Composing Visual Syntax for Domain Specific Languages

Luis Pedro[1], Matteo Risoldi[1], Didier Buchs[1],
Bruno Barroca[2], and Vasco Amaral[2]

[1] Centre Universitaire d'Informatique
Université de Genève, Switzerland
{Didier.Buchs,Luis.Pedro,Matteo.Risoldi}@unige.ch
[2] Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal
{Bruno.Barroca,Vasco.Amaral}@di.fct.unl.pt

Abstract. With the increasing interest in metamodeling techniques for Domain Specific Modeling Languages (DSML) definition, there is a strong need to improve the language modeling process. One of the problems to solve is language evolution. Possible solutions include maximizing the reuse of metamodel patterns, composing them to form new, more expressive DSMLs.

In this paper we improve the process of rapid prototyping of DSML graphical editors in meta-modeling tools, by defining composition rules for the graphical syntax layer. The goal is to provide formally defined operators to specify what happens to graphical mappings when their respective metamodels are composed. This improves reuse of Domain Specific Modeling Languages definitions and reduces development time.

1 Introduction

With increasing reports of success stories both in industry and academy, Software Languages Engineering is becoming a more mature field. It is supported today by proper conceptual and practical tools. Several domains of application are now receiving benefit from the clarity and flexibility of specification that a Domain Specific Modeling Language (DSML) approach provides.

Among supporting tools, some allow providing a DSML with a graphical syntax and an editor, based on the DSML's metamodel. However, providing a graphical syntax to a DSML is still a very time consuming task. The main reasons are that it is a task with a very low level of reusability and that integration with the supporting tools is difficult. This is particularly relevant in the DSML prototyping phase where modifications are performed very often.

Our motivation is improving the state of the art in what concerns rapid development of graphical editors for DSMLs. In previous work, we defined what happens to the semantic mappings of DSMLs (expressed by model transformations) when two DSML metamodels are composed. This work goes further in defining a layer for graphical syntax composition. Operators are defined for the

J.A. Jacko (Ed.): Human-Computer Interaction, Part II, HCII 2009, LNCS 5611, pp. 889–898, 2009.

composition of metamodels. We explore the consequences of metamodel composition on graphical syntax and on the generation of graphical editors. We use an example to illustrate the technique.

Section 2 will introduce some domain concepts we will use for composition examples. A metamodel and a graphical syntax will be given for each domain concept. Section 3 will define the composition operators and show how to compose the example metamodels and their associated graphical syntax. Section 4 will reference related work. Section 5 will resume the article and discuss limits and future work.

2 Case Study

Fig. 1 shows three simple metamodels. They represent very simple DSMLs, each of which can be used to model a different concept. The first (a) allows the specification of hierarchies of `Object`s. An `Object` may have one or more children `Object`s. The second metamodel (b) allows the specification of finite state machines (FSMs). A `State` may have outgoing `Transition`s, each one with a destination `State`. A `State` has an attribute `initial` (a boolean). The third metamodel (c) allows the specification of an event trigger. A `Trigger` is associated to one or more `Event`s, which are triggered when the `Trigger` is activated.

Each of these metamodels has an associated graphical syntax, shown in Fig. 2. The graphical syntax for objects (a) represents an `Object` as a rounded box, and the `children` relationship as the containment of boxes (children are inside parents). The graphical syntax for FSMs (b) represents a `State` as a box, and a `Transition` as an arrow linking boxes. The direction of the arrow is given by the relationships. The `initial` attribute of states is represented by a dot in a corner of a state box (dot=true; no dot=false). The graphical syntax for triggers

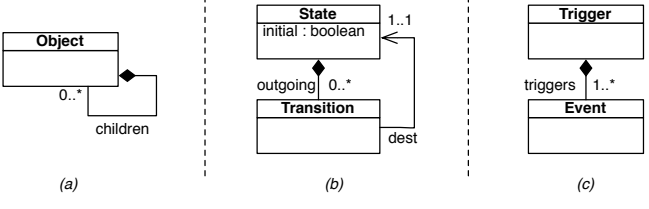

Fig. 1. Object, FSM and Trigger metamodels

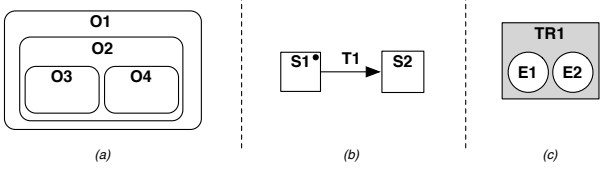

Fig. 2. Object, FSM and Trigger graphical syntax

(c) represents a `Trigger` as a gray box, containing a circle for each triggered `Event`. In all cases, visual elements are labeled with the elements ID.

In the following section we will show how, given these metamodels, a developer can define DSMLs to address each of the following example requirements:

1. Each object must contain an FSM.
2. Each object must be associated to an FSM. Several objects can be associated to the same FSM.
3. In an FSM, reaching a state may trigger events.
4. An FSM can be hierarchical; a state can contain an FSM. Getting to that state leads to the initial state of the internal FSM. From each internal state it is always possible to leave the external state.

3 Composition Approach

The composition approach extends previous work [13] on composition operators for DSML metamodel blocks. This work perceives DSML definition as an incremental task: a DSML's metamodel is created by composing several sub-metamodels, or *domain concepts*. Each concept is associated to one or more concrete graphical syntaxes. The semantic aspects of composition have been tackled in [13]. The work presented in this article goes further in exploring the syntactic aspects of composition at the graphical layer.

We use four composition operators [7] to compose metamodels and reuse graphical definitions. We assume that a domain concept is a metamodel with an associated graphical syntax and that in the process of developing a DSML several concepts are going to be composed. This section will give details about what happens to graphical syntax when these composition operators are used on metamodels. For all the examples and definitions that follow it is also assumed that there are three metamodels: a left metamodel mm_{left}; a right metamodel mm_{right}; and a resulting composed metamodel mm'. A composition of metamodels $Comp$ is defined by a mapping function $\varphi_{operator}$ as

$$mm' = Comp(mm_{left},\ mm_{right},\ \varphi_{operator})$$

where *operator* stands for the name of the composition operator to be specified. The metamodels used as an example are those presented in Fig. 1. Their associated graphical syntax is shown in Fig. 2.

3.1 Containment

This operator allows to create containment relations to provide hierarchical constructs. The hierarchy is defined by an aggregation relationship from one class of the left metamodel to a set of classes of the right metamodel. The containment operator is defined by $\varphi_{containment}$, a total function creating a map between elements of mm_{left} and mm_{right}:

$$\varphi_{containment} : mm_{left} \rightarrow mm_{right}, C$$

where C stands for the cardinality of the aggregation relationship between the metaclass of the mm_{left} metamodel and the metaclass(es) of the mm_{right} metamodel.

As an example, let mm_{left} be the `Object` metamodel in Fig. 1 (a) and as mm_{right} the `State` metamodel in Fig. 1 (b). Let us assume that the developer wants to define a language to specify an FSM inside each modeled object. The mapping function is defined as:

$$\varphi_{containment} = \{\langle Object, \{\langle State, 1..1\rangle\}\rangle\}$$

The containment operator is applied to obtain a composed metamodel, shown in Fig. 3. The composition produced a `state` relationship which aggregates the `State` class inside the `Object` class.

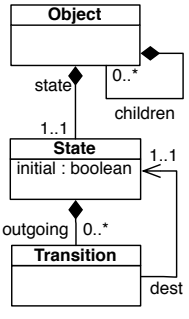

Fig. 3. Result of application of $\varphi_{containment} = \{\langle Object, \{\langle State, 1..1\rangle\}\rangle\}$

The pattern produced by the composition (the `state` aggregation) is reflected in the graphical syntax by the possibility of defining states inside objects, as shown in Fig. 4. The graphical symbols for objects, states and transitions do not change.

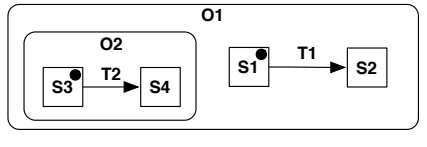

Fig. 4. Composed visual syntax for containment between `Object` and `State`

3.2 Association

The association operator is very similar to the containment operator in what concerns metamodel composition. However, instead of creating an aggregation, this operator creates a reference between a class of the mm_{left} metamodel and a class of the mm_{right} metamodel. The association operator is defined by the $\varphi_{association}$ mapping function:

$$\varphi_{association} : mm_{left} \rightarrow mm_{right}, C$$

where C is the cardinality of the reference between the metaclass of the mm_{left} metamodel and the metaclass of the mm_{right} metamodel.

As an example, let mm_{left} be the `Object` metamodel in Fig. 1 (a) and as mm_{right} the `State` metamodel in Fig. 1 (b). The developer wants to define a language to specify an FSM for each object; however, rather than defining the FSM inside the object, he just wants objects to reference an FSM, so that the same FSM can be used for several objects. The mapping function is defined as:

$$\varphi_{association} = \{\langle Object, \{\langle State, 1..1\rangle\}\rangle\}$$

Fig. 5 shows the result of applying the association operator. The composition produced a `state` reference, which relates the `Object` and `State` classes.

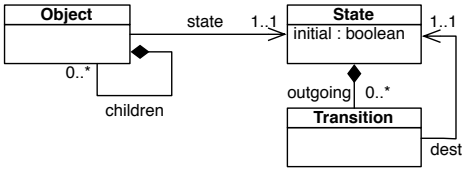

Fig. 5. Result of application of $\varphi_{association} = \{\langle Object, \{\langle State, 1..1\rangle\}\rangle\}$

The pattern produced by the composition (the `state` reference) is reflected in the graphical syntax by the possibility of creating a dashed arrow from an object to a state, as shown in Fig. 6. In this case too, graphical symbols for objects, states and transitions do not change.

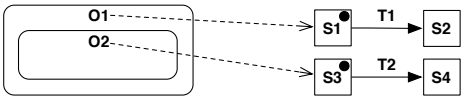

Fig. 6. Composed visual syntax for association between `Object` and `State`

3.3 Inheritance

The inheritance operator allows to compose metamodels with a UML-like inheritance concept. It specifies that a class of the left metamodel is specialized by a class of the right metamodel. The inheritance operator is defined by the $\varphi_{inherit}$ mapping function:

$$\varphi_{inherit} : mm_{left} \rightarrow mm_{right}$$

where the metaclass in mm_{left} is the specialized class and the metaclass in mm_{right} is the specialization class.

As an example, let mm_{left} be the `Object` metamodel in Fig. 1 (a) and as mm_{right} the `Trigger` metamodel in Fig. 1 (c). The developer wants to define a

language for specifying FSMs, in which reaching a certain state triggers events. The mapping function is defined as:

$$\varphi_{inherit} = \{\langle State, Trigger \rangle\}$$

Fig. 7 shows the result of applying the inheritance operator. The `Trigger` class specializes the `State` class.

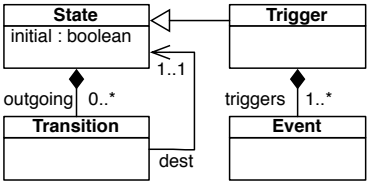

Fig. 7. Result of application of $\varphi_{inherit} = \{\langle State, Trigger \rangle\}$

The specialization pattern is reflected in the graphical syntax by substitution of the graphical symbol of the specialized class by that of the specializing class, as shown in Fig. 8. This allows defining FSMs in which a transition has a trigger – instead of a state – as source and/or destination. The graphical symbol for a trigger can be used in place of the graphical symbol for a state. Triggers and states keep their original graphical definition (e.g. one can still define events inside triggers, but not inside states).

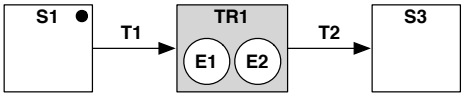

Fig. 8. Composed visual syntax for inheritance between `State` and `Trigger`

3.4 Parameterized Merge

This operator is used as "join point" to compose two DSMLs. Originally, package merge was specified in the MOF specification [12]. Package merge allows the merging of packages to specialize metamodel classes having matching names. The necessity for metamodel classes to have matching names is however a strong limitation, which is not always satisfied.

In a more practical approach we defined the parameterized merge operation. φ_{merge} is a total function that creates a map between elements of mm_{left} and mm_{right}

$$\varphi_{merge} : mm_{left} \rightarrow mm_{right}$$

The merge between mm_{left} and mm_{right} is not performed on metamodel classes with matching names, but rather by defining a mapping fuction between classes:

- φ_{merge} is a function that maps metaclasses of the left and right metamodels;
- the merge is done on mm_{right} meaning that the element of mm_{right} defined in the φ_{merge} function keeps its name.

As an example, let mm_{left} be the `Object` metamodel in Fig. 1 (a) and as mm_{right} the `State` metamodel in Fig. 1 (b). Let us say that the developer wants to define a language to specify hierarchical finite state machines [3]. The mapping function is defined as:

$$\varphi_{merge} = \{\langle Object, State\rangle\}$$

The merge operator is applied to obtain a composed metamodel, shown in Fig. 9. The new `State` class has all attributes and relationships from the old `State` and `Object` classes, including the `child` aggregation.

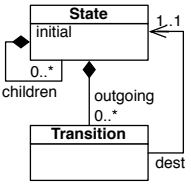

Fig. 9. Result of application of $\varphi_{merge} = \{\langle Object, State\rangle\}$

Here, the pattern is a graphical representation of a class having features of both merged classes. At the graphical syntax layer, it means that the graphical symbol for the `State` class keeps its graphical notation; however the newly added `child` aggregation makes it possible to define states inside other states (just like objects of the `Object` metamodel could be defined inside other objects). The resulting graphical syntax is shown in Fig. 10.

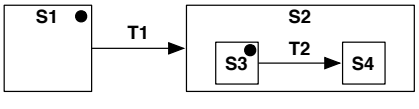

Fig. 10. Composed visual syntax for parameterized merge between `Object` and `State`

4 Related Work

Several tools can be found which are able to generate graphical editors from the metamodel of a language. These tools have different approaches to specify the concrete graphical syntax for DSMLs.

The Graphical Modeling Framework (GMF) [4] is an Eclipse project, with which a DSML's graphical editor can be generated. The resources for editor generation are the EMF metamodel of the language and several additional presentation models (gmfgraph, gmftool, gmfmap) mapped to the EMF metamodel.

The presentation models define the DSML's concrete syntax (diagram elements, toolbar options, etc).

The Generic Modeling Environment (GME) [6] is a highly configurable metamodeling tool. Instead of having external descriptions for the concrete syntax, GME gives the ability to decorate the DSML's metamodel with special entities called *views*. For each element of the metamodel the designer can define a concrete visual syntax in the generated DSML's graphical editor.

Meta-CASE editors (e.g. MetaEdit+ [9]) are environments capable of generating CASE tools. They allow creating tool definitions in a high-level graphical environment. However, the user interface is coded manually. These environments store concrete syntax definitions directly in the metamodel properties.

Another framework is the Diagram Editor Generator (DiaGen) [11], which is an efficient solution to create visual editors for DSMLs. DiaGen is not based on metamodeling techniques; it uses its own specification language for defining the structure of diagrams. DiaGen supports editing of the concrete syntax in a graphical context, but in a tree control-based form only, where there is no support to define the graphical shape of elements. Concrete syntax in DiaGen is based on properties. DiaGen can generate an editor based on the specification using hypergraph grammars and transformations.

AToM3 (A Tool for Multi-formalism and Meta-Modelling) [1]) is a flexible modeling tool. It employs an appearance editor to define the graphical shape of model elements. It uses model properties to store the concrete syntax (model definitions are extended with visualization-based attributes). AToM3 can generate plug-ins that use the defined syntax, but the code generation is not based on a presentation DSL. Model views are generated with triple graph grammars.

Visual Modeling and Transformation System (VMTS) [10] is an n-layer metamodeling environment that unifies the metamodeling techniques used by the common modeling tools. It employs model transformation applying graph rewriting as the underlying mechanism. Concrete syntax definitions are created by instantiating VMTS's Presentation DSL (VPD) and they define how model items are visualized. These definitions are afterwards mapped to the DSML's abstract syntax.

Although these modeling frameworks support concrete graphical syntax definitions, very little work has been done to support the composition of these definitions.

In [8] and [5], the first steps of DSML composition in GME are presented. In these works the composition operators (e.g. metamodel interfacing, class refinement) only affect the metamodel elements referring to the DSML's abstract syntax. The corresponding effects on the visual concrete syntax are left implicit, as part of the composition operator's semantics. In [7] an overview over the several composition operators for metamodel composition is provided, on which our work is based.

In [2], a new composition operator was introduced: template instantiation, in the context of GME. Again this composition operator does not focus on the visual concrete syntax, but only on the abstract syntax.

In [13] several new metamodel composition methods were presented, both at the syntactic and semantic level. These methods can be parameterized to specify what are the consequences of metamodel composition at the semantic level of a DSML (expressed as a model transformations to a target platform).

5 Conclusions

We presented a proposal for automatically mapping composition patterns to graphical syntax. A limitation of the presented work is that the composition operators might not cover all possible compositions. We might have a case in which we want to apply the metamodel composition operator and not have the composition of the graphical syntax. For example, we might want to define instances of class `State` inside class `Object`, but at the same time be able to have instances of `State` which are not contained in any object. This would require either manual composition, or identification of more complex composition patterns. Also, there are cases in which the proposed graphical representations for patterns might create ambiguity with pre-existing graphical notations of the composed metamodels. For example, if a metamodels already uses dashed arrows in its graphical syntax, then applying the Association operator to it might introduce more dashed arrows with a different meaning than the pre-existing ones. However, based on mentioned related work, we think the presented operators cover a large number of use cases.

Future work includes the identification of more complex patterns based on case studies. In order to do this, we have to explore the implementation of the proposed approaches using standardized tools. Our current point of view is that the most viable frameworks for implementation are GEF and GMF. Implementation in GEF might be easier to develop as the graphical syntax definition is at a lower level. The level of abstraction of definitions in GMF could make implementation more difficult; however, from the end user point of view it might be more usable.

Although further investigation is needed, we think that this approach is relevant in the context of defining the graphical layer of DSMLs. This task is usually non-trivial in current language development environments. Being able to reuse previously defined graphical patterns can boost DSML development productivity.

References

1. de Lara, J., Vangheluwe, H.: Using AToM3 as a Meta-CASE tool. In: 4th International Conference on Enterprise Information Systems, pp. 642–649 (2002)
2. Emerson, M., Sztipanovits, J.: Techniques for Metamodel Composition. In: OOPSLA 6th Workshop on Domain Specific Modeling, pp. 123–139 (2006)
3. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(6), 742–760 (1999)

4. GMF Team. Eclipse graphical modeling framework, http://www.eclipse.org/modeling/gmf/ (visited in 2008).
5. Karsai, G., Maroti, M., Lédeczi, Á., Gray, J., Sztipanovits, J.: Composition and cloning in modeling and meta-modeling. IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling 12, 263–278 (2004)
6. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer 34(11), 4451 (2001)
7. Lédeczi, Á., Nordstrom, G., Karsai, G., Völgyesi, P., Maróti, M.: On metamodel composition. In: Proceedings of the 2001 IEEE International Conference on Control Applications, 2001 (CCA 2001), Mexico City, Mexico, pp. 756–760. IEEE Computer Society Press, Los Alamitos (2001)
8. Lédeczi, Á., Vólgyesi, P., Karsai, G.: Metamodel composition in the Generic Modeling Environment. In: Communications at workshop on Adaptive Object- Models and Metamodeling Techniques, ECOOP 2001 (2001)
9. MetaCase Consulting. Metaedit, http://www.metacase.com (visited in 2008)
10. Mezei, G., Levendovszky, T., Charaf, H.: A model transformation for automated concrete syntax definitions of metamodeled visual languages. In: Proc. of 2nd InternationalWorkshop on Graph and Model Transformation 2006, Electronic Communications of the EASST, vol. 4 (2006)
11. Minas, M.: Specifying graph-like diagrams with DiaGen. Electronic Notes in Theoretical Computer Science 72(2), 102–111 (2002); GraBaTs 2002, Graph-Based Tools (First International Conference on Graph Transformation)
12. Object Management Group members. Meta-Object Facility 2.0 core specification. Technical report, OMG (January 2007), http://www.omg.org/cgi-bin/doc?formal/2006-01-01
13. Pedro, L., Amaral, V., Buchs, D.: Foundations for a domain specific modeling language prototyping environmen: A compositional approach. In: Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM), University of Jyvaskylan (October 2008)