Language-independent model transformation verification.

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

Published version

LANO, Kevin, RAHIMI, Shekoufeh Kolahdouz and CLARK, Tony (2014). Language-independent model transformation verification. In: AMRANI, Moussa, SYRIANI, Eugene and WIMMER, Manuel, (eds.) VOLT 2014 : verification of model transformations : proceedings of the Third International Workshop on Verification of Model Transformations co-located with Software Technologies: Applications and Foundations (STAF 2014), York, Uk, July 21, 2014. CEUR Workshop Proceedings (1325). Tilburg University, 36-45.

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. *EntityType* represents classes and interfaces, *DataFeature* represents both attributes and associations. *mult*1*lower* and *mult*1*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 *mult*2*lower* and *mult*2*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. *isOrdered* refers to the ordering of the target end.

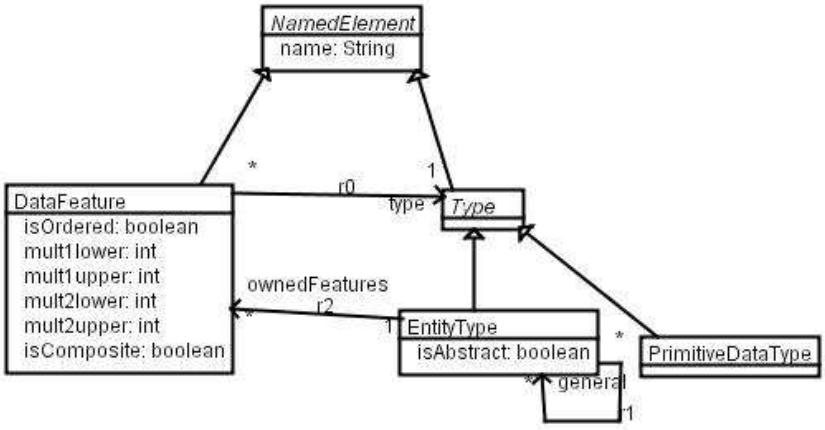

Fig. 1. Minimal metamodel ($\mathcal{LMM}$) for modelling languages

A metaclass *Language* representing languages has a set *entityTypes* of *EntityType*, a set *features* of *DataFeature*, and a set *constraints* of *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 *null* and *invalid*. The set of all expressions formed over a given base language L (a class diagram) is denoted $Exp(L)$. 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 *mult*2*lower* $= 0$ and *mult*2*upper* $= 1$) is formally represented as a set (or sequence) of size 0 or 1. A denumerable type

Modelling concept	$\mathcal{LMM}$ *representation*	*Formal semantics*
Entity type E	Element of *EntityType*	Type symbol E', denoting the set of instances of E
Single-valued attribute $att : Typ$ of E	*DataFeature* element with $mult2upper = 1$, $mult2lower = 1$ and $type \in PrimitiveDataType$	Function symbol $att' : E' \to Typ'$ where Typ' represents Typ
Single-valued role r of E with target entity type $E1$	*DataFeature* element with $mult2upper = 1$, $mult2lower = 1$ and $type \in EntityType$	Function symbol $r' : E' \to E1'$
Unordered many-valued role r of E with target entity type $E1$	*DataFeature* element with $mult2upper \neq 1$ or $mult2lower \neq 1$ and $isOrdered = false$ and $type \in EntityType$	Function symbol $r' : E' \to \mathbb{F}(E1')$
Supertype $E1$ of E	$E1 \in E.general$	Type $E1'$ with $E' \subseteq E1'$

Table 1. Correspondence of $\mathcal{LMM}$ and first-order logics

Object_OBJ is included in each $\mathcal{L}_L$ to represent the set of all possible object references. A finite set $objects \subseteq Object\_OBJ$ represents the set of all existing objects at any point in time. Each entity type E has $E \subseteq objects$. Constraints in the expression language $Exp(L)$ are mathematically represented as first-order set theory axioms in $\mathcal{L}_L$, 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 $\mathcal{TMM}$ 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 $\mathcal{TMM}$. In turn, formal representations of transformation specifications can be generated from representations in $\mathcal{TMM}$, into a range of formalisms such as B [9], Z3 [15] or Alloy [1], to support semantic analysis. The metamodel

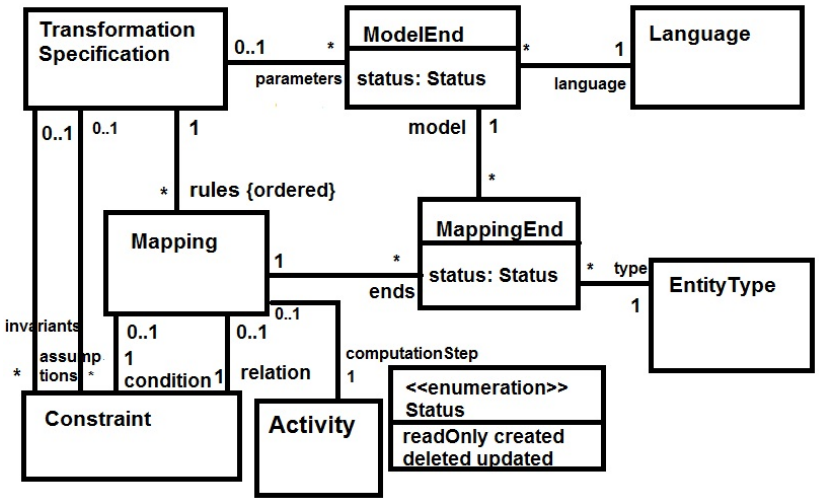

Fig. 2. Transformation specification metamodel $\mathcal{TMM}$

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 $s : S_i$) 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.

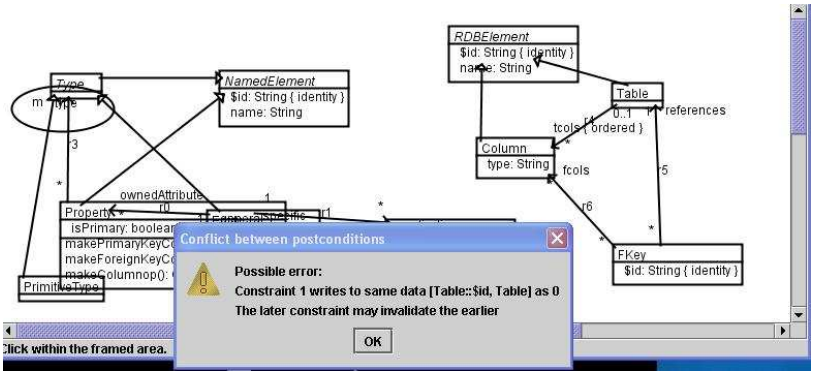

Fig. 3. Data dependency analysis example

Figure 3 shows an example of data dependency analysis being performed on the $\mathcal{TMM}$ 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 $\mathcal{TMM}$. We assume that all rule inheritance has been statically expanded out so that only leaf rules exist.

We denote the $\mathcal{TMM}$ 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 { v1 : Typ1 = val1; ... }
  to t1 : T1 (TCond1), ..., tk : Tk (TCondk)
  do (Stat)
}
...
rule rn { ... }
```

ATL element	*Transformation metamodel representation*
Module	TransformationSpecification
Matched rule	Mapping(s)
Lazy rule	Operation invoked from mapping constraint
Unique lazy rule	Operation using caching to avoid re-application
Implicit rule invocation	Object lookup using object indexing or traces
Using variable	Variable defined in mapping condition
Functional helper	Query operation
Attribute helper	Operation defining navigation expression
Action blocks	Operation with activity, invoked from mapping constraint

Table 2. Mapping of ATL to $\mathcal{TMM}$

the $\mathcal{TMM}$ representation of M is a transformation specification M', and each matched rule r_i is represented by two mappings $r_i Matching$ and $r_i Initialisation$ of M', which both have *readOnly* mapping ends the $sj : Sj$. The $tk : Tk$ are *created* mapping ends of $r_i Matching$, and *updated* ends of $r_i Initialisation$. The input conditions $SCond$ are translated to a condition conjunct $SCond'$ of $r_i Matching$ and $r_i Initialisation$. 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 r_i mappings. If an action block $Stat$ is specified, then a new update operation $opri(s2 : S2, ..., sm : Sm, t1 : T1, ..., tk : Tk)$ is introduced, this operation has activity given by the interpretation $Stat'$ of $Stat$ as a UML activity. The resulting relation of $r_i Matching$ has context $S1$ and predicate

$$T1 \rightarrow exists(t1 \mid t1.\$id = \$id) \ and \ ... \ and \ Tk \rightarrow exists(tk \mid tk.\$id = \$id)$$

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 $\$id$, which is added to each root class of the source or target metamodels. The relation of $r_i Initialisation$ has context $S1$ and predicate

$$\begin{aligned} T1 \rightarrow exists(t1 \mid t1.\$id = \$id \ and \ ... \ and \\ Tk \rightarrow exists(tk \mid tk.\$id = \$id \ and \ TCond1' \ and \ ... \ and \ TCondk' \ and \\ s1.opri(s2, ..., tk))...) \end{aligned}$$

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 $\ll cached \gg$ in the case of unique lazy rules[2]. All $r_i Matching$ rules are listed

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

before all the $r_i Initialisation$ rules in $M'.rules$. The translation from ATL to $\mathcal{TMM}$ can be extended to update-in-place transformations, i.e., to the *refining* mode of ATL (ATL 2010 version). The translation from ATL to $\mathcal{TMM}$ has been implemented in the UML-RSDS tools [11].

The translated transformation in $\mathcal{TMM}$ 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:

```
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 )
```

```
    { t.tcols->includes(thisModule.makeColumn(att)); }
    }
  }
} }

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 $C0$ and $C1$ 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) )
```

$C1$ has the condition and relation of *makeTableMatching*, and creates table objects and sets their attribute values. $C2$ 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 *makePrimaryKeyColumnop*, *makeForeignKeyColumnop* 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 $C0$ and $C1$. (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.

Property	*ATL via* $\mathcal{TMM}$
Termination	By data-dependency analysis
Foreign key columns contained in table columns	20 proof obligations, 15 automatically proved

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 apparant attribute clones from the diagram by applying 'pull up feature' refactorings. 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 $generalisation \rightarrow size() \leq 1$ on *Entity*; (iii) preservation of the property of no attribute name conflicts within classes, formalised as the invariant $ownedAttribute \rightarrow isUnique(name)$ of *Entity*.

Since the transformation involves fixed-point iteration, termination is shown by proving that $Property.allInstances() \rightarrow size()$ 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.

Property	*ATL via* $\mathcal{TMM}$
Termination	6 proof obligations, 5 automatically proved
Single inheritance	13 proof obligations, all automatically proved
No name conflicts	15 proof obligations, 13 automatically proved

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

1. K. Anastasakis, B. Bordbar, J. Kuster, *Analysis of Model Transformations via Alloy*, Modevva 2007.
2. V. Bollati, J. Vara, A. Jimenez, E. Marcos, *Applying MDE to the (semi-)automatic development of model transformations*, Information and Software Technology, 2013.
3. F. Buttner, M. Egea, J. Cabot, M. Gogolla, *Verification of ATL transformations using transformation models and model finders*, ICFEM 2012.
4. J. Cabot, R. Clariso, E. Guerra, J. De Lara, *Verification and Validation of Declarative Model-to-Model Transformations Through Invariants*, Journal of Systems and Software, 2010.
5. E. Guerra, J. de Lara, D. Kolovos, R. Paige, O. Marchi dos Santos, *transML: A family of languages to model model transformations*, MODELS 2010, LNCS vol. 6394, Springer-Verlag, 2010.
6. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, *ATL: A model transformation tool*, Sci. Comput. Program. **72**(1-2) (2008) 31–39.
7. S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, *Evaluation of model transformation approaches for model refactoring*, Science of Computer Programming, 2013, http://dx.doi.org/10.1016/j.scico.2013.07.013.
8. D. Kolovos, R. Paige, F. Polack, *The Epsilon Transformation Language*, in ICMT 2008, LNCS Vol. 5063, pp. 46–60, Springer-Verlag, 2008.
9. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparing verification techniques for model transformations*, Modevva workshop, MODELS 2012.
10. K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, vol 88, no. 2, February 2013, pp. 412–436.
11. K. Lano, *The UML-RSDS Manual*, www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf, 2014.
12. K. Lano, *Null considered harmfull (for transformation verification)*, VOLT 2014.
13. OMG, *MOF 2.0 Query/View/Transformation Specification*, 2011.
14. OMG, *Object Constraint Language 2.3 Specification*, 2012.
15. Z3 Theorem Prover, http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2012.