Electronic Communications of the EASST
Volume 4 (2006)

Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006)

A Model Transformation for Automated Concrete Syntax Definitions of Metamodeled Visual Languages

Gergely Mezei, László Lengyel, Tihamér Levendovszky, Hassan Charaf

12 pages

Guest Editors: Gabor Karsai, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Model Transformation for Automated Concrete Syntax Definitions of Metamodeled Visual Languages

Gergely Mezei[1], László Lengyel[2], Tihamér Levendovszky[3], Hassan Charaf[4]

{gmezei[1], lengyel[2], tihamer[3], hassan[4]}@aut.bme.hu
Budapest University of Technology and Economics
Goldmann György tér 3., 1111 Budapest, Hungary

Abstract: Metamodeling techniques are popular in describing the rules of special domains, but these techniques do not support defining presentation for these domains , namely the concrete syntax. The aim of our research is to provide a method to create the concrete syntax for metamodeling systems in a flexible, efficient way. Several domain-specific languages have been created that support defining the concrete syntax, i.e. the visualization. The main concern of this paper is to present a model transformation method that processes our presentation definitions and transforms them automatically into source code. The source code implements a plug-in capable of editing the models. A termination analysis for the presented method is also provided.

Keywords: Model Transformation, Concrete Syntax, Domain-Specific Modeling

1 Introduction

Special domains of interest require flexible modeling languages. Domain-Specific Modeling Languages (DSML) supported by metamodeling techniques are a widely adopted way to create environments for visual modeling languages. A metamodel acts as a set of rules for the model level: it defines the available model elements, their attributes and the possible connections between them. The definition is constructed using a default, domain-independent notation, often called the abstract syntax. Since metamodeling can fulfill the structural requirements of the selected domain only, additional techniques are required to define the domain-specific presentation of the elements, namely the concrete syntax.

1.1 Problem statement

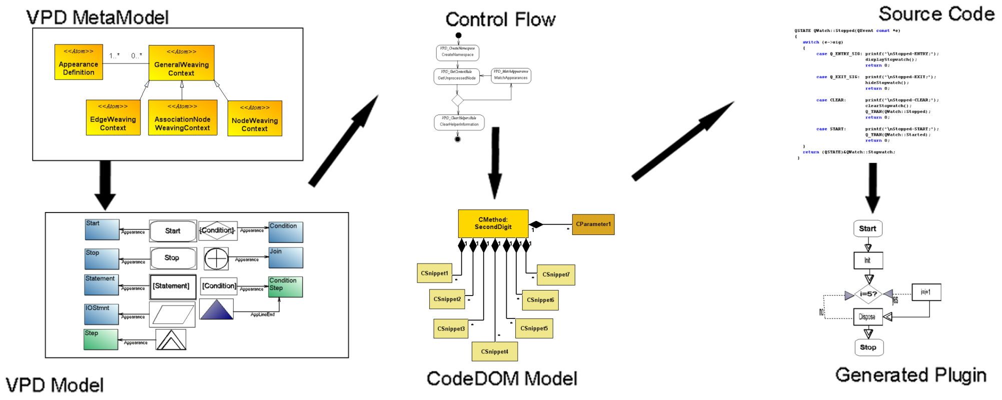
The instantiation relationship and the metamodeling itself are defined by standards, although there are alternative ways. In contrast, handling the concrete syntax definitions is not yet standardized. The custom solutions used in the modeling frameworks are often inefficient and inflexible. The following solution types can be distinguished: (i) manually coding the presentation logic in the modeling framework, (ii) extending the DSML definitions with new properties focusing on the presentation, (iii) using a special DSL that defines the presentation, and then binding the concrete syntax and the structural definition of the metamodel. Previous work [MLHVL06] has introduced these solutions in detail and has found that the third solution is the most straightforward. This solution models concrete syntax definitions by using a common Domain-Specific

Language (referred to as Presentation DSL). The concrete syntax definitions are the models of this *Presentation DSL*. The ability to handle the concrete syntax in the same way as normal DSMLs makes editing much simpler, thus, it means uniformity and flexibility. Another advantage of the solution is that it allows multiple concrete syntax definitions for a single DSML.

Visual Modeling and Transformation System (VMTS) [VMTS] is an n-layer metamodeling environment that unifies the metamodeling techniques used by the common modeling tools, and employs model transformation applying graph rewriting as the underlying mechanism. A metamodeling environment is based on the VMTS Presentation Framework (VPF) [MLHPF05] that is a flexible, graphical modeling framework using a plug-in-based architecture. VPF promotes creating models for UML 2.0 diagrams and other popular domains such as Mobile Resource Editor, or Feature Modeling. VPF plug-ins must be customized for each DSML. The base classes of the framework must be subclassed for each model element to provide customized drawing and event-handling code. The concrete syntax used by the VPF plug-ins was originally defined by manual coding, which meant a huge amount of additional work. The open issues are the following: (i) Is the solution based on Presentation DSL more efficient? (ii) Is a model transformation flexible enough to create source code from the concrete syntax definitions? (iii) Can the transformation engine grant that the transformation will always terminate?

1.2 Architectural overview

VMTS Presentation DSL (VPD) is a *Presentation DSL* realized in VMTS. Fig. 1 shows the main steps of the concrete syntax definition and processing.



```
QSTATE QWatch::Stopped(QEvent const *e)
{
   switch (e->sig)
   {
        case Q_ENTRY_SIG: printf("\nStopped-ENTRY;");
                          displayStopwatch();
                          return 0;

        case Q_EXIT_SIG:  printf("\nStopped-EXIT;");
                          hideStopwatch();
                          return 0;

        case CLEAR:       printf("\nStopped-CLEAR;");
                          clearStopwatch();
                          Q_TRAN(QWatch::Stopped);
                          return 0;

        case START:       printf("\nStopped-START;");
                          Q_TRAN(QWatch::Started);
                          return 0;
   }
   return (QSTATE)&QWatch::Stopwatch;
 }
```

Figure 1: Concrete Syntax Definition - Overview

The VPD metamodel defines the metamodel for VPD, i.e. the structure of the concrete syntax definitions. By instantiating the VPD metamodel, concrete syntax definitions can be created. In order to facilitate the creation of the concrete syntax, a plug-in (the *VPDPlugin*) was implemented based on VMTS Presentation Framework. To improve the effectiveness, there is support for processing VPD models automatically, using model transformation techniques [LL+06]. The transformation describes by the control flow converts VPD Models to CodeDOM models. Code-

DOM is an abstract code representation. From the CodeDOM model, source code is generated with the .NET CodeDOM technology [TH03]. The source code implements a plug-in that can be used directly in VPF. This approach made it possible to avoid manual coding, and create plug-ins in a user-friendly, graphical way in the same environment as common DSL models. The paper [MLHVL06] has presented the VMTS *Presentation DSL* in detail, [MLHC06] has given an overview about the method, but the model transformation has not been introduced in detail. This paper fills this gap and introduces both the transformation control flow and the transformation rules. Termination properties of the transformation are also discussed in detail.

2 Related work

The Generic Modeling Environment (GME) [LBM+01] is a highly configurable metamodeling tool supporting two layers: a metamodel, and a modeling layer. The concrete syntax definitions can be coded either manually, or set by properties both on the metamodel and on the model level. GME supports a special type of property definitions: the registry entries. These entries are assigned to model elements and they can also customize the appearance.

Meta-CASE editors (e.g. MetaEdit+ [MEDIT]) are environments capable of generating CASE tools. They allow creating the tool definitions in a high-level graphical environment, but they supply a manually coded user interface. These environments store concrete syntax definitions in the metamodel properties.

Another framework is the Diagram Editor Generator (DiaGen) [DIAGEN], which is an efficient solution to create visual editors for DSLs. DiaGen is not based on metamodeling techniques; it uses its own specification language for defining the structure of diagrams. DiaGen supports editing the concrete syntax in a graphical context, but in a tree control-based form only, where there is no support to define the shape of the elements graphically. Concrete syntax in DiaGen is based on properties. DiaGen can generate an editor based on the specification using hypergraph grammars and transformations.

AToM^3 (A Tool for Multi-formalism and Meta-Modelling) [LV02]) is a flexible modeling tool. It employs an appearance editor to define the shape of the model elements graphically; it uses model level properties to store the concrete syntax (model definitions are extended with visualization-based attributes). AToM3 can generate plug-ins that use the defined syntax, but the code generation is not based on a *Presentation DSL*. The views of the models are generated with triple graph grammars.

Eclipse [ECLIPSE] is probably the most popular, highly flexible, open source modeling platform that supports metamodeling. The Eclipse Modeling Framework (EMF) can generate source code from models defined by the class diagram definition of UML, but it does not contain concrete syntax definitions. The Graphical Editing Framework (GEF) is also a part of the Eclipse project. GEF provides methods for creating visual editors. EMF does not support code generation for GEF, therefore GEF plug-ins require manual coding to support the concrete syntax.

GenGed [GENGED] is a tool to generate visualization code with graph transformation. GenGed has been replaced by the project Transformation-Based Generation of Modeling Environments (TIGER) [EEHT] that uses precise visual language (VL) definitions and offers a graphical environment based on GEF. TIGER can generate source code from the visual language definitions that implements a plug-in based on GEF. VL specifications can be created graphically. Java is the only language supported in plug-in generation. At the moment TIGER can generate editors for Activity Diagrams and Petri nets.

The Graphical Modeling Framework (GMF) is also an Eclipse project. The goal of GMF is to form a generative bridge between EMF and GEF, whereby a diagram definition is linked to a domain model as an input to the generation of a visual editor. GMF uses a *Presentation DSL* to define the concrete syntax. The result (the linked concrete, and structural definitions) are processed further to produce source code. The mapping between the domain model and the model items of the concrete syntax is also supported in GMF. The generated source code relies on the features of GEF and EMF. Although the concept of GMF is straightforward, it has some weaknesses: (i) the generation is not based on model transformation. Consequently, the compilation steps are coded manually, thus, changing the transformation needs changing the source code and rebuilding the compiler. In case of model transformation such modifications can be accomplished at run-time. (ii) Because of EMF, GMF is restricted to Java only.

3 Defining the concrete syntax

*Concrete Syntax Model*s, namely concrete syntax definitions are created by instantiating VMTS *Presentation DSL* (VPD). *Concrete Syntax Model*s define how the model items of the *Subject Model*, namely, the models of the subject domain are visualized, and how they behave. Fig. 2 shows the metamodel - model, and the structural definition - concrete syntax relationships.

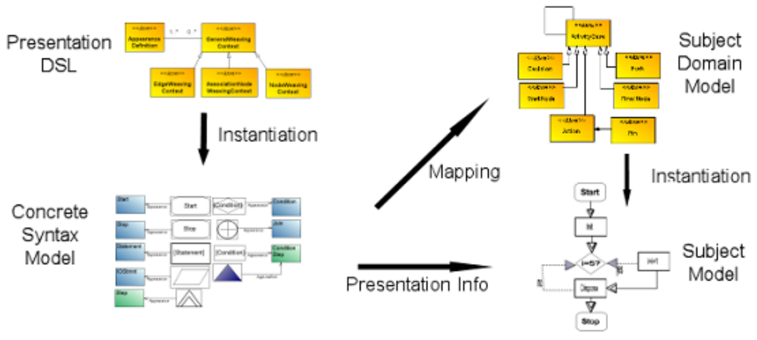


Figure 2: Structural definition - Concrete Syntax Relationship

*Concrete Syntax Model*s are instantiations of *Presentation DSL*. The models of the domain, the *Subject Domain Model*s are created, and the concrete syntax is mapped to the structural definition. *Subject Model*s are created by instantiating the *Subject Domain Model*. The framework displays the model using the generated plug-in by automatically combining the abstract and the concrete syntax.

3.1 The VPD metamodel

The VPD metamodel consists of five nodes as shown in Fig. 3. *Appearance Definition* can describe the graphical notation of the model elements. *Weaving Context*s are used to define behavioral attributes and to store mapping information between the concrete and the structure definition. The name *Weaving Context* describes that these elements weaves two different aspects of the model, the data and the visual definition, namely the abstract and the concrete syntax. For example, in case of ControlFlow diagrams *Appearance Definition*s define the graphical notation, such as rectangle for statement, or diamond for conditions. *Weaving Context*s have a reference to the appropriate metamodel item, thus, *StatementContext* has a reference to *MetaStatement* item in the ControlFlow metamodel. *Weaving Context*s also contain behavioral attributes, such as the minimum size of the model element. A relation between *Weaving Context* and *Appearance Definition*s can be constructed using *Attribute Reference* relations. In the metamodel, the multiplicity of this relation is many-to-many, which means that the appearance definitions are reusable, and the weaving contexts can have several appearances. This reusability is necessary because modeling languages have a tendency to use the same notation in different languages. For example StartState in UML statechart diagrams and InitialState in UML activity diagrams are denoted the same way. Similar separation between the behavioral attributes and the mapping information could be created, but we have found that customized behavioral attributes are harder to reuse. For example, *Input* pins should be aligned to one side of its container in activity diagrams. This property is handled by behavioral attributes (positioning constraints). The constraints describe alignment rules useful only for this type of elements.

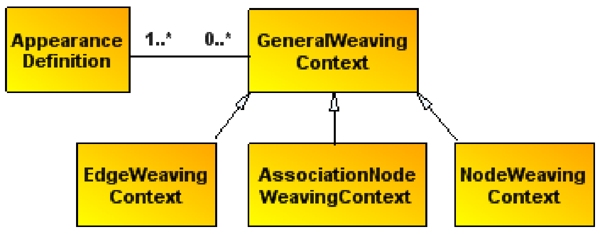


Figure 3: VMTS Presentation DSL - Metamodel

Different fundamental types can have different behavioral and visualization properties, thus, they are distinguished. In VMTS, there are three fundamental types: *Nodes*, *Edges* (relations between nodes) and *AssociationNodes* (e.g. AssociationClass in a class diagram). VMTS *Presentation DSL* mirrors these fundamental types by customizing *General Weaving Context*. For example, the model item *NodeWeavingContext* can express the mapping information for a *Node*. Visualization information described in *Appearance Definition*s is based on *Region*s. *Region*s are graphical units that are independent from each other. A *Region* is responsible for visualizing a part of the model item, or the whole model item. Since the *Region*s are independent from each other, they can be edited separately, and the model representation can be composed of the *Region*s when displaying the model item. *Region* definitions consist of simple graphical objects, called *primitive*s. *Primitive*s are, for example, lines, Bzier splines, or rectangles. A *Region* can contain several primitives, for example an Actor in a Use-case model is defined in a single *Region* although it consists of several primitives (head, body, arms, legs). More information on the VPD metamodel and concrete syntax definition can be found in [MLHVL06].

4 The transformation

Several techniques exist to create source code from a given model. Model transformations can be modeled in a visual way, they can be changed easily and they can use the efficient graph transformation techniques, along with high level transformation constraints, thus, they are one of the most popular solutions. VMTS uses Visual Model Processors (VMPs) to process models with graph rewriting-based transformation techniques. The inputs of a VMTS VMP are the input model and metamodel, the output metamodel, and the control flow model which defines the transformation. The result of the transformation is the output model. The input model is an instance of the input metamodel, and the output model is an instance of the output metamodel. Fig. 4 shows an overview of model transformation.

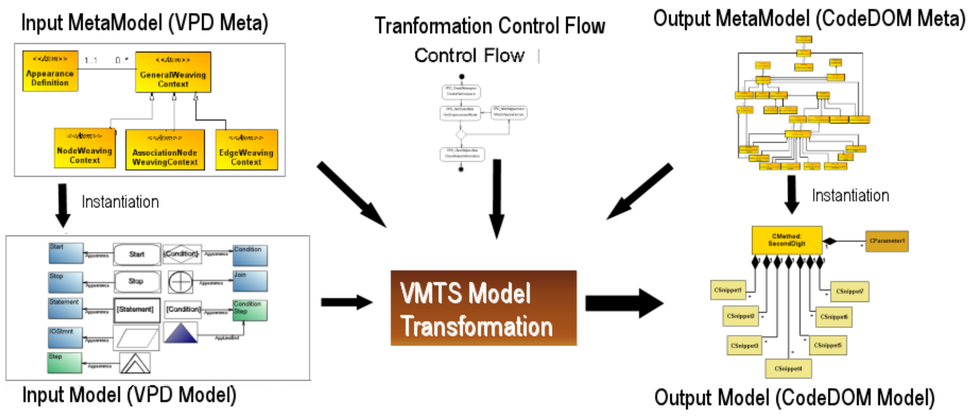

Input MetaModel (VPD Meta)
Tranformation Control Flow
Control Flow
Output MetaModel (CodeDOM Meta)
Instantiation
Instantiation
VMTS Model Transformation
Input Model (VPD Model)
Output Model (CodeDOM Model)

Figure 4: Model Transformation in VMTS - Overview

In this case *Concrete Syntax Model*s, namely, the concrete syntax definitions, are transformed to CodeDOM models [TH03]. CodeDOM supports describing source code as a language independent tree and then generating source code to other languages automatically. Therefore source code generation is easy from the output of the transformation. In VMTS, the control flow for the transformation can be constructed using the Visual Control Flow Language (VCFL) [LL+06]. VCFL is a domain-specific language based on stereotyped activity diagrams. The transformation rules in the control flow specify the operational behavior of model processing. In VMTS, this technique is based on graph transformations [Roz97]. The atoms of graph transformations defined by control flow are graph-rewriting rules. Rewriting rules consists of two parts: Left-Hand Side (LHS) describes the pattern we are searching for, while the Right-Hand Side (RHS) defines the replacement pattern. In VMTS, the LHS and RHS of the transformation rules are built from metamodel elements. Besides the rewriting rules, VCFL also supports decisions, fork, and join items. Model transformation algorithms often require parameter passing between the subsequent transformation rules. In VCFL, external causalities can be defined to pass parameters. In the next sections, we elaborate on the control flow and the corresponding steps of the VPD transformation.

4.1 VPD transformation overview

The control flow of the model transformation is shown in Fig. 5. The first rewriting rule (*CreateNamespace*) is an initialization step for the further operations. The second step (*GetUnprocessedNode*) searches for an unprocessed weaving context in the host model, namely, in the *Concrete Syntax Model*. If it does not find any, then there is no item to process in the model, thus the transformation ends. If there is an unprocessed model item, then the next step (*MatchAppearances*) pass the associated *Appearance Definition*s using the *Appearance Relation*s to navigate, and generates the required CodeDOM items. The control flow uses external causalities and decorates the host model to pass the matching information between the rewriting steps to indicate the current context.

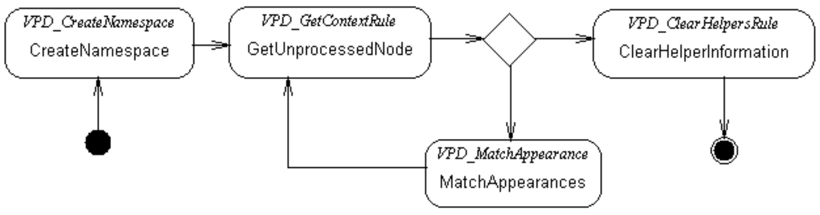


Figure 5: VMTS *Presentation DSL* - Control Flow

4.2 The transformation rules

The transformation initialization consists of two steps: (i) the initialization of the CodeDOM model and (ii) the initialization of the environment of the generated code. The start node of the control flow creates a new model in the underlying database. The newly created model is an empty CodeDOM model that is used as the output model in the later steps. The rule *CreateNamespace* constructs a namespace in the CodeDOM model. Each plug-in class generated later will be contained by this namespace. The step also creates a *DiagramModel* class. In VMTS Presentation Framework, *DiagramModel* classes are used to create a binding between the plug-in and the subject domain. The concrete syntax definition is processed in the steps *GetUnprocessedNode* and *MatchAppearances*. These steps are connected in a loop using a decision item. In VCFL, the decision steps can use OCL constraints, or simply the result of the previous rewriting steps to decide on which branch they continue the execution. In this case the loop exits only if there are no unprocessed node (weaving context) left in the concrete syntax definition (the step *GetUnprocessedNode* was unsuccessful). The step *GetUnprocessedNode* is simple: both LHS and RHS contains a *General WeavingContext* node. Matching information is accomplished using an OCL constraint and a *Virtual Attribute*. Virtual attribute is a special, temporary attribute added to the matched elements during model transformation [ML+06]. Using virtual attributes, the original model items can be decorated without changing their meta definitions. Virtual attributes are removed at the end of the transformation. In this case the rule *GetUnprocessedNode* is based on the virtual attribute *IsProcessed*. The weaving context in LHS is extended by an OCL constraint that ensures that the matched node has not been matched before. The rule also contains a modify type internal causality. Internal causality is a relationship between LHS and RHS nodes, and they define attribute computations. This causality adds the *IsProcessed* attribute to the matched node. The rule *MatchAppearances* is more complex (Fig. 6). It matches the weaving context along with the associated appearance definitions. The *Context* element of the LHS is passed to the rule from the *GetUnprocessedNode* rule using an external causality. An external causality is a parameter passing mechanism which facilitates to assign a host graph node matched to an RHS element to an LHS element of a subsequent rule. The matching algorithm considers these assignments compulsory. A single weaving context can have multiple appearances as mentioned before, thus the relation has a multiplicity of 1..*. The matched context and appearance CodeDOM elements are generated in the RHS. The rule consists of create type internal causalities only.

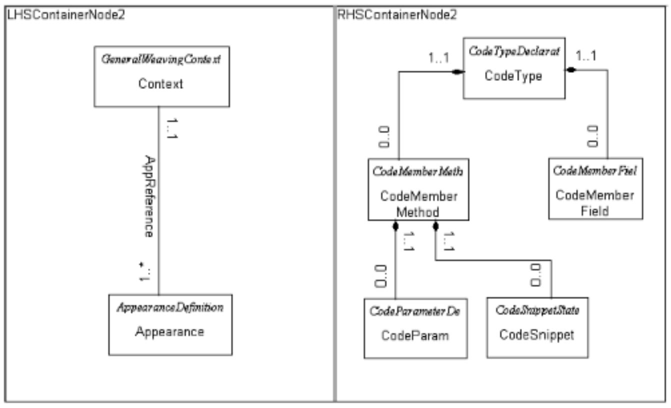
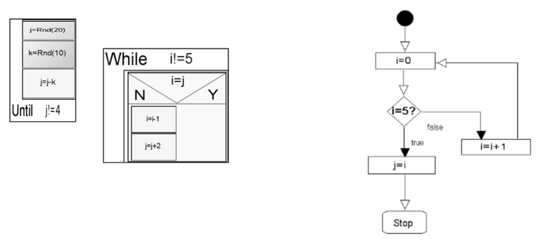


Figure 6: The rule *MatchAppearances*

From the weaving context, three classes (type declaration), a Model, a View and a Controller class are generated according to the MVC-architecture used in VPF [MLHPF05]. The fundamental types, namely, the types of the weaving contexts result in different base classes. For example, an *AssociationNodeWeavingContext* creates type declarations inherited from *AssociationNode* base classes defined in VPF. Binding between the structural definition and the plug-in items is constructed by an attribute containing the ID of the target model item according to the requirements of VPF. Other properties and methods of the classes are defined only if they override the default behavior. Each *Appearance Definition* generates a method in the View class. The methods are called when the model item is drawn out. The main loop of the transformation exits if the CodeDOM model is complete. The step *ClearHelperInformation* deletes the *IsProcessed* attribute from the weaving contexts. The rule is defined as a *MultipleMatch* rule, which means it is applied for each weaving context in the host model. *EndNode* supports a special type of action, *After Action* that is executed at the end of the transformation. This special action is used to process the CodeDOM model and generate the plug-in. VMTS offers a built-in method to apply this task. The generated plug-in can be used directly in VPF.

Fig. 7 shows two examples: the well-known FlowChart and the Nassie-Schneidermann plug-in that were constructed using the introduced method. The concrete syntax was defined in two steps: (i) the notation of the model items were created in a graphical notation editor; (ii) mapping and behavioral properties has been added, such as position constraints for contained elements in Nassie-Schneidermann diagrams. Then, the concrete syntax definition was transformed to source code by a Visual Model Processor, based on the presented control flow. The transformation was not customized for the models, the *same* transformation is used for *every* domain. The generated source code, namely the plugin was compiled, and used to edit the models. The time spent with the construction of the plugins was approximately seven times less, than it would be using manual coding. The FlowChart example is described in more detail (focusing the construction of the concrete syntax, and the generated source code) in [VMTS].



(a) (b)

Figure 7: Example plug-ins - (a) Nassie-Schneidermann (b) Flowchart

5 Termination analysis

Using a model transformation to convert the concrete syntax definition into source code is a straightforward solution, because changes in the framework or in the modeling structure can be easily adopted. 'Easily' means that coding can be avoided; only the transformation control flow and the rewriting rules need to be modified. In contrast, classic model to source code compilers would fail for example if a new fundamental type is required. This flexibility has also some drawbacks: if the transformation changes, then its correctness must be proven again. Using constraints in transformation rules can help in creating a validated model, but there are transformation-level properties, such as the question of termination, which require further examination. The aim of our analysis is to prove that the transformation terminates for every valid finite input model. We use the definitions and theorems presented in [LPE06] to make the proving method simpler. These theorems are proven to injective rules only, but this is not a problem, because the VPD transformation uses injective matches only.

Definition 1 An E-concurrent production p^* is an E-based composition if there is at least one input graph G_0 with an E-related transformation $G_0 \overset{p^*}{\Longrightarrow} H$.

Definition 2 Consider a possibly infinite sequence of graph productions p_i, $(i = 1,2,...)$ and a sequence of E-dependency relations $((E_i, e_i^*, e_{i+1}))$ leading to a sequence of their E-based compositions $(p_i^* = (L_i^* \leftarrow K_i^* \rightarrow R_i^*))$ with $p_1^* = p_1$ and $p_n^* = (p_1 *_{E_1} p_2) *_{E_2} \dots *_{E_n} p_n$.

A cumulative LHS series of this sequence is the graph series L_n^* consisting of the left-hand side graphs of p_n^*. Moreover, a cumulative size series of a production sequence is the nonnegative integer series $|L_n^*|$.

Theorem 1 *A $GTS = (P)$ terminates if for all infinite cumulative LHS sequences (L_i^*) of the graph productions created from the members of P, it holds that*

$$\lim_{i \to \infty} |L_i^*| = \infty.$$

Note that we assume finite input graphs and injective matches.

Proposition 1 *The transformation VPD (depicted in Figure 5) always terminates.*

Proof. At first the transformation rules are examined whether they can affect the termination. The initial, final step, and the *CreateNamespace* and *ClearHelperInformation* rules are executed only once. They are not exhaustive, thus, they do not affect the termination. In contrast, the loop containing the *GetUnprocessedNode*, the decision object and the *MatchAppearances* step are critical. When the transformation is running, the loop is executed until *GetUnprocessedNode* can be matched. We unify the execution of consequent rules in the loop, namely we create the E-based composition of the rules, a new rule that has an equivalent effect on the host graph. The key of the proving method is to show that this unification produces an LHS sequence that exceeds all limits. Recall that the basics of the proving method is borrowed from [LPE06].

The first step in the E-based composition is to unify a single execution of *GetUnprocessedNode* and *MatchAppearances*. Fig. 8/a shows the composition in detail. No other composition structure is valid, because of the external causality between the rules. Empty and crossed circles represent weaving contexts, the cross in the circle means that the *IsProcessed* attribute is set to *true*. Filled circles are used to show appearance definitions. The generated CodeDOM model is not shown, because the CodeDOM model is just an output model, nodes in the CodeDOM model are never matched in the rules of the transformation. Next, the composition is further composed by the next step in the loop. In this case the first rule is the composite rule, the another rule is *GetUnprocessedNode*. The composition step is shown in Fig. 8/b. The new weaving context has an additional circle to show that it is different from the original one. It can be seen that R_{21} and L_{22} cannot be the same node, because R_{21} has the *IsProcessed* attribute set to true, it cannot be matched again. Therefore the composition represented by the figure is the only valid composition. This means also that every time the loop is executed, at least one new node appears in the LHS of the composed rule. Therefore, the LHS sequence in the E-based composition exceeds all limits, thus, the transformation always terminates according to Theorem 1. □

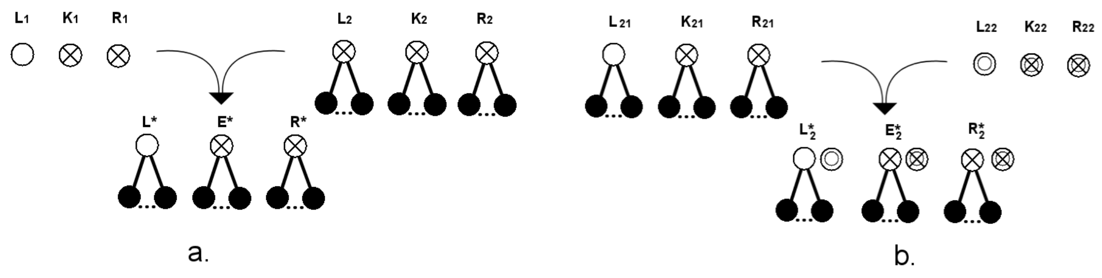


Figure 8: E-based composition

6 Conclusions

While structural definitions of DSMLs can be constructed in an efficient yet user-friendly way using metamodeling, handling the concrete syntax does not have such a well-accepted method. Our approach is a way to solve this problem. Previous work [MLHVL06] has presented the VMTS Presentation DSL, a domain-specific modeling language that can express concrete syntax definitions. This paper has completed the introduction of our approach by presenting the model transformation method that can create plug-ins from the concrete syntax definitions. The transformation control flow and the transformation rules were also presented in detail, including the examination of the termination properties of the transformation. The presented technique grants that the constraints enforced in the metamodel are treated separately from presentation of the concrete syntax. We have provided a simple, expressive model transformation based on graph rewriting to process the VPD models. The presented approach is easier and faster to use than manual coding. The presented transformation is flexible enough to convert the model to source code automatically. It has also been shown that the transformation always terminates. The presented transformation and *Presentation DSL* have been successfully used in practice to model several domains, such as FlowChart, Nassie-Schneidermann, and UML Activity diagrams. Thus, the introduced open issues have been solved.

Different visualization states for model items are currently supported by attaching several appearance definitions to a single context. These definitions are transformed to method definitions in the source code, but current version does not support modeling dynamic behavior: always the default appearance is used. The generated plug-in can be customized by a few lines of code, but our aim is to eliminate coding. The behavior and the different states of the model items can be modeled as a statechart diagram and attaching this behavioral information to the static visualization definitions can solve the problem. Thus, future work focuses on a higher level of automatization.

Acknowledgements: The paper is established by the support of the National Office for Research and Technology (Hungary).

Bibliography

[MLHVL06] Mezei, G., Levendovszky, T., Charaf, H.: A Domain-Specific Language for Visualizing Modeling Languages, In Proceedings of the Information Systems Implementation and Modelling conference, Prerov, Czech Republic, 2006, pp. 67-74.

[VMTS] VMTS Official Homapage, http://vmts.aut.bme.hu/

[MLHPF05] Mezei, G., Levendovszky, T., Charaf, H.: A Presentation Framework for Metamodeling Environments, Workshop in Software Model Engineering, Montego Bay, Jamaica, 2005 (to appear)

[LL+06] Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Control Flow Support for Model Transformation Frameworks: An Overview, In Proceedings of the MicroCad conference, Miskolc, Hungary, 2006, pp 193-199

[TH03] Thuan, T.,Hoang, L.: .NET Framework Essential, O'Reilly, 2003.

[MLHC06] Levendovszky, T., Mezei, G., Charaf, H.: Automatized Concrete Syntax Definition For Domain Specific Langauges, International Conference on Technical Informatics, Timisoara, 2006

[LBM+01] Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments, IEEE Computer 34(11), November, 2001, pp. 44-51

[MEDIT] Meta-case official homepage, http://www.metacase.com/

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

[LV02] de Lara, J., Vangheluwe, H.: AToM^3 as a Meta-Case Environment, 4th International Conference on Enterprise Information Systems, 2002, pp 642 - 649

[ECLIPSE] The Eclipse Modeling Framework Framework, http://eclipse.org/

[GENGED] GenGed, tfs.cs.tu-berlin.de/ genged/

[EEHT] Erhig, K., Ermel, C., Hansgen, S., Taentzer, G.: Generation of Visual Editors as Eclipse Plug-Ins, http://www.tfs.cs.tu-berlin.de/ tigerprj/papers/

[Roz97] Rozenberg, G.: Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol.1 World Scientific, 1997.

[ML+06] Mezei, G., Lengyel, L., Levendovszky, T., Charaf, H.: Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality, 10th International Conference on Intelligent Engineering Systems, 2006

[LPE06] Levendovszky, T., Prange, U., Ehrig, H., Termination Criteria for DPO Transformations with Injective Matches, Graph Transformation for Verification and Concurrency, 2006