Presentation Framework – an Environment for Editing Metamodels

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
{gmezei, tihamer, hassan}@aut.bme.hu

Abstract: The growing complexity of the software systems made the model-based application development one of the most focused research fields. General purpose modeling languages, like UML are not always flexible enough to express domain specific features. Domain Specific Modeling Languages (DSMLs) are defined to model the special features, and the rules of these domains. One way to define DSMLS is metamodeling. Metamodeling techniques facilitate creating domain-specific modeling environment in an efficient and simple way. Although metamodeling techniques are capable of expressing the domain-specific constraint of visual languages, the presentation still cannot follow this flexibility. Editing frameworks are required that support customization with minimal programming effort. The Visual Modeling and Transformation System (VMTS) is a metamodeling environment that offers graphical metamodel editing features using the VMTS Presentation Framework (VPF). The goal of this paper is to present the metamodeling environment based on VPF with all of their metamodel specific features and compare the capabilities of VPF with other metamodeling environments.

Keywords: DSML, metamodeling, modeling framework

1 Introduction

Model-driven software eingineering became an essential approach in software development. Classic modeling tools have hard-wired model definitions, hence they cannot be used in domains that require customized model definition languages. Domain Specific Modeling Languages (DSML) constitute a way to create customized models for special domains where generic modeling languages would fail. Metamodeling is the way to achieve a more dynamic model definition and model handling. The proliferation of high level languages, object-oriented technologies and the proliferation of CASE tools made metamodeling even more important. Each model has its metamodel that defines the available elements (and their attributes) and the topology between elements (e.g. a ClassDiagram is the metamodel of an ObjectDiagram). Although metamodeling is flexible enough to fulfill the requirements of the domain specific modeling, it does not define the way of visualization of the elements (the concrete syntax). The abstract syntax (the metamodel definition) often fulfills the needs of the developers and researchers, but a wide variety of presentation is required for the end-user, who might not be familiar with the inner abstract graph representation of the tool. Solutions are needed which are able to handle models and their metamodels uniformly and provide a user-friendly, graphical way to edit the models.

Visual Modeling and Transformation System (VMTS) is an n-layer metamodeling environment [1]. The VMTS Presentation Framework (VPF) is the graphical environment part of VMTS used for displaying and editing the models with their proprietary representation. VPF has been successfully used in many different domains: feature models, UML 2.0 diagrams (e.g. class, object, statechart) as well as resource editor for Symbian mobile telephones. Although the framework was written in C# and it is based on .NET technologies, the solutions discussed here are reusable in every high-level programming environment.

There are several frameworks that have graphical editing support for the DSMLs in a more or less user-friendly way, but none of them is fully capable of expressing the domain-specific constraints. Although the visualization of DSMLs is not fully supported in these frameworks they use many notable solutions. Besides the introduction to metamodeling in VPF, this paper also introduces the features available in other metamodeling frameworks.

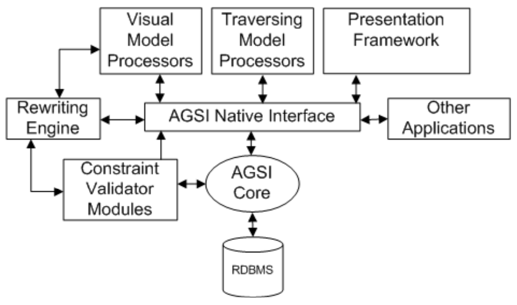


Figure 1
The structure of VMTS

2 Architecture

VMTS consists of several subsystems. The complete structure can be seen on Fig. 1. Attributed Graph Architecture Supporting Interface (AGSI) is responsible for the graph database actions and it offers a high-level interface for the other components. The Rule Editor and the Rewriting Engine are used for graph transformations; Traversing Processors are used for traversing the models in order to generate program code or other artifacts. Presentation Framework is the graphical environment part of the VMTS used for displaying and editing the models with their proprietary representation.

Presentation Framework consists of metamodel-variant, and metamodel-invariant parts. The metamodel-variant parts use plugins to offer individual visualization and editing features (Fig. 2). The plugin-based architecture guarantees the required flexibility and customization facilities. The framework offers a core functionality, and it can be extended by the model-specific plugins. A plugin is always attached to a metamodel whose models it supports (e.g. UML statechart metamodel for the statechart plugin). The connection between the plugin classes and the metamodel elements is represented by a unique metamodel ID provided by AGSI. On loading a model, if no plugin assigned to its metamodel was found, then a default plugin is used, which is referred to as Abstract Syntax Plugin (Fig. 2c). This general environment is also used for editing the first layer metamodels. The Abstract Syntax Plugin is a part of the framework, it supports neither special visualization methods, nor event-handling features, but it offers a feature to visualize and edit an arbitrary model in an abstract syntax view. VMTS offers several services and the commonly used features to make plugin writing simpler.

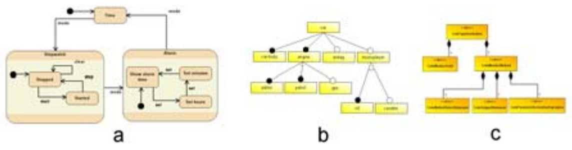
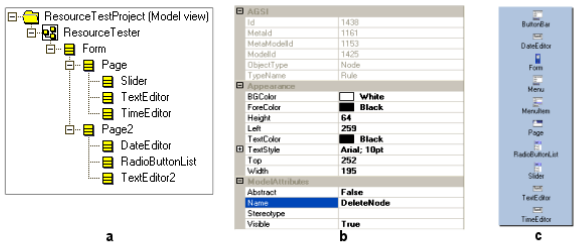


Figure 2

Plugin-based visualization

VPF supports three main types of the metamodel-invariant subsystems (i) the tree representation of the model and the visualization structure (Fig. 3a), (ii) the attribute panel, which displays the metamodel-defined data and the visualization information (Fig. 3b), and (iii) the toolbar, which contains the model elements that can be dropped on the canvases (Fig. 3c). These parts are discussed in detail in the next chapters.



```
ResourceTestProject (Model view)
  ResourceTester
    Form
      Page
        Slider
        TextEditor
        TimeEditor
      Page2
        DateEditor
        RadioButtonList
        TextEditor2
```

AGSI	
Id	1438
MetaId	1161
MetaModelId	1153
ModelId	1425
ObjectType	Node
TypeName	Rule
Appearance	
BGColor	⬜ White
ForeColor	⬛ Black
Height	64
Left	259
TextColor	⬛ Black
TextStyle	Arial; 10pt
Top	252
Width	195
ModelAttributes	
Abstract	False
Name	DeleteNode
Stereotype	
Visible	True

Figure 3

Model-invariant parts

3 Metamodel Relations

VMTS is not a common metamodeling tool based on the MOF specification. It uses is an n-layer metamodeling environment with full transparency between the layers, which means that each layer is handled by the same functions. Moreover, VMTS unifies the metamodeling techniques used by the common modeling tools, and uses model transformation applying graph rewriting as the underlying mechanism (Visual Model Processor). The fundamental data structure is a mathematical object: a labeled directed graph. The modeled objects are represented as nodes, and the connection between them are treated as edges.

The primary reference for metamodel-based tools and representations is the OMG's MOF specification [2]. Metamodeling techniques facilitate the reuse of the metamodels thus the reuse of the domain described by the metamodel.

The available model elements are defined in the metamodel. The toolbar (Fig. 3c) enlists the non-abstract model elements from the metamodel. The user can drag an element from this list, and drop it onto a canvas. The underlying technique is the popular *Prototype* design pattern [3]. Although this representation is appropriate in most cases, it is not elegant e.g. when the UML object diagram is displayed. In this case the metamodel is a UML class diagram possibly with many classes and associations. The traditional approach is to display an Object and a Link shapes, and the user selects the desired type after dropping them onto the canvas. Generality is not restricted, since the toolbar should work for models with arbitrary metamodels. We defined a *MetaMetaBased* mode for the toolbar, which means that the control displays one element for the model elements having the same meta-metamodel element. This is exactly the behavior we wanted for the object diagrams, since only one item is displayed for all objects (since their meta-metamodel element is metaclass), and the associations are treated similarly. If an object or a link is dropped on the canvas, a pop-up list appears, and offers the available model elements in the metamodel. Moreover, this toolbar is capable of obtaining the icons from plugins with general reflection mechanisms.

Considering the instantiation, there are two issues: the topological rules (e.g. the instantiation of the association in the UML class diagram as links in the UML object diagram), and attribute-related instantiation rules (e.g. the UML Class attributes can have values in the object diagram).

Topological rules mainly define rules for relations. Relations between the modeling elements are represented with edges between the nodes. There are three main type of relation in VMTS: *System Inheritance*, *System Containment* and *Association (System Relationship)*. *System Inheritance* expresses inheritance relation between the model elements, the properties are concatenated. *System Containment* expresses containment relation like between motherboard and ICs. *Association* expresses a connection between the elements that they can use to communicate. These three fundamental relation types are built-in, hard-wired relationships to provide basic functionalities. For the simplified treatment of the edges on different layers there are two another relation types in VMTS. *SystemMetaInheritance* expresses that the edge will be a *SystemInheritance* edge on the next level, *SystemMetaContainment* express that edge will be a System Containment edge. The edges on every level should use one of these five types as the type of relation.

The attributes of the model elements are presented in AGSI as an XML document (the *Property XML*). This XML document describes every important information about the model element (in metamodel level), and this is used as the base of validation on the instance level. VMTS offers an editing tool for this XML document using a *PropertyGrid*. The validation is made with an XSD (*Schema XSD*), that is generated from the *Property XML* stored in the metamodel, using an XSL script (Fig. 4).

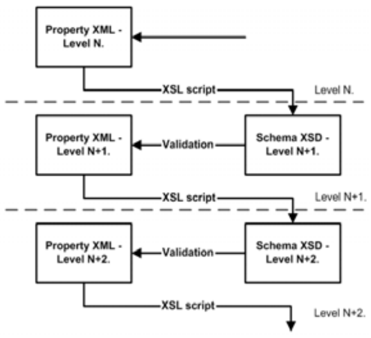


Figure 4

Attribute instantiation

In VMTS there are three elements to describe the structure of the attributes: *InstanceName, Attribute,* and *ComplexType*. The *InstanceName* tag contains the name of the instance level object (*Class* in the example). *Attribute* defines an attribute, it is defined as a *ComplexType,* thus, it is possible to group the related information. The tag *Name* contains the name, and *TypeExpression* describes the type of the attribute. The *TypeExpression* is either a primitive type (that can be processed with XSD), or the name of a *ComplexType*. Attributes tag holds the publicity information; *Multiplicity* contains the minimum and maximum number of the attribute. The short description of the attribute can be given in the *Description* tag. *Visible* tag is made for hidden elements. The *Editor* tag is perhaps the most complex. It should be supplied only if the attribute editor of .NET is not good enough, then it contains the name of the custom editor class. *RefreshAction* stores the default actions (like *Draw*) that should run after the value of the attribute changed. *ReadOnly*, and *Value* tags are used for storing the read only information, and the value provided in advance. An *Attribute* can have more *RefreshAttriutes, Description* or *Attributes* tag.

The *ComplexType* definitions contain the structure of non-primitive attribute types. These complex types make sure that the base types of XSD can modified to the particular task flexible. The *Name* tag contains the name of the *ComplexType*. This name should be given in the *TypeExpression* tag of the related *Attribute*. After giving the name, we can define the Attributes (the parts) of the complex type. The *ComplexType* definitions can be nested. *IsEnum* can be used to define *enum* types. This usage of the complex type is displayed as a dropdown list.

Recall that the attributes are displayed in the property grid (Fig. 3b) that offers a standard, user-friendly way to edit the values. Besides the default presentation, VPF has to ability to define VMTS Custom Property Editors for editing the values of the properties visually. The main goal of this feature is to handle special attributes, such as file saving, or code editing. These custom editors can be used for example to offer dropdown list and to force the users to select one of the listed elements. VPF has several built-in editors for the attribute types used most often.

4 Organization

4.1 Modeling Aspects

The most crucial part of a modeling framework is the organization of the models. Requirements such as supporting model aspects and multiple views of the same elements need flexibility in the diagram structure (e.g. a class diagram must be divided into namespaces and packages). In VPF, *canvases* can be defined for the models, which display a part of the system, typically, an aspect of the model. Each model can have multiple canvases. Since a canvas can be considered as an aspect or a view of the model, it is a natural requirement that a model element can appear on more than one canvas, possibly with different visualization properties such as line color. The framework supports multiple presentation and customized visualization of the model elements. Furthermore, it synchronizes the same elements on different canvases. The synchronization is necessary, since the simultaneous presentation of different canvases can require an automatic update of the common properties on every canvas. For example, if the name of the class changes in a class diagram on the canvas representing the business aspect, then the name should be updated automatically in the other aspects (e.g. in the resource aspect).

To display the model elements, including canvases, VPF uses the Model-View-Controller design pattern. Applying this approach, the design of the notification mechanisms is rather straightforward. The *Model* stores all the canvas-independent data, the *Controller* is responsible for the event handling, and the *View* for the visualization. The canvas-dependent (visualization) information is stored in the *View*. For each model element there is only one *Model*, but a *Controller*, and a *View* is created for each *Canvas* that contains the model element. As it is usually the case with MVC pattern, the classes have reference to each other, for instance, the *Controller* can easily obtain a reference to the *View* element that belongs to the same *Canvas*.

VPF offers not only extensibility and interfaces through the MVC architecture, but also built-in features and services that are used frequently in the visualization of a model element (e.g. resizing, relocation or docking). To offer these services, the framework contains generic base classes (*BaseModel, BaseView, BaseController*) for the MVC architecture. On developing a plugin, one needs to derive a class form each of these classes (e.g. *NodeModel, NodeView*, and *NodeController* for handling nodes in a graph plugin), and define the customized behavior of their model-elements only (e.g. appearance), because the general functionalities are implemented in the base classes. To provide more common features, the shape and line classes are separated according to the type (node or edge) of the model element. The previously mentioned *Node* classes should be inherited from the *Shape* classes (e.g. *NodeModel* from *ShapeModel*). The following features are implemented in these classes: creation, automatic saving and loading; containment with drag and drop support, event handling, drawing, and helper properties for accessing the visualization information (e.g. the line style).

4.2 Hierarchy

There are several model types that require containment with drag and drop support between the elements, such as statechart diagrams, hence the framework should support the containment of model elements. The containment is handled only between nodes; the edges cannot contain other model-elements. Containment relations are defined in the metamodel by *SystemContainment* edges. The elements in the same container are ordered, similarly to the windows of the applications in a graphical window system. Obviously, the order can be changed by the users during editing the diagrams. In the containment hierarchy, each model element must have exactly one container. This uniqueness is necessary, because the further model-processing algorithms (e.g. traversing model processors, code generators) are much simpler, and faster if this condition succeeds. Therefore, VPF defines a containment chain between the *Model* objects. There are two variables in each model object that supply the connection between the elements of the hierarchy. *Children* is a list with the contained model elements, and *Container* holds a reference to the parent element. These two properties facilitate the simple navigation in the containment hierarchy. The list of the children is an ordered list; the position in the list is used for the hierarchical and visual ordering.

Although the containment hierarchy defined by the metamodel is more or less straightforward, this issue needs additional mechanism, when several views and change of the order are considered. Since the model elements on the canvas depend only on the user, i.e. which elements are placed on a canvas by using the drag and drop functions. Fig. 5 shows an example. Fig. 5a represents the simplified steps of a phone call. Major steps are focused, minor steps are hidden. Fig. 5b shows the details of the *Dialing* state (darker color). In the first case the container of the *Dialing* state is the *Active* state, but in second case the canvas is the container. Moreover, these hierarchies are *compatible* with the hierarchy defined in the metamodel, but not necessary *identical*.

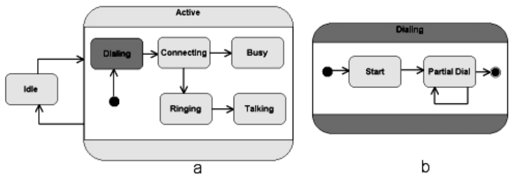


Figure 5
Different hierarchies

On the one hand the concept of the unique container should be maintained. On the other hand a model element can have different containers on different canvases. The hierarchy defined by the metamodel is followed by the hierarchy of the *Model* components. However, the actual order on a canvas must be recorded in either of the canvas-dependent components. Since this hierarchy is closely related to the event handling mechanism, which is the responsibility of the *Controller*, a *Controller-based* hierarchy is established along with the *Model-based* hierarchy. With this solution, the real topology and the visualization have successfully been separated, only one problem remains to be solved, namely, how the different hierarchies affect each other. If the user is about to change the *Controller-based* hierarchy, such that it is not compatible with the model hierarchy any more, then the following steps are performed. (i) The *Model-based* hierarchy is checked whether it can be made compatible with the new structure (the model element was placed in a container which can be its parent according to the metamodel). (ii) If the metamodel permits, then the framework checks whether the *Controller* can affect the *Model-based* hierarchy. The validation is based on a property of the *Controller*. (iii) The drag and drop operation is completed. If the *Controller* is a reference controller, then the *Controller*- and *Model-based* hierarchy-, otherwise only the *Controller-based* hierarchy is changed. (iv) Finally, other canvases are notified.

VPF visualizes the different hierarchies in two tree controls (Fig. 3a). The *Model-based* tree control provides a precise representation of the model repository. The other tree control is called *Visualization Tree View*. It presents the *Controller-based* hierarchies, and contains all visualization data. The tree controls are always synchronized with the canvases.

5 Event-handling and Persistance

5.1 Event-handling

VPF handles the events using the *Chain of Responsibility* design pattern along the *Controller-based* hierarchy. The canvas (that is the top of the *Controller-based* container chain) always encompasses a form class from the programming environment that is the actual entry point of the events. If a control contains elements, it forwards the event to its children with the depth-first-search algorithm. If a container and the contained item both can handle the event, then it is handled by the contained item.

The behavioral commonalities (between the class and the event-handling structure) allow VPF to provide implementation for the behavior of the model elements. The *Controller* objects in the framework have an attribute *State* that can change according to the user actions. The event-handling and the visualization mechanisms are based on this attribute (e.g. the selected items should display a selection border). VPF supports events for common drag and drop, mouse, and drawing events and for containment-, and attribute changing. The notification strategy is based on delegates (method references). If an event occurs, each subscribed object receives a message and can react to the event.

VPF realizes the undo and redo operations also by combining the *Memento* and *Command* design patterns [3]. The common basis for the commands makes it possible to create two stacks for them: an *UndoStack* and a *RedoStack*. If the undo action is processed, the last action is popped from the *UndoStack* and pushed into the *RedoStack*.

5.2 Persistance

VPF uses the *Model-based* containment hierarchy for saving the model elements. First the model itself is saved, then the canvases. The model elements are saved using a depth-first search. The *Model*-based containment information is stored in the graph representation. The *Controller*-based data is stored as visualization information. We defined an attribute called *ZOrder* for storing the ordering information. VMTS separates the visualization information from attributes defined by the metamodels, but stores both of them in the labels of the model graph.

The loading algorithm also uses the containment chain. First, the model is loaded then the canvases. Loading the model elements is performed in several steps. (i) Nodes are loaded (using depth-first-search), (ii) the *Model*-based containment is restored for nodes. (iii) Edges are loaded, and (iv) the *Model-based* containment is restored for edges. (v) Finally, *Controller-based* containment hierarchies are restored for each model-element: both edges and nodes.

6 Related Work

The Generic Modeling Environment (GME) [4] is a highly configurable metamodeling tool supporting two layers: a metamodel-, and a modeling layer. GME uses a plugin-based architecture. Plugins can be defined using a COM interface. In GME the basis of the modeling is the *modeling paradigm*. Model paradigms act as the metamodel for the particular domain specific language. GME is a graphical metamodeling environment that supports the basic requirements for editing metamodels. Moreover, it can be used only for modeling with the MOF-based metamodeling hierarchy (it does not support n-layer metamodeling, like VMTS does). GME is the metamodeling tool from which VMTS has borrowed its base concepts.

Eclipse [5] is possibly the most popular, highly flexible, open source modeling platform that supports metamodeling. Eclipse is based on plugins, that grants the required flexibility. Eclipse Modeling Framework (EMF) can generate source code from models defined using the Class Diagram definition of UML. EMF definitions contain the abstract syntax (the metamodel) only, the concrete syntax (the visualization) cannot be defined this way. The generated code contains base classes for editing the models, but the appearance is not customized. Graphical Editing Framework (GEF) is a part of the Eclipse project that provides methods for creating visual editors. EMF does not support code generation for GEF. Graphical Modeling Framework (GMF) is a new Eclipse project that is under validation. The goal of GMF is to form a generative bridge between EMF and GEF, whereby a diagram definition will be linked to a domain model as input to the generation of a visual editor.

The GenGed (Generation of Graphical Environments for Design) [6] environment is suitable for creating visual language definitions. It is rather presentation oriented: instead of metamodeling, 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. GenGed uses AGG [6] as the internal graph transformation engine. For the editing features, a graphical editor is also generated to support the newly created visual languages. Transformation-Based Generation of Modeling Environments (TIGER) [7] is the successor of GenGed. It uses precise visual language (VL) definitions and offers a graphical environment based on GEF.

JKOGGE [8] is a tool for generating CASE tools. The tools built with JKOGGE consist of three parts: a base system, components, and documents. Documents are represented as distributed graphs. Components are realized with plugins that perform a well-defined task, e.g. editors.

Another framework is the Diagram Editor Generator (DiaGen) [9] that uses its own specification language for defining diagrams. The specification is checked and structurally analyzed using hypergraph transformations and grammars.

MetaEdit+ [10] offers a tool suite for defining a domain-specific modeling language with CASE support. The tool offers a full CASE support for the defined language, allowing developers to model using concepts that represent the product domain. MetaEdit+ allows viewing the design data in diagrams, tables and matrices. It offers an API for accessing components and enhances debugging.

Conclusions

The main ideas and design decisions of the VPF were presented in this paper. The metamodel-based static structure of the toolbar, tree view and attribute panel facilitates the general operation for an arbitrary model without further customization. The plugin architecture enables the metamodel-based customization of the models. The framework offers base classes that cover the most common tasks. A set of model elements can have many views; the different views of the same model elements are synchronized automatically. Two types of containment hierarchy are handled to support both the metamodel-defined, and the visualization containments. The hierarchies are synchronized to avoid inconsistency. The *Model-based* hierarchy is used in serialization, while the *Controller-based* hierarchy is the basis for the event handling mechanisms. The framework provides functionalities for internal state handling and event refinement. More detailed information on VPF can be found in [1].

Future work includes devaloping a domain specific modeling language, which expresses the information required by a plugin, along with the necessary generator.

References

[1] Visual Modeling and Transformation System http://avalon.aut.bme.hu/~tihamer/research/vmts

[2] Meta- Object Facility (MOF™), version 1.4, http://www.omg.org

[3] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series

[4] 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

[5] Graphical Editing Framework, http://www.eclipse.org/gef/

[6] Taentzer G: AGG: A Graph Transformation Environment for Modeling and Validation of Software, In J. Pfaltz, M. Nagl, and B. Boehlen (eds.), *Application of Graph Transformations with Industrial Relevance (AGTIVE'03), volume 3062,* Springer LNCS, 200

[7] 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/

[8] JKOGGE http://www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Projects/MetaCase

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

[10] MetaEdit+, http://www.metacase.com/