

A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS

Tihamér Levendovszky¹, László Lengyel², Gergely Mezei³
and Hassan Charaf⁴

*Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary*

Abstract

Highly configurable metamodeling environments and graph transformation techniques have been applied successfully in software system modeling and other areas. In this paper a uniform treatment of these two methods is illustrated by a tool called Visual Modeling and Transformation System. The concepts of an n-layer metamodeling environment is outlined with the related topological and attribute issues. Built on metamodeling techniques two alternatives for model transformation are elaborated, namely, the traversal and the graph-rewriting approaches. In our implementation all of the aforementioned mechanisms use metamodel as a common formalism, which can be considered as a uniform basis for storing, creating and transforming visual languages. The feasibility of the approach is illustrated by a transformation which generates C/C++ code from UML statecharts.

Key words: Metamodeling Environment, Metamodel-based Transformations, Model-Driven Software Development, Software Engineering Tools

1 Introduction

Visual modeling environments and model transformations are highly researched fields motivated by, among others, the vision of the Model Driven Architecture [1] issued by OMG. This paper presents the principles of a systematic

¹ Email: tihamer.levendovszky@aut.bme.hu

² Email: laszlo.lengyel@aut.bme.hu

³ Email: gergely.mezei@aut.bme.hu

⁴ Email: hassan.charaf@aut.bme.hu

metamodel-based approach to modeling environments and model transformation via a model editor/storage and transformation software package called Visual Modeling and Transformation System (VMTS). VMTS illustrates an approach in which model storage and model transformation can be treated uniformly and what links them together is the notion of the metamodel. Modeling environments built on metamodeling are highly configurable (visual) modeling tools allowing constraints to be specified in advance. Model transformations can be used for model and code generation or modifying models. One of the most promising directions is to create general transformation systems: the most used UML processors need to incorporate rich semantic information to specify the transformation rules, since currently the standard UML has semantics in plain English, which cannot be formalized to help the rule formulation. As it is illustrated in VMTS metamodeling can be the basis of model transformation methods as well.

The design objectives of VMTS were the following: it is a proof-of-concept implementation for the uniform approach to metamodel-based storage and transformation. UML compliance is an important feature to ensure the practical applicability. Its basic formalism is a labeled directed graph to ease the creation of the formal background and to directly illustrate the mathematical results. VMTS applies standard technologies like XML and enforces the separation of concerns: the presentation, storage and validation modules have been separated as clearly as possible. VMTS has been implemented in .NET using C#, but here language and environment independent principles are focused, only XML support and an object-oriented language are assumed. This paper is devoted to the concise discussion of concepts without delineating the implementation details. For additional information (constraint handling, implementation details, comparison to MOF, UML metamodels) please refer to [2].

2 An n-layer metamodeling environment

Modeling environments have to face the challenge of change (even in standard UML, e.g. compare UML v1.4 and v2.0) and the varieties of models (UML sequence diagram, class diagram and feature models). To save development efforts meeting these requirements, VMTS use metamodels to create a flexible, visually configurable modeling environment. Metamodeling is based on the instantiation relationship i.e. the relationship between the UML class diagram elements and UML object diagram elements. It means that VMTS uses a simplified UML class diagram as a metamodel language to define models (e.g. class diagram, state diagram). If the UML class diagram is instantiated there are three layers involved: the UML object diagram, the UML class diagram and the metamodel of the UML class diagram. Beside these there are two more layers in VMTS: the read-only meta-metamodel which specifies the metamodeling language and the one in the internal structure (this is a labeled directed

graph). The model storage part of VMTS is called *Attributed Graph Architecture Supporting Inheritance* (AGSI). AGSI layers are designed so as every model can be a metamodel for others, but the five layers discussed above have turned out to be enough in practical scenarios. Figure 1 presents the outline of read-only meta-metamodel (bottommost layer), the statechart metamodel (middle layer), and an example statechart diagram (topmost layer).

2.1 Topological Considerations

AGSI can handle graphs via three basic constructs: (i) nodes, (ii) directed edges and (iii) labels assigned to nodes and edges. So far it would be a storage structure for directed labeled graphs, thus metamodeling support needs to be added. Each node and edge holds a bidirectional connection to other nodes and edges, respectively: this is the type-instance mapping. These mappings must not form a loop so as graphs can be organized into tiers, each of which corresponds to a modeling layer. Although this structure is capable of storing model topology, attributes and appearance information in the labels, modern modeling languages and environments (including UML) claim for additional notions: (i) Models should be traversed via a containment hierarchy: every node has a unique parent which contains it possibly with several other elements. Basically this construct is more suitable for easier traversal and better-structured storing, than displaying the model in a tree view. Consequently this structure is not a concept supporting visual presentation even if the presentation benefits from it. For containment hierarchies AGSI maintains a parent-child bidirectional mapping. (ii) Inheritance support (a directed mapping from the descendants to the ancestors which must not contain loops) is a natural requirement for every modeling system even in the metamodel levels. (iii) Association classes are unusual constructs, because a class (association class) is connected to the middle of an edge (association). Although this arrangement could be resolved by inserting a pseudo-node with no semantic meaning, but able to be stored in regular graph structure, for conceptual reasons we decided to add support for this model element as well.

All the aforementioned features have support hard-wired in AGSI. Furthermore, inheritance needs special treatment from the topological angle: descendant types inherit the relationship types from their ancestor types, i.e. instances of the descendant types can be adjacent to the instances of the types adjacent to the ancestor type as well. For instance in Figure 1 two `TransitionEndPoints` can be connected via `Transition` links, therefore two `States` can also be connected this way. The read-only root metamodel of AGSI is depicted in Figure 1. *SystemNode* and *SystemRelationship* are provided by AGSI. These are hard-wired constructs corresponding to the node and the directed edge elements of the labeled directed graph model.

The metamodel specifies an abstract `TransitionEndPoint`, which is the base for all elements that can be connected by a directed transition (`Synchroniza-`

tionBar, State). A State can contain zero or more TransitionEndPoints. On the topmost layer a simple statechart is depicted, which asks for a two digit code. There is a 5 second time limit for entering the code, and in case of a wrong character the device can be cleared to start typing again. On success the protected door is opened. A more complex case study taken from the UML 1.5 standard has also been solved, and it is available in [2].

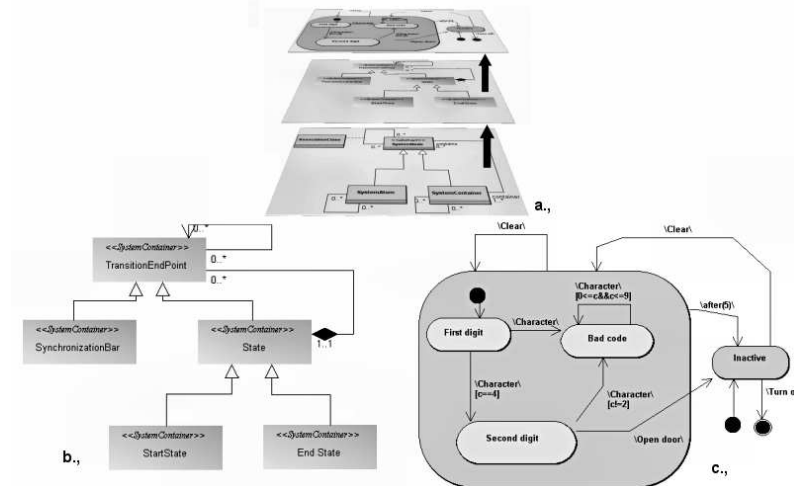


Fig. 1. The metamodel hierarchy of AGSI with a sample statechart diagram

2.2 Attribute Issues

In the labeled directed graph model attributes are stored in the labels. In AGSI the labels are XML documents, and the attributes are represented in an XMI-like format. If the attribute data structures are instantiated, there are two types of attributes: (i) metaattributes, which can be instantiated, and (ii) attributes residing in metamodel elements, but not appearing in the instantiated model elements. Typical examples are attributes and methods in UML classes. Attributes (e.g. $x : integer$) are instantiated (e.g. $x = 10$), but methods (e.g. $f(x : integer) : bool$) do not appear in the corresponding objects. In AGSI the metaattributes are converted to an XSD file, which is the schema for the XML file storing the attributes on the instance level. This method is quite flexible: adding, removing and changing attributes requires altering an XML file which can be processed by tools easily in any modern programming environment. Figure 2 illustrates the XML documents for the metastate-state instantiation chain along with the visual presentation.

There are, however, attribute-related issues of the features discussed in the previous section. The descendants inherit the attributes of the ancestors, thus the XML files should be merged along the hierarchy such that the root element has the lowest priority and the lowest descendant has the highest priority if an attribute with the same name is specified in more than one element in

the inheritance hierarchy path. Abstract nodes cannot be instantiated. Since edges (SystemRelationship) can have attributes as well, AGSI does not need association classes to specify metaattributes for edges, however this facility is supported as well. The attributes stored in the edges have higher priority than the ones residing in the association node when they are merged.

Meta layer XML file and Attributes panel:

```
<Class>
  <InstanceName>State</InstanceName>
  <Attribute>
    <Name>Name</Name>
    <TypeExpression>xsd:string</TypeExpression>
    <Multiplicity>1</Multiplicity>
    <Description>The name of the state</Description>
  </Attribute>
  <Attribute>
    <Name>History</Name>
    <TypeExpression>xsd:string</TypeExpression>
    <Multiplicity>1</Multiplicity>
    <Description>History support</Description>
  </Attribute>
  <Attribute>
    <Name>InternalTransitions</Name>
    <TypeExpression>InternalTransition</TypeExpression>
    <Multiplicity>0..*</Multiplicity>
    <Description>Internal transitions</Description>
  </Attribute>
</Class>
```

Instance layer XML file and Attributes panel:

```
<State>
  <Name>SecondDigit</Name>
  <History>Deep</History>
  <InternalTransition>
    <EventName>OpenDoor</EventName>
    <Parameter></Parameter>
    <GuardCondition></GuardCondition>
    <Action>Q_TRAN(&Pin::Inactive)</Action>
  </InternalTransition>
</State>
```

Fig. 2. Attribute instantiation in AGSI for metastate-state

3 Model Transformations

Considering software applications there are two important categories of transforming a labeled directed graph. The simplest and the most universal way is the traversal approach, the other uses graph rewriting as a transformation mechanism. This section introduces these approaches and reveals another aspect of the motivation: applying these already invented mathematical results to create a formal background for metamodel-based transformation methods, and the extension of the existing results with respect to the presence of the metamodel as the transformation and the modeling formalism.

Model transformation means converting an input model that is available at the beginning of the transformation process to an output model. Model Driven Architecture (MDA) [1] sets out a more restrictive definition: the output model should describe the same system as the input model. As VMTS has been designed to be able to specify more general transformation systems, not only MDA model compilers, we omit the equivalence caveat of MDA from our definition.

3.1 *Traversing Model Processors*

The simplest method to transform models is to traverse them using a specific programming language and changing the appropriate parts of the input models or producing an output model. Traversing Model Processors (TMP) offering this approach usually offer five basic graph operations: (i) create node, (ii) connect nodes, (iii) delete node, (iv) delete edge and (v) set label. Of course these operations may vary, but these categories can usually be observed. As far as model transformations are concerned there are usually specific parameters of these operations, e.g. node creation needs type information of the node to be created and set label usually supports attributes.

The models and their elements in VMTS traversing processors are regular objects in an object oriented programming language. The creation of a node having a specific type is the creation of an object having the corresponding type; deletion of a node means destroying the related object; the attributes can be set via member variables named after the corresponding attribute names. The deletion of an edge means removing a reference from one object to another. Edge creation can be performed by adding a reference to the target object in the appropriate array member of the source object, or vice versa as well, if it is a bidirectional edge.

To generate model processor in an object oriented programming language two model levels must be considered: (i) the current layer, which we want to traverse (these will appear in the form of the objects in the programming language) and (ii) the metamodel layer of the current layer, because that will provide the class definitions for the objects. For each metamodel a generator (TraversingProcessorWizard) automatically creates the framework and metamodel-specific code, so as the programmer can traverse the model by the object container, or by variables (references) named after the name attribute of the model element. For instance, if the name of a state is s in the modeling environment, a programming language object is created with the name s instantiating the *State* class). Model processing code generators are special TMPs, in practice they are used most often. In VMTS the framework generation for a traversing processor always includes a code generation part: the metamodel layer is regarded as an UML class diagram, and classes are generated from it.

3.2 *Visual Model Processors*

Visual Model Processors (VMP) do not replace TMPs, instead, they provide a visual alternative way of model transformation. In VMTS VMPs use graph rewriting as the transformation technique. Basically a graph rewriting system is a set of rules that transforms a graph instance to another graph. In graph rewriting there are production rules consisting of the left hand side graph (LHS) and right hand side graph (RHS). Firing a graph rewriting production rule consists of three steps: (i) Finding an occurrence of the LHS in the host

graph. This means a subgraph of the host graph that is isomorphic to LHS. This subgraph is called *match*. (ii) Remove that nodes and edges from the redex graph which are in the LHS but not in RHS. (iii) Glue that nodes and edges to the redex which are in RHS, but not in LHS. This technique has been used successfully in several transformation systems, some of them are pointed out in Section 4. In model transformation systems the host graph is the input model of a transformation step, that (LHS, RHS pairs) consist of input model elements. VMTS, however, takes a novel approach introduced in [3], which specifies the rewriting rules (LHS and RHS) in terms of the metamodel. Consequently, instead of finding a direct occurrence of LHS, a part of the input model must be found which instantiates LHS.

In this paper we do not concern the sequencing of the transformation steps but examine one step. In this simplified approach we call input model the model which the rule is applied to, and the output model is the result of the transformation step. In VMTS LHS and RHS are defined using UML class diagram syntax. The constructs that make up a metamodel-based LHS specification are inheritance and multiplicity support. Inheritance support is analogous to the natural type compatibility of OO languages: the derived class can always be passed where the ancestor class is expected. It means that a class element in LHS is always matched to its descendant types in the input model. That enables generalization in the rules as well as abstract types. Multiplicity support is accomplished by allowing multiplicity values on the association ends. The most important differences between the LHS and a metamodel are the following: (i) LHS is required to be connected (i.e. a path must exist from any class to any class ignoring the association types), (ii) the match found for LHS is maximal in a sense that the actual matched multiplicity value (the number of links matched to the association) is the greatest possible value from the specified multiplicity interval, that do not contradict to the other part of the match. It is a local construct, so all the matches in the input model are not found and compared to achieve the greatest actual multiplicity, it only means that no more links can be added to the match which is also a match for a particular LHS.

Figure 3 depicts the rule specification in AGSI. The rule creates the CodeDOM tree from statechart diagrams. CodeDOM is an abstract graph-based code representation, from which .NET framework is able to generate e.g. C++ code automatically. We used the Quantum Framework [4] libraries along with the generated code to run the state machine. The concept of CodeDOM is platform independent, a corresponding construct can be for instance the Java Document Model of the Eclipse JDT Core [5]. After building a metamodel for CodeDOM, a rewriting rule in Figure 3 is applied exhaustively. The rule takes a state together with all the states targeted by the outgoing transitions (if any) and generates the corresponding class declaration, member fields and methods with parameters and code parts. In the rules only metamodel elements are used both from statechart meta and CodeDOM meta, and the instantiations

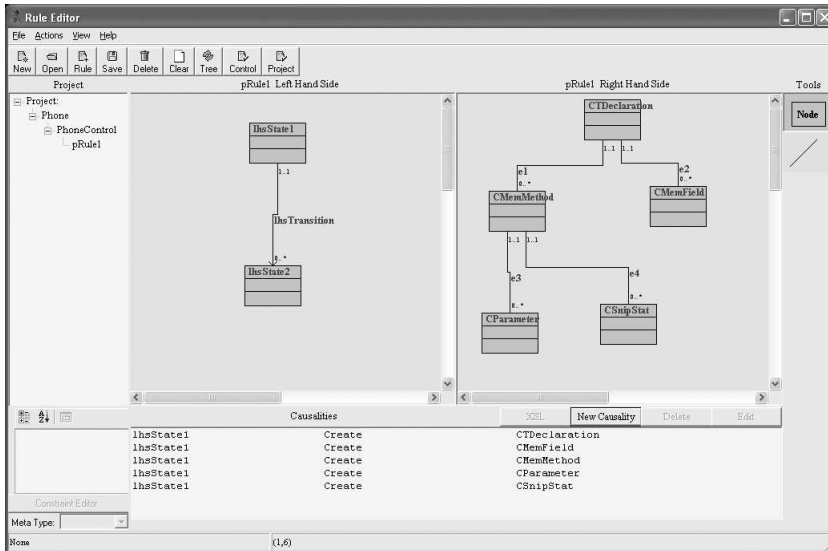


Fig. 3. Rule specification in AGSI

are found for the LHS and created for RHS.

The semantics for LHS has been described, but processing RHS (firing the rule), however, is related to the attribute transformation issues: if an edge in RHS has an exact multiplicity, then the related object must be filled with attributes; if the multiplicity value is not determined uniquely (like *), the number of the created attribute set determines the objects to be created. The attribute transformation is accomplished by XSLT scripts. LHS elements can have so-called causality relationships to RHS elements. Causalities can express the modification or removal of an LHS element, and the creation of an RHS element. Figure 3 shows the causalities for the statechart code generation. XSLT scripts can access the attributes of the object matched to LHS elements, and produce a set of attributes for the RHS element the causality points to. This method was successfully applied in code generation from state diagrams with VMP, and the idea can be extended easily (even to a more sophisticated language than XSL) when it is needed. The XSL snippet in Figure 4 shows how the transformation creates a new method for a state in Quantum Framework.

```

<CodeMember Method>
  <Name><xsl:value-of select="//Name"/></Name>
  <ReturnType>QSTATE</ReturnType>
  <Attributes>private</Attributes>
</CodeMember Method>

```

Fig. 4. XSL snippet creating new method for a state

4 Related Work

VMTS introduces a new approach unifying the metamodeling environments and model transformation on a metamodel based common formalism. Since it is strongly built on the research and experiments included in existing systems from both individual fields. Generic Modeling Environment (GME) [6] is a highly configurable metamodeling tool supporting two layers: a metamodel layer and a modeling layer. GME can be used for metamodel editing when the metamodel is the edited model, and via a traversal processing procedure (MetaInterpreter) it is converted into a metamodel layer for the model editing usage. GME interpreters are analogous to TMPs. The model and the metamodel layer in GME cannot be treated uniformly, as in the two-layered Universal Data Model (UDM) [7], however, the metamodel and the model layer are implemented with the same data structure. This concept has been generalized in VMTS to an n-layer metamodeling environment. Meta-Object Facility (MOF) [8] defines a four layered architecture, but modeling environments are not among the intended usage scenarios, thus guidelines are not provided for that purpose. NetBeans Metadata Repository (MDR) [9] is a MOF compliant repository which offers access to the repository via Java Metadata Interfaces (JMI) and XMI instead of explicit transformation support. Using its metamodel-independent reflexive API, it is possible to write TMPs. Eclipse Modeling Framework (EMF) [5] is a modeling environment with TMP support. There are numerous graph transformation systems, e.g. [10][11][12], VMTS has been most influenced by the GREAT model transformation system [13] (causalities, parameter passing) and PROGRES [14](cardinality assertions).

5 Conclusions

This work has presented a system unifying model storage and transformation facilities. What brings together these techniques is the metamodel-based specification. The paper has illustrated the feasibility of this systematic and uniform treatment of the modeling environment and the transformation system. The flexibility and the configurability of the storage system have been achieved by metamodeling. It has been illustrated that a modeling environment built on metamodeling techniques needs a few hard-wired constructs: node, edge, labels (practically in XML format), inheritance, containment hierarchy and association class. The existence of the metamodel has facilitated mapping the model elements to any object oriented programming language in case of traversing model processors. Visual processors use metamodel elements in the rule specification, ensuring a uniform treatment of powerful novel constructs and those applied in other transformation systems. Due to their overlapping, but not equivalent application areas, visual and traversing model processors should co-exist as appropriate tools for different objectives. It has been shown, that one type of instantiation (the one between UML class

and object diagram elements) is enough to build a n-layered metamodeling environment (as opposed to MOF), and metamodels can be specified in a way suitable for all of the discussed purposes. If a developer is familiar with UML (class diagram, object diagram, OCL), it is really easy to use the same concepts and language for describing model transformations as well.

References

- [1] MDA Guide Version 1.0.1, OMG, document number: omg/2003-06-01, 12th June 2003. <http://www.omg.org/docs/omg/03-06-01.pdf>
- [2] Visual Modeling and Transformation System Homepage. <http://avalon.aut.bme.hu/~tihamer/research/vmts/>
- [3] Levendovszky T., G. Karsai, M. Maroti, A. Ledeczi, H. Charaf, “*Model reuse with metamodel-based transformations*”, Lecture Notes in Computer Science - ICSR7, Austin, TX, April 18, (2002), 166-178
- [4] Samek M., “*Practical Statecharts in C/C++*”, CMP Books, 2002
- [5] Budinsky F., D. Steinberg, E. Merks, R. Ellersick, T.J. Grose, “*Eclipse Modeling Framework*”, Addison-Wesley, 2003
- [6] Ledeczi A., A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, “*Composing Domain-Specific Design Environments*”, Computer, November, (2001), 44-51.
- [7] Magyari E., A. Bakay, A. Lang, T. Paka, A. Vizhanyo, A. Agrawal, G. Karsai, “*UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages*”, The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anaheim, California, October 26, (2003)
- [8] OMG Meta-Object Facility <http://www.omg.org>.
- [9] NetBeans Metadata Repository Homepage: <http://mdr.netbeans.org/>.
- [10] Varró, D., “*Automated Model Transformations for the Analysis of IT Systems*”, Ph.D. thesis, Budapest University of Technology and Economics, Budapest, 2003.
- [11] FUJABA (From Uml to Java And Back Again) Homepage: <http://wwwcs.upb.de/cs/fujaba/>.
- [12] The Attributed Graph Grammar System (AGG) Homepage: <http://tfs.cs.tu-berlin.de/agg/>.
- [13] Karsai G., A. Agrawal, F. Shi, “*On the Use of Graph Transformations for the Formal Specification of Model Interpreters*”, Journal of Universal Computer Science, Volume 9, Issue 11, November, (2003), 1296-1321
- [14] Zündorf A., “*Graph Pattern Matching in PROGRES*”, Graph Grammars and Their Applications in Computer Science, LNCS 1073, J. Cuny et al. (eds), Springer-Verlag, (1996), 454-468.