Two Basic Correctness Properties for ATL Transformations: Executability and Coverage

Elena Planas[1], Jordi Cabot[2], and Cristina Gómez[3]

[1] Universitat Oberta de Catalunya (Spain), eplanash@uoc.edu
[2] École des Mines de Nantes - INRIA (France), jordi.cabot@inria.fr
[3] Universitat Politècnica de Catalunya (Spain), cristina@essi.upc.edu

Abstract. Model transformations play a cornerstone role with the emergence of Model Driven Engineering (MDE), where models are transformed from higher to lower levels of abstraction. Unfortunately, a quick and easy way to check the correctness of model transformations is still missing, which compromises their quality (and in turn, the quality of the target models generated from them).
In this paper we propose a lightweight and efficient method that performs a static analysis of the ATL rules with respect to two correctness properties we define: (1) *weak executability*, which determines if there is some scenario in which an ATL rule can be safely applied without breaking the target metamodel integrity constraints; and (2) *coverage*, which ensures a set of ATL rules allow addressing all elements of the source and target metamodels. In both cases, our method returns meaningful feedback that helps repairing the possible detected inconsistencies.

1 Introduction

Model transformations play a cornerstone role with the emergence of Model Driven Engineering (MDE), where models are transformed from higher to lower levels of abstraction. In consequence, the quality of the whole transformation process strongly depends on the quality of the model transformations. However, even though there is a wide set of research proposals focused on model transformations, there is still further work to be done regarding its analysis and verification. In particular, there is a lack of efficient methods to analyze rule correctness with respect to the source and target metamodels.

In order to alleviate this situation, we first define two basic correctness properties of model transformation rules. First property, *weak executability of a rule*, analyzes whether a rule may be safely applied without breaking the target metamodel integrity constraints. Second property, *covering of a rule set*, analyzes whether a set of rules allow addressing all elements of the source and target metamodels.

We also propose a lightweight analysis method to check these properties on rules expressed in ATL language [1]. In order to improve its efficiency, our method works at design time without any need to execute the rules. If the checked

property is not satisfied, the method returns meaningful feedback suggesting possible corrections to repair the detected errors.

Paper organization. The rest of the paper is structured as follows. Section 2 introduces a running example that will be used in the rest of the paper. Sections 3 and 4 specify the *executability* and *coverage* properties and provide a method to check them. Finally, section 5 discusses the related work and section 6 presents our conclusions and further work.

2 Running Example

This section introduces briefly some preliminary concepts on ATL model transformations and presents our running example, which describes a simple transformation of a Person model (source model) to a Student model (target model).

2.1 Metamodels

ATL rules describe the transformation from a source model (which conforms to a source metamodel) to a target model (which conforms to a target metamodel) by relating its metamodels. The source and target metamodels are described in the Ecore language (which allows to formally define its structure).

Example. The Person metamodel (`PersonMM`) (see Fig. 1, left) consists of people having a name, surname, age and college name. Besides, people may know other people (in a symmetric way). The Student metamodel (`StudentMM`) (see Fig. 1, right) consists of students having a full name. Students study at a college and they are enrolled in at least one subject.

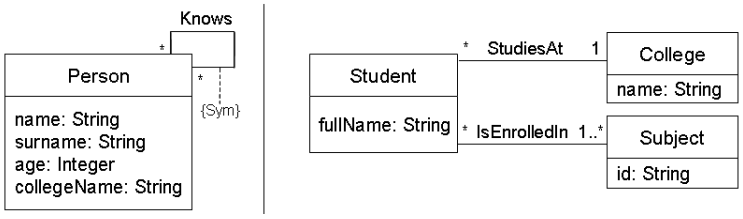

Fig. 1. `PersonMM` (source metamodel) and `StudentMM` (target metamodel).

2.2 ATL Rules Specification

The presented work only addresses ATL matched rules, that is, declarative rules that define how the source model elements are matched and navigated to create and initialize the target model elements. Although our method could be extended in order to address other kinds of ATL rules.

An ATL rule is introduced by the keyword "rule" followed by the rule's name. The source pattern is used to declare which element type of the source model has to be transformed. It consists of the keyword "from", a source variable declaration and optionally a filter (that is, an OCL expression restricting the

rule to elements of the source model that satisfy certain constraints). The target pattern is used to declare which element(s) of the target model the source pattern has to be transformed to. It starts with the keyword "to" and consists of a variable declaration and a sequence of bindings (assignments).

Example. We define a simple rule ("Person2Student") to transform people into students. For each Person instance, a Student instance has to be created. The full name of a Student has to be set linking together the name and surname of the Person (using a helper). A College instance is also created and related to the new student.

The corresponding ATL code is the following:

```
module Person2Student;
create OUT: StudentMM from IN: PersonMM;

helper context PersonMM!Person def:
  getFullName(): String = self.name + ' ' + self.surname;

rule Person2Student {
 from p: PersonMM!Person
 to s: StudentMM!Student (
       fullName ← p.getFullName(),
       college ← c
    ),
    c: StudentMM!College (
       name ← p.collegeName
    )
}
```

3 Executability of ATL rules

We consider an ATL rule r is *weakly executable* if it has a chance of being successfully executed. That is, if there is at least a given set of elements that matches with the source model for which the execution of the rule r generates a target model consistent with the target metamodel and its integrity constraints. Otherwise r is useless, as every time it is executed, an error arises because the target model violates some integrity constraints.

We define our *executability* property as *weak executability* since we do not require all executions of the rule on a matching source model to be successful, which could be defined as *strong executability*. Weak executability is a prerequisite for strong executability (the latter implies the former). Hence, designers can check first weak executability, which is simpler to verify, and then they can apply other techniques to determine the stronger property if necessary.

As an example, rule "Person2Student" is not weakly executable since, every time we create a new Student and we do not associate it to any subject, we reach an erroneous model where the minimum 1 cardinality of the "IsEnrolledIn" association is violated. Our method can report that, in order to create a new Student, we need to assign at least one subject within the same execution.

To determine if a rule is weakly executable we proceed by applying a two-step verification process (see Fig. 2). This process may be automated and integrated into a tool for editing ATL rules.

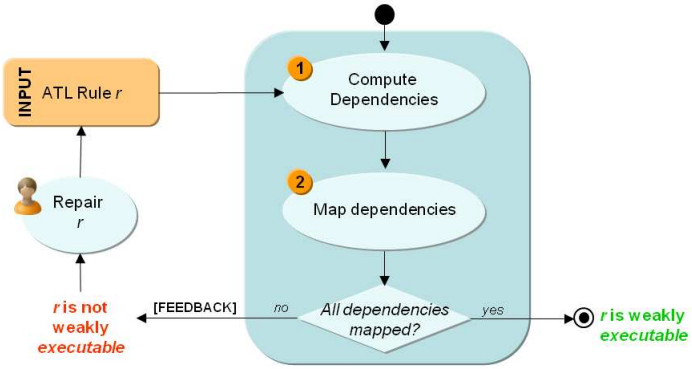

Fig. 2. Weak executability overview.

Step 1: Computing Dependencies. The executability of an ATL rule depends on the updates it performs. In particular, problems may arise when some update requires the presence of other updates within the same execution in order to reach a consistent model (i.e. a state that violates no constraint) after executing the rule. Therefore, to be executable, a rule r must satisfy all *dependencies* for every update in r. A dependency from an update $up_1$to another update up_2 expresses that up_2 must be included in all rules where up_1 appears to avoid violating the metamodel constraints. Dependencies for a specific update are drawn from the structure and constraints of the target metamodel and from the kind of effect the update performs (create an object, initialize an attribute or create a link).

Table 1 provides the rules to compute the dependencies for each kind of update. First column (*Rule update*) shows the description and the corresponding ATL code for several possible updates in the rule[1]. Second column (*Constraint*) states when the dependency must hold[2]. Finally, third column (*Required update*) shows the description and the corresponding ATL code of the required update to satisfy the constraint in the same row.

As an example, we discuss the first row of Table 1, which describes the required updates to ensure that an object creation does not violate a mandatory

[1] Note that the Table 1 only shows the creating object/link updates, given that initializing an attribute has not any dependency.

[2] Our method covers the following constraint types: *Mand(attr,cl)*, which expresses a mandatory constraint over an attribute *attr* of class *cl*, that is, the attribute *attr* must have at least one value; *Cmin(as,r)*, which defines the minimum cardinality constraint of an association *as*, expressing the minimum multiplicity of the member end (i.e. role) r of *as*; and *Sym(as)*, which expresses a symmetric constraint over a recursive association *as*, that guarantees that if o_1 is *as*-related to o_2, then o_2 is *as*-related to o_1.

Table 1. Dependencies for creating actions.

Rule Update		Constraint	Required Update	
Description	ATL code		Description	ATL code
Create object	to o:MM!cl	Mand(*attr*,*cl*)	Initialize attribute *attr*	o: ... attr ← value
		Cmin(*as*,*r*) ≠ 0	Create mandatory links	o: ... $r \leftarrow obj$ where *obj*→size≥ Cmin(*as*,*r*)
Create link	o: ... $r_1 \leftarrow obj$	Sym(*as*), where r_1 and r_2 are its member ends	Create the symmetric link	o: ... $r_2 \leftarrow o$

constraint of an attribute ($Mand(attr, cl)$). In order to avoid violate this constraint type, the attribute *attr* must be initialized within the same execution. Last colum of first row shows the necessary ATL code to satisfy this dependency.

Example. In the following the dependencies for rule "Person2Student" are shown:

Dependencies for rule "Person2Student"
Create an object of type Student requires:
(dep_1) Initialize its attribute "fullName"
(dep_2) Create one link of "StudiesAt" association
(dep_3) Create at least one link of "IsEnrolledIn" association
Create an object of type College requires:
(dep_4) Initialize its attribute "name"

Rule "Person2Student" has four dependencies. Creating an object of type Student requires update its mandatory attribute "fullName" (dep_1) and create mandatory links with associations "StudiesAt" and "IsEnrolledTo" (dep_2 and dep_3 respectively). Besides, creating an object of type College requires update its mandatory attribute "name" (dep_4).

Step 2: Mapping the dependencies. After computing the dependencies for a rule, we have to check if the required updates are satisfied by the rule itself. If all required updates are satisfied, the rule is classified as weak executable. If not, the rule is marked as non-weak executable and the corresponding corrective feedback is provided to the designer to help repairing the detected inconsistencies.

A required update is satisfied by a rule r if it can be mapped to r. An extract of required ATL code may be mapped to the rule's ATL code when the following conditions are satisfied: (1) both codes perform the same update (creating an element, initializing an attribute or creating a link); (2) the elements referenced by the code coincide (e.g. both initializations are of the same attribute); and (3) all instance-level variables of the required code can be bound to the variables in the rule (free variables introduced by the dependencies can be bound to any variable value in the rule's code, while fixed ones must have the same identifier in both codes).

Example. Table 2 shows the mapping for rule "Person2Student".

Dependencies dep_1, dep_2 and dep_4 of rule "Person2Student" are satisfied by the rule given that the required code appears in the rule (replacing the free variable "$value_1$" with the value "p.getFullName()", the free variable "obj_1" with the object "c", and the free variable "$value_2$" with the value "p.collegeName"). Otherwise, dep_3 is not satisfied given that the required code does not appear in the rule. Hence, we can conclude that this rule is not weak executable. Dependency dep_3 will be returned as feedback and should be added to the rule in order to make it weakly executable.

Table 2. Mapping for rule "Person2Student". Free variables are showed in *italics*. {sat} states that the dependency is satisfied, while {not sat} states the opposite.

Dependency	Required ATL code	Rule ATL code
dep_1 {sat}	out: ... fullName ← $value_1$	out: ... fullName ← p.getFullName()
dep_2 {sat}	out: ... college ← obj_1	out: ... college ← c
dep_3 {not sat}	out: ... subject ← obj_2	
dep_4 {sat}	out: ... name ← $value_2$	out: ... name ← p.collegeName

4 Coverage of ATL rule set

We consider a set of ATL rules is *covering* if it allows addressing all elements of the source and target metamodels. This property may be viewed regarding two perspectives:

- *Source-coverage*: We consider that a set of ATL rules is *source-covering* when all elements of the source metamodel may be navigated through the execution of these rules. Otherwise, there will be elements of the source metamodel with no relevance in the transformation. More formally, a rule set $set_{rl} = \{rl_1,\ldots,rl_n\}$ is source-covering when, for each element e in the source metamodel, there is at least one rule (rl_i) that navigates e. For instance, the set composed by the single rule "Person2Student" is not source-covering since rules to navigate "age" attribute and "Knows" association are not specified.
- *Target-coverage*: We consider that a set of ATL rules is *target-covering* when all elements of the target metamodel may be created and initialized through the execution of these rules. Otherwise, there will be elements of the target metamodel completely useless since no rules address their modification. More formally, a rule set $set_{rl} = \{rl_1,\ldots,rl_n\}$ is target-covering when, for each element e in the target metamodel, there is at least one rule (rl_i) that creates or initializes e. For instance, the set composed by the single rule "Person2Student" is neither target-covering since, for instance, rules to create objects of type "Subject" are not specified, forbidding users to create new subjects on the target model.

Even though not coverage does not indicate explicitly an error but only a warning, we feel this property is important to guarantee that no behavioral elements are missing in the rules.

To determine if a set of ATL rules is covering we propose applying a three step process (see Fig. 3). This process may be automated and integrated into a tool for editing ATL rules.

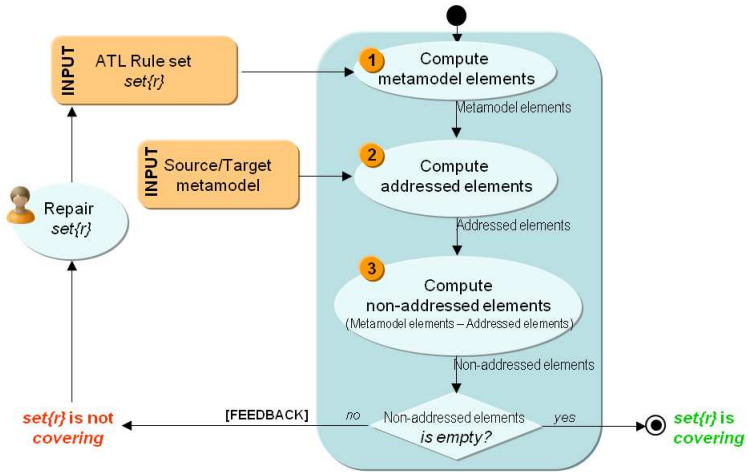

Fig. 3. Coverage overview.

Step 1: Computing metamodel elements. First step consists in determining the metamodel elements that should be addressed by the rules. In source metamodels, these elements are all those describe the metamodel (classes, attributes and associations). In target metamodels, these elements are all possible modifiable elements (non-abstract classes, classes which are not the supertype of a complete generalization set, non-derived attributes and non-derived associations). First column of tables 3 and 4 shows the metamodel elements for our running source and target metamodels respectively.

Step 2: Computing addressed elements. Second step consists in determining the addressed elements in the rule set. For source metamodels, navigated elements are those which appear in the "from" part of the rules or those which are navigated in the "to" part. For target metamodels, created and initialized elements are those which are created or initialized in the "to" part of the rules. Second column of tables 3 and 4 shows the addressed elements for our running source and target metamodels respectively.

Step 3: Computing non addressed elements. Last step consists in computing the difference between metamodel elements and addressed elements. This difference represent the elements that should be addressed but they are not treat

in any rule. They are returned as a feedback and their treat should be added in some rule to make the set covering. Three column of tables 3 and 4 shows the non-addressed elements for our running source and target metamodels respectively.

Example. Tables 3 and 4 show the source and target coverage of our running example.

Table 3. Source-coverage of our running example.

Metamodel elements (from source metamodel)	Addressed elements (navigated elements)	Non addressed elements (non-navigated elements)
Person class	Person class	
name attribute	name attribute	
surname attribute	surname attribute	
age attribute		age attribute
collegeName attribute	collegeName attribute	
Knows association		Knows association

The set composed by the single rule "Person2Student" is not source-covering since there is no rule that allows to navigate "age" attribute and "Knows" association of the source metamodel.

Table 4. Target-coverage of our running example.

Metamodel elements (from target metamodel)	Addressed elements (created/initialized elements)	Non addressed elements (non-created/initialized elements)
Student class	Student class	
fullName attribute	fullName attribute	
College class	College class	
name attribute	name attribute	
Subject class		Subject class
id attribute		id attribute
StudiesAt association	StudiesAt association	
IsEnrolledIn association		IsEnrolledIn association

The set composed by the single rule "Person2Student" is neither target-covering since there is no rule that allows to create objects of type "Subject", links of type "IsEnrolledIn" association neither initialize "id" attribute.

5 Related Work

Model transformation analysis and testing has been addressed in several works. For instance, [3] represents QVT [6] model transformations in Alloy (a first-order relational specification language) and simulates these transformations to verify its executability; [2] describes a visualization technique to analyze the metamodel coverage; [5] uses test cases for verifying the coverage of model transformations; and [4] present various model transformation testing approaches and demonstrate the challenges involved. Most of the methods above rely on the use of

simulation and testing techniques to analyze model transformations, compromising the efficiency of the approach.

Our approach performs a static analysis (no animation/simulation is required) and, thus, is more efficient. Our static reasoning techniques have been used in other fields. For instance, [7] uses similar static techniques to check the executability and coverage of UML operations, while [8] also uses these techniques to analyze the executability of graph transformation rules.

6 Conclusions and Further Work

We have defined two basic correctness properties of ATL matched rules: (1) *weak executability of a rule*, which studies whether a rule may be safely applied without breaking the target metamodel integrity constraints; and (2)*coverage of a rule set*, which studies whether a set of rules allow navigating all elements of the source metamodel and create/initialize all modifiable elements of the target metamodel. We also have proposed a lightweight method for the design-time analysis of these properties. The method provides feedback enabling the correction of the rules.

As a further work we would like to extend our method by addressing other types of ATL rules, verifying new properties like redundancies in ATL rules and by giving feedback regarding possible modifications in the metamodel (and not only in the rules themselves). We also plan to provide tool support, implementing the method and integrating it into a tool for ATL as Eclipse platform [1].

References

1. ATLAS INRIA Research Group. ATL technology: http://www.eclipse.org/atl/.
2. M. V. Amstel and M. van den Brand. Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In *ICMT*, volume 6707 of *LNCS*, pages 108–122. Springer, 2011.
3. K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In *MoDeVVa*, pages 47–56, 2007.
4. B. Baudry, T. Dinh-trong, J. marie Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon. Model Transformation Testing Challenges. In *IMDT*, 2006.
5. J. M. Küster and M. Abd-El-Razik. Validation of Model Transformations - First Experiences Using a White Box Approach. In *MoDELS Workshops*, volume 4364 of *LNCS*, pages 193–204. Springer, 2006.
6. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), version 1.1, www.omg.org/spec/QVT. 2011.
7. E. Planas, J. Cabot, and C. Gómez. Verifying Action Semantics Specifications in UML Behavioral Models. In *CAiSE*, volume 5565 of *LNCS*, pages 125–140. Springer, 2009.
8. E. Planas, J. Cabot, C. Gómez, E. Guerra, and J. de Lara. Lightweight Executability Analysis of Graph Transformation Rules. In *VL/HCC*, pages 127–130, 2010.