Advanced Execution Modes for Model Transformation Languages

Salvador Martínez and Jordi Cabot

AtlanMod, INRIA & École des Mines de Nantes, France
{salvador.martinez_perez, jordi.cabot}@inria.fr

1 Introduction

Model transformations are a core activity in Model Driven Engineering. As a consequence, in the last years the MDE community has proposed several languages and tools to define and execute model transformations. Thanks to these efforts, we have now a relatively stable set of core technologies designers have started to use in their day to day work to manipulate models. This has confirmed the usefulness of these model transformation techniques but at the same time has raised new challenges, mainly linked to scalability and efficiency problems, that we must respond to in order to make sure model transformation becomes a technology mature enough to be used in real industrial scenarios. We believe the key innovation to solve these new challenges consists in providing different execution modes for the same model transformation, depending on its specific requirements and characteristics. This demo will present three new execution modes for the AtlanMod model-to-model Transformation Language (ATL) [2] addressing the mentioned challenges for the industrial adoption of model transformation. It is important to note that for the implementation of all these execution modes the syntax of the language does not need no be modified, which enables the reuse of existing transformations and the adoption by transformation developers.

2 Refining Mode

Some transformations aim only to perform slight modifications on the input model (i.e. the target model is identical to the source model with the exception of a set of changes that is relatively small compared to the overall model size). We refer to this special kind of transformations as refinement transformations. Languages designed to translate read-only input models towards write-only output models are not directly applicable to this kind of transformations. The refining mode [5] we present in this demo is aimed to solve this problem.

This execution mode implements an in-place strategy, e.g., the changes are directly performed on the source model without making any copy of the elements. The rules in the transformation only need to specify the values that have changed whereas all the other elements just remain untouched. The transformations in this mode are performed in two steps. In the first step, the transformation engine executes the rules which, as a result, produce a set of changes that are temporally

Listing 1.2. Private2Public.atl

```
1
2  rule PrivateAttribute {
3    from
4      s : ClassDiagram!Attribute (s.isPrivate and
5        s.owner.op->exists(o|o.name = 'get' +
6        s.name.toUpperCase() and o.returnType = s.type)
7        )
8    to
9      t : ClassDiagram ! Attribute (
10       isPrivate <- false
11     )
12 }
13 rule DeleteOperation {
14   from
15     s : ClassDiagram!Operation (s.owner.attr
16       ->exists(a|a.name = s.name.toUpperCase().substring(3, s.name->size
             ↪())
17         and a.isPrivate))
18   to
19     drop
20 }
```

stored in a differences model. In the second step, this set changes are applied directly on the source model.

From the semantic point of view, it is worth to note that a delete change can easily generate a conflict with another modification. This happens because we chose to provide delete with a cascade-delete semantics for composition associations (i.e. the deletion of a container triggers the deletion of the contained elements). For this reason, the second step in the transformation that is in charge of applying the changes is implemented by first applying all the creation changes, then all the modification changes, and finally all the deletion changes.

Althoug, as stated in the introduction, the syntax does not need to be modified for the implementation of new execution modes in ATL, in this particular case we considered that introducing new language constructs would ease the development of transformations. In the following examples we show these new concrete syntax elements.

First, the refining mode is selected by simply replacing the from keyword by the new *REFINING* keyword in the transformation header. Listing 1.1 shows how the header of a refactoring transformation on a ClassDiagram metamodel may look like.

Listing 1.1. ATL header of a Class refactoring transformation

```
1 module Public2PrivateAndGetter;
2 create OUT : ClassDiagram refining IN : ClassDiagram;
```

The listing 1.2 shows an ATL refining transformation that transforms private attributes to public attributes and delete the corresponding getters unig the new provided keyword *DROP*

3 Incremental Mode

Up to now, the execution of ATL transformations has always followed a two- step algorithm: 1) matching all rules, 2) applying all matched rules. This algorithm does not support incremental execution, meaning that even if only small part of the source model is updated, the whole transformation must be executed again to regenerate the complete target model, which brings efficiency problems for large models. A similar situation happens in all model-to-model transformation languages. The new ATL incremental mode [3] provides a new execution algorithm where changes in a source model are incrementally propagated to the target model, minimizing the excerpts of the target model that must be recomputed.

This new execution mode rely on two runtime mechanisms:

1. **Tracking dependencies of OCL[1] expressions** (Note that, in ATL, all expressions are expressed in OCL). During the evaluation of OCL expressions, we collect dependency information. When a change takes place in the source model, we thus know which OCL expressions are impacted (i.e., may have a new value as a consequence of the change).
2. **Controlling individual rule execution**. The connection between source models and rules happens at the OCL expression level. Therefore, the information gathered in the first mechanism is used in order to control rule execution. In standard execution mode, all rules are matched and applied on whole source models. Instead, we enable precise control over the matching and application of each rule for each source element. Then, we just need to actually trigger the rules in response to changes in the source models.

Once we have these two mechanism available we only need to track source model change events. Concretely, the change events that can occur in the source model and its impact in the transformation are as follow:

- Element creation has an impact over rule matching. If the created element is also added to a specific property (e.g., an Attribute) that is created and added to a Class separate property change events will be received. Therefore, element creation only impacts rule matching.
- Element deletion has an impact over rule matching. Similarly to element creation, related changes in specific property values will trigger property changes.
- Property change is sent each time a property of an element is changed. Depending on which OCL expression depends on the specific property for the specific source element, the impact may be on: 1) rule matching if a filter expression depends on it, or 2) a binding if a binding expression depends on it. Actually, several bindings or rules, or both bindings or rules may be impacted by the change of a single property on a single element.

4 Lazy Mode

Many times the user is only interested in accessing a small part of the whole target model. In those situations generating the full target model may be a waste

of time and resources. To improve this situation, the lazy mode contributes a new lazy execution algorithm for ATL where elements of the target model are generated on demand only when (and if) they are accessed.

The implementation of the algorithm to allow lazy generation is based in the following mechanism:

- To initiate the lazy generation process, consumption requests on the target model have to be tracked. This requires extending the model navigation mechanism the consumer uses, to intercept the requests and activate a corresponding generation in the transformation engine. If this adaptation can be performed in a transparent way, the client system will not notice that the model it is handling is lazily built. For performing transparent lazy access with EMF[4], we have reimplemented part of its API in order to trigger calls to the transformation engine operations.
- The transformation engine has been modified to provide the means to launch the computation of a single model element (as in incremental mode) or a single property of the target model.
- Finally, the lazy engine can keep track of the status of the partial transformation, and use it as a context for the execution of new computations. The stored context is exploited by the lazy system to avoid recomputations. In transformation systems this context usually includes trace links that map elements in the target model with their corresponding sources. We make use of these traceability links to perform live lazy transformations (the trace information should be serialized in order to perform offline lazy transformations)

Once modified the engine to provide this infrastructure, our lazy generation algorithm works in three steps:

1. the consumer requests a new target element, and the call gets intercepted by the navigation interface of the target model;
2. the navigation interface (e.g. the lazy model) requests to the engine the generation of the single requested property or element;
3. the engine determines the computations to activate, based on the current status of the transformation.

References

1. Object Constraint Language (OCL), OMG available specification, version 2.0, 2006.
2. F. Jouault and I. Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138, 2005.
3. F. Jouault and M. Tisi. Towards Incremental Execution of ATL Transformations. In L. Tratt and M. Gogolla, editors, *ICMT2010 - Intl. Conference on Model Transformation*, volume 6142 of *LNCS*, pages 123–137, Malaga, Espagne, 2010.
4. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (The Eclipse Series)*. Addison-Wesley Professional, 2008.
5. M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Refining Models with Rule-based Model Transformations. Rapport de recherche RR-7582, INRIA, Mar. 2011.