A Domain-Specific Language for Visualizing Modeling Languages

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf
{gmezei, tihamer, hassan}@aut.bme.hu

Abstract: Software development based on Domain-Specific Modeling Languages is one of the most focused research fields. Metamodeling techniques can describe the topological rules of the target domains and express the domain-specific constraints of visual languages, but the presentation still cannot follow this flexibility. The aim of our research was to create a flexible visualization method for metamodeling environments. The main concern of this paper is to present the concepts of a domain-specific modeling language for specifying the presentation of arbitrary DSLs and standard UML models. The introduced language can fully describe the visualization and behavior of the linked model. The concrete syntax definitions can be bound to the models of the target domain using model transformation.

Key Words: Metamodeling, Graphical environment, Domain Specific Language

1 Introduction

Using Domain-Specific Modeling Languages (DSML) is a widely adopted way to create customized models for special domains where generic modeling languages would fail. DSMLs always need an abstract syntax definition. Metamodeling is a means to avoid coding the language definitions manually and to create DSMLs. Metamodels act as the abstract syntax definition for the model level: they define the available model elements, their attributes, and the possible connections between them. Although metamodeling is flexible enough to fulfill the topological requirements of domain specific modeling, it does not define the presentation of the elements: the concrete syntax. The primary aim of this paper is to introduce a solution that can automate the creation and handling of the concrete syntax.

The lack of the standardization in the concrete syntax definition process made custom solutions necessary that are often ineffective and inflexible. The most popular solutions are: (i) manual coding the presentation logic in the modeling framework (ii) extending the DSL definitions with new properties focusing on the presentation, or (iii) using a DSL that defines the presentation, and then binds the concrete syntax and the abstract syntax definitions.

Manual coding is flexible, it can be customized simply, but it also means a great amount of additional work, and the standardization of the concrete syntax definitions is very hard to manage. The presentation logic can be metamodel-dependent, or metamodel-independent. The typical realization is the plugin-based architecture, where the presentation is coded in plugins assigned to a metamodel.

Concrete syntax based on the metamodel properties is also a popular solution. The presentation logic uses information extracted from the metamodel items to display a model item, thus model items with a common metamodel can be treated uniformly. The presentation can be customized using the presentation-focused properties of the metamodels. This solution has also its limits: if the method is not combined with manual coding, then either the customization facilities are strongly limited, or the core framework is extremely complex. Another weakness of the solution is, that different models with the same metamodel cannot be visualized differently. Finally, this solution does not clearly separate the content and the presentation usually, which can cause problems of tangled code for the editing process.

The third solution is more straightforward. The concrete syntax definitions are modeled using a common Domain-Specific Language (referred to as *Presentation DSL*), thus, the concrete syntax definitions are the models of the *Presentation DSL*. The ability to handle the concrete syntax in the same way as normal DSL models means uniformity, flexibility, and makes editing much easier. Another advantage of the solution is that it allows multiple concrete syntax definitions for a single DSL. The method is even more effective if concrete syntax models can be bound to the abstract syntax definitions automatically (e.g. the presentation logic can be transformed to source code that is used to edit the abstract syntax).

Visual Modeling and Transformation System (VMTS) [1] is an n-layer metamodeling environment that unifies the metamodeling techniques used by the common modeling tools, and employs model transformation applying graph rewriting as the underlying mechanism. Previous work [2] has presented the VMTS Presentation Framework (VPF). VPF offers many features such as plugin-based architecture, support for multiple canvases on a single model, and customizable event-handling for the model items. VPF promotes creating models for UML 2.0 diagrams and other popular domains such as Mobile Resource Editor. VPF supports metamodel-based validation of the models, for example the model topology is automatically checked when editing the models, and the attributes are validated using OCL constraints. VPF plugins must be customized for each DSML. VPF is based on the MVC architecture, therefore each plugin requires three classes to be defined (a Model, a View, and a Controller). These fundamental classes must be subclassed for each model element to provide the drawing code. Earlier the concrete syntax used by the VPF plugins was defined by manual coding, which meant a huge amount of additional work.

VMTS Presentation DSL (VPD) is a *Presentation DSL* realized in VMTS. VPD metamodel definies a metamodel for this language; VPD models can describe the concrete syntax of a domain. To edit the VPD Models, a plugin (the VPD Plugin) was implemented based on VPF. To improve the effectiveness of the solution, there is a support for processing VPD models automatically, using model transformation techniques [3]. The control flow of the transformation describes the transformation steps converting the VPD Models to a CodeDOM model. From the CodeDOM model, source code is generated with the .NET CodeDOM technology [4]. This source code implements a plugin: the generated plugin that can be used directly in VPF. Fig. 1 shows the main steps of the process. This approach made it possible to avoid manual coding, and create plugins in a user-friendly, graphical way in the same environment as common DSL models. The main goal of this paper is to introduce the VMTS Presentation DSL, whereas the model transformation and the code generation are not discussed. The transformation control flow and the transformation rules are to found in [1].

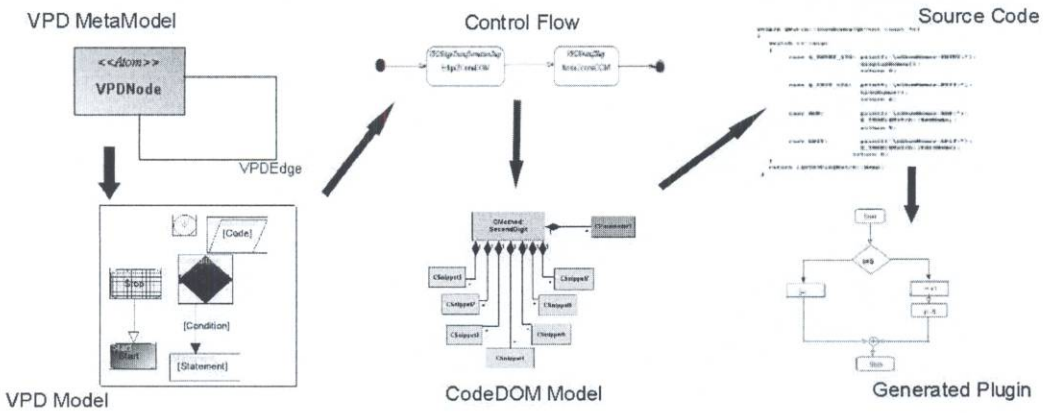

Fig 1. Plugin generation steps

2 Related work

Generic Modeling Environment (GME) [5] is a highly configurable metamodeling tool supporting two layers: a metamodel-, and a modeling layer. The concrete syntax definitions can be coded either manually, or set by properties both on the metamodel and on the model level. GME supports a special type of property definitions: the registry entries. These entries are assigned to model elements and they can also customize the appearance.

Meta-CASE editors (e.g. JKOGGE [6] or MetaEdit+ [7]) are environments capable of generating CASE tools. They allow creating the CASE tool definitions in a high-level graphical environment, but they supply manually coded user interface. These environments store concrete syntax in the properties of the metamodel elements.

Another framework is the Diagram Editor Generator (DiaGen) [8] that is an efficient solution to create visual editors for DSLs. DiaGen uses its own specification language for defining diagrams (i.e. the abstract syntax). DiaGen supports editing the concrete syntax in a graphical context, but in a tree control-based form only, where there is no support to define the shape of the elements graphically. DiaGen can generate an editor based on the specification using hypergraph grammars and transformations. Concrete syntax in DiaGen is based on properties.

AToM^3 (A Tool for Multi-formalism and Meta-Modelling) [9] is a flexible modeling tool. It employs an appearance editor to define the shape of the model elements graphically. AToM^3 can generate plugins that use the defined syntax, but it uses properties of the model for storing the concrete syntax.

Eclipse [10] is a flexible, open source modeling platform that supports metamodeling. Eclipse Modeling Framework (EMF) can generate source code from models defined by the class diagram definition of UML, but it does not contain concrete syntax definitions. Graphical Editing Framework (GEF) is also a part of the Eclipse project. GEF provides methods for creating visual editors. EMF does not support code generation for GEF, therefore GEF plugins require manual coding to support the concrete syntax.

GenGed [11] generates visualization code with graph transformation. It is rather presentation oriented: instead of domain specific modeling, it specifies graphical symbols, constraints and their connections; from this information graph rewriting rules (Alphabet Rules) are generated, which serve as the graph grammar used to parse the visual language. For the editing features, a graphical editor is also generated to support the newly created visual languages.

GenGed has been replaced by a new project Transformation-Based Generation of Modeling Environments (TIGER) [12] that uses precise visual language (VL) definitions and offers a graphical environment based on GEF. TIGER can generate source code from the visual language definitions that implements a plugin based on GEF. VL specifications can be created graphically. At the moment TIGER can generate editors for Activity Diagram and Petri nets.

Graphical Modeling Framework (GMF) is also an Eclipse project [13]. The goal of GMF is to form a generative bridge between EMF and GEF, whereby a diagram definition is linked to a domain model as input to the generation of a visual editor. GMF uses a *Presentation DSL* to define the concrete syntax. The result (the linked concrete-, and abstract syntax definitions) are processed further to produce source code. The mapping between the domain model and the model items of the concrete syntax is also supported in GMF. The generated source code relies on the features of GEF and EMF. Although the concept of GMF is straightforward, it has some weaknesses: (i) the generation is not based on model transformation. Consequently the compilation steps are coded manually, and constraints on the compilation steps are hard to define. (ii) There is no built-in editor to change the appearance of the model items graphically. (iii) GMF is a new project, thus not all features are documented enough to use the framework.

Table 1 summarizes the comparison of the related work.

	GME	MetaEdit+	DiaGen	AToM^3	GEF+ EMF	GenGed	TIGER	GMF
Coding	Yes	No	No	No	Yes	No	No	No
Properties	Yes	Yes	Yes	Yes	No	No	No	No
Presentation DSL	No	No	No	No	No	Yes	Yes	Yes

Table 1. Related applications

3 The VMTS Presentation DSL

In VMTS, the *Presentation DSL* is named VMTS Presentation DSL (VPD). *Concrete Syntax Models* are created by instantiating VPD, they define concrete syntax for an arbitrary Domain Specific Model (for the *Subject Domain Model*). Thus, *Concrete Syntax Models* define how the model items of the *Subject Model* are visualized, and how they behave. The complete specification of the VMTS Presentation DSL can be found in [1]. Fig. 2 shows the metamodel – model, and the abstract syntax – concrete syntax relationships: (i) The Concrete Syntax Models are created by instantiating the Presentation DSL. (ii) The abstract syntax of the domain, namely the Subject Domain Model is created, and the concrete syntax is bound to the abstract syntax. From the Concrete Syntax Definition, a plugin is generated that is used to display the models of the target domain. (iii) Subject Models are created by instantiating the Subject Domain Model. (iv) The framework displays the model using the generated plugin.

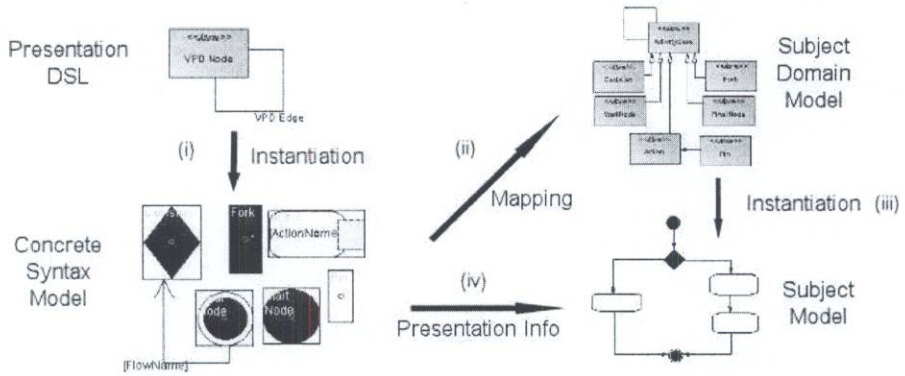

Fig 2. Abstract and concrete syntax definitons

3.1 Mapping

Modeling environments often distinguish between different kinds of fundamental modeling types, for example, model nodes and connections between the model nodes are handled differently. Handling different fundamental types is the basis of the mapping between the concrete and the abstract syntax definition, because different fundamental types have different attributes. Thus each fundamental type has an appropriate model item in the *Presentation DSL*. In VMTS, there are three fundamental types: *Nodes* (e.g. *Activity*), *Edges* (relations between nodes, e.g. *ObjectFlow*), and *AssociationNodes* (e.g. *AssociationClass* in a class diagram). VMTS Presentation DSL contains a Node (*VPDNode*) and an Edge (*VPDEdge*) only (Fig. 2), support for *AssociationNode* is the subject of future work.

The second step of the mapping is to bind the model items of the *Subject Domain Model* and the syntax definitions available in the *Concrete Syntax Model*. The structure of the mapping information depends on the data structure used by the modeler application, but the placement of the mapping information is a general problem: it can be stored either (i) in the abstract syntax definition (in the *Subject Domain Model)*, or (ii) in the concrete syntax definition (in the *Concrete Syntax Models*), or (iii) in an external data source. Storing the mapping in external sources is used for example in GMF. This solution allows many-to-many relationships between the concrete syntax and the abstract syntax definitions, but it means a large synchronization overhead between the models and the external resource when binding the concrete and the abstract syntax. If the mapping information is stored in the *Subject Domain Model* then the metamodels can have one representation only. This is a problem, because many Domain-Specific Languages can have more then one representation, for example resource editors for mobile phones can have different visualization (Fig. 3.). To summarize, it is useful to store this mapping information in the *Concrete Syntax Models*, thus the information can be reached easily if it is required. This means also that no many-to-many relations are allowed between *Subject Domain Model*s and *Concrete Syntax Model*s, but this is not a common need. In VMTS Presentation DSL, the mapping is represented by a reference to the *Subject Domain Model*, and another reference to the specific metamodel element. The mapping information is stored in the *Concrete Syntax Model*.

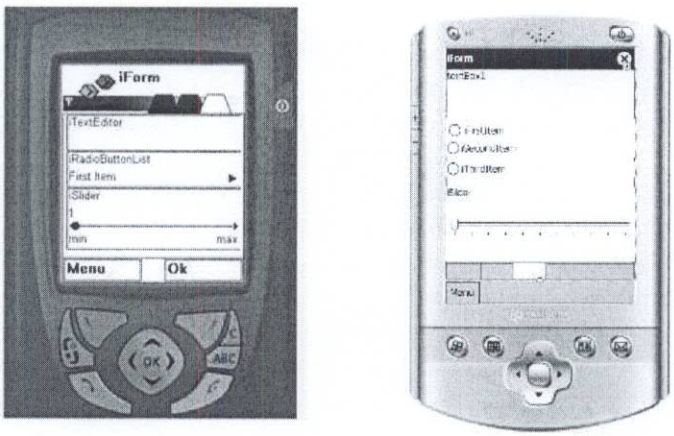

Fig 3. Different visualization for the same abstract syntax

3.2 Extending the behavior

There are many attributes that can make the syntax definitions more appropriate, and more customizable, but they usually rely on system-specific features:

(i) *Naming* the model items in the *Concrete Syntax Model,* and naming the *Concrete Syntax Model* itself can be useful to identify the model items. In plugin-based architectures similar to GMF, these names can be used as a base class name for the plugin items. In VPD, the naming is also supported: the name of the plugin item (e.g. *Decision*) is used as the base class name; the name of the model (e.g. *ActivityDiagram*) is transformed into the name of the project.

(ii) *Extending the fundamental types* is usually supported in modeling frameworks by specialized subtypes that are inherited from the fundamental types. For example, a rectangular shape that can handle docking is based on the common shape definition. This extension makes it possible to handle more specialized behavior in the DSL realizations in a simple

way. A typical example is the plugin-based architecture of Eclipse, where the plugin items can use either the base classes, or their specialized classes. In VPF, *Nodes* can be customized to *Shape* or to *DockShape.* VPD supports the specification of the sub-types for *VPDNodes.*

(iii) *Positioning constraints* are a common need in modeling when handling containment or attaching relationships. For example, *Input* pins should be aligned to one side of its container in Activity Diagram. Another example is specifying connection points on a rectangular shape to define where connected lines can be bound. Since these constraints specify the behavior of the model items, they are handled in concrete syntax definitions. In VPF, the connection points are managed by the framework, therefore VPD supports only aligning constraints.

(iv) *Resizing constraints* are similar to positioning constraints, but they specify how the model items react to resizing events. Typical example is the default size of the model item, or the maximum/minimum size of the item. This is useful to avoid too small or too large model elements. In VPD, the size constraints are fully supported.

(v) *System-specific attributes* are efficient, but they rely on the features of the modeling framework. For example there are models, for example object diagrams in UML, where the metamodel contains many model elements with the same appearance definitions, but different attribute values. This problem was described in detail in [2].

3.3 Visualization attributes

Since each Domain Specific Language has its own notation, the visualization is definitely the most often customized and most important part of the concrete syntax definitions. Building the visualization information using *primitive*s, i.e. simple geometric shapes, is a straightforward solution. In this way the framework has to support only handling the primitve shapes, and not the complex notations defined in the DSL. Complex figures can be divided into *primitive*s in general only if the *primitives* are flexible enough and can describe any possible shape, thus, the list of the supported primitives is important. Another important requirement is to be able to express location constraints for the *primitives* within a figure, for example the subtitle (text *primitive*) in a box (rectangle *primitive*) should be centered, or aligned to one side of the box.

In VPD, the visual representation is based on *Region*s. A *Region* is a graphical unit that is independent from the other *Region*s. A *Region* is responsible for visualizing a part of the model item, or the whole model item. *Region* definitions consist of *primitives* and *extended primitives. Primitives* are lines, Bézier splines, triangles, rectangles, rounded rectangles, diamonds, ellipses, crossed circles, and static texts. *Extended primitives* are dynamic texts to display the properties of the model items, and docking regions to define rectangles for the docked elements. Certain shapes contain only one *primitive*, others are more complex. For example, *Action*s are visualized using a rounded rectangle and a text label. Location constraints are handled by extending the *primitive* objects with alignment information. This alignment information shows how the appropriate *primitve* is positioned in the *Region.*

The relation between the model items in the *Concrete Syntax Model*s and the *Region*s is one-to-many, because model items can have any number of *Region*s, but a region is defined in exactly one model item. Since the *Region*s are independent from each other, they can be edited separately, and the model representation can be composed from the *Region*s when displaying the model item. For example, editor parts in an Eclipse-based plugin would contain typically only one *Region*, because the editor parts visualize the atomic information.

The presented solution is not system-specific, thus, *Region*-based visualization information can be used in any other modeling environment.

In VMTS, the solution is specialized as follows: *Node*s have only one *Region*, because they do not have independent visualization parts. Contrarily, the visualization of *Edges* uses five different *Regions:* two for the line ends, and three icons near the ends, or at the center of the line. This decision needs more explanation: (i) *Edge* does not require a *Region* for displaying the line, because only the line style, and the line width can be set in VPF. (ii) VPF supports setting the display style of the line ends, and the subtitles of the *Edges* independently (e.g. they can be enabled/disabled independently). The concept of *Region*-based visualization means that *Edges* require a *Region* for each independent part. (iii) Since the line ends, and the subtitles can rotate according to the orientation of the line points the associated *Region*s should also rotate. This rotation is handled in VMTS Presentation Framework, not in the code generated from a VMTS Presentation DSL model. To summarize, VMTS Presentation DSL handles the visualization information of the concrete syntax, using *Region*s that consist of graphical primitives. This solution ensures the required flexibility and uniformity between the different concrete syntax definitions.

4 Processing the concrete syntax definitions

The VMTS Presentation DSL is able to describe the concrete syntax of the model objects. Since the primary goal of this paper is to give an efficient solution to create concrete syntax definitions visually, a method is required to create and edit the *Concrete Syntax Models*. The main advantage of VPD is that it is a DSL, therefore *Concrete Syntax Models* can be created by instantiating the VPD, and using the common DSL editing environment. To make editing easier, a plugin, the VPD plugin was developed. This plugin provides a user-friendly environment to define *Concrete Syntax Models*, and to specify the concrete syntax using the facilities of VPF.

According to the VMTS Presentation DSL definition, VPD Plugin supports two fundamental types: *VPDNode* and *VPDEdge*. The key feature of VPD Plugin is the ability to define VMTS Custom Property Editors [2]. VPD Plugin defines a custom attribute editor - *Appearance Editor* - for editing the appearance of the shapes. To minimize the differences between the visualization of *Appearance Editor* and the final representation, VPD Plugin shows the model items in the modeler according to the defined concrete syntax.

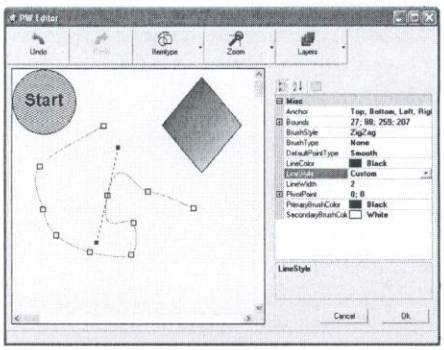

Fig 4. Appearance Editor

Appearance Editor (Fig. 4) is an image editing form with support for modeling the *primitives*. *Primitives* can be created using drag-and-drop actions, they can be resized and moved, or changed (for example points can be added to or removed from splines). Properties of the *primitives* can also be edited simply, since they appear in the property grid residing on

the right hand side of the form. *Appearance Editor* has additional facilities such as zooming, support for undo/redo operations, or layer management.

5 Conclusions

This work has described a technique to specify the concrete syntax of domain-specific modeling languages visually, accompanying to the abstract syntax rules provided by the DSL. The presentation information is specified as a mapping between the metamodel elements and their visual representation, and behavior data. This mapping is described by a domain-specific modeling language, namely, by the VMTS Presentation DSML. VMTS offers a tool support for VPD using metamodeling techniques. From the VPD models, a visual model processor creates a VMTS plugin. The plugin is launched by VMTS, offering the designed visual modeling environment for DSLs or UML. The result is a flexible method to specify the presentation of DSLs separately from the abstract syntax definition in an easy-to-use graphical way.

Future work includes the dynamic presentation of the attribute values. Currently, rectangular areas can be selected. Having an arbitrary shape, the user is offered a rectangular bounding box only. Thus, shape-specific bounding areas need to be developed.

Acknowledgements

The found of "Mobile Innovation Centre" has supported, in part, the activities described in this paper.

References

[1] VMTS Web Site, http://avalon.aut.bme.hu/~tihamer/research/vmts, 2006. 03. 05. 07:50

[2] Levendovszky, T., Mezei, G., Charaf, H.: *A Presentation Framework for Metamodeling Environments*, Workshop in Software Model Engineering 2005 (to appear)

[3] Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: *Control Flow Support for Model Transformation Frameworks: An Overview*, MicroCad, 2006

[4] Thuan Thai and Hoang Lam, *.NET Framework Essentials*, O'Reilly, 2003.

[5] GME Lédeczi, Á., Bakay, Á., Maróti, M., Vőlgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: *Composing Domain-Specific Design Environments*, IEEE Computer 34(11), November, 2001, pp. 44-51

[6] JKOGGE http://www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Projects/MetaCase, 2006. 03. 05. 07:51

[7] MetaEdit+ http://www.metacase.com/, 2006. 03. 05. 07:51

[8] Minas M: *Specifying Graph-like diagrams with DIAGEN*, Science of Computer Programming 44:157–180, 2002

[9] de Lara, J., Vangheluwe, H.: *AToM³ as a Meta-CaseEnvironment*, International Conference on Enterprise Information Systems, 2002

[10] The Eclipse Modeling Framework Framework http://www.eclipse.org/, 2006. 03. 05.

[11] GenGed tfs.cs.tu-berlin.de/~genged/, 2006. 03. 05. 07:52

[12] Erhig, K., Ermel, C., Hansgen, S., Taentzer, G.: *Generation of Visual Editors as Eclipse Plug-Ins*, http://www.tfs.cs.tu-berlin.de/~tigerprj/papers/, 2006. 03. 05. 07:52

[13] GMF http://www.eclipse.org/gmf/, 2006. 03. 05. 07:53

[14] UML http://www.uml.org/, 2006. 03. 05. 07:54