Language-independent model transformation verification.

LANO, Kevin, RAHIMI, Shekoufeh Kolahdouz and CLARK, Tony
<http://orcid.org/0000-0003-3167-0739>

Available from Sheffield Hallam University Research Archive (SHURA) at:
http://shura.shu.ac.uk/12068/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version


Copyright and re-use policy

See http://shura.shu.ac.uk/information.html
Language-Independent Model Transformation Verification

K. Lano, S. Kolahdouz-Rahimi, T. Clark

King’s College London; University of Middlesex

Abstract. One hinderance to model transformation verification is the large number of different MT languages which exist, resulting in a large number of different language-specific analysis tools. As an alternative, we define a single analysis process which can, in principle, analyse specifications in several different transformation languages, by making use of a common intermediate representation to express the semantics of transformations in any of these languages. Some analyses can be performed directly on the intermediate representation, and further semantic models in specific verification formalisms can be derived from it. We illustrate the approach by applying it to ATL.

1 Introduction

A large number of different transformation languages exist, ranging from declarative languages such as QVT-R [13] and Triple Graph Grammars, to hybrid languages such as ATL [6] and UML-RSDS [10], to imperative. There are nonetheless many commonalities between the processing carried out by transformations, regardless of the language they are written in.

In this paper we describe a general language-independent framework for transformation verification, and describe a range of language-independent verification techniques. The framework is based upon metamodels which provide language-independent representation of transformation specifications (Section 2). These representations can then be mapped via semantic mappings to suitable formalisms which support verification (such as theorem provers or constraint satisfaction checkers). This approach means that only one semantic mapping needs to be defined and verified for each target formalism, rather than semantic maps for each different transformation language and target formalism.

2 Metamodels for model transformations

Transformations operate on various models or texts which conform to some metamodel/language. There may be several input (source) models used by a transformation, and possibly several output (target) models. A transformation is termed update-in-place if a model is both an input and output, otherwise it is a separate-models transformation. Languages can be specified in many different ways, e.g., by UML class diagrams, by BNF syntax definitions, etc. Here we
will use UML class diagrams, together with OCL constraints. These form the concrete syntax of language descriptions.

Figure 1 shows a generic metamodel (termed $\mathcal{LMM}$) which can serve directly or indirectly as a metamodel (abstract syntax) for a wide range of modelling languages. The metamodel is also self-representative. $\text{EntityType}$ represents classes and interfaces, $\text{DataFeature}$ represents both attributes and associations. $\text{mult}1\text{lower}$ and $\text{mult}1\text{upper}$ refer to the multiplicity range of the source side of the attribute/association (i.e., the side with the entity type which owns the data feature), whilst $\text{mult}2\text{lower}$ and $\text{mult}2\text{upper}$ refer to the multiplicity range of the target side (the side with the type entity type). A value of $-1$ for an upper bound indicates a *-multiplicity at that end. $\text{isOrdered}$ refers to the ordering of the target end.

![Fig. 1. Minimal metamodel ($\mathcal{LMM}$) for modelling languages](image)

A metaclass $\text{Language}$ representing languages has a set $\text{entityTypes}$ of $\text{EntityType}$, a set $\text{features}$ of $\text{DataFeature}$, and a set $\text{constraints}$ of $\text{Constraint}$. Many variations on constraint languages could be used, here we adopt the subset of OCL 2.3 used in UML-RSDS [11, 12]. This includes set and sequence collections from OCL, operation calls and other feature application and navigation expressions, but omits $\text{null}$ and $\text{invalid}$. The set of all expressions formed over a given base language $L$ (a class diagram) is denoted $\text{Exp}(L)$. To support verification, we define a proof theory and (logical) model theory with respect to any language $L$, by associating a formal first order language and logic to each instance of $\mathcal{LMM}$. Table 1 gives examples showing how a formal first-order set theory (FOL) language $\mathcal{L}_L$ can be associated to instances $L$ of $\mathcal{LMM}$.

An optional association end (with $\text{mult}2\text{lower} = 0$ and $\text{mult}2\text{upper} = 1$) is formally represented as a set (or sequence) of size $0$ or $1$. A denumerable type
<table>
<thead>
<tr>
<th>Modelling concept</th>
<th>LMM representation</th>
<th>Formal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity type $E$</td>
<td>Element of EntityType</td>
<td>Type symbol $E'$, denoting the set of instances of $E$</td>
</tr>
<tr>
<td>Single-valued attribute $att : \text{Typ of } E$</td>
<td>DataFeature element with $\text{mult2upper} = 1$, $\text{mult2lower} = 1$ and type $\in \text{PrimitiveDataType}$</td>
<td>Function symbol $\text{att'} : E' \rightarrow \text{Typ'}$ where $\text{Typ'}$ represents $\text{Typ}$</td>
</tr>
<tr>
<td>Single-valued role $r$ of $E$ with target entity type $E1$</td>
<td>DataFeature element with $\text{mult2upper} = 1$, $\text{mult2lower} = 1$ and type $\in \text{EntityType}$</td>
<td>Function symbol $r' : E' \rightarrow E1'$</td>
</tr>
<tr>
<td>Unordered many-valued role $r$ of $E$ with target entity type $E1$</td>
<td>DataFeature element with $\text{mult2upper} \neq 1$ or $\text{mult2lower} \neq 1$ and isOrdered $= \text{false}$ and type $\in \text{EntityType}$</td>
<td>Function symbol $r' : E' \rightarrow \text{F}(E1')$</td>
</tr>
<tr>
<td>Supertype $E1$ of $E$</td>
<td>$E1 \in E\text{.general}$</td>
<td>Type $E1'$ with $E' \subseteq E1'$</td>
</tr>
</tbody>
</table>

**Table 1.** Correspondence of $LMM$ and first-order logics

Object $\text{OBJ}$ is included in each $L_L$ to represent the set of all possible object references. A finite set $\text{objects} \subseteq \text{ObjectOBJ}$ represents the set of all existing objects at any point in time. Each entity type $E$ has $E \subseteq \text{objects}$. Constraints in the expression language $\text{Exp}(L)$ are mathematically represented as first-order set theory axioms in $L_L$, for each base language $L$.

At the specification level, the effect of a transformation can be characterised by a collection of mapping specifications, which relate model elements of one or more models involved in the transformation to each other [5]. These mapping specifications define the intended relationships which the transformation should establish between the input (source) and output (target) models of the transformation, at its termination. That is, they define the postconditions $Post$ of the transformation. In the case of in-place transformations, the initial values of entity types and features can be notated as $E@pre, f@pre$ in postconditions to distinguish them from their post-state values. We adapt the mapping metamodel of [5] to represent transformation specifications (Figure 2). This metamodel is referred to as $TMM$ in the following.

In this figure, Mapping, TransformationSpecification, ModelEnd, and MappingEnd are subclasses of NamedElement. Each mapping has a corresponding computation step which defines the application of the mapping to specific source elements that satisfy its condition. Transformation specifications in declarative transformation languages can be expressed in this metamodel, using abstraction techniques such as those defined for TGG (triple graph grammars) and QVT-R in [4] and for declarative ATL in [3]; these techniques express the intended effect of transformations by OCL constraints describing the poststate of the transformation. In this paper we show how hybrid languages such as ATL can also be expressed in $TMM$. In turn, formal representations of transformation specifications can be generated from representations in $TMM$, into a range of formalisms such as B [9], Z3 [15] or Alloy [1], to support semantic analysis. The metamodel
Fig. 2. Transformation specification metamodel $\mathcal{TMM}$

could be extended to include generalisation relations between mappings, as in ATL and ETL [8].

The rules.relation constraints express the postconditions $Post$ of the transformation. Typically each mapping relation constraint $Cn \in Post$ has the form of an implication $SCond \implies Succ$ forall-quantified over elements (the source mapping ends $s : S$) of the source models. The assumptions express the preconditions $Asm$ of the transformation. The invariants define properties $Inv$ which should be true initially, and which should be preserved by each computation step of the transformation.

3 Verification techniques

The following verification techniques can be applied to the $\mathcal{TMM}$ representation of transformations, as follows:

- Syntactic analysis to identify the definedness conditions $def(Cn)$ and determinacy conditions $det(Cn)$ of each mapping constraint $Cn$. These are the conditions necessary for $Cn$ to have a defined and determinate value, respectively [10].
- Data dependency analysis, to compute the read-frame $rd(Cn)$ and write-frame $wr(Cn)$ for each mapping constraint $Cn$\(^1\) and to identify cases of invalid data-dependency and data-use.

\(^1\) $wr(Cn)$ is the set of feature names and entity type names which $Cn$’s implementation may modify. $rd(Cn)$ is the set which may be read.
– Syntactic analysis to establish confluence and termination in the case of transformations not involving fixed-point iteration.
– Translation to B AMN, to verify transformation invariants, syntactic correctness (that the transformation produces target models which conform to the target metamodel) and model-level semantic preservation (that the source and target models have equivalent semantics).
– Translation to Z3, to identify counter-examples to syntactic correctness.

Figure 3 shows an example of data dependency analysis being performed on the TMM representation of an ATL specification: two rules are identified as writing to the same entity type and feature, so are potentially in conflict.

4 Mapping from ATL to transformation metamodel

Table 2 shows the mapping of ATL (standard mode) [6] elements to elements of TMM. We assume that all rule inheritance has been statically expanded out so that only leaf rules exist.

We denote the TMM expression equivalent of an ATL expression $e$ by $e'$. Given an ATL module

```
module M;
create OUT : T from IN : S;
rule r1
{ from s1 : S1, ..., sm : Sm (SCond)
  using { vi : Typ1 = val1;  }
  to t1 : T1 (TCond1), ..., tk : Tk (TCondk)
  do (Stat)
}
...  
rule rn { ... }
```

```
the TMM representation of M is a transformation specification M′, and each matched rule ri is represented by two mappings ri,Matching and ri,Initialisation of M′, which both have readOnly mapping ends the sj : Sj. The tk : Tk are created mapping ends of ri,Matching, and updated ends of ri,Initialisation. The input conditions SCond are translated to a condition conjunct SCond′ of ri,Matching and ri,Initialisation. Any using variables are expressed by a conjunct vt = valt′ of the condition. The output variables become exists-quantified variables of the relations of the ri mappings. If an action block Stat is specified, then a new update operation \( \text{opi}(s2 : S2, ..., sm : Sm, t1 : T1, ..., tk : Tk) \) is introduced, this operation has activity given by the interpretation Stat′ of Stat as a UML activity. The resulting relation of ri,Matching has context S1 and predicate

\[ T1 \rightarrow \exists (t1 | t1.id = \text{id}) \text{ and } ... \text{ and } Tk \rightarrow \exists (tk | tk.id = \text{id}) \]

This creates target model elements corresponding to the source model elements that satisfy SCond′, and establishes tracing relations from s1 to t1, ..., tk by assignment of an identifier attribute \( \text{id} \), which is added to each root class of the source or target metamodels. The relation of ri,Initialisation has context S1 and predicate

\[ T1 \rightarrow \exists (t1 | t1.id = \text{id} \text{ and } ... \text{ and } Tk \rightarrow \exists (tk | tk.id = \text{id} \text{ and } TCond1' \text{ and } ... \text{ and } TCondk' \text{ and } s1.\text{opi}(s2, ..., tk))) \]

In this rule the previously created tj corresponding to s1 are looked-up and their features initialised. Lazy rules r with first input entity type S1 are translated to operations rop of S1. Calls thisModule.r(v1, ..., vm) to the rule are interpreted as calls v1.rop(v2, ..., vm) of this operation. The operation rop is stereotyped as \( \ll \text{cached} \gg \) in the case of unique lazy rules\(^2\). All ri,Matching rules are listed

2 Iterative target patterns are not treated by this translation. These are a deprecated feature of ATL since ATL 2.0.

### Table 2. Mapping of ATL to TMM

<table>
<thead>
<tr>
<th>ATL element</th>
<th>Transformation metamodel representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>TransformationSpecification</td>
</tr>
<tr>
<td>Matched rule</td>
<td>Mapping(s)</td>
</tr>
<tr>
<td>Lazy rule</td>
<td>Operation invoked from mapping constraint</td>
</tr>
<tr>
<td>Unique lazy rule</td>
<td>Operation using caching to avoid re-application</td>
</tr>
<tr>
<td>Implicit rule invocation</td>
<td>Object lookup using object indexing or traces</td>
</tr>
<tr>
<td>Using variable</td>
<td>Variable defined in mapping condition</td>
</tr>
<tr>
<td>Functional helper</td>
<td>Query operation</td>
</tr>
<tr>
<td>Attribute helper</td>
<td>Operation defining navigation expression</td>
</tr>
<tr>
<td>Action blocks</td>
<td>Operation with activity, invoked from mapping constraint</td>
</tr>
</tbody>
</table>
before all the $r_{Initialisation}$ rules in $M'_{rules}$. The translation from ATL to $T_{MM}$ can be extended to update-in-place transformations, i.e., to the refining mode of ATL (ATL 2010 version). The translation from ATL to $T_{MM}$ has been implemented in the UML-RSDS tools [11].

The translated transformation in $T_{MM}$ represents the usual execution semantics of the ATL transformation, in which target objects are created in a first phase, followed by links between objects [6]. Thus every computation of the translated specification corresponds to a computation of the original specification. The computation steps (rule applications) of the translation correspond to the steps of the original specification, so any invariant of the translation will also be an invariant of the original.

Using B, syntactic correctness can often be proved by using invariants of the transformation to relate the target model to the source model. Thus a proof of syntactic correctness of this kind also demonstrates syntactic correctness for the original transformation. Similarly, model-level semantic preservation can often be shown by invariant-based reasoning, which can be transferred from the translated specification to the original.

A counter-example produced by Z3 or another satisfaction-checking tool gives a pair $(m, n)$ of a source and target model which can result from completed execution of the translated transformation, and where $n$ violates some constraint of $T$. Such a pair is also a counter-example to syntactic correctness of the original ATL transformation. For refining mode transformations, termination and confluence of the translated specification establishes these properties of the original specification.

## 5 Evaluation

In this section we evaluate the approach by considering two example case studies: the simple UML to relational database example from the ATL Zoo, and the refactoring transformation of [7]. We use an imperative version of the UML to relational database ATL transformation. This has three lazy rules and one matched rule with an action block:

```plaintext
rule makeTable {
  from c : Entity ( c.generalisation->size() = 0 )
  to t : Table ( rdbname <- c.name )
  do {
    for ( att in c.ownedAttribute )
      { if ( att.isPrimary )
          { t.tcols->includes(thisModule.makePrimaryKeyColumn(att)); } }
    for ( att in c.ownedAttribute )
      { if ( att.type.oclIsKindOf(Entity) )
          { t.tcols->includes(thisModule.makeForeignKeyColumn(att)); } }
    for ( att in c.ownedAttribute )
      { if ( att.type.oclIsKindOf(PrimitiveType) or att.isPrimary = false )
```
lazy rule makePrimaryKeyColumn {
  from p : Property ( p.isPrimary )
  to c : Column ( rdbname <- p.name, type <- p.type.name )
}

lazy rule makeForeignKeyColumn {
  from p : Property ( p.type.oclIsKindOf(Entity) )
  to c : Column ( rdbname <- p.name, type <- "String" ),
                 f : FKey ( references <- p.type, fcols <- Set{ c } )
}

lazy rule makeColumn {
  from p : Property ( true )
  to c : Column ( rdbname <- p.name, type <- p.type.name )
}

The matched rule is translated to two constraints $C_0$ and $C_1$ with context Entity:

generalisation.size = 0  implies  
  Table->exists( t | t.$id = $id and t.rdbname = name )

generalisation.size = 0  implies  
  Table->exists( t | t.$id = $id and self.makeTable1op(t) )

$C_1$ has the condition and relation of makeTableMatching, and creates table objects and sets their attribute values. $C_2$ has the condition and relation of makeTableInitialisation, and looks up table objects using the introduced primary key $\$id$ and creates and links columns to the tables. makeTable1op(t) carries out the procedural code of makeTable’s action block, invoking operations makePrimaryColumnop, makeForeignColumnop and makeColumnop of Property corresponding to the lazy rules. For example:

makeColumnop(): Column
  post:
    Column->exists( c | c.$id = $id and c.rdbname = name and
                   c.type = type.name and result = c )

Figure 3 shows an example of data-dependency analysis on the translated specification. Verification properties of interest for this case study include: (i) termination; (ii) confluence (meaning that semantically equivalent source models are always mapped to equivalent target models); (iii) that the columns of the foreign keys are always contained in the set of columns of the tables, formalised as the transformation invariant $fcols \subseteq Table.tcols$ on $FKey$.

For this specification (i) can be shown by data-dependency analysis, establishing that a bounded iteration implementation is sufficient for $C_0$ and $C_1$. (ii)
fails because the association end Table :: tcols is ordered, so that different representations of the same input model may result in semantically distinct result models. Proof of (iii) can be carried out by establishing it as a transformation invariant in B. Table 3 shows the results of verification for the transformation. Atelier B version 4.0 was used.

<table>
<thead>
<tr>
<th>Property</th>
<th>ATL via TMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>By data-dependency analysis</td>
</tr>
<tr>
<td>Foreign key columns contained in table columns</td>
<td>20 proof obligations, 15 automatically proved</td>
</tr>
</tbody>
</table>

Table 3. Proof effort for refinement transformation

The refactoring transformation of [7] is an update-in-place transformation operating on simplified UML class diagrams. It removes apparent attribute clones from the diagram by applying ‘pull up feature’ refactoring. We use the extended version of ATL refining mode supported by our translator, to define an ATL solution with a single rule. For this transformation, there are several required correctness properties that should be verified: (i) termination; (ii) preservation of the property of single inheritance, formalised as the invariant \( \text{generalisation} \rightarrow \text{size}() \leq 1 \) on Entity; (iii) preservation of the property of no attribute name conflicts within classes, formalised as the invariant \( \text{ownedAttribute} \rightarrow \text{isUnique}(\text{name}) \) of Entity.

Since the transformation involves fixed-point iteration, termination is shown by proving that Property.allInstances() \( \rightarrow \text{size}() \) is a variant, i.e., that the number of Property instances in the model is always strictly decreased by applying the transformation rule. Table 4 shows the results of verification for the transformation.

<table>
<thead>
<tr>
<th>Property</th>
<th>ATL via TMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>6 proof obligations, 5 automatically proved</td>
</tr>
<tr>
<td>Single inheritance</td>
<td>13 proof obligations, all automatically proved</td>
</tr>
<tr>
<td>No name conflicts</td>
<td>15 proof obligations, 13 automatically proved</td>
</tr>
</tbody>
</table>

Table 4. Proof effort for refactoring transformation

6 Related work

In [2], techniques for the forward engineering of ATL and other MT language specifications from a platform-independent transformation representation are defined. The focus is on development of new transformations rather than upon the reverse-engineering and verification of existing transformations. We explicitly
represent the semantics of the MT languages within our language-independent representation, including execution phases and the mechanism of target element lookup/implicit rule invocation. Such semantic representation is not detailed in [2]. In contrast to the ATL to OCL translations of [3, 4], we define a behavioural representation of ATL transformations, capturing the semantics of transformation computation steps (individual rule applications), instead of a static post-state relation. This permits analysis of invariants and other behavioural properties.

7 Conclusion

We have outlined a language-independent approach for model transformation verification, and illustrated this approach by applying it to ATL. We also intend to apply this approach to other hybrid MT languages, such as ETL, Flock, GrGen.NET and QVT-O.

References

12. K. Lano, Null considered harmfull (for transformation verification), VOLT 2014.