Architecture of an In-Memory Transformation Engine

Tamás Vajk, Gergely Mezei, Hassan Charaf

{tamas.vajk, gmezei, hassan}@aut.bme.hu

Abstract: General purpose modeling languages, such as UML, had a great impact on reliable software engineering. After realizing the need for automated code generation from models, the more appropriately customizable Domain-Specific Languages emerged. The creation of these languages requires metamodel-based environments, in which new languages can be designed in a visual way with minimal amount of coding. Translations between different domain-specific models can be performed automatically by model transformation systems if the necessary conversion steps are defined. Usually, model transformation systems store their models in memory, however in this way the model distribution is hardly possible. The performance of database-based modeling environments is considerably lower than those of in-memory versions. This paper introduces the steps of converting a database-based modeling system into a modeling environment that is able to work in both database and memory without having to duplicate the previously implemented algorithms.

Keywords: metamodeling, model transformation, VMTS

1 Introduction

Modeling languages, such as the Unified Modeling Language (UML) [1], are among the most frequently used abstraction methods in reliable software engineering. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system. It is important to notice that UML stands for the different types of diagrams and for the model itself. UML diagrams are used to visualize a model; from the software development point of view, UML models are much more important than their representations, because the model can grant the preferred level of abstraction in defining the application details. By forcing the developer to design an appropriate model for the application, the overall quality of the implemented system is improved. Although the creation of UML-based modeling systems had a great impact on software development, UML has a few weaknesses, which are caused by having an imprecisely specified abstract syntax and being a too general-purpose modeling language, which aims to model all possible domains. These problems make source code generation hardly possible. In general, this means that there are cases when UML-based modeling tools are not flexible and not powerful enough to express all the constraints which arise in a special domain. In these cases, more flexible languages are needed to handle the specialties of the selected domain.

Using Domain-Specific (Modeling) Languages (DSMLs or DSLs) [2] is a widely adopted way to overcome these problems. DSLs tend to support a higher-level of abstraction, than general-purpose modeling languages, therefore, they require less effort and fewer low-level details to specify a given system. Since DSLs contains only domain-specific model elements, they allow precise code generation for the given field. Being free from the manual creation and maintenance of source code means that DSLs can significantly improve developer productivity. Obviously, to achieve this high productivity, one needs to have a proper modeling language for the domain under investigation. To define a language, another language is needed to specify the definition in. The language used in specifying a model is often called a metamodel; hence the language for defining a modeling language is a metametamodel. Metamodeling [3] is a user-friendly, graphically supported way of avoiding manually coding the DSLs unnecessarily. Metamodeling tools give the ability to edit a metamodel, which defines the rules of a model. The metamodel determines which types of objects are allowed during the modeling process and what kind of attributes or relations they can have. A metamodel can also contain textual constraints that should be enforced by the environment in the modeling process. The two-layer model-metamodel system can be extended to an arbitrary layered one. For instance, if we define a metamodel for the UML class diagram, we create a three-layer instantiation model, in which the topmost layer is the defined metamodel, the class diagram represents the middle layer, while the lowest modeling layer is the object diagram. Notice that in this chain, the middle layer behaves as an instance of the upmost layer, and – at the same time – it is the metamodel of the object diagram.

After being able to create customizable models based on metamodels, the natural need to transform a model into another arises. Model transformation is the process of converting a model conforming to a metamodel to another model, which conforms to another metamodel. The first metamodel is called the source metamodel, the second is referred to as the target metamodel. (The two metamodels do not necessarily differ.) Additionally, sometimes it is useful to handle transformations sequentially as a chain of transformation rules in which the output of one transformation is the input of another. This method of translating a model into another is not trivial, QVT [4] has been proposed by the Object Management Group to handle this task.

As it can be seen a model transformation environment, such as the Visual Modeling and Transformation System [5], has to allow flexible metamodel-based software modeling and also needs to be able to process and execute transformations on the models created.

2 Related Work

Visual Modeling and Transformation System (VMTS) [5] is an n-layer metamodeling and model transformation environment. VMTS has been fully implemented in Microsoft .NET Visual C#. The system benefits from the results of the mathematical background of formal languages, graph theory, category theory, graph rewriting and metamodel-based software model transformation. In VMTS, metamodel rules are automatically forced when editing models. Model transformations are based on graph rewriting techniques. A domain-specific control flow language is used to specify the transformation-steps and rewriting rules are also specified by a visual language. During the model transformation process, the tool facilitates the validation of the constraints specified in the transformation rules. Moreover, VMTS supports OCL-based [6] textual constraints, which can be attached to metamodel elements. Models and transformation rules are formalized and stored as directed, labeled graphs. VMTS consist of several parts as shown in Fig. 1.

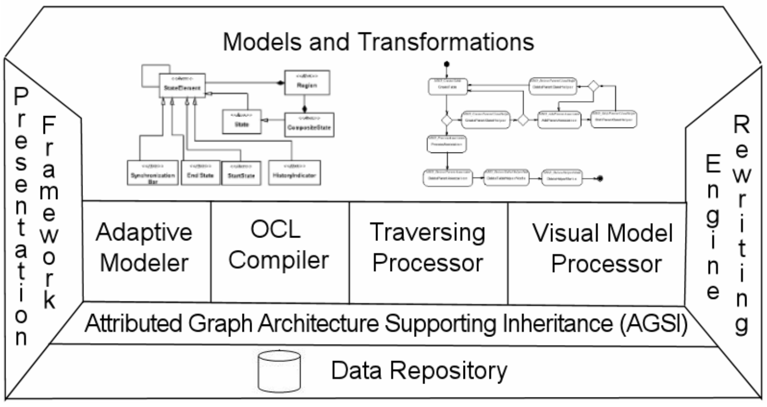


Figure 1
The schematic structure of VMTS

The Attributed Graph Architecture Supporting Inheritance (AGSI) [5] offers a high-level graph interface for the other components to reach the data repository. As previously mentioned, VMTS handles the models as graphs; every graph is stored in the underlying relational database. In this way, the models can be reached from different computers at the same time, thus a team can work on the same project from different locations.

3 Contribution and Improvement

In a modeling tool, such as VMTS, storing the graphs in a database seems to be a good idea; however, in several cases it means a huge inconvenience. Database-based storage raises performance issues in several algorithms as every time a model element is needed by the system for a calculation, the framework needs to query the data from the database. Thus, comparing the performance of an algorithm to other transformation systems is hardly possible in this way. Therefore in-memory data representation is highly needed in VMTS. Obviously, the advantages of the database storage should not be lost in the new system, thus the new solution has to be compatible with the previous one. Furthermore, since the algorithms have already been implemented for the database version, the in-memory solution should not duplicate the code of the algorithms. The following sections introduce the architecture of the developed in-memory transformation system, called AGSI Compact.

3.1 Model Elements

In AGSI Compact, we decided to use a common interface for reaching the model elements with abstract base classes, which have their mode-dependent (DB or Compact) derived classes. Fig. 2 depicts the hierarchy of the base classes.

As previously mentioned, VMTS utilizes graphs as the mathematical background for its models. Vertices and edges are represented in the system by *AgisNodeBase* and *AgsiEdgeBase* classes. An edge can connect two nodes in the system, as it seems obvious. However, a third model element is also present in the system, named *AgsiAssociationEdgeBase*, which acts as an association node defined in UML. A node and an edge can be connected by an association edge, which is rarely needed, however, there are cases when software modeling tasks cannot be solved without this abstraction. The three low level classes are generalized into a model element class, *AgisModelElementBase*, which allows the non-differentiating handling of these classes. In VMTS, the creation of nested, hierarchical models is possible, a node may contain any kind of model elements, therefore, the *AgsiNodeBase* implements the *IAgsiContainer* interface, which declares functions for reaching the child elements.

Creating the abstraction of a modeling task requires a model class, which represents an aspect of the problem, or the whole problem. This model class, *AgsiModelBase*, serves as a container for the nodes and edges created during modeling, thus it implements the *IAgsiContainer* interface. Furthermore, as a model represents a whole modeling task, it also implements the *IAgsiPackage* interface. However, as models have their metamodels and properties as well, it derives from the same class as the model elements, the *AgsiModelItemBase*.

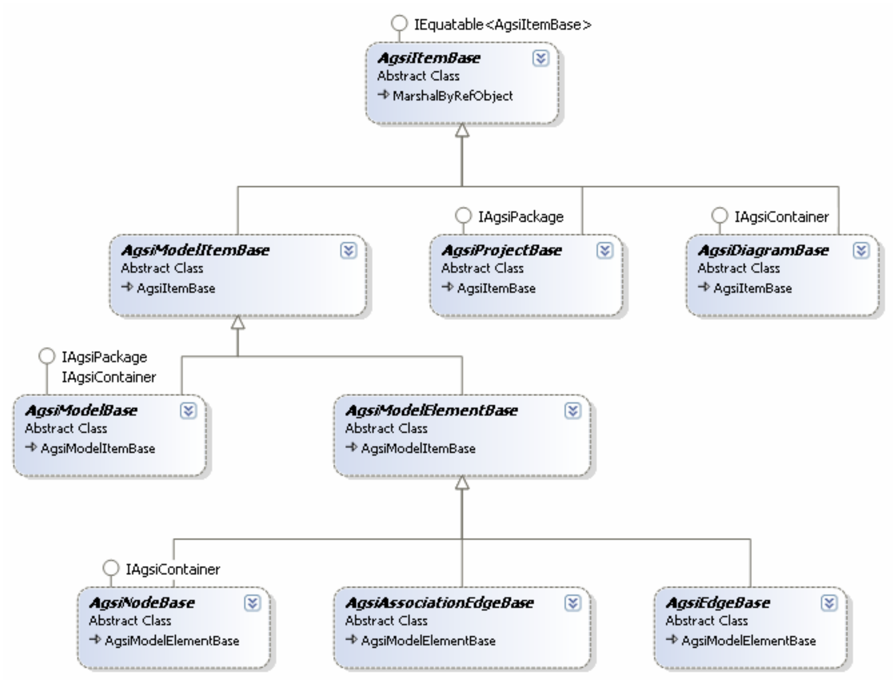


Figure 2
Hierarchy of the base classes

On the next level, three classes are defined (*AgsiModelItemBase*, *AgsiProjectBase*, *AgsiDiagramBase*), which act in different ways. Recall that the *AgsiModelItemBase* class is responsible for the representation of metamodel-based items. The project class acts as a whole task, which may contain several different modeling subtasks. And finally, the *AgsiDiagramBase* is the class that is responsible for the visualization of the models. These three topmost classes derive from the *AgsiItemBase* root class, which only holds a unique identifier and a name as attributes.

3.2 Derived Classes

The general architecture of the classes has been introduced in the previous section. Recall that each abstract base class has its implementations on both platforms (DB, Compact). In this section, the *AgsiNode* classes are presented; however, all the other classes follow the same implementation concept. Fig. 3 shows the *AgsiNodeBase*, *AgsiNodeCompact* and *AgsiNodeDB* versions of the node class. Recall that VMTS is written in Microsoft .NET C# [7], thus the implementation may be based on C# specific constructs, such as properties; however, the introduced solutions may be applied in any object-oriented language.

The obvious difference between the Compact and the database version is that the Compact can keep every data in memory, while the database version has to query everything from the underlying database, even if two consecutive requests need the same attribute value. The base class defines protected member variables, thus those are available in both derived versions. The Compact version utilizes these in-memory variables, however the DB version overrides the defined properties, which retrieves or sets the values. It performs a database query to the appropriate node row in the database every time a value is needed. The unique identifier of the node is stored in the *_node* protected member variable defined in the *AgsiNodeDB* class.

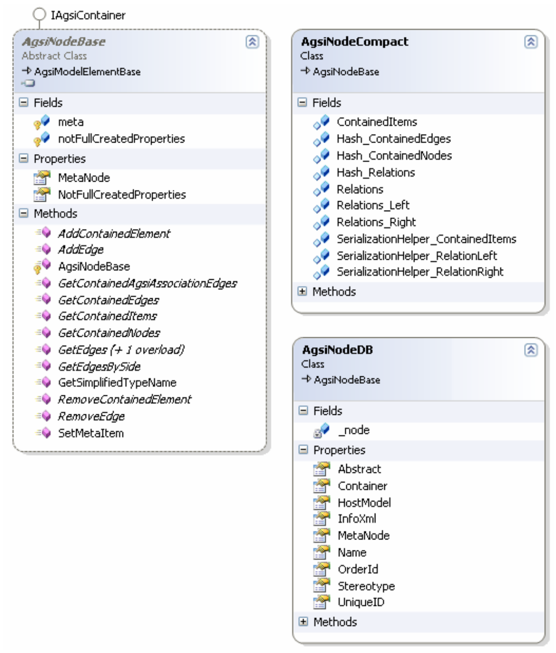


Figure 3
AgsiNode classes

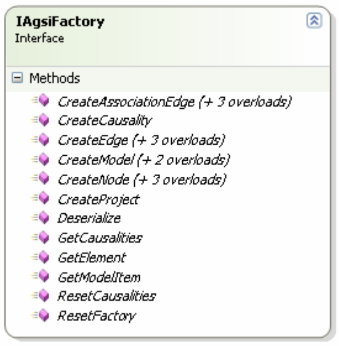
On one hand, the constant database queries decrease the performance in the DB version and it is hardly possible to achieve any increase because the bottleneck of the system is the database. On the other hand, in case of the in-memory version, the performance can be increased towards, for example with hashtables. For instance, in AGSI Compact, a node has not only a list with the edges connected, but has several hashtables indexed by the type of edges. This means that if we want to find a connected inheritance edge, then we have to check the hashtable of inheritance edges only. Caching hashtables are automatically created and maintained by the environment.

Every item has a *delete()* method, which removes the actual element from the system. Naturally, if an element is deleted, the contained elements should also be deleted, and the linked edges as well. The corresponding hash tables have to be maintained in the system, thus the appropriate entries have to be deleted.

3.3 Reaching the Model Elements

It has been shown how each model element behaves in both versions of the system. The AGSI framework has to be connected to several parts of VMTS, which have the tasks of creating/retrieving models, nodes, etc.

To handle these kind of accessibility issues, factory classes have been utilized, which implement the *IAgsiFactory* interface. Fig. 4 shows the interface with its methods.



IAgsiFactory
Interface

Methods
- *CreateAssociationEdge (+ 3 overloads)*
- *CreateCausality*
- *CreateEdge (+ 3 overloads)*
- *CreateModel (+ 2 overloads)*
- *CreateNode (+ 3 overloads)*
- *CreateProject*
- *Deserialize*
- *GetCausalities*
- *GetElement*
- *GetModelItem*
- *ResetCausalities*
- *ResetFactory*

Figure 4
IAgsiFactory interface

All of the 'create' functions work as a factory method, thus, if a requested model item does not exist, then the method creates one with the given identifier. In this way the new and existing items can be handled transparently.

Obviously, in case of the in-memory version it would be difficult to retrieve all the created nodes without a list of the nodes. These requests which concern all of the model items are also placed in an interface, which is implemented by a class for the Compact and DB versions respectively.

As previously only database-based models were supported in VMTS, the identification of a model element was simply based in unique identifiers. In this new multi-mode version, this would not be appropriate, the identifiers have to be changed to base class objects. In this way, the system does not have to know which mode is running currently.

4 Performance

The best way of comparing the performance of our system to others is choosing a benchmark test and run it on several different transformation systems. The basis of the comparison was the case study of [8].

Fig. 5 shows the time required by matching a specific pattern in different transformation tools. The benchmark model and transformation is borrowed form [8], from which we utilized the case of long transformations (long TS) without optimization. The results are measured in ms, for a single application of the rules.

Model size	AGG	PROGRES	Fujaba	DB-based	VMTS DB	VMTS Compact
21	1.86	0.62	0.15	4.15	1326	0.21
5001	1116.34	269.58	0.26	20.47	30000	1.2

Figure 5
Performance of model transformation approaches [8]

According to the results, VMTS DB was very slow because of the underlying relational database. The results show that VMTS Compact requires approximately three orders of magnitude less time to apply the same transformation as VMTS DB. Furthermore, the tests have shown that VMTS (based on the AGSI Compact Framework) is one of the fastest transformation engines.

5 Future Work

Future work includes several tasks which have not been implemented fully in the Compact version. As the AGSI Compact Framework was created primarily to increase the performance of the system and support parallel model transformations [9], no visualization data is supported in the current version. In the database version, the visualization is based on XML data, which stores the visualization information, such as current position, size, and color of each model element. The implementation of this can be expected in the near future.

The switching between modes is not supported either. This means, that currently the user has to choose between the two modes (DB, Compact) before the application is started. It would be highly useful to be able to switch between the modes at run-time. In this case, the user could modify the shared model in the database and – after changing to Compact mode – the transformation could be run in memory, which would mean better performance. In this way, the advantages of both modes could be kept in a single system. Any further work which is not dependent on a specific mode will be implemented on top of the common interface; therefore both versions will become available for the same implementation of the algorithm.

Conclusions

This paper introduced a database-based model transformation system, Visual Modeling and Transformation System developed at our department. The paper has shown how this system can be transformed into another one which supports both database and in-memory use without having to duplicate previously existing code. Naturally, both modes have their advantages, such as in case of database mode, one does not need to care about the distribution of the created models, and several users can modify the same model at the same time. While in case of in-memory use, higher performance of model transformations can be achieved as one of the main bottlenecks of the system is eliminated. Section 4 can be considered part of the conclusion as it shows the achieved performance raise in the system. By eliminating the database from the system, the transformations are executed in three orders of magnitude less time then previously.

References

[1] Unified Modeling Language, http://www.uml.org, September 2007

[2] Mezei, G., Levendovszky, T., Charaf, H.:A Domain-Specific Language for Visualizing Modeling Languages, In Proceedings of the Information Systems Implementation and Modelling conference, ISIM, Prerov, Czech Republic, 2006, pp. 67-74

[3] Colin Atkinson, Thomas Kühne: The Role of Metamodeling in MDA, October 2002

[4] Object Management Group: MOF QVT Specification, http://www.omg.org/docs/ptc/05-11-01.pdf, September 2007

[5] Visual Modeling and Transformation System, http://vmts.aut.bme.hu, September 2007

[6] Object Management Group: Object Constraint Language Specification, http://www.omg.org/docs/ptc/03-10-14.pdf, September 2007

[7] Microsoft Developer Network,
http://msdn2.microsoft.com/, September 2007

[8] Varró, G., Schürr, A., Varró, D.: Benchmarking for Graph Transformation. In Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 05), Dallas, Texas, USA, 2005, pp. 79-88

[9] Mezei, G.: Supporting Transformation-Level Parallelism in Model Transformations. In Proceedings of the Automation and Applied Computer Science Workshop, Budapest, Hungary, 2007