Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 211 (2008) 181–190
www.elsevier.com/locate/entcs

View Creation of Meta Models by Using Modified Triple Graph Grammars

Johannes Jakob[1] Andy Schürr[2]

Real-Time Systems Lab
Darmstadt University of Technology
Darmstadt, Germany

Abstract

Model-based software development is a hot topic of the software engineering community. Most activities in this area including the standardization efforts of the OMG are targeted towards the development of meta modeling tools, adaptable code generators, and model transformations tools. The needs for the specification of model views that simplify the definition of model transformation, abstract from details of specific modeling languages and tools or support the adaptation of generic modeling approaches to a specific domain are usually out of scope. This paper presents, therefore, a unified approach for the declarative definition of updatable model views. New interpretations of the well-known concept of triple graph grammars are used for that purpose which support, for the first time, the construction of non-materialized views. The adaptation of the presented approach to the world of the Model Driven Application development standards of the OMG and the recently finalized model transformation language QVT is under development.

Keywords: view creation, meta modeling, triple graph grammars

1 Introduction

Any system engineering process requires the manipulation of development artifacts at various levels of abstraction. Keeping all these artifacts and their traceability relationships in a consistent state often turns out to be a nightmare. This is especially true for model-based software engineering, where often many modeling tools are used in parallel, e.g. for requirements elicitation purposes, safety and security analysis, and software design. Today available model integration and transformation approaches offer semi-automatic support for preserving the consistency of the data of these tools. They start with the definition of meta models for the regarded modeling languages on a specific level of abstraction and they add rules for constraint checking and update propagation purposes between instances of different

[1] Email: Johannes.Jakob@es.tu-darmstadt.de

[2] Email: Andy.Schuerr@es.tu-darmstadt.de

doi:10.1016/j.entcs.2008.04.040

meta models. The specification of different layers of abstraction as *meta model views* is usually out of scope. Future meta modeling and model transformation languages should offer better support for the views construction purposes for the following reasons:

- So-called *view points* are a popular software engineering concept that allows one to look at the same integrated model from different perspectives. These viewpoints are just a special case of views that abstract from and hide irrelevant details of the underlying model.
- Domain-specific or even *project-specific modeling approaches* are either based on meta-case tool technology that excludes the usage of standard modeling tools or resort to the implementation of specific wrappers and add-ons on top of standard tools. In some cases these wrappers are a special kind of (meta) model view.
- Furthermore, it is often necessary to decouple the implementation of model checking and model transformation tools from *vendor-specific tool interfaces*. Again views are a standard technology to standardize the modeling concepts and interfaces of a family of tools for the same domain and to simplify the replacement of a specific tool.

These were our motivations for a new line of research that uses the declarative model transformation approach of triple graph grammars (*TGGs*) as a starting point. A new variant of *TGGs*, so-called *VTGGs*, are introduced for view specification purposes. Combining *TGGs* and *VTGGs* results in a *unified meta-model-based view definition and model transformation approach*.

The rest of this paper introduces *VTGGs* and is structured as follows: In Section 2 an example of two interdependent meta models is introduced, where one meta model plays the role of a view onto the other one. A short discussion of related work concerning the construction of (database) views is presented in Section 3. The following Section 4 introduces *VTGGs*, a modified variant of *TGGs* used for view specification purposes. Section 5 concludes this paper, discusses open issues, and future work. Please note that a detailed presentation of the translation process of *VTGG* rules into view implementing graph transformation rules had to be omitted due to lack of space.

2 Running Example

This section introduces the running example that is used throughout the rest of the paper. The overall scenario we have in mind is related to a model-based software development process (Fig. 1a). A software engineer uses two different COTS (commercial of the shelf) tools for requirements elicitation and software design purposes (like DOORS and Matlab/Simulink). Initially, these tools are not related to each other and offer rather generic means as well as APIs for any kind of software engineering project. Engineers using these tools are faced with the following problems: they have no means to abstract from the details of a specific tool's API, and they are usually running into problems when domain-specific development data integration and transformation rules have to be specified. Not only for the last purpose, model transformation languages are needed that are able to operate on top of model views. In more detail, a view definition approach is needed that supports

- definition of multiple overlapping views for a single model,
- update propagation from views to models and vice versa,
- synchroneous and asynchroneous propagation of changes,
- manipulation of virtual (non-materialized) views,
- high-level declarative specification of views,
- model-to-view mappings with complex restructuring operations.

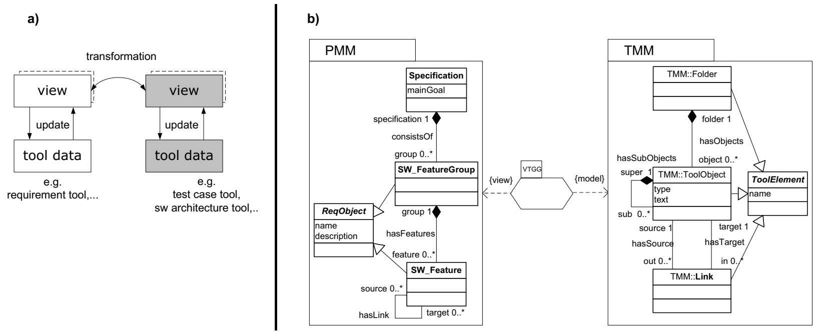

Fig. 1. a) Range of use of view creation, b) VTGG meta model (schema)

In the sequel, we will first present the example of a project-specific meta model (view) definition (PMM) together with its related tool-specific meta model (TMM) using MOF 2.0 as a meta modeling language (cf. Fig. 1b). The PMM defines some basic concepts for a feature-oriented requirements engineering process, whereas the TMM reflects to a limited extent the data models of general purpose requirements engineering tools like DOORS. In principle, specialization is only with the usage of non-derivable abstract super-classes possible. Due to lack of space, the correspondences are simplified to the VTGG package between both meta models.

2.1 *Project Meta Model*

Fig. 1b) (PMM package) shows the project-specific meta model of our requirements engineering example. The class `Specification` represents the root of a two-level hierarchy of features of a regarded software product. Each project has only one `Specification` instance (a project is represented by the PMM). In addition, the `Specification` has an attribute `mainGoal`, a text block which explains the most important goals of the software development project. Furthermore, a specification may have an arbitrary number of `SW_FeatureGroup` (software feature group) elements. For the purpose of this paper we present only one kind of feature group and omit, e.g., a distinction between system, hardware, and software features. The `SW_FeatureGroup` as well as `SW_Feature` inherit the attributes `name` and `description` from the abstract class `ReqObject`. The attribute `name` introduces a short identifier, the attribute `description` a text block with a more detailed explanation of the regarded requirement. Moreover, `SW_FeatureGroup` is a structural element that is used as a container for a group of related software describing features `SW_Feature` (cf. association `hasFeatures`). Each `SW_Feature` belongs to only one `SW_FeatureGroup`. Finally, the association `hasLink`) is used to link related features to each other.

2.2 Tool Meta Model

The tool-specific meta model depicted in Fig. 1b) (TMM package) represents a cut-out of the data structure of a typical requirements engineering CASE tool. The class `folder` is the top-level structural element of this tool. A `Folder` contains a set of `ToolObjects` (association `hasObjects`). Any `ToolObject` possesses a `type` attribute as well as a `text` block attribute. Additionally, it contains a set of `ToolObject` instances in turn (cf. association `hasSubObjects`). The `type` attribute is used to clearly distinguish different sorts of requirements objects stored in the regarded tool; text fragments of (almost) arbitrary length are assigned to `text` attributes. Furthermore, two `ToolObjects` may point to each other via associations to separate `Link` class instances (cf. associations `hasSource, hasTarget`). All introduced classes of the TMM inherit the attribute `name` from the abstract class `ToolElement`.

3 Related Work

In the previous section we have already outlined our requirements for a model transformation approach that supports definition and manipulation of model views. When we are looking for tools that offer this kind of support we have to distinguish two different categories: meta-case tools like Pounamu [17] or MetaEdit+ [11] mainly use the term "model view" as a short-hand for "visualization of a model" (Model-View-Controller design pattern). What we have in mind in this paper is something different: logical model views in the sense of the database community that are again models and not just visualizations of models. A majority of the aforementioned tools offers rather specific support for the visualization of models, but no support for the definition of logical views. Approaches like MViews [5] or view transformations for AHL nets [3] are borderline cases. They support the definition of logical views, but presented examples deal with visualizations of models only.

To the best of our knowledge no meta-case tool or model transformation approach fulfills our model view definition requirements. This is especially true for OMG's QVT (Query, View, and Transformation) [1] language standard which excludes in its current version explicitly any support for the definition of views. Of course, one may argue that a view of a model is just another model which is kept consistent with its underlying model using standard model transformation techniques. ATOM3 is a prominent example of a meta modeling tool that favors this approach [6]. ATOM3 adopted the proposal made in [13] and uses a special form of triple graph grammars (TGGs) for the declarative definition of updatable views.

These views are mainly used for visualization purposes and complex restructuring operations like mapping of view associations onto model objects (or vice versa) are not yet supported.

The main drawback of all view definition solutions based on unmodified model transformation techniques is that we have to *materialize* all views instead of using more light-weight software engineering concepts for the construction of abstraction layers. What we would like to achieve is that views on top of extensionally defined models are implemented as functional API layers. Therefore, we are looking for an approach that supports the declarative specification of (meta) model views plus the automatic generation of "light-weight" view API implementations based on standard adapter design patterns.

The view definition approaches presented in the database management literature suffer from similar drawbacks. Relational database management systems like Oracle offer very limited support only for the definition of updatable views; traditionally updatable views are restricted to projection of columns and selection of rows of an underlying base table. More sophisticated data integration approaches for data warehouses or federated database systems like AMOS-II [16] often limit their support to queries that are translated and propagated using so-called mediator concepts. And even database management systems like SBQL [9] with their advanced versions of "instead of trigger" concepts rely on rather low-level procedural implementations of the translation of basic query and update operations on views into queries and updates of the underlying databases. The most interesting view definition concepts have been recently added to federated P2P DBMS. They rely on bidirectional schema transformations and maybe used to keep a set of databases with a set of materialized derived views in a consistent state [10]. The basic constructs of these schema transformations are comparable to the low-level operational model manipulation constructs of QVT; higher level specification concepts are not yet available.

To summarize, we are not aware of any higher-level languages and tools that offer support for an integrated specification of model transformations and non-materialized updatable views comparable to the modified triple graph grammar concept presented here.

4 Triple Graph Grammars

This section describes the basics of our unified model view definition approach based on *triple graph grammars (TGGs)*. Since the introduction of TGGs in [14], quite a number of modifications and extension have been published [2,7,8]. The model view definition approach presented here is based on the model transformation extensions of [7], where TGGs have been combined with MOF 2.0. Thus, the meta models of our running example are also MOF 2.0 compliant (Fig. 1b), i.e. MOF 2.0 plays the role of a graph schema definition language for TGGs.

In principle, a triple graph grammar is a regular graph grammar with the empty graph as the axiom and a set of graph grammar rules. These rules generate a language of graphs or, more precisely, the set of all consistent graphs which is a subset of the set of all schema-compliant graphs. The interesting point of a TGG is the fact that specified rules consist of three subrules and generated graphs consist of three related subgraphs. Two components always represent a pair of related graphs and the third component introduces traceability relationships between the regarded pair of graphs. Furthermore, TGGs are used as input for a transformation process that yields a set of regular so-called *operational graph transformation rules* tailored for a specific purpose like “forward” model transformation or “backward” propagation of changes.

In the following, we will introduce a new variant of TGGs, called *VTGGs* for model view definition purposes. VTGGs introduce a new set of restrictions for their rules combined with a new way how to translate TGG rules into regular graph transformations rules.

The following rules demonstrate how to use TGGs for view definition purposes and what modifications have been required for that purpose. The rules are depicted within the scope of the introduced example in chapter 2. The mapping relationships from virtually existing PMM objects to really existing TMM objects are modeled as links between *objects* combined with the tag `{map}` (correspondence node). In all cases elements on the left side of a rule belong to the virtually existing *view* computed from the really existing elements on the right of side of a regarded rule. According to the approach of [8], the creation of new objects or links are denotated with the `{new}` tag. Invariant constraints are denoted as OCL constraints.

4.1 Creating the Initial Model and View Elements

This subsection introduces the initial VTGG rule that matches the empty axiom graph and creates the top-level object of the underlying model and its view. The application of a *TGG* rule to a pair of related model graphs is based on finding matches for all untagged elements of the rule. In the initial state of the derivation of a model and its view there are no objects which could be matched by any pattern.

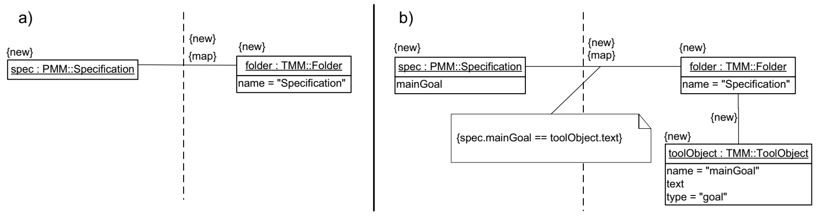

Fig. 2. Mapping of classes, attributes and attributes to classes

Fig. 2a shows a simplified version of the first rule of our example, which is extracted form the rule in Fig. 2b. It specifies that a new `Folder` instance is really created, when we want to create a new `Specification` instance. For this purpose both the `Specification` instance on the PMM side and the `Folder` instance on the TMM side carry the tag `{new}`. All view objects creating VTGG rules have a similar form. A single instance of a new virtual *view* object is mapped onto a single instance of a new real object (often called “seed object” in the DBMS literature). This seed object maybe linked to an arbitrary number of new additional helper objects in the general case. The class name `Specification` (modeled in the PMM 1b) is assigned to the attribute `name` in order to identify the folder as a `Specification`.

More generally spoken, this view shows the simplest form of a VTGG rule, an object to object mapping. Furthermore, the rule shows the simplest case of handling the attributes of virtual (view) and real objects. In contradiction to the meta model of Fig. 1b we do assume that `Specification` instances do not possess any attributes and that they are translated into `Folder` objects with the value `"Specification"` assigned to their attribute `name`. More complex relationships between attribute values of view and real objects are explained later on.

4.2 Creating Isolated Model and View Objects

The following rule shows in addition to the rule above, how to map a virtual attributed *view* object onto a set of real attributed objects.

Fig. 2b displays the entire initial VTGG rule for our running example. Additional to the rule fragment described above (Fig. 2a), this rule maps the attribute `mainGoal` of class `Specification` to an own object `toolObject` on TMM side. Furthermore, the two attributes `name` and `type` are set to constant values for identification purposes. The OCL constraint annotation of the `{map}` relation specifies that the value of the attribute `mainGoal` is mapped to the `toolObjects` attribute `text`. For navigation purposes as modeled in the TMM, a link between the `folder` and `toolObject` is created. Of course, there is no corresponding link on PMM side.

4.3 Creating Context-Dependent Model and View Objeccts

This subsection describes a VTGG rule for creating new objects that are linked to already existing objects.

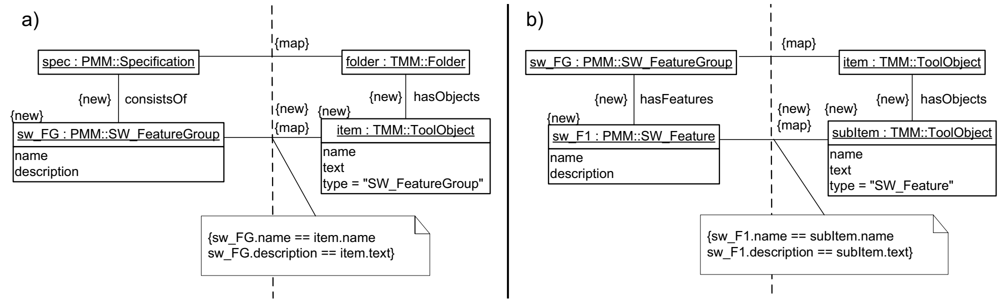

Fig. 3. Mapping of associated classes

In Fig. 3a the new object `sw_FG`, which represents an instance of the class `SW_FeatureGroupe`, is created with an additional new link to the already existing `Specification`. Also `sw_FG` is mapped to a `ToolObject` instance `item` as a *view*, like described in the section before. For identification purposes we assign the constant "SW_FeatureGroup" to the attribute type. Furthermore, the OCL constraint of the rule states that sw_FG.name is mapped onto item.name and that sw_FG.description is mapped onto item.text. In addition, the association of the created sw_FG instance with the context object spec corresponds to the association of the created item instance with the context object folder. In the same manner the rule in Fig. 3b is modeled. The creation of a new instance of SW_Feature corresponds to the creation of a ToolObject instance, subItem. Also the link will be created like described before. The second rule represents the association of ToolObject to itself as shown by the TMM (Fig. 1b).

4.4 Mapping of Associations

As an enhancement of the hitherto existing TGGs (dealing with object mappings only), this subsection describes a new type of rules that translates a virtual association between two objects into an arbitrarily complex substructure of the underlying model graph.

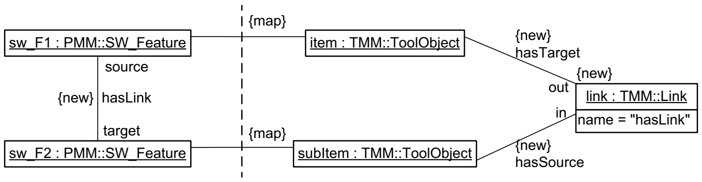

Fig. 4. Mapping of an association to a class

As depicted in Fig. 4, the new link hasLink between the two “existing” instances of SW_Feature corresponds to the new instance of the class Link together with its two associated links hasTarget and hasSource. These links are created at the same time as the new Link object and establish the needed associations to the regarded ToolObject instances. This rule can be used for creating cross-reference relationships between already existing related requirement instances. In order to be able to distinguish different kinds of cross-references the class Link has the attribute name (parallel links are not possible). The association name "hasLink" as a string, is assigned to this attribute.

4.5 Interpretations of VTGGs

One goal of using VTGGs is the automatic translation of VTGG rules into executable Java code. The translation process may associate quite different operational semantics with a VTGG as follows: in a first step we translate a VTGG graph schema into a regular graph schema (MOF 2.0 meta model) and the set of VTGG rules into a set of regular graph transformation rules. Different translations are under development for maintenance of fully materialized views (as described in [de Lara]) “light-weight” views implemented by a layer of adapter objects (object adapter pattern [4] p. 141), and purely virtual views which reuse model objects as adapter objects and which do not create any additional objects or links at all (class adapter pattern [4] p. 141). The generated rules are then compiled into Java code using the standard code generator of the graph transformation tool Fujaba [15]. This flexibility of (V)TGGs in general is one of the main advantages of the presented view specification approach.

5 Conclusion

In this paper we have introduced a modified version of triple graph grammars for view specification purposes. These VTGGs differ from regular TGGs in four ways:

- The view defining side (subrule) of a VTGG rule consists of a single new object (also with links to already existing objects) or link only; this single intensionally defined view element is mapped onto an arbitrarily complex substructure of the extensionally defined underlying model.
- TGG rules used for model integration purposes always add at least one object to each regarded model, whereas VTGG rules for associations add a single link to the model view only.
- The translation of a VTGG into a regular Fujaba graph transformation system does not preserve the involved meta models, but creates a new meta model by weaving the class hierarchies of the input meta models. (not presented here due to lack of space)
- Graph transformation rules generated from VTGG rules do not manipulate two related model instances, but translate read and write operations on the virtually existing model view into read and write operations of the actually existing underlying model. (not presented here due to lack of space)

Precise definitions of different variants of VTGGs are under development as well as an implementation of the (V)TGG approach as a plug-in of the Fujaba/MOFLON meta modeling environment [12].

References

[1] *QVT Merge Group: Revised Submission for MOF 2.0 Query, View, Transformation Request For Proposal (ad/2005-03-02) Version 2.0* (2005).

[2] Becker, S. and B. Westfechtel, *Incremental Integration Tools for Chemical Engineering: An Industrial Application of TGGs*, in: *29th Intl. Workshop Graph-Theoretic Concepts in Computer Science*, LNCS **2880** (2003), pp. 46–57.

[3] Ermel, C. and K. Ehrig, *View transformation in visual environments applied to algebraic high-level nets.*, ENTCS **127** (2005), pp. 61–86.

[4] Gamma, E., R. Helm, R. Johnson and J. Vlissides, Addison-Wesley PCS, Addison-Wesley Publishing Company Inc., USA, 1995.

[5] Grundy, J. and J. Hosking, *Constructing integrated software development environments with mviews*, in: *International Journal of Applied Software Technology Vol.2*, 1996.

[6] Guerra, E. and J. de Lara, *Event-driven grammars: Towards the integration of meta-modelling and graph transformation.*, in: *ICGT*, 2004, pp. 54–69.

[7] Königs, A. and A. Schürr, *MDI - a Rule-Based Multi-Document and Tool Integration Approach*, Special Section on Model-based Tool Integration in Journal of Software&System Modeling (2005).

[8] Königs, A. and A. Schürr, *Tool Integration with Triple Graph Grammars - A Survey*, in: *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, ENTCS (2005).

[9] Kozankiewicz, H. and K. Subieta, *SBQL Views - Prototype of Updateable Views.*, in: *ADBIS*, 2004.

[10] McBrien, P. and A. Poulovassilis, *Defining Peer-to-Peer Data Integration Using Both as View Rules.*, in: *DBISP2P*, 2003, pp. 91–107.

[11] MetaCase, "MetaEdit+," (2005).
URL http://www.metacase.com

[12] Real-Time Systems Lab - TU Darmstadt, "MOFLON," (2005).
URL http://www.moflon.org

[13] Rekers, J. and A. Schürr, *A Graph Based Framework for the Implementation of Visual Environments*, in: *Proc. VL'96 12th Int. IEEE Symp. on Visual Languages, Boulder, Colorado* (1996), pp. 148–155.

[14] Schürr, A., *Specification of graph translators with triple graph grammars*, in: Mayr and Schmidt, editors, *Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*, 1994, pp. 151–163.

[15] Software Engineering Group - University of Paderborn, "FUJABA," (2005).
URL http://www.fujaba.de

[16] T. Risch, T. K., V. Josifovski, *Functional Data Integration in a Distributed Mediator System*, in: *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data* (2003).

[17] Zhu, N., J. C. Grundy and J. G. Hosking, *Pounamu: A meta-yool for multi-view visual language environment construction.*, in: *VL/HCC*, 2004, pp. 254–256.