Reusing Model Transformations Through Typing Requirements Models

Juan de Lara[1], Juri Di Rocco[2(✉)], Davide Di Ruscio[2], Esther Guerra[1], Ludovico Iovino[3], Alfonso Pierantonio[2], and Jesús Sánchez Cuadrado[1]

[1] Universidad Autónoma de Madrid, Madrid, Spain
[2] University of L'Aquila, L'Aquila, Italy
juri.dirocco@univaq.it
[3] Gran Sasso Science Institute, L'Aquila, Italy

Abstract. Model transformations are key elements of Model-Driven Engineering (MDE), where they are used to automate the manipulation of models. However, they are typed with respect to concrete source and target meta-models and hence their reuse for other (even similar) meta-models becomes challenging.

In this paper, we describe a method to extract a *typing requirements model* (TRM) from an ATL model-to-model transformation. A TRM describes the requirements that the transformation needs from the source and target meta-models in order to obtain a transformation with a syntactically correct typing. A TRM is made of three parts, two of them describing the requirements for the source and target meta-models, and the last expressing dependencies between both. We define a notion of conformance of meta-model pairs with respect to TRMs. This way, the transformation can be used with any meta-model conforming to the TRM. We present tool support and an experimental validation of correctness and completeness using meta-model mutation techniques, obtaining promising results.

1 Introduction

Model-Driven Engineering [19] (MDE) employs models as first-class assets during the software development life cycle. Models are typically constructed using Domain-Specific Languages (DSLs), specially tailored to a particular domain. In MDE, the abstract syntax of a DSL is described through a meta-model, which describes the structure of the models considered valid. Therefore, it does not come as a surprise that meta-models proliferate in MDE as a means of formalising application domains [23]. Sometimes, these meta-models are variants of known languages like state-machines or workflow languages [17], for which services, like model transformations, already exist.

Model transformations are key to MDE, because they can leverage automation in model manipulation and management. Model transformations are typed with respect to the involved (source and target) meta-models. Therefore, reusing transformations is difficult, because they are not immediately applicable to other

M. Huisman and J. Rubin (Eds.): FASE 2017, LNCS 10202, pp. 264–282, 2017.
DOI: 10.1007/978-3-662-54494-5_15

meta-models, different from the ones they were initially conceived for. Hence, techniques to enhance transformation reusability are needed [1,15] since developing (non-trivial) transformations from scratch is typically a complex and time-consuming task.

Some works propose transformation reuse based on *concepts* [9] to express meta-model requirements, and bindings from those concepts into concrete meta-models. The binding induces an adaptation of the transformation, which becomes applicable to the concrete meta-models. However, concepts have limitations: they have to be manually created, and they present limited expressiveness, as for instance when variability must be described (e.g., when a feature can be typed according to a set of allowed types). Other approaches extract *effective meta-models* [20] by pruning unused typing information of the source/target domains according to the syntactical requirements in the transformation. Similarly to concept-based techniques, they also present limited expressiveness, although the procedure can be partly automated.

In this paper, we propose using a transformation *typing requirements model* (TRM) to express the syntactical needs of a transformation with respect to its source and target domains. TRMs support variability regarding, e.g., the concrete types of attributes, the inheritance relations between classes, the allowed targets for references, or the existence of classes with certain features but for which the class name is unknown or irrelevant. We propose an algorithm to automatically infer a TRM from an ATL model-to-model transformation, as ATL is one of the most widely used transformation languages nowadays [14]. Moreover, as ATL transformations consider several meta-models (typically source and target), dependencies between the allowed feature types in the source and target meta-models are required. This way, the transformation can be reused *as-is* with any pair of meta-models conforming to the extracted TRM.

The main advantages of TRMs with respect to existing techniques are: *(i)* TRM extraction is automatic; *(ii)* source and target meta-models are not needed to extract the TRMs; *(iii)* TRMs permit more expressive requirements (e.g., variability) that lead to improved reuse possibilities; and *(iv)* dependencies cross-linking requirements over source and target meta-models can be given in terms of feature models.

A preliminary evaluation is provided by means of a prototype tool. For this purpose, TRMs of third-party transformations have been extracted and variants of source and target meta-models have been defined by means of mutation techniques. The correctness and completeness of the method is empirically assessed by measuring the degree in which the transformation is correctly typed with meta-models conformant to the TRM, and incorrectly with meta-models not conformant to the TRM. Correctness of typing is checked with the anATLyzer tool as oracle [6]. The evaluation shows promising results, encouraging further investigation of transformation reuse based on TRMs.

Paper Organization. Section 2 discusses applicability scenarios. Section 3 introduces TRMs, and a notion of conformance. Section 4 explains how to extract TRMs from ATL transformations. Section 5 validates the approach over a set

of transformations developed by third parties. Section 6 compares with related work and Sect. 7 concludes the paper.

2 Motivating Scenarios and Running Example

Figure 1 describes our approach for model transformation reuse. Model transformations are typed with respect to source and target meta-models. However, these meta-models might not be available (e.g., for transformations found in code repositories like GitHub or BitBucket), or we might want to reuse the transformation with other meta-models, different from the ones the transformation was designed for. Therefore, given an existing transformation, we extract its typing requirements model (TRM, see label 1) that consists of three parts: the requirements for the source and target meta-models, and a compatibility model specifying the dependencies between them. The TRM can be used in different ways. For example, to query an existing meta-model repository in order to find conforming meta-model pairs (see 2a). In particular, such queries are OCL expressions [16], generated from the TRM. Any meta-model pair $\langle MM_s, MM_t \rangle$ conforming to the TRM can be used as source/target meta-models of the transformation. The TRM can also be used to generate suitable meta-model pairs (see 2b), so that the transformation can be executed on instances of them (see 3).

We illustrate our proposal using ATL since it is one of the most widely accepted transformation languages. However, the approach can be adapted to most of the existing model-to-model transformation languages. ATL [14] provides a mixture of declarative and imperative constructs to develop model-to-model transformations. Listing 1 shows our running example, partially taken from the ATL Zoo[1]. The transformation creates a table with the number of times each method in a piece of Java code is called within any declared method. The transformation is defined by a module specification consisting of a header section (lines 1–2), helpers (lines 4–7) and transformation rules (lines 9–23). The header specifies the source and target models of the transformation together with their corresponding meta-models. This way, the JavaSource2Table module is a one-to-one transformation, which generates a target model conforming to a Table meta-model from a source JavaSource model (see line 2).

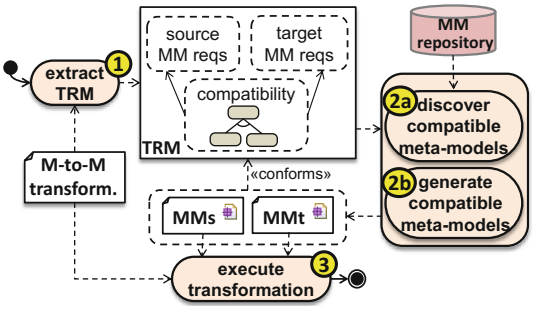

Fig. 1. Overview of our approach

[1] http://www.eclipse.org/atl/atlTransformations/#Java2Table.

```
1 module JavaSource2Table;
2 create OUT : Table from IN : JavaSource;
3
4 helper context JavaSource!MethodDefinition
5         def : computeContent(col : JavaSource!MethodDefinition) : String =
6             self.invocations->select(i | i.method.name = col.name
7             and i.method.class.name = col.class.name)->size().toString();
8
9 rule Table {
10     from s : JavaSource!ClassDeclaration
11     to t : Table!Table ( rows <- s.methods  )
12 }
13 rule MethodDefinition {
14     from m : JavaSource!MethodDefinition
15     to   row : Table!Row (
16         cells <- Sequence{JavaSource!MethodDefinition.allInstances()
17                 ->collect(md | thisModule.DataCells(md, m))}
18     )
19 }
20 lazy rule DataCells {
21     from md: JavaSource!MethodDefinition, m: JavaSource!MethodDefinition
22     to   cell: Table!Cell ( content <- m.computeContent(md) )
23 }
```

Listing 1. Fragment of a sample ATL transformation

Helpers and rules are the main ATL constructs to specify the transformation behaviour. The source pattern of rules (e.g., line 10) consists of types from the source meta-model. Thus, a rule gets applied for any instance of the given source types that satisfies the optional OCL rule guard. Rules also specify a target pattern (e.g., line 11) indicating the target objects created by the rule application, and a set of bindings to initialize their features (attributes and references). For example, the binding rows ← s.methods (line 11) initializes the rows feature of the target type Table with the elements created by the rules applied on the input elements referred by s.methods.

Rule MethodDefinition (lines 13–19) creates a target Row from each source MethodDefinition. The binding in this rule assigns to the reference cells a sequence of elements created by an OCL expression, which selects all the source MethodDefinition objects and apply on them the lazy rule DataCells. Differently from *matched* rules (like rules Table and MethodDefinition), lazy rules are executed only when explicitly called and use the passed parameters. The DataCell rule takes two MethodDefinition objects as input and generates a target Cell containing a number calculated by the helper of lines 4–7. Helpers are auxiliary operations that permit defining complex model queries using OCL. In particular, the helper computeContent returns a string with the number of occurrences of the received MethodDefinition object.

Our goal is to extract, from this transformation, a description (a TRM) of the features needed in source and target meta-models for the transformation to work. This way, the transformation can be reused with any meta-model satisfying the TRM, and not just with the ones used for its definition. Details about the TRM are given in Sect. 3, whereas the algorithm able to extract TRMs from ATL transformations is detailed in Sect. 4.

3 Representing Transformation Typing Requirements

This section explains how we describe transformation typing requirements through TRMs. TRMs contain three parts, two describing the typing requirements from source and target meta-models (Sect. 3.1), and a compatibility model relating both (Sect. 3.2).

3.1 Describing Single Meta-Model Requirements

Domain Typing Requirements. We use the meta-model in Fig. 2 to represent structural requirements for single meta-models. Its instances, called *domain typing requirements models* (DRMs), resemble meta-models but where some decisions can be left open if they are irrelevant for the problem at hand, like class names, the type of attributes, the target of references, or the cardinality of features. This way, a potentially infinite set of meta-models may conform to a DRM.

We consider two kinds of classes: named and anonymous. While the former have a name, in the latter the name is irrelevant as the class can have any name. Classes can be flagged as abstract, for which we use a three-valued enum type UBoolean which allows us to require the class to be abstract, concrete or any. A class defines a collection of features. The flag mandatoryAllowed permits a class to have more mandatory fields than those indicated in collection feats, while there is no constraint concerning the number of extra non-mandatory fields. A class may defer the conformance checking to all its concrete subclasses, which is indicated by the subsAllowed flag. A class may be required to inherit (directly or indirectly) from another class, and this is specified through relation ancs. Conversely, a class is forbidden to inherit from those in relation antiacs. More precisely, if B ∈ A.antiancs, then we reject meta-models in which B is an ancestor of A, or both share a common subclass.

Features have minimum and maximum cardinality, which can be a number, many, or we might allow any cardinality. If the maximum is many, it can also be

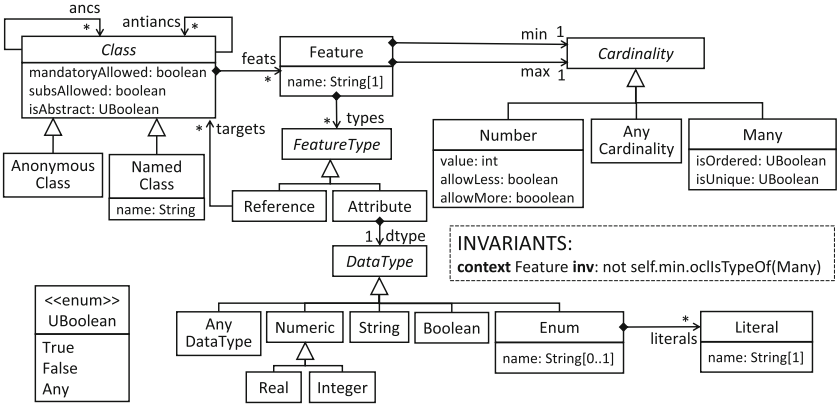

Fig. 2. Domain typing requirements meta-model (excerpt)

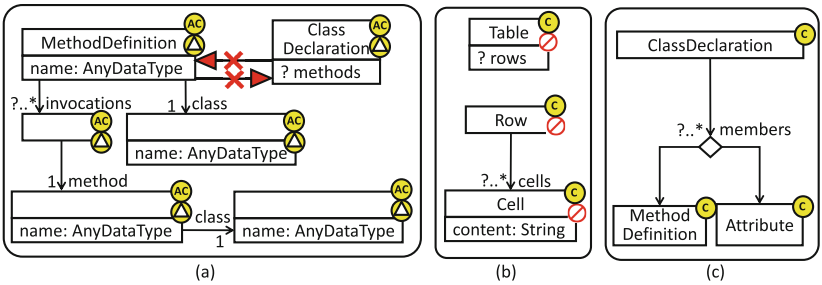

Fig. 3. DRM examples: (a) Source DRM of Listing 1. (b) Target DRM of Listing 1. (c) Multiple compatible reference targets.

specified whether the feature should be ordered or unique using UBoolean values. For the case of a number, we can define whether the cardinality is allowed to be lower (allowLess) or upper (allowMore) than this number. Features always have a name, and optionally, they may have a type which can be Reference, Attribute or both. References can indicate the admissible compatible target types, some of which can be anonymous classes. Attributes can specify their data type, or it can be left open using the AnyDataType class.

Example. Figure 3 shows three DRM examples. A specific concrete syntax has been adopted to denote additional characteristics. In particular, in the upper-right corner of a class is specified whether *(a)* it can be either abstract or concrete (AC), only abstract (A), or only concrete (C); *(b)* it can defer the conformance checking to its subclasses (encircled inheritance-like triangle); and finally *(c)* it forbids extra mandatory features (crossed-out circle). In addition, the anti-ancestor relation is shown as a crossed-out red inheritance relation.

DRM (a) has been extracted from the source domain of the transformation in Listing 1. The extraction procedure is described in Sect. 4. The DRM requires two classes named ClassDeclaration and MethodDefinition, which cannot inherit from each other otherwise the transformation would raise a runtime error due to multiple matches on the same element, that is not allowed in ATL. The latter class should have an attribute name whose type can be any, and two references named class and invocations to anonymous classes (i.e., their name is unimportant). The lower bound of invocations is open. In its turn, ClassDeclaration requires a feature methods which can be an attribute or a reference (we use a "?" prefix to denote this). The DRM also demands four anonymous classes for which only certain features are required. These classes could be matched by the same or different classes in concrete meta-models, or even by the same classes conforming to the named classes.

DRM (b) has been extracted from the target domain of Listing 1. It requires three named concrete classes. Class Table requires a feature rows which can be an attribute or a reference. As shown in Sect. 3.2, the transformation requires the types of Table.rows and ClassDeclaration.methods in DRM (a) to be correlated, for

which we will introduce a compatibility model. None of the classes are allowed to have extra mandatory features (which is represented with a crossed-out circle).

Finally, DRM (c) shows that a reference can be required to be compatible with several target types. In a concrete meta-model, this could be realized by reference members targeting a (possibly indirect) common superclass of MethodDefinition and Attribute.

Meta-Model Conformance. Next, the notion of conformance of a meta-model with respect to a DRM is introduced. For this purpose we define predicate $conf_{MM}$ which applies to a requirements model RM and a meta-model MM, and checks if for every Class in RM, there is a conforming class in MM.

$$conf_{MM}(RM, MM) \triangleq \forall RC \in RM \; \exists C \in MM \; \bullet \; conf_C(RC, C) \tag{1}$$

where RC is a Class in RM, C is a class in MM, and we use the predicate $conf_C$ to check conformance of the latter with respect to the former. As defined in Eq. (2), this accounts to assessing conformance of names ($conf_{name}$), abstractness ($conf_{abs}$), features ($conf_{feat}$) and ancestors ($conf_{ancs}$). In the case of abstract meta-model classes, compatibility may also come from the compatibility of all their concrete subclasses ($conf_{subs}$). Instead, for concrete meta-model classes, there is no need to check the compatibility of their subclasses because if a class is conformant, so will be its subclasses as they inherit the class features and ancestors. In the following equations, we use *isTypeOf* to check if the type of an object is compatible with the given type parameter. Moreover, given a class $C \in MM$, $C.feats^*$ yields its owned and inherited features, $C.ancs^*$ yields its direct and indirect superclasses, and $C.subs^*$ yields the set of its direct and indirect subclasses including C.

$$\begin{aligned} conf_C(RC, C) \triangleq \; & conf_{name}(RC, C) \wedge \\ & (\; (conf_{abs}(RC, C) \; \wedge conf_{feat}(RC, C) \; \wedge conf_{ancs}(RC, C)) \vee \\ & \quad conf_{subs}(RC, C) \;) \end{aligned} \tag{2}$$

$$conf_{name}(RC, C) \triangleq \; RC.isTypeOf(AnonymousClass) \vee RC.name = C.name \tag{3}$$

$$\begin{aligned} conf_{abs}(RC, C) \triangleq \; & (RC.isAbstract = true \implies C.isAbstract = true) \wedge \\ & (RC.isAbstract = false \implies C.isAbstract = false) \end{aligned} \tag{4}$$

$$\begin{aligned} conf_{feat}(RC, C) \triangleq \; & \forall rf \in RC.feats \; \exists f \in C.feats^* \bullet conf_F(rf, f) \wedge \\ & \neg RC.mandatoryAllowed \implies \\ & \quad |\{fm \mid fm \in C.feats^* \wedge fm.isMand\}| \\ = \; & \quad |\{fm \mid fm \in RC.feats \; \wedge fm.isMand\}| \end{aligned} \tag{5}$$

$$\begin{aligned} conf_{ancs}(RC, C) \triangleq \; & \forall RC_A \in RC.ancs, \; \exists C_A \in C.ancs^* \bullet conf_C(RC_A, C_A) \wedge \\ & \forall RC_A \in RC.antiancs, \forall \; C' \in MM \; \bullet \; conf_C(RC_A, C') \implies \\ & \quad C' \notin C.ancs^* \wedge \nexists C'' \in MM \; \bullet \; \{C, C'\} \subseteq C''.ancs^* \end{aligned} \tag{6}$$

$$conf_{subs}(RC, C) \triangleq \begin{array}{l} RC.subsAllowed \ \wedge RC.isAbstract \in \{any, true\} \ \wedge \\ C.isAbstract = true \ \wedge \\ \forall C' \in C.subs^* \ \bullet C'.isAbstract = false \implies \\ \quad (conf_{feat}(RC, C') \ \wedge conf_{ancs}(RC, C')) \end{array} \quad (7)$$

In particular, predicate $conf_{name}$ (Eq. (3)) requires classes to have the same name, or if RC is an AnonymousClass, no name checking is performed. Predicate $conf_{abs}$ (Eq. (4)) checks compatibility of the isAbstract flag, which may have value true, false or any. Equation (5) checks that every feature of RC is matched by some owned or inherited feature in C. If RC forbids additional mandatory features (i.e., mandatoryAllowed is false), then the set of mandatory features of C should be exactly that required by RC[2]. We use $f.isMand$ to check if feature f is mandatory. Equation (6) checks that the ancestor set of C includes classes matching those in $RC.ancs$, and none from $RC.antiancs$. Finally, $conf_{subs}$ checks conformance when RC allows abstractness and subsAllowed is true. In that case, if C is abstract, then the conformance relation is required for all its concrete subclasses. Typically, subsAllowed will be true on classes of the input transformation domain, whenever no isTypeOf OCL operator is used on them.

For features, the $conf_F$ predicate in Eq. (8) checks the conformance of their names (which are always known), cardinalities (using predicates $conf_{min}$ and $conf_{max}$), and types (either there is no type requirement, in which case any reference and attribute would match, or some allowed type in types should match as reference or as attribute).

$$conf_F(rf, f) \triangleq \begin{array}{l} rf.name = f.name \ \wedge conf_{min}(rf, f) \wedge conf_{max}(rf, f) \ \wedge \\ (\ rf.types = nil \ \vee \exists t \in rf.types \ \bullet \\ \quad (t.isTypeOf(Reference) \ \wedge conf_{ref}(t, f)) \ \vee \\ \quad (t.isTypeOf(Attribute) \ \wedge conf_{att}(t, f))) \end{array} \quad (8)$$

A reference $f \in MM$ matches $t \in RM$ if, in addition to the conditions in Eq. (8), both have compatible target types. This is so if t.targets is empty as any target type would be valid, or if every target in t.targets is matched by the target class of f or a subclass. Predicate $conf_{ref}$ in Eq. (9) checks this compatibility condition. Similarly, predicate $conf_{att}$ (omitted) checks the compatibility of attribute types. Hence, this predicate holds if no specific attribute type is required, if it is AnyDataType, or if the type of f is compatible with that of rf.

$$conf_{ref}(t, f) \triangleq \ \forall RC \in t.targets \ \bullet \ \exists D \in f.target.subs^* \ \bullet conf_C(RC, D) \quad (9)$$

We omit the formulation of predicates $conf_{min}(rf, f)$ and $conf_{max}(rf, f)$ for space constraints. The former holds when the required minimum cardinality of a feature is AnyCardinality, or when the minimum cardinalities of f and rf are the same (or less or more if allowed). The latter predicate is similar but for the

[2] For simplicity, this formalization ignores opposite references. In practice, we exclude from this set the mandatory features which are opposite of already matched references.

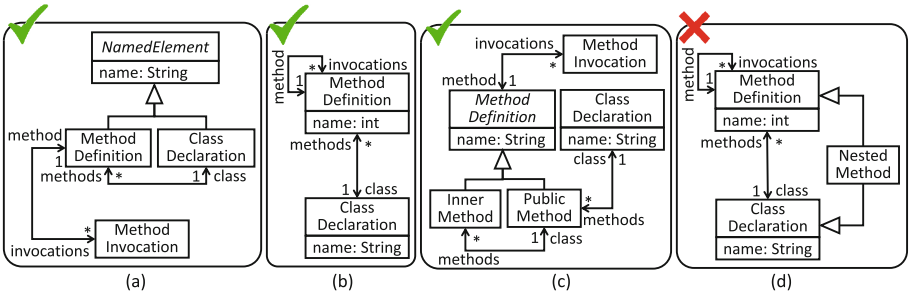

Fig. 4. Conformance examples with respect to DRM (a) in Fig. 3.

maximum cardinality of features. Moreover, in this case, rf can also be Many (a collection), for which (non-)uniqueness and (non-)ordered is checked if required.

Example. Figure 4 shows conforming (a, b, c) and non-conforming (d) meta-models with respect to DRM (a) in Fig. 3. Meta-model (a) conforms to the DRM because both MethodDefinition and ClassDeclaration inherit a name attribute from NamedElement. Moreover, MethodInvocation conforms to one of the anonymous classes in the DRM, MethodDefinition conforms to another anonymous class, and ClassDeclaration to two of them. The feature methods in the DRM, which can be either a reference or an attribute, has been matched by the meta-model reference ClassDeclaration.methods.

Meta-model (b) also conforms to the DRM. In this case, the name attribute is directly owned by the classes and has different types. In addition, there is no class MethodInvocation, whose role is played by MethodDefinition. This way, the four anonymous classes in the DRM are matched by the two meta-model classes. Meta-model (c) is conforming because all concrete subclasses of the abstract class MethodDefinition structurally conform to MethodDefinition in the DRM. Finally, meta-model (d) does not conform because NestedMethod inherits from both MethodDefinition and ClassDeclaration, which is forbidden by the antiancs relations in the DRM. With reference to the transformation in Listing 1, some instances of this meta-model could cause a runtime error as NestedMethod objects would be matched by rules Table and MethodDefinition.

In essence, the proposed conformance relation performs a structural comparison of classes, as required features can be owned or inherited by meta-model classes. However, it does not rely on an explicit mapping between classes and features of RM and MM. While several classes in a meta-model may conform to an anonymous class in RM, our conformity just checks that any such class exists. An explicit definition of the mapping, allowing adaptation (e.g., class renamings) through adapters [4], is left for future work.

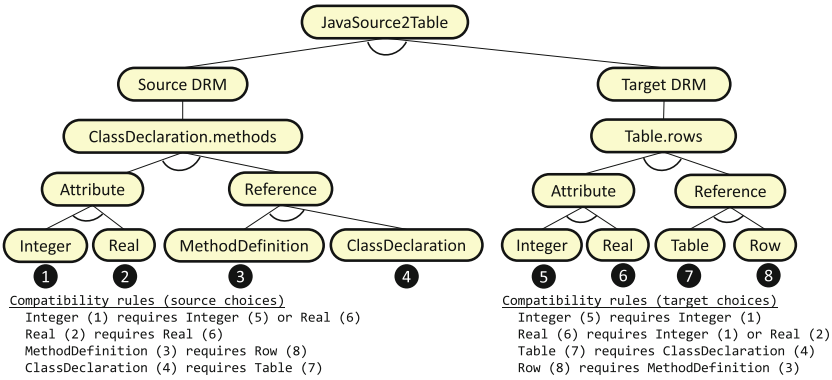

Fig. 5. Excerpt of the compatibility model for the running example.

3.2 Expressing Compatibility Requirements

The DRM implicitly describes possible choices for a concrete meta-model to satisfy the conformance relationship introduced above. However, a given choice for an open element of the source (or target) DRM may forbid some choices of the target (or source) DRM in case such choices break the syntactic correctness of the transformation. For instance, in Listing 1, the binding rows ← s.methods constrains the possible types of the rows and methods features to those that yield a non-faulty execution.

Hence, we gather the inter-dependencies between the source and target DRMs in a *compatibility model* which makes explicit how the choices for one DRM restrict the choices in the other DRM. We represent this compatibility model as a feature model where the different choices are depicted as nodes and the compatibility requirements are shown as dependencies between child nodes, so that the occurrence of a child node forces the presence of the dependent nodes.

Figure 5 shows an excerpt of the compatibility model for the running example, which focuses on the admissible types for attributes (i.e., data types) and references (i.e., target classes). Feature ClassDeclaration.methods can be either an attribute or a reference, as it is only used in line 11 as part of a binding. If it is an attribute, then it can have any data type (the figure only shows Integer and Real). However, the particular selection restricts the choices for feature Table.rows in the target DRM to keep the transformation syntactically correct. Similarly, if methods is a reference with type MethodDefinition, then the type of Table.rows must be Row because, otherwise, the binding will assign an incorrect target value. These dependencies also work from target to source.

4 Extracting Typing Requirements from ATL Transformations

This section explains the procedure for extracting TRMs out of existing ATL transformations. To this end, we rely on the Attribute Grammar formalism,

Table 1. Fragment of the developed ATL attribute grammar (AG_{ATL})

#	Productions	Computation Rules
p1	⟨matchedRule⟩::= **rule** ID { ⟨inPattern⟩ ⟨outPattern⟩* }	
p2	⟨inPattern⟩::= **from** ⟨inPatternElement⟩*	
p3	⟨InPatternElement⟩::= ID:⟨oclModelElement⟩	*type*(⟨InPatternElement⟩) ← *addClassToSourceDRM*(*type*(⟨oclModelElement⟩))
p4	⟨outPattern⟩::= **to** ⟨outPatternElement⟩	
p5	⟨OutPatternElement⟩::= ID:⟨oclModelElement⟩ (⟨binding⟩*)	*type*(⟨OutPatternElement⟩) ← *addClassToTargetDRM*(*type*(⟨oclModelElement⟩))
p6	⟨binding⟩::= ID '<-' ⟨oclExpression⟩;	*leftType* ← *createFeature*(*name*(ID), *type*(⟨oclExpression⟩)) *rightType* ← *type*(⟨oclExpression⟩) *type*(⟨bindings⟩) ← *addClassToTargetDRM*(*owner*(*leftType*)) *analyseCompatibilityNodes*(*leftType*, *rightType*)
p7	⟨oclModelElement⟩::= ID_1 !ID_2	*type*(⟨oclModelElement⟩) ← *createClass*(*name*(ID_2))
p8	⟨oclExpression⟩::= ⟨navigationOrAttributeCallExp⟩ | ⟨oclModelElement⟩ | ...	

p9	⟨navigationOrAttributeCallExp⟩::= ⟨oclExpression⟩.ID;	*type*(⟨oclExpression⟩) ←
		if (*isNavigationOrAttributeCallExp*(⟨oclExpression⟩) **then**
		createReference(*type*(⟨oclExpression⟩), "AnonymousClass")
		type(⟨navigationOrAttributeCallExp⟩) ←
		if (*isOperation*(name(ID))) **then**
		createFeatureByOperation(
		name(ID), *getReferenceClass*(⟨oclExpression⟩))
		else
		createFeature(*name*(ID), *getReferenceClass*(⟨oclExpression⟩)))

which represents an elegant and powerful mechanism to describe computations over syntax trees [21].

Attribute grammars extend context-free grammars by associating *attributes* with the symbols of the underlying context-free grammar. The values of such attributes are computed by rules, which are executed while traversing the syntax trees as needed. More formally, let $G = (N, T, P, S)$ be a context-free grammar for a language L_G where N is the set of non-terminals, T is the set of terminals, P is the set of productions, and $S \in N$ is the start symbol. An *attribute grammar* AG is a triple (G, A, AR), where G is a context-free grammar, A associates each grammar symbol $X \in N \cup T$ with a set of attributes, and AR associates each production $R \in P$ with a set of attribute computation rules. While traversing syntax trees, values can be passed from a node to its parent (by means of *synthesized attributes*), or from the current node to a child (by means of *inherited attributes*). Attribute values can be assigned, modified, and checked at any node in the considered syntax tree.

Table 1 shows a fragment of the ATL attribute grammar (AG_{ATL}) we have developed to create TRMs while traversing the syntax tree of the considered ATL transformations. It is important to remark that the shown grammar is a simplification of the real one. The aim of such a simplification is to give a flavour of how the proposed extraction mechanism works, without compromising the readability of the explanation. However, the developed tool

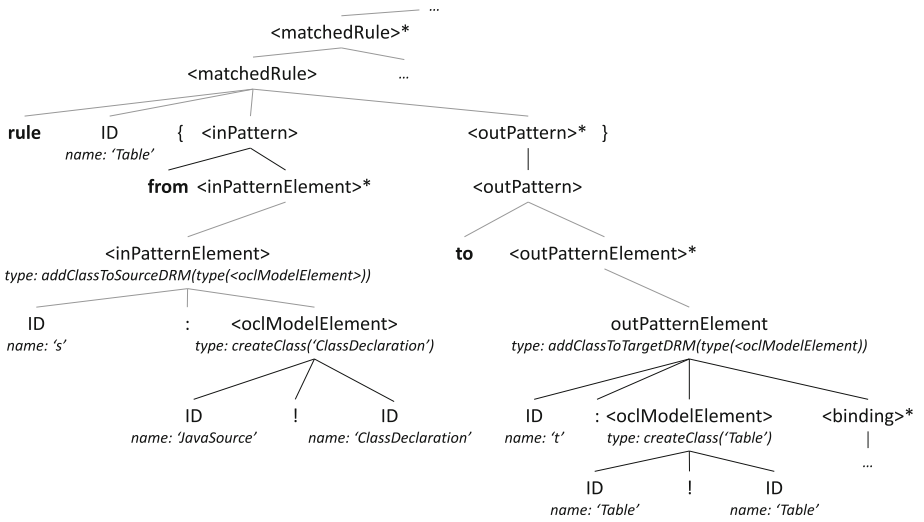

Fig. 6. A sample AG_{ATL} parse tree

available online[3] takes into account all the productions defined for the actual AG_{ATL} by implementing all the concepts presented in Sect. 3.

The first column of Table 1 contains ATL grammar productions. For each production, computation rules are given. The defined computations aim at inferring the value of the attribute *type* of the parsed elements and thus generating the DRMs as presented in Sect. 3.1. The attribute *type* behaves both as inherited and synthesized, thus it is initialized during a top-down phase, and it is updated during a bottom-up phase.

Figure 6 shows a fragment of the AG_{ATL} parse tree related to the rule Table of the transformation given in Listing 1. Each node of the tree is decorated with the corresponding computation rules according to the grammar given in Table 1. Such computation rules makes use of the auxiliary functions described below, developed to properly create and update elements in the TRM while traversing the syntax tree:

▷ *createClass(name: String):* it creates and returns a new class named *name*. The function is used in the production *p7* to manage the non-terminal ⟨oclModelElement⟩ like JavaSource!ClassDeclaration and Table!Table of the sample ATL transformation. The actual DRMs including the newly created classes are decided later in the process while traversing the tree bottom-up.

▷ *addClassToSourceDRM(c: Class)* and *addClassToTargetDRM(c: Class)*: they add a new class of type *c* in the source and target DRM, respectively. They are used in the production *p3* to manage the non-terminal ⟨InPatternElement⟩ like the element s:JavaSource!ClassDeclaration, and in *p5* for managing ⟨OutPatternElement⟩

[3] http://github.com/totem-mde/totem.

like Table!Table. In both cases, the new classes previously generated by the function *createClass* (e.g., *ClassDeclaration* and *Table*) are added in the corresponding DRMs. The value of the mandatoryAllowed attribute for the created classes is specified as true (false) for those added in the source (target) DRMs. The value of the isAbstract attribute is specified as Any for the classes added in the source DRMs, and false otherwise. The antiancs relation is set between any non-anonymous classes of the source domain, which were created by the production *p2* applied on ⟨inPattern⟩ elements consisting of only one ⟨inPatternElement⟩.

▷ *isNavigationOrAttributeCallExp(o: OclExpression):* since the non-terminal element ⟨oclExpression⟩ can be matched in several cases (see production *p8*), this function checks if the input OCL expression is a ⟨navigationOrAttributeCallExp⟩. Examples of ⟨navigationOrAttributeCallExp⟩ are i.method.name and s.methods, which use the infix "." operator to call properties and to navigate across association ends, respectively.

▷ *getReferenceClass(o: OclExpression):* it returns the class of the DRM being generated related to the input OCL expression.

▷ *isOperation(c: String):* it checks if the input string is the name of an OCL operation (e.g., *size*, *sum*, and *exists*) defined over OCL data types. The function is used in the production *p9* to check if the last part of the matched ⟨navigationOrAttributeCallExp⟩ is an operation. If it is not (e.g., name in the expression i.method.name) then a new feature is added in the class, which is being created because of the matched ⟨oclExpression⟩ element (e.g., i.method). If *isOperation* returns true then a new feature is created by means of the *createFeatureByOperation* function (see below).

▷ *createFeature(name: String, c: Class):* it creates a new feature typed by the input class *c*. It is used in the productions *p6* and *p9*. The former is for managing the non-terminal ⟨binding⟩ like rows <- s.methods at line 11 of Listing 1, whereas the latter is for managing the non-terminal ⟨NavigationOrAttributeCallExp⟩ like i.method.name at line 6. In the case at line 11, a new feature named rows is added in the target DRM and its type is inferred from the type of the OCL expression s.methods. In the case at line 6, the production *p9* would match i.method with ⟨oclExpression⟩ and name with ID. Since name is not an operator, a new feature named name will be created in the class referred by the reference i.method. Concerning the cardinality of the created features, when a Number element is created, the corresponding attribute allowMore is true in the minimum cardinality of the source DRM, or in the maximum cardinality of the target DRM. The value of the attribute allowLess is true in the maximum cardinality of the source DRM, or in the minimum cardinality of the target DRM.

▷ *createFeatureByOperation(opName: String, c: Class)*: it creates a new feature and its cardinality is specified according to the operation name given as input. For instance, if the operation is *size*, then it means that the matched expression refers to a collection and, consequently, the max cardinality of the created feature has to be Many.

▷ *createReference(f: Feature):* given a previously created feature as input, it specializes it as a Reference element. It is used in *p9* in case the matched ⟨oclExpression⟩ is a ⟨navigationOrAttributeCallExp⟩. In such a case, the previously created feature has to be specialized to a reference typed with a new AnonymousClass.

▷ *analyseCompatibilityNodes(left: Class, right: Class)*: it is used in the production *p6* for adding elements in the compatibility model being generated. In particular, it does a case analysis between the left and right types of the matched ⟨binding⟩ element by checking compatibility issues like cardinality or problems regarding the types of resolving rules. Then, it creates the corresponding nodes of the feature model accordingly.

5 Implementation and Validation

The approach has been implemented as an Eclipse plugin called TOTEM. This is able extract a TRM from an ATL transformation, and check the conformance of meta-models with respect to the TRM. The tool, a screencast demonstration, and the evaluation results are available at http://github.com/totem-mde/totem.

Next, we evaluate the precision of our TRM extraction process and the flexibility of the conformance relationship. While a formal proof of correctness and completeness of the TRM extraction method would be desirable and will be tackled in future work, ATL is an unformalised language[4]. Therefore, we opted for an empirical evaluation using mutation-based testing. This has the advantage of validating the approach in practice, testing the specificities of real transformations and the particularities of the EMF framework (e.g., opposite references, compositions, etc.).

We use the following ATL transformations in our evaluation: JavaSource2Table (the original version of the running example), PetriNet2PNML (a translation from Petri nets to the PNML document format), KM32EMF (a conversion between OO formalisms), and HSM2FSM (a flattening of hierarchical state machines). The selection criterion was to choose transformations written by a third-party, with no errors or very easily fixable not to introduce a bias. In particular, the first three transformations belong to the ATL Zoo, while the latter is presented in [2].

For each transformation, we extract its TRM (i.e., source and target DRMs and compatibility model) using TOTEM. Then, we generate first-order mutants[5] of the original source and target meta-models (which are also available in the ATL Zoo together with the transformations) by systematically applying the meta-model modifications identified in [3]. Our aim is generating many slightly different variants of the original meta-models, so that some break the transformation, while others do not. Finally, we evaluate whether our algorithm correctly

[4] Some efforts exist to express the *execution* semantics of ATL by compilation into Maude [22]. However, formal typing rules for ATL, including OCL, are not available.

[5] First-order mutants are obtained by applying a mutation operator to the original artifact once.

classifies each mutant as conformant when the transformation can use it safely, and non-conformant otherwise. To determine if the classification is correct, we use the anATLyzer [6] ATL static type checker as an oracle of the typing relationship between the mutated meta-model and the transformation.

For each meta-model mutant, we may obtain one of the following results: our conformance method correctly categorizes the mutant as conformant (*true positive, TP*) or non-conformant (*true negative, TN*), or it incorrectly categorizes the mutant as conformant (*false positive, FP*) or non-conformant (*false negative, FN*). Then, we compute *precision* (an indicator of correctness) as $\frac{\#TP}{\#TP+\#FP}$, and *recall* (an indicator of completeness) as $\frac{\#TP}{\#TP+\#FN}$. The transformations, meta-models and scripts to run the experiment, as well as the evaluation results, are available in the tool website.

Table 2 summarizes the obtained results. There are no false negatives, and thus recall is 100%, signifying that our method classifies correctly non-conforming meta-models as such. There are some false positives though, meaning that some non-conformant meta-models get incorrectly classified as conformant, and the transformation may raise runtime errors if executed with them. Nevertheless, the overall precision is still high. An example of false positive occurs in the expression `i.method.name` of the running example (line 6). In the original meta-model, the `name` attribute is compulsory, but one meta-model mutant relaxes this cardinality to `0..1`. The extracted DRM is not precise enough to put any restriction about the cardinality, however anATLyzer does signal this typing problem, and thus it is reported as a false positive. We have observed that false positives occur due to limitations in the extraction process. To solve these cases, we plan to combine our TRM extraction mechanism with information from anATLyzer's static analysis. However, this is only possible if the source and target meta-models are available.

Table 2. Evaluation results.

	JavaSource2Table	HSM2FSM	PetriNet2PNML	KM32EMF
Mutants	141	316	325	2,515
True positives	70	150	185	1,785
True negatives	64	154	131	695
False positives	7	12	9	35
False negatives	0	0	0	0
Precision	91%	93%	95%	98%
Recall	100%	100%	100%	100%
Conforming	70	150	185	1,785
Non-conforming	62	141	113	684
Incompatible	2	13	18	11

To analyse the effects of the mutations, the second and third last rows of the table show the number of conforming and non-conforming generated meta-model mutants. The numbers are comparable in the first three transformations. Notably, there is a high number of conforming meta-models correctly classified by our algorithm, which means that we can reuse the transformations with many meta-models (more than 2,000) different from the ones used to develop the transformations. The last row of the table shows how many meta-models individually conform to the DRMs but do not satisfy the compatibility model. This shows the usefulness of this model.

We have manually revised the extracted DRMs and some mutants to analyse whether the evaluation demanded a flexible typing from our conformance relationship. We found several interesting cases. For instance, PNML2PetriNet exercised the subsAllowed flag (illustrated in Fig. 4(c) for the running example), since some features of an abstract class Arc were located in all subclasses. Mutations like *pull meta-property*, *push meta-property*, *inline meta-class* and *flatten hierarchy* generate variants which require structural typing to enable conformance. All these cases were correctly handled by our conformance algorithm.

Threats to Validity. A few aspects may threat the internal validity of the experiment. The number of transformations in the evaluation is low, and it will be expanded in future evaluations. However, our first results are promising and encourage us to follow this line of research. In any case, the number of generated meta-model mutants is relatively high (around 3,300). Still, the set of considered meta-model mutation operators might be not complete, potentially preventing exercising all features of our conformance relationship. Finally, we use anATLyzer as oracle to well-typedness. Although anATLyzer has been reported to have high precision and recall [6–8], it is not infallible. However, we have manually revised the dubious cases and have not find any incorrect result.

6 Related Work

The closest related work is by Zschaler [24], who uses logic to express meta-model requirements extracted from toy in-place transformations. Instead, we use a model to represent requirements. The advantage is that we can process those models to, e.g., generate OCL queries, check meta-model conformance, or synthesize meta-models. Our TRM includes abstractions to express common model-to-model transformation requirements (e.g., that a class may have extra mandatory features), and includes a compatibility model, which is novel. Finally, extracting the requirements from ATL transformations is more challenging as we need to deal with OCL expressions, mechanisms like automated binding resolution, and dependencies between meta-models.

In the previous work of some of the authors [4,5], we developed the notion of *concepts* to enable transformation reuse. Concepts are meta-models representing the transformation interface, which need to be bound to meta-models. Instead, in this work we propose using TRMs, which provide further flexibility to express meta-model requirements, like (dis-)allowing extra mandatory features in classes, or declaring features which can be references or attributes. While bindings may encode some of these requirements, the TRMs make explicit constraints that bindings ought to obey. Moreover, concepts lack the notion of compatibility model. On the other hand, bindings permit bridging heterogeneities between concepts and concrete meta-models, while resolving heterogeneities between TRMs and meta-models is future work.

Other approaches to reusability [10,12] are based on establishing a subtyping relationship or binding between the transformation meta-model and other meta-model. However, these approaches still describe the transformation interface in terms of meta-models, while TRMs are more expressive.

Several works analyse the model transformations to obtain their footprint [6,13]. This is the part of the input and output meta-models accessed by the transformation, which is itself a meta-model. While these works rely on the actual transformation meta-models, our analysis is done without them. Moreover, we produce a TRM, which is more general as it allows using the transformation with different meta-models.

Transformation *intents* [18] describe semantical properties that ensure a correct transformation reuse according to the designer expectations. In our case, we aim at ensuring syntactical correctness, but it would be interesting to incorporate such intents into our framework in the future.

Finally, Famelis et al. [11] propose a meta-model independent approach to express uncertainty in models, which is applicable to meta-models. We use a DRM meta-model as it allows expressing domain-specific aspects in a more natural way, like the possibility of features to be both attributes and references, the semantics of flags mandatoryAllowed and subsAllowed, or defining transformation-specific compatibility constraints.

7 Conclusions and Future Work

In this paper, we have presented a new approach, based on TRMs, for model transformation reusability. TRMs are automatically extracted from model transformations, and contain a compatibility model constraining the possible options in the source and target meta-models. We have implemented prototype tool support and presented an experiment, based on meta-model mutation, showing promising results.

In the future, we would like to add the notion of *binding* into our conformance relationship in order to improve reusability. Such bindings may resolve heterogeneities (e.g., class renamings) between the TRMs and the meta-models, inducing a transformation adaptation like in [4]. We plan to explore heuristics for automatic meta-model generation from TRMs. As our checks are syntactical, we would like to incorporate the notion of *transformation intent*. Finally, we are working on building a graphical modelling tool to visualize and bind TRMs, and on formal proofs of correctness of the TRM extraction procedure.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitivity, grants TIN2014-52129-R and TIN2015-73968-JIN (AEI/FEDER, UE), and the Madrid Region (S2013/ICE-3006).

References

1. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated chaining of model transformations with incompatible metamodels. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 602–618. Springer, Cham (2014). doi:10.1007/978-3-319-11653-2_37

2. Cheng, Z., Monahan, R., Power, J.F.: Formalised EMFTVM bytecode language for sound verification of model transformations. Softw. Syst. Model. 1–29 (2016, in press)
3. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, EDOC 2008, pp. 222–231. IEEE Computer Society (2008)
4. Cuadrado, J.S., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Trans. Softw. Eng. **40**(11), 1042–1060 (2014)
5. Cuadrado, J.S., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 186–201. Springer, Cham (2014). doi:10.1007/978-3-319-08789-4_14
6. Cuadrado, J.S., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: 25th IEEE International Symposium on Software Reliability Engineering, ISSRE, pp. 34–44. IEEE Computer Society (2014)
7. Cuadrado, J.S., Guerra, E., de Lara, J.: Quick fixing ATL transformations with speculative analysis. Softw. Syst. Model. 1–32 (2016, in press). Springer
8. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Trans. Softw. Eng. 1–32 (2017, in press)
9. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Softw. Syst. Model. **12**(3), 453–474 (2011)
10. de Lara, J., Guerra, E., Cuadrado, J.S.: A-posteriori typing for model-driven engineering. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, pp. 156–165. IEEE (2015)
11. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: 34th International Conference on Software Engineering, ICSE 2012, 2–9 June 2012, Zurich, Switzerland, pp. 573–583. IEEE Computer Society (2012)
12. Guy, C., Combemale, B., Derrien, S., Steel, J.R.H., Jézéquel, J.-M.: On model subtyping. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 400–415. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31491-9_30
13. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011, pp. 601–610. ACM (2011)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
15. Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in model-to-model transformation languages: are we there yet? Softw. Syst. Model. **14**(2), 537–572 (2015)
16. Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf
17. Pescador, A., Garmendia, A., Guerra, E., Cuadrado, J.S., de Lara, J.: Pattern-based development of domain-specific modelling languages. In: MODELS, pp. 166–175. IEEE (2015)
18. Salay, R., Zschaler, S., Chechik, M.: Correct reuse of transformations is hard to guarantee. In: Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 107–122. Springer, Cham (2016). doi:10.1007/978-3-319-42064-6_8
19. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. Computer **39**(2), 25–31 (2006)

20. Sen, S., Moha, N., Baudry, B., Jézéquel, J.-M.: Meta-model pruning. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 32–46. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04425-0_4
21. Slonneger, K., Kurtz, B.L.: Formal Syntax and Semantics of Programming Languages, vol. 340. Addison-Wesley, Reading (1995)
22. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. J. Object Technol. **10**(5), 1–29 (2011)
23. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
24. Zschaler, S.: Towards constraint-based model types: a generalised formal foundation for model genericity. In: VAO, pp. 11:11–11:18. ACM, New York (2014)