Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?

Shane Sendall

Software Modeling and Verification Laboratory
Computer Science Department
University of Geneva, CH-1211 Geneva 4, Switzerland
Shane.Sendall@cui.unige.ch

For the Model Driven Architecture (MDA) vision to become a mainstream reality, high-level languages are needed to concisely express and execute model transformations. Both generative and graph transformation approaches for model transformation specification and execution have much to offer in this regard. If one were to combine techniques from both worlds, would one end up with a powerful and effective tool or a complicated and inconsistent mess? In this short paper, I briefly highlight how I have combined some of these techniques in a model transformation language called Genermorphous (Gmorph for short), which attempts to package these two powerful techniques into a single coherent language. My goal with this short paper is to stimulate discussion in the workshop towards the discovery of optimal ways to use generative and graph rewriting techniques together for MDA-oriented transformation languages.

1. Introduction

The intention of the MDA initiative is to automate the generation of platform specific models (PSM for short) from platform independent models (PIM for short) and code from PSMs, and to ensure synchronization between models both in the same and different levels. These activities, together with other model engineering activities, such as, refactoring and pattern application, can be described in terms of model transformations. A model transformation takes one or more source models as input and produces one or more target models as output, following a set of rules.

One key element needed in the realization of the MDA vision is high-level languages that support the specification and execution of model transformations. Unsurprisingly, there are a number of challenges one needs to overcome in defining a language for model transformation [SK03]. On the one hand, it must provide for complete automation, and it must be expressive, unambiguous, and Turing complete for it to be generally applicable. On the other hand, it must balance ease-of-understanding, precision, concision and ease-of-modification.

2. Graph Transformation and Generative Programming

Graph transformation systems make use of graph rewriting techniques to manipulate graphs [Roz97]. A graph transformation is defined in terms of a set of production rules. A production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the source graph, it is replaced by the RHS graph. Graph transformation techniques have particular relevance to transformations performed on graphical models because graphical models are themselves graphs in one form or another. A further draw-card of graph transformation is the ability to use pattern matching in source element selection. Graph transformation systems use the LHS graph of each rule as a pattern to match, abstracting the mechanism of pattern matching from the rule specifier.

Generative Programming (GP) is an approach that allows one to automatically generate software from a generative domain model [CE00]. A generative domain model is a model of a system family that consists of a problem space, a solution space, and the configuration knowledge. The problem space defines the appropriate domain-specific concepts and features. The solution space defines the target model elements that can be generated and all possible variations. The configuration knowledge specifies illegal feature combinations, default settings, default dependencies, construction rules, and optimization rules. GP introduces generators as the mechanisms for producing the target. In general, a GP generator performs the following tasks:

- Checks the validity of the input specification and reports warnings and errors if necessary
- Completes the specification using default settings if necessary
- Performs optimizations
- Generates the target code

GP techniques have particular relevance to transformations that map one model to another different model, because the description of the mapping rules can be expressed concisely. A further draw-card of GP is the ability to make use of parameterization in the generation of models. Parameterized code generation is supported by such mechanisms as templates, boilerplates, frames, etc.

The underlying mechanism of graph transformation is replacement, which, in the context of MDA, makes it well suited to model refactoring, synchronization and refinement. In contrast, the underlying mechanism of GP is generation, which, in the context of MDA, makes it well suited to view generation, forward engineering and reverse engineering. These aspects of both techniques are complementary and provide good coverage of the requirements for model transformation in MDA. This supposition motivates an amalgamation of techniques, which is discussed in the next section with the presentation of the Genermorphous language (Gmoph for short).

3. The Gmorph Language

The Gmorph language [SMV03], currently under development by the author, is intended for the specification and execution of model transformations, where source and target models are instances of MOF-compliant metamodels. The language embodies a merger of techniques from both graph transformation and GP approaches. In particular, Gmorph specifications are largely graphical in nature and consist of production rules that define left-hand side and right-hand sides, following the declarative style of graph transformations. And, both the LHS and RHS descriptions offer parameterization and direct mapping, following the GP style.

Figure 1 shows a Gmorph transformation rule for mapping attributes of a UML class, part of the source model, to fields of an existing Java class, part of the target model. The transformation rule consists of a signature, shown at the top of Figure 1, the LHS of the rule, shown on the left side, and the RHS of the rule, shown on the right side. The rule was defined such that the LHS and RHS graphs conform to the metamodels of the source and target models, respectively. The metamodel used for the source model is defined by the UML 2.0 specification [Omg03], and the metamodel used for the target model is defined by Dedic and Matula in [DM03].

The signature of the rule defines the name of the rule: "UML Attributes to Java Fields", and the input and output parameters of the rule[1], which happen to be a Java class, J1. The LHS defines both a pattern to match and a guard condition that must be fulfilled for the rule to be allowed to fire. The graphical part of the rule consists of two object boxes with a composition relationship between them and a box that encircles the object AT1. The two associated objects define a pattern that is to be matched by the rule. In addition, one or more attributes of C1, matching AT1, are to be matched. This multi-match is defined by the box with the expression "Col: A, 1..*" in the top left-hand corner. This region signifies a collection of matches, which are referenced by the label A; this collection has a matching range of 1..*, which means that one or more matches are allowed.

The rule is allowed to fire only if a match can be found for the LHS pattern. This means that there should exist a class in the source model that has the same name as the supplied Java class and has at least one attribute. The first constraint is defined by the textual part of the LHS under the heading "Additional Selection Constraints". It is an OCL predicate that ensures equality between the names of C1 and J1. The second constraint is defined by the 1..* multiplicity of A. Having a lower bound of 1 implies that C1 must have at least one attribute matching AT1.

[1] It is possible to compose transformation rules in Gmorph; see [SMV03] for more details.

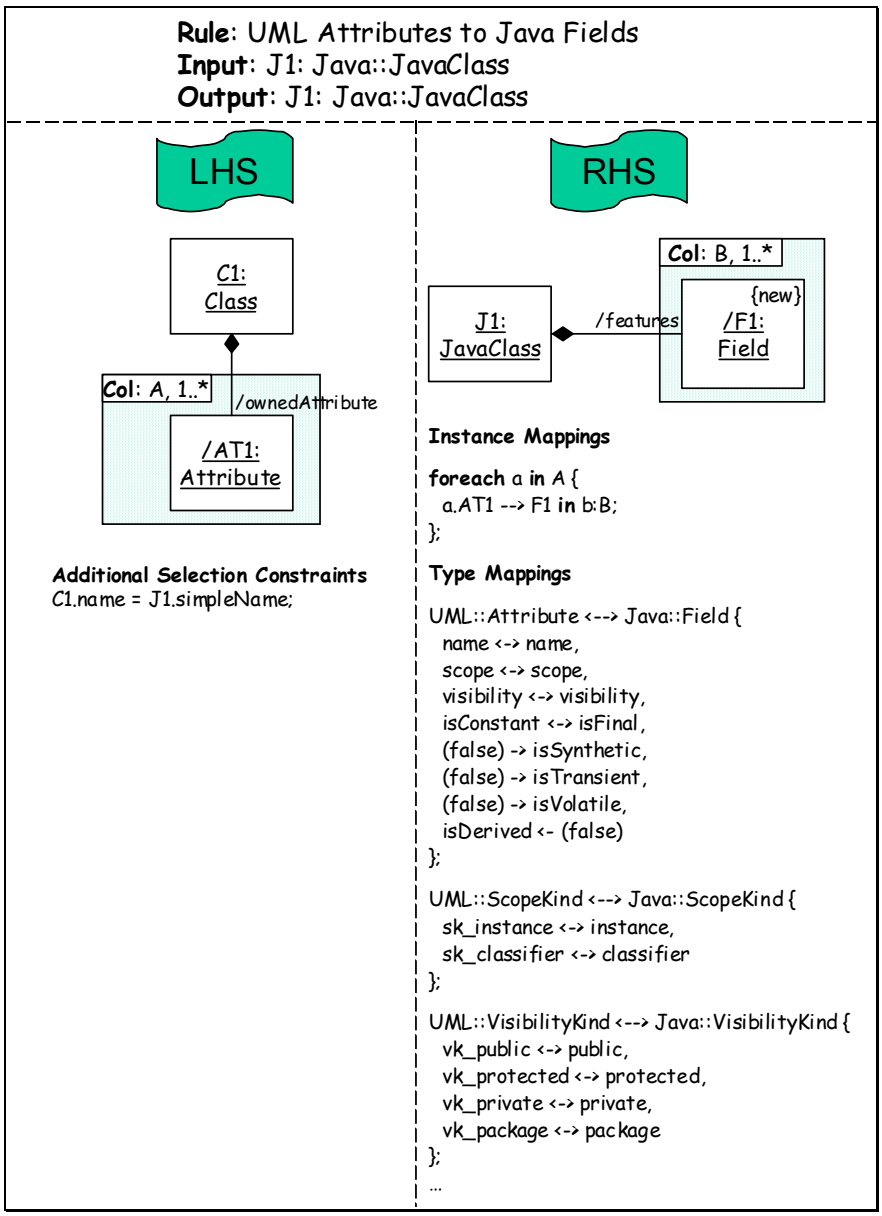

Figure 1: A Gmorph transformation rule, which maps UML attributes to Java fields.

If the LHS pattern is matched, then the RHS is applied to the target model. Unlike in Graph Transformation approaches, creation (and deletion for that matter) are explicitly stated. Thus, only the Field objects in the RHS depicted in Figure 1 are created (signaled by the "new" predefined identifier placed within curly brackets). The JavaClass J1 is part of the RHS so as to indicate how the new elements are integrated

into the already existing (target) model. The two text sections of the RHS, shown below the diagram, define the mappings between the elements in the source and target models. The instance mapping relates each AT1 element to a newly created F1 element. The type mappings relate the different properties of each element. In some cases, the mapping is bi-directional (denoted by the <-> operator), and in the other cases, it is only in a single direction (denoted by the <- and -> operators). Note that the direction of the arrow indicates in which direction the mapping can be performed. Most of the configuration information defined by the "Type Mappings" clause can be defined independently of the transformation, so as to be available for reuse by other rules.

In summary, the rule depicted in Figure 1, takes a Java class as input and finds the corresponding UML class; it matches the corresponding UML class with all its attributes, part of the source model, and generates a field in the supplied Java class, part of the target model, for each attribute matched.

Figure 2 shows an alternative form of the LHS pattern, shown in Figure 1. It uses the notation of the model directly, in contrast to the metamodel objects (which is independent of the models form). More details on this alternative view for a LHS pattern can be found in [Sen03].

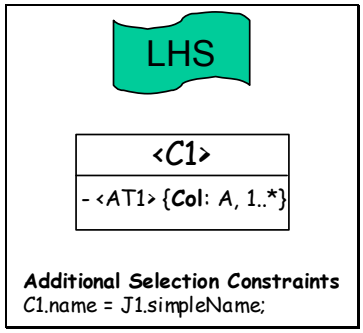

Figure 2: Alternative graphical view of the LHS of rule depicted in Figure 1

Reflecting upon the example, it is interesting to see the way the pattern matching and graph description techniques of Graph Transformation have been combined with the mapping and parameterization techniques of GP. The example also highlights how one can integrate model fragments, as defined by the RHS, into existing models.

5. Summary

One key element needed in the realization of the MDA vision is high-level languages that support the specification and execution of model transformations. In this short paper, I briefly highlighted how I have combined some of the techniques of Graph Transformation and Generative Programming in a model transformation language

called Genermorphous (Gmorph for short), which attempts to package these two powerful techniques into a single coherent language.

References

[AKS03] A. Agrawal, G. Karsai, and F. Shi; "A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations". International Journal on Software and Systems Modeling, (Submitted), 2003.

[CE00] K. Czarnecki and U. Eisenecker; "Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[DM03] S. Dedic and M. Matula; "Metamodel for the Java language". NetBeans.org http://java.netbeans.org/models/java/java-model.html

[KWB03] A. Kleppe, J. Warmer, and W. Bast; "MDA Explained: the Practice and Promise of Model-Driven Architecture". Addison-Wesley, 2003.

[Omg01] OMG Architecture Board ORMSC; "Model Driven Architecture (MDA)", 2001. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01

[Omg02] OMG TC; "MOF 2.0 Query/Views/Transformations RFP", 2002. http://cgi.omg.org/cgi-bin/doc?ad/02-04-10

[Omg03] OMG Unified Modeling Language Revision Task Force; "UML 2.0 Superstructure Final Adopted Specification". Version 2.0, April 2003. http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

[Roz97] G. Rozenberg (ed.); "Handbook of graph grammars and computing by graph transformation: Volume I Foundations". World Scientific Publishing, 1997.

[Sen03] S. Sendall; "Source Element Selection In Model Transformation". UML '03 Workshop entitled "Workshop in Software Model Engineering (WiSME)", 2003.

[SK03] S. Sendall and W. Kozaczynski; "Model Transformation – the Heart and Soul of Model-Driven Software Development". To appear in IEEE Software, Special Issue on Model Driven Software Development, September/October 2003.

[SMV03] Software Modeling and Verification Lab.; "Genermorphous home page". http://cui.unige.ch/smv/gmorph

[SPG+03] S. Sendall, G. Perrouin, N. Guelfi, and O. Biberstein; "Supporting Model-to-Model Transformations: The VMT Approach". Workshop on Model Driven Architecture: Foundations and Applications; Holland, 2003.

[SWZ97] A. Schürr, A. Winter and A. Zündorf; "The Progres Approach: Language and environment". In Chapter13 of G. Rozenberg, Handbook of graph grammars and computing by graph transformation: volume I foundations, World Scientific Publishing, 1997.

[WK98] J. Warmer and A. Kleppe; "The Object Constraint Language: Precise Modeling With UML". Addison-Wesley 1998.