

Blackbox Composition of Model Transformations using Domain-Specific Modelling Languages

Dennis Wagelaar*

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be

Abstract. In Model-Driven Engineering, multiple model transformations often have to be composed to produce a common result. One composition approach is to compose rules from existing transformations into one transformation, which generally requires the transformations to be written in the same language. This paper focuses on chaining model transformations together by having them pass models to each other, thus treating model transformations as blackbox components. Similarly to blackbox software components, not all model transformations can be combined. There are also constraints to the order in which model transformations have to be used. In the domain of component composition, Domain-Specific Modelling Languages (DSMLs) have been used to drive verification of compositions and automatic generation of composition code. We propose to apply DSML techniques for the composition of model transformations.

1 Introduction

There are many scenarios in Model-Driven Engineering in which a number of model transformations have to be composed in order to produce a common result. These scenarios range from one transformation language applied to one meta-model to many transformation languages applied to multiple meta-models. Each of these scenarios has different possibilities and limitations for composition. In some scenarios, it is possible to compose rules from existing transformations into one transformation. This generally requires the transformation rules to be written in the same language. This paper focuses on chaining model transformations together by having them pass models to each other, thus treating model transformations as blackbox components. This approach should be applicable in all scenarios and can be combined with the first approach.

Similarly to blackbox software components, not all model transformations can be combined. For the model transformations that can be combined, there are often constraints to the order in which they have to be applied. In the domain of component composition, Domain-Specific Languages (DSLs) have been used

* The author's work is part of the CoDAMoS project, which is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders)

to drive verification of compositions and automatic generation of composition code [1]. In the case of a Domain-Specific Modelling Language (DSML) [2][3], the configuration rules and constraints are encoded in the meta-model of the DSML.

A DSL (or DSML) typically contains language constructs that refer to a particular application domain (e.g. web registration systems or instant messaging), whereas general-purpose languages use far more abstract language constructs and force the programs/models to fill in the concrete semantics.

We propose to apply DSML techniques, such as meta-modelling and transformation, for the verification of model transformation compositions and automatic generation of composition code. In the remainder of this paper, we will illustrate the use of meta-models and transformations for expressing composition rules and for generating build scripts that implement the composition.

2 Expressing Composition Rules

Using the Eclipse Modeling Framework [4], one can define a meta-model for a domain-specific composition language. As an example domain, a set of Java-specific refinement transformations is used. These transformations apply local refinements to model written in the Unified Modeling Language (UML) version 1.4 and use Java as an Action Language. The kinds of refinement include refining associations to attributes, attributes to accessor methods, observer stereotypes to a Java implementation, applet stereotypes to a Java implementation and a few others¹. Fig. 1 shows (part of) the meta-model for the “Java-refinement-specific” composition language.

The meta-model enforces the order in which the transformations must occur: each `RefinementConfiguration` must start with an `AssociationAttributesRefinement`. `AssociationAttributesRefinement` is an abstract meta-class, which has two concrete subclasses: `AssociationAttributes` and `Java2AssociationAttributes`. Note that the meta-model also enforces one to choose exactly one `AssociationAttributesRefinement`, since the *first* attribute of `RefinementConfiguration` has a multiplicity of 1. The meta-model then goes on by enforcing that the *next* transformation after `AssociationAttributesRefinement` should be an `AccessorsRefinement`. This is again an abstract class with two concrete subclasses: `Accessors` and `Java2Accessors`. After `AccessorsRefinement` come `ObserverRefinement`, `AppletRefinement` and other that are no longer shown.

Note that there is another rule to the composition of `AssociationAttributesRefinement` and `AccessorsRefinement`: if one chooses to use `AssociationAttributes`, one has to choose `Accessors`. Alternatively, if one chooses to use `Java2AssociationAttributes`, one has to choose `Java2Accessors`. This is because both transformations use Java collection types to implement multiple value attributes and its accessor methods. The Java types used in both transformations have to be the same. In order to express this in the meta-model, the *next* attribute

¹ The transformations themselves can be found at <http://ssel.vub.ac.be/ssel/research:mdd:casestudies>

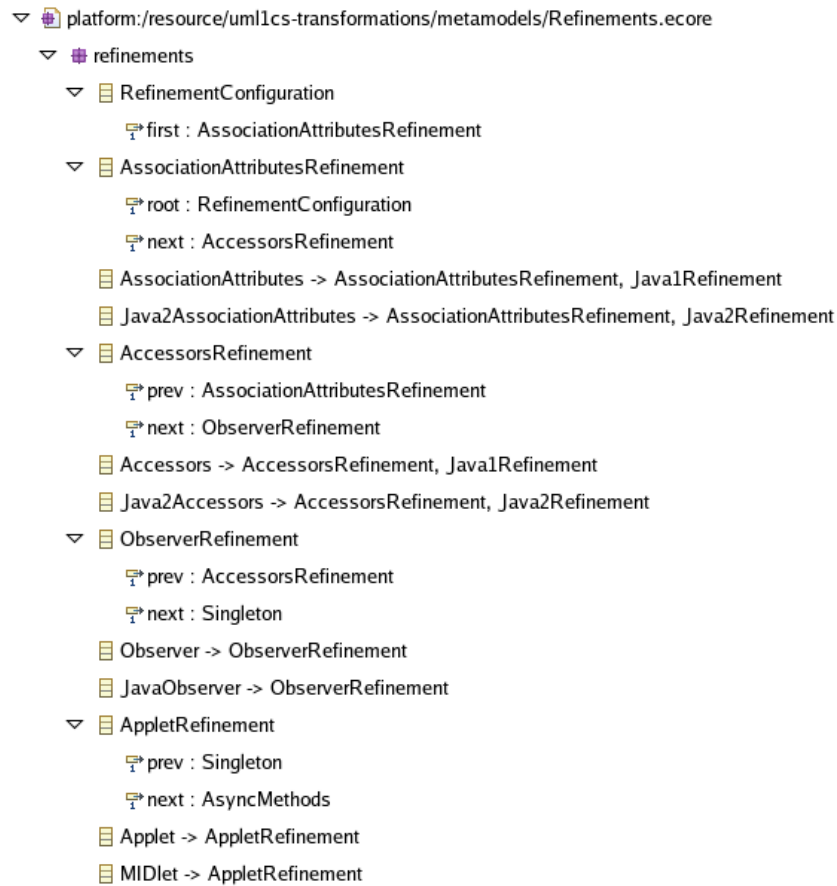


Fig. 1. Part of the EMF meta-model of the refinement model transformations

of `AssociationAttributesRefinement` would have to be pushed down to its subclasses and point to the correct `AccessorsRefinement` subclass. While this can still be done relatively efficiently for transformations that reside next to each other in the execution order, it would have been very cumbersome if `AccessorsRefinement` came after `AppletRefinement`, for example. It also tangles the order enforcing rules with the Java type consistency rules. In our example meta-model, we've chosen to introduce two new abstract meta-classes, `Java1Refinement` and `Java2Refinement`, which can be used in a model transformation that checks type consistency, such as the following model transformation written in ATL [5]:

```
rule Java1Java2 {
  from s : DSL!Java1Refinement (
    DSL!Java2Refinement.allInstances()->notEmpty()
  ) to t : REPORT!Error mapsTo s (
    message <- 'Java1 and Java2 refinements cannot be combined'
  )
}
```

Note that this transformation rule does not need to know about any concrete transformations or the order in which they must be executed. It just checks for combinations of Java1Refinement and Java2Refinement instances.

3 Generating Composition Code

In addition to validation of transformation compositions, it is also possible to generate build scripts from transformation compositions. Since these transformation compositions are again models, adhering to the rules expressed in the meta-model (see Fig. 1), we can use model transformations to generate the build scripts. Below is a partial example ATL transformation that generates code for an Ant build.xml file:

```

query GenerateBuildFile = DSL!RefinementConfiguration->allInstances()
->collect(e|e->toString()->writeTo('build.xml'));

helper context DSL!RefinementConfiguration def : toString() : String =
  thisModule->header() +
  self.first->toString(input) +
  thisModule->footer();

helper def : header() : String =
  '<?xml version="1.0" encoding="UTF-8"?>\n' +
  '<project name="refinement" default="transform">\n' +
  '  <target name="transform" depends="clean">\n' +
  '    <atl>\n';

helper def : footer() : String =
  '  </target>\n' +
  '</project>';

helper context String def : atlRefineCommand(input : String, merge : String)
: String =
  '<!-- ' + self + ' -->\n' +
  '<arg line="--trans ${transf.uri}' + self + '.asm"/>\n' +
  '<arg line="--in IN=' + input + '.ecore UML=${mmodel.uml} EMF"/>\n' +
  '<arg line="--in MERGE=' + merge + '.ecore UML=${mmodel.uml} EMF"/>\n' +
  '<arg line="--out OUT=' + input + '.r.ecore UML=${mmodel.uml} EMF"/>\n' +
  '<arg line="--lib Java=${lib.java}"/>\n';

helper context DSL!AssociationAttributes def : toString(input : String)
: String = 'AssociationAttributes'->atlRefineCommand(
  input, '${rmodel.ocltypes}') + self.next->toString(input + 'r');

helper context DSL!Java2AssociationAttributes def : toString(input : String)
: String = 'Java2AssociationAttributes'->atlRefineCommand(
  input, '${rmodel.ocltypes}') + self.next->toString(input + 'r');

helper context DSL!Accessors def : toString(input : String) : String =
  'Accessors'->atlRefineCommand(input, '${rmodel.ocltypes}') +
  self.next->toString(input + 'r');

helper context DSL!Java2Accessors def : toString(input : String) : String =
  'Java2Accessors'->atlRefineCommand(input, '${rmodel.ocltypes}') +
  self.next->toString(input + 'r');
...

```

4 Conclusion

This paper has illustrated how DSML techniques can be used to perform black-box composition of model transformations. A meta-model is used to express the rules for composition, along with any additional model transformations for validation. Since the transformation compositions are models themselves, it is possible to use model transformations for generating build scripts. Such build scripts can be used to implement the transformation composition.

In our example, the execution order of the transformations, which is already encoded in the DSML meta-model using the *next* and *prev* attributes, is explicitly navigated for each meta-class in this build script generator. With the current meta-model used, this cannot be avoided. If we would have modelled the *next* and *prev* attributes in a general superclass in the meta-model, the build script generator could have used one general helper method to navigate from one transformation to the next. Doing this would however also remove the execution order rules from the meta-model itself and an additional validation transformation is necessary to check the ordering.

From the above observation, it seems that the DSML meta-model should be mainly used as an enabler for validation and generation transformations. By using abstract meta-classes to model a kind of rule, such as “execution order” or “type consistency”, external validation transformations can be used to check such rules while being oblivious of other rules. Similarly, build script generator transformations can be oblivious of the composition rules and only needs to navigate through the composition model.

References

1. Deursen, A.v., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology* **10** (2002) 1–17
2. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. *IEEE Computer* **34** (2001) 44–51
3. Tolvanen, J.P., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003), Anaheim, CA, USA, ACM Press (2003) 92–93
4. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse Series. Addison Wesley Professional (2003)
5. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica. (2005)