide diagram editors from the textual specifications. The DSL has a declarative style, comprising primitives that bind metamodel elements to diagrammatic elements. Our strategy of using a textual DSL to express the mappings is grounded on the simplicity of terse textual declarations to describe mappings to metamodel elements, as well as notational aspects and their visibility/layout constraints. We developed the language iteratively, using several widely-used modeling languages as case studies (e.g., class diagrams, statecharts, feature models) in order to assess the suitability and usability of our language primitives.

Xdiagram is based on Eclipse technologies, supporting EMF (Steinberg et al. 2009) as the metamodeling abstraction. The Xdiagram DSL is developed with Xtext (Eclipse 2016d), whereas the diagramming support is developed on top of the Graphiti framework (Eclipse 2016b). Our tool is based on model interpretation, rather than code generation. The specifications are interpreted at runtime and the diagram editors are instantiated dynamically, allowing domain engineers to perform quick iterations on experimentation of graphical syntax alternatives. Our tool is suitable for iterative user-centered design (Barisic et al. 2012) of EMF-based DSMLs with graphical notation.

2. Related Work

In the realm of Eclipse-based technologies, EMF provides an infrastructure for developing and using metamodels compliant with OMG's MOF standard (OMG 2006). With respect to the development of concrete syntax for EMF models, several language workbenches have emerged for addressing both *textual* (e.g., Xtext (Eclipse 2016d), EMFText (Dev-

[*] A prototype implementation of Xdiagram is available at: `github.com/andre-santos-pt/xdiagram`

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4447-0/16/10...$15.00
http://dx.doi.org/10.1145/2997364.2997377

Boost 2016)) and *graphical* (e.g., GMF (Eclipse 2016a), EuGENia (Kolovos et al. 2010), Sirius (Eclipse 2016c)) concrete syntax. Despite the numerous available technologies, there is a general perception based on anecdotal evidence, that the development of graphical syntax for EMF models is still a challenging endeavour, especially when compared to the development of textual syntax. JetBrains MPS (JetBrains 2016) for textual notation, and MetaEdit (MetaCase 2016) for graphical notation, are two examples of industry-strength language workbenches in the realm of non-MOF modeling technologies.

Graphical syntax is defined in both GMF and Sirius through dedicated forms to manipulate the mappings between abstract and concrete syntax. As opposed to textual specifications, this kind of specifications has disadvantages regarding editing and version control. The mappings in EuGENia are based on embedding the graphical notation information in the abstract syntax model itself, from which GMF editors are generated. While this strategy allows developers to quickly obtain simple diagram editors, it has limitations with respect to complex figures and visibility/graphical constraints.

Pounamu (Zhu et al. 2007) is a language workbench for defining diagram editors where mappings between abstract and concrete syntax are expressed through tool-specific GUI forms, after having designed the diagram elements also through a GUI. Building on the experience gained in Pounamu, Marama (Grundy et al. 2008) embodies an Eclipse-based language workbench with support for multi-view modeling languages. In the Generic Modeling Environment (GME) (Lédeczi et al. 2001) the definition of graphical syntax is interleaved with the metamodel definition (as properties of its elements).

In our work we are exploring the design of a textual and declarative language for mapping abstract syntax to diagrammatic notation. Our approach is novel with respect to this aspect, since the mentioned tools do not rely on an independent language, being their mapping definitions performed through the GUI of the language workbenches. To our knowledge, the open-source Spray project (Eclipse Labs 2016), apparently not active and whose usage did not become widespread, is the only approach that addresses the same problem with a DSL that has a similar nature as ours.

3. Example Domain: Conceptual Modeling

In order to illustrate the primitives of the Xdiagram DSL, consider the domain of conceptual models as an example. Figure 1 presents a metamodel that describes the abstract syntax of a *conceptual model*. For clarity and conciseness of presentation, this notion of conceptual modeling is slightly simplified. A conceptual model is composed of several *entities* (*abstract* or not) that may hold several *attributes*. An entity may optionally *extend* another entity, and may hold

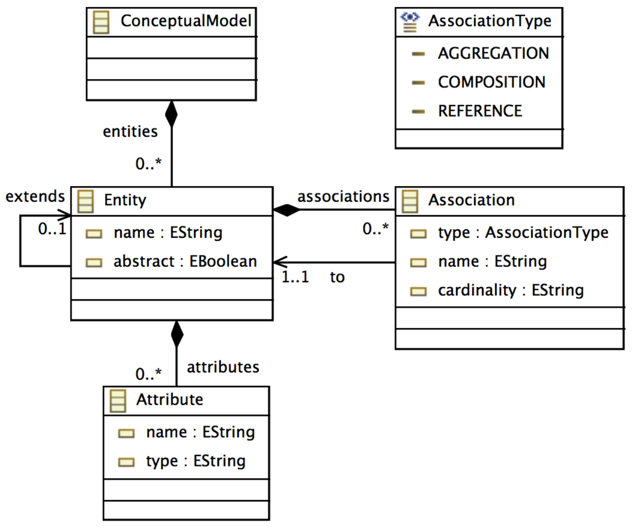

Figure 1. Example metamodel described in EMF: conceptual models with entities, attributes, and relationships.

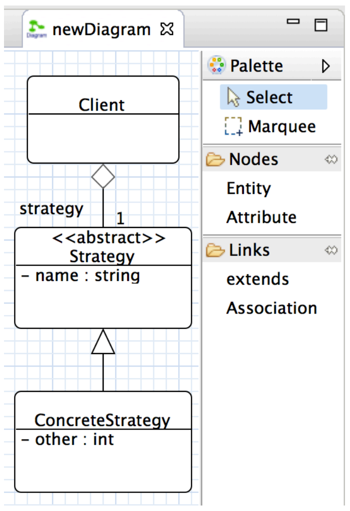

Figure 2. Diagram editor for the language definition of Figure 1, obtained with the Xdiagram specification given in Figures 3 and 4.

associations to other entities. An association has one of the following types: *aggregation*, *composition*, or *reference*.

One of the possible concrete graphical syntax for the given abstract syntax could be a UML-like class diagram, as the one illustrated in Figure 2. The screenshot shows a diagram editor obtained by a specification in Xdiagram (details on next section). Despite the fact that the example is simple, it covers a wide range of cases regarding mapping abstract to concrete syntax, such as alternate ways of mapping diagram links (class extension mapped to a reference, whereas associations mapped to a class), node containments (attributes inside classes), and conditional appearance of diagram elements (stereotype `abstract`) .

4. Xdiagram DSL Primitives

In order to use Xdiagram there must exist a metamodel described in EMF that represents the abstract syntax of the DSML for which one wants to develop support for graphical syntax. This section explains the main language features using the example of the previous section as an illustration.

Each graphical element contained in the diagram is either a *node* (e.g., entities) or a *link* that binds two nodes (e.g., extend relationship). Nodes may contain other nodes, establishing a graphical parent-child relationship (e.g., entity attributes). Every graphical element may have its visibility conditioned by the state of the model (e.g., stereotype `abstract` is visible if an entity is abstract).

4.1 Bindings to Abstract Syntax

The diagram descriptions in Xdiagram contain hard-linked references to the metamodel. The references may target classes and their features (either attributes holding values or references to other objects). Changes in the metamodel may cause compilation errors in the Xdiagram description, for instance, due to a no longer existing element of the metamodel or changes in some of its properties. Further, there are other static verifications being performed to ensure a valid and less fragile specification. For instance, validation ensures that the attributes referenced in the specification have to be part of the class associated with the parent figure.

Throughout the examples, references to metamodel elements in Xdiagram specifications are underlined, and they can be tracked to the meta-model of Figure 1. The specifications given on Figures 3 and 4 are complete and define the diagram editor given on Figure 2.

4.2 Diagram and Nodes

Figure 3 exemplifies the definition of diagram nodes. When specifying a diagram it is necessary to indicate the class of the metamodel whose instances are described by the diagram as a whole. Further, one has to indicate which references of that class hold the objects that are to be contained in the diagram area. Note that a class might have several references, and hence, it is necessary to have a way of indicating which of them should be used. The diagram is associated to a `ConceptualModel` object (line 1), which contains `Entity` objects held in the reference `entities` (line 2).

Node types are defined by associating metamodel classes to graphical figures, as illustrated in the example in two cases. In the first case (lines 5–33), `Entity` objects are defined to be rendered as rectangles, which is turn are composed of several child figures specified in nested blocks (`child` *figure* {...}). Visual attributes may be defined for each figure in a CSS-like manner, such as dimension, position, colors, as well as figure-specific features (e.g., a corner size for round rectangles). The values for dimension (`size`) and `position` are given in number of pixels, using a top-to-bottom and left-to-right (x, y) coordinate system. These

```
 1 diagram ConceptualModel {
 2  contains ConceptualModel.entities;
 3 }
 4
 5 node Entity rectangle {
 6  size 200 100 resizable;
 7  corner 10;
 8
 9  child label {
10   text "<<abstract>>";
11   position 50% 0;
12   size 150 15;
13   invisible if Entity.abstract = false;
14  }
15
16  child label {
17   position 50% 15;
18   text Entity.name;
19   size 150 15;
20  }
21
22  child hline {
23   position 0 30;
24   size 0];
25  }
26
27  child invisible {
28   contains Entity.attributes;
29   size 5] 5];
30   position 5 30;
31   layout vertical;
32  }
33 }
34
35 node Attribute label {
36  size 100 15;
37  text "- " Attribute.name " : " Attribute.type;
38 }
```

Figure 3. Xdiagram DSL: specification of diagram and nodes against the metamodel of Figure 1 (referenced metamodel elements are underlined).

values may be defined as a proportion (using `%`) relative to their parent figure. Dimension may also be defined as an "attachment" (using `]`) at the bottom (for height) or left-hand side (for width) of the parent figure. Although not illustrated in the example, figures can be defined independently and reused across different node types.

Conditional visibility/style. The first child figure of the entity is a label for the abstract stereotype. The description of the appearance of this element is straightforward. However, the last clause defines that the element is invisible if the attribute `abstract` of the entity is set to false. Any visual element may have its visibility constrained in this way. The remaining child elements of the entity have no visibility constraints, and therefore they are always visible. We found this sort of conditional style primitive essential, as it reveals useful in many cases (3 situations in this small example). We present two other examples when exemplifying diagram links ahead.

Node containment. The last child figure of the entity figure defines that it `contains` other nodes, as in the case of the top-level diagram instance. The association of the container to the `Attribute` objects is defined through the metamodel reference `attributes` (line 28). These can only be added to the container in the Entity nodes, since its the only `contains` definition whose reference type is compatible. The attributes as nodes are defined separately in their own definition (lines 35–38), an example that also illustrates the combination of static and dynamic label text.

4.3 Links

Figure 4 exemplifies the definition of diagram links. Link types may be defined in two different forms, both illustrated in the example.

Reference links. This is the simplest form and is based on associating a link type to a metamodel reference. In the specification, the `extends` relationship (lines 1–9) is mapped to the reference `extends` of `Entity` (line 1). In this way, one can create links from the objects of the type that owns the reference to the objects whose type is compatible with the reference type.

Class links. This form is based on having a metamodel class whose objects are represented by links in the diagram. In the specification, the class `Association` is defined as a link type (lines 11–33), having the `source` reference `associations` of `Entity` holding the link objects and the reference `to` of the link class `Association` providing the link `target` object. In this way, one can create links from the object that owns the source reference to the objects whose type is compatible with the target reference type.

Decorators. One may attach decorator figures to links, which are basically like the child figures of the nodes, placed at a certain location of the link connection (relative to the link origin). For reference links, decorators cannot have dynamic parts, since the reference does not hold attributes. On the other hand, the decorators of class links may refer to the attributes of its class.

Anchors. Although not discussed in the context of this example, it is possible to define that a figure is an anchor for incoming or outgoing links (referencing the metamodel reference which the link is representing). In this way, one can define exactly where the link end points are attached. When no anchors are defined, a link can connect anywhere in the figure of the source/target node.

5. Implementation

The Xdiagram DSL is developed with Xtext (Eclipse 2016d), whereas the diagram support is developed on top of Graphiti (Eclipse 2016b). Graphiti is a relatively recent framework, which is in turn developed on top of GEF (Eclipse 2016), that can be used to develop diagram editors by means of programming a framework instantiation. Xdiagram can be thought of as a "meta-instantiation" of Graphiti, in the sense that it embodies a generic editor implementation that becomes concrete in the presence of a specification written in the Xdiagram DSL. Our specification files (Figures 3 and 4) are parsed to an in-memory semantic model (Fowler 2010) that is interpreted at runtime, and the diagram editor components adapt accordingly. We do not perform any sort of code generation, and hence, if our interpretation framework is updated (not involving changes in the DSL syntax) there is no need to perform any changes in existing diagram definitions.

The metamodel of our language currently has 55 classes. Although some language features are still under development, the main primitives of the language have stabilized, given that they were found useful and practical for describing various widely-used diagrammatic notations (e.g., feature diagrams, component diagrams, state machine).

```
 1 link reference Entity.extends {
 2  foreground black;
 3
 4  decorator 100% triangle {
 5   foreground black;
 6   background white;
 7   size 20 20;
 8  }
 9 }
10
11 link class Association
12 source Entity.associations target Association.to {
13
14  decorator 0% rhombus {
15   size 20 20;
16   background black if Association.type = COMPOSITION;
17   background white if Association.type = AGGREGATION;
18   invisible if Association.type = REFERENCE;
19  }
20
21  decorator 50% label {
22   text Association.name;
23  }
24
25  decorator 85% label {
26   text Association.cardinality;
27  }
28
29  decorator 100% arrow {
30   invisible if Association.type <> REFERENCE;
31   size 15 15;
32  }
33 }
```

Figure 4. Xdiagram DSL: specification of links against the metamodel of Figure 1 (referenced metamodel elements are underlined).

6. Conclusions

Upon the development of the main language features of Xdiagram, while assessing their suitability for building several modeling languages, we conclude that declarative text is an effective means for specifying graphical syntax. As future work, we plan to develop support for linking diagrams and to include interaction aspects as part of the language, such as the semantics of copy-paste and element deletion. Further, in order to fully evaluate the tool, we plan to carry out a systematic study to assess the suitability of Xdiagram to represent a wide range of graphical notations in a practical manner. The study will allow us to identify other shortcomings of the approach and to evaluate the usability of the language in a broader context.

References

A. Barisic, P. C. Monteiro, V. Amaral, M. Goulão, and M. P. Monteiro. Patterns for evaluating usability of domain-specific languages. In *Pattern Languages of Programs Conference 2012*. ACM, 2012.

DevBoost. EMFText. www.emftext.org, 2016.

Eclipse. GEF: Graphical editing framework. www.eclipse.org/gef/, 2016.

Eclipse. GMF: Graphical modeling framework. www.eclipse.org/modeling/gmp/, 2016a.

Eclipse. Graphiti. www.eclipse.org/graphiti/, 2016b.

Eclipse. Sirius. www.eclipse.org/sirius, 2016c.

Eclipse. Xtext. www.eclipse.org/Xtext/, 2016d.

Eclipse Labs. Spray. code.google.com/a/eclipselabs.org/p/spray/, 2016.

M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

J. Grundy, J. Hosking, J. Huh, and K. N.-L. Li. Marama: An Eclipse meta-toolset for generating multi-view environments. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 819–822, New York, NY, USA, 2008. ACM.

JetBrains. MPS: Meta programming system. https://www.jetbrains.com/mps, 2016.

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

D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS'10, pages 211–225, Berlin, Heidelberg, 2010. Springer-Verlag.

A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, Nov. 2001.

MetaCase. MetaEdit+. http://www.metacase.com, 2016.

OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

N. Zhu, J. Grundy, J. Hosking, N. Liu, S. Cao, and A. Mehra. Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, 80(8):1390–1407, Aug. 2007.