TOWARDS MODEL TRANSFORMATION DESIGN PATTERNS

Jean Bézivin, Frédéric Jouault, Jean Paliès
ATLAS Group (INRIA & LINA, University of Nantes)
{bezivin | frederic.jouault | jean.palies}@gmail.com

Abstract The use of design patterns in software engineering has already been broadly discussed. This is a good way of capitalizing experience and best practice so as to increase reusability. What is true for Object Oriented Programming should also apply to Model Driven Engineering (MDE) [6]. There are many ways to design a metamodel or a model transformation. MDE would benefit from having a large collection of model transformation design patterns. This paper presents some initial work on the construction of a collection of MDE patterns. MDE patterns cover the design of metamodels, transformations, compositions and other operations on models.

Keywords: Model, Metamodel, Transformation, Design pattern.

1. Introduction

In this work, we focus on the use of design patterns in Model-Driven Engineering (MDE). Design patterns have been discussed, from Alexander's work [1] until now, passing by the *Gang of Four* [5]. In the first part of the paper we will come back on the concept of model transformation, then we discuss our needs for design patterns with MDE, and we provide two examples from our initial work.

2. Design patterns and MDE

Model transformation

Model transformation is the process of converting one model to another model of the same system says the Model Driven Architecture guide [7]. This means going from a source model to a target one. Both of them have to conform to their respective metamodel. Both of these metamodels conform to a metametamodel. The transformation itself is a model, described in a language, i.e. conforming to its metamodel. This transformation metamodel conforms to the metametamodel.

The transformation metamodel defines the transformation language, which may for example be declarative or imperative. We use the ATL [3] language in this paper. Our results could also apply to other QVT-like [9] language.

Design patterns and model transformations

ATL has been used for some time now and its different users are occasionally reporting recurrent situations that may be described by MDE patterns.

We have a quite large library of transformations in ATL from which we can draw some feedback. Among these transformations some have been redefined several times so that we learnt good practice. The following non exhaustive list describes some ATL programs, most of which can be found on the GMT project [4]:

- BibTeXML to Docbook: from the XML dialect for BibTeX to the Docbook standard
- Book to Publication
- Class to Relational: from object-oriented to relational tables
- Geometrical Transformations: a transformation that performs 3D geometric operations
- Java source to Table: extraction of a static method call graph from a Java program
- KM3 to DOT: from a textual description of a metamodel to its graphical representation
- Monitor to Semaphore: from Hoare's monitors to Dijkstra's semaphores
- UML to Amble: from a UML model of a distributed program to its implementation
- UML to Java
- UML Activity Diagram to MSProject: from a loop free activity diagram to a corresponding MSProject presentation
- UMLDI to SVG: from UML class diagrams to their visual representation
- XSLT to XQuery: from one XML transformation language to another

Patterns Examples

The two patterns examples given below are representative of the collection we are currently building. They were found when facing implementation problems in ATL that had already been met previously.

Transformation parameters

In many model transformations some attributes of the target metamodel, that are not related to the true idea of the transformation, are hard-coded. They could become parameters for this transformation as they have side effects (for example pretty printing) on the result. What does this kind of change mean to a transformation model?

Motivation

Transformations often have such parameters. They are in fact elements of the target metamodel that cannot be matched within the source metamodel, but that the transformation designer does want to set to a certain value.

Basically, they are hard-coded, for example in helpers (for ATL). If one wants to change their value, then one has to edit the transformation model. One may improve this by placing all these parameters helpers in the same part of the transformation description, with some proper documentation.

This is a recurrent problem in programming. Its solution is well known: parameters should be set without editing the whole source code. To do this within MDE, a parameter metamodel has to be created. But how?

- A solution could be to create one specific parameter metamodel for each transformation, or each technical space. This enables really advanced settings, as it is very specific.
- Another could be to create a generic parameter metamodel. It would be very simple, as it should be useable with any transformation. This brings less expression power.

Solution

The solution of a generic parameter is better in terms of software and model driven engineering, as it increases reuseability. The generic parameter metamodel could be presented as *Figure 1*.

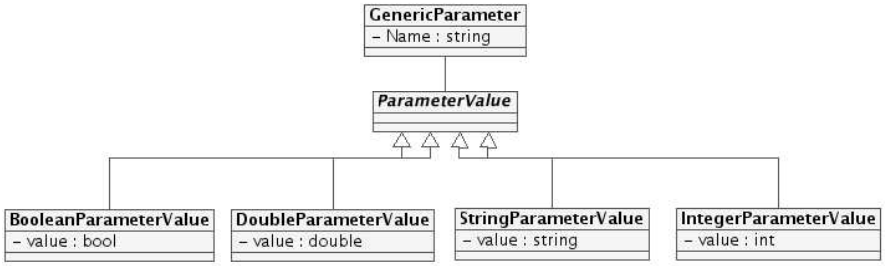

Figure 1. GenericParameter Metamodel

Consequences

Transformations have to use one more source model (and metamodel). Yet it is much more comfortable for the user. One does not have to know the whole transformation language to change the output options for example.

To adapt a transformation already using inline helpers to set the options is relatively easy. The helpers just have to fetch the results in the parameters model instead of having their value hard-coded.

The question of specifying what is an external parameter is left to the transformation designer. It depends on what the transformation is truly about. For example, in a conversion from one format to another, the pretty printing can be considered as an external concern. Most of the time these parameters are elements of the target metamodel for which no corresponding element can be found in the source metamodel. Then they have to be set manually.

This design pattern is for languages that do not have the feature of parameters given to a tranformation. However, everything should be a model. Were the parameters attributes of the transformation model, there would still be the problem of creating a proper parameter metamodel for these.
We experienced the need for this design pattern in many cases, for example when designing the KM3 to DOT transformation. In this example, a textual representation was to be transformed into a graphical one. The latter had different concerns from the first, for example dimensions. These were set as parameters.

This example, along with others, brought us to think on another design pattern: *Multiple matching patterns*.

Multiple matching patterns

Sometimes to perform a transformation there has to be at least one rule which matches not one but a pair or a collection of source elements. When does this happen? How can it be achieved in any case?

Motivation

First of all, some tranformation languages support that kind of matching method, whereas some others do not. ATL in its current version does not. Thus comes the question of how to do this in the current version of ATL, without any precomputed engine solution.

For example, one wants to transform a couple *Rectangle-Circle* into a Triangle. But one Rectangle may be linked to many Circles. Then, one has to match every couple.

Solution

For a language that allows to match multiple elements in one rule, there is a left-hand-side containing several patterns to be matched. The matching will then be done on all combinations that exist in the model(s). There may be a

guard set on the whole global pattern, specifying the relation that is to be met within the pairs/collections. In the extended left-hand-side, there may be:

- several input elements (at least one).
- one boolean expression that expresses the conditions that are to be met by the collection of input elements.

Using the example given above, in ATL, with the Shape metamodel to describe shapes:
rule RectangleAndCircle2Triangle {
from rect: Shape!Rectangle, circ: Shape!Circle (rect.circles->includes(circ))
to triang: Shape!Triangle(...
}

Consequences

If the transformation language does not support multiple matching patterns, then it can still be done using precomputation. In ATL, the above example with pairs of Circles and Rectangles would be:
rule RectangleAndCircle2Triangle {
from rect: Shape!Rectangle
to distinct triang: Shape!Triangle foreach(circle in rect.circles) (...
}

This design pattern illustrates the kind of problems that may be found while designing transformations. Even though this feature is an interesting one to have for a model transformation engine, it is still important to know how to manage without it.

3. Conclusions

The two examples provided in this paper come from a collection of MDE design patterns we are currently building. We have found 23 MDE design patterns for the moment. We are still expanding this collection when receiving new feedback from ATL users.

We are also discovering different categories of MDE design patterns. Some are related to metamodel structure, others to transformations description for instance. There are other possible categories, such as megamodels design patterns, or OCL design patterns. While finding new MDE design patterns, we are studying what makes them belong to a specific category, and how this could be captured in a MDE design pattern metamodel.

4. Acknowledgements

We would like to thank Marcos Didonet Del Fabro, Freddy Allilaire, David Touzet, Ivan Kurtev, Patrick Valduriez and all the members of the AMMA group for their numerous contributions to this document. This work has been

supported in part by the IST European project "ModelWare" (contract 517131) and by a grant from Microsoft Research (Cambridge).

References

[1] Alexander, C et al (1977). A Pattern Language: Towns, Buildings, Construction. *Oxford University Press*, New York.

[2] Bezivin, J. (Aug. 2001). From Object Composition to Model Transformation with the MDA. *proceedings of TOOLS'USA, Volume IEEE TOOLS-39*, Santa Barbara.

[3] Bezivin, J, Dupe, G, Jouault, F, Pitette, G and Rougui, E J (2003). First experiments with the ATL model transformation language: Transforming XSLT into XQuery. *OOPSLA 2003 Workshop.*, Anaheim, California..

[4] Generative Model Transformer. http://www.eclipse.org/gmt

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). Design patterns - Elements of Reusable Object-Oriented Software *Addison-Wesley.*

[6] Kent, S. (2002). Model Driven Engineering. *Proceedings of IFM2002, LNCS2235*, Springer.

[7] OMG (2003). MDA guide version 1.0.1. *OMG document* omg/2003-06-01.

[8] OMG (2002). Meta-Object Facility (MOF^{TM}), version 1.4 *OMG formal specification* formal/2002-04-03.

[9] OMG (2002). Query/View/Transformation Request For Propsal. *OMG Request For Proposal* ad/02-04-10.