On the Use of Operators for the Co-Evolution of Metamodels and Transformations

Steffen Kruse

OFFIS - Institute for Information Technology
Escherweg 2, 26121 Oldenburg, Germany
steffen.kruse@offis.de
http//www.OFFIS.de

Abstract. The artefacts used in model driven approaches are often tightly coupled. Besides models being bound to metamodels by a conformance relation, transformation descriptions are defined on metamodels to perform their function. When the metamodels are changed during development or due to changing requirements, existing transformations need to be adapted. We propose a set of operators to ease this task. The operators are applied to metamodels to perform a change and allow the (semi-) automatic co-evolution of transformations.

Keywords: metamodel, model transformation, co-evolution, operators

1 Introduction

Model Driven Software Development (MDSD) is an approach to software engineering to ease the handling of complexity in the development and maintenance of modern software systems. MDSD is based on the concept of a model as a representation of a (software) system [2]. Models are expressed in languages, models of which in turn are called metamodels. Further techniques used in MDSD are code generation (where source code is generated from a set of models) and model to model transformation, of which code generation can be seen as a special case. The artefacts used in MDSD projects are tightly coupled. Models must be conformant to their respective metamodel to be valid as model transformations must match one or more metamodels to perform their function. This can become a problem when faced with evolution: as metamodels are extended or adapted, the dependencies may break until all dependent artefacts are adapted accordingly. The problem of co-evolution for metamodels and models has been addressed by a number of approaches (see for example [3, 8–10]). While being related, the problem of the co-evolution of metamodels and model transformations poses different and unique problems.

In this paper, we propose an initial set of operators that can be applied on a given metamodel to perform changes and allow the automatic or semi-automatic co-evolution of transformation descriptions. Atomic operators can be combined to form more complex ones. We implemented these operators in Java and tested them on copy transformations described in ATL to gain initial insight into the viability of this approach.

2 Impact of metamodel changes on transformations

Model to model transformations play an important role in model driven software development (MDSD) [16]. Numerous languages have been developed, dedicated solely to specify such transformations [4]. We chose the declarative parts of the Atlas Transformation Language (ATL) [11] as a starting point for our operator set. In essence, a model transformation defined in ATL is made up of a set of rules, where each rule produces one or more target model elements from a source model element. The application of a rule is determined by a source pattern, defined in terms of a source metamodel. The created elements are conformant to a target metamodel. ATL relies on a slightly adapted version of the Object Constraint Language (OCL) to perform model navigation and to calculate values for target properties. To analyse the impact on metamodel changes on transformation rules in ATL, we first discern which metamodel involved in the transformation is changed. We refer to the source metamodel as the left-hand-side metamodel (LHS) and the target metamodel as right-hand-side (RHS) respectively.

Figure 1 shows an excerpt of the (informal) syntax of an ATL Matched Rule, taken from the ATL Language Guide [1]. Metamodel changes to the LHS can impact the following parts of an ATL rule: ① The source pattern ② conditions imposed on the source pattern (expressed in OCL) and ④ the binding assignment. Changes to the RHS are potentially reflected in: ③ the target pattern (each rule can have numerous target patterns) and ④ the definition of properties in the binding. We leave out the imperative part ((the *do*-statement)) and the variable section (see the ATL User Guide [1] for further detail). Both are to be looked at in the future, building on the work presented here.

```
rule rule_name {
    from
        in_var : in_type [in model_name]? [(
            condition
        )]?
    [using {
        var1 : var_type1 = init_exp1;
        ...
        varn : var_typen = init_expn;
    }]?
    to
        out_var1 : out_type1 [in model_name]? (
            bindings1
        ),
        out_var2 : distinct out_type2 foreach(e in collection)(
            bindings2
        ),
        ...
        out_varn : out_typen [in model_name]? (
            bindingsn
        )
    [do {
        statements
    }]?
}
```

Fig. 1. ATL MatchedRule Syntax

OCL is used for two purposes in ATL: to formulate conditions on the applicability of rules and to calculate values in the property bindings. In both cases, OCL statements may navigate over models and can potentially involve any part of the LHS metamodel. This can lead to very complex source patterns for complex rules. We simplify the influence of OCL expressions on the evolution process for a start, by either disabling the application of operators when they impact expressions or delegating to human intervention (bar rename operations which can be easily undertaken for OCL expressions). Depending on the operator and the expression structure, further adaption of the expressions may be possible, which would broaden the use of the operator. We leave this investigation for future work. Markovic and Baar have shown how OCL expressions can be migrated along with UML models [13], which gives good pointers to the adaptation of complex OCL expressions in transformation rules.

3 Operators

Table 1 summarizes the influence of the operators on transformation rules, depending on where they occur and which part of the rule they affect. For the LHS these are the source pattern element (Source P.), the condition, and the binding assignment (Binding A.). For the LHS they are the target pattern (Target P.) and the property binding (Binding). The kind of adaptation possible is given as follows: [A] stands for automatic adaptation, [-] the operator has no influence and [H] human intervention is needed. In cases where we list the two possibilities [A/H], automatic adaptation is possible most of the time – depending on further detail on the kind of change or the structure of the rule (see the detailed description of the operators.) The next sections discuss the operators and their influence in more detail.

3.1 Atomic Operators

- **Rename Class/Attribute/Relation**: Elements (classes/attributes/relations) of a metamodel are renamed. These operators are very similar – both in their pre-conditions and influence on transformation rules. Names of elements are required to be unique in their given namespace, so existing names are excluded by pre-condition when applying these operators on elements. For attributes and relations this also has to be checked for inheritance. When the pre-conditions hold, metamodel references can be adapted automatically in all parts of transformation rules.
- **Add Class**: A new class is added to the metamodel. The class must be unique in its namespace. The syntactic correctness is preserved for transformation rules on both sides, as existing rules are not influenced by new elements. Depending on the semantics of the transformation, new rules may have to be created by hand to provide for the new class.
- **Add Relation / Attribute**: A new relation or attribute (property) is added to a class in the metamodel. Its name must be unique for the owning class

Operator	LHS			RHS	
	Source P.	Condition	Binding A.	Target P.	Binding
Rename Class	A	A	A	A	A
Rename Attribute	-	A	A	-	A
Rename Relation	-	A	A	-	A
Add Class	A	A	A	A	A
Add Relation	-	A	A	-	A/H
Add Attribute	-	A	A	-	A/H
Delete Class	A	-	-	-	A
Delete Relation	-	A	A/H	-	A
Delete Attribute	-	A	A/H	-	A
Pull Up Attribute	-	A	A	-	A/H
Push Down Attribute	-	A	A/H	-	A
Pull Up Relation	-	A	A	-	A/H
Push Down Relation	-	A	A/H	-	A
Introduce Inheritance	-	A	A	-	A/H
Extract Superclass	A	A	A	A	A/H
Flatten Hierarchy	A	A	A	A	A

Table 1. Operator Influence

and classes along the inheritance hierarchy. For the LHS metamodel, transformation rules do not have to be adapted to preserver syntactic correctness. For the RHS, human intervention is required only when the added property is obligatory and no default value is supplied in the metamodel. In this case, all rules which feature a target pattern for the owning class or descending classes must be adapted to supply a value for the new property by hand.

– **Delete Class**: An empty and unreferenced class is deleted from the metamodel. To empty a class and remove all references to it first, the *Delete Property*, *Push Down Property* etc. operators can be applied prior. For the LHS, transformation rules that feature the deleted class as source pattern element can be removed. As the operator can only be used when the class is not referenced, the class cannot feature in the condition or binding assignment parts of rules for other source pattern elements. For the RHS, the target patterns for the class can be removed. Should this leave a rule without any target pattern, the entire rule can be removed.
– **Delete Relation / Attribute**: A relation or attribute (property) is deleted from a class. Rules that need to be adapted either refer to the class from which the property is removed or any subclass; or contain OCL expressions that refer to the removed property. The adaptation is as follows:
  • For the LHS:
    ∗ Conditions - If the deleted property is part of a condition restricting the application of a rule, the condition can be removed. This means that the rule potentially applies more often. The user should be warned of this behaviour.
    ∗ Binding Assignment - If the property is featured in a binding assignment of an RHS property, the whole assignment can be removed

if the RHS property is non-obligatory or has a default value. This would preserve syntactic correctness. Otherwise, human intervention is needed.

- RHS: The property can have a binding for the target pattern of the class or the subclasses it was removed from. The binding assignment can be removed to preserve syntactic correctness.

– **Pull Up Relation / Attribute**: A relation / attribute (property) is pulled from all subclasses into a superclass and removed from the subclasses. This may mean that the user first has to introduce the property to subclasses that don't have the given property. This can be done by applying the *Add Reference /Attribute* operators.
  - Rules involving the superclass: All rules that match the superclass to which the property is pulled can be treated analogous to the *Add Relation/Attribute* operators. This means, for the LHS case, no change is needed, as the addition of a property does not break the syntactic correctness of the rules. For the RHS, human intervention is only needed if the property is obligatory and no default value is given. In all other cases no change is needed.
  - Rules involving the subclasses: All rules that match the subclasses from which the property is pulled require no adaptation, as the property remains available through inheritance.
– **Push Down Relation / Attribute**: A property is pushed from a superclass A into all subclasses that are removed from the superclass by exactly one level of inheritance. Subsequent application of the *Push Down Property* operator along the inheritance tree can push properties to further removed classes. Pushing properties to all child classes makes little sense on its own, it is expected that the user applies further operations like removing the property with the *Delete Property* operator on subclasses that don't use the property (this was the original intention of Fowlers Push Down Field Refactoring [7]) or removing the superclass with *Delete Class* if it contains no further properties.
  For this operator we have to discern between rules that use the superclass from which the property is removed and the subclasses that receive the property. For rules on the subclasses, the impact is simple: no adaption is needed as the property was previously available through inheritance. For the superclass, the behaviour is like the *Delete Relation / Attribute* operators:
  - For the LHS:
    * Conditions - If the removed property is part of a condition restricting the application of a rule, the condition can be removed. This means that the rule potentially applies more often. The user should be warned of this behaviour.
    * Binding Assignment - If the property is featured in a binding assignment of an RHS property, the whole assignment can be removed if the RHS property is non-obligatory or has a default value. This would preserve syntactic correctness. Otherwise, human intervention is needed.

  - RHS: The property can have a binding for the target pattern of the class it was pushed down from. The binding assignment can be removed to preserve syntactic correctness.
- **Introduce Inheritance**: Inheritance is introduced between two classes. For the LHS, no change is needed; the syntactic correctness of all rules is preserved. For the RHS, human intervention is only needed when the new superclass contains obligatory properties (without default values). Then all rules matching the new subclass must be adapted to supply a value for the newly inherited property. This is analogous to the *Add Attribute/Reference* operators.

3.2 Complex Operators

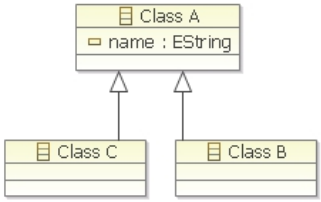
Complex operators are (in part) made up of simple operators. There are two reasons for providing complex operators: 1) convenience – to sum up common cases in one operator and 2) the use of the complex operator provides more information that can be used for the adaptation of transformations than just the combined use of simple operators. *Extract Superclass* is an example of the first case, while *Flatten Hierarchy* applies to both.

- **Extract Superclass**: A new superclass of a given class is introduced and some properties of the subclass are moved to the new superclass. This operator can be wholly made up of the Add Class, Introduce Inheritance and repeated use of Pull Up Property Operators. The same constraints and adaptions apply. This operator is very much like Fowlers *Extract Superclass* refactoring [7].

- **Flatten Hierarchy**: The purpose of this operator is to reduce the inheritance graph. At first glance, this operator can be implemented using the *Push Down Property* operator on all properties of the superclass until it is empty and then use the *Delete Class* operator to remove the superclass. Yet, a special case exists where this is impractical: Assume this operator is used on the LHS metamodel and there are rules that apply to the superclass in the source pattern. Before the change, the rule would be executed for all instances of subclasses as well as for those of the superclass. If the rule simply gets deleted (as with the Delete Class operator) target elements are no longer created for the subclass instances. We consider this to be an unexpected and inconvenient result for the user. Since we have more information available when the *Flatten Hierarchy* operator is used explicitly, we propose creating copies of rules that match the superclass for each subclass and changing the source pattern to match each subclass. Then the original rule for the superclass can be removed.

4 Evaluation

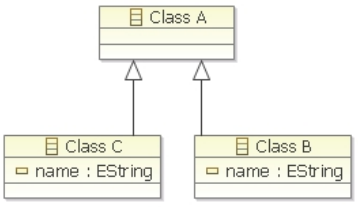
We implemented the operators in Java and perform them on metamodels in Ecore and the abstract syntax model of ATL transformations in the Eclipse




```
rule Class_A_COPY {
    from
        class_A : Metamodel!"Class A"
        (
            class_A.oclIsTypeOf(Metamodel!"Class A")
        )
    to
        class_A_COPY : Metamodel!"Class A" {
            name <- class_A.name
        )
}

rule Class_C_COPY {
    from
        class_C : Metamodel!"Class C"
        (
            class_C.oclIsTypeOf(Metamodel!"Class C")
        )
    to
        class_C_COPY : Metamodel!"Class C" {
            name <- class_C.name
        )
}
```

Fig. 2. Flatten Hierarchy: Initital State




```
rule Class_A_COPY {
    from
        class_A : Metamodel!"Class A"
        (
            class_A.oclIsTypeOf(Metamodel!"Class A")
        )
    to
        class_A_COPY : Metamodel!"Class A" {
        )
}

rule Class_C_COPY {
    from
        class_C : Metamodel!"Class C"
        (
            class_C.oclIsTypeOf(Metamodel!"Class C")
        )
    to
        class_C_COPY : Metamodel!"Class C" {
            name <- class_C.name
        )
}
```

Fig. 3. Flatten Hierarchy: after Push Down Attribute


```
rule Class_C_COPY {
    from
        class_C : Metamodel!"Class C"
        (
            class_C.oclIsTypeOf(Metamodel!"Class C")
        )
    to
        class_C_COPY : Metamodel!"Class C" {
            name <- class_C.name
        )
}
```

Fig. 4. Flatten Hierarchy: Final State

ATL infrastructure[1]. For an initial evaluation of the operators, we apply them to *copy transformations*. A copy transformation for a given metamodel produces an exact copy of all models conforming to the metamodel. It features the same metamodel both as source and target. Copy transformations can be generated by a higher-order transformation (HOT). This HOT takes any given metamodel and produces a matching copy transformation. Copy transformations can be used e.g. as foundations for model refinement [12]. To use operators for the evolution of copy transformations makes little sense in practice, as the transformation can be re-generated anytime the metamodel evolves. Yet, by comparing the two, we gain some initial insight into how the operators fare.

As an example, we discuss the use of the complex *Flatten Hierarchy* operator on a simple metamodel. We generated the copy transformation for the metamodel using Xpand of the Eclipse Model To Text (M2T) project[2]. The initial state of the metamodel and the corresponding copy transformation are shown in figure 2. (We have removed the copy rule for Class B for brevity in the figure; it is an exact duplicate of the rule for Class C.) The generated copy transformation contains a rule for each class, and the value of the *name* attribute is copied in the binding section. The condition in each source pattern ensures that the rule only matches exactly the given class and not subclasses (who have their own copy rules). We apply the *Flatten Hierarchy* operator to metamodel and transformation and interpret the change to have taken place on the RHS. The operator first uses the *Push Down Attribute* operator to move the *name* attribute from the superclass A to the subclasses B and C. As a result, the binding assignment for the attribute is removed for the copy rule of class A. The other rules are left unchanged (see figure 3). After the superclass A is relieved of its attribute, it can be removed. The result is shown in figure 4. In consequence, the copy rule for class A is removed from the copy transformation.

Generating the copy transformation for the new metamodel yields the same resulting set of rules. We take this as an indication that the approach is at least viable.

[1] see `http://eclipse.org/atl/`
[2] See `http://www.eclipse.org/modeling/m2t/?project=xpand`

4.1 Discussion

First results for copy transformations indicate that an operator based approach can be beneficial to the co-evolution problem for metamodels and model transformations. In many cases, at least syntactic correctness of transformations can be preserved when making changes. Yet, the semantics of transformations are very difficult to cater for. This is one of the main differences between the co-evolution transformations vs. co-evolution of models. The semantics of the conformance relation between metamodels and models is well defined, while the semantics of transformations very much depend on their purpose. For example, copy transformations have to regard every class and attribute of a metamodel. When a class is added to the metamodel, the rules of the copy transformation can be left unchanged for the transformation to be syntactically correct – yet it no longer performs according to its purpose.

Furthermore, the impact on transformations depends not only on the type of change performed on the metamodel, but also how the affected elements are used in transformation rules. It is possible that one transformation can be automatically adapted to a change in a metamodel, while another transformation using the same metamodel requires human intervention. We hope to find further insight in the future as to which kinds of transformations lend themselves well to automatic co-evolution and which don't.

5 Future and Related Work

Further operators need to be defined to make the set complete (any metamodel can be de- and reconstructed using operators.) Regarding the adaptation of OCL expressions along with the rules may broaden the use of operators. So far, we have only looked at transformations defined in ATL, but the given approach may lend itself well to other transformation languages and a comparison is seen to be beneficial. Finally, we plan to compare our operators to those used for metamodel and model co-evolution as a combined approach may make evolution easier – if it is possible.

The use of operators on co-evolution issues has been covered well for models [3, 8–10]. For metamodel and transformation co-evolution, Mndez et al. have described the problem and given first pointers to an operator-based approach [14]. Di Ruscio et al.[5] describe an initial approach using a DSL for both metamodel and model and transformation co-evolution. For OCL, Markovic and Baar have shown how OCL expressions can be migrated along with UML models [13]. Other approaches to the problem exist, for example based on evolution in the ontology space [15] or by always generating transformations for all metamodel versions [6]. The co-evolution problem is complex and has many facetts, as transformations have a vast field of application. Which approach is well suited for which case remains to be evaluated in the future.

References

1. Atl/user guide - the atl language, `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`, last seen: 14.07.2011
2. Bzivin, J.: In search of a basic principle for model driven engineering. In: UPGRADE. CEPIS (Council of European Professional Informatics Societies), NOVTICA (2004)
3. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. Enterprise Distributed Object Computing Conference, IEEE International 0, 222–231 (2008)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. 45(3), 621–645 (2006)
5. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd International Workshop on Model Comparison in Practice. pp. 30–38. IWMCP '11, ACM, New York, NY, USA (2011)
6. Didonet Del Fabro, M., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Software and Systems Modeling 8, 305–324 (2009)
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
8. Garces, K., Jouault, F., Cointe, P., Bézivin, J.: Adaptation of Models to Evolving Metamodels. Research Report RR-6723, INRIA (2008)
9. Gruschko, B., Kolovos, D.S., Paige, R.F.: Towards synchronizing models with evolving metamodels. In: Proceedings of the Workshop on Model-Driven Software Evolution - MoDSE2007 at the 11th European Conference on Software Maintenance and Reengineering - CSMR 2007 (2007)
10. Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Proceedings of the Third international conference on Software language engineering. pp. 163–182. SLE'10, Springer-Verlag, Berlin, Heidelberg (2011)
11. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.M. (ed.) MoDELS Satellite Events. Lecture Notes in Computer Science, vol. 3844, pp. 128–138. Springer (2005)
12. Kapova, L., Goldschmidt, T.: Automated feature model-based generation of refinement transformations. Software Engineering and Advanced Applications, Euromicro Conference 0, 141–148 (2009)
13. Markovic, S., Baar, T.: Refactoring ocl annotated uml class diagrams. Software and System Modeling 7(1), 25–47 (2008)
14. Méndez, D., Etien, A., Muller, A., Casallas, R.: Transformation migration after metamodel evolution. In: International Workshop on Model and Evolution. Oslo, Norway (October 2010)
15. Roser, S., Bauer, B.: Journal on data semantics xi. chap. Automatic Generation and Evolution of Model Transformations Using Ontology Engineering Space, pp. 32–64. Springer-Verlag, Berlin, Heidelberg (2008)
16. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software 20, 42–45 (2003)