Verification of ATL Transformations Using Transformation Models and Model Finders

Fabian Buettner, Marina Egea, Jordi Cabot, Martin Gogolla

Verification of ATL Transformations Using Transformation Models and Model Finders

Fabian Büttner[1⋆], Marina Egea[2⋆⋆], Jordi Cabot[1], Martin Gogolla[3]

[1] AtlanMod Research Group, INRIA / Ecole des Mines de Nantes
[2] Atos Research & Innovation Dept., Madrid
[3] Database Systems Group, University of Bremen

fabian.buettner@inria.fr, marina.egea@atosresearch.eu,
jordi.cabot@inria.fr, gogolla@tzi.de

Abstract. In model-driven engineering, models constitute pivotal elements of the software to be built. If models are specified well, transformations can be employed for different purposes, e.g., to produce final code. However, it is important that models produced by a transformation from valid input models are valid, too, where validity refers to the metamodel constraints, often written in OCL. Transformation models are a way to describe this Hoare-style notion of partial correctness of model transformations using only metamodels and constraints. In this paper, we provide an automatic translation of declarative, rule-based ATL transformations into such transformation models, providing an intuitive and versatile encoding of ATL into OCL that can be used for the analysis of various properties of transformations. We furthermore show how existing model verifiers (satisfiability checkers) for OCL-annotated metamodels can be applied for the verification of the translated ATL transformations, providing evidence for the effectiveness of our approach in practice.

Keywords: Model transformation, Verification, ATL, OCL

1 Introduction

In model-driven engineering (MDE), models constitute pivotal elements of the software to be built. Ideally, if these models are specified sufficiently well, model transformations can be employed for different purposes, e.g., they may be used to finally produce code. The increasingly popularity of MDE has led to a growing complexity in both models and transformations, and it is essential that transformations are correct if they are to play their key role. Otherwise, errors introduced by transformations will be propagated and may produce more errors in the subsequent MDE steps.

Our work focuses on checking partial correctness of declarative, rule-based transformations between constrained metamodels. More specifically, we consider the transformation language ATL [15] and metamodels in MOF [21] style (e.g., EMF [25],

⋆ This research was partially funded by the Nouvelles Eq́uipes program of the Pays de la Loire region (France)
⋆⋆ This research was partially funded by the EU project NESSoS (FP7 256890)

KM3 [16]) that employ OCL [20,27] constraints to precisely describe their domain. These ingredients are popular due to their sophisticated tool support (in particular on the Eclipse platform) and because OCL is employed in almost all OMG specifications. Model transformations can be considered as programs that operate on instances of metamodels. In this sense, we can also apply the classical notion of correctness to model transformations. In this paper, we are interested in a Hoare-style notion of partial correctness, i.e., in the correctness of a transformation with respect to the constraints of the involved metamodels. In other words, we are interested in whether the output model produced by an ATL transformation is valid for any valid input model.

In this paper we present a verification approach based on *transformation models*. Transformation models are a specific kind of what is commonly called a 'trace model'. Given an ATL transformation $T : \mathcal{M}_I \to \mathcal{M}_F$ from a source metamodel $\mathcal{M}_I$ to a target metamodel[4] $\mathcal{M}_F$, a transformation model $\mathcal{M}_T$ is a metamodel that includes $\mathcal{M}_I$ and $\mathcal{M}_F$, and additional structural modeling elements and constraints in order to capture the execution semantics of T. In our opinion, this approach brings advantage because it reduces the problem of verifying rule-based transformations between constrained metamodels to the problem of verifying constrained metamodels only. This way, in terms of automated verification, we can reuse existing implementations and work for model verification, benefiting from the results achieved by a broad community over a decade.

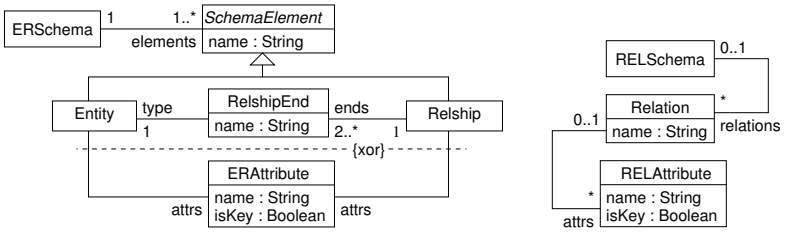
The transformation model methodology was first presented in [11] and [6]. We provided a first sketch of how to apply the methodology to ATL in [4]. In this paper, we now present a precise description of how to automatically generate transformation models from declarative ATL transformations. Furthermore, we show how existing model finders for OCL-annotated metamodels can be employed 'off-the-shelf' in practical verification. We employ a transformation ER-to-Relational (ER2REL) to illustrate our approach, as this example is well-known and conceptually 'dense' (it contains only few classes but comparatively many constraints). We show how the transformation model is derived using our algorithm and how it can be used to effectively verify the ATL transformation using UML2Alloy [1] and Alloy as a bounded model verification tool (with Alloy being based on SAT in turn). Notice, however, that the methodology is independent from a specific verification technique.

Organization. Sect. 2 describes the running example ER2REL. Sect. 3 shows how to derive transformation models for ATL. In Sect. 4 we present how UML2Alloy could be employed to validate ER2REL (based on the derived transformation model). Sect. 5 puts our contribution in the context of related work. We conclude in Sect. 6.

2 Running Example

We have chosen an ATL transformation (ER2REL) from a simple Entity-Relationship (ER) to a simple relational (REL) data model as a running example for our paper for

[4] For typographical reasons we use $\mathcal{M}_I$ ('initial') and $\mathcal{M}_F$ ('final') to denote the input and output.



(a) ER (b) REL

Fig. 1. ER and REL metamodels

```
context ERSchema inv ER_EN:              -- element names are unique in schema
  self.elements->forAll(e1,e2 |  e1.name=e2.name implies e1=e2)
context Entity inv ER_EAN:               -- attr names are unique in entity
  self.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2)
context Relship inv ER_RAN:              -- attr names are unique in relship
  self.attrs->forAll(a1,a2 | a1.name = a2.name implies a1=a2)
context Entity inv ER_EK:                -- entities have a key
  self.attrs->exists(a | a.isKey)
context Relship inv ER_RK:               -- relships do not have a key
  not attrs->exists(a1 | a1.isKey)
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
context RELSchema inv REL_RN:            -- relation names are unique in schema
  relations->forall(r1,r2|   r1.name=r2.name implies r1=r2)
context Relation inv REL_AN:             -- attribute names  unique in relation
  self.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2)
context Relation inv REL_K:              -- relations have a key
  self.attrs->exists(a | a.isKey)
context RELSchema inv REL_mult1:  self.relations->size() > 0   -- mult. 1..*
context Relation inv REL_mult2:   self.schema <> null          -- mult. 1..1
context Relation inv REL_mult3:   self.relations->size() > 0   -- mult. 1..*
context RELAttribute inv REL_mult4: self.relation <> null      -- mult. 1..1
```

Fig. 2. OCL constraints for ER and REL

two reasons. First, this domain is well-known (the results can be easily validated). Second, almost all elements are constrained by one or more invariants, including several universal quantifiers. This makes the verification of this transformation reasonably hard.

Fig. 1 depicts the ER and REL metamodels[5]. Fig. 2 shows the corresponding OCL constraints. The constraints are as expected: names must be unique within their respective contexts, entities and relation must have a key, relationships must not have a key. Notice that we encoded the multiplicity constraints for REL as explicit OCL constraints (REL_mult). We only left the unrestricted multiplicities of $0..1$ (for object-typed navigations) and $0..*$ (for collection-typed navigations) in the class diagram, because we want to verify the validity of ER2REL w.r.t. these multiplicities later.

The ATL transformation ER2REL is shown in Fig. 3 and contains six rules: The first rule S2S maps ER schemas to REL schemas, the second rule E2R maps each entity to a relation, and the third rule R2R maps each relationship to a relation. The remaining

[5] Notice that we simply refer to the elements as entities, relationships, and relations instead of entity types, relationship types, and relation types.

```
module ER2REL; create OUT : REL from IN : ER;

rule S2S {
from s : ER!ERSchema
to   t : REL!RELSchema ( relations <- s.entities->union(s.relships) )}

rule E2R {
from s : ER!Entity
to   t : REL!Relation (name<-s.name, schema<-s.schema) }

rule R2R {
from s : ER!Relship
to   t : REL!Relation (name <-s.name, schema<-s.schema) }

rule EA2A {
from att : ER!ERAttribute, ent : ER!Entity (att.entity=ent)
to   t   : REL!RELAttribute (name<-att.name, isKey<-att.isKey, relation<-ent)}

rule RA2A {
from att : ER!ERAttribute, rs : ER!Relship (att.relship=rs)
to   t   : REL!RELAttribute (name<-att.name, isKey<-att.isKey, relation<-rs)}

rule RA2AK {
from att : ER!ERAttribute,
     rse : ER!RelshipEnd (att.entity=rse.entity and att.isKey=true)
to     t : REL!RELAttribute
                 (name<-att.name, isKey<-att.isKey, relation<-rse.relship)}
```

Fig. 3. Initial version of the ATL transformation ER2REL

three rules generate attributes for the relations. Both entity and relationship attributes are mapped to relation attributes (rules `EA2A` and `RA2A`). Furthermore, the key attributes of the participating entities are mapped to relation attributes as well (rule `RA2AK`).

All six rules of ER2REL are *matched rules*, which are the main constructs of ATL. A *matched rule* is composed of a source pattern and a target pattern. The source pattern specifies a set of objects of the source metamodel and uses, optionally, an OCL expression as a filtering condition. The target pattern specifies a set of objects of the target metamodel plus a set of bindings. The bindings describe assignments to features (i.e., attributes, references, and association ends) of the target objects. The execution semantics of matched rules can be described in three steps: First, the source patterns of all rules are matched against input model elements. Second, for every matched source pattern, the target pattern is followed to create objects in the target model. Notice that the execution of an ATL transformation always starts with an empty target model. In the third step, the bindings of the target patterns are executed. These bindings are performed straight-forwardly with one exception: If a value that is assigned to a property is an object of the input model, and if this object has been mapped by a rule in the previous step, then instead of the input object the (first) output object that has been created by this rule is used. By default, the ATL execution engine reports an error if no or multiple of such *matches* exist.

Next, in order to illustrate the ATL execution semantics, we explain how it works, for instance, for the rule `RA2A`. This rule is applied to every combination of an `ERAttribute att` and a `Relship rs` instance for which the condition `att.relship=rs` holds. For each such match, one `RELAttribute t` is created. The values of the `name` and `isKey` properties of `t` are simply copied from `att`. For the binding of the property relation, the implicit resolution strategy of ATL will replace the value of the input pattern element `rs` (which is an object of the source model) by a

reference to the Relation object that has been created by `R2R` for `rs`. In this case, `R2R` is the only rule that can be used to resolve `Relation`-objects. However, in general, there can be multiple rules for each type.

3 Transformation Models for ATL

Model transformations can be considered as programs that operate on instances of metamodels. In this sense, we can also apply notions of correctness for programs to model transformations. We will consider the input and output models of a transformation as valid if and only if they conform to the structure and to the constraints of their metamodels. Partial correctness then states that *if* the transformation produces an output model from a valid input model, that output model is valid as well. Total correctness extends this notion and states that the transformation produces a valid output for every valid input model (i.e., that the transformation terminates for every valid input model and does not abort with an error message).

Our notion of a transformation model $\mathcal{M}_T$ of a transformation $T : \mathcal{M}_I \rightarrow \mathcal{M}_F$ aims to support the verification of partial correctness of T using $\mathcal{M}_T$ as an equivalent surrogate as follows. A transformation model $\mathcal{M}_T$ is a metamodel (i.e., a structural specification of classes, associations, and constraints) that integrates $\mathcal{M}_I$ and $\mathcal{M}_F$ and additional structural modeling elements and constraints that capture the execution semantics of T. A pair of an $\mathcal{M}_I$ instance M_I and an $\mathcal{M}_F$ instance M_F is related by T if and only if there is an instance of $\mathcal{M}_T$, valid w.r.t. to all constraints, whose $\mathcal{M}_I$ part is M_I and whose $\mathcal{M}_F$ part is M_F. In practice, we want to loosen this equivalence to hold only for those M_I for which T terminates. However, for the declarative subset of ATL that we consider, recursive OCL helper operations are the only source of non-termination, as the actual execution of ATL rules is non-recursive and non-looping (and also deterministic [17]).

Having such an equivalent transformation model, we can verify partial correctness of T using 'off-the-shelf' model finders (e.g., based on SAT solving). In the remaining section, we show how to systematically derive such transformation models for ATL transformations. We provide a general algorithm for this (Sect. 3.1) and discuss the validity of our translation (Sect. 3.2).

3.1 An Algorithm to Derive Transformation Models for ATL

Our translation does cover a significant subset of ATL, namely *matched rules*, which are the workhorse of ATL, in the form provided in Fig. 4. We presume that all expressions and bindings in the transformation are correctly typed. We do not support imperative extensions, called or lazy rules at the moment, and we do not allow recursive OCL helper operations.

The algorithm that creates $\mathcal{M}_T$ for $T : \mathcal{M}_I \rightarrow \mathcal{M}_F$ is depicted in Fig. 5. It consists of four main steps. The results of the algorithm for ER2REL is shown in Fig. 6 (generated classes and associations) and Fig. 7 (generated constraints). The first step includes all elements (i.e., classes, associations, attributes, constraints) of $\mathcal{M}_I$ and $\mathcal{M}_F$. The second step adds a new class c_r for each rule r in T (step 1a; e.g., class 'S2S' in Fig. 6),

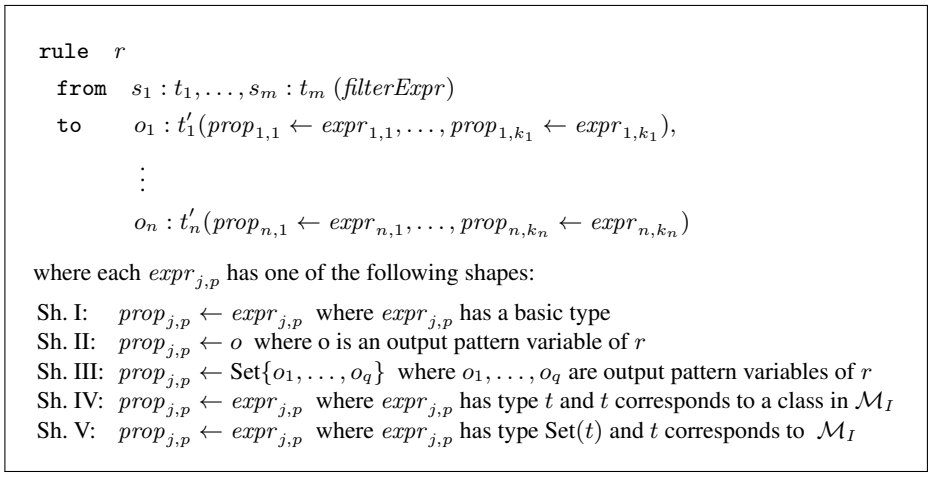


Fig. 4. ATL matched rule's patterns currently supported by our mapping.

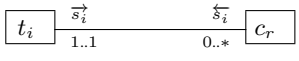
connects c_r to the types of the input and output pattern variables (steps 1b and 1c). Notice that for rules with multiple pattern elements, the same input object can participate several times (with different partners), hence the '0..*' multiplicity. In the next step we add *matching constraints* that ensure that exactly those combinations of M_I objects are connected to a c_r object that are matched by r (steps 1d and 1e; e.g., match_EA2A and match_EA2A_cond in Fig. 7). For each binding to an output pattern object, corresponding *binding constraints* over c_r are added (step 1f; e.g., bind_E2R_t_name). For unassigned properties, a constraint is added that ensures that these properties are null (step 2g). The third step considers each class in $\mathcal{M}_F$ and adds a *creation constraint* to ensure that each M_F object is created by exactly one rule of T (e.g., create_Relation in Fig. 7). The fourth step is specific to those transformations that have potentially overlapping patterns. Recall that ATL does not allow a combination of M_I objects to be matched by more than one rule (the engine would abort in this case). The fourth step corresponding *mutual exclusion constraints* for all pairs of potentially overlapping rules (ER2REL does not contain such rules).

We make use of some auxiliary functions in the description of the algorithm that generate OCL expressions for the more complex constraints. We define them below. To create the associations that connect the classes c_r to the resp. class in $\mathcal{M}_I$ and $\mathcal{M}_F$, we assume $\overrightarrow{s}$ and $\overrightarrow{o}$ to name the the corresponding navigable association ends for the pattern variables s and o (from the perspective of the rule class), and $\overleftarrow{s}$ and $\overleftarrow{o}$ to generate unique opposite association end names (from the perspective of the resp. classes in $\mathcal{M}_I$ and $\mathcal{M}_F$). We use the hat notation $\hat{z}$ to denote a fresh variable.

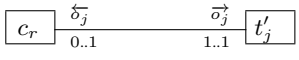
Auxiliary function $matchExpr(r)$. The function $matchExpr(r)$ that we use in step 1d yields a Boolean OCL expression of m nested 'forAll' expressions for the m input pattern elements of r such that for each combination of objects in M_I that matches r

1. Copy all model elements of $\mathcal{M}_I$ and $\mathcal{M}_F$.
2. For each matched rule r in T let $s_1 : t_1, \ldots, s_m : t_m$ denote the input pattern variables of r and $o_1 : t'_1, \ldots, o_n : t'_n$ the output pattern variables of r. Then:
   (a) Add a class c_r.
   (b) If $m = 1$ (i.e., r has only single input pattern variable), add an association

   Else, if $m > 1$, add the following association for each $1 \leq i \leq m$



   (c) For each output pattern variable $o_j : t'_j$ of r with $1 \leq j \leq n$ add an association



   (d) Add a constraint **context** t_1 **inv** : $matchExpr(r)$.
   (e) Add a constraint **context** c_r **inv** : $filterExpr'$ where $filterExpr' = filterExpr[s_1 \ldots s_m]/[\text{self}.\overrightarrow{s_1} \ldots \text{self}.\overrightarrow{s_m}]$ is the filter expression with all input pattern variables are replaced by navigations from the rule object.
   (f) For each binding $prop_{j,p} \leftarrow expr_{j,p}$ to an output pattern variable o_j of r with $1 \leq j \leq n$ and $1 \leq p \leq k_n$, add a constraint **context** c_r **inv** : $\text{self}.\overrightarrow{o_j}.prop_{j,p} = resolve[\![\, expr'_{j,p} \,]\!]$ where $expr'_{j,p} = [s_1 \ldots s_m]/[\text{self}.\overrightarrow{s_1} \ldots \text{self}.\overrightarrow{s_m}]$.

   If $expr_{j,p}$ is of shape IV furthermore add a constraint **context** c_r **inv** : $(expr'_{j,p} = \text{null}) = (resolve[\![\, expr'_{j,p} \,]\!] = \text{null})$.

   If $expr_{j,p}$ is of shape V furthermore add a constraint **context** c_r **inv** : $expr'_{j,p} \rightarrow \text{size}() = resolve[\![\, expr'_{j,p} \,]\!] \rightarrow \text{size}()$.
   (g) For each property $prop$ of o_j that is not bound by r, we add a constraint **context** c_r **inv** : $\text{self}.\overrightarrow{o_j}.prop = \text{null}$.
3. For each class c in $\mathcal{M}_F$, if $\{o_1 : t'_1, \ldots, o_q : t'_q\} = creators(c)$ then add a constraint **context** c **inv** : $\text{self}.\overleftarrow{o_1} \rightarrow \text{size}() + \cdots + \text{self}.\overleftarrow{o_q} \rightarrow \text{size}() = 1$.
   Otherwise, when there are no creators for c, add a constraint **context** c **inv** : false.
4. For each pair of rules r, r' in T that have input patterns of the same size m and each sequence of $\mathcal{M}_I$ types $t''_1, \ldots, t''_m$, add a mutual exclusion constraint if r and r' potentially overlap on $t''_1, \ldots t''_m$:
   context t_1 **inv** : $mutexExpr(r, r', \langle t''_1, \ldots, t''_m \rangle)$
   The rules r and r' overlap on $t''_1, \ldots t''_m$ when $t''_i \leq t_i$ and $t''_i \leq t'_i$ holds for each i with $1 \leq i \leq m$.

Fig. 5. Algorithm

exactly one instance of c_r is connected to these objects. It is defined as follows.

$$\begin{aligned} matchExpr(r) := t_1 \rightarrow \text{forAll}(\hat{s}_1 \mid t_2 \rightarrow \text{forAll}(\hat{s}_2 \mid \ldots t_m \rightarrow \text{forAll}(\hat{s}_m \mid \\ \textit{filterExpr}' \text{ implies } c_r.\text{allInstances}() \rightarrow \text{one}(\hat{z} \mid \\ \hat{z}.\overrightarrow{s_1} = \hat{s}_1 \text{ and } \ldots \text{ and } \hat{z}.\overrightarrow{s_m} = \hat{s}_m) \cdots) \end{aligned}$$

where $\textit{filterExpr}' = \textit{filterExpr}[s_1 \ldots s_m]/[\hat{s}_1 \ldots \hat{s}_m]$ is the filter expression of r in which the pattern variables are replaced by the variables used in the above iteration.

Auxiliary function $resolve[\![\, expr \,]\!]$. The function $resolve[\![\, expr \,]\!]$ that we use in step 1f is the most complex one. We use it to translate the implicit resolve mechanism of ATL into OCL. Recall that ATL, when processing a binding $prop \leftarrow expr$, replaces each object value from M_I by an object value from M_F. To do this, it uses the first output pattern variable of the (unary input pattern) rule that matched the resp. object in M_I. Let t be the type (for shape IV) resp. the element type (shape V) of $expr$. Let $\{(x_1 : t_1, y_1 : t'_1), \ldots, (x_q : t_q, y_q : t'_q)\}$ be the set of pairs $(x_i : t_i, y_i : t'_i)$ of the (only) input pattern variable and the first output pattern variable with $t \leq t_i$ or $t_i \leq t$, taken from all rules in T that have a unary input pattern (these are the rules that can map an object of type t). Notice that in this set we consider pattern variables of multiple rules in T.

- For shapes I, II, and III, no resolution is required, as the result is either a basic type or a (collection) value of M_F – recall that we have already replaced all target pattern variables o by self.$\overrightarrow{o}$ in step (2f). We have $resolve[\![\, expr \,]\!] := expr$.
- For shape IV we distinguish two cases. When we have $q = 1$ (there is only one rule that can possibly match this type), then we can translate the resolution into two simple navigation steps[6] (the type cast may be omitted when *expr* already has a sufficient specific type):

  $$resolve[\![\, expr \,]\!] := expr.\text{oclAsType}(t_1).\overleftarrow{x_1}.\overrightarrow{y_1}.$$

  When we have $q > 1$, then there are multiple potential rules to be used for this resolution step. Notice that there cannot be two rules applied at the same time (we guarantee this by mutual exclusion constraints), so there is at most one non-null element (M_F object) in the set expression below, and we can use the 'any' operator to deterministically select it.

  $$\begin{aligned} resolve[\![\, expr \,]\!] := \text{ Set}\{ & expr.\text{oclAsType}(t_1).\overleftarrow{x_1}.\overrightarrow{y_1}, \\ & \ldots, \\ & expr.\text{oclAsType}(t_q).\overleftarrow{x_q}.\overrightarrow{y_q}\} \rightarrow \text{any}(\hat{z} \mid \hat{z} <> \text{null}) \end{aligned}$$

- For shape V, the translation is similar to the previous one, but now we have to apply the resolution step to each element of the collection (using 'collect'). The intermediate result is a set that contains one bag (multi-set) for each rule that can

[6] Recall that only matched rules with unary input patterns are used, so $\overleftarrow{x_1}$ is an object-valued navigation, cf. step 2b

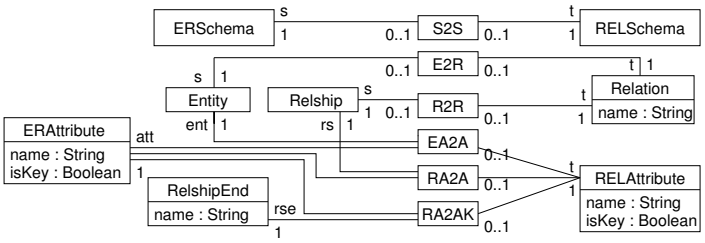


Fig. 6. Class diagram of the generated transformation model $\mathcal{M}_{\text{ER2REL}}$

potentially map t. We turn this into a flat set and remove the unmapped elements.

$$\begin{aligned} \mathit{resolve}[\![\ \mathit{expr}\ ]\!] := \text{Set}\{\ \mathit{expr} \rightarrow \text{collect}(\hat{z}|\hat{z}.\text{oclAsType}(t_1).\overleftarrow{x_1}.\overrightarrow{y_1}), \\ \ldots, \\ \mathit{expr} \rightarrow \text{collect}(\hat{z}|\hat{z}.\text{oclAsType}(t_q).\overleftarrow{x_q}.\overrightarrow{y_q})\} \\ \rightarrow \text{flatten}() \rightarrow \text{select}(\hat{z}|\hat{z} <> \text{null}) \end{aligned}$$

Auxiliary function $\mathit{creators}(c)$. We use $\mathit{creators}(c)$ to identify all locations where a class of $\mathcal{M}_F$ may be instantiated. This is the set of all output pattern variables $\{o_1 : t_1, \ldots, o_q : t_q\}$ from the set of all rules of T with $t_j \leq c$ for each j with $1 \leq j \leq q$.

Auxiliary function $\mathit{mutexExpr}(r, r', \langle t''_1, \ldots, t''_n \rangle)$. This function yields a mutual exclusion expression for a pair of potentially overlapping rules r, r' in step 4. Recall that each tuple of M_I objects can be matched by at most one rule, otherwise the ATL engine aborts. Let $s_1 : t_1, \ldots s_m : t_m$ and $s'_1 : t'_1, \ldots s'_m : t'_m$ be the input pattern variables of the rules r and r'. Let $t''_1, \ldots, t''_m$ denote a sequence of $\mathcal{M}_I$ object types that can be matched potentially by both r and r'. The function $\mathit{mutexExpr}(r, r', \langle t''_1, \ldots, t''_n \rangle)$ generates an Boolean OCL expression that states that no combination of instances of $t''_1, \ldots t''_m$ can be connected to both a c_r and a $c_{r'}$ instance.

$$\begin{aligned} &\mathit{mutexExpr}(s, s', \langle t''_1, \ldots, t''_n \rangle) := \\ &\quad t''_1.\text{allInstances}() \rightarrow \text{forAll}(\hat{s_1}| \cdots t''_m.\text{allInstances}() \rightarrow \text{forAll}(\hat{s_m}| \\ &\qquad \text{not}(\ c_r.\text{allInstances}() \rightarrow \text{exists}(\hat{z}|\hat{z}.\overrightarrow{s_1} = \hat{s_1} \text{ and } \cdots \text{ and } \hat{z}.\overrightarrow{s_m} = \hat{s_m}) \text{ and} \\ &\qquad\quad c_{r'}.\text{allInstances}() \rightarrow \text{exists}(\hat{z'}|\hat{z'}.\overrightarrow{s'_1} = \hat{s_1} \text{ and } \cdots \text{ and } \hat{z'}.\overrightarrow{s'_m} = \hat{s_m}))\cdots) \end{aligned}$$

3.2 Validity of the Translation

As said in the beginning of this section, a transformation model $\mathcal{M}_T$ shall be equivalent to T (for transformations that do not employ recursive helper operations), in order to use $\mathcal{M}_T$ as a surrogate to verify the (partial) correctness of T. Recall that we defined the notion of a transformation model as follows: A pair of an $\mathcal{M}_I$ instance M_I and an $\mathcal{M}_F$ instance M_F is related by T *if and only if* there is an instance of $\mathcal{M}_T$ whose $\mathcal{M}_I$ part is M_I and whose $\mathcal{M}_F$ part is M_F. As there is not official formal semantics for ATL so far, we cannot prove formally that our axiomatization is correct. However, we justify

```
-- constraints generated by steps 2d and 2e: matching constraints
context ERSchema inv match_S2S:
  ERSchema.allInstances()->forAll(x1 : ERSchema |
    S2S.allInstances()->one(z : S2S | z.s = x1))

context Entity inv match_E2R:
  Entity.allInstances()->forAll(x1 : Entity |
    E2R.allInstances()->one(z : E2R | z.s = x1))

context Relship inv match_R2R:
  Relship.allInstances()->forAll(x1 : Relship |
    R2R.allInstances()->one(z : R2R | z.s = x1))

context ERAttribute inv match_EA2A:
  ERAttribute.allInstances()->forAll(x1 : ERAttribute |
    Entity.allInstances()->forAll(l_ent : Entity | x1.entity=(l_ent) implies
      EA2A.allInstances()->one(z : EA2A | z.att = x1 and z.ent = l_ent)))
context EA2A inv match_EA2A_cond: self.att.entity = self.ent

context ERAttribute inv match_RA2A:
  ERAttribute.allInstances()->forAll(x1 : ERAttribute |
    Relship.allInstances()->forAll(x2 : Relship | x1.relship=x2 implies
      RA2A.allInstances()->one(z : RA2A | z.att = x1 and z.rs = x2)))
context RA2A inv match_RA2A_cond: self.att.relship = self.rs

context ERAttribute inv match_RA2AK:
  ERAttribute.allInstances()->forAll(x1 : ERAttribute |
    RelshipEnd.allInstances()->forAll(x2 : RelshipEnd |
      x1.entity=x2.entity and x1.isKey implies
      RA2AK.allInstances()->one(z : RA2AK | z.att = x1 and z.rse = x2)))
context RA2AK inv match_RA2AK_cond: self.att.entity = self.rse.entity and
     self.att.isKey

-- constraints generated by step 2f: binding constraints
context S2S inv bind_S2S_t_relations:  self.t.relations =
    Set{self.s.elements->collect(z|z.oclAsType(Entity).e2r.t),
        self.s.elements->collect(z|z.oclAsType(Relship).r2r.t)}
    ->flatten()->select(z|z <> null)

context E2R inv bind_E2R_t_name: self.t.name = self.s.name
context R2R inv bind_R2R_t_name: self.t.name = self.s.name

context EA2A inv bind_EA2A_t_relation: self.t.relation = self.ent.e2r.t
context EA2A inv bind_EA2A_t_name: self.t.name = self.att.name
context EA2A inv bind_EA2A_t_isKey: self.t.isKey = self.att.isKey

context RA2A inv bind_RA2A_t_name: self.t.name = self.att.name
context RA2A inv bind_RA2A_t_relation: self.t.relation = self.rs.r2r.t
context RA2A inv bind_RA2A_t_isKey: self.t.isKey = self.att.isKey

context RA2AK inv bind_RA2AK_t_isKey: self.t.isKey = self.att.isKey
context RA2AK inv bind_RA2AK_t_relation: self.t.relation =
     self.rse.relship.r2r.t
context RA2AK inv bind_RA2AK_t_name: self.t.name = self.att.name

-- constraints generated by step 3: creation constraints
context RELSchema inv create_RELSchema: self.s2s->size() = 1
context Relation inv create_Relation: self.e2r->size() + self.r2r->size() = 1
context RELAttribute inv create_RELAttribute:
  self.ea2a->size() + self.ra2a->size() + self.ra2ak->size() = 1

-- no constraints generated by step 4 (mutual exclusion constraints)
```

Fig. 7. Constraints of the generated transformation model $\mathcal{M}_{\text{ER2REL}}$

our OCL axiomatization informally. In the following, we consider the different aspects of the execution semantics of ATL matched rules and give reasons why our translation into OCL constraints is appropriate. For the sake of brevity we simply say 'M_T over M_I and M_F' to state that M_T is an instance of $\mathcal{M}_T$ whose $\mathcal{M}_I$ part is M_I and whose $\mathcal{M}_F$ part is M_F.

Abnormal termination. For the considered subset of ATL (well-typed matched rules, no imperative extensions, no recursive helper operations), the engine will always halt, and there are only two abnormal terminations of applying a transformation T to an input model M_I. The first one is when two or more rules are applied to the same tuple of M_I objects. Our translation prevents this by mutual exclusion constraints (generated in step 4). The second one abnormal termination condition is when an M_I object cannot be resolved to an M_F object when processing the bindings. This condition is excluded by the constraints generated in step 2f. Thus, when T aborts on M_I, there is no instance of $\mathcal{M}_T$ that completes M_I.

Matching. The constraints generated in step 2d require that every tuple of objects that matches the input pattern of a rule r must be connected to exactly on instance of c_r. The 1..1 multiplicities generated for the input associations for c_r ensure that no other instances of c_r exist. Thus, taking also into account that $\mathcal{M}_T$ does exclude M_I instances that would result in abnormal termination on multiple matches, the matching constraints in $\mathcal{M}_T$ encode exactly the matching semantics of ATL.

Binding and Resolution. In ATL, an M_F object can only be created by one rule, and only by this rule the properties of that object are assigned. This is mirrored one-to-one by the binding constraints we generate in step 2f. We already justified that our auxiliary function $resolve$ encodes the implicit resolution mechanism of ATL. Thus, taking also into account that $\mathcal{M}_T$ does exclude M_I instances that would leave unresolved references, the binding constraints in $\mathcal{M}_T$ encode exactly the binding semantics of ATL.

Frame problem. So far, we have justified by the matching and binding constraints that an instance M_T over M_I and M_F exists *if* $M_F = T(M_I)$. The creation constraints created in step 3 guarantee that M_T does not contain any $\mathcal{M}_F$ objects that are not generated by a rule (as the transformation always starts with an empty output model). Furthermore, step 2g guarantees that properties are null unless they are assigned by a rule. Together, this concludes the *if and only if* correspondence regarding T and $\mathcal{M}_T$.

4 Employing Model Finders to Verify ATL Transformations

Having translated an ATL transformation T into a purely structural transformation model $\mathcal{M}_T$ (i.e., a metamodel consisting of classes and their properties, and constraints), we can employ 'off-the-shelf' model finders (model satisfiability checkers) to verify partial correctness of T w.r.t. the metamodel constraints of $\mathcal{M}_F$ using $\mathcal{M}_T$.

In particular, we can check whether T might turn a valid input model M_I into an invalid output model M_F as follows: Let con_i with $1 \leq i \leq n$ denote the i-th

constraints of $\mathcal{M}_F$. Let $\mathcal{M}_{\overline{F}_i}$ denote a modified version of $\mathcal{M}_F$ stripped of all its constraints and having one new constraint $negcon_i$ that is the negation of con_i. Let $\mathcal{M}_{\overline{T}_i}$ denote the transformation model constructed for $T : \mathcal{M}_I \rightarrow \mathcal{M}_{\overline{F}_i}$. T is correct w.r.t. con_i if and only if $\mathcal{M}_{\overline{T}_i}$ has no instance. If such an instance exist, its $\mathcal{M}_I$ is a counter example for which T produces an invalid result.

4.1 Verification using UML2Alloy

We have implemented the presented translation as a so-called 'higher-order' ATL transformation, that is, an ATL transformation that takes an ATL transformation (the one to be verified, including the input and output metamodels) that produces the corresponding transformation model. The metamodels and constraints are technically represented using EMF and OCLinEcore. We employed the UML2Alloy model finder [1] to check the 'negated' transformation model (as explained before) for satisfiability.

UML2Alloy translates the metamodel and the OCL constraints into a specification for the Alloy tool, which implements bounded verification of relational logic. In the resulting specification, each class is represented as an Alloy signature each OCL constraint is represented by exactly one Alloy fact with the same name as the OCL constraint. Thus, we can check for the constraint subsumption easily by disabling and negating the facts (one after another) for the $\mathcal{M}_F$ constraints.

Table 1 shows the verification results for ER2REL. We verified all seven constraints of REL using an increasing number of objects per class (the maximum extent per signature must be specified when running Alloy). We can see that a counter example for (only) the constraints REL_AN can be found using at least three objects per class. This means that there exists a valid ER instance that is transformed into an invalid REL instance by ER2REL. Alloy presents the counter example in both an XML format and in a graphical, object-diagram like notation.

Figure 8 depicts such a counter example for REL_AN. Notice that the instances of the transformation model have a natural interpretation as a trace model of the original transformation, the counter example directly shows which objects are mapped by which rules. In Fig. 8, apparently, ER2REL does not treat reflexive relationships appropriately, while all attribute names are unique within their owning entities and relationships in the input model, the transformation generates identical attribute names within one relation in the output model. There are several ways to deal with this particular problem in ER2REL. As one solution we could modify rule RA2AK to use the name of the relationship end (instead of the key attribute) to determine the name of a foreign key attribute. But in this case, we must disallow combined keys, or we will get another violation of REL_AN in the next verification round. As a more general solution we could introduce qualified names for foreign keys (combining the name of the association end and the name of the key attribute). We leave it to the reader to decide what is the most appropriate solution for which situation. Instead we want to consider again Fig. 8 and emphasize the benefits of the counter examples that our method produces: The counter examples present at the same time the offending input model (that reveals the problem) and an explanation of the transformation execution (how the rules turn the input model into an invalid output model). In our view, this makes our method an intuitive and powerful tool for transformation developers.

Obj/Class	Obj/Total	REL_RN	REL_AN†	REL_K	REL_M1	REL_M2	REL_M3	REL_M4
2	28	0.06	* 0.06	0.05	0.05	0.05	0.7	0.05
3	42	0.15	0.11	0.10	0.11	0.11	0.11	0.09
5	70	3.12	0.51	0.70	0.40	0.21	0.52	0.20
7	98	38.62	0.58	4.21	1.21	0.54	3.93	0.48
10	140	543.93	1.70	136.61	4.96	1.53	17.03	1.33

Table 1. Avg. solving times (in seconds) using Alloy. Non-subsumed constraint marked with †. Undetected counter example marked by (*). All checks were conducted several times on a 2.2 Ghz office laptop running Alloy 4.1, Windows 7, and Java 7.

4.2 Scalability

Table 1 also provides some insights on the scalability of the verification method. Depending on the constraint, the verification time starts to become significant above 100 objects. Of course, these numbers are highly dependent on the constraint complexity. While the ER2REL example is simple in terms of the number of classes and associations, we consider it to have a comparatively high constraint complexity per class. We could confirm that larger class diagrams / larger instance sets do not necessarily increase the solving times, whereas harder (more overlapping, less tractable) constraints do. In this sense, we are confident that our method is applicable to larger metamodels as well. However, for the verification of industrial size metamodels and transformations, we expect that further heuristics and separation of concerns strategies will be required (e.g., metamodel pruning [23]).

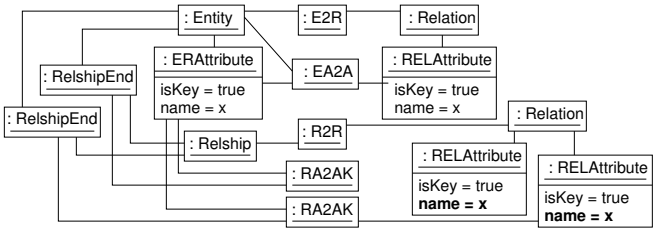


Fig. 8. Counter example: REL_AN violation

With respect to the chosen model verification tools (UML2Alloy and Alloy), it is important to remark again that these tools can only perform bounded verification. Thus, if Alloy cannot find a counter example, this does not mean that no counter example exists outside the fixed search bounds.

5 Related Work

We can relate our paper to several works. There are a couple of approaches that address partial, Hoare-style correctness of model transformation with respect to metamodel constraints as transformation pre- and postconditions. Inaba et al. automatically

infer schema (i.e., metamodel) conformance for transformations based on the UnCAL query language using the MONA solver [14]. The schema expressiveness in this approach is more restricted than OCL and describes only the typing of the graph. For example, uniqueness of names, as in ER and REL, could not be expressed. Asztalos et al. infer assertions for graph transformation-based model transformations [2]. They use an assertion language based on graph patterns, to enforce or avoid certain patterns in the model, which is a different paradigm than OCL. They provide explicit deduction rules for the verification (implemented in Prolog). On the contrary, we do not propose new deduction rules but rely on existing model finders. In similar vein, Rensink and Lucio et al. use model checking for the verification of first-order linear temporal [22] and pattern-based properties [19].

More specifically, there are also approaches that translate model transformations into transformation models in a similar fashion as we do: In a previous work, we translate triple graph grammars (which have a different execution semantics than ATL) and verify various conditions such weak and strong executability [7]. This work addresses executability but focuses on partial correctness (although we expect that executability could be expressed for ATL, too, using a tailored version of our algorithm). In similar vein Guerra et al. use triple graph grammar based transformation specifications and generate OCL invariants to check the satisfaction of these specifications by models [13]. To our knowledge, we are the only ones to present such a verification approach for ATL. Our paper is a successor of earlier results [4]. In that previous work, we gave a first sketch of the translation, but did not provide a complete algorithmic translation into OCL, as we do in our current contribution.

Related to the transformation model concept, the works of Braga et al., Cariou et al., and Gogolla and Vallecillo use OCL constraints to axiomatize properties of rule-based model transformation in terms of transformation contracts (but they do not generate them from a transformation specification as we do) [3,9,12].

To our knowledge, there are only two other approaches for the verification of ATL: First, Troya and Vallecillo provide a rewriting logic semantics for ATL and uses Maude to simulate and verify transformations, but do not consider the verification of Hoare style correctness [26]. Second, we recently presented an alternative approach to the formal verification of partial correctness of ATL using SMT solvers and a direct translation of the ATL transformation into first-order logic [5]. This approach is complementary to our current one and to other bounded verification approaches for ATL: It reasons symbolically and does not require bounds on the model extent, but it is incomplete (not all properties can be automatically decided this way, although it is refutationally complete in many cases). It can be used to verify several pre-post implications, but is not well suited to find counter examples. Furthermore, it builds on the translation of OCL to first-order logic by Egea and Clavel [10] which can only handle a subset of OCL. While the lightweight OCL axiomatization presented in our current work is fine for bounded model finders (and has an intuitive interpretation of counter examples as trace models), we were not able to employ SMT solvers for its verification. Using a direct translation of ATL+OCL into FOL [5] we could automatically prove several desired implications using the Z3 theorem prover solver (for the price that this approach requires of a full FOL encoding of ATL and OCL).

In this paper, we employed UML2Alloy [1] to perform the actual model verification. The community has developed several strong alternative approaches for the formal verification of models with constraints that we could use as well. They have in common that the model is translated into a formalism that has a well-defined semantics. Most approaches employ automated reasoning in the target formalism, for example, relational logic [18], constraint satisfaction problems [8], first-order logic [10], or propositional logic [24].

6 Conclusion and Future Work

In our paper, we have presented an approach that eases the verification of ATL transformations and thus helps to improve the quality of the MDE methodology in practice. As its core it is based on an automatic translation from ATL into a transformation model, which is a constrained metamodel that can be used as a surrogate for the verification of partial transformation correctness w.r.t. to the constraints of the input and output metamodels. We have presented a precise, executable description of the translation for a significant subset of ATL. We have also shown how this methodology can be implemented in practice using an ATL higher-order transformation and an 'off-the-shelf' model satisfiability checker (UML2Alloy). To our knowledge, we are the first ones to provide such an automatic approach for the verification of partial correctness for ATL.

We want to emphasize that the verification process can be automated as a "black box" technology, in the sense that the transformation developer is in contact only with models, in which the generated transformation models and their instances have a familiar representation for him.

In the future, we plan to explore the capabilities of different model finders as backends to our approach, in order to evaluate which are best suited for this kind of verification. Regarding ATL, we have already implemented an important subset of ATL, but we will incorporate (a restricted form) of so called *lazy rules*, which can be found in several transformations. Last but not least, comprehensive case studies must give more feedback on the applicability of our work.

References

1. Anastasakis, K., Bordbar, B., Georg, G., I.Ray: UML2Alloy: A Challenging Model Transformation. In: MoDELS 2007. LNCS, vol. 4735. Springer (2007)
2. Asztalos, M., Lengyel, L., Levendovszky, T.: Towards Automated, Formal Verification of Model Transformations. In: ICST'2010, Proc. pp. 15–24. IEEE Computer Society (2010)
3. Braga, C., Menezes, R., Comicio, T., Santos, C., Landim, E.: On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts. In: SBMF 2011. LNCS, vol. 7021. Springer (2011)
4. Büttner, F., Cabot, J., Gogolla, M.: On Validation of ATL Transformation Rules By Transformation Models. In: MoDeVVa'2011, Proc. ACM Digital Library (2012)
5. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: MoDELS'2012, to appear. LNCS, Springer (2012)
6. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: MoDELS 2006. LNCS, vol. 4199. Springer (2006)

7. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)
8. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Automated Software Engineering, ASE 2007, Proc. ACM (2007)
9. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. Electronic Communications of the EASST 24 (2009)
10. Clavel, M., Egea, M., de Dios, M.A.G.: Checking Unsatisfiability for OCL Constraints. Electronic Communications of the EASST 24, 1–13 (2009)
11. Gogolla, M.: Tales of ER and RE Syntax and Semantics. In: Transformation Techniques in Software Engineering. IBFI (2005), dagstuhl Seminar Proc. 05161
12. Gogolla, M., Vallecillo, A.: Tractable Model Transformation Testing. In: ECMFA 2011. LNCS, vol. 6698. Springer (2011)
13. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: A Visual Specification Language for Model-to-Model Transformations. In: 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010). pp. 119–126. IEEE Computer Society (2010)
14. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP, 2011, Proc. pp. 17–28. ACM (2011)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
16. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems, FMOODS 2006, Proc. LNCS, vol. 4037, pp. 171–185 (2006)
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Proc. of the Model Transformations in Practice Workshop at MoDELS 2005 (2005)
18. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: TOOLS 201. LNCS, vol. 6705, pp. 290–306. Springer (2011)
19. Lucio, L., Barroca, B., Amaral, V.: A Technique for Automatic Validation of Model Transformations. In: MODELS 2010, Part I. LNCS, vol. 6394. Springer (2010)
20. OMG: The Object Constraint Language Specification v. 2.2 (Document formal/2010-02-01). Object Management Group, Inc., Internet: http://www.omg.org/spec/OCL/2.2/ (2010)
21. OMG: Meta Object Facility (MOF) Core Specification 2.4.1 (Document formal/2011-08-07). Object Management Group, Inc., Internet: http://www.omg.org (2011)
22. Rensink, A.: Explicit State Model Checking for Graph Grammars. In: Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. LNCS, vol. 5065, pp. 114–132. Springer (2008)
23. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model Pruning. In: MODELS 2009, Proc. LNCS, vol. 5795, pp. 32–46. Springer (2009)
24. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer (2011)
25. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Longman, Amsterdam, 2nd edn. (2008)
26. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. Journal of Object Technology 10, 5: 1–29 (2011)
27. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, 2nd edn. (2003)