Automatic generation of platform-specific transformation

Ákos Horváth, Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1521 Budapest, Magyar tudósok körútja 2., Hungary
ha442@hszk.bme.hu, varro@mit.bme.hu

Gergely Varró
Department of Computer Science and Information Theory
Budapest University of Technology and Economics
H-1521 Budapest, Magyar tudósok körútja 2., Hungary
gervarro@szit.bme.hu

Abstract:
The current paper presents a new approach using generic and meta-transformations for generating platform-specific transformer plugins from model transformation specifications defined by a combination of graph transformation and abstract state machine rules (as used within the VIATRA2 framework). The essence of the approach is to store transformation rules as ordinary models in the model space, which can be processed later by the meta-transformations, which generates the Java transformer plugin. These meta rules highly rely on generic patterns (i.e. patterns with type parameters), which provide high-level reuse of basic transformation elements. Graph algorithms used for search plan generation are integrated as abstract state machines, while the final code generation step is carried out by code templates. As a result, the porting of a transformer plugin to a new underlying platform can be accelerated significantly.

Key words: meta-transformation, generic transformation, code generation

1 Introduction

Nowadays, model-driven system development [3] (MDSD) is an emerging paradigm in software development. A main challenge for MDSD is accommodate to the accelerating changes of business and technology. Based on high-level model standards (such as the Unified Modeling Language – UML [12]), MDSD separates business and application logic from underlying platform technology. Platform-independent models (PIM) capture the core business functionality independently from the underlying implementation technology, which are incorporated later on in platform-specific models (PSM). The source code of the system under design is generated afterwards from such platform-specific models. The success of the MDSD highly depends on automated model transformations (MT), which generate PSMs from PIMs, and executable source code from PSMs.

In MDSD, models are frequently captured by a graph structure, and the transformations are specified as graph transformations. Informally, a graph transformation (GT [11,7]) rule performs local manipulation on graph models by finding a matching of the pattern prescribed by its left-hand side (LHS) graph in the model, and changing it according to the right-hand side (RHS) graph.

The main objective of the VIATRA2 (VIsual Automated model TRAnsformations) framework developed at the Department of Measurement and Information Systems at Budapest University of Technology and Economics is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains. Since September 2005, VIATRA2 is part of the Eclipse Generative Modeling Tools subproject.

Advanced model transformation tools frequently aim at separating the design of a transformation from its execution by using high-level model transformation rules in design time and deriving executable platform-specific transformer plugins from these high level models. The role of design-time transformation frameworks (also called as platform independent transformers PIT) is to ease the development of model transformations, while the role of compiled standalone versions of a model transformation (Platform (language) specific transformers (PST)) in an underlying platform (e.g. Java) are more efficient from runtime performance aspects.

Code generators deriving the standalone transformers, are typically implemented in a standard programming language for specific model transformations, thus, it is difficult to reuse existing code generators to different platforms with conceptual similarities (e.g. from Java to Enterprise Java Beans) or to integrate them into other MT tools.

The current paper presents a new approach using generic and meta-transformations [14] for generating platform-specific transformer plugins from model transformation specifications defined by a combination of graph transformation and abstract state machine rules (as used within the VIATRA2 framework).

The essence of the approach is to store transformation rules as ordinary models in the model space, which can be processed later by the meta-transformations, which generates the standalone Java transformer plugin. These meta rules highly rely on generic patterns (i.e. patterns with type parameters), which provide high-level reuse of basic transformation elements. Graph algorithms used for search plan generation are integrated as abstract state machines, while the final code generation step is carried out by code templates.

As a result, the porting of a transformer plugin to a new underlying platform can be accelerated significantly.

2 Overview of the approach

The proposed workflow of the meta-transformation for PST generation is summarized in Fig. 1.

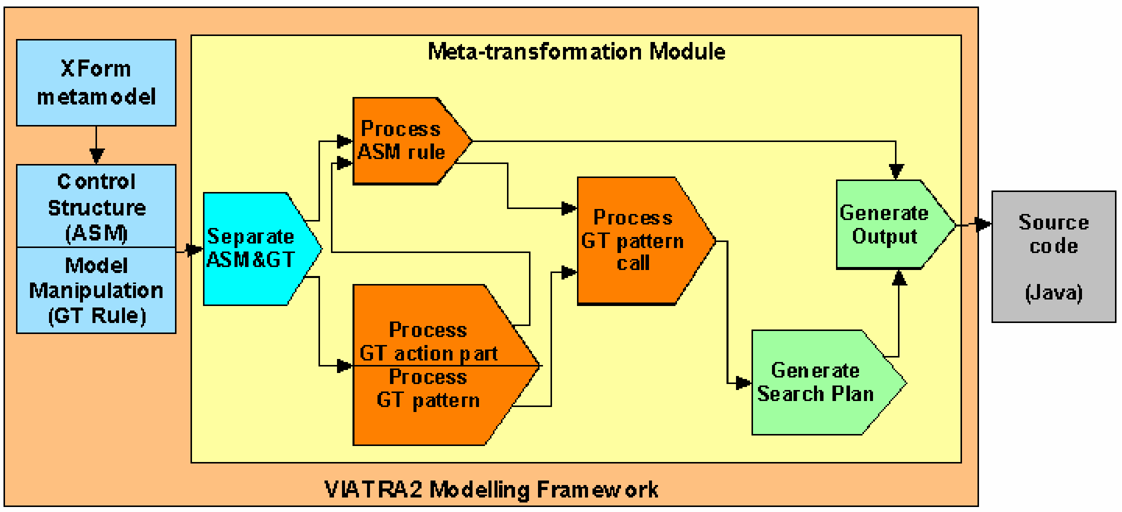


Fig. 1. Overview of the meta-transformation based generation

In VIATRA2, transformations can be defined by the combination of graph transformation (GT [7]) and abstract state machines (ASM [4]). The Transformation (XForm) metamodel (to be discussed in details in Sec. 3.2) consists of an ASM part for control structures and a graph transformation part for
elementary model manipulation.

The steps of the plugin generator transformation are the following:

- As ASM and GT rules are processed differently, we separate them in the first step. Since ASM rules are (semantically) very close to traditional high-level programming language constructs, their handling is not discussed.
- GT rules are processed in two substeps. The LHS of the rule should be handled as a GT pattern, while the action part described by the difference of RHS and LHS (and potentially additional ASM rules).
- For each pattern call initiating a graph pattern matching process, different search graphs are generated. (See [15] for a detailed discussion of search graph generation.)
- An optimized search plan (i.e. the traversal order of pattern nodes) is generated for every search graph in order to sequence the matching of the GT pattern.
- Finally, Java output is generated by code templates. For every different implementation platform only these code templates have to be replaced.

Note that the presented transformer plugin generation approach is implemented in the VIATRA2 framework, which improves extensibility and portability. In the rest of the paper, we first provide a brief overview of the models and transformations used in VIATRA2 (in Sec. 3). Then, the main part of the paper discusses (in Sec. 4) the meta-transformation developed for the PST generation and focuses on the graph pattern matching phase, as it is the most critical step for the performance of graph transformation. Finally, Sec. 5 concludes the paper.

3 Models and Transformations in VIATRA2

3.1 The VPM Metamodeling Language

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e. abstract syntax) of modeling languages. Metamodels are represented in a metamodeling language, which is another modeling language for capturing metamodels.

The VPM (Visual Precise Metamodeling) [13], which is the metamodel language of VIATRA2, consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). Entities represent basic concepts of a (modeling) domain, while relations represent the relationships between other model elements. Model elements are arranged into a strict containment hierarchy, which constitute the VPM model space. Within a container entity, each model element has a unique local name, but each model element also has a globally unique identifier, which is called a fully qualified name (FQN).

There are two special relationships between model elements: the **supertypeOf** (inheritance, generalization) relation represents binary superclass-subclass relationships (like the UML generalization concept), while the **instanceOf** relation represents type-instance relationships (between meta-levels). By using explicit **instanceOf** relationship, metamodels and models can be stored in the same model space in a compact way.

3.2 Transformation Language

Transformation descriptions in VIATRA2 consist of the combination of three paradigms: (i) graph patterns, (ii) graph transformation (GT [7]) rules and (iii) abstract state machine (ASM [4]).

Graph patterns

Graph patterns (referred as GT patterns) are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. A model (i.e. part of the model space) can satisfy a graph pattern, if the pattern can be matched to a subgraph of the model (by graph pattern matching).

An example GT pattern is depicted in Fig. 2. The GT pattern of Fig. 2 is fulfilled if there exists a class *CS* that has an attribute *A* and a parent class *CP*.

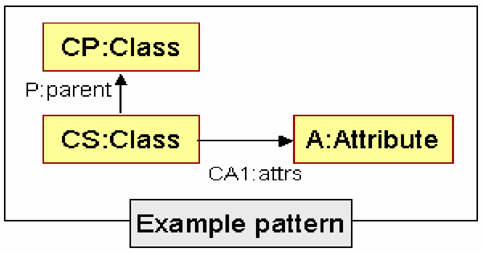


Fig. 2. Example GT pattern

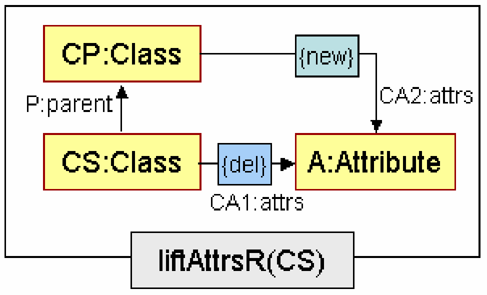
Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules, which heavily rely on graph patterns for defining the application criteria of transformation steps. The application of a GT rule on a given model replaces an image of its left-hand side (LHS) pattern with an image of its right-hand side (RHS) pattern (following the single pushout approach [8]).

The meta-model used for the graph transformation rules in VIATRA2 framework extends the core formalism by: (i) negative conditions can be embedded into each other in an arbitrary depth, (ii) supports the use of ASM rules in the action part of a GT rule, and (iii) supports the notation of standalone GT patterns.

The sample graph transformation rule in Fig. 3 defines a refactoring step, which moves an attribute from the child to the parent class. This means that if the child class has an attribute, it will be moved to its parent.

```
gtrule liftAttrs(in CS) =
    {
         condition pattern cond (CS)
=
         {
       Class(CP);
       Class.attrs(P, CP, CS);
       Class(CS);
       Attribute(A);
       new class.attrs(CA1, CP, A);
       del class.attrs(CA2, CS, A);
        }
}
```



(a) code view **(b) graph view**

Fig. 3. The GT rule liftAttrs

The rule contains a simple pattern (marked with keyword **condition**), that jointly defines the left hand side (LHS) of the graph transformation rule, and the actions to be carried out. Pattern elements marked with keyword new are created after a matching for the LHS is found (and therefore, they do not participate in the pattern matching), and elements marked with keyword del are deleted after pattern matching.

Control Structure

To control the execution order and the mode of graph transformation, abstract state machines [4] are used. ASMs provide complex model transformations with all the necessary control structures including the sequencing operator (**seq**), ASM rule invocation (**call**), variable declarations and updates (**let** and **update** constructs), **if-then-else** structures, non-deterministically selected (**random**) and executed rules (**choose**), iterative execution (applying a rule as long as possible **iterate**), and the deterministic parallel rule application at all possible matchings (locations) satisfying a condition (**forall**).

4 Generation of PST with Meta-transformations

To give an overview how the automatic PST generation process can be implemented over model transformations, three conceptually critical fragments are discussed in this section. The first example (in Sec. 4.1) shows how the type of the elements in a GT pattern is determined by a combination of GT patterns and ASM rules (using explicit instanceOf relations). The second example (in Sec. 4.2) gives an overview how the algorithms of the search plan generation are implemented in the framework. While the third (in Sec. 4.3) shows how the Java representation of relation (association) traversal is generated by a code template.

4.1 Processing the pattern elements of the graph transformation

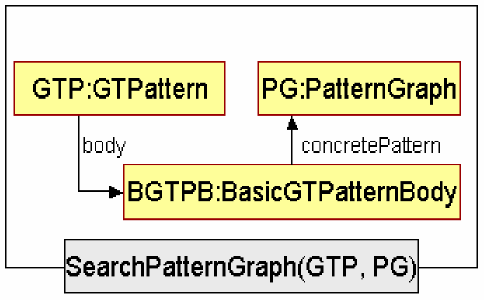
Our approach is using generic model transformations on the graph pattern rules presented as models in the VIATRA2 modelspace. Generic patterns in VIATRA2 use explicit **instanceOf** relations, which denote type variables.

This approach of the PST generation consists of two GT patterns *SearchPatternGraph, directType* and they are called from an ASM rule *processGTPattern.*

The meta-pattern SearchPatternGraph

The pattern **SearchPatternGraph** of Fig. 4 denotes that the PG is the pattern graph of the GT pattern GTP. In the transformation model, the *PatternGraph* is connected to the *GTPattern* through the *BasicGTPatternBody* (BGTPB) entity, along a body and a *concretePattern* relation.

```
//PG is the pattern graph
of GTP(GTPattern)
pattern aPatternGraph(GTP,PG) =
{
'GTPattern'(GTP);
'GTPattern'.'BasicGTPatternBody'
(BGTPB);
'GTPattern'.'BasicGTPatternBody'
.'PatternGraph'(PG);
'GTPattern'.body(Bo,GTP,BGTPB);
//relations
'GTPattern'.'BasicGTPatternBody'
.concretePattern(Con,BGTPB,PG)
}
```



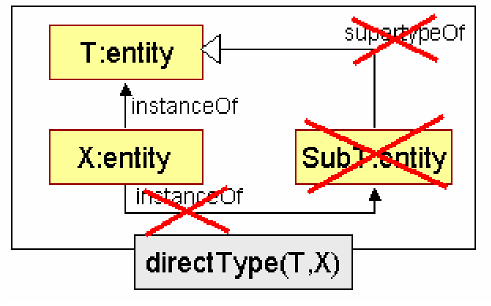
(b) graph view

Graph GT pattern

The generic pattern directType

The pattern *directType* (depicted in Fig. 5) is used to return the direct type of the input parameter X. The outer (positive) pattern matches the metamodel entity, which represents the type of X by the explicit **instanceOf** relation. The inner (negative) pattern can be satisfied if the input entity T has a subType, which is connected to X by an **instanceOf** relation. In this case the execution of the whole rule is violated.

```
pattern directType(X, T) =
{//X is an entity of Type T
      entity(X);
      entity(T);
      instanceOf(X,T);
      //T is not a type of X's
      //supertypes
      neg pattern hasSubType(T,X) =
      {
         entity(T);
         entity(SubT);
         supertypeOf(T, SubT);
         instanceOf(X,SubT);
      }
```



(a) code view **(b) graph view**

Fig. 5. The directType GT pattern

This generic pattern can handle several situations where essentially the same rule pattern should be applied on objects of different types. The type variables used in the pattern are instantiated by the **instanceOf** relation as concrete entities/relations from the metamodel (similarly to ordinary pattern variables).

The ASM rule processGTPattern

The ASM rule *processGTPattern* determines the direct type of the elements in the graph pattern PG. Type entities must be under the input parameter *Metamodel* in the containment hierarchy, while PG is the pattern graph of the input parameter GT pattern *InGTPattern*. The steps of the rule are the following:

(i) The *choose* selects the pattern graph of the GT pattern *InGTPattern* with the GT pattern *SearchPatternGraph* and puts it into the variable PG.

(ii) The *forall* enumerates all the combinations of the elements given in the scope one by one and tries to match the *directType* GT pattern. If a part of the model satisfies the pattern then its values are stored in variables *X* and *T*.

(iii) The ASM rule *processEntityBuildSG* is called with parameters PG, *X* and *T* in order to add this new element to the search graph of the GT pattern *PG*.

The VIATRA2 transformation rule is as follows:

```
//GTPatternHolder holds the pattern, and MetaModel is the //metamodel of the
//entities used in the GTPattern(s)
rule processGTPattern(in InGTPattern, in MetaModel) = seq
{
//selects the GTPatternGraph below the input GTPattern
choose PG with find aPatternGraph(InGTPattern,PG) do
//selects the the type(T) in the Metamodel of the entity X
      forall T below MetaModel, X in PG with
      find directType(X, T) do

      //processes the entity further and adds to the search graph
            call processEntityBuildSG(PG,X,T);
}
```

4.2 Search plan evaluation

As the most critical step for the performance of a graph rewriting framework is the graph pattern matching phase, our approach uses local search algorithms for evaluating the traversal order of the pattern matching. A weighted search graph is a directed graph with numeric weights on its edges, having a starting node connected to each other node with an edge. A search tree is a spanning tree of the weighted search graph. As the starting node has no incoming edges, all other nodes should be reachable on a directed path from the starting node. A search plan is one possible traversal of a search tree. A traversal defines a sequence in which edges are traversed.

The Java code representation of the optimized traversal order is also generated by model transformation, which consist of three phases:

(i) In the first phase, a weighted search graph is generated from the input GT pattern also taking into account all constraints on VPM entities of the pattern.

(ii) By using Chu-Liu and Edmonds algorithm [5,6] combined with a simple greedy algorithm, a low cost search plan is calculated.

(iii) Finally, Java code is generated based on the search plan (discussed later in Sec. 4.3).

As abstract state machines are widely used to formalize algorithms [10], is straightforward to implement them in VIATRA2. The following example demonstrates this on the well known greedy algorithm used in the search plan evaluation to select a low cost search plan from a search tree.

Simple greedy algorithm.

Initially, the list *P* consists only the starting node. The algorithm simply selects the smallest edge that goes out from the search graph nodes that are already in *P*, and adds the target of the selected edge as the last element of *P*.

The **iterate choose** construct selects the smallest edge that leading out of *P*, by using the ASM function *nodes* and *values* to store the edge with the smallest weight. Then the second **choose** selects the target node of the edge and adds it to *P* by setting the value of the node to *P*. The recursion terminates when the counter of nodes in *P* reaches the number of the nodes in the search graph (stored in the ASM function values).

```
//SG is the spanning tree of the search graph
rule sPlan(in SG) = seq {
  update values("Min") = "infinite"; //init values
  update nodes("MinEdge") = "-1";
  //selecting the lowest weighted edge
iterate choose No below SG, NextNode below SG, Edge below SG, Owe
    below SG withfind(searchPlan(No,NextNode,Edge,Owe)) do
      //checking the edge values, smaller then Min and not in P
      if((value(Owe) < values("Min") && value(NextNode)!="P") seq
      {update values("Min") = value(Owe);
      update nodes("MinEdge") = Edge; //weights are updated
      };
  // update the value of the element by P
  choose No below SG, NextNode below SG, Owe below SG
```

```
   with find(searchPlan(No,NextNode,nodes("MinEdge"))) do seq
      {setValue(NextNode, "P"); //it is now part of the 'list' P
      update values("searchPlan") = values("searchPlan")+1;
      }
  if(values("searchPlan") != values("nodeMaxNumber"))
  call sPlan(SG); //recurvise call
}
```

4.3 Source code generation

In this section, we propose a source code generation technique for model transformer plugins in Java based on VIATRA2 code templates. The template concept is similar to the one introduced in the Apache Velocity [1] language, but uses the formal ASM and GT paradigms as its control language whose constructs can be referred by the #() notation.

As an example, we use the template rule *printTraversalArb*, which generates the Java equivalent of a simple traversal of a relation with arbitrary multiplicity. In case of arbitrary multiplicity in the traversed direction (one-to-many or many-to-many), an **iterator** is generated to investigate all possible continuations.

The input of the template is the source (*Source*) and target (*Target*) entities of the relation, the type (*Type*) of the target element, the name of the relation (*Relation*) and the next (*Next*) element in the traversal order. The ASM function *name* returns the name of the model element. The steps of the traversal
order are processed recursively by calling the ASM rule *processNextStep* in order to generate the Java equivalents of internal code blocks.

```
//code generation the traversal of a relation with arbitrary
multiplicity
template printTraversalArb(in Target, in Source, in Relation,in
Type, in Next)= {
Iterator iter_#(name(Target))=
#(name(Source)).get#(name(Relation))().iterator();
      while(iter_#(name(Target)).hasNext()){
      try{
            I#(name(Type)) #(name(Target)) =
                  (I#(name(Type))) iter_#(name(Target)).next();

            //call recursively the next step in the order of
            //traversal
            #(call processNextStep(Next);)
      } catch (ClassCastException e) {} }
}
```

5 Conclusion

In the current paper, we proposed to use generic and meta-transformations for generating platform-specific transformer plugins from transformation specifications given by the combination of graph transformation rules and abstract state machines in the VIATRA2 framework.

The main advantage of our approach is reusability: only final code generation templates need to be altered when porting plugins to other object oriented languages. Up to now, we have a complete implementation for Java, but we plan to port the plugin transformers to other underlying platforms (e.g, Eclipse Model Framework, EMF) and to perform numeric measurements on the transformers.

Experimental evaluation of the generated transformer plugins was carried out in [2] using Enterprise Java Beans 3.0 [9] as the underlying plugin technology.

The generated transformer plugins were able to handle persistent models stored in relational databases with several million graph objects. A next challenge for the future is to integrate transformer plugins to the VIATRA2 framework itself. After successful integration, an optimized compiled version of native Java transformations can be executed instead of the interpreted version.

References

[1] Apache, Velocity homepage, http://jakarta.apache.org/velocity/index.html.

[2] Balogh, A., G. Varró, D. Varró and A. Pataricza, Compiling model transformations to EJB3-specific transformer plugins, in: ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006) (2006), pp. 1288–1295.

[3] Bettin, J., Ensuring structural constraints in graph-based models with type inheritance, in: M. Cerioli, editor, Proc. 8th Int. Conf on Fundamental Approaches to Software Engineering (FASE 2005), LNCS 3442 (2005), pp. 64–79.

[4] Börger, E. and R. Stark, “Abstract State Machines. A method for High-Level System Design and Analysis,” Springer-Verlag, 2003.

[5] Chu, Y. J. and T. H. Liu, On the shortest arborescence of a directed graph, Science Sinica 14 (1965), pp. 1396–1400.

[6] Edmonds, J., Optimum branchings, Journal Research of the National Bureau of Standards (1967), pp. 233–240.

[7] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, “Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools,” World Scientific, 1999.

[8] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, “In [11],” World Scientific, 1997 pp. 247–312.

[9] Enterprise Java Beans 3.0, Sun Microsystems, http://java.sun.com/products/ejb/docs.html.

[10] Gurevich, Y., The sequential ASM thesis, Bulletin of the European Association for Theoretical Computer Science 67 (1999), pp. 93–124.

[11] Rozenberg, G., editor, “Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations,” World Scientific, 1997.

[12] Rumbaugh, J., I. Jacobson and G. Booch, “The Unified Modeling Language Reference Manual,” Addison-Wesley, 1999.

[13] Varró, D. and A. Pataricza, VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML, Journal of Software and Systems Modeling 2 (2003), pp. 187–210.

[14] Varró, D. and A. Pataricza, Generic and meta-transformations for model transformation engineering, in: T. Baar, A. Strohmeier, A. Moreira and S. Mellor, editors, Proc. UML 2004: 7th International Conference on the Unified Modeling Language, LNCS 3273 (2004), pp. 290–304.

[15] Varró, G., D. Varró and K. Friedl, Adaptive graph pattern matching for model transformations using model-sensitive search plans, in: G. Karsai and G. Taentzer, editors, GraMot 2005, International Workshop on Graph and

Model Transformations, ENTCS vol. 42, pp 191--205