Translating ATL Model Transformations to Algebraic Graph Transformations

Elie Richa, Etienne Borde, Laurent Pautet

Translating ATL Model Transformations to Algebraic Graph Transformations

Elie Richa[1,2], Etienne Borde[1], and Laurent Pautet[1]

[1] Institut Telecom; TELECOM ParisTech; LTCI - UMR 5141
46 Rue Barrault 75013 Paris, France
firstname.lastname@telecom-paristech.fr
[2] AdaCore, 46 Rue d'Amsterdam 75009 Paris, France
lastname@adacore.com

Abstract. Analyzing and reasoning on model transformations has become very relevant for various applications such as ensuring the correctness of transformations. ATL is a model transformation language with rich semantics and a focus on usability, making its analysis not straightforward. Conversely, Algebraic Graph Transformation (AGT) is an approach with strong theoretical foundations allowing for formal analyses that would be valuable in the context of ATL. In this paper we propose a translation of ATL to the AGT framework in the objective of bringing theoretical analyses of AGT to ATL transformations. We validate our proposal by translating a set of feature-rich ATL transformations to the Henshin AGT framework. We execute the ATL and AGT versions on the same set of models and verify that the result is the same.

Keywords: ATL, Henshin, algebraic graph transformation, OCL, nested graph conditions, analysis of model transformations

1 Introduction

Model transformations play a central role in Model Driven Engineering (MDE) processes. They formalize and automate design decisions (*e.g.* optimisations), implementation strategies (*e.g.* code generation) or translations/synchronization between different model representations. Analyzing model transformations and reasoning about them has therefore become increasingly interesting for various concerns such as demonstrating the correctness of transformations via testing or static formal analysis. Many transformation approaches have been proposed with varying languages and semantics targeting different concerns.

ATL [11] is a widely used model transformation language, both in academia and in the industry. It features a hybrid rule-based language with a rich execution semantics allowing for a mostly declarative and user-friendly specification. *Algebraic Graph Transformation* (AGT) [8] is a formal framework that provides mathematical definitions to express graph manipulation. Its strong theoretical foundations allow for powerful analyses such as state space reachability analysis

and formal proof of termination, confluence and correctness. Given the graph-like structure of models in the sense of MDE, the theoretical results of AGT are increasingly being used to reason on model transformations.

Various analyses have already been proposed for ATL without relying on AGT. This includes test generation [9] and verification of correctness properties [6,16] through translations of ATL to other analyzable specifications. However we are interested in an analysis that is not possible with existing formalisations of ATL: the construction of *Weakest Precondition* (WP) [10]. This analysis operates on *constraints* and transforms a *postcondition* into an equivalent *precondition* of a transformation. It is defined in AGT and used in several scenarios such as synthesizing transformation preconditions that ensure the preservation of validity constraints [7], and formally proving the correctness of transformations [13]. Moreover in a previous publication [15], we have proposed a new use of this analysis to support the testing of model transformation chains. In that context we use WP construction as a way to propagate unit test requirements of intermediate steps of a chain into equivalent integration test requirements over the input of the chain which are easier to satisfy and maintain. We believe that WP-based analyses would be valuable for ATL transformations (and chains) and therefore propose to make them possible via a translation to AGT.

In this paper we propose a translation of ATL transformations to equivalent AGT analysable transformations and provide an implementation in our tool *ATLAnalyser*[3]. The first challenge in this work is handling ATL's *default* and *non-default resolve mechanisms* which do not have an equivalent in the AGT semantics. The second challenge is the translation of OCL constraints and queries of ATL rules into application conditions in the form of *Nested Graph Conditions* (NGC) in AGT. While translations of OCL to NGC have been proposed in the literature [3,4], they do not support ordered collections which we found to be an important limitation for ATL transformations. Our work extends the existing translations with support for ordered sets. Finally, we validate our proposal by considering a set of representative ATL transformations taken from the ATL Zoo [1] and other sources. We translate each transformation to the Henshin AGT framework [2] and verify that the execution of both the ATL and AGT versions over the same set of input models gives the same results.

The remainder of the paper is organised as follows. We start by recalling the semantics of ATL and AGT in Sec. 2. Then we present in Sec. 3 the main contribution of this paper: the translation of ATL to AGT. Section 4 reports on the experimental validation and the limitations of our proposal. Related work is discussed in Sec. 5 before concluding with future work in Sec. 6.

2 Semantics of ATL and AGT

2.1 ATL and OCL

ATL [11] is a model-to-model transformation language combining declarative and imperative approaches in a hybrid semantics. ATL transformations are primarily

[3] *ATLAnalyser*, `https://github.com/elericha/atlanalyser`

out-place, *i.e.* they produce an output model different from the input model (though both may be in the same language), and a so-called *refining mode* allows for *in-place* model refinement transformations. In the scope of this paper, we focus only on the declarative features of ATL in the standard *out-place* mode.

A transformation consists of a set of declarative *matched rules*, each specifying a *source pattern* (the **from** section) and a *target pattern* (the **to** section). The source pattern is a set of objects of the input metamodel and an optional OCL [12] constraint acting as a *guard*. The target pattern is a set of objects of the output metamodel and a set of *bindings* that assign values to the attributes and references of the output objects. For example in Fig. 1, **R1** has one source pattern element **s** and two target pattern elements: **t1** with 3 bindings and **t2** with 1 binding.

```
1  rule R1 {
2  from s  : IN!A
3     (s.refB->exists(b | b.name = 'Hodor'))
4  to t1 : OUT!D
5       (name <- s.name + '1',
6        refD <- t2,
7        refE <- s.refB),
8     t2 : OUT!D
9       (name <- s.name + '2') }
10 rule R2 {
11 from s : IN!B
12 to t : OUT!E
13      (refD <- thisModule.resolveTemp(s.refA, 't2') ) }
```

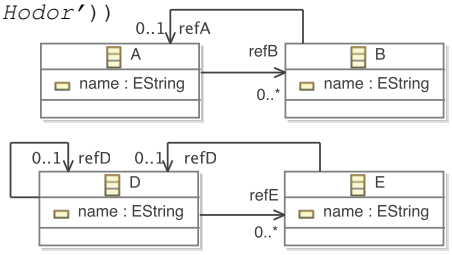


Fig. 1. Example of ATL rules

An ATL transformation is executed in two phases. First, the matching phase searches in the input model for objects matching the source patterns of rules (*i.e.* satisfying their filtering guards). For each match of a rule's source pattern, the objects specified in the target pattern are instantiated. Second, the target elements' initialization phase executes the bindings for each triggered rule.

A *binding* defines a *target property* which is an attribute or a reference on the left side of the **<-** symbol, and an OCL query on the right side of the symbol. A binding maps a scalar value to a target attribute (line 5), target objects (instantiated by the same rule) to a target reference (line 6), or source objects to a target reference (line 7). In the latter case, a *resolve* operation is automatically performed to find the rule that matched the source objects, and the *first* output pattern object created by that rule is used for the assignment to the target reference. This is referred to as the *default resolve mechanism*. For example in Fig. 1, the binding at line 7 resolves the objects in **s.refB** into the output objects of type **E** created by **R2**, and assigns them to **t1.refE**.

Another *non-default resolve mechanism* allows resolving a (set of) source object(s) to an arbitrary target pattern object instead of the first one as in the

default mechanism. It is invoked via the following ATL standard operations:
`thisModule.resolveTemp(obj, tgtPatternName)`
`thisModule.resolveTemp(Sequence{obj1, ...}, tgtPatternName)`
The former is used to resolve with rules having one source pattern element while the latter is used to resolve with rules having multiple source pattern elements. For example, the execution of the binding on line 13 in rule **`R2`** will retrieve the target object **`t2`** (instead of **`t1`** as with the default resolve) that was created by **`R1`** when it matched **`s.refA`**.

2.2 AGT and Nested Graph Conditions

Algebraic Graph Transformation (AGT) [8] is a formal framework that provides mathematical definitions to model graph transformations. We will be using the *Henshin* [2] graph transformation framework which applies the theoretical semantics to standard EMF models in the Eclipse platform. The details of the formal foundations of Henshin can be found in [5] and are only briefly recalled here. A graph transformation is composed of two main elements: a set of *transformation rules*, and a *high-level program* defining the sequencing of rules.

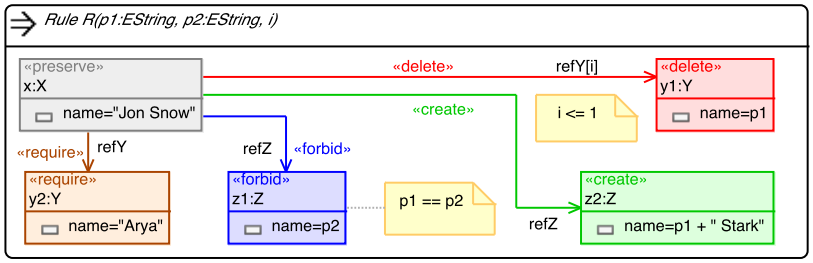


Fig. 2. Henshin graphical representation of an AGT rule

An AGT rule consists of a Left-Hand Side (LHS) graph and a Right-Hand Side (RHS) graph both depicted on the same diagram as in Fig. 2. LHS elements are annotated with *«preserve»* or *«delete»* while RHS elements are annotated with *«preserve»* or *«create»*. Roughly, a rule is executed by finding a match of LHS in the transformed graph, deleting the elements of $LHS - RHS$ (*«delete»*), and creating the elements of $RHS - LHS$ (*«create»*). Elements of $LHS \cap RHS$ are preserved (*«preserve»*). A rule transforms elements matched by the LHS into the RHS, therefore an AGT is an *in-place* rewriting of the input model. For example, rule R in Fig. 2 matches nodes x of type X and $y1$ of type Y and edge $refY$ in the transformed graph, deletes the node matched by $y1$ and the edge matched by $refY$, and creates node $z2$ of type Z and the edge $refZ$.

Matches of a rule may be restricted with additional constraints by assigning *attribute values* to nodes. For example the rule in Fig. 2 can only match an object x when $x.name =$ "Jon Snow". Moreover, attribute values may be stored in *rule parameters* such as in $y1.name = p1$ where the *name* attribute of the object matched by $y1$ is stored in the rule parameter $p1$. Finally, a rule may

assign new values to attributes such as in $z2$ where $z2.name$ is initialized to $p1$ concatenated to the string *" Stark"*.

In Henshin, edges typed by a *multi-valued ordered reference* (*i.e.* with upper bound higher than 1) can be labeled with an *index*. This feature will play an important role in the handling of the ATL resolve mechanisms and the support of ordered sets in Sec. 3. A literal integer index such as $ref[2]$ represents a matching constraint: only the object at index 2 may be matched by the rule. A rule parameter index such as $ref[i]$ allows to read an object's index in the ordered reference and store it in the parameter. For example in Fig. 2, $refY[i]$ indicates that the index of $y1$ is stored in i. Edge indexes are zero-based.

An AGT rule can have an *application condition* (AC) which constrains its possible matches. An AC is a *Nested Graph Condition* (NGC) over the LHS. Formally, a NGC over a graph P is of the form *true* or $\exists(a \,|\, \gamma, c)$ where $a : P \hookrightarrow C$ is an injective morphism, γ is a boolean expression over rule parameters and c is a NGC over C. A match $p : P \hookrightarrow G$ of P in a graph G satisfies an AC $\exists(a \,|\, \gamma, c)$ if there exists a match $q : C \hookrightarrow G$ of C in G such that $p = q \circ a$ and γ evaluates to *true* under the parameter assignment defined by p and q satisfies c. Boolean formulas can be constructed such as the negation $\neg c$, the conjunction $\bigwedge_i c_i$ and the disjunction $\bigvee_i c_i$ of NGCs c_i over P. We use short notations $\forall(a, c)$ and $c_1 \implies c_2$ for $\neg\exists(a, \neg c)$ and $\neg c_1 \vee c_2$ respectively. For example the AC in Fig. 3 defined for rule R requires the existence of a node $y2$ whose *name* attribute is *"Arya"* and forbids the existence of a node $z1$ with the same *name* as $y1$. The boolean expression $i <= 1$ constrains the rule to match only for the first two objects in the ordered reference $x.refY$. Note that P is omitted from the notation when it can be inferred from the context, and so are γ and c when they are *true*. The AC is graphically represented in Fig. 2 using the annotations *«require»* and *«forbid»*, however this is only possible for one level of nesting in the AC. For complete NGCs the full notation of Fig. 3 is necessary. In Sec. 3 we will translate OCL guards and bindings into suitable ACs of AGT rules.

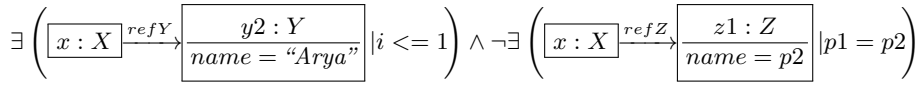


Fig. 3. Example of a Nested Graph Condition

Finally we define a so-called *high-level program* which specifies in which order AGT rules are applied. A program can be (1) elementary, consisting of a rule r, (2) the sequencing of two programs P and Q denoted by $(P; Q)$, or (3) the iteration of a program P as long as possible, denoted by $P\downarrow$, which is equivalent to a sequencing $(P; (P; (P \cdots)$ until the program P can no longer be applied.

3 Translating ATL to AGT

Having presented the semantics of ATL and AGT, we now tackle the main contribution of this paper: the translation of ATL transformations to AGT transformations. Section 3.1 focuses on the first challenge, the emulation of the ATL

resolve mechanisms in AGT, and Section 3.2 addresses the second challenge, the translation of OCL constraints and queries embedded in ATL transformation with support for ordered sets. To avoid confusion between ATL and AGT transformation rules, we will denote them respectively by $rule_{ATL}$ and $rule_{AGT}$.

3.1 Translating the ATL Resolve Mechanism

Given the *out-place* nature of the ATL transformations we consider and *in-place* nature of AGT we propose to model the ATL transformation in AGT as a refinement of the input model which only adds the elements of the output model without modifying the input elements.

Challenges. A first challenge is dealing with the ATL resolve mechanisms. In AGT no such mechanisms exist, and any objects that a rule_{AGT} needs to use must already exist in the transformed graph and must be matched by the rule_{AGT}'s LHS. If a rule_{AGT} $R1$ needs to use an object created by rule_{AGT} $R2$, then $R2$ must be executed before $R1$. This becomes a problem if $R1$ and $R2$ mutually require objects created by each other which is a perfectly valid scenario in ATL that cannot be solved with simple rule_{AGT} sequencing. Moreover, the non-default resolve mechanism requires to relate output objects to output pattern identifiers so that we can retrieve the object corresponding to a specific output pattern identifier given as argument to the **`resolveTemp`** operation.

General Solution. We propose to construct the AGT transformation similarly to the ATL execution semantics, as two sequential phases: an *instantiation* phase followed by a *resolving* phase. Moreover, we introduce *trace nodes* that maintain the relationship between input and output elements. The first phase applies a sequence of *instantiation* $rules_{AGT}$ that create output objects without initializing their attributes and references, and relate them to input objects through *trace nodes*. Each rule_{ATL}, *e.g.* **`R1`** from Fig. 1, yields one instantiation rule_{AGT} $R1_{Inst}$ that matches the same objects as **`R1`**. $R1_{Inst}$ is iterated as long as possible so that all matches in the input model are processed. The order of application of instantiation $\text{rules}_{\text{AGT}}$ is irrelevant as they do not interfere with each other since objects are allowed to match for only one rule_{ATL}, as per the ATL semantics.

The second phase of the transformation applies a set of *resolving* $rules_{AGT}$ which initialize references and attributes of output objects. Each binding in a rule_{ATL} is translated to one or more resolving $\text{rules}_{\text{AGT}}$ as will be discussed shortly. For example, **`R1`** yields 4 resolving $\text{rules}_{\text{AGT}}$ $R1_{Res}^{t1,name}$, $R1_{Res}^{t1,refD}$, $R1_{Res}^{t1,refE}$ and $R1_{Res}^{t2,name}$. Resolving $\text{rules}_{\text{AGT}}$ navigate the input model and rely on the trace nodes created in the instantiation phase to perform the resolving and retrieve the corresponding output objects if needed. Like instantiation $\text{rules}_{\text{AGT}}$, resolving $\text{rules}_{\text{AGT}}$ are also iterated as long as possible so that bindings are applied to all output objects. The resulting AGT transformation is the following:

$$R1_{Inst}\!\downarrow;\ R2_{Inst}\!\downarrow;\ R1_{Res}^{t1,name}\!\downarrow;\ R1_{Res}^{t1,refD}\!\downarrow;\ R1_{Res}^{t1,refE}\!\downarrow;\ R1_{Res}^{t2,name}\!\downarrow;\ R2_{Res}^{t,refD}\!\downarrow$$

This scheme addresses the highlighted concerns regarding the resolve mechanism. Separating the creation of output objects from their use allows resolving $\text{rules}_{\text{AGT}}$ to use any output object even in the case of mutual resolve dependencies. Moreover, the trace nodes maintain the information required to perform the resolving as explained next.

Trace Nodes. The *trace nodes* we introduce are typed by a set of metaclasses produced by our translation. We assume that both the input and output metamodels define a root abstract metaclass from which all other metaclasses inherit directly or transitively[4] and refer to them respectively as *RootIn* and *RootOut*. The trace metaclasses are produced as follows. First, an *abstract* metaclass *Trace* is defined with a *from* reference to *RootIn* and a *to* reference to *RootOut* (Fig. 4.a). For each rule_{ATL}, *e.g.* **R1**, a so-called *typed trace* metaclass named *R1_Trace* inheriting the abstract *Trace* metaclass is created. For each input and output pattern element of the rule_{ATL}, a reference with the same name is created from the typed trace to the type of the pattern element. For **R1** this yields references *s*, *t1* and *t2* in Fig. 4.a.

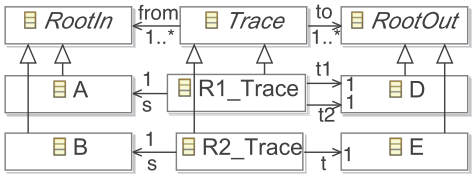


Fig. 4.a. Trace metamodel

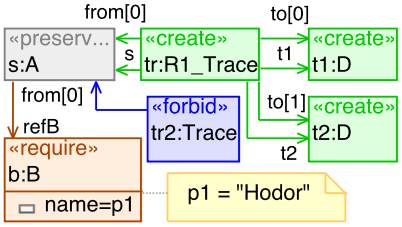
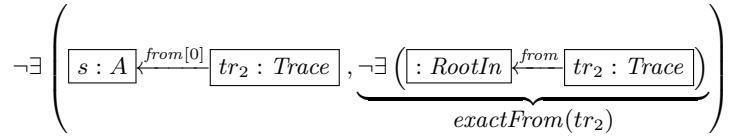


Fig. 4.b. Instantiation rule_{AGT} $R1_{Inst}$

Instantiation $\text{Rules}_{\text{AGT}}$. Each rule_{ATL}, **R1** for example, yields one instantiation rule_{AGT}, $R1_{Inst}$, which matches the same objects as **R1** and creates the output objects as well as a typed trace node. As can be seen in Fig. 4.b, the instantiation rule_{AGT} is constructed by creating a *«preserve»* node for each input pattern element (node $s : A$). Then the OCL rule_{ATL} guard is translated to an AC as per Sec. 3.2. This yields the *«require»* navigation to node $b : B$ with $name = p1$ and $p1 =$ "*Hodor*". Then, a *«create»* node is created for each output pattern element (nodes $t1 : D$, $t2 : D$) as well as a typed trace node ($tr : R1\_Trace$). The trace node is connected to input nodes with generic *from* references and typed references (s) and to output node with generic *to* references and typed references (*t1* and *t2*). The order of input and output pattern elements is preserved in *from* and *to* references by indexing the created edges accordingly ($from[0]$, $to[0]$ and $to[1]$). This will allow resolve $\text{rules}_{\text{AGT}}$ to retrieve the first output object ($to[0]$) for the default resolve mechanism or any arbitrary output object (*t1* or *t2*) for the non-default resolve mechanism. Finally, since a rule_{ATL} only applies once per match, we add a negative AC preventing the application of the rule_{AGT} if *another* trace node tr_2 with the exact same *from* elements already exists. That AC is as follows:

[4] if it is not the case, such a root abstract metaclass can be added automatically



The NGC $exactFrom(tr_2)$ (not visible on Fig. 4.b) is needed to express the fact that the object s is allowed to participate in another rule_{ATL} if there are other objects in the source pattern (*i.e.* the set of *from* elements is not exactly the same). *exactFrom* is reused for resolving $\text{rules}_{\text{AGT}}$ in the following sections.

Resolving $\text{Rules}_{\text{AGT}}$ with Default Resolving. Each binding in a rule_{ATL} is translated to one resolving rule_{AGT}. Let us first consider the case of bindings with default resolving or no resolving at all. Each such binding results in one rule_{AGT} that matches the same elements as the OCL query in the binding, performs the default resolving if needed, and initializes the target attribute/reference of the binding. Let us consider a binding of the following general shape:

tgtObj **:** *tgtType* **(** *tgtProp* **<-** *oclQuery* **)**

The supported subset of OCL in *oclQuery* will be defined in Sec. 3.2, however the general translation remains the same. Such a binding is translated to a resolving rule_{AGT} $R_{Res}^{tgtObj,tgtProp}$ according to the algorithm presented in Table 1. The translation depends on the type of the target property *tgtProp* hence the tabular presentation. Note that multi-valued target attributes are not supported at the current stage.

Figure 5 shows the steps of the translation of binding **`t1:D (refE <- s.refB)`** in **`R1`** (Fig. 1) to rule_{AGT} $R1_{Res}^{t1,refE}$. Note that *to*[0] in *Step 3* allows to retrieve the first target pattern element as per the default resolve semantics. Moreover, for multivalued target references such as **`t1.refE`**, the translation is a sort of a *flattening* whereby the result elements of the OCL query **`s.refB`** are not handled all at once but one by one. Each application of $R1_{Res}^{t1,refE}$ matches one element in **`s.refB`** and appends the corresponding output object to the target reference **`t1.refE`**. However, since there are no guarantees in AGT on the order in which elements are matched, $R1_{Res}^{t1,refE}$ as presented in Fig. 5 is only correct if **`refB`** is a non-ordered reference. This will be detailed and addressed in Sec. 3.2.

Resolving $\text{Rules}_{\text{AGT}}$ with Non-Default Resolving. Let us now consider bindings involving non-default resolving which have the following shape:

tgtObj **:** *tgtType* **(** *tgtRef* **<-**
`thisModule.resolveTemp(Sequence{`*navExp*$_1$**`, ... ,`** *navExp*$_N$**`},`** *tgtPat*`))`

The construction of the resolving rule_{AGT} $R_{Res}^{tgtObj,tgtRef}$ operates in the same steps as Table 1 except for steps 2 and 3 which are presented in Table 2. The case where the first parameter of **`resolveTemp`** is an object is treated in the same way as a **`Sequence`** containing only that object.

In this section we have presented the general ATL to AGT translation scheme focusing on the emulation of the resolve mechanisms by introducing trace nodes. The next section will focus on the translation of OCL guards and queries.

Table 1. Translation of an ATL binding with default resolving

	Binding *tgtObj* **:** *tgtType* **(** *tgtProp* **<-** *oclQuery* **)**		
	Single-valued Attribute $tgtAtt \equiv tgtProp$	**Single-valued Reference** $tgtRef \equiv tgtProp$	**Multi-valued Reference** $tgtRef \equiv tgtProp$
Step 1	Initialize LHS with $\boxed{:\text{<ruleName>}\_\,Trace} \xrightarrow{tgtObj} \boxed{tgtObj : tgtType}$		
Step 2	Translate *oclQuery* as per Sec. 3.2. This will complement the LHS with the required navigations and ACs and return a result.		
	Result is an expression *expr* over rule_{AGT} parameters	Result is a node $\boxed{qNode}$ representing the query result	Result is a node $\boxed{qNode}$ representing one element of the result set
Step 3	Not Applicable. Step 3 is specific to reference target properties.	If the node is a source model element, perform a *default resolve* by matching a trace node with the exact *from* object using the following in the LHS: $\boxed{qNode} \xleftarrow{from[0]} \boxed{tr : Trace} \xrightarrow{to[0]} \boxed{rNode : type(tgtRef)}$ and the AC $exactFrom(tr)$	
		If not, let $\boxed{rNode} \equiv \boxed{qNode}$	
Step 4	Create the following attribute in the RHS $\boxed{\frac{tgtObj}{tgtAtt = expr}}$	Create $\boxed{tgtObj} \xrightarrow{tgtRef} \boxed{rNode}$ in the RHS	
Step 5	Add a negative AC to force the application of the rule *once* per match		
	$\neg\exists \left(\begin{array}{c} \boxed{\frac{tgtObj}{tgtAtt = p1}} \mid \\ p1 = expr \end{array} \right)$	$\neg\exists \left(\boxed{tgtObj} \xrightarrow{tgtRef} \boxed{rNode} \right)$	

3.2 Translating OCL Guards and Binding Expressions

As explained previously, rule_{ATL} guards and binding expressions are translated to ACs of respectively *instantiation* and *resolving* $\text{rules}_{\text{AGT}}$. Despite the considerable difference between NGC and OCL, NGC has been shown to be expressively equivalent to first order logic [13] which is the core of OCL. Translations of subsets of OCL to NGC have been proposed in [3] with a highly theoretical approach and in [4] with a wider supported OCL subset and an experimental approach. We have taken inspiration from both works and have found that none of them supports ordered sets, leading us to tackle this problem in particular. In

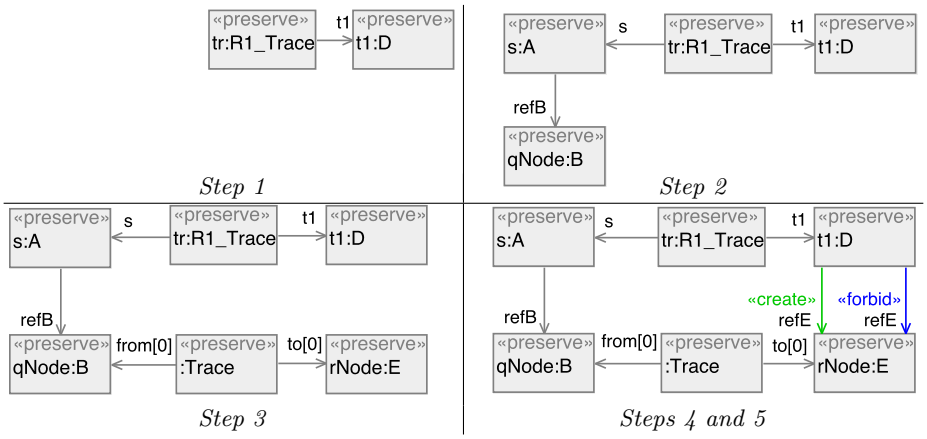


Fig. 5. Construction of resolving rule_{AGT} $R1^{t1,refE}_{Res}$

the following we will only recall the general principles of the existing translations and detail the problem at hand and our proposal.

The main idea is to translate OCL object queries into graphs that match the objects in the query's result set, and OCL constraints into NGCs that are satisfied under the same conditions. For example, the navigation of a reference **`s.refB`** is translated by creating the graph $\boxed{s} \xrightarrow{refB} \boxed{r}$ where r represents one object in the result set of the query and is returned as a result of the translation (see Step 2 in Table 1). As for the navigation of an object attribute such as in **`s.name + '1'`**, it is translated by creating a rule parameter p and assigning the attribute value to it "$name = p$" in the node s, and returning the expression $p+$"1" as a result of the translation (Step 2 in Table 1). The supported subset of OCL is similar to the one in [4] which is limited to basic navigation, first order logic constructs and **`Set`** as the only collection type with basic set operations such as **`select()`**, **`collect()`**, **`union()`**. We extend this support to **`OrderedSet`** with support for indexing **`at(i)`** and the preservation of order in output collections.

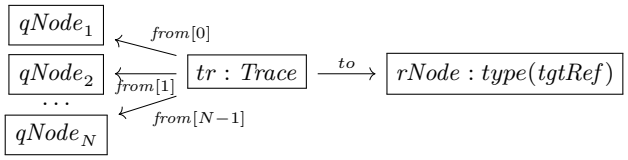
Challenge 1. A first challenge is the handling of bindings that aggregate results of several queries. This is the case of the following binding shapes where in (1) resolved objects in *tgtRef* should be in the same order as the source objects in the **`OrderedSet`**, and in (2) $oclQuery_1$ should be resolved before $oclQuery_2$.

$$tgtRef \texttt{<- OrderedSet\{} oclQuery_1\texttt{,}\ oclQuery_2 \ldots oclQuery_N \texttt{\}} \quad (1)$$

$$tgtRef \texttt{<-}\ oclQuery_1\texttt{->union(}oclQuery_2\texttt{)} \quad (2)$$

Solution 1. To preserve the ordering of elements, we propose to translate such bindings as separate successive bindings: *tgtRef* **`<-`** $oclQuery_1$, *tgtRef* **`<-`** $oclQuery_2$, ... Each such binding results in a separate resolving rule_{AGT} and the $\text{rules}_{\text{AGT}}$ are sequenced in the same order as the queries in the original binding.

Table 2. Translation of an ATL binding with non-default resolving

	resolveTemp(Sequence{$navExp_1$**, ..., **$navExp_N$**}, **$tgtPat$**)**
Step 2	Translate each $navExp_i$ as per Sec. 3.2. This will complement the LHS with the required navigations and ACs and return as a result a set of nodes $qNode_i$ representing the navigated objects
Step 3.a	Perform a *non-default resolve* by matching a trace node with the exact *from* tuple. Differently than for the default resolve, *to* is not indexed.  and add the AC $exactFrom(tr)$
Step 3.b	Compute *CRules* as the set of all candidate $\text{rules}_{\text{ATL}}$ that have N source pattern elements and *tgtPat* as one of their target pattern elements.
Step 3.c	Add to the rule_{AGT}'s AC the following disjunction: $\bigvee_{cRule \in CRules} \exists \left(tr : <cRule>\_Trace \xrightarrow{tgtPat} rNode \right)$

Consequently objects are appended to *tgtRef* in the right order at run-time. Therefore (1) is translated to N sequential resolving $\text{rules}_{\text{AGT}}$ and (2) is translated to 2 sequential resolving $\text{rules}_{\text{AGT}}$.

Challenge 2. The second challenge is the navigation of ordered multi-valued references. Let us illustrate this problem with the following binding from **R1** in Fig. 1: **t1 : OUT!D (refE <- s.refB)**
refB is a multivalued reference (*i.e.* upper bound larger than 1). We have previously shown the translation of this binding in Fig. 5 under the assumption that **refB** is a *non-ordered* reference. The navigation **s.refB** is *flattened*, meaning that the elements of the collection are not handled all at once, but rather one by one thanks to the iteration of $R1^{t1,refE}_{Res}$. According to AGT graph matching, objects in **s.refB** may be matched in any order. Therefore objects in **t1.refE** may end up in a different order than their counterparts in **s.refB** which is a problem if **refB** and **refE** are ordered. That constitutes a divergence from the ATL semantics which honours the order of objects in collections. Therefore we need a way to force the matching of objects in **s.refB** in an orderly fashion.

Solution 2. We propose to complement the regular translation of navigation expressions [3,4] with an additional NGC forcing objects to be matched in the correct order. Intuitively, this NGC should express the fact that an object in **s.refB** should be matched only if all preceding objects in **s.refB** have already been handled by the resolving rule_{AGT}. This corresponds to the following NGC:

$$orderingAC = \exists \Big(\boxed{s:A} \xrightarrow{refB[i]} \boxed{qNode:B}, \forall \Big(\boxed{s:A} \xrightarrow{refB[j]} \boxed{qNode_1:B} \mid j < i,\ wasResolved_{R1}^{t1,refE}(qNode_1) \Big) \Big)$$

Where:

- i : index of the object *qNode* currently being handled.
- j : index of the object $qNode_1$ which iterates over objects preceding *qNode*.
- $wasResolved_{R1}^{t1,refE}(n)$: A NGC which evaluates to *true* if node n has already been handled by the resolving rule_{AGT}.

Now we need to define $wasResolved_{R1}^{t1,refE}(n)$. We can determine that a node n has been already handled by checking if the node to which it resolves exists in the target reference *t1.refE*. Therefore the following definition is suitable: $wasResolved_{R1}^{t1,refE}(n) =$

$$\exists \Big(\boxed{n} \xleftarrow{from[0]} \boxed{tr:Trace} \xrightarrow{to[0]} \boxed{:E} \xleftarrow{refE} \boxed{t1:D},\ exactFrom(tr) \Big)$$

With the above definitions, adding *orderingAC* as an application condition of $R1_{Res}^{t1,refE}$ ensures that objects in **`s.refB`** are processed in the correct order, thus honoring the ATL semantics. Let us now generalize this reasoning to the case where the navigation is filtered with a **`select`** operation:

`t1 : OUT!D ( refE <- s.refB->select(e |` *body*(**`e`**) **`)`**

Now an object in **`s.refB`** should be matched only if it satisfies the **`select`** condition, and if all preceding objects in **`s.refB`** which also satisfy the **`select`** condition have been handled by the resolving rule_{AGT}. Therefore the AC that would ensure the orderly processing of objects is the following:

$$orderingAC = \exists \Big(\boxed{s:A} \xrightarrow{refB[i]} \boxed{qNode:B},\ tr_{body}(qNode) \bigwedge \forall \Big(\boxed{s:A} \xrightarrow{refB[j]} \boxed{qNode_1:B} \mid j < i,\ tr_{body}(qNode_1) \implies wasResolved_{R1}^{t1,refE}(qNode_1) \Big) \Big)$$

Where $tr_{body}(n)$ is the NGC resulting from the translation of the OCL constraint *body*(**`e`**), applied to a node n. The generalization can be extended to all supported OCL expressions but will not be detailed here for lack of space.

4 Experiments and Validation

4.1 Validation Protocol

We have used the *Henshin* Eclipse framework as the target of the translation as it is well integrated with EMF and allows the execution of AGT transformations on standard EMF models. The ATL to AGT translation is implemented in our Java-based tool *ATLAnalyser* and validated by considering a set of ATL transformations from the ATL Zoo [1] and from other sources. Each transformation

is translated to AGT using our implementation and the resulting AGT transformation is validated manually by review. Then both the ATL and AGT versions are executed over a set of input models and in each case the output models of the two versions are checked to be identical using EMFCompare. Except for the manual review, this experimental validation protocol is fully automated (using JUnit) which allows to easily expand our test base with new transformations and models, and monitor the non-regression of existing tests as the prototype evolves. We have also identified the ATL features that each transformation contains to make sure we exercise all aspects of the translation.

Our prototype was successfully validated with the transformations listed in Table 3. *Simulink CodeGen* is a simplified version of an industrial Simulink to C code generator[5]. Note that in *Families2Persons*, the high number of resolving rules (relative to only 2 bindings) is due to the translation scheme of nested **`if-then-else`** binding queries which has not been developed in this paper.

Table 3. List of test transformations and tested features

	Families2-Persons [1]	**Class2-Relational** [1]	**ER2REL** [6]	**Simulink CodeGen**[4]
Metrics				
ATL rules	2	6	6	6
ATL bindings	2	22	13	30
Instantiation rules	2	6	6	6
Resolving rules	8	23	15	32
ATL Features				
Default Resolve	X	X	X	X
`resolveTemp`				X
Helpers	X	X		X
Attribute binding	X	X	X	X
Reference binding		X	X	X
`OrderedSet{}`		X		X
`union()`		X	X	X
`select()`		X		X
`collect()`, **`at()`**				X

4.2 Limitations and Threats to Validity

A first limitation of our proposal is the lack of formal evidence of its validity. Though part of our translation is based on the one in [3] which is formally proven to be correct, our OCL subset is significantly wider preventing any direct claim of correctness of the complete translation. Second, while the addressed scope was found sufficient to translate a wide range of ATL transformations, features like non-unique collections (**`Bag`**, **`Sequence`**), collections of collections, and special values (**`null`**, **`invalid`**) are not supported because they cannot be represented in the AGT framework used in this paper. Finally, with the validation scheme

[5] Project P, `http://www.open-do.org/projects/p`

presented in Sec. 4.1, we are faced with the challenge of any test-based validation which is the coverage and relevance of the test transformations and test models. We have tried to address this issue by identifying the ATL features used by each transformation to make sure that all aspects of the translation are tested (see Table 3). However we acknowledge that our tool is essentially a language compiler, and the verification of such tools is known to be a difficult problem.

5 Related Work

Though translations of OCL to NGC have been conducted [3,4], no previous work has proposed a translation of ATL to AGT to the best of our knowledge. In [14] the authors propose to translate model transformations from the Epsilon language family (arguably similar to ATL and OCL) to AGT to show through formal proof that a given pair of unidirectional transformations forms a bidirectional transformation. However this work is still at an early stage and an automatic translation is not yet proposed.

In the broader context of the analysis of model transformations several works have translated ATL to other formalisms. ATL transformations are translated in [6] to a transformation model with suitable constraints expressing the ATL semantics and in [16] to a Maude specification with a rewriting logic arguably similar to our graph rewriting transformation. The analyses made possible by these and other formalisations include Hoare-style correctness analyses, *i.e.* verifying that an ATL model transformation ensures a postcondition under the assumption of a precondition [6], and reachability analysis [16] to find errors in the ATL transformation. Despite these existing results, we have targeted AGT in our work to benefit from the construction of *weakest precondition* (WP) in AGT [10] which is the translation of a postcondition NGC on the output of a transformation into an equivalent precondition NGC on its input. This analysis which is not possible in the existing formalisations of ATL is used for the synthesis of validity-preserving preconditions [7] and for the formal proof of Hoare-style correctness [13]. Applying it to ATL using our translation is one of our main future prospects in a novel approach for testing model transformation chains [15].

6 Conclusion

This paper has presented a translation of ATL transformations to the formal framework of Algebraic Graph Transformation (AGT). The main challenges of this work were the translation of the ATL resolve mechanisms which do not have a direct equivalent in AGT, and the translation of OCL guards and queries to suitable Nested Graph Conditions (NGC). In the latter translation, we have complemented existing OCL to NGC translations with support for ordered sets, allowing to faithfully translate a wider range of ATL transformations. We have implemented our translation targeting the Henshin AGT framework and have validated it by translating a set of representative ATL transformations from various sources, and comparing the execution of both ATL and AGT versions.

In future work, we plan to extend the translation to support more ATL and OCL features such as arbitrary sorting with **`sortedBy`** as well as multi-valued attributes. A more challenging task will be to support imperative features of ATL such as lazy rules and **`do`** blocks. As a first intuition we believe this would require enriching trace nodes with more information and using more imperative features of AGT. Finally, we plan to use the proposed translation to apply AGT formal analyses to ATL transformation, starting with the construction of weakest preconditions as a way to generate tests for ATL transformation chains [15].

References

1. ATL Transformation Zoo. `http://www.eclipse.org/atl/atlTransformations/`.
2. The Henshin project. `http://www.eclipse.org/henshin`.
3. T. Arendt, A. Habel, H. Radke, and G. Taentzer. From core OCL invariants to nested graph constraints. In *Graph Transformation*, LNCS 8571, pages 97–112. Springer, 2014.
4. G. Bergmann. Translating OCL to graph patterns. In *Model-Driven Engineering Languages and Systems*, LNCS 8767, pages 670–686. Springer, 2014.
5. E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.
6. F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *Formal Methods and Software Engineering*, LNCS 7635, pages 198–213. Springer, 2012.
7. F. Deckwerth and G. Varró. Attribute handling for generating preconditions from graph constraints. In *Graph Transformation*, LNCS 8571, pages 81–96. Springer, 2014.
8. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer, 2006.
9. C. González and J. Cabot. ATLTest: A white-box test generation approach for ATL transformations. In *Model Driven Engineering Languages and Systems*, LNCS 7590, pages 449–464. Springer, 2012.
10. A. Habel, K.-H. Pennemann, and A. Rensink. Weakest preconditions for high-level programs. In *Graph Transformations*, LNCS 4178, pages 445–460. Springer, 2006.
11. F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, LNCS 3844, pages 128–138. Springer, 2006.
12. Object Management Group (OMG). Object Constraint Language (OCL) 2.4. `http://www.omg.org/spec/OCL/2.4`, 2012.
13. C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, University of York, 2013.
14. C. M. Poskitt, M. Dodds, R. F. Paige, and A. Rensink. Towards rigorously faking bidirectional model transformations. In *AMT 2014 Workshop Proceedings*, pages 70–75, 2014.
15. E. Richa, E. Borde, L. Pautet, M. Bordin, and J. F. Ruiz. Towards testing model transformation chains using precondition construction in algebraic graph transformation. In *AMT 2014 Workshop Proceedings*, pages 34–43, 2014.
16. J. Troya and A. Vallecillo. A rewriting logic semantics for ATL. *Journal of Object Technology*, 10(5):1–29, 2011.