Electronic Notes in Theoretical Computer Science 72 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 10 pages

An Open Visualization Framework for Metamodel-Based Modeling Languages

Péter Domokos [1,2] Dániel Varró [1,3]

Department of Measurement and Information Systems
Budapest University for Technology and Economics
H-1111, Magyar tudósok körútja 2, Budapest, Hungary

Abstract

In the paper, we propose an automated, SVG-based visualization framework for modeling languages defined by metamodeling techniques. Our framework combines XML standards with existing graph transformation and graph drawing technologies in order to provide an open, tool-independent architecture.

Keywords: visualization, metamodeling, XMI, SVG, model transformation

1 Introduction

Domain specific design environments capture specifications and automatically generate or configure the target applications in particular engineering fields. These visual environments provide one of the most popular application field of graph-based tools.

In general, there are two main approaches fulfilling the needs of such environments: the foundations of **visual languages**, and the paradigm of **metamodeling**. Both approaches follow the idea of the Model Driven Architecture (MDA): in a universal modeling language one must be able to simultaneously design modeling languages *and* concrete target systems, which is the point where the Unified Modeling Language (UML) has certain disadvantages.

Domain specific visual environments (like GenGEd [4], or DiaGen [8]) generalize the theory of traditional formal languages to graph-based visual representations: (i) typically, we first construct some visual alphabet where certain constraints are imposed on the appearance of graphical objects (like circles, rectangles, etc.); (ii)

[1] This work was supported by the Hungarian Scientific Grants FKFP 0193/1999, and OTKA T038027,
[2] Email: pdomokos@mit.bme.hu
[3] Email: varro@mit.bme.hu

afterwards, a well-formed sentence of the visual language is defined by a set of rules, where well-formedness of a sentence is decided by graphical parsing algorithms.

On the other hand, the aim of metamodeling tools (such as GME [9], PROGRES [11] or VIATRA [13]) is to define first the abstract syntax of arbitrary modeling languages in a visual UML-based notation, while the concrete visual representations of domain specific objects are typically based on visual stereotypes. As its benchmark application, the Unified Modeling Language (UML) itself was defined by metamodeling techniques in a meta-circular way. However, metamodeling can simultaneously be applied to capture the static structure of various mathematical domains (like Petri nets, automata, etc.).

A common problem in each approach is to provide a standardized way to integrate them into existing engineering applications, which has been addressed recently by creating XML-based model interchange formats for their internal models. However, while the design of common XML-based descriptions is rather straightforward for the abstract logical syntax of models (see, for instance, such initiatives as PNML [2], or GXL & GTXL [12]), it seems to be hard to design a uniform XML representation for the concrete visual syntax of domain models.

Although there also exists an XML standard called Scalable Vector Graphics (SVG) [15] for representing vector-based drawings (built up from drawing primitives such as circles or lines) with automated rendering support for web browsers, a manual design of an SVG representation does not scale up well for individual modeling languages.

In the paper, we propose a standard, SVG-based batch visualization framework for modeling languages defined by metamodeling techniques. Our framework combines XML standards with existing graph transformation and graph drawing technologies in order to provide an open, tool-independent architecture. However, as the handling of user interaction and complex graphical constraints is out of scope of the current paper, our proposal should be regarded as a feasibility study for a complex SVG-based visual language environment.

2 The Visualization Framework

An overview of our graph transformation-based visualization framework is depicted in Fig. 1. The general concepts of our framework is demonstrated on the concrete domain of Petri nets, which is a well-known visual mathematical formalism frequently used for modeling distributed and concurrent systems.

Our overall aim is to visualize a concrete model (i.e., a Petri net) by transforming it into an SVG representation, which can be rendered by web browsers.

For this reason, the visualization process requires the following inputs.

- **Petri net metamodel.** This metamodel defines the abstract syntax of Petri nets by introducing, for instance, the notions of places, transitions, etc. in a UML notation.

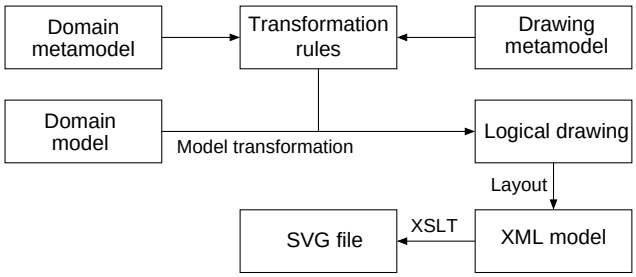

Fig. 1. An overview of the visualization process

- **Petri net model to be visualized.** A concrete Petri net model is an instance of the previous metamodel. As we concentrate on visualization and not verification, we suppose that this model fulfills all additional well-formedness constraints as well. At this point, our abstract model is represented by typed and attributed graphs.
- **Mapping of abstract syntax into concrete syntax.** This mapping between the abstract Petri net model and some predefined drawing primitives like circles or rectangles is specified in the form of graph transformation rules.

As there is a huge abstraction gap between an abstract, graph-based representation of Petri nets and a concrete, SVG-based textual format, the entire visualization process is divided into three main steps:

- **From abstract syntax to drawing primitives.** As the initial step, the abstract Petri net model is transformed into a logical drawing that contains drawing primitives (like circles or rectangles) with logical relations between them (e.g., containment). This model transformation is driven by the input set of graph transformation rules and generates an XMI [10] representation for the logical drawing model.
- **Layout calculation by graph drawing techniques.** The drawing primitives created in the first step only have logical relations between them such as containment of objects (e.g., a net contains places and transitions) or source-target relations for arrows (an input arc is leading from a place to a transition). The concrete physical coordinates of Petri net objects (such as the center and radius of a circle representing a place) are determined in this second step using off-the-shelf graph drawing tools and exported as attributes into the previous XMI representation of the drawing model.
- **Generation of the SVG representation.** Now, we have all the information required for the visualization of Petri nets. For this reason, the XMI representation of the drawing is transformed into the standard SVG format.
- **Rendering by a web browser.** The final step of the visualization process is to open the SVG file of the Petri net in a web browser.

In the following, the steps of the visualization process are overviewed by an example, which is the visualization of Petri nets.

2.1 Metamodeling

Domain metamodel: Petri nets

Metamodels capture the abstract structure of the problem domain visually, in a UML notation. For instance, the metamodel of Petri nets is a formal definition of the language of Petri nets and a Petri net model is a well-formed instance of the metamodel. This instance-of relationship is frequently formalized by means of type graphs and typing homomorphism [5].

In Fig. 2, a metamodel of Petri nets is depicted for demonstration purposes. According to this metamodel, Petri nets consist of *Places* and *Transitions*, connected by two types of arcs: *InputArcs* leading from places to transitions and *OutputArcs* leading from transitions to places. These notions of Petri nets are represented by *UML classes* in the metamodel.

The relations that connect the objects to each other are defined by *UML associations*. Such relations are those that connect arcs to places and transitions (e.g., *fromPlace*, *toPlace* in the example), but also the containment relations (e.g., *places*, *transitions*) that connect Petri net objects to the net itself belong here.

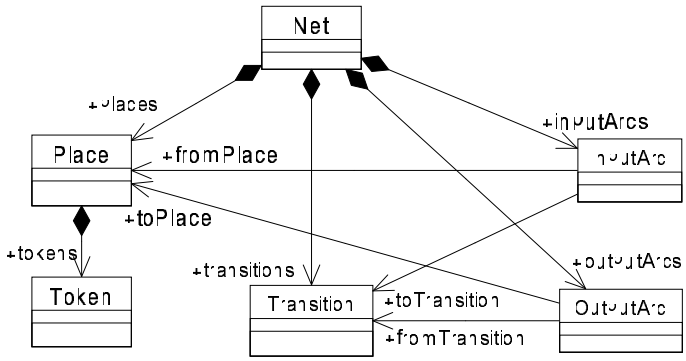

Fig. 2. A metamodel of Petri nets

In the metamodel, classes and associations define the abstract syntax of Petri nets. However, in the concrete syntax of Petri nets, arcs are represented by arrows, places by circles, transitions by rectangles and tokens by black circles. In the sequel, we define a modeling language for the representation of vector graphics drawing primitives.

A metamodel of drawing primitives

An extract from our proposed metamodel for drawing primitives is depicted in Fig. 3. Each Petri net object is transformed into a graphical object (called *GrObject*) which serves as a container of (one or more) drawing primitives. For instance, the GrObject of a Petri net place merely contains a circle, however, in case of finite state machines, the accepting states require to be composed of two separate circles.

The metamodel introduces a **graph representation for drawing primitives**. In this sense, the abstract *GrObject* class (i.e., it may not have instances) is the

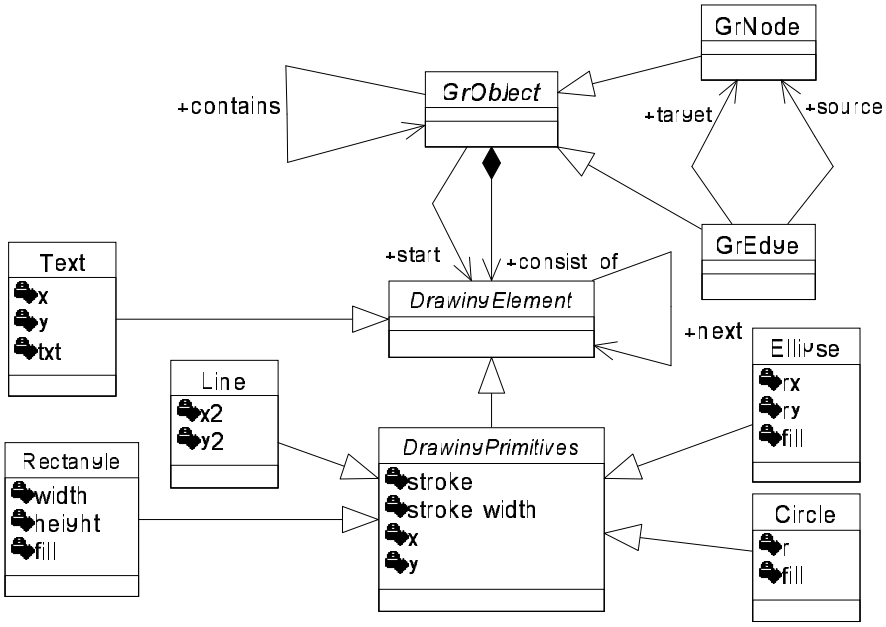

Fig. 3. An extract from the metamodel of drawing primitives

common ancestor of *GrNode* and *GrEdge*. According to the traditional representation, places, transitions and tokens of Petri nets should be transformed to *GrNodes*, while InputArcs and OutputArcs (together with the connecting relations) will be expressed as *GrEdges*.

A *GrObject* may contain several *DrawingElements* that represent the drawing primitives. *DrawingElement* (which is also abstract) is the common ancestor of the drawing primitives such as *lines, rectangles, circles* or *text*.

The drawing primitives have several attributes that have default values introduced by the mapping from the abstract syntax to the concrete syntax or they will be calculated later by the graph drawing tool. For instance, the attribute *fill* defines the fill color, *stroke* defines the color of the line and *stroke-width* defines the width of the line.

In many cases, the graphical notation requires to draw several primitives on the top of each other (e.g. the final state in case of state machines is represented by a smaller filled black circle is placed above a filled white circle). For this reason, *start* and *next* relations are introduced. The *start* relations define which primitive should be drawn first (there may only be a single *start* edge in a *GrObject*) and a *next* edge points to the next primitive to be drawn.

When a *GrObject* consists of more than a single drawing primitive, the relation of these drawing primitives to each other must be defined (e.g., a circle is above a rectangle etc.). For this reason, we use local coordinates: a compound object is drawn in its local coordinate system.

Currently, two kinds of logical relations (constraints) are supported in the metamodel, however, it is possible to introduce further graphical constraints.

- The associations *source* and *target* depict the source and target nodes of an

GrEdge.

- The *contains* association denotes a containment relation, i.e., that one *GrObject* may contain another. For instance, a Petri net place may contain tokens. In this case, both tokens and places are different objects, however, tokens should be linked to places.

2.2 Model transformation

The *mapping between the metamodel classes and the drawing primitives* is defined by *model transformations*, which is formally a combination of specially structured graph transformation rules driven by control structures [13]. The source language of the transformation is defined by the domain metamodel (i.e., the metamodel of Petri nets in our case), while the fixed target language is described by the previous metamodel of drawing primitives.

Generally speaking, the transformation from domain models to drawing models basically consists of the following steps.

- **Transforming domain objects into drawing primitives**. At the initial step, each domain class is transformed into drawing objects composed of one or more drawing primitives. In our Petri net example, places, transitions, and tokens are transformed into *GrNodes*, while input and output arcs are projected into *GrEdges*.
- **Creating the source/target relations** between *GrEdges* and *GrNodes*. At this point, input and output arcs of Petri nets are connected to places and transitions, which is a kind of graphical constraint (however, no information on physical coordinates is added at this point).
- **Creating the containment relations**. At this final point, for instance, tokens are connected to places that contain them. This containment relation also represent some constraints imposed on valid visualizations of Petri nets.

The result of the model transformation step is a *logical* drawing that contains drawing primitives like circles or lines without concrete physical coordinates, edges and the logical relations between the groups of these primitives (such as the source and target of an edge or containment constraints).

2.3 XMI representation of the logical drawing

The logical drawing is exported in an XMI format [10], which is a standard XML dialect derived automatically from the corresponding metamodel. For instance, the following piece of XMI code represents a Petri net place containing a token.

- **Place**. *GrNode 13* represents the Petri net place composed of a single white circle drawn with a stroke width of 2 pixels. The fact of token containment is represented by the *contains* relation with ID references to the contained tokens (note the *xmi.idref* attributes).

```
<GrNode xmi.id="13">
  <contains>
    <GrNode xmi.idref="24"/>
  </contains>
  <start>
    <Circle xmi.idref="14"/>
  </start>
  <consist_of>
    <Circle xmi.id="14">
      <r>25</r>
      <stroke>black</stroke>
      <stroke_width>2</stroke_width>
      <fill>white</fill>
      <next/>
    </Circle>
  </consist_of>
</GrNode>
```

```
<GrNode xmi.id="24">
  <contains/>


  <start>
    <Circle xmi.idref="25"/>
  </start>
  <consist_of>
    <Circle xmi.id="25">
      <r>10</r>
      <stroke>black</stroke>
      <stroke_width>2</stroke_width>
      <fill>black</fill>
      <next/>
    </Circle>
  </consist_of>
</GrNode>
```

Fig. 4. The XMI representation of drawing primitives

- **Token**. A token (*GrNode 24* in the example) is represented by a smaller, black circle (the radius *r* is less than in the previous case).

2.4 Layout

The purpose of the next, so-called, layout step is to arrange the *GrObjects* into an appropriate layout format, which requires to calculate the physical coordinates of drawing objects (according to their bounding boxes).

As the graph drawing problem already has a rich theory and tool support, we have decided to rely on an off-the-shelf graph drawing tool for the layout step instead of an own implementation. We used the Java-based IBM GFC [7] (Graph Foundation Classes, which is a Java class library for graph creation, manipulation and visualization) toolkit for our experiments due to the rich support of Java for XML processing. On the other hand, we also experienced certain limitations of the tool, especially, when visualizing non-planar graphs, which might require more sophisticated solutions for future applications.

The main steps of the layout calculation process are the following.

- **Processing the XMI file.** As the interface (result) of the model transformation step is an XML file it should be parsed and stored in the appropriate internal representation of the graph drawing tool.
- **Creating drawing layers.** The logical drawing can be grouped into several layers according to the containment relation. The top-most level is the first layer, and each group of objects that is connected to this layer with a *contains* edge belongs to a separate layer. The reason for the layer separation is that frequently, each layer can be processed almost separately during the calculation of node coordinates, which reduces the computational complexity of this subsequent step.
- **Calculating node coordinates.** Typically, the node coordinates are calculated

for each drawing layer separately. In the case of Petri nets, we compute, for instance, the coordinates of circles representing places, and rectangles representing transitions.

- **Resizing.** If the different layers are processed separately we might need to resize nodes to allow that the sub-layer with all the graphical objects contained by the node fit into the node.
- **Writing the XMI file.** The result of the calculation is exported from the internal structure of the tool into the same XMI format of logical drawings by introducing new or recalculating existing variables.

As the result of these steps, we may obtain the following piece of XMI code (Fig. 5), which is an extension of the previous XMI file by concrete coordinates for width, height, etc.

```
<GrNode xmi.id="13">
  ...
  <consist_of>
    <Circle xmi.id="14">
      <cx>25</cx> %Coordi-
nates added
      <cy>25</cy>
      ...
    </Circle>
  </consist_of>
</GrNode>
```

```
<GrNode xmi.id="24">
  ...
  <consist_of>
    <Circle xmi.id="25">
      <cx>25</cx>
      <cy>25</cy>
      ...
    </Circle>
  </consist_of>
</GrNode>
```

Fig. 5. Extended XMI representation of drawing primitives

2.5 Transformation to SVG

The result of the layout step is an XMI file that contains the drawing primitives with the physical coordinates of the nodes grouped into layers. The aim of the final step in the visualization process is to generate an SVG file which can be rendered directly by opening it in a web browser.

SVG (Scalable Vector Graphics) [15] is an XML-based language which was designed to describe vector graphical images. It supports several drawing primitives (like lines, circles, rectangles etc.), offers the possibility of grouping these primitives and provides transformation on them like translation to a specified position, scaling and rotating.

As both the input and the output of this transformation step is an XML file, moreover, the projection from the logical drawing (containing all relevant information such as physical coordinates, layers, etc.) is rather syntactic, we can use an XSLT-based transformation engine [14]. Although this is another transformation technique, it is invisible to the user as being identical for different modeling languages. The visualization process starting from the logical representation of Petri nets (Fig. 5) is expected to derive the following SVG code (see Fig. 6).

```
<svg width="100%" height="100%" viewBox="0 0 200 600">
<g transform="translate(125 125)">
 <g>   <!-- place -->
  <circle cx='25' cy='25' r='25'
          stroke='black' stroke-width='2' fill='white'/>
 </g>
 <g>   <!-- token -->
  <circle cx='25' cy='25' r='10'
          stroke='black' stroke-width='2' fill='black'/>
 </g>
</g>
</svg>
```

Fig. 6. SVG representation of a Petri net place with a token

3 Conclusions

In the current paper, we presented an off-line open visualization framework for modeling languages defined by metamodeling techniques. Our framework combines XML standards with existing graph transformation and graph drawing technologies in order to provide an SVG-based output that can be rendered by web browsers. The entire architecture is independent of the concrete graph drawing and model transformation tool as the intermediate models of our framework are all based on the XMI standard. [4]

Our first experiments (visualization of Petri nets and hierarchical automata – a popular semantic domain of UML statecharts) were all based on modeling languages defined by metamodeling techniques, where an off-line (batch) visualization approach is still useful for documenting the logical structure of the model in the domain specific visual notation.

However, more sophisticated techniques are required for an SVG-based framework integrated as the user interface of visual language tools as SVG has certain drawbacks concerning interaction and complex graphical constraints. A potential candidate for such purposes is CSVG [3] (Constraint Scalable Vector Graphics), which is an extension to SVG that can handle graphical constraints defined in a mathematical form. In the future, we plan to do further investigations in that field.

References

[1] "GraphML," .
URL `http://www.graphdrawing.org`

[2] "Petri Net Markup Language," .
URL `http://www.informatik.hu-berlin.de/top/pnml`

[3] Badros, G. J., J. J. Tirtowidjojo, K. Marriott, B. Meyer, W. Portnoy and A. Borning, *A constraint extension to Scalable Vector Graphics*, Technical report, School of

[4] In the future, intermediate models might be represented in GraphML [1]. Currently, due to the lack of appropriate tools it does not provide advantages over an XMI-based solution.

Computer Science and Software Engineering, Monash Univ. and Dept. of Computer Science and Engineering, Univ. of Washington.
URL http://www.cs.washington.edu/research/constraints/web/csvg-www10

[4] Bardohl, R. and H. Ehrig, *Conceptual model of the graphical editor GENGED for the visual definition of visual languages*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Proc. Theory and Application to Graph Transformations (TAGT'98)*, LNCS **1764** (2000), pp. 252–266.

[5] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), pp. 241–265.

[6] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, **2: Applications, Languages and Tools**, World Scientific, 1999.

[7] IBM, "Graph Foundation Classes for Java," .
URL http://www.alphaworks.ibm.com/tech/gfc

[8] Köth, O. and M. Minas, *Generating diagram editors providing free-hand editing as well as syntax-directed editing*, in: H. Ehrig and G. Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, Berlin, Germany, 2000, pp. 32–39.

[9] Ledeczi., A., M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle and P. Volgyesi, *The Generic Modeling Environment*, in: *Proc. Workshop on Intelligent Signal Processing*, 2001.

[10] Object Management Group, "XML Metadata Interchange," .
URL
http://www.omg.org/technology/documents/formal/xmi.htm

[11] Schürr, A., A. J. Winter and A. Zündorf, "In [6]," World Scientific, 1999 pp. 487–550.

[12] Taentzer, G., *Towards common exchange formats for graphs and graph transformation systems*, in: J. Padberg, editor, *UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, ENTCS **44 (4)**, 2001.

[13] Varró, D., G. Varró and A. Pataricza, *Designing the automatic transformation of visual languages*, Science of Computer Programming **44** (2002), pp. 205–227.

[14] World Wide Web Consortium, "XSL Transformations Version 1.0," .
URL http://www.w3.org/TR/xslt

[15] World Wide Web Consortium, "Scalable Vector Graphics (SVG) 1.0 Specification," (2001).
URL http://www.w3.org/TR/SVG/