Optimising Model-transformations using Design Patterns

Kevin Lano and Shekoufeh Kolahdouz-Rahimi
Dept. of Informatics, King's College London, London, U.K.

Keywords: Model-driven Development, Model Transformations, Design Patterns.

Abstract: This paper identifies how metrics of model transformation complexity can be used to guide the choice and application of design patterns to improve the quality and efficiency of model transformation specifications. Heuristics for choosing design patterns based on the metrics are defined, and the process is applied to an example transformation.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Model transformations are considered to be an essential part of model-driven development, and with the increasing scale and complexity of models utilised within software development, there has been a consequent increase in the scale and complexity of model transformations.

Design patterns for model transformations have been introduced to provide solutions for a number of model transformation specification and design problems, and to improve the quality of model transformation specifications and designs. With a range of different patterns available to apply, it may not be clear to a developer which patterns should be used, and which may produce the most significant improvement in quality in a transformation specification. We propose to use measures of syntactic and semantic complexity to identify problems with a specification, to guide the selection of appropriate patterns, and to evaluate quality improvements gained by applying patterns.

Model transformation patterns are usually applicable to all rule-based transformation languages. In this paper we use a generic rule-based notation to illustrate the patterns.

Section 2 summarises some model transformation design patterns, and section 3 defines heuristics for the selection of model transformation design patterns. Section 4 gives an example of applying these heuristics, section 5 describes related work, and section 6 gives conclusions.

2 DESIGN PATTERNS FOR MODEL TRANSFORMATIONS

We define a model transformation design pattern as "A general repeatable solution to a commonly-occurring model transformation design problem" (Iacob et al., 2008), and similarly define model transformation specification and implementation patterns. Patterns for model transformations have been proposed by several researchers (Agrawal et al., 2005; Bezivin et al., 2003; Cuadrado et al., 2008; Duddy et al., 2003; Iacob et al., 2008; Johannes et al., 2009; Lano and Kolahdouz-Rahimi, 2011). These apply the general concept of software design pattern to the model transformation domain. Since design patterns can improve the flexibility, reusability and comprehensibility of software systems, the same benefits should hold for model transformation patterns.

Two distinct main categories of model transformation design patterns can be identified:

Modularisation/Decomposition Patterns: concerned with restructuring the transformation specification or design to increase its modularity, and to enable the decomposition of a complex transformation into simpler subtransformations, composed eg., sequentially or in parallel.

Optimisation Patterns: concerned with increasing the efficiency of transformation execution, by removing redundant or repeated expression evaluations, reducing data storage requirements, etc.

The following summarises the patterns that we consider here, and their classifications.

Lano K. and Kolahdouz-Rahimi S..
Optimising Model-transformations using Design Patterns.
DOI: 10.5220/0004305100770082
In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2013), pages 77-82
ISBN: 978-989-8565-42-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

Modularisation Patterns:

Phased Construction: Construct target model from source in phases, based on composition hierarchy of target language: sequentially construct successive levels of this hierarchy, either working up or down the hierarchy levels. Individual rules should not update more than one hierarchy level of a model, and rules should be separated into distinct rules for separate hierarchy levels if this condition is violated.

Structure Preservation: 1-1 mapping of entities and entity instances from source to target.

Entity Splitting/Structure Elaboration: Separate the creation of different target entity types into separate rules.

Entity Merging/Structure Abstraction: If multiple source entity types are used to create/update a single target entity type, separate these updates into separate rules.

Map Objects before Links: to map recursive or complex object structures, map source objects to target objects in one phase (or a set of phases), and then create links between target objects in successive phases.

Auxiliary Metamodel: Use additional entities/features to assist in computation of complex/repeated expressions, by storing intermediate results or precomputing repeatedly evaluated expressions. Auxiliary data can also be used to store transformation parameters or traces.

Optimisation Patterns:

Object Indexing: for entities with a primary key attribute, index instances of the entity by the primary key value, in order to provide fast lookup facilities.

Omit Negative Application Conditions: if it can be deduced that the antecedent of a rule always contradicts its succedent, omit a check for falsity of the succedent before applying the rule.

Decompose Complex Navigations: Using navigation chains of collection-valued association ends as domains to select elements to iterate over is inefficient. This pattern decomposes these chains.

Remove Duplicated Expression Evaluations: factor out duplicated expressions from a specification, if these expressions evaluate to the same value.

Implicit Copy: if a transformation mainly copies source to target entities without changing their structure, use an implicit copy mechanism to simplify the specification.

Some patterns are often used together, eg., Phased construction can use Object indexing to look up elements created by a preceding rule. Remove duplicated expression evaluations can use Auxiliary metamodel to precompute the duplicated expressions.

Two examples of pattern definitions are given in the following sections.

2.1 Phased Construction

Application Conditions. Several warning signs in a transformation specification can indicate that this pattern should be applied: (i) if a transformation rule refers to entities or features across more than two levels of a metamodel composition hierarchy; (ii) if a rule contains, implicitly or explicitly, an alternation of quantifiers $\forall\exists\forall$ or longer alternation chains; (iii) if it involves the creation of more than one target instance.

These are signs of a lack of a coherent processing strategy within the transformation, and of excessively complex rules which can hinder comprehension, verification and reuse.

Solution. The identified rule should be split into separate rules, each relating one source model element (or a group of source elements) to one target model element, and navigating no further than one step higher or lower in the entity composition hierarchy (and not both).

Figure 1 shows a typical structure of this pattern.

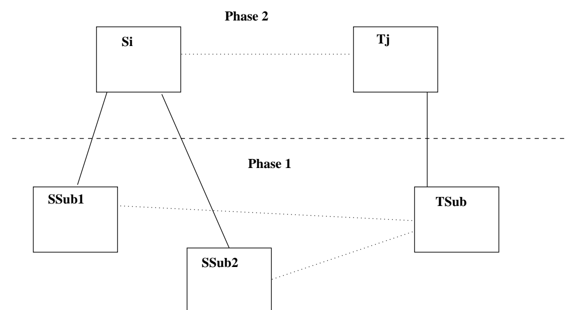

Figure 1: Phased construction pattern.

Schematically, a rule conforming to this pattern should look like:

for each $s : S_i$ **satisfying** $SCond_{i,j}$
create $t : T_j$ **satisfying** $TCond_{i,j}\ and\ Post_{i,j}$

where the S_i are entities of the source language S, the T_j are entities of the target language T, $SCond_{i,j}$ is

a predicate on s (identifying which elements the rule should apply to), and $Post_{i,j}$ defines t in terms of s. These predicates may use features of s and t, but not any longer navigations. There should not be further alternations of quantifiers in $Post$, because such complexity hinders comprehension.

2.2 Object Indexing

Application Conditions. Required when frequent access is needed to objects or sets of objects based upon some unique identifier attribute (a primary key).

Lookup of objects by means of a $select$ expression of the form

$$C.allInstances()\rightarrow select(id = v)\rightarrow any()$$

or $C.allInstances()\rightarrow select(id : v)$ can be very inefficient, with a worst case time complexity proportional to the number of elements of C.

Solution. Maintain an index map data structure $cmap$ of type $IndType \nrightarrow C$, where C is the entity to be indexed, and $IndType$ the type of its primary key ind. Access to a C object with key value v is then obtained by applying $cmap$ to v: $cmap.get(v)$. When a new C object c is created, add $c.ind \mapsto c$ to $cmap$. When c is deleted, remove this pair from $cmap$.

Figure 2 shows the structure of the pattern. The map $cmap$ is a qualified association, and is an auxiliary metamodel element used to facilitate separation of the specification into loosely coupled rules.

The C object with primary key value v is denoted by $C[v]$. Likewise for the set of C objects with a key value in v, if v is a set.

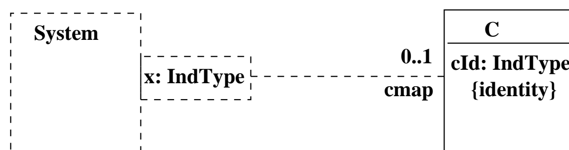

Figure 2: Object indexing structure.

Expressions

$$C.allInstances()\rightarrow select(id = v)\rightarrow any()$$

in the specification are then replaced by $C[v]$ for accessing single C objects by identity, and expressions $C.allInstances()\rightarrow select(id : v)$ for accessing sets of C objects are also replaced by $C[v]$, reducing the syntactic complexity of the specification.

3 SELECTION OF PATTERNS

By formally defining metrics of model transformation specifications, it is possible to give precise guidance to transformation developers about which patterns are appropriate to apply, and to measure the benefits obtained by applying a pattern.

We consider the following measures of model transformations:

1. Syntactic complexity 1: count of entity references and feature references in a rule.
2. Syntactic complexity 2: count of operator occurrences in a rule.
3. Alternation of quantifiers: count of forAll exists forAll nestings in a rule.
4. Multiple creation: count of number of distinct element creation actions within a rule.
5. Multiple expression occurrences: count of number of cloned expressions within a rule (or within a complete specification).
6. Maximally complex read subexpression: syntactic complexity (1 + 2) of most complex read subexpression in rules.
7. For each source entity in a migration transformation, the proportion of source entity features f which are simply copied to a target entity feature: $t.g = s.f$.
8. For each target entity, a count of the number of rules that create/modify its elements.
9. The number of self-associations or cyclic structures of associations in the source language which are mapped to such structures in the target.
10. The number of expressions with the form

    $$E.allInstances()\rightarrow select(id = v)$$

    or $E.allInstances()\rightarrow select(id : v)$ where id is a primary key of entity E.
11. The number of rule universal quantification ranges (on the antecedent of rules) which consist of navigations through two or more collection-valued features.

The first two measures are considered to be a basic metric of syntactic complexity:

Syntactic complexity = Syntactic complexity 1 + Syntactic complexity 2

An alternation of quantifiers value greater than 0 in a rule suggests applying the “Phased construction” pattern to the rule. A multiple creation value greater

than 1 also indicates this pattern, or the "Entity splitting" pattern (depending on how the targets are derived from the source). A multiple expression occurrence count > 1 suggests the "Remove duplicated expression evaluations" pattern: by using let expressions if the clones occur only in one rule, or by precomputation and the "Auxiliary metamodel" pattern if they occur in multiple rules. A high value of maximum complexity for a read subexpression also suggests the auxiliary metamodel pattern, even if there is only one occurrence of the subexpression.

In each case, applying a pattern should reduce the complexity metrics 1 and 2 for the selected rule, and not increase the metrics of any other existing rule.

If there is a high proportion of feature copying from source to target, then the Structure preservation and Implicit copy patterns are relevant.

If a target entity is updated by more than one rule, this indicates use of the Entity merging pattern for this entity.

If there are cases of self-associations or cyclic association structures in the source being mapped to such structures in the target, then Map objects before links is relevant for this part of the transformation.

If there are any occurrences of

$$E.allInstances()\rightarrow select(id = v)$$

or $E.allInstances()\rightarrow select(id : v)$ expressions in rules, this suggests use of Object indexing for E.

If there are any quantifier range navigations through multiple collection-valued features in the antecedent of a rule, this indicates applying the "Decompose complex navigations" pattern.

4 CASE STUDY

A simple example of the application of metrics to guide restructuring is the following basic version of a refinement transformation to map UML class diagrams to relational database tables.

Figure 3 shows the metamodels.

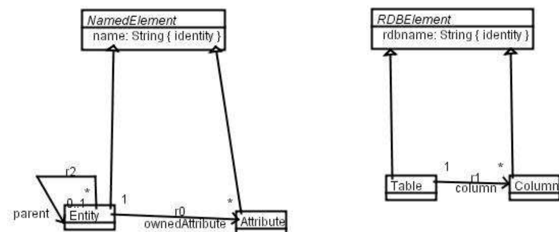

Figure 3: Metamodels of UML to relational database mapping.

In the original version of this transformation there are two transformation rules: $(R1)$:

for each $e : Entity$ **satisfying** $e.parent = \{\}$
create $t : Table$ **satisfying**
$t.rdbname = e.name\ and$
$Column\rightarrow exists(k\ |$
$k.rdbname = e.name + \text{"_Key"}\ and$
$k : t.column)$

which creates a table and its primary key for each root class e, and $(R2)$:

for each $c : Entity$; $a : c.ownedAttribute$;
$t : Table.allInstances()\rightarrow select($
$rdbname : c.parent.closure.name)$
create $k : Column$ **satisfying**
$k.rdbname = a.name\ and$
$k : t.column$

which maps each attribute a of each entity c to a column k, which is added to the table corresponding to the root ancestor of c. $parent.closure$ is the transitive closure of the $parent$ association. Only one table t will exist with a name in $c.parent.closure.name$, ie., the table for the root class of c.

Table 1: Complexity metrics of rules.

Rule	Syntactic complexity	Multiple creation	Max. expression complexity
$R1$	19	1	2
$R2$	21	0	8
Total:	40	1	10

The metrics for these rules are as follows (Table 1). It can be seen that $R1$ has moderate complexity, but that it creates multiple objects, suggesting the use of the Phased construction pattern. $R2$ has a high value of maximum expression complexity: the subexpression $Table.allInstances()\rightarrow select(rdbname : c.parent.closure.name)$ has syntactic complexity 8, which suggests applying the Auxiliary metamodel pattern to simplify its computation. This subexpression also exhibits multiple duplicated evaluations: the closure of the $parent$ association will be repeatedly evaluated for classes near the top of the inheritance hierarchy. The expression also matches exactly the specific conditions for introducing object indexing, so this pattern is introduced first for $R2$.

Applying Phased construction to $R1$ and using Object indexing to look up $Table$ instances produces the following rules:

$(R1a)$:
for each $e : Entity$ **satisfying** $e.parent = \{\}$
create $t : Table$ **satisfying** $t.rdbname = e.name$

$(R1b)$:
for each $e : Entity$ **satisfying** $e.parent = \{\}$
create $k : Column$ **satisfying**
$k.rdbname = e.name + \text{"_Key"}\ and$
$k : Table[e.name].column$

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

The complexity of each of these rules is lower than that of $R1$, and the problem of multiple object creation has been solved (Table 2).

Introducing object indexing for $R2$ leads to the simpler constraint $R2a$:

for each $c : Entity;\ a : c.ownedAttribute;$
$t : Table[c.parent.closure.name]$
create $k : Column$ **satisfying**
$k.rdbname = a.name\ and$
$k : t.column$

which can be further simplified to:

for each $c : Entity;\ a : c.ownedAttribute$
create $k : Column$ **satisfying**
$k.rdbname = a.name\ and$
$k : Table[c.parent.closure.name].column$

The complexity has been reduced to 16, however the remaining problem in $R2a$ is the repeatedly-evaluated complex subexpression $c.parent.closure.name$. This repetition can be avoided by applying the auxiliary metamodel pattern, and precomputing the root class of each class, and storing this in an auxiliary association

$$rootClass : Entity \rightarrow Entity$$

prior to execution of the transformation. Alternatively, only the name of the root class could be stored.

Adopting the first option, $R2a$ can be simplified to:

$(R2b):$
for each $c : Entity;\ a : c.ownedAttribute$
create $k : Column$ **satisfying**
$k.rdbname = a.name\ and$
$k : Table[c.rootClass.name].column$

Table 2: Complexity metrics of restructured rules.

Rule	Syntactic complexity	Multiple creation	Max. expression complexity
$R1a$	9	0	1
$R1b$	15	0	3
$R2b$	15	0	4
Total:	39	0	8

The average complexity of the rules in the revised system has been reduced to 13, compared with 20 in the original system. The total complexity has also been reduced, and the problem characteristics have been removed.

We also measured the improvement in efficiency in the restructured specification, using a set of six test cases:

1. Test case 1: 100 classes, arranged in a vertical inheritance hierarchy (class n+1 is a subclass of class n, etc), with 10 attributes each, all names are distinct.
2. Test case 2: same as case 1, with 500 classes.
3. Test case 3: same as case 1, with 1000 classes.
4. Test case 4: 100 classes, arranged in a flat inheritance hierarchy (all classes are root classes), with 10 attributes each, all names are distinct.
5. Test case 5: same as case 4, with 500 classes.
6. Test case 6: same as case 4, with 1000 classes.

We executed the transformation using the UML-RSDS tools (Lano, 2012), on a Pentium 4 machine running Windows XP.

Table 3 shows the execution times (in ms) of the original and revised specifications of the transformation on these case studies. It can be seen that the original specification is impractical for processing models with deep inheritance hierarchies, and that the revised version has substantially better performance on such models.

Table 3: Efficiency comparison.

Test case	Original specification	Restructured specification
1	320	60
2	23514	1182
3	Out of memory	3826
4	70	50
5	1072	1022
6	4016	3805

5 RELATED WORK

The use of metrics to guide design choices and to assist in automation of software engineering is a topic within the field of *search-based software engineering* (SBSE) (Harman and Jones, 2001). SBSE techniques use various search mechanisms, such as linear programming and genetic algorithms, to search for optimal values of some fitness function, which can represent the quality of a software design or other desired aspect of the design, such as testability. The paper (Lutz, 2001) uses a fitness function which represents the quality of a hierarchical decomposition of a software system. The aim of the optimisation process in this case is to produce simpler and more understandable decompositions. There has also been substantial work on automated modularisation of software to increase cohesion and decrease coupling: the paper (Mancoridis et al., 1999) introduced automated clustering techniques as an aid to software maintenance. Rather than using a single fitness function, we have defined a collection of measures, which enable us to

diagnose specific problems in a transformation specification, and to propose specific solutions.

Another modularisation measurement approach is described in (Tzerpos et al., 1999). A metric of dissimilarity between clusters is used to compare clustering approaches, but this only indirectly provides a measure of quality of the clustering, since an optimally-modularised version of a system is needed as a reference point.

6 CONCLUSIONS

We have shown that metrics can be used to guide the choice of specification improvement steps such as pattern applications. The metrics and guidelines described here have been implemented in UML-RSDS (Lano, 2012). Although we have focussed on model transformation patterns, we consider that our approach could be generally applicable to a wide range of pattern categories, such as patterns for EIS architectures or service-oriented architectures.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

REFERENCES

Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G. (2005). Reusable Idioms and Patterns in Graph Transformation Languages, *Electronic notes in Theoretical Computer Science*, pp. 181–192.

Bezivin, J., Jouault, F., Palies, J. (2003). *Towards Model Transformation Design Patterns*, ATLAS group, University of Nantes.

Cuadrado, J., Jouault, F., Molina, J., Bezivin, J. (2008). Optimization patterns for OCL-based model transformations, *MODELS 2008*, vol. 5421 LNCS, Springer-Verlag, pp. 273–284, 2008.

Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J. (2003). Model transformation: a declarative, reusable pattern approach. In *7th International Enterprise Distributed Object Computing Conference (EDOC '03)*.

Harman, M., Jones, B. (2001). Search-based software engineering, *Information and Software Technology*, 43 (14), pp. 833–839, 2001.

Iacob, M., Steen, M. Heerink, L. (2008). Reusable model transformation patterns, *Enterprise Distributed Object Computing Conference*.

Johannes J., Zschaler, S., Fernandez, M., Castillo, A., Kolovos, D., Paige, R. (2009). Abstracting complex languages through transformation and composition, *MODELS 2009*, LNCS 5795, pp. 546–550.

Lano, K., Kolahdouz-Rahimi, S. (2011). Design patterns for model transformations, *ICSEA 2011*.

Lano, K. (2012). *UML-RSDS manual*, http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf.

Lutz, R. (2001). Evolving good hierarchical decompositions of complex systems, *Journal of Systems Architecture*, 47, pp. 613–634.

Mancoridis, S., Mitchell, B., Chen, Y., Gansner, E. (1999). Bunch: a clustering tool for the recovery and maintenance of software system structures, *IEEE International Conference on Software Maintenance*, pp. 50–59, IEEE Press.

Tzerpos, V., Holt, R. (1999). *MoJo: A distance metric for software clustering*, University of Toronto.