Universal Domain-Specific Code Generator

László Siroki, Gergely Mezei, Tihamér Levendovszky, Tamás Mészáros

Budapest University of Technology and Economics
{lsiroki, gmezei, tihamer, meszaros}@aut.bme.hu

Abstract: Nowadays, supporting domain-specific modeling is essential in software development. We have created several domain-specific modeler applications in the last few years. By summarizing our experiences and knowledge, we are now building a highly flexible and efficient solution, a new version of our modeling environment, Visual Modeling and Transformation System. Flexibility is reached by modular design, while efficiency is gained mainly by using strongly typed model elements and compiled, domain-specific classes. One of the key components of the new system is the code generator that can produce these specialized classes by processing domain definitions. In this paper, we present the main solutions of our modeling system and we introduce the generator component in detail.

Keywords: modeling system, metamodeling, domain-specific, source code generation

1 Introduction

Domain-specific visual languages play an essential role in software engineering, especially in the field of model-driven development. By illustrating the problems in a graphical way, these languages permit to raise the level of abstraction and help to define the steps of the software lifecycle. The increasing popularity of domain-specific languages requires applications that are capable of visualizing these languages and offer a user-friendly way to edit the models interactively.

The Visual Modeling and Transformation System (VMTS) [1] is a transparent N-layer modeling and metamodeling framework developed by our research team. Three versions of the VMTS framework has been implemented previously, we are now developing the fourth version, using our experiences from the previous works, the features and solutions learned from other frameworks and considering some missing features of existing tools. The main requirements for the design of the new VMTS framework are as follows: (i) The framework must support different business domains, where the model elements, their appearance and editor functions can be different (diversity). (ii) The framework must support multiple storage types, these storage formats should be extendable. (iii) There is a need to process the models with different algorithms, so the representation of the models must provide an API for reading and modifying the models. (iv) The performance of the framework is essential, which is conflicting with the need for advanced features. To resolve this problem, our new framework supports generating different representations of the same model with different features for different purposes.

The paper is organized as follows: Section 2 presents the background of our research; it elaborates similar modeling tools and their features. Section 3 introduces the new architecture of the VMTS. Section 4 explains the role of the code generator as well. The most significant aspects and the details of the implementation of the generator are described in the Section 5. In order to show how the approach works, Section 5 also includes a few simplified examples.

2 Related Work

Eclipse [2] is a highly generic modeling environment. Eclipse Graphical Editing Framework (GEF) [3] is an open source infrastructure for creating and using graphical editors based on Eclipse. The underlying architecture of GEF is the Model-View-Controller [4] pattern. The manageable data (model), the visualization (view) and the interaction features (controller) are separated into different classes. The model classes can be arbitrary Java classes (POJO), however, the controller classes should derive from the common *EditParts* class. The visualization can be performed by an arbitrary Java class by implementing a predefined interface (*IFigure*). GEF does not support automatic change-notification services, due to the underlying arbitrary model object. Notification mechanisms should be performed through manually written listener and adapter objects.

Graphical Modeling Framework [5] (GMF) is another Eclipse project, GMF utilizes GEF. Compared to GEF, GMF uses Eclipse Modeling Framework [6] (EMF) models as the underlying model object. EMF facilitates the serialization of models in various formats and the change notification mechanism is also a built-in feature. The editor environments in GMF are generated. The code generation is based on four models: (i) the EMF model (domain model) which serves as the model of the visual language; (ii) the Graphical Definition Model that defines the visualization of each model element; (iii) the Tooling Definition Model, which describes the additional visual elements required for the user interface (including editor palettes, menus) and (iv) the Mapping Definition Model, which realizes the mapping between the domain model and the visual models. The creation of models (ii)-(iv) and the generation of the editor from the input models is supported by the GMF Dashboard wizard.

The TIGER [7] (transformation-based generation of environments) tool generates visual editor plugins for Eclipse from typed grammar-based visual language specifications. TIGER is based on EMF to store models in the memory and to perform serialization and uses GEF to visualize models.

DiaMeta [8] is a framework for generating graphical diagram editors. It uses hypergraph grammars to specify visual languages. Editors generated with DiaMeta are capable of running online structural and syntactic analysis on the edited models. DiaMeta also employs EMF to define visual languages. Thus, it provides a similar approach to the one presented in connection with TIGER: change notification events are generated by the underlying EMF model on attribute changes, and the generated view and controller objects update the visualization as response to these events. In contrast with TIGER, DiaMeta uses custom classes for model visualization instead of utilizing GEF.

Generic Modeling Environment (GME) [9] is a general purpose metamodeling and program synthesis environment. The modeling concepts of GME are represented by the MultiGraph Architecture (MGA) object network. MGA (and also GME) is built on the COM [10] architecture. MGA objects provide change notification, transaction handling and support of multi-client collaboration on the same model. Model elements can be persisted through a unified storage interface to various storage types including relational database and XML formats. MGA provides general classes to represent model elements the properties of which can be reached through the same interface. However, it is also possible to edit model properties through a typed interface called Builder Object Network (BON), which is generated for a specific domain. BON also builds on core MGA objects and only wraps their features. Compared to GME, our framework does not use wrappers to support writing model processors. Our framework generates the typed interfaces and objects for each domain.

A common drawback of all the tools above is that each of them uses the same class and object hierarchy to perform various operations on them (including visualization, model transformation, custom model processing) and do not support to use optimized solutions for different purposes. In contrast, our framework provides optimized data-representation for different kinds of applications.

Table 1

Comparison of existing tools

	GEF	GMF	TIGER	DiaMeta	GME
Diversity	+	+	+	+	-
Event notification	-	+	+	+	+
Multiple storages	-	+	+	-	+
Model processor	-	-	+	-	+
Performance tuning	-	-	-	-	-

3 The New VMTS Architecture

In order to fulfill the requirements mentioned in the introduction, we have designed a component-based architecture, where the well-defined interfaces allow replacing the components seamlessly.

Visual Modeling and Transformation System (VMTS) is a graph-based metamodeling system. Metamodeling means that we can create models not only for predefined modeling languages, but we can create new modeling languages as well. New languages are defined by creating models of models, called metamodels. VMTS uses n-layer, layer transparent metamodeling [11]. This means that we can also create the metamodels of metamodels, since the base functions are the same for all modeling layers.

In VMTS, models consist of nodes and edges. An edge connects two nodes. On the one hand, our model elements (model, node, or edge) implement interfaces independently from the current domain, but on the other hand, they are customized for the corresponding domain. The common interfaces are thin, they contain basic functionalities only, such as name retrieving. Attributes and references to another typed model elements are handled in the domain-specific part. This means that we use mainly typed model elements in order to maximize performance and minimize memory consumption.

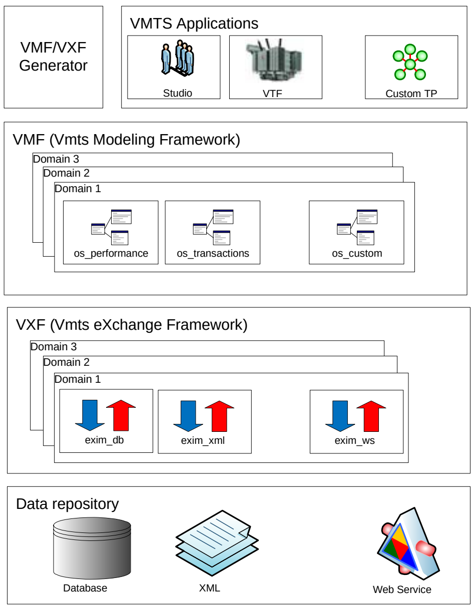

Figure 1
The architecture of VMTS

The architecture of our system is depicted on Fig. 1. The bottom most level is the *Data Repository* that represents the different data persisting stores that can be connected to the system. Above the repository layer, the *VMTS Exchange Framework* (VXF) and *VMTS Modeling Framework* (VMF) can be found. VXF and VMF form together the *VMTS Data Interface* (VDI). These two layers (VXF and VMF) are used to store the data in the repositories and to store the model data in memory during editing. At the top level of the architecture, different components (*VMTS Applications*) can be found; these components use the VDI to manipulate the data.

As depicted in Fig. 1, multiple domains are shown in illustration. Different VMF and VXF components belong to each domain. In VMF, models are stored in an object-oriented way as domain-specific class instances. Every domain has a set of interfaces for the model, the nodes, the edges and the attributes of the domain. There are also implementation classes that implement these interfaces, but there can be multiple sets of them. The set of data classes that represent a domain is called the Object Space (OS) of the domain. A domain can have multiple OSs, which share the common interface, but can have different features because of different implementations. We may need an OS that contain types implementing built-in change notifications to simplify visualization and editing of the elements in the editor. However, in case of model processors, there may be no need for this overhead since we do not always visualize the model items. Moreover, in case of model processors, we may require transactions in order to roll back changes, when a part of the transformation fails. Basically, we need different OSs to be able to fine tune the system.

The information stored in instances of the data types of the VMF components can be persisted in different data stores called repositories by the VXF components. A VXF component is created to each domain, as in the case of VMF components. The different classes in VXF are called Export Import Classes (EXIMs). In each domain, an EXIM is created for each available repository type. The EXIM class is capable of saving and loading the data of the objects of the VMF to and from the associated repository. Some examples of the repository types that we support are: (i) database-based repository (for compatibility with older version of VMTS and parallel model editing), (ii) XML-based repository to store model data in platform-independent files, (iii) GXL, (iv) XMI. The EXIM classes read and write data by using the common VMF interfaces, therefore, EXIM classes are OS independent and it is easy to persist the data to multiple repositories at one time.

The presented system needs additional *VMTS applications* to visualize and process the modeled elements and to make generation of components easier. The *Adaptive Modeler Studio* is a user interface, where the domains can be specified by means of metamodels. The *VMF/VXF Generator* is used to automatically generate the VMF and VXF components from domain specifications. This means that components do not need to be implemented by hand. This component is the key of domain-specific behavior. The *VMF/VXF Generator* is described in the following sections.

4 The Role of the VMF/VXF Generator

Recall that our system uses metamodeling techniques to define and handle domain-specific languages. This means that we do not have a predefined language definition, but we have to process the metamodel in order to obtain the modeling structure. There is only one exception: at the top of the modeling layers, we have the *SystemRootMeta* model that is the root metamodel of all other models and that is defined by itself. *SystemRootMeta* was defined by hand, it is read-only. However, for all of the other models, we have the Generator component, which processes the definition of the metamodel and create OS and EXIM classes for the instance models. In Fig. 2, the role of the Generator is illustrated.

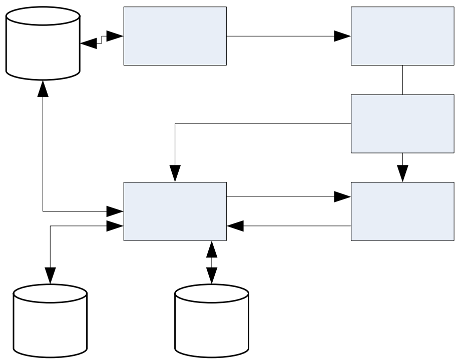

Figure 2
EXIM and OS generation process

Firstly, the Layer N EXIM loads the Layer N model to Object Space representation. Secondly, the Generator generates the Layer N+1 EXIM and OS for different repositories from the Layer N OS. Here, Layer N OS determines the common interface shared between all the different OSs.

The Generator creates the Object Space and EXIM interfaces and classes for each metamodel separately. For the *SystemRootMeta* model, these are implemented by hand, but for other models, they are generated by the VMF/VXF Generator.

This mechanism is layer-transparent, namely the models in the Layer N are the metamodels of the Layer N+1, thus the code is generated from the Layer N model and it is capable of storing model instances of the Layer N+1. This can be done for unlimited depth. For every node and edge in the model, attributes can be defined by a name and a type. These attributes become properties in the next layer. Moreover, every element can have stereotypes, which can modify their behavior.

From each model item in the metamodel, the following outputs are generated: (i) An OS-independent interface extending the basic, common interface. Recall that this basic interface contains atomic, system-wide used properties, such as the name, or stereotype of the item. (ii) Multiple OS implementations which implement the OS independent common interface, but have different features.

In case of nodes, the OS independent interface contains all attributes and all connecting edges as properties, except inheritance edges that are represented by inheritance relation between the corresponding interfaces. From each edge in the metamodel, similar interfaces are generated as for the nodes, because the edges can also have attributes.

Currently, OS implementations can have the following features: (i) change notification, (ii) transaction management and (iii) quick relation navigability. The change notification is important if a user interface is used to display the model – perhaps in many different views – while the model is changing because of editing or transformation. The transaction management is important for model transformations, where we should be able to undo or roll back a complex transformation step. Quick relation navigability means that relations are not only stored as edges with references to the endpoints, but as properties directly referencing the other side of the relation. This is useful for performance reasons if the attributes of the relations are not used or not present.

Typed EXIM classes are also generated by the VMF/VXF Generator. These classes support loading and saving of the model and the model elements from and to the specified repository.

The relation of the common interfaces, OSs and EXIMs are depicted on Fig. 3. The figure also shows that the common interfaces are used to resolve the N-by-M problem between the different OSs and EXIMs.

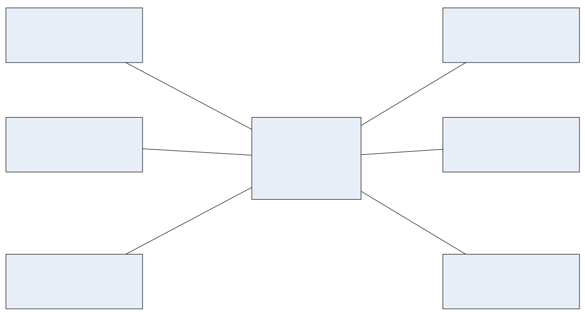

Figure 3
Relation between the common interfaces, OS-es and EXIMs

This way the Export-Import modules for different repositories can load and save the models independently from the actual Object Space implementations, because all of them share a common interface.

5 Implementing the VMF/VXF Generator

The Code Generator consists of two main sub-modules: the Language-independent Code Model Generator and the Language Plug-ins.

The language-independent code generator traverses all nodes and edges in the metamodel and creates a generic model of the program code to be generated. The features of the modern programming languages (interfaces, classes, class and interface inheritance, interface implementation, class fields, interface and class properties, simple types, collections, event notifications) are modeled in a language independent manner. Fig. 4 depicts the main steps of generation: from the metamodel, language-independent models are generated, which are used to generate source code in different languages, using the language plug-ins.

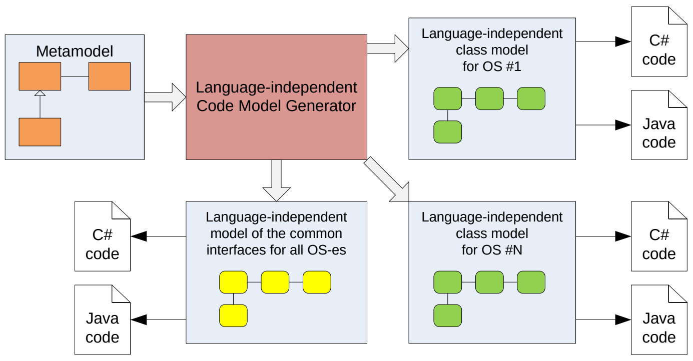
Metamodel
Language-independent Code Model Generator
Language-independent class model for OS #1
C# code
Java code
Language-independent model of the common interfaces for all OS-es
C# code
Java code
Language-independent class model for OS #N
C# code
Java code

Figure 4
Main steps of generating domain-specific source code

Language plugins are responsible for generating a compilable code in the desired target language by using the associated code model. For example, a simple string property in the model will be transformed into a property in C# (because the language supports properties), while it will become a getter-setter pair in Java (because it is a convention for JavaBeans). If change notification is required, the C# version will be generated using the so called *Dependency Properties*, which provide two-way data-binding functionality in WPF [12], while the Java version could use some third party data binding library. The language plug-ins also validate that the given code model is correct, e.g. the class names are valid identifiers in the destination language and there is no conflict between them.

The Code Generator creates a representation of the models the following way:

- Each node in the metamodel is represented by a set of interfaces and corresponding implementation classes per Object Space. Fig. 5 shows the generated interfaces and the classes for one OS (UI) for a sample node called ‘Book’, which has a complex property named BookData.

- Each attribute of a node is represented by a property in the attribute interface corresponding to the node (BookAttributes in Fig. 5) and in the implementation classes (BookAttributes_UI for the UI OS). Attributes with higher multiplicities are represented by generic collections. Complex attributes can be defined by defining so called complex types, which are represented by distinct classes (See CTBookData interface and CTBookData_UI class in Fig. 5).

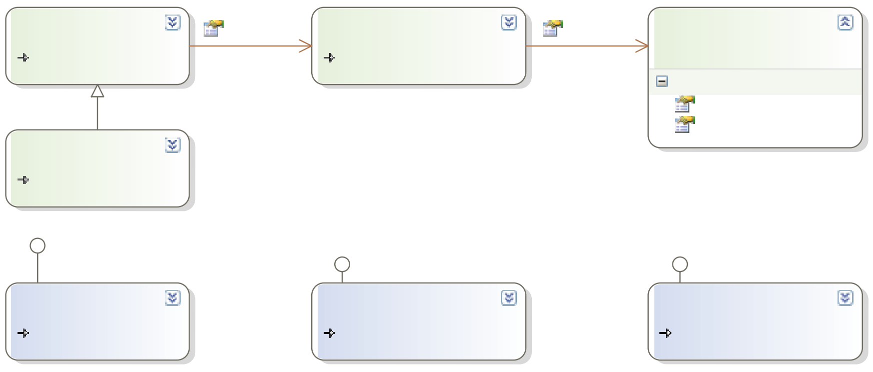

Figure 5
The interfaces and classes related to the Book node

- Inheritance edges are represented by interface inheritances. This can be seen in Fig. 6, where Novel inherits from Book.

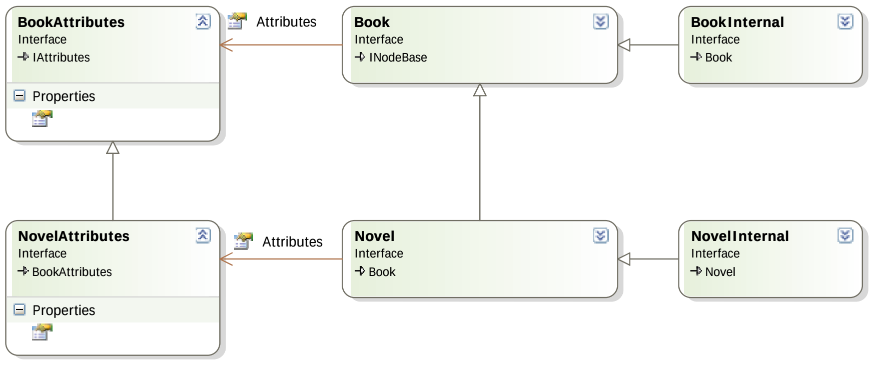

Figure 6
The interfaces related to the Book and Novel node, which are in inheritance relation

- Containment and association edges are represented by interfaces and implementation classes similar to the nodes and reference properties for the two endpoints of the edge. Depending on navigability, there are also references in the nodes for the connecting edges, with different getter methods for the relations connecting with the left ends and the right ends.

- Edge attributes are represented in the interfaces and implementation classes corresponding to the edge, similarly as for the node attributes.
- For convenience and performance reasons, the nodes on the other end of the relations are reachable directly through another property. The property for the collection of edge objects of the same type is named after the relation's name, whereas the property for the collection of the neighboring nodes is named after the other side's role name. Note that edges in the metamodel have InstanceName, LeftRole and RightRole properties.

For large models, it would not be feasible to load the whole model into the memory at once, so in some of the implementations, the related nodes or edges are loaded lazily when the program tries to access the property which is used for getting the connecting edges or neighbouring nodes. The loaded results are then stored in a field, and later no reload is necessary. The properties of the data storage classes increase the usability, but do not increase the memory footprint of the objects, because properties are class methods in practice.

After generating the source codes, they are compiled into DLLs by using CodeDOM in the case of the .NET platform. These compiled DLLs can be loaded at run-time without restarting the program. This feature makes possible that after creating a metamodel, a model conforming to this newly created metamodel can also be created. In this case the Generator creates the necessary VMF and VXF interfaces and classes and compiles them. Generated code that is not .NET-specific, can be used by other modeling tools.

CodeDOM could be used also for source code generation, but it has some bottlenecks, e.g. it can only generate source for a few of the .NET platform languages, and does not support every programming construct which we would like to use.

Conclusions

Domain-specific visual languages became essential in software engineering. Domain-specific modeler tools capable of displaying, editing and transforming visual models are required in many cases. There exist several applications for this purpose, but the requirements for these tools are increasing. We have implemented three versions of our modeling and transformation tool, VMTS before. Based on our experiences we found four main requirements that a domain-specific framework should fulfill. (i) The modeling framework must support the diversity of the domains. In VMTS, models are represented by attributed graphs. Metamodels can be defined to determine what kind of model elements can be used in the domain model. (ii) There is a need for various input/output formats. Most existing tools use their own storage types, or some domain-specific standard file format that are hard coded and cannot be changed. The new VMTS supports multiple model repository types and can be extended by new ones easily. This is accomplished by using storage-specific export-import classes, which are generated automatically for each domain separately for performance reasons. (iii) The framework should provide a way to change, or evaluate models. VMTS provides a model transformation engine and simplifies to create custom model traversers based on the code generator. (iv) The framework should use the computing resources efficiently to provide good performance even when dealing with large models. Also the models should be visualizable and visually editable, thus, having a notification mechanism for model changes is highly desirable. Moreover, for some algorithms transaction support is required. VMTS addresses these conflicting requirements in such a way that the Generator component is capable of generating different implementations for the data structure storing the model. These implementations share a common interface but can have different features.

The code generator component is an essential part of the new VMTS. The Generator provides the functionality to create strongly typed representations of the models and strongly typed utility classes to load and save the models using different model repositories and formats. The component is also capable of generating different implementations for the same domain. Furthermore, it can produce the code in multiple languages, using language plugins. Therefore, the code generator has two roles: On one hand it supports the VMTS by generating the data structure for new models in C#. On the other hand it can generate the data structure implementation in other languages for use in other applications.

We have elaborated the architecture of the generator and created implementation for the ‘UI’ Object Space – which can be used for visual editing purposes – and for the ‘XML’ EXIM – which stores the models in our XML format. We are working on the new GUI, the Adaptive Modeler Studio and on other OS and EXIM implementations. The code generator can be improved further to provide more easier and general way for defining the transformation from the metamodel to the representing source code. We are also working on a solution which allows defining the metamodel of the language-independent code model and uses a textual specification to transform the code model into source code.

Acknowledgement

The fund of ”Mobile Innovation Centre” has partly supported the activities described in this paper.

References

[1] VMTS homepage: http://vmts.aut.bme.hu

[2] Eclipse homepage: http://www.eclipse.org

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

[4] E. Gamma et al.: Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley Professional Computing Series)

[5] Graphical Modeling Framework homepage: http://www.eclipse.org/gmf

[6] Eclipse Modeling Framework homepage: http://www.eclipse.org/emf

[7] Erhig, K. et al.: “Generation of Visual Editors as Eclipse Plug-Ins”, http://www.tfs.cs.tu-berlin.de/~tigerprj, last visited on 2008. 08. 12.

[8] Minas, M.: Generating Meta-Model-based Freehand Editors, Electronic Communications of the EASST, Proc. of 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), 2006

[9] Lédeczi, Á. et al.: Composing Domain-Specific Design Environments, IEEE Computer 34 (11), November, 2001, pp. 44-51

[10] Component Object Model homepage: http://www.microsoft.com/COM

[11] G. Mezei: Transformation-based support for visual languages, Ph.D. Thesis, http://vmts.aut.bme.hu/GMezei_Thesis.pdf

[12] Nathan, A.: Windows Presentation Foundation, Pearson Education, 2007