Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines

Marten Sijtema

Universiteit Twente
Software Engineering Group, Enschede, The Netherlands
m.sijtema@ewi.utwente.nl

Abstract. Various approaches show that software product lines can be implemented using the Model-Driven Engineering concept of successive model refinements. An important aspect of Product-Line Engineering (PLE) is the management of variability. In this paper we propose a strategy to let the model transformation language ATL handle the variability. We consider a transformation sequence that can generate a family of products. Furthermore, we model the variability separately in a feature diagram. In our case, the features from this diagram will have corresponding feature realization artefacts whose blueprints are defined as meta classes residing in meta models throughout the transformation sequence. We use model-to-model transformations written in ATL to instantiate these feature realization artefacts from the meta models, guided by the feature model's feature selection. This paper shows that the conventional language constructs of ATL (ie. rules) are ineffective in managing variability this way. We therefore extend the concrete syntax of ATL with the concept of variability rules. This yields a first-class language construct for variability management. Variability rules are declarative, use implicit scheduling and are a true modular extension; they inherit from the normal rule class in the ATL meta model. This means that they have the same quality properties as normal rules. The execution semantics of variability rules – execute iff the corresponding variant in the feature model is selected – is implemented in a higher-order transformation, which compiles an extended ATL model back to a normal ATL model, therefore no new ATL plugin needs to be installed.

1 Introduction

Various approaches show that software product lines can be implemented using the Model-Driven Engineering concept of successive model refinement. An important aspect of PLE is the management of variabilities. This can be done in various ways [5][3][1][2].

This paper aims to provide first-class variability management means for the model transformation language ATL [4], using a separate feature model for configuration. The basic modular construct of rule-based model transformation languages, a rule, is used as a starting point. In ATL, there are five types of rules: normal rules, abstract rules, rules that inherit from other rules (ie. sub rules), lazy rules and called rules. This paper introduces **variability rules**. Ideally, variability rules should have the same quality properties as normal rules.

Variability rules are best used in the context of a transformation sequence which successively refines models, and where blueprints of feature realization artefacts reside in the meta models. Thus, meta models define the complete product space by defining feature realizations for every feature. The variability rules instantiate and integrate a particular set of feature realizations into a final product, according to the feature model's selection.

The reason for using a transformation rule as key construct for managing variability is the observation that features from a feature model have to be associated with, or mapped to, feature realization artefacts in order to enable automatic product derivation. Specifically, the feature model should be the driver for configuring/assembling a product family member. Rules are purpose-built for defining mappings. Furthermore, since rules have mature mechanisms for matching, querying and creating (instances of) meta model elements, and thus (instances of) feature realization elements, they are well-suited for integrating feature realizations with the rest of the system. Furthermore, rules are modular constructs with a declarative nature, which makes them easy to use and allow elegant implementations as opposed to more imperative solutions.

2 Paper Outline

This paper first explains the need for variability management in model transformations in model transformations in Section 3. Section 4 describes the concept of variability rules, their syntax and semantics. Section 5 explains how the variability rules are implemented in a higher-order transformation.

During this project, a more complex type of variability was discovered. This type of variability not only has a dependency on the feature model, but also on the input model of the transformation. For handling this type of variability, the variability rule concept and concrete syntax are extended a bit further, and this is explained in Section 6. This last part is currently work-in-progress, but there are already some results worth mentioning.

The true open issues are explained in Section 7, followed by Section 8 which shows related work. We conclude this paper in Section 9.

3 The Need for Variability Management in Model Transformations

Model transformations are a central concept in MDE, and by using a transformation sequence as a software product line generator, model transformations should be capable of dealing with variabilities.

3.1 Example of Model Transformations Managing Variability

To illustrate variability management in model transformations, we describe an example case where transformations steer the feature assembly process. This example is based on a real case where input meta models conforming to Ecore are transformed into a model of a web-based, data-centric information systems with basic create/read/update/delete (CRUD) operation support, which is in turn transformed into code. The model-to-model step is shown in Figure 1. The meta model *outMM* is an excerpt of the real case meta model, and only shows a *Model* class (note: a Model from the Model-View-Controller pattern), as well as a *DatabaseTechnology* element.

Every derived instance model of *outMM* will contain a *Model* element *for each* instance of an *EClass* element from the source meta model *inM*. This *Model* element will have the same name as the *EClass*'s name attribute, concatenated with the string ‘Model’. In this case there is only one instance of *EClass* in the source meta model, *Person*. This results in the instantiation of a *Model* element with the name attribute set to ‘PersonModel’. Note that a different input meta model is likely to have multiple instances of *EClass*, and will therefore transform in a model with multiple instances of the *Model* meta class. In the real case, an MVC pattern is generated for each *EClass*.

The variability in this case (apart from varying the input model) is the type of database technology that a resulting applications uses. There are two variants, as shown in the feature

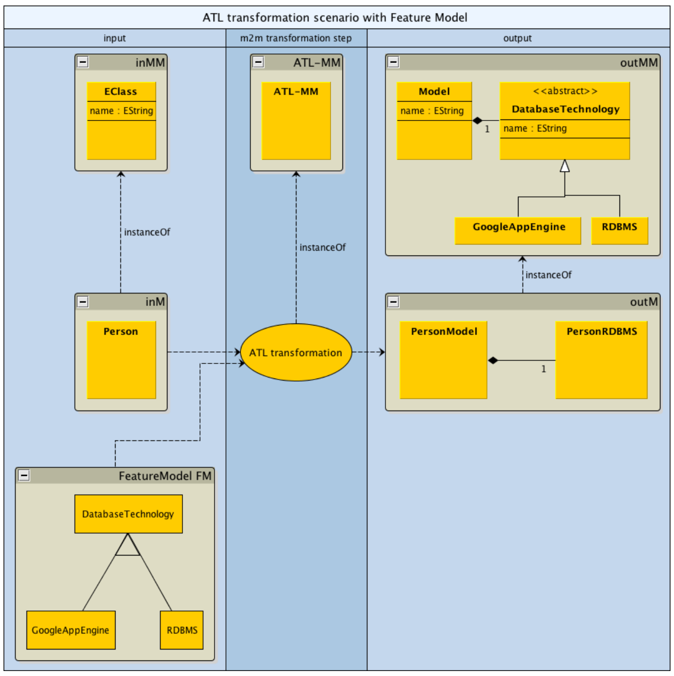

Fig. 1. Typical model-to-model transformation scenario, using ATL and a feature model FM.

model *FM*: a relational database denoted by *RDBMS*, and an object database *GoogleAppEngine*. The meta model *outMM* shows that each *Model* element has a *DatabaseTechnology* element aggregated.

In the feature model *FM* of this example, the *RDBMS* variant is selected, so the resulting model *outM* has an instance of an *RDBMS* realization artefact for each *EClass* source element.

The next section shows how implementing this using conventional ATL constructs raises some problems.

3.2 First attempt — Using Rule Inheritance

In ATL, rule inheritance could be used for guiding the feature assembly process in the above example, as shown (partially) in Listing 1.1.

```
rule EClass2Model {
   from
       a:   inMM ! EClass
   to
      model : outMM ! Model (name <- a.name+'Model'),
      db : outMM ! DatabaseTechnology
}

rule EClass2Model_RDBMS extends EClass2Model {
   from
       a:   inMM ! EClass (true)
   to
      db : outMM ! RDBMS
}

rule EClass2Model_GoogleAppEngine extends EClass2Model {
   from
        a:   inMM ! EClass (false)
   to
       db : outMM ! GoogleAppEngine
}
```

Listing 1.1. Using rule inheritance in normal ATL to deal with variability management is insufficient.

Note that the *from*-clause in the rule *EClass2Model* matches on *inMM!EClass*, stating that the instantiation of the feature realization artefact should be done *for each EClass* instance. We use rule inheritance to instantiate a specific variant.

There is a sub rule for each feature, both specializing the rule *EClass2Model* with a specific database technology. According to the selected feature, here simply switched by *true* if selected, and *false* otherwise, the correct sub rule is called, and the super rule's content is inherited.

This situation shows that feature realizations often have to be integrated into the common base, *according to some rationale*, in this case for each *EClass*. Therefore this case uses rule inheritance, which seems a necessary strategy, but the problem with this is that ATL does not support matching a single source model element by more than one rule. So, what if there are more features, apart from database technology, that also have to be instantiated *for each EClass*? Or what if both features are allowed, because they are, for instance, both optional (ie. both *from*-clauses are set to *true*)? Then, multiple rules should be called for a single source element, which is not supported.

Note that also, the true/false must be switched manually here, or helpers have to be implemented that navigate the feature model and check if a particular feature is selected.

3.3 Second Attempt — Matching on the Feature Model Elements

As an alternative, a rule which matches on elements from the feature model FM could be used, instead of matching on the input meta model. This usually avoids having different rules with the same *from*-clause. But doing something 'for each EClass' is now not possible, which was specifically intended. One could use imperative code, but this is not recommended, nor is it elegant.

Thus, there is not a good way to let features guide a model transformation, at least not without resorting to imperative ATL code. We provide a more elegant, easy to use solution, by extending ATL with the concept of variability rules, as the next section will explain.

4 The Concept of Variability Rules

This section describes the concept of variability rules. The syntax and semantics are not much different from normal ATL rules.

4.1 Syntax

A variability rule looks like a normal rule, prepended with the *variability* keyword:

```
variability rule RDBMS configures EClass2Model {
    from
```

```
        a : inMM ! EClass
    to
        db : outMM ! RDBMS (
            name <- a.name+'RDBMS'
        )
}
```

Listing 1.2. Extended concrete syntax of ATL with variability rules.

Observing the example from Listing 1.2, the syntax is a bit different than normal rules. In the first line, 'RDBMS' refers to the feature from the feature model. Secondly, 'EClass2Model' refers to a normal rule. This variability rule is said to 'configure' a normal rule. Finally, the *from*-clause is identical to the one in ECLass2Model.

4.2 Semantics

The semantics is as follows: i) a variability rule 'configures' a normal rule, specializing its implementation, just like rule inheritance, ii) A variability rule is executed for each match, but if and only if the corresponding feature is selected in the feature model, iii) multiple variability rules can have the same *from*-clause, iv) multiple variability rules can configure the same normal rule.

This way, specific feature realization artefacts can be instantiated from the target meta model. Also, multiple rules that match on a single source model element can be called. This solution is very comparable to a conventional declarative mapping in a normal rule, the implicit execution order property still holds.

Furthermore, the concern of selecting features is now cleanly separated. It is the feature model editor's responsibility to facilitate/constraint the selection. Variability rules act as a template; a modular piece of code that is not part of an (explicitly defined) imperative process, used for instantiating a particular feature. Thus, a developer can write one or more variability rules for each feature, without having to worry about concerns like selection constraints or execution order.The developer can just declaratively define what should be instantiated if a certain feature is selected.

The next section shows how variability rules are implemented.

5 Implementing Variability Rules using a Higher-order Transformation

The variability rules are implemented by a higher-order transformation (HOT), as shown in Figure 2. The HOT is created as follows: i) The ATL meta model is extended with a *VariabilityRule* EClass, yielding the ATL' meta model. ii) The concrete syntax of ATL is extended and a corresponding editor is created. iii) A HOT is developed in ATL which transforms an ATL' model into an ATL model. The HOT will be a used as preprocessing step in a transformation sequence. The resulting ATL model is subsequently executed, to yield the aimed result.

The compilation step of the HOT works as follows. First all the *to*-clauses of variability rules whose feature is selected are gathered, and these are grouped by the rule they configure. Then, these consolidated *to*-clauses are added to the normal rule that is configured. This results in a set of normal rules whose *to*-clauses only contain the *to*-clauses from the relevant variability rules, effectively resulting in a transformation that transforms a source model into a target model containing the selected features. In other words, the result is semantically equivalent to executing multiple rules that match on the same source model element.

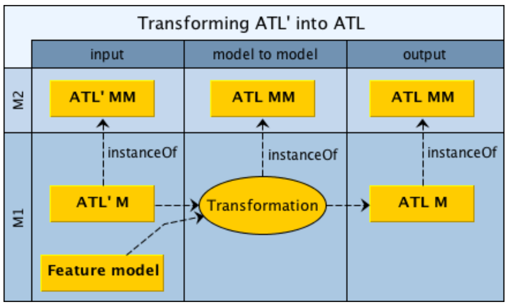

Fig. 2. A feature model together with a variability rule enriched ATL file (ATL' M) are transformed into a native ATL file with only the selected features transformed into normal ATL rules.

Note that since there is no de-facto standard for feature (meta) models, we used our own, which is not shown here. If one would want to use its own, a custom version of the (simple) ATL helper function $isSelected(f : Feature) : Boolean$ (residing in the HOT) should be included in the higher-order transformation.

6 A Different Type of Variability — Source Model Dependent Variability

During this research we identified a different type of variability, which still cannot be solved using the version of variability rules discussed so far, but appeared very relevant. This type is called *source model dependent variability*, whereas the variability type discussed above is called *source model independent variability*. The name explains what it means: there is a type of variability that also depends on elements from the source model of the transformation. In other words, whether or not a variability rule should be executed not only depends on the feature model, but also on elements from the source model.

At this stage, the implementation of variability rules cannot handle source model dependent variability, because there is no means to define a mapping between a feature from the feature model and a particular (instantiated) class. Therefore the syntax and semantics of variability rules are extended further, as this section shows.

To make things clear, we first give an example of this variability type.

6.1 Example of Source Model Dependent Variability

Consider the input model in Figure 3. The model has a *Person* class, which has a reference to a *Task* class. In our real-life transformation sequence case, for each class a MVC pattern is created. This means that there is a View class generated for a Person *and* for a *Task*. At this point, assume that there are two types of views: a *Grid* and a *Tree* view, both displaying the data in their own way. It could very well be that the *Person* should have a *Grid* while a *Task* should have a *Tree* view. As can be seen, the transformation should only execute a variability rule if its matching element is annotated with *Grid* or *Tree*.

The next section explains how the concrete syntax is extended, to allow defining this mapping between source model elements and features.

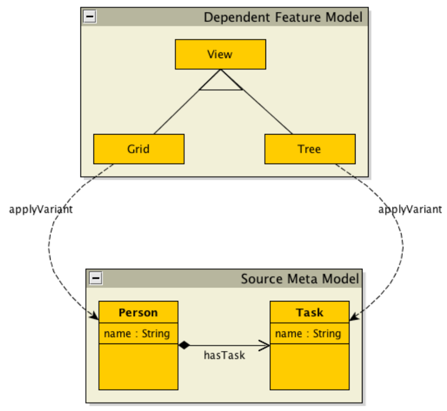

Fig. 3. In the case of source model dependent variability, features from the feature model are also related to source model elements. This means that annotation is needed in the source model, to provide enough information for the transformation to run.

6.2 Extending Syntax of Variability Rules Further

The variability rule shown in Listing 1.3 shows a small syntax change. Next to referring to a base rule that is configured, also a variable from this base rule's *to*-clause is specified, in this case *c* (lines 16 and 26). This target element is the point where the elements from the *to*-clause of the variability rules will be bound.

This syntax provides enough information for the HOT to compile this into the code shown in Listing 1.4.

```
 1 --Base rule
 2 rule EClass2Model {
 3         from
 4             a: inMM ! EClass
 5         to
 6            c: outMM ! Controller (
 7                ...
 8                -- After running the HOT, the views are
 9                -- bound here, since the variability
10                -- rules specified this
11            )
12
13 }
14
15 variability rule GridView configures
16                EClass2Model.c {
17     from
18         a : inMM ! EClass
19     to
20         views: outMM ! Grid (
21            name <- a.name+'TreeView'
22         )
23 }
24
```

```
variability rule TreeView configures
             EClass2Model.c {
    from
        a : inMM ! EClass
    to
        views: outMM ! Tree (
         name <- a.name+'TreeView'
        )
}
```

Listing 1.3. Extended concrete syntax of ATL with variability rules again to handle dependent variability.

6.3 Resulting Code after Running the HOT

After the HOT is run, the resulting ATL model will look as shown in Listing 1.4. Every dependent variability rule will be compiled into a lazy rule. This lazy rule is called by the rule that is configured. The point from which it is called is, in this case, the *Controller* with variable *c*, as the*configures*-clause stated. The lazy rule call is part of an if-then-else expression, where the rule is only called if the source model element $isAnnotatedWith$ a certain feature.

This if-then-else statement is dynamic (ie. not hardcoded) from the perspective of the compiled ATL model, because it is the nature of the mapping. Hardcoding the binding is not possible, since it can differ per EClass in this case. Thus, the binding needs to be postponed to run-time of the ATL model that is resulted from the HOT. This was done so that the HOT (as shown in Figure 2) does not require the input meta model as an input; just the feature model and the ATL' model. This makes the HOT more modular, and reusable.

The reader might have noticed that this approach allows for illegal feature selections, if the annotations in the input meta model are wrongly put. To see this, assume that the class *Person* has two annotations: $<<$ *Grid* $>>$ and $<<$ *Tree* $>>$. But the feature model states that these are *alternatives*, so only one is allowed. This check can no longer be the responsibility of the feature model editor, as was the case with source model independent variability. In fact, the semantics of what is allowed and what not can be interpreted in various ways: (i) a *Grid* and a *Tree* are alternative *per* matching element (in this case $inMM!EClass$), or (ii) model-wide, ie. in the whole model there can be either $GridxorTree$ views. In our approach situation (i) seems appropriate, and this is what we will implement. This validity check is currently work-in-progress.

```
--Base rule, after the HOT has run
rule EClass2Model {
        from
            a: inMM ! EClass
        to
           c: outMM ! Controller (
                ...
         views <-if
                    thisModule.isAnnotatedWith(a, 'GridView')
                    then
                        thisModule.Grid(a)
                    else
                        Sequence {}
                    endif,
         views <-if
```

```
                    thisModule.isAnnotatedWith(a, 'TreeView')
                    then
                         thisModule.Tree(a)
                    else
                        Sequence {}
                    endif,
             )

}

-- The variability rules are
-- compiled into lazy rules
lazy rule Grid {
    from
         a : inMM ! EClass
    to
         views: outMM ! GridView (
            name <- a.name+'TreeView'
         )
}

lazy rule Tree {
    from
         a : inMM ! EClass
    to
         views: outMM ! TreeView (
          name <- a.name+'TreeView'
         )
}
```

Listing 1.4. The HOT compiles the variability rules to lazy rules and the bindings are done in the specified meta element in the base rule.

The method of annotating the model is also work-in-progress. Our approach will be similar to [6], where a separate annotation model is used. A very simple concrete syntax is currently being developed, that can be compared with Cascading Style Sheets (CSS) [7]. CSS is widely-adopted to annotate HTML elements with style features. In our case, we use it to annotated source model elements with features from the feature model. Having a separate annotation model keeps the source model from being cluttered by annotations.

With this syntax, one is still able to create elements according to a rationale, in this case for each EClass. In other words, one can still apply the rationale of matching elements, like that is used when defining ordinary ATL rules.

7 Open Issues

Two things are currently work-in-progress and have not been solved completely. First, there is not yet a check that validates an annotated model when it comes to the selection of source model dependent features. As discussed, this responsibility cannot be outsourced to the feature model editor. Second, the annotation model implementation is not yet complete.

8 Related Work

Our focus is not to give a complete overview of the related work, but to focus on handling variability in step-wise model refinement, therefore we explain different related approaches briefly. The approach of [5] is to weave variability capabilities into meta models using aspects. This means that the variability is not modeled in a separate model, which differs from our approach, where we separate this concern. In [3], also aspects are used, but this approach *does* use a separate variability model. This approach provides variability management mechanisms in transformation languages where aspects are used in the model transformation language XTend to implement variability management. Our implementation approach (using a HOT) differs from their aspect-oriented implementation. Furthermore, their model transformation language is different. The authors of [1] provide an algebraic framework for managing variabilities, which is fundamentally different. The approach in [6] describes a method of annotating the input model which drives the transformation, using a separate annotation model. This approach is combined with our approach in a solution for managing *source model dependent* variability as described in the previous section. The approach from [2] also uses specialized types of rules to manage variability. However, their rules are implemented using native ATL [4] language constructs, or put differently, their approach is more a software pattern for managing variability. The usage of this pattern means that there is imperative code involved, which our approach avoids.

9 Conclusion

In this paper we addressed the problem of handling variability in a product line that is implemented using a transformation sequence. We proposed a way for handling variability in the model transformation language ATL. We have shown that normal ATL rules are able to handle variability, but not without problems (ie. imperative code, changing the transformation after the feature selection has changed). Therefore the approach was to create variability management means by introducing a new type of rule called variability rules. The motivation for adding a new type of rule was the observation that rules are purpose-built to do mappings, thus they are a good candidate to do the required mapping between feature model and meta models. Also, rules provide mature mechanisms for matching, querying, and instantiating meta model elements. Our concept achieved at least three things. First, the implicit execution order is maintained, yielding the same modularity properties as normal rules. Second, all the advanced mapping features of normal ATL can be used, as well as the rationale that is behind conventional rules; the developer can use the same rationale (of matching elements) for variability rules as for normal rules. Third, because it is a higher-order transformation, the ATL engine is not modified, so there is not a new ATL version that has to be installed (and maintained!). Instead, one could just chain the HOT into the transformation sequence. We have shown ways to solve two types of variability: source model independent, and source model dependent variability. For the latter, there are some open issues left. Firstly, there is not yet a way to check if a selection is valid. This check can no longer be outsourced to the feature model editor because it needs information from the source model. Secondly, the dependent case needs annotation on the source model of the transformation. This mapping will be done using a separate annotation model. This has not yet been implemented fully.

Finally, we are investigating cases of variabilities where *parts* of rules are affected, like *to*-clauses or bindings.

References

1. Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement, 2003.
2. K. Garces, C. Parra, H. Arboleda, A. Yie, and R. Casallas. Variability management in a model-driven software product line. In *Avances en Sistemas e Informática*, volume 4 No. 2, pages 3–12, Sept 2007.
3. I. Groher and M. Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. *Lecture Notes in Computer Science*, 5560:111–+, 2009.
4. F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.
5. Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. Weaving variability into domain metamodels. *Model Driven Engineering Languages and Systems*, pages 690–705, 2009.
6. Juan Manuel Vara, Veronica Andrea Bollati, Belén Vela, and Esperanza Marcos. Leveraging model transformations by means of annotation models. In F. Jouault, editor, *MtATL*, pages 96–102, 2009.
7. w3 consortium. Cascading style sheets, http://www.w3.org/style/css/.