Graphical Concrete Syntax rendering with SVG

Frédéric Fondement

ENSISA, MIPS
Université de Haute Alsace
12, rue des frères Lumière
F-68093 Mulhouse, France
frederic.fondement@uha.fr

Abstract. Model-based techniques place modeling at the cornerstone of software development. Because of the large number of domains and levels of abstraction one can encounter in software systems, a large number of modeling languages is necessary. Modeling languages need to be properly defined regarding concrete syntax in addition to abstract syntax and semantics. Most modeling languages use a graphical concrete syntax, and solutions to model those syntaxes appeared. If those solutions are convincing to support the rapid development of graphical modeling tools, they are often restrictive in the range of possible concrete syntaxes for a given abstract syntax, and rely on dedicated technologies. In previous works, we proposed such a solution based on a representation model which was more flexible in that it abstracted away purely graphical concerns. Those concerns include actual design for representation icons, how the design reacts to representation variations within the icons, possible interactions with an icon, and synchronization between the graphical representation and the graphical model. In this paper, we show how to solve those four last points using the SVG open standard for vector graphics. We propose to define representation icons by SVG templates complemented by layout constraints, a predefined and extensible library of possible user interactions using DOM, and a specific approach based on events to synchronize the graphical representation with the graphical model. Thus, our solution solves the concrete realization of an modeling environment cumulating advantages of a clear separation between abstract and concrete syntaxes at the modeling level, while benefiting from the expertise of the vector graphics community.

Keywords. MDE, MDA, Language Engineering, Graphical Concrete Syntax, XML, SVG, DOM.

1 Introduction an related works

Model-based techniques to software-intensive system engineering, such as Model Driven Engineering (MDE) [1], place models at the cornerstone of development activities. In parallel, long held research showed advantages of Domain Specific Languages over general purpose languages, provided those languages are properly supported and able to interoperate [2]. Of course, this DSL approach also applies for modeling languages

[3]. As a consequence, because of the multiplicity of domains and levels of abstraction implied even in a single software-intensive development project, there is a need for a large number of well-supported modeling languages. Thus, much is to be awaited from comprehensive and ergonomic techniques to modeling-language engineering.

A (modeling) language is properly defined by an abstract syntax, semantics, and a set of concrete syntaxes [4]. Metamodeling is a convincing technique to capture the abstract syntax of a language in which a model (so called a metamodel) states the vocabulary and the taxonomy of a language. Thanks to these metamodels, automated tools make possible to manipulate and exchange conforming models, such as MDR [5]. Capturing semantics is still a research issue, even though solutions for support were already proposed, for example in [6]. Concrete syntaxes may be either textual or graphical, but are usually a mix of both. As an example, we proposed in [7] a domain specific language as a mean to support textual editing and representation of models.

Solutions like GEF [8], Topcased [9] or MetaEdit [3] apply the same approach to support graphical concrete syntaxes for modeling languages: a domain specific language makes possible to describe how a modeling language is to be graphically represented. Automatic tools turn this specification into a complete graphical editing environment. Approaches such as AToM^3 [10] or Tiger [11] are similar, even though their DSLs are given graphical (or hybrid) concrete syntaxes, and better apply lessons learned in the visual language community (e.g. by permitting a precise definition of user interactions). A problem with these approaches is that they restrict the range of possible concrete syntaxes for a given metamodel since structure of abstract syntax constraints structure of concrete syntax (with the notable exception of AToM^3). They also need to provide a specific language for graphical depicting of icons.

Another approach is to describe the mapping to a representation language, as applied in [12], by a bidirectional model transformation. A problem with this approach is that model transformation languages are not as well suited as a DSL (as those presented above) for expressing graphical concrete syntaxes.

A last kind of approach we presented in [13] makes use of a concrete syntax graph which is synchronized with the abstract syntax graph (i.e. the model) while it is edited, (following in this the philosophy of AToM^3). Constraints are designed to prescribe the synchronization schemes, leaving the possibility to let a (verifiable) model transformation or a constraint solver realizing the actual synchronization. An important advantage is that abstract and concrete syntaxes are completely decoupled thus encouraging reusability of concrete syntaxes and avoiding pollution of the abstract syntax by concrete syntax concerns. Moreover, the approach follows the results of the visual language community (the interested reader may refer to a synthesis in [14]) to modeling languages. However, an important drawback is that the graphical description of icons is left unclear.

In this paper, extended from [15], we complement the latter approach by formally defining icons, trying to reuse best practices in the field of vector graphics. We propose to port to the metamodeling technological space the open standard Scalable Vector Graphics (SVG) [16], for clearly defining icons involved in the concrete syntax of modeling languages. An advantage is that the SVG standard can formally define 2D graph-

ics. Moreover, designers who use SVG to represent icons of a language do not need to be (meta-)model specialists.

To specify a graphical concrete syntaxes, one has to state what are the *icons* of the representable concepts of the abstract syntax, what are the *variation points* and their synchronization with the model (e.g. an editable text to represent a name feature), and how the icons *react to variations*. Moreover, in order for the specification to be turned into a graphical editing environment, language engineers need to specify possible *user interactions*, e.g. that an icon can freely be moved on the diagramming scene, or that a path can be added an intermediate point.

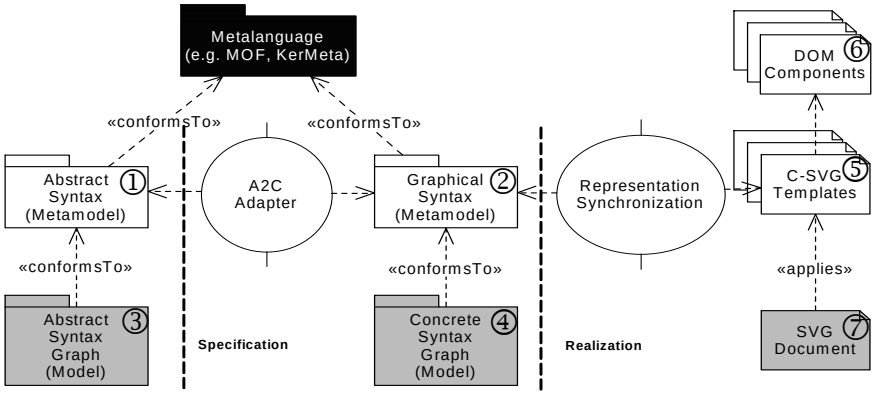

Fig. 1. General Architecture

Figure 1depicts the overall process. As said above, approach described in [13] *specify* the concrete syntax of a modeling language by formalizing the structure of the concrete syntax graph under the form of a metamodel ②. The concrete syntax graph ④ is kept synchronized with the model ③ as specified with constraints. Moreover, additional constraints fix the spatial relationship between representation icons. The approach presented in this paper complements the specification by *realizing* the graphical representation. Icons are described by SVG templates ⑤. Constraints have long proven their ability to handle variability in graphical environments [17]. C-SVG [18] is an environment for supporting constraints in SVG that we propose as a mean to handle variability within icons. SVG templates should thus be complemented with constraints that can be expressed in the C-SVG language. Since SVG is an XML dialect, we propose an extensible set of predefined user interactions using the DOM API [19] to manipulate XML documents at runtime ⑥. We propose a lightweight mean for synchronizing representation with the concrete syntax graph based on events, in order to render variation points. Finally, the modeling environment is an interactive SVG document ⑦ (the diagram) dynamically showed in an SVG renderer.

The rest of the document is organized as follows. Section 2 presents an example for a modeling language and its specification following the approach we presented in [13] ①②. Sections 3, 4, and 5 detail the approach along with the same example by presenting icon definition (including reactions to variations) ⑤, user interactions ⑥, and rela-

tion with the concrete syntax graph (further called the graphical model) ④, respectively. Section 6 end the paper with concluding remarks.

2 Specifying Concrete Syntax for Statecharts

In this section, we detail an example for a modeling language. The abstract syntax of the language is specified by a metamodel, and a graphical concrete syntax is specified as presented in [13].

Statecharts are described in [20]. We show here a metamodel to state its abstract syntax (see figure 1 ①). For sake of simplicity and readability, we will restrict ourselves to a simplified subset of these concepts, as shown by figure 2. State vertices might be connected by transitions. A transition has exactly one source vertex and one target vertex. A vertex is either a pseudo state (initial state, choice,...) or a state, which is in turn either a composite state (i.e. containing other vertices and transitions), a simple state, or a final state. Transitions are triggered by events. A state machine is given by its top state.

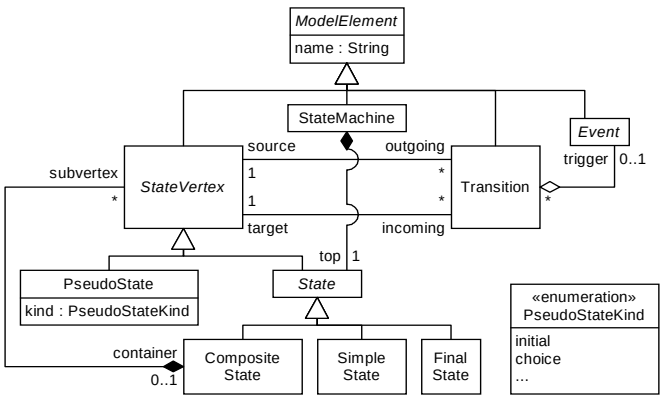

Fig. 2. The Simplified Statechart Metamodel

The concepts of the statechart language can be represented by the symbols shown in figure 4. There is no need to define the `StateMachine` symbol since a state machine cannot be represented. “event”, “name”, and “contents” parts of icons are variation points of the icons: the “event” text should be replaced (if necessary) by the name of the event that triggers the transition, “name” text should be replaced by the name of the represented state, and “content” text points the placeholder for sub-states of the represented composite state. The icons can be freely moved and resized in the diagram, except icons for transitions that have to connect representations for the source and target states of the transition.

Figure 3 shows an excerpt of the specification for the graphical concrete syntax of the statechart language informally described above. The figure is separated in three parts. The part in the left recalls the metamodel. The part in the right defines the graph-

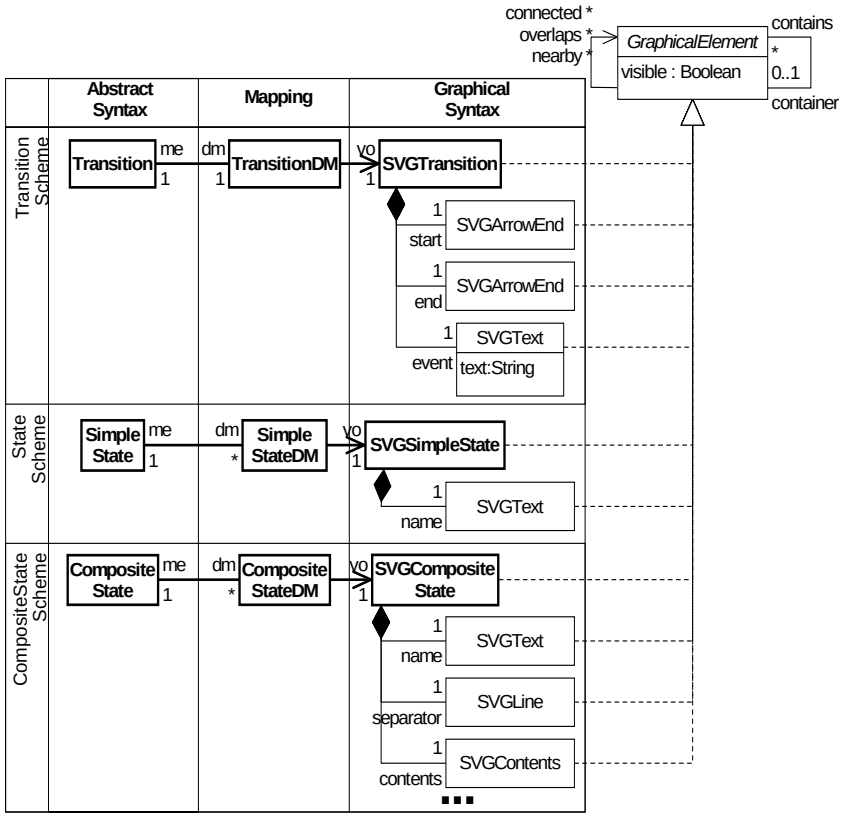

Fig. 3. Excerpt of the Statechart Graphical Concrete Syntax Specification

Transition	SimpleState	Composite State	FinalState	PseudoState (initial)	PseudoState (choice)
event	name	name / contents			

Fig. 4. Symbols for the Statechart Concepts

ical elements as a metamodel (see figure 1 ②) by decomposing the graphical icons in different elements. Each graphical element extends the `GraphicalElement` abstract class, which holds relations exposing spatial relationships. As an example, the icon for `CompositeState` is decomposed into a text, a line, and a placeholder for contained states. Possible variations in the elements of graphical objects state the possible variations in the icons (e.g. the value of the `text` attribute in `SVGText`). The mapping between abstract syntax and graphical syntax is described using mapping classes as shown in the middle part of figure 3. Those classes are connected to repre-

sentable classes of the metamodel to graphical elements. Constraints, that can be written in OCL [21], can make more precise synchronization (as exemplified in figure 5), and fix spatial relationships between graphical elements (as exemplified in figure 6).

```
context TransitionDM inv:
  if self.me.trigger->isEmpty()
  then self.vo.event.text.size() = 0
  else self.vo.event.text = self.me.trigger.name
endif
```

Fig. 5. Synchronization Constraint: Text shown on Transitions is Name of Triggering Event

```
context CompositeStateDM
inv: self.me.subvertex->includesAll(
       State.allInstances().dm
           ->select(sdm|self.vo.contains(sdm.vo)).me)
```

Fig. 6. Spatial Relationship Constraint: Containment of Composite States

A major problem with that approach is that concrete representation of graphical elements and its evolution (including information held by the `GraphicalElement` class) is left unspecified. In the rest of the paper, we propose an approach to overcome this impediment.

3 SVG Templates

In this section, we detail the process of defining SVG template icons of a graphical modeling language, with the ability to react to variation.

Principle of the approach is the following: a diagram is an *SVG document* (see figure 1 ⑦) in which a system engineer may freely add new predefined SVG elements as copied from *SVG templates* (see figure 1 ⑤). Each one of these SVG templates corresponds to a *main graphical element* (see figure 3, right part) i.e. a graphical class that has a connection to a mapping class. In the example of figure 3, main graphical elements are `SVGTransition`, `SVGSimpleState`, and `SVGCompositeState`. Composed graphical elements should be described in the template of their topmost container: in the example, a section of the SVG template for `SVGSimpleState` must describe the `name` part. Note that it is possible to synchronize the representation directly with the model (see figure 1 ③), but in this case, structure of the concrete syntax is forced to follow structure of the abstract syntax.

When a system engineer decides to add a new element to his/her model, say a `SimpleState`, a copy of the SVG template for `SVGSimpleState` is integrated into the SVG document (see figure 1 ⑦). In the meantime, an `SVGSimpleState` and an `SVGText` graphical objects are created, and a relation between the template copy (i.e.

the template instance) and the SVGSimpleState graphical object is maintained. According to specification described in section 2, the creation of an SVGSimpleState graphical object should trigger the creation of a SimpleStateDM manager. Finally an associated SimpleState object, together with a synchronization between the value of the name slot of the State object and the value of the text slot of the SVGText graphical object should be created, as described in section 2. Relation between the template instance and the graphical object (in a model repository) will be further discussed in section 5, while possible interactions with the template instances in the diagramming scene will be described in section 4.

Figure 7 exemplifies this template-based approach. Main display classes, which are emboldened on the figure, have a corresponding SVG template, and each one of contained display classes has an SVG counterpart in the template. As an example, a start section appears in the SVG template for SVGTransition, which corresponds to the contained start SVGArrowEnd display object. Note that the SVG section for the end display object is different, even though it corresponds to the same SVGArrowEnd display class. When the system engineer decides to place a new transition in the diagram (i.e. the SVG scene), any $$ occurrence in the SVG template is replaced by an identifier specific to the template instance in the SVG diagram so that the various SVG elements in the scene can be identified as part of a specific template instance.

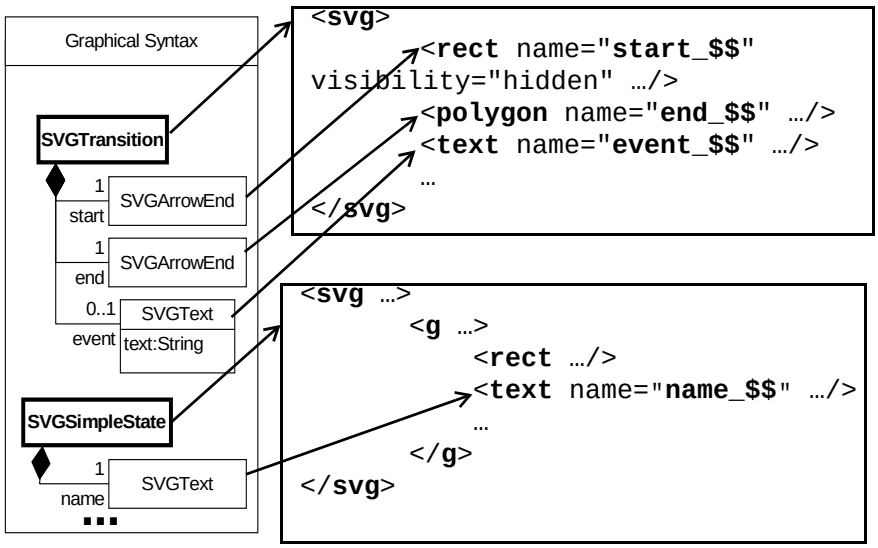

Fig. 7. SVG Templates for Statecharts

As explained before, template instances are subject to variations according to user interactions. In the example of simple states, if name is changed, the containing rectangle needs to grow accordingly. This means that template instances have some dynamic behavior, and may need to be reorganized. Templates thus need to specify a layout mechanism to state how an automatic reorganization may happen. Constraints have

long proven to be a comprehensive mean to specify such layout mechanism [17], and we decided to rely on C-SVG [18] that is specialized in constraining SVG documents. In the simple state template, the growing-name problem is solved as shown in figure 8.

```
<svg …> <g …>
 <c:variable name="w_$$" value=
   "c:max(c:width(c:bbox(id('name_$$'))) + 20, 150)" />
 <rect …>
   <c:constraint attributeName="width" value="$w_$$"/>
 </rect>
 <text id="name_$$" …>Simple State Name</text>
</g> </svg>
```

Fig. 8. `SimpleState` SVG Template: CSVG Constraint to Handle Text Growth

First, a variable named `w_$$` tracks an arithmetic expression in which the computed width of the `name_$$` text plays a central role. A constraint, placed in the rectangle, forces that rectangle to be as wide as the value of the `w_$$` variable. Automatic tools can place listeners in the SVG documents so that the C-SVG constraint keep satisfied. If contents of `name_$$` is changed, the computed value for `w_$$` is updated, which triggers a new computation for the rectangle's width.

4 User Interactions

SVG documents are not primarily intended to interact, e.g. by mean of mouse or keyboard, as it is necessary for modeling a system. In this section, we show how to enable user interactions in template instances.

The principle we propose is the following: SVG is an XML dialect, and an SVG document is an XML tree. DOM is an API that programming languages such as Java use to read and alter XML trees [19]. Thus, a program making use of the DOM API may alter an SVG document. We will further call such kind of program a *DOM component*. We chose an architecture in which user actions (e.g. mouse moved, mouse clicked, or key hit) trigger execution of some DOM components, which may alter the SVG document that represents the diagram scene. Those DOM components may behave differently depending on the context (e.g. what are the selected elements, what are the elements under the mouse). It is important for the SVG graphical renderer to dynamically adapt the shown diagram to alterations of the SVG document, as it is the case for the Apache Batik toolset [22].

The user interactions we propose may be compared to graph grammars to enable user interactions. The difference is that they transform XML trees rather than graphs. An advantage is that the SVG language (or more precisely the DOM interface) automatically brings genericity so that an interaction only poorly rely on the transformed elements.

Possible user interactions with representation icons (or parts of them) are recurrent. For instance, behaviors like move, connect, or resize, apply in a wide range of contexts,

regardless they should impact the model (see figure 1 ③) or not. An important point is to have the possibility to choose exactly where those interactions are enabled. That is why we developed a library of standard interactions independent from the context, for them not to be implemented again and again depending on the SVG kind of element that has to expose the behavior. We developed those interactions as parametrized DOM components with the help of the DoPIdom framework [23]; as a consequence, the result of those interactions can only alter the SVG document that represent the diagram. The interactions are triggered by a controller that treats mouse and keyboard events. Parameters are stored in the SVG diagram. Note that some of these interactions are pure queries and do not alter the diagram; these query interactions are intended to be used by other interactions. We list below some of the interactions we implemented (the interested reader may find a more complete list in [24, 25]). Of course, if one needs an additional interaction, it is possible to make the list evolve thanks to DoPidom.

Interactions dedicated to position:

- `Locatable`: finds the coordinates of the holding node in the scene in terms of position, width and height.
- `BorderFindable`: finds a list of points in the scene drawing the outline of the holding SVG node.

Interactions dedicated to movement:

- `Translatable`: moves an SVG node according to given a vector.
- `BorderSlidable`: makes the holding SVG node affix to the outline of an SVG element exposing the `BorderFindable` interaction, as can be found thanks to an `attachedComponent` parameter.
- `Resizeable`: emphasizes the holding SVG node of a given factor.

Interactions dedicated to text editing:

- `CharacterHitable`: places a caret at a given index of a `Text` SVG node.
- `CharacterInstertable` and `CharacterDeletable`: inserts/removes a given character at a given position of a `Text` SVG node.

Interactions dedicated to scene management:

- `Selectable`: places the holding SVG node as part of the selection of the scene.

Interactions dedicated to connection-based languages (see [14]):

- `Link`: such an SVG node will be holder for a connection and should not be rendered in the scene. A `Link` interaction makes the connection between SVG nodes participating in a connection by declaring what is the SVG node for the path (by mean of a `curvedLine` parameter) and what are the elements represented at the ends of the connection (by mean of `start` and `end` parameters).
- `CurvedLine`: such an SVG node may be placed as a connection for a link. They overload a possible `Selectable` behavior by creating line handler in order to change its route (i.e. its intermediate points). Original link is registered by mean of a `parentLink` parameter.
- `Arrow`: such an SVG node can be placed at the `start` or the `end` of a link. A `position` parameter states whether the SVG node is at the beginning or at the end of the connection.

Interactions dedicated to containment:

- `Container`: such an SVG node is able to contain SVG nodes declaring the `Containable` interaction. Contained elements are placed in a `contents` parameter. Interaction changes an eventual `Translatable` behavior by making the contained nodes follow the same movement. This notion is independent from the notion of SVG group.
- `Contained`: such an SVG node may be part of the contents of an SVG node declaring the `Container` interaction. Container is placed in a `container` parameter. Interaction changes the `Translatable` behavior by attaching or detaching the SVG node from its container according to its target position.

```
<svg …>
  <g dpi:component="Translatable, Contained,…" …>
   <rect id="border_$$" dpi:component="BorderFindable,
…"…/>
    <text name="name" dpi:component="CharacterHitable,
CharacterInstertable, CharacterDeletable, …" …/>
    …
  </g>
</svg>
```

Fig. 9. `SimpleState` SVG Template: Declaring DOM Components

An example, shown in figure 9, is the template definition for `SVGSimpleState`. In this code snippet, the first element is an SVG group that declares the `Contained` and `Translatable` interactions. These declarations makes thus possible both to move freely the representation for a simple state template instance as a whole, and to make it containable by another SVG element declaring the `Container` interaction (as it should be the case for the composite state template). The group contains a rectangle that is responsible for being the outline of the state in that it declares the `BorderFindable` interaction; as such, another SVG node declaring the `BorderSlidable` interaction can be affixed to the rectangle (e.g. an end in the representation for a transition). The group also contains a text that is the placeholder for the name of the state. That is why this text must be editable and declares the `CharacterHitable`, `CharacterInsertable` and `CharacterDeletable` interfaces.

5 Relationship with the model

Our choice to develop reusable behaviors, which only act on the SVG representation, prevents from directly updating the information of the graphical elements (and of the model as an indirect result). We show here how to complement SVG templates and predefined events for the modeling information to be updated.

To do so, we propose to add listeners that can be declared on the SVG templates to synchronize atomic information in the SVG document with atomic information in the graphical model. We call atomic information a datum that is either a character string, a

boolean, an integer or a real. This information may be processed in the document (e.g. using a C-SVG constraint or an XSL transformation) to be properly represented.

Before realizing the synchronization, the relationship between the SVG representation and the graphical model needs to be established. The solution we propose is to maintain variables in the SVG document for each SVG node that corresponds to a graphical object. Those variables are to be filled at template instantiation time using an action language. Moreover, actions may have to be performed in case the template instance is removed from the scene. Variable declarations, initialization and removal actions can be specified in the SVG templates, as exemplified in figure 10.

```
<svg onCreation="{Java|
 t = model.getSVGText().createSVTText();
 s = model.getSVGSimpleState().createSVGSimpleState();
 s.setName(t);}"
   onDeletion="{Java|
 s.refDelete();}" …>
<g id="$$" …>
 <m:variable name="self" value="$s" />
 <rect id="border_$$" …/>
 …
 <text id="name_$$" …>
   <m:variable name="self" value="$t" />
   newState</text>
</g>
</svg>
```

Fig. 10. `SimpleState` SVG Template: Variables

In the figure, the SVG template for simple states is complemented by a creation action written in the java language (using the JMI API [26]) where a variable `model` plays the role of the model repository. In the action, two graphical objects (an `SVGSimpleState` and an `SVGText` further referred to `s` and `t`, respectively) are instantiated and associated as prescribed in figure 3. A deletion script states that the `s` object should be deleted when suppressing the template instance (note that the `t` object does not need to be suppressed explicitly because of the composition relationship between the `SVGSimmpleState` and `SVGText` elements). Moreover, new `variable` XML nodes are added to the SVG template to handle the local dependencies of the representation to the graphical model, as suggested by arrows in figure 7. At template instantiation time, the action is executed, and the local variables are set to the references resulting from the execution of the action. In the simple state example, those variables are either initialized to `s` (for the group and the rectangle) or to `t` (for the text) as declared by the values of the `variable` variable XML nodes. Note that an SVG node may declare different variables.

Once the relationship between the SVG representation and the graphical model is established, it is possible to synchronize atomic information between them. To do so,

we introduce a new `update` XML node. The `location` XML attribute of the `update` node states, with an XPath expression [27], where does the atomic information to synchronize appears in the document. Two more XML attributes of the update XML node state what are the graphical object and the slot to observe.

```
<svg …>
<g id="$$" …>
 <rect id="border_$$" …/>
 …
 <text id="name_$$" value="newState"
    dpi:component="CharacterInstertable, …" …>
    <m:variable name="self" value="$t" />
    <m:variable name="displayed" value="newState" />
    <c:tval
    value="../variable[@name='displayed'][1]/@value" />
    <m:updater var_source="$t" slot="text"
    location="../variable[@name='displayed'][1]/@value"
     />
 </text>
</g>
</svg>
```

Fig. 11. `SimpleState` SVG Template: Updater

Figure 11 shows such a declaration of synchronization in the case of the simple state template: the `text` slot of the `SVGText` graphical object `t` has to be rendered in the (editable) `text` SVG node. To do so, we introduce a new variable `displayed` which will be rendered by the `text` SVG node thanks to a `tval` C-SVG constraint. An updater synchronizes the value of the `display` variable with the information in the model. When the text is edited in the representation, the C-SVG constraint changes the value of the `display` variable, which triggers propagation of the new text to the corresponding graphical object. When the text changes in the model, the updater is notified and changes the value for the variable, which is then rendered according to the C-SVG constraint.

The last information that has to be reflected both in the graphical objects and in the SVG diagram is the information held by the `GraphicalElement` class, which are the spatial relationships and the visibility. The information is automatically updated by a double synchronization mechanism implemented by observers. On one hand, observers track actions performed by the interactions. On the other hand, other observers track changes in graphical objects as stored in the `isVisible`, `container`, `contained`, `nearby`, `overlaps`, and `connected` features. `isVisible` (as found in the graphical object maintained by the `self` variable of the representation node) is synchronized depending on the `display` SVG attribute. The other slots are updated during the execution of interactions e.g. `Container`, `Contained`, or `BorderSlidable`, i.e. any interaction able to change spatial relationship. Note that

all these interactions can be vetoed (in case a constraint fails at model level - see section 2) or forced (in case the graphical model changes).

6 Conclusion

We proposed here a technique for concretely representing a model in a diagram, once the abstract syntax (i.e. the metamodel) is known. We took advantage from the widely accepted SVG standard to specify vector graphics. The approach is intended to be used in conjunction with the approach presented in [13], which clearly separates modeling data from graphical data, and which leads the concrete realization, following the example of a component realizing its specification interfaces.

When combining those two approaches, the steps to specify a graphical concrete syntax are thus the following:

1. create a mapping class for each model element of the metamodel that needs to be represented,
2. create for each mapping class one or more graphical class and its different parts reflecting the structure and the variability of the icon,
3. write the constraints on the mapping classes for abstract/concrete synchronization, representation alternatives and inter-icons relationship (e.g. spatial relationship),
4. write the SVG template for each root graphical class,
5. specify allowed interactions from a reusable and generic library acting on the SVG representation,
6. complement the SVG templates with graphical constraints (e.g. using C-SVG) to handle intra-relationships between the various SVG elements composing the templates,
7. create variables and initialization/deletion scripts to establish relation of the representation to the graphical model,
8. declare updaters so that the representation and the graphical model keep synchronized.

The approach we propose is certainly more verbose than other existing approaches (as GMF or GME). However, it manages a broader range of graphical concrete syntaxes. For example, the approach is not limited to connection-based languages thanks to its explicit management of spatial relationships. Moreover, we rely on one hand on metamodeling techniques (for the specification part) which is properly mastered by modeling language engineers, and on the other hand on SVG which is properly mastered by graphical designers. Finally, the modeling language engineer only needs a few knowledge about SVG to place action scripts, variable and updaters in the SVG templates. As such, if other approaches seem more adapted to prototype a graphical language, we believe that our approach is more adapted to realize the graphical concrete syntax of a modeling language.

Compared to [15], we simplified the process of synchronizing the representation with the graphical model. Indeed, in [15], interactions were explicitly sending events that had to be answered by dedicated action scripts as parameters. Here, representation

and graphical model are more tightly coupled thus avoiding such mechanisms. Moreover, those event action scripts were over-specifying synchronization rules between the model and the graphical model, thus dramatically limiting the interest of the graphical model and the implied reusability.

One of the main drawback of our approach is that the specification of the synchronization rules between the model and the graphical model is done using constraints. Thus, one need either a constraint solver (which are usually slow) or an additional bidirectional and incremental model transformation that realizes the constraints, which needs to be proved. Moreover, there is redundancy of information between the model and the graphical model. We plan to change this synchronization specification by making the graphical model a view on the model, following the example of the concept of view in databases.

A prototype implementation can be found in [28]. More insight about interactions is given in [24] and about variables and action scripts in [25].

References

[1] Kent, S.: Model Driven Engineering. In Butler, M.J., Petre, L., Sere, K., eds.: *IFM*. Volume 2335 of *Lecture Notes in Computer Science*., Springer (2002) 286–298

[2] Iivari, J.: Why Are Case Tools Not Used? *Commun. ACM* **39**(10) (1996) 94–103

[3] Pohjonen, R.: Boosting Embedded Systems Development with Domain-Specific Modeling. *RTC Magazine* (April 2003) 57–61

[4] Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? *Computer* **37**(10) (2004) 64–72

[5] Sun Microsystems: Metadata repository (MDR) (December 2005)

[6] Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In Akehurst, D.H., Vogel, R., Paige, R.F., eds.: *ECMDA-FA*. Volume 4530 of *Lecture Notes in Computer Science*., Springer (2007) 157–171

[7] Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: *MoDELS*. Volume 4199 of *Lecture Notes in Computer Science*., Springer (2006) 98–110

[8] Eclipse Consortium: Eclipse Graphical Editing Framework (GEF) http://www.eclipse.org/gef.

[9] Vernadat, F., Percebois, C., Farail, P., Vingerhoeds, R., Rossignol, A., Talpin, J.P., Chemouil, D.: The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development. In: *Data Systems In Aerospace (DASIA), Berlin, Germany, 22/05/2006-25/05/2006*, http://www.esa.int/publications, European Space Agency (ESA Publications) (May 2006)

[10] Guerra, E., de Lara, J.: Event-driven grammars: relating abstract and concrete levels of visual languages. *Software and Systems Modeling, special section on ICGT'04* **6** (2007) 317–347

[11] Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: *ASE*, ACM (2005) 134–143

[12] Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Meta-modelling: A Foundation for Language-Driven Development (2005)

[13] Fondement, F., Baar, T.: Making Metamodels Aware of Concrete Syntax. In: *European Conference on Model Driven Architecture (ECMDA)*. Volume 3748 of *Lecture Notes in Computer Science*. (2005) 190 – 204

[14] Costagliola, G., Lucia, A.D., Orefice, S., Polese, G.: A Classification Framework to Support the Design of Visual Languages. *J. Vis. Lang. Comput.* **13**(6) (2002) 573–600

[15] Fondement, F.: *Concrete syntax definition for modeling languages*. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL) (2007)

[16] Jackson, D., Northway, C.: Scalable Vector Graphics (SVG) Full 1.2 specification. World Wide Web Consortium, Working Draft WD-SVG12-20050413 (April 2005)

[17] Borning, A., Marriott, K., Stuckey, P.J., Xiao, Y.: Solving Linear Arithmetic Constraints for User Interface Applications. In: *ACM Symposium on User Interface Software and Technology.* (1997) 87–96

[18] McCormack, C.L., Marriott, K., Meyer, B.: Constraint SVG. In: *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, New York, NY, USA, ACM Press (2004) 310–311

[19] Hors, A.L., Hégaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) level 3 core specification. World Wide Web Consortium (April 2004)

[20] Harel, D.: Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming* **8**(3) (1987) 231–274

[21] Adaptive Ltd., Boldsoft, France Telecom, International Business Machines Corporation, IONA Technologies, Object Management Group: Object Constraint Language specification, v2.0. OMG Document formal/06-05-01 (May 2006)

[22] Apache Foundation - XML Graphics Project: Batik SVG toolkit. http://xmlgraphics.apache.org/batik/

[23] Beaudoux, O.: DoPIdom: une approche de l'interaction et de la collaboration centrée sur les documents. In: *IHM '06: Proceedings of the 18th international conference on Association Francophone d'Interaction Homme-Machine*, New York, NY, USA, ACM Press (2006) 19–26

[24] Hong, F.: Provide behaviour to XML-SVG. Bachelor Semester Project, École Polytechnique Fédérale de Lausanne (EPFL) (2005)

[25] Rohrer, F., Helg, F.: Synchronization between display objects and representation templates in graphical language construction. Bachelor Semester Project, École Polytechnique Fédérale de Lausanne (EPFL) (2006)

[26] Java Community Process: Java(TM) Metadata Interface API specification 1.0 final release. JSR-000040 (June 2002)

[27] Clark, J., DeRose, S.: XML path language (XPath). World Wide Web Consortium (November 1999)

[28] Fondement, F.: SVG-based modeling tools (2007) http://fondement.free.fr/lgl/projects/probxs/.