

# Language Features for Re-Use and Maintainability of MDA Transformations

Michael Lawley, Keith Duddy, Anna Gerber, Kerry Raymond  
CRC for Enterprise Distributed Systems (DSTC)  
{lawley,dud,agerber,kerry}@dstc.edu.au

## Abstract

*DSTC's model transformation tool, Tefkat, is an implementation of a declarative transformation language proposed in response to the OMG's MOF 2.0 QVT RFP. The language contains several distinct features aimed specifically at enabling re-use and enhancing the maintainability of large transformation specifications. In this position paper we describe these features, how they contribute to the language and the definition of transformations, and the benefits when compared to other approaches.*

## 1 Introduction

DSTC has been researching model-based transformations as part of its current 7-year research programme, using a number of different approaches. Some of our experiments are described in [2]. These experiments and an examination of the experiments of others, have led us to develop a set of requirements, in addition those given in the OMG's MOF 2.0 QVT RFP [3], for a transformation language. A language satisfying these requirements will be suitable for describing transformations in a precise but readable manner; the kind of language required to describe the model to model mappings needed to realise the MDA vision.

This paper is structured as follows: we briefly describe our transformation language's structure and features, then we introduce a simple transformation problem and use it as a basis for discussion of how particular features of our language contribute to the ability to write re-usable and maintainable model transformations.

## 2 The Transformation Language

A transformation in our language [1] consists of the following major concepts: transformation rules, tracking relationships, and pattern definitions.

**Transformation rules** are used to describe the things that should exist in a target extent based on the things that are matched in a source extent. Transformation rules can

be extended, allowing for modular and incremental description of transformations. More powerfully, a transformation rule may also supersede another transformation rule. This allows for general case rules to be written, and then special cases dealt with via superseding rules. For example, one might write a naive transformation rule initially, then supersede it with a more sophisticated rule that can only be applied under certain circumstances. Superseding is not only ideal for rule optimization and rule parameterization, but also enhances reusability since general purpose rules can be tailored after-the-fact without having to modify them directly.

**Tracking relationships** are used to associate a target element with the source elements that lead to its creation. Since a tracking relationship is generally established by several separate rules, they allow other rules to match elements based on the tracking relationship independently of which rules were applied or how a target element was created. This allows one set of rules to define what constitutes a particular relationship, while another set depends only on the existence of the relationship without needing to know how it was defined. This kind of rule decoupling is essential for rule reuse via extending and superseding to be useful.

**Pattern definitions** are used to label common structures that may be repeated throughout a transformation. A pattern definition has a name, a set of parameter variables, a set of local variables, and a term. Parameter variables can also be thought of as formal by-reference parameters. Pattern definitions are used to name a query or pattern-match defined by the term. The result of applying a pattern definition via a pattern use is a collection of bindings for the pattern definition's parameter variables.

## 3 An Example

To set the scene, consider the problem of representing data described by an object-oriented class diagram in a relational database. Figures 1(a) and 1(b) show possible models of the concepts involved in these two domains.

This is an interesting transformation problem because the two models are familiar to many people, and they are

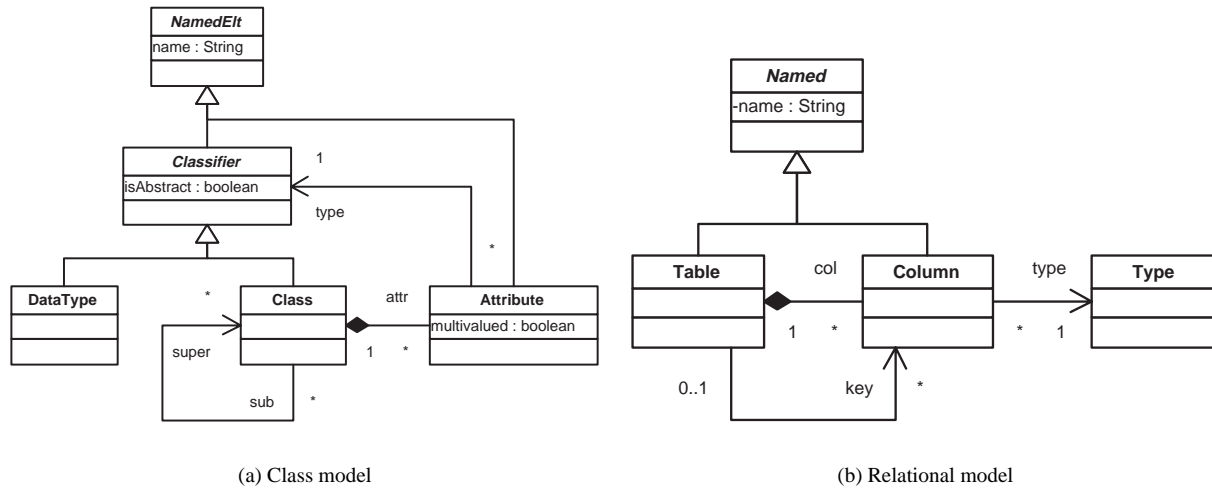


Figure 1. Simple source and target models

not so similar that a transformation is essentially one-to-one. Additionally, there are simple alternative transformations possible that result in different performance characteristics that we can also explore.

Let us consider what such a transformation might involve. Firstly, we might expect each Class to have a corresponding Table, each DataType to have a corresponding Type, and each Attribute of each Class to have a corresponding Column in the corresponding Table.

Since our class model doesn't contain any information about the combination of attributes (if any) that constitute a key, we will also need a Column per Table to store an *object-id* that represents an object's identity.

Noting that an Attribute may be multi-valued, it would be rather inefficient to store all the other attributes multiple times in order to store the multi-valued attribute. Instead, we want a separate table for each Column corresponding to a multi-valued Attribute along with a Column for the object-id so we can perform the requisite joins.

If an Attribute's type is a DataType, then the corresponding Column's type would be the Type corresponding to the DataType, but if the Attribute's type is a Class, then the corresponding Column would need to be a foreign key, so its type would be that of the (primary) key of the referenced table which is the object-id Column.

## 4 Features supporting re-use and maintainability

### 4.1 Declarative rules and implicit object creation

Reflecting on the above description, the first thing to notice is that the description is declarative – it describes the

things we want to generate and the reasons why we want to generate them, but it does not say how or when to create them, nor does it involve a description of traversing either the source or the target models. Additionally, while there are natural dependencies between parts of the description, there is no explicit statement of “do this then do that”; rule ordering is implicit, not explicit.

By adopting a declarative approach to transformation specification and letting the transformation engine determine the order of rule application based on the implicit dependencies, objects are just created as needed.

This means that the transformation specifier is relieved of the burden of ensuring an object has been created at the right time and with the right type. For example, a rule may want an object to actually be an instance of a sub-class of the type it was given when another rule created it.

Additionally, relying on the engine to create objects at the right time and with the right type makes it much simpler to make changes to a transformation specification, or when re-using existing transformation rules.

### 4.2 Loosely coupled rules

Our experiences have shown that there are 3 fairly common styles to structuring a large or complex transformation, reflecting the nature of the transformation. They are:

**source-driven**, in which each transformation rule is a simple pattern (often selecting a single instance of a class or association link). The matched element(s) are transformed to some larger set of target elements. This style is often used in high-level to low-level transformations (e.g., compilations) and tends to favour a traversal style

of transformation specification. This works well when the source instance is tree-like, but is less suited to graph-like sources;

**target-driven**, in which each transformation rule is a complex pattern of source elements (involving some highly constrained selection of various classes and association links). The matched elements are transformed to a simple target pattern (often consisting of a single element). This style is often used for reverse-engineering (low-level to high-level) or for performing optimizations (e.g., replacing a large set of very similar elements with a common generic element);

**aspect-driven**, in which the transformation rule is not structured around objects and links in either the source or target, but more typically around semantic concepts, e.g., transforming all imperial measurements to metric ones, replacing one naming system with another, or the various parts of the object-relational transformation described above.

We believe that a good transformation language must support all three styles of transformation specification. However many transformation languages adopt an explicit rule-invocation style that makes aspect-driven rules difficult or impossible to write. This is because rules are only ever applied as a result of traversing a model and explicitly invoking them.

A consequence of the need to explicitly invoke a rule is that all the rules become coupled, and that to properly understand any individual rule one must understand all the rules it invokes transitively and the context in which it is invoked. This need to understand large numbers of rules all linked together is a barrier to maintainability.

By adopting a semantics where all rules are applied together rather than starting from an entry-rule and relying on invocation, the problem of having to understand large numbers of coupled rules is avoided.

Note, sometimes it is desirable to be able to specify rules that capture the structure that is common to multiple rules in a single place. Such rules can be marked as abstract to indicate that they should not be applied except when they are extended by another rule.

Additionally, given a source metamodel,  $M_S$ , and a transformation specification,  $T$  that transforms instances of  $M_S$ , it becomes difficult to re-use and extend  $T$  to define a transformation  $T'$  that transforms instances of the metamodel  $T'_S$  that extends  $T_S$ . This is because rules in  $T'$  can call rules in  $T$ , but not the other way around, so polymorphism in the source metamodel cannot be exploited.

The use of tracking relationships creates indirect links between rules which can operate both from  $T'$  to  $T$  and from  $T$  to  $T'$ . This makes it much easier to extend an exist-

ing transformation because each tracking relationship effectively establishes a logical extension point that spans rules.

### 4.3 Rule overriding

When dealing with transformation libraries or legacy transformations, it may be necessary to disable a particular rule, or to otherwise restrict the set of objects to which it applies, and to perform a different rule instead.

The concept of rule superseding is introduced in our language to allow for this without requiring the original rule to be modified in any way.

## 5 Conclusion

We believe that for MDA to succeed as an approach for building large-scale, and long-lived systems, the transformation language used must include features to support maintainability and re-use of transformation specifications.

Using the implementation<sup>1</sup> of our language we will demonstrate how the language features described in this paper are used to implement the example mapping described in Section 3.

## Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science and Training).

## References

- [1] DSTC, IBM, CBOP. MOF 2.0 Query/Views/Transformations RFP. OMG Document: ad/03-02-03, Mar. 2003.
- [2] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozemberg, editors, *Proceedings of ICGT'02*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer Verlag, 2002.
- [3] Request for Proposal: MOF 2.0 Query/Views/Transformations RFP. OMG Document: ad/02-04-10, Apr. 2002.

---

<sup>1</sup>Further details can be found here:  
<http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/>