Making Metamodels Aware of Concrete Syntax*

Frédéric Fondement and Thomas Baar

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{frederic.fondement, thomas.baar}@epfl.ch

Abstract. Language-centric methodologies, triggered by the success of Domain Specific Languages, rely on precise specifications of modeling languages. While the definition of the abstract syntax is standardized by the 4-layer metamodel architecture of the OMG, most language specifications are held informally for the description of the semantics and the (graphical) concrete syntax. This paper is tackling the problem of specifying the concrete syntax of a language in a formal and non-ambiguous way. We propose to define the concrete syntax by an extension of the already existing metamodel of the abstract syntax, which describes the concepts of the language, with a second layer describing the graphical representation of concepts by visual elements. In addition, an intermediate layer defines how elements of both layers are related to each other. Unlike similar approaches that became the basis of some CASE tools, the intermediate layer is not a pure mapping from abstract to concrete syntax but connects both layers in a flexible, declarative way. We illustrate our approach with a simplified form of statecharts.

Keywords: Metamodeling, MOF, UML, OCL, Concrete Syntax Description, Visual Languages

1 Introduction

Productivity gains brought by Domain Specific Languages (DSL) [1] have shown the importance of using appropriate modeling languages in the early phases of the software lifecycle. DSLs have triggered the new trend of language-centric methodologies (see [2, 3] for first proposals) and are based on the idea that the first step to efficiently treat a problem is to create or to customize a language that allows to describe the problem adequately. The precise definition of DSLs is in practice often a task for domain or methodology specialists who have only basic knowledge on language design. To minimize the effort, all phases of the language definition should be standardized and supported by tools or frameworks.

A modeling language is usually defined in three major steps. The first one is to define concepts of the language, i.e. its vocabulary and taxonomy, as captured by its abstract syntax. Then, its semantics should be described in such a form that the concepts are clearly understood by the users of the language. Finally,

* This work was supported by Hasler-Foundation, project DICS-1850.

it is necessary to precisely describe the notation, as captured by its concrete syntax. Whereas the semantics definition is out of the scope of this paper, we will concentrate on the concrete syntax part, and especially on its relations to the abstract syntax.

The clear separation between abstract and concrete syntax is a technique to cope with the complexity of real-world language definitions since it allows to define the language concepts independently from their representation. For language designers, it is of primary importance to agree on language concepts and on the semantics of these concepts. The graphical representation of the concepts is often considered less important and is described in many language specifications only informally. However, an intuitive graphical representation is crucial for usability and indispensable for tool vendors who want to support a new modeling language with graphical editors, model animators, debuggers, etc. Sometimes, it is appropriate to have for one language more than one graphical representation, for instance when different stakeholders use the same language but need different views on the model. An example of such a language is ORM [4] that provides a graphical syntax intended for ontology engineers and a pseudo-natural syntax intended for non-specialists.

Metamodeling is a widely used technique to capture the abstract syntax of a language. A well defined set of metamodeling constructs such as classes, associations, attributes, etc., complemented with a constraint language such as Object Constraint Language (OCL) allows one to define the concepts of the language and the relationships between them [5]. The abstract syntax is doubtlessly one of the most important parts of language definitions. Each sentence of the language can be represented without loss of semantic information as an instance of the metamodel. Such an instance can be represented in a standardized, textual format based on the general-purpose representation language XMI [6]. Model representations based on XMI are useful for interchanging models between tools but humans need more comprehensible views on models.

Our approach defines the graphical concrete syntax of modeling languages by complementing each metaclass in the metamodel with a *display scheme*. A display scheme contains an iconic and a constraining part. The iconic part introduces a new layer of *display classes* that define the visual objects for the representation of language concepts. The constraining part defines the connection between the instances of the metaclasses and their graphical representation by the instances of the display classes. Technically, the constraining part of a display scheme consists of a *display manager class* that is placed between the metaclass and the display class. Furthermore, display manager classes serve as anchor points for OCL constraints that are used to describe the connection declaratively.

The definition of our representation classes is heavily inspired by visual language definition techniques [7]. Representation classes take for instance into account the spatial relationships between visual objects such as *overlap*, *right*, *hiddenBy*, etc., and whether visual objects are connected by a polyline, curved line, etc. However, there are some noteworthy differences between our approach and

common approaches to define a visual language. Firstly, many visual language definitions do not explicitly distinguish between concrete and abstract syntax. In our approach, the classes for the abstract syntax are completely separated from classes for concrete syntax. Secondly, the mainstream approach to define a visual language is by graph grammars (see [8] for an overview). The underlying idea is to generate all syntactically correct sentences of the visual language as derivations of the grammar rules. In order to decide the question, whether or not a given diagram is syntactically correct, a derivation of the graph grammar rules must be constructed. The same question is decided following our approach just by evaluation of constraints attached to the display manager classes.

The rest of the paper is organized as follows. First, in Sect. 2, a simplified version of the statechart language is briefly described. This language will be used as a case study in Sect. 3 where our approach is stepwise developed. Section 4 will give an overview on related approaches. Finally, Sect. 5 will present conclusions and future work directions.

2 The Statechart Language

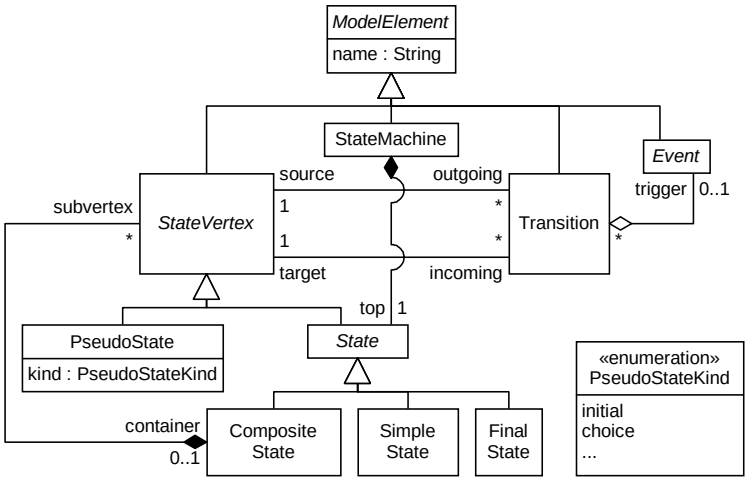


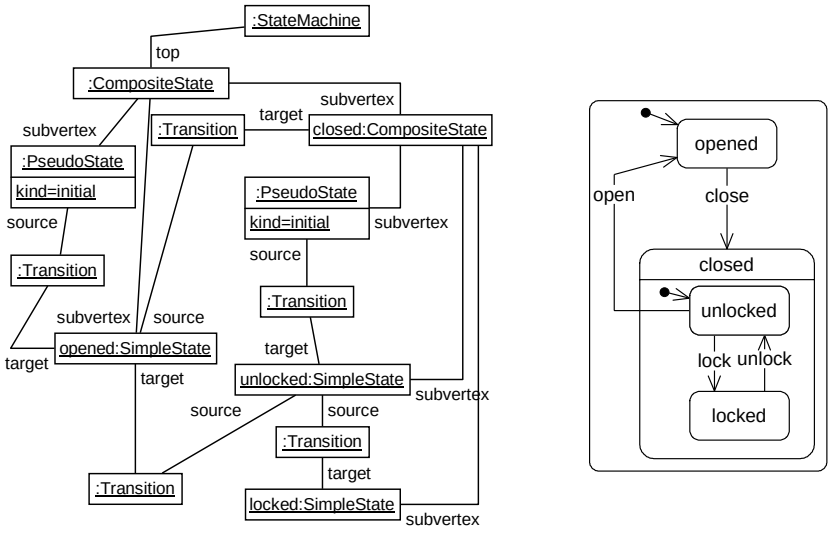
Fig. 1. A simplified metamodel for statecharts

We briefly introduce here the concepts of a simplified, but yet illustrative version of statecharts [9] whose metamodel is shown in Fig. 1. 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, etc.) 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

Table 1. Symbols for representation of concepts

Transition	SimpleState	Composite State	FinalState	PseudoState (initial)	PseudoState (choice)
–event⟩	name	name / contents	◉	●	○

by events. A state machine is given by its top state. Not shown in Fig. 1 are well-formedness rules that complement the metamodel and stipulate, for example, that a final state can never be the source vertex and an initial pseudo-state can never be the target vertex of any transition.



(a) Instantiation of metamodel

(b) More human-friendly, graphical notation

Fig. 2. Two representations of the same statechart

An informal concrete syntax definition might propose the symbols shown in Table 1 for the representation of language concepts defined in the metamodel. Note that there is no need to define a symbol for `StateMachine` because the state machine is represented by its top state. The intended meaning for the concrete syntax definition is illustrated with Fig. 2. Here, the same statechart sentence is shown both as an raw instance of the metamodel and, in a more intuitive form, using the intended concrete syntax. For the sake of keeping Fig. 2(a) compact, events have been omitted in the metamodel instance.

This simple example reveals already some of the weaknesses of informal concrete syntax definitions. The mapping of concepts to visual objects as given in Table 1 must be complemented by comments stating that the name of a composite state is optional in the upper part of the symbol whereas the lower part optionally shows the representation of the substates. A transition representation (an arrow) starts in one representation of the transition's source and ends in one representation of its target. If a symbol contains parameters as placeholders for additional information, e.g. transitions are supplemented with events, then it must be specified where the information come from, e.g. that the event attached to a transition is indeed the same event that triggers the transition. Another problem is that there may be different icons for the same concept, as for `PseudoState`. Here, it is necessary to describe precisely all conditions for the selection of the correct icon. It is also possible to represent the same concept with variants of the same icon. For instance, a composite state is displayed with or without its name what requires to display or to suppress the name compartment of the symbol.

3 A Scheme-Based Approach to Concrete Syntax Definition

This section presents our approach to define a concrete syntax of a given language. We concentrate here on the definition of a graphical syntax for two reasons. First, most modeling languages provide nowadays an (often informally defined) graphical notation. Second, graphical notations are more challenging as, for example, purely textual notations. In fact, our approach can also be applied for the definition of textual notations. In this case, the display classes on the concrete syntax layer would represent tokens and would be extensions of `String` with additional attributes to encode the location of the currently represented model element.

The concrete syntax is defined by a set of *display schemes*. A display scheme is attached to each metaclass of the metamodel. Although schemes have a formal structure and can be processed by tools, the syntax definition they provide is nevertheless easily accessible by humans.

The scheme-based approach differs in two respects from related approaches. First, we do not aim to define a completely new language but concentrate just on the concrete syntax. This goal is different from what most approaches based on graph grammars aiming at. They define a language from scratch and have to capture in a way both the concrete and the abstract syntax of a language. In most cases, the clear separation between abstract and concrete syntax gets lost. Second, a scheme-based syntax definition intentionally ignores many problems related to tool support for the defined syntax. For example, a graphical editor usually stores the elements of both the abstract syntax layer (the model elements) and the concrete syntax layer (the display objects). The scheme-based syntax definition will provide simple criteria to decide whether or not the model elements are represented correctly by the display objects. However, it is out of

the scope of the concrete syntax definition to describe the mechanisms how editors can keep abstract and concrete syntax layers in sync. For example, if the user of the editor creates a new visual object, then, internally, the editor has also to create an instance of the corresponding metaclass and to connect both instances.

3.1 Visual Language Theory

Almost each of todays modeling languages comes with a graphical representation in order to improve readability and usability. Thus, the concrete syntax of modeling languages is usually defined in terms of a visual language. For this reason, we summarize here the relevant basic terms from visual language theory before we explain our approach in detail in the next subsection.

A visual language describes a set of visual sentences which in turn are given by a set of visual elements. A visual element can be seen as an object characterized by values of some attributes. It depends on the language which attributes are important for a graphical element[1], some of the most frequently used attributes are *shape*, *color*, *size*, *position*, *attach regions*.

Visual elements are placed in the two- or higher-dimensional space (we assume two dimensional space in the rest of the paper, but our approach is not restricted to that). For some languages, classified in [7] as *geometric-based languages*, the position of visual elements is an important information. Other languages ignore the position of elements but focus on the connections between them (*connection-based languages*). In fact, most real-world languages show characteristics of both geometric-based and connection-based languages and are thus called *hybrid languages*. The strong classification into geometric-based and connection-based languages is notwithstanding extremely helpful since it uncovers the 'ingredients' a visual language can have.

For geometric-based languages there are two possibilities to encode the position of a visual element. If *absolute positions* are taken into account then a sentence consisting of a circle and a square placed at point (1,0) and (2,0), respectively, is different from the sentence where the circle is placed at (1,0) and the square is placed at (3,0). If relative positions (spatial relationships) are used then both sentences would be described as that the square is placed right from the circle. Some of the most frequently used spatial relationships are *right*, *up*, *contain*, *overlap* (see [7] for a more complete list). It heavily depends on the visual language which of the spatial relationships are considered to be important. Sometimes, languages are geometric-based even if it seems that the visual elements can be arranged freely. One example is the language of UML class diagrams. At a first glance, rectangles for classes can be placed freely at any point in the space. For instance, a diagram consisting of two rectangles labeled with `A` and `B` would always be read as the same sentence no matter where the (rectangles

[1] There is a common classification of attributes into graphical, syntactical and semantic attributes. Only the first two classes of attributes are relevant for our approach because semantic attributes are already captured by the abstract syntax definition.

for) class `A` and `B` are placed. However, there is one exception from this rule: If - lets say - the rectangle for `B` appears completely inside the rectangle for `A`, then the class `A` is read to be composed of class `B`. Thus, the spatial relation *contain* is important to define the visual representation of class diagrams whereas the relations *right*, *up*, etc. do not play any role here.

Connection-based languages allow visual elements to be placed arbitrarily in the space. None of the spatial relationships has an influence on the parsing of a sentence of such languages. Instead, it is an important information whether two elements are connected by a connector (usually a line, polyline, curved line) or not. Connectors start and end in special regions of visual elements, so-called *attach regions*. A visual object can have one or more attach regions which sometimes collapse to attach points. As already mentioned, visual language definitions formalize an attach region of a visual element just as an attribute of it. This abstracts from the problem to define where an attach region is exactly located in respect of the visual element (e.g. in the lower right corner). However, some symbol editors, e.g. VLDESK [10] or AToM^3 [11], allow to exactly define the position of attach regions inside a visual element. They also solve the very similar problem of defining a shape for visual elements.

3.2 Scheme-based Definition of Concrete Syntax

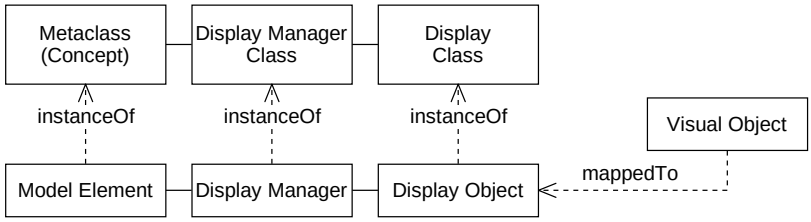


Fig. 3. Scheme definition architecture

The definition of a concrete syntax means to define (1) a visual language, i.e. visual elements with relevant attributes and relationships between them and (2) how the visual elements are connected to the concepts of the language they are supposed to represent. Figure 3 gives an overview how both goals are basically achieved by our approach: A sentence of a visual language, i.e. a set of visual objects, is first mapped to a set of display objects. This mapping and the formalism to define the graphical rendering of visual objects is intentionally left open in our approach. We have experienced with Scalable Vector Graphics (SVG) [12], a language to describe diagrams, but other formalisms or existing tools as symbol editors can be applied for this purpose as well.

Display classes declare for display objects attributes and operations what helps to lift up the abstraction level on which the syntax definition is given. An

attribute of a display object summarizes the value of more low level attributes of the underlying visual object such as *xpos*, *ypos*, *size*, *shape*, *color*, etc. The operations of a display object – as we will see later, operations correspond to spatial relationships such as *contain*, *overlap*, etc. – have to be implemented by the underlying visual object. If SVG is taken as a formalism to describe visual objects, the implementation can be done smoothly. If another formalism is taken, some additional adapter classes might be required.

The connection between display objects and model elements is given by display managers which are attached to model elements. Usually, each metaclass is connected with exactly one display manager class that in turn is connected with the display class defining the graphical representation. The criteria for a syntactically correct representation are defined in form of OCL invariants attached to the display manager classes. A set of display objects is a syntactically correct representation of a model, i.e. a set of model elements, if and only if the display managers attached to the model elements satisfy all invariants of the display manager classes.

3.3 A Concrete Syntax Definition for Statecharts

We illustrate our approach with a formal definition of the concrete syntax of statecharts whose abstract syntax was given in Sect. 2. Prior to the formal definition, an informal version of it should illustrate the gap between the abstract and concrete syntax, as already introduced in Sect. 2:

Problem 1 A text is shown on the top of transitions to represent the triggering event if it exists;

Problem 2 Depending on the viewer's choice, a composite state is depicted either by a text showing the name of the composite state, or by a region showing the contents of the composite state (i.e. its contained states), or both. In the latter case, the two regions are separated by a line;

Problem 3 The plain side of the transition icon is connected to a representation of its source state; the arrow side is connected to a representation of its target state;

Problem 4 The shape of a pseudo-state representation depends on its kind.

Figure 4 shows the backbone of the statechart concrete syntax definition. Four display schemes for graphically representable concepts of statecharts are defined: `Transition`, `SimpleState`, `CompositeState`, and `PseudoState`. The display scheme for `FinalState` has been omitted for the sake of brevity. All other concepts defined in the metamodel are either abstract (`ModelElement`, `State`) and thus will be depicted by the scheme of their subclass, or are displayed implicitly by the concepts they are attached to (`StateMachine` by its top state and `Event` by the transition it triggers). Note that the missing display scheme for `Event` might make the graphical representation of a model incomplete. If an event does not trigger any transition (according to the metamodel it is not mandatory to trigger at least on transition) then this event is not shown in the representation.

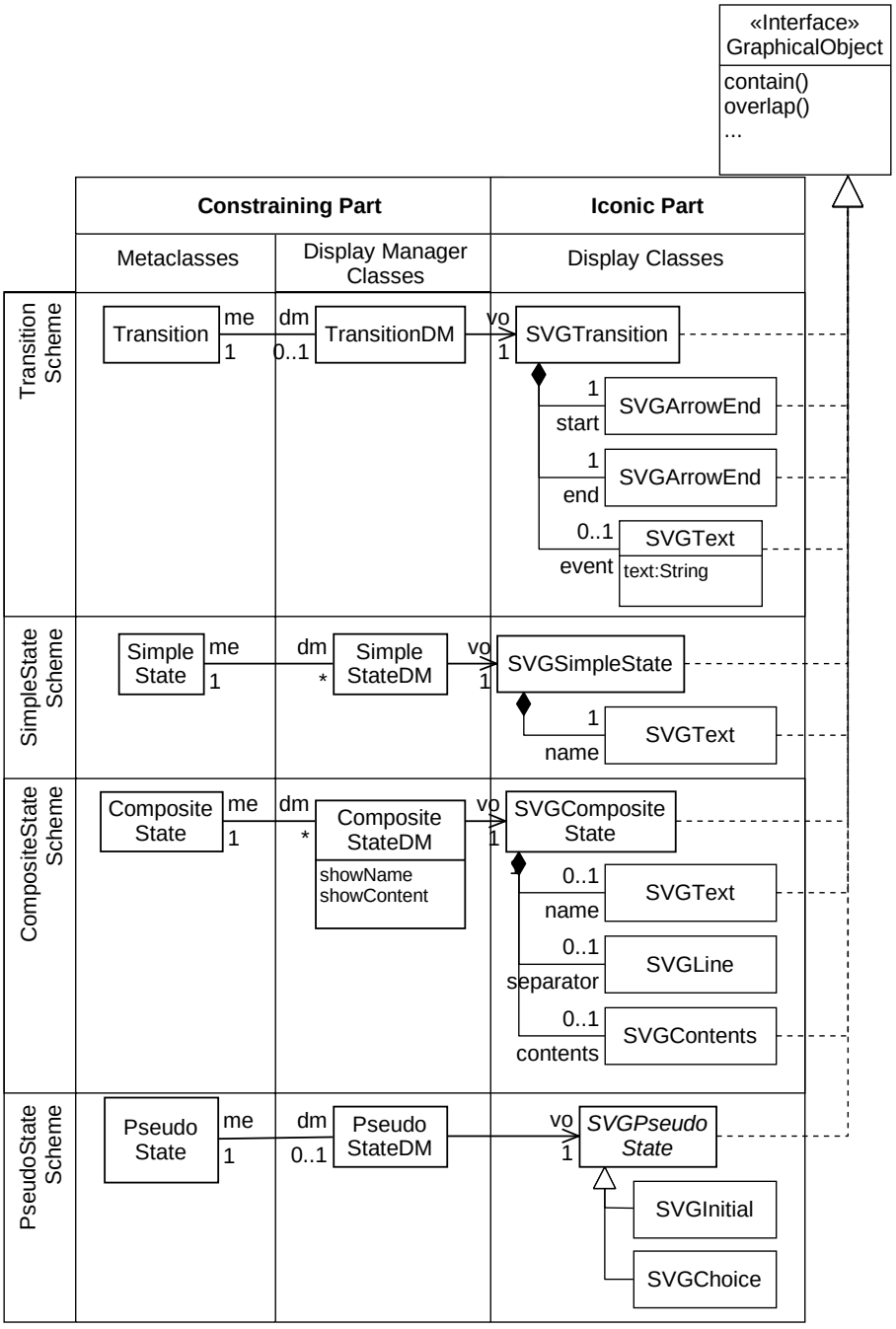


Fig. 4. Display schemes for statechart metaclasses

Each display scheme can be split into two parts. The iconic part defines the graphical rendering of visual objects. The constraining part fills the gap between the model elements and the display objects. A display manager class is connected to exactly one metaclass by the association end `me` (for model element); thus, every display manager refers to exactly one model element. The cardinality of the opposite association end `dm` (for display manager) encodes how many different display managers can exist for each model element. Following the syntax definition given in Fig. 4, model elements of `Transition` and `PseudoState` can only be depicted at most once whereas instances of `SimpleState` and `CompositeState` may be represented arbitrarily often. Thus, also such representations of a statechart are syntactically correct that omit parts of the model or show some states more than once.

Iconic part of a Scheme For each display manager class there is always a standard association to the corresponding display class with multiplicity 1 and role name `vo` (abbreviation for visual object) on the end of the display class. A display class represents an abstraction of visual objects that have to be defined in terms of *shape*, *color*, etc. It also declares some query facilities. Some standard queries, as introduced in Sect. 3.1, are declared in interface `GraphicalObject` that must be implemented by every display class. This ensures that any display object is capable to respond to such queries. As seen below, queries are heavily used in the OCL invariants attached to display manager classes.

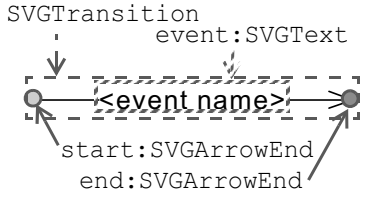


Fig. 5. The icon for `Transition`

Often, a model element is not displayed just by one atomic visual object but rather by a composition of such objects. Thus, the *main display object* linked to the display manager is composed of sub-objects whose position, size, etc., is controlled by the main display object. Figure 5 illustrates the internal definition of `SVGTransition`, the display class to represent transitions. Objects of `SVGTransition` are composed of one sub-object to display the event and two sub-objects representing attach points. Whereas the sub-object to display the event is optional, the two sub-objects of type `SVGArrowEnd` are mandatory.

Besides composing display classes, it is also sometimes necessary to subclass them. The class `SVGPseudoState` is an example. The concept `PseudoState` is

represented depending on the value of attribute `kind` by completely different shapes. Each of these shapes is defined by a single display class (e.g. `SVGInitial` for initial states, `SVGChoice` for choices) that inherits from `SVGPseudoState`. The class `SVGPseudoState` itself is declared as abstract.

Constraining part of a Scheme Based on the backbone shown in Fig. 4, the relationship between abstract and concrete syntax layers can be formalized by OCL invariants. In order to illustrate the expressive power of these formal constraints, we discuss now each of the four, already sketched problems of informal syntax definitions.

Problem 1 requires to keep attribute values for model elements and representing display objects in sync. This problem can be resolved by the following constraint:

```
-- Problem 1
   context TransitionDM
   inv: if self.me.trigger->isEmpty()
        then self.vo.event->isEmpty()
        else self.vo.event.text = self.me.trigger.name
        endif
```

The constraint ensures that the correct name of the event is displayed whenever a transition is triggered by an event. Sometimes, it might be appropriate to relax this rule so that the triggering event of a transition can be suppressed in the graphical representation. In this case, the invariant looks as follows:

```
-- Problem 1 with optional event display
   context TransitionDM
   inv: self.vo.event->notEmpty() implies
        self.vo.event.text = self.me.trigger.name
```

Problem 2 is an example for user-directed representation policies which can be encoded by attributes of type `Boolean` in the display manager class. Problem 2 is captured by the following constraint:

```
-- Problem 2
   context CompositeStateDM
   inv: self.showName = self.vo.name->notEmpty() and
        self.showContent = self.vo.contents->notEmpty() and
        (self.showName and self.showContent)
        = self.vo.separator->notEmpty()
```

Problem 3 is similar to Problem 1 but the synchronization between model and representation cannot be achieved by constraining the values of attributes. Instead, the spatial relationships between display objects have to be taken into account. This can be done by an OCL invariant due to the declaration of `overlap(GraphicalObject):Boolean` in interface `GraphicalObject` that is implemented by all display classes. Note that the semantics of the query `overlap` is hidden in the implementation of the visual objects. Thus, the 'correctness' of the

OCL invariant depends on the 'correctness' of the implementation of `overlap` in the visual objects.

```
-- Problem 3
context TransitionDM
inv: self.me.source.dm.vo->one(svo|
                    self.vo.start.overlap(svo)) and
     self.me.target.dm.vo->one(tvo|
                    self.vo.end.overlap(tvo))
```

Problem 4 is an example for the representation of modeling elements belonging to the same concept (`PseudoState`) by different shapes. The OCL invariant takes advantage of OCL's ability to check the actual type of an expression with `oclIsKindOf()`.

```
-- Problem 4
context PseudoStateDM
inv: let kindAsso:Set(TupleType(kind:PseudoStateKind,
                                type:OclType)) =
           Set{Tuple{kind = PseudoStateKind::initial,
                     type = SVGInitial},
               Tuple{kind = PseudoStateKind::choice,
                     type = SVGChoice}}
     in
          self.vo.oclIsKindOf(
             kindAsso->any(t| t.kind = self.me.kind).type)
```

4 Related Work

The problem of defining a graphical concrete syntax on the top of a metamodel has already been addressed by the OMG and numerous authors.

The OMG has adopted a standard for diagram interchange for UML2.0 (UML-DI [13]) to overcome the shortcomings of model interchange based on XMI. Indeed, XMI focuses on transmitting pure modeling data, given by the abstract syntax of models, and ignores graphical information. UML-DI provides a generic metamodel for extending any other metamodel so that graphical information can also become part of the data interchanged in XMI. However, UML-DI only concentrates on gathering graphical data and does not focus on how those data are structured. Consequently, these data are still ambiguous and tools interchanging them still need to agree on their meaning. For instance, the UML-DI meaning for UML is defined using an XMI to SVG translator that cannot be reused for a completely new language. Moreover, neither UML-DI nor XMI-to-SVG translators capture spacial relationships in order to express, for example, that two graphical elements do overlap.

Other approaches, like XMF [14], argue that the concrete syntax involves a representation language. An example of such languages is Scalable Vector Graphics (SVG) [12], but the XMF framework provides its own graphical language in form of a representation metamodel with well defined semantics. Here, semantics

corresponds to a graphical representation rendering. Thus, to define the representation of a given language whose abstract syntax is given by a metamodel, it is sufficient to define a model transformation between the metamodel of the language and the representation metamodel.

Another approach is taken by most meta-CASE tools, like GME [15], DOME [16], MetaCASE [17], or AToM^3 [11]. In principle, they define a representation template for each metaclass in the abstract syntax. A template includes a set of representation language constructs, as instances of the representation language metamodel, together with some holes to make variants in the representation possible. Again, each of these tools impose its own graphical language. When a model element has to be represented, the holes are replaced depending on relevant information from the current model. Unfortunately, while most of these tools provide a constraint language that can be used to impose restrictions on the abstract syntax, they do not provide access to the concrete syntax. A notable exception is AToM^3 that allows constraints written in Python to select among variations of the icons. However, also in AToM^3 the definition of the concrete syntax is done at a much lower level as our approach which uses OCL as the main language to specify the concrete syntax.

Graph-grammar based language definitions (as Triple-Graph-Grammar [18], GenGED [19]) are constructive and aim at finding a derivation for a given diagram. In addition to rules, GenGED offers the possibility to attach constraints to the concrete syntax classes (called type graph nodes in the GenGED terminology), but the purpose of the constraints is merely the computation of a possible layout for a diagram. The language definition itself is still based on graph grammar rules (see [20] for the GenGED definition of the same statechart fragment as we used here for illustration).

5 Conclusion

We have presented a way to specify the concrete syntax of languages whose abstract syntax is already available in form of a metamodel. The main idea is to complement the metamodel with display schemes.

The iconic part of a scheme defines some display classes for representing model elements. Variants in the representation are expressed by attributes or methods attached to the display classes. We do not impose any language to define display classes but assume that display classes do implement the interface `GraphicalObject`.

The constraining part of a scheme consists of a display manager class together with associations to metaclasses and display classes as well as a number of constraints. The purpose of the constraints is to stipulate restrictions for the visualization of model elements. The expressive power of constraints has been illustrated by applying them on a simplified version of the statechart language.

Our scheme-based approach resides at a higher level abstraction than most other approaches. Except for the shape information for the icons, only OCL constraints have to be attached to display manager classes. The relatively high

number of new classes that must be defined is outweighed by the fact that many of these classes can be defined mechanically.

We are currently implementing our approach in form of a free editor that is customizable with modeling language specifications. The user is able to place different symbols on a canvas and by creating a symbol (instance of a display class) the corresponding display manager and model element is created as well. At any time, the user can ask the editor whether the current diagram is syntactically correct or not. Internally, the editor evaluates then all OCL constraints attached to the extended metamodel. This check might be costly if implemented naively because each constraint must be checked for a rather high number of objects. A solution for this problem is to use strategies to determine only those constraints that might be broken by the changes in the diagram made after the last check (see [21]). However, the efficiency aspect becomes less important if our free editor is seen as a reference implementation for concrete syntax definitions. Other tool vendors, that have implemented the same language using more efficient techniques, could test whether their tools comply with the formally given syntax of the modeling language or not.

References

1. Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 542–552, 1996.
2. Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.
3. Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.
4. Terry Halpin. *Information Modeling and Relational Databases : From Conceptual Analysis to Logical Design*. Morgan Kaufmann, second edition, 2003.
5. OMG. Meta-Object Facility (MOF) 1.4. OMG Document formal/02-04-03, April 2002.
6. OMG. XML Metadata Interchange (XMI) 2.0. OMG Document formal/03-05-02, May 2003.
7. Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573–600, 2002.
8. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
9. David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
10. Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(4):431–487, 2004.

11. Juan de Lara and Hans Vangheluwe. Using AToM^3 as a meta-case tool. In *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649, 2002.
12. W3. Scalable Vector Graphics (SVG) 1.1 Specification, January 2003.
13. OMG. UML 2.0 diagram interchange specification - final adopted specification. OMG Document ptc/03-09-01, September 2003.
14. Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language-driven development. Available at http://albini.xactium.com, 2005.
15. Matthew J. Emerson, Janos Sztipanovits, and Ted Bapty. A MOF-based metamodeling environment. *Journal of Universal Computer Science*, 10(10):1357–1382, 2004.
16. Honeywell. Dome users guide. http://www.htc.honeywell.com/dome/support.htm, 2000.
17. MetaCase. Abc to metacase technology. http://www.metacase.com/papers, 2004. White Paper.
18. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, volume 903 of *LNCS*, pages 151–163. Springer, 1995.
19. GenGED Team. GenGED homepage. http://tfs.cs.tu-berlin.de/~genged/, 2005.
20. Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, and Gabriele Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In *Proceedings of 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *LNCS*, pages 214–228. Springer, 2004.
21. Jordi Cabot and Ernest Teniente. Determining the structural events that may violate an integrity constraint. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 320–334. Springer, 2004.