Model Transformer Plugin Generation

Ákos Horváth

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Magyar tudósok krt 2, H-1521 Budapest, Hungary
ha442@hszk.bme.hu

Endre Borbély

Institute of Communication Engineering
Kandó Kálmán Faculty of Electrical Engineering
Budapest Tech
Tavaszmező utca 17, H-1084 Budapest, Hungary
borbely.endre@kvk.bmf.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 platform-specific (Java,C++, C#, etc.) 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. As a result, the porting of a transformer plugin to a new underlying platform can be accelerated significantly.

Keywords: meta-transformation, platform-specific transformer, generic transformation, code generation

1 Introduction

Nowadays, the immense role of model transformation (MT) concepts and tools is unquestionable for the success of the model-driven system development (MDSD [3]) in order to capture the integration and evolution of models. In MDSD, models are frequently captured by a graph structure, and the transformations are specified as graph transformations. Informally, a graph transformation (GT [13,6]) 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.

Advanced model transformation tools usually separate the design (validation, maintenance) of a transformation from its execution by providing both tool dependent interpreter and standalone compiled plugins. Interpreters (also called as platform-independent transformers (PIT) in [4, 17]) ease the development (and testing, debugging and validation) of model transformations within a single transformation framework without relying on a highly optimised target transformation technology. Platform (language) specific transformers (PST) are compiled standalone versions of a model transformation in an underlying platform (e.g. Java) which is optimized for efficient performance. This transformer plugin is generated in a complex model transformation and/or code generation step.

However, the implementation of these generators for platform-specific transformers are typically use standard programming languages. As a consequence, it is difficult to port existing plugin generators to different platforms with conceptual similarities (e.g. from Java to C#).

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). Our code generator for Java transformer plugins highly builds on the fact that transformation rules are stored as ordinary models which can be processed later by the meta-transformation 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. 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.

In Viatra2, transformations can be defined by the combination of *graph transformation* (GT [6]) and *abstract state machines* (ASM [5]). The Xform metamodel (to be discussed in details in Sec. 3.2) symbolizes the transformation metamodel, which consists of an ASM part for control structures and a graph transformation part for model manipulation.

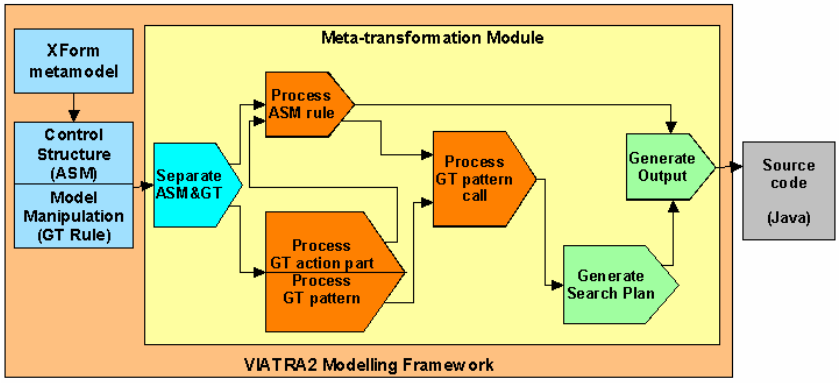


Figure 1

Overview of the meta-transformation based generation

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.
- ASM rules are processed as an ordinary programming language
- 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 potencially additional ASM rules).
- For each pattern call initiating a graph pattern matching process, different search graphs are generated. (See [18] for a detailed discussion of search graph generation)
- An optimized search plan (i.e. traversal order of the 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 all parts of the PST generation are implemented in the Viatra2 framework which improves extensibility and portability to other underlying implementation platform. 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. A comparison with related work is given in Sec. 5, and Sec. 6 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 metamodeling language, which is another modeling language for capturing metamodels.

The VPM (Visual Precise Metamodeling) [16] 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 constitutes 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. A VPM sample metamodel of the representation of GT rules is depicted in Fig. 4.

3.2 Transformation Language

Transformation descriptions in Viatra2 consist the combination of three paradigms: (i) graph patterns, (ii) graph transformation (GT [6]) rules and (iii) abstract state machine (ASM [5]). The transformation language implementing all these concepts is the Viatra Textual Command Language (VTCL).

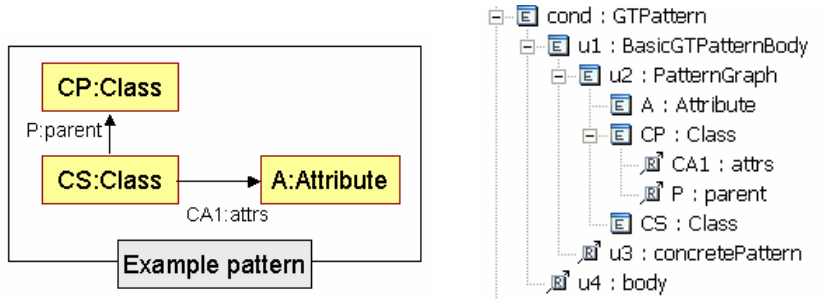
Graph patterns

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 graph representation of the pattern is depicted in Fig. 2(a), while Fig. 2(b) shows the Viatra2 model space representation, which is discussed later in Sec. 3.3.

The GT pattern of Fig. 2(a) is fulfilled if there exists a class CS which has

an attribute A and a parent class CP.



(a) graph view (b) tree view

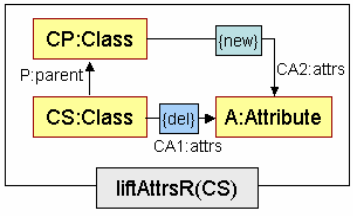
Figure 2
GTPattern example

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 [7]).

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);
        del class.attrs(CA1, CP, A);
        new class.attrs(CA2, CS, A);
       }
}
```



(a) code view (b) graph view

Figure 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 has been 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 mode of graph transformation, abstract state machines [5] 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**).

3.3 Graph Transformation Metamodel in Viatra2

An extract of the GT rule metmodel is depicted in Fig. 4. The whole GT rule metamodel contains 33 VPM model elements.

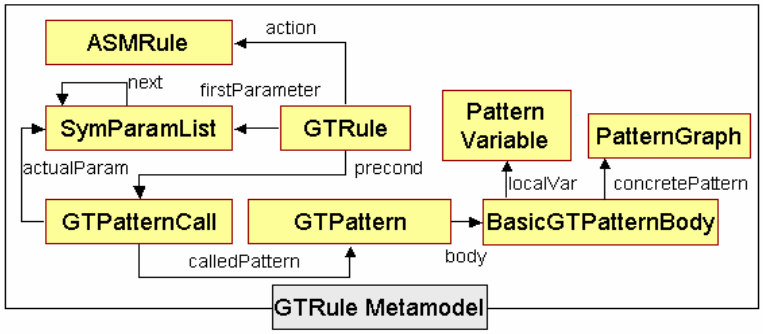


Figure 4
GTRule metamodel

All GT rules have a compulsory *GT pattern call* which represents the context (e.g type, mode, variables) and optionally an action part which stores the ASM rules called by the GT rule. Each GT pattern has a *BasicGTPatternBody* element, which stores the pattern graph *PatternGraph* and the in/out parameters *Pattern Variable*. Between GT rules, GT pattern calls, and GT patterns information is shared with parameter passing using the *SymParamList* and the *Pattern Variable* elements.

Fig. 2(b) shows an instance of the GT metamodel, the entities below the *PatternGraph* are instances of the domain metamodel and they represent the concrete pattern graph, while the rest of the representation conforms to the GT rule metamodel.

While being conceptually close to it, the metamodel used for the graph transformation rules in Viatra2 framework has a few difference with the public standard GT metamodel proposal GTXL (See in [15]): (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.

4 Generation of PST with Meta-transformations

Due to space restrictions, only two but conceptually critical fragments of the automatic PST generation process are discussed. The first example (in Sec. 4.1) shows how the type of the elements in a GT pattern graph are determined by a combination of GT patterns and ASM rules (using explicit **instanceOf** relations). The second example (in Sec. 4.2) shows how the Java representation of relation (association) traversal is generated by a code template.

The remaining parts of the transformation has similar construct with the two examples, namely the necessary information of the model is selected by GT patterns, processed or transformed by GT and ASM rules, and the output source code is generated by templates.

4.1 Processing the Graph Transformation Pattern Elements

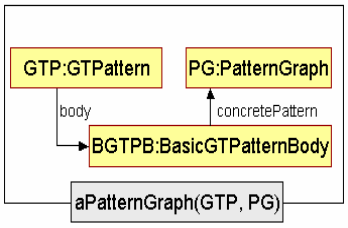
This part of the PST generation consists two GT patterns *aPatternGraph*, *directType* and called from an ASM rule *processGTPattern*.

The meta-pattern aPatternGraph

The pattern *aPatternGraph* of Fig. 5 denotes that the PG is the pattern graph of the graph transformation pattern GTP. According to the metamodel (see Fig. 4) of the GT rule, the *PatternGraph* is connected to the *GTPattern* through the *BasicGTPatternBody* (BGTPB) entity, along a body and concretePattern relations.

```
//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)
}
```

(a) code view



(b) graph view

Figure 5
The aPatternGraph GT Pattern

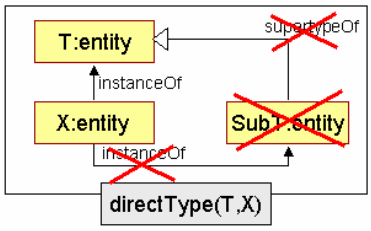
The generic pattern directType

The pattern *directType* (depicted in Fig. 6) 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.

Generic patterns in Viatra2 use explicit **instanceOf** relations, which denote type variables (such as T). These type variables are instantiated as concrete entities/relations from the metamodel (similarly to ordinary pattern variables).

```
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

Figure 6
The directType GT Pattern

The ASM rule processGTPattern

The ASM rule *processGTPatterns* determines the direct type of the elements in the graph pattern PG. Type entities must be under the input parameter Metamodel, 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 *aPatternGraph* and puts it into the variable PG. (ii) The **forall** iterates over all the combination 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 search graph of the GT pattern PG.

The VTCL source code of the 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 Output Generation

While Sec. 4.1 demonstrated the meta-transformation based model processing, this section focuses on the code template based output generation. The pattern 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 to using the #() notation.

The template rule *printTraversalArb* 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) {} }
}
```

To demonstrate how easily the plugin generator can be extended to generate other OO language specific transformers, the following example shows the C# code template for the arbitrary multiplicity relation traversal:

```
template printTraversalArb(in Target, in Source, in Relation,in
Type, in Next) = {
ICollection coll__#(name(Target))=
#(name(Source)).get#(name(Relation))();

foreach( DictionaryEntry item_#(name(Target)) in coll )
        {try
                {I#(name(Type)) #(name(Target)) =
                        (I#(name(Type))) item_#(name(Target)).Value;
                #(call processNextStep(Next);)
        } catch(Exception e) {} }
}
```

The Java and C# source code of the GT pattern printTraversalArb are partially generated by these code templates and they are presented in App. A and App. B, respectively.

5 Related Work

While there is already a large set of model transformation tools available using graph transformation languages, here we give a brief overview on tools providing both an interpreter and a compiled transformer.

Fujaba [12] compiles visual specifications of transformations [8] into executable Java code. Our approach also shows similarity with the new templatebased code generation plugin [11] of Fujaba. The main difference is that in our approach, code templates are a part of the transformation language allowing to design and manage the whole plugin generation within the Viatra2 framework.

The pattern matching engine of compiled GReAT [19] generates optimised executable C++ code. In GReAT, GT rules also have a corresponding metamodel, but since the tool builds upon the OMG's MOF metamodeling approach, generic and meta-transformations are not yet supported, and the transformation themselves are not part of the model space. As for the output generation step, the GReAT source code generator is implemented in native C++, which directly accesses the model manipulation and traversal API.

MOLA [10] is a graphical procedural transformation language also with a clear separation of transformation design and execution time. Its main distinguishing

features are advanced graphical pattern definitions and control structures defined using an off-the-shelf UML tool.

PROGRES [14] supports both interpreted and compiled execution (generating C code) of programmed graph transformation systems. Only PROGRES allows the use of type parameters in rules to support higher-order transformations, but meta-transformations are not yet supported, and the model of the transformations are not part of the model space.

Finally, the concepts of meta-transformations defined by graph transformation rules was first discussed in [9].

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 (and the related EJB3 platform in [2]), but a C# solution is an ongoing project.

The entire code generator for Java transformer plugins consists of about 150 (ASM and GT) transformation rules, and it has been implemented within the (interpreted) Viatra2 framework. Experimental comparison of the generated transformer plugins (for both Java and EJB-based solutions) based on benchmark measurements are discussed in [2].

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/

[2] Balogh, A., G. Varró, D. Varró and A. Pataricza, Generation of platform-specific model transformation plugins for EJB 3.0, accepted to SAC 2006, Model Transformation Track

[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] B´ezivin, J., N. Farcet, J.-M. Jézéquel, B. Langlois and D. Pollet, Reflective model driven engineering, in: P. Stevens, J. Whittle and G. Booch, editors, Proc. UML 2003: 6th International Conference on the Unified Modeling Language, LNCS 2863 (2003), pp. 175-189

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

[6] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, 2: Applications, Languages and Tools in: "Handbook of Graph Grammars and Computing by Graph Transformations", World Scientific, 1999

[7] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach "In [13]," World Scientific, 1997 pp. 247-312

[8] Fischer, T., J. Niere, L. Torunski and A. Z¨undorf, Story diagrams: A new graph transformation language based on UML and Java, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, Proc. Theory and Application to Graph Transformations (TAGT'98), LNCS 1764 (2000)

[9] Hesse,W., Two-level graph grammars, in: V. Claus, H. Ehrig and G. Rozenberg, editors, International Workshop on Graph-Grammars and Their Application to Computer Science and Biology, October 30 - November 3, 1978, Lecture Notes in Computer Science 73 (1979), pp. 255-269

[10] Kalnins, A., J. Barzdins and E. Celms, Model transformation language MOLA, in: Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004), Linköping, Sweden, 2004, pp. 14-28

[11] L. Geiger, C. R. and C. Schneider, Template- and model based code generation for MDA-tools, in: Proc. of 3rd International Fujaba Days (IFD05), Carsten Reckord Heinz Nixdorf Institute, Paderborn, Germany, 2005

[12] Nickel, U., J. Niere and A. Z¨undorf, Tool demonstration: The FUJABA environment, in: The 22nd International Conference on Software Engineering (ICSE) (2000)

[13] Rozenberg, G., editor, Handbook of Graph Grammars and Computing by Graph Transformations: Foundations, World Scientific, 1997

[14] Schürr, A., Introduction to PROGRES, an attributed graph grammar based specification language, in: M. Nagl, editor, Graph–Theoretic Concepts in Computer Science, LNCS 411 (1990), pp. 151-165

[15] Taentzer, G., Towards common exchange formats for graphs and graph transformation systems, in: J. Padberg, editor, UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, ENTCS 44 (4), 2001

[16] 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

[17] 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

[18] 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, in press

[19] Vizhanyo, A., A. Agrawal and F. Shi, Towards generation of efficient transformations, in: G. Karsai and E. Visser, editors, Proc. of 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004), LNCS 3286 (2004), pp. 298-316

A Generated Java Source Code

```
public static HashMap liftAttrs(Object CSFIX) throws
RuleFailedException {
HashMap result = new HashMap();
boolean success = false;
try{ //FIX incoming variables
        iClass CS = (iClass) CSFIX;
        //many-to-one relation
        try{
                IClass CP = (IClass) CS.getParent();
                //one-to-many relation
                Iterator iter_A = CS.getAttrs().iterator();
                while(iter_A.hasNext()){
                try{
                        IAttribute A = (IAttribute) iter_A.next();
                        //**ACTION part of the Rule
                        CS.deleteAttrs(A);
                        CP.addAttrs(A);
                        success = true;
                } catch (ClassCastException e) {} }
        } catch (ClassCastException e) {}
} catch (ClassCastException e) {}
if(success)
        return result; //there are no output parameter
else
        throw new RuleFailedException();}
```

B Generated C# Source Code

```
public Hashtable liftAttrs(Object CSFIX)
{ Hashtable result = new Hashtable();
bool success = false;
try { //FIX incoming variables
        IClass CS = (IClass) CSFIX;
        //many-to-one relation
        try
                { IClass CP = (IClass) CS.getParent();
```

```
                //one-to-many relation
                ICollection coll = CS.getAttrs();
                foreach( DictionaryEntry item_A in coll )
                { try
                        {IAttribute A = (IAttribute) item_A.Value();
                        //**ACTION part of the Rule
                        CS.deleteAttrs(A);
                        CP.addAttrs(A);
                        success = true;
                } catch(Exception e) {} }
        } catch (Exception e) {}
} catch (Exception e) {}
if(success)
        return result;
else
        throw new RuleFailedException();}
```