Automated Reuse of Model Transformations through Typing Requirements Models

JUAN DE LARA, Universidad Autónoma de Madrid (Spain)
ESTHER GUERRA, Universidad Autónoma de Madrid (Spain)
DAVIDE DI RUSCIO, University of L'Aquila (Italy)
JURI DI ROCCO, University of L'Aquila (Italy)
JESUS SANCHEZ CUADRADO, Universidad de Murcia (Spain)
LUDOVICO IOVINO, Gran Sasso Science Institute, L'Aquila (Italy)
ALFONSO PIERANTONIO, University of L'Aquila (Italy)

Model transformations are key elements of model-driven engineering, where they are used to automate the manipulation of models. However, they are typed with respect to concrete source and target meta-models, making their reuse for other (even similar) meta-models challenging.

To improve this situation, we propose capturing the typing requirements for reusing a transformation with other meta-models by the notion of *typing requirements model* (TRM). A TRM describes the prerequisites that a model transformation imposes on the source and target meta-models to obtain a correct typing. The key observation is that any meta-model pair that satisfies the TRM is a valid reuse context for the transformation at hand.

A TRM is made of two *domain requirement models* (DRMs) describing the requirements for the source and target meta-models, and a compatibility model expressing dependencies between them. We define a notion of refinement between DRMs, and see meta-models as a special case of DRM. We provide a catalogue of valid refinements and describe how to automatically extract a TRM from an ATL transformation. The approach is supported by our tool TOTEM. We report on two experiments – based on transformations developed by third parties and meta-model mutation techniques – validating the correctness and completeness of our TRM extraction procedure and confirming the power of TRMs to encode variability and support flexible reuse.

Categories and Subject Descriptors: [**Software and its engineering**]: Model-driven software engineering; Domain-specific languages; System modeling languages; Reusability

Additional Key Words and Phrases: Model transformation, Model transformation reuse, Refinement, ATL, Meta-modelling

ACM Reference Format:
Juan de Lara, Esther Guerra, Davide di Ruscio, Juri di Rocco, Jesús Sánchez Cuadrado, Ludovico Iovino and Alfonso Pierantonio, 2018. Automated Reuse of Model Transformations through Typing Requirements Models. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 57 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Model-driven engineering (MDE) employs models as the main assets during the software development life cycle [Schmidt 2006; Whittle et al. 2014]. Models are typically constructed using domain-specific languages (DSLs) tailored to a particular domain. In MDE, the abstract syntax of a DSL is specified through a meta-model, which describes the structure of the models considered valid.

Author's addresses: J. de Lara, Computer Science Department, Universidad Autónoma de Madrid (Spain); E. Guerra, Computer Science Department, Universidad Autónoma de Madrid (Spain); D. di Ruscio, Computer Science Department, University of L'Aquila (Italy); J. di Rocco, Computer Science Department, University of L'Aquila (Italy); J. Sánchez Cuadrado, Computer Science Department, Universidad de Murcia (Spain); L. Iovino, Gran Sasso Science Institute, L'Aquila (Italy); A. Pierantonio, Computer Science Department, University of L'Aquila (Italy).

DOI: http://dx.doi.org/10.1145/0000000.0000000

Therefore, meta-models proliferate in MDE as a means of formalising application domains [van Deursen et al. 2000]. Sometimes, meta-models are variants of known languages like state-machines or workflow languages [Pescador et al. 2015], for which services, like transformations, already exist.

Model transformations are key to MDE because they leverage automation in model manipulation. However, MDE frequently relies on domain-specific modelling, which fosters the creation of meta-models for the domain at hand, even on a per-project basis. Unfortunately, in general, it is not possible to reuse a transformation developed for a meta-model with a different one, even if they are similar. For example, the ATL zoo (an open repository of transformations) contains different transformations to calculate metrics for UML class diagrams and for KM3 models[1] [Sánchez Cuadrado et al. 2014a]. Both transformations provide essentially the same functionality but were created separately because they apply to different meta-models, even though UML class diagrams and KM3 share many concepts. With proper reuse techniques, just one transformation would be applicable to both languages, and to other similar ones. Developing non-trivial transformations is typically complex, time-consuming and error-prone [Troya et al. 2018; Sánchez Cuadrado et al. 2017]. Hence, mechanisms to enhance transformation reusability are needed [Basciani et al. 2014; Kusel et al. 2015; Bruel et al. 2018] to scale MDE techniques to industrial use.

Model transformations are typed with respect to source and target meta-models. Therefore, reusing transformations is difficult because they are not immediately applicable to other meta-models different from the ones they were initially conceived for. In previous works, we proposed transformation reuse based on *concepts* to express meta-model requirements [de Lara and Guerra 2011]. Concepts can be bound to meta-models, which permits obtaining a new version of the transformation adapted to the bound meta-models. However, concepts have limitations: on the one hand, they have to be manually created, and on the other, they present limited expressiveness to describe variability (e.g., when a required field can be typed according to a set of allowed types). Other approaches extract *effective meta-models* [Sen et al. 2009] by pruning unused typing information from the source/target meta-models according to the syntactical needs of the transformation. Similarly to concepts, requirements based on effective meta-models have limited expressiveness, although their computation can be partly automated.

Contributions. In this paper, we propose a novel automated approach to model transformation reuse that relies on a transformation *typing requirements model* (TRM) to express the syntactical needs of a transformation with respect to its source and target domains. This way, the TRM becomes the reuse interface for a model-to-model transformation. A TRM contains two *domain requirement models* (DRMs) declaring requirements to be satisfied by the source and target meta-models over which the transformation is to be reused. DRMs support variability regarding the type for attributes, the allowed target for references, the inheritance relations between classes, or the existence of classes with certain features but for which the class name is irrelevant. In addition, the TRM captures the dependencies between the two DRMs by means of a feature model [Kang et al. 1990]. In this way, a transformation can be reused with any pair of meta-models that conform to the TRM. We show that this conformance relation can be expressed as a refinement relation and that meta-models can be seen as a special case of DRM. Moreover, we identify refinement operations useful to constrain a TRM while ensuring that the original transformation remains applicable to the meta-models conforming to the refined TRM.

The advantages of TRMs with respect to existing techniques for transformation reuse are the following: *i)* TRMs can be automatically extracted from transformations by means of a static analysis that is agnostic of the declared meta-models (i.e., source and target meta-models are not needed to define TRMs); *ii)* TRMs are more expressive than current techniques to represent requirements (e.g., they allow expressing variability), leading to improved reuse possibilities; *iii)* our method is able to extract the dependencies cross-linking requirements over the source and target meta-models, and express those in terms of feature models.

[1]KM3 is a meta-modelling notation similar to class diagrams [Jouault and Bézivin 2006].

TRMs can be employed with any transformation language, but we illustrate their use for the Atlas Transformation Language (ATL) [Jouault et al. 2008], one of the most widely used transformation languages nowadays. For this purpose, we provide an algorithm to automatically build a TRM from an ATL transformation, so that the transformation can be reused *as-is* with any pair of meta-models conforming to the extracted TRM. The extraction of a TRM from an ATL transformation is supported by our tool TOTEM[2], which also permits graphical visualization and refinement of the extracted TRMs, and conformance checking of meta-models with respect to TRMs.

Finally, we report on an evaluation of our approach with the aim of answering the following research questions:

RQ1: *Is the TRM extraction mechanism from ATL transformations correct and complete?*
RQ2: *To what extent does the variability encoded in the extracted TRMs enable transformation reuse?*

To this end, we have considered seven ATL transformations developed by third-parties. In particular, we have extracted the TRM of these transformations, and have built more than 26,000 variants of their source and target meta-models using model mutation. Then, we have empirically assessed the correctness and completeness of our method 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. The obtained results are very positive, confirming that TRMs are a suitable technique to automate transformation reuse in MDE. Using the extracted TRMs, a second experiment evaluates their power to encode variability and support flexible reuse. Overall, we found that TRM constructs like anonymous classes and untyped features contribute to express variability in comparison to plain meta-models, while the extracted compatibility models were able to capture hundreds of thousands of valid feature combinations (i.e., of reuse contexts).

This is an extended version of our preliminary work in [de Lara et al. 2017]. Specifically, we have extended the formal theory of our approach, providing proofs of refinement correctness supported by Alloy [Jackson 2006][3]. We have recast meta-models as special cases of DRMs and defined a catalogue of formally proven correct refinements. We have improved our tool to support refining TRMs and visualize the dependencies of source and target DRMs as a feature model. We have also extended the evaluation with new transformations, whose result strengthens the confidence in our method, and report on an additional experiment evaluating the power of TRMs to express variability.

Paper organization. Section 2 introduces the ATL model transformation language and a running example, and Section 3 discusses applicability scenarios of our reuse approach. Next, Section 4 defines DRMs, and Section 5 introduces the notion of refinement, proposing a catalogue. Section 6 presents TRMs, and extends the notions of refinement for them. Then, Section 7 explains how to extract TRMs from ATL transformations, and provides hints on how to generalize the procedure to other languages. Section 8 introduces our tool TOTEM. Section 9 validates the approach over a set of transformations developed by third parties. Finally, Section 10 compares with related work, and Section 11 draws some conclusions and lines for future work. The appendix contains the proofs of the main results presented in the paper.

2. BACKGROUND AND RUNNING EXAMPLE

Model transformations are the main mechanism for achieving automation in MDE [Sendall and Kozaczynski 2003], whether it be code generation, traceability, or model management. Transformations are typically used to generate new models starting from existing ones, and they are often used in toolchains that automate complex tasks within modelling environments [Di Ruscio et al. 2012;

[2]Tool and source code available at http://github.com/MDEGroup/totem.
[3]Alloy specifications available at http://miso.es/trms/

Basciani et al. 2014]. If we restrict our attention to unidirectional transformations, we cannot neglect one of the most prominent transformation languages represented by ATL [Jouault et al. 2008].

ATL provides a mixture of declarative and imperative constructs to develop model-to-model transformations, being the source model *read-only*, and the target model *write-only*. The listing to the left of Figure 1 shows our running example, based on a transformation from the ATL Zoo[4] (a public repository of model transformations written in ATL). Given a piece of Java code represented as a model, the transformation creates a table with the number of times each Java method is called from any declared method. The transformation is defined by a module specification consisting of a header section (lines 1–2), a helper (lines 4–8), and transformation rules (lines 10–27). The header specifies the source and target models of the transformation together with their corresponding meta-models. This way, the JavaSource2Table module is a transformation that generates a target model conforming to the Table meta-model from a source JavaSource model (see line 2). For convenience, Figure 1 shows both involved meta-models to the right.

```
 1 module JavaSource2Table;
 2 create OUT : Table from IN : JavaSource;
 3
 4 helper context JavaSource!MethodDefiniti
 5   computeContent(col : JavaSource!Method
 6   self.invocations->select(i |
 7     i.method.name = col.name
 8     and i.method.class.name = col.class.
 9
10 rule Table {
11   from s : JavaSource!ClassDeclaration
12   to t : Table!Table ( rows <- s.methods
13 }
14
15 rule MethodDefinition {
16   from m : JavaSource!MethodDefinition
17   to row : Table!Row (
18      cells <- Sequence{JavaSource!Method
19               ->collect(md | thisModule.
20   )
21 }
22
23 lazy rule DataCells {
24   from md: JavaSource!MethodDefinition,
25         m: JavaSource!MethodDefinition
26   to cell: Table!Cell ( content <- m.com
27 }
```

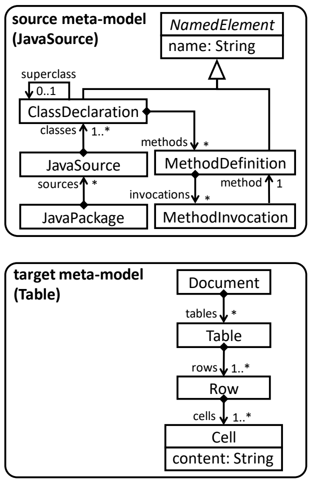


Fig. 1: Fragment of a sample ATL transformation (left) and its source and target meta-models (right).

Helpers and rules are the main ATL constructs to specify the transformation behaviour. The source pattern of rules (e.g., line 11) 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 [Object Management Group 2005] rule guard. Each rule also specifies a target pattern (e.g., line 12) indicating the target objects to be created, and a set of bindings to initialize their features (attributes and references). In case of references, a binding may assign them objects of the source model. In that case, a binding resolution mechanism takes place, which assigns to the reference the target objects created by some rule from the specified source objects. For example, the binding rows ← s.methods in line 12 initializes the reference rows of the Table object with the target objects created by rule MethodDefinition from the source elements (of type MethodDefinition) referred by s.methods.

The rule MethodDefinition (lines 15–21) creates a target Row from each source MethodDefinition. The binding in this rule assigns to reference cells a sequence of objects created by an OCL expression. This expression selects all source MethodDefinition objects and applies on them the lazy rule DataCells.

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

Differently from *matched* rules like Table and MethodDefinition, *lazy* rules are executed only when explicitly called, and use the received parameters. The DataCells lazy rule (lines 23–27) takes two MethodDefinition objects as input and generates a target Cell containing a number calculated by the helper computeContent. Helpers are auxiliary operations that permit defining complex model queries using OCL. In particular, the computeContent helper (lines 4–8) returns a string with the number of occurrences of the received MethodDefinition object.

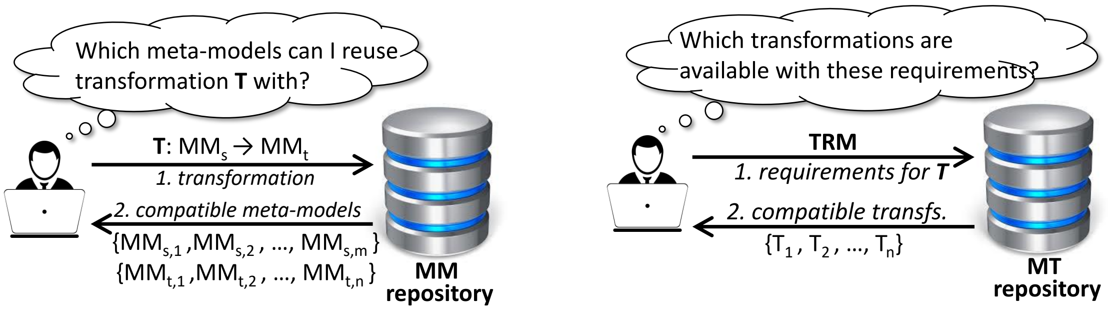
The transformation so defined can be applied to instances of the specified meta-models. In the following section, we present scenarios that require reusing a transformation for meta-models different from the specified ones and outline our proposal to tackle these scenarios.

3. MOTIVATING SCENARIOS

Model transformations can be complex and, therefore, they require from development techniques akin to traditional software artefacts. Consequently, for increasing both the development productivity and the model transformation quality, advanced reuse mechanisms are necessary [Chechik et al. 2016]. Even though over the last years there have been several proposals to deal with the problem of model transformation reuse [Bruel et al. 2018], it is a problem still far from being satisfactorily solved [Kusel et al. 2015]. In this paper, we describe mechanisms to cope with the following two reuse scenarios (cf. Figure 2):

— *Transformation reuse via meta-model querying and synthesis*: Instead of building a new transformation from scratch, a more cost-effective solution is to reuse an existing transformation from a public repository like GitHub or Bitbucket. Unfortunately, such repositories are not specialized in managing the dependencies among transformations and meta-models. Thus, it might happen that the developer wants to reuse an available transformation $T : MM_s \rightarrow MM_t$ but it is not clear for which other meta-models T is reusable. It might even be the case that the definitions of MM_s and MM_t are missing (e.g., a process which is mining a software repository). To be able to reuse the transformation, it is important to extract the requirements expected from potential meta-models, in order to understand whether it would work with other alternative meta-models. Hence, in this scenario illustrated in Figure 2(a), the developer retrieves from a repository a set of meta-model pairs that can be used instead of MM_s and MM_t to execute the transformation T. Alternatively, such a meta-model pair could be synthesized on purpose.

— *Transformation reuse via transformation querying*: In this scenario, illustrated by Figure 2(b), a developer looks in a repository for transformations that fulfil a set of given requirements. The


Which meta-models can I reuse transformation **T** with?
T: MM$_s$ → MM$_t$
1. transformation
2. compatible meta-models
{MM$_{s,1}$, MM$_{s,2}$, ..., MM$_{s,m}$}
{MM$_{t,1}$, MM$_{t,2}$, ..., MM$_{t,n}$}
MM repository
Which transformations are available with these requirements?
TRM
1. requirements for **T**
2. compatible transfs.
{T$_1$, T$_2$, ..., T$_n$}
MT repository

(b) Transformation reuse via transformation querying

Fig. 2: Motivating scenarios for transformation reuse.

In the remaining of this section, we outline our approach to tackle these two scenarios.

3.1. Transformation reuse via meta-model querying and synthesis

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), or we might want to reuse a transformation with meta-models different from the ones the transformation was designed for [Bruel et al. 2018]. To tackle both cases, we propose the approach depicted in Figure 3. Given an existing transformation T, we extract its so-called *typing requirements model* (TRM, see label 1) which describes the structural requirements that meta-models have to fulfil to be used as source or target of the transformation. A TRM consists of three parts: the requirements for the source and target meta-models – named source and target *domain requirement models* (DRMs) – and a compatibility model specifying dependencies between them. The transformation T can be reused with any meta-model satisfying the TRM, and not just with the ones used for its definition. Please note that in general, a transformation may involve more than two models. For simplicity, we restrict to the case of transformations between two meta-models, but our approach could be applicable to transformations involving more meta-models as well.

In this scenario, the TRM can be used to query a meta-model repository to find meta-model pairs conforming to the TRM (see 2a). In this paper, we provide a conformance test between meta-model pairs and TRMs[5]. In this way, any meta-model pair $\langle MM_s, MM_t \rangle$ conforming to the TRM can be used as source/target of the transformation. Moreover, 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). This last po ants to execute the transformat

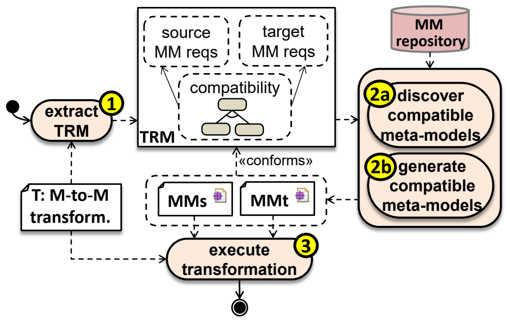


Fig. 3: Proposed approach to support the scenario *transformation reuse via meta-model querying and synthesis*.

As an example, from rule Table in lines 10–13 of Listing 1, our approach would extract a TRM consisting of two DRMs and one compatibility model specifying the following requirements, among others detailed later in the paper:

— *Source DRM*: The meta-models that can be source of the transformation must contain a class named ClassDeclaration due to the source input pattern of rule Table. As the rule defines the binding rows $\leftarrow$ s.methods, ClassDeclaration must have a feature named methods. The transformation does not provide enough information to statically identify if this feature is a reference or an attribute. Thus, the DRM represents this variability point and permits both possibilities;
— *Target DRM:* The meta-models that can be target of the transformation must contain a class Table with a feature rows, requirements which are derived from the output pattern and binding in rule Table. Again, it is not possible to determine whether rows is an attribute or a reference. In any case, it has to be compatible with the type of feature s.methods, because the binding assigns one to the other. This dependency justifies the need for compatibility models as discussed below;

[5]Actually, we provide a notion of refinement between TRMs, and consider a meta-model pair as a special case of TRM.

— *Compatibility model:* This model declares the dependencies between the source and target DRMs. In this example, it establishes a dependency between the types of features rows in class Table and methods in class ClassDeclaration. If the developer resolves the variability related to feature methods, e.g., by indicating that it is a string attribute, then rows needs to have the same type.

Sections 4–6 will provide the definitions of DRM, compatibility model and TRM, while Section 7 will detail the algorithm to extract a TRM from an ATL transformation.

3.2. Transformation reuse via transformation querying

While the previous scenario implies the automated extraction of TRMs from existing transformations, developers can also define TRMs by hand with the aim of querying existing repositories of reusable transformations (see 1 in Figure 4). In this case, the TRM would be the input to a discovery mechanism that selects all transformations compatible with the TRM (see 2). Intuitively, a model transformation $T : MM_s \rightarrow MM_t$ is compatible with a TRM if the meta-model pair $\langle MM_s, MM_t \rangle$ conforms to the DRMs in the TRM and satisfies the conditions in its compatibility model (see 2a). The result of this exploration is a set of transformations compatible with the TRM. If this set is too large to be inspected by the developer, the TRM can be refined by adding more requirements and thus reducing the number of candidate transformations (see 3). This refinement operation induces a hierarchy of TRMs, as we will discuss in Sections 5 and 6.

The described scenario starts with a general TRM that is used to find transformations T_i whose input and output meta-models are more specific than the TRM (variant 2a in Figure 4). Alternatively, we may wish to find transformations T_i whose TRM is more general than a given input meta-model pair. In such a case, the input TRM would consist of a pair of meta-models with no variability and no compatibility model (variant 2b in Figure 4), and the result would consist of all transformations that can be reused with the given meta-model pair. As a meta-model pair is a special case of TRM, the two variants of this scenario amount to either extracting the most specific reuse interface of a transformation (the meta-model pair) and then checking the extracted TRM against a more general input TRM; or .M) and then checking the e

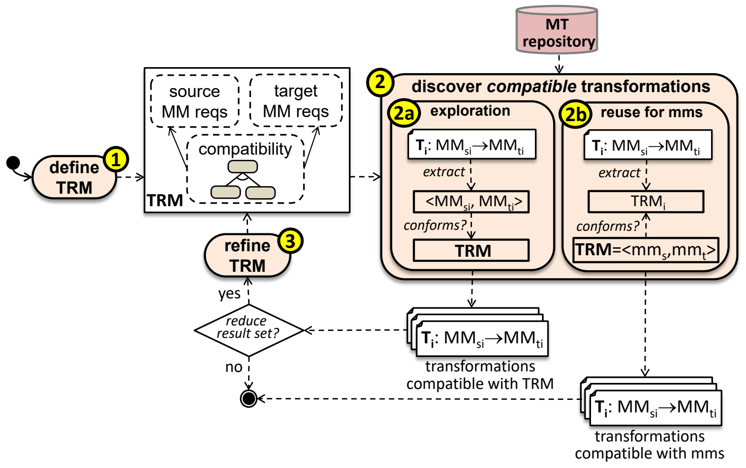


Fig. 4: Proposed approach to support the scenario *transformation reuse via transformation querying*.

In both scenarios, we assume a syntactical approach to reuse, that is, we look for compatible meta-models or transformations which together are syntactically correct. However, in some cases, it is useful to express expectations on the semantics of the source and target languages – e.g., using transformation intents [Salay et al. 2016] – to discard semantically meaningless reuse attempts. We leave the investigation of the combination of TRMs and intents for future work.

4. DOMAIN REQUIREMENTS MODELS

Next, we introduce the notion of *domain requirement model* (DRM) as an abstraction mechanism to decouple model operations from concrete meta-models. A DRM describes the typing requirements that a given model operation (a transformation) needs from the meta-model it is typed on.

We start by providing an overview of the rationale and use of DRMs in Section 4.1. This overview will be used to structure the following sections. Then, Section 4.2 defines DRMs.

4.1. Using DRMs: Overview

The left of Figure 5 depicts a schema of the usage of DRMs for transformation reuse. Given a model operation q defined over a meta-model MM, we derive a DRM that: (i) slices the portion of MM required by q and (ii) abstracts the elements of the slice to the minimum typing requirements from the point of view of q. A (possibly infinite) set of meta-models, including MM, are said to *refine* this DRM. While q was originally defined over MM, it can now be seen as defined over the DRM, and applicable to every meta-model refining the DRM. We will show how to extract a DRM given an ATL model transformation in Section 7.

$$\begin{tikzcd}
 & DRM \arrow[d, "refinedBy"] \arrow[d, phantom, "=", shift right=4ex] & \\
q \arrow[ru, dotted, bend left, "defined"] \arrow[r, "defined"] & MM \arrow[r, phantom, "(1)"] & {} \\
 & {} \arrow[r, phantom, "="] & M \arrow[lu, bend left, "type"'] \arrow[luu, dotted, bend right, "type*"']
\end{tikzcd}
\qquad
\begin{tikzcd}
q \arrow[r, "defined"] \arrow[rd, dotted, bend right, "defined"'] & DRM \arrow[d, "refinedBy"] \arrow[d, phantom, "=", shift right=6ex] \\
 & MM'
\end{tikzcd}$$

Fig. 5: DRMs as abstractions for model operations (left). Correctness property for DRMs (right).

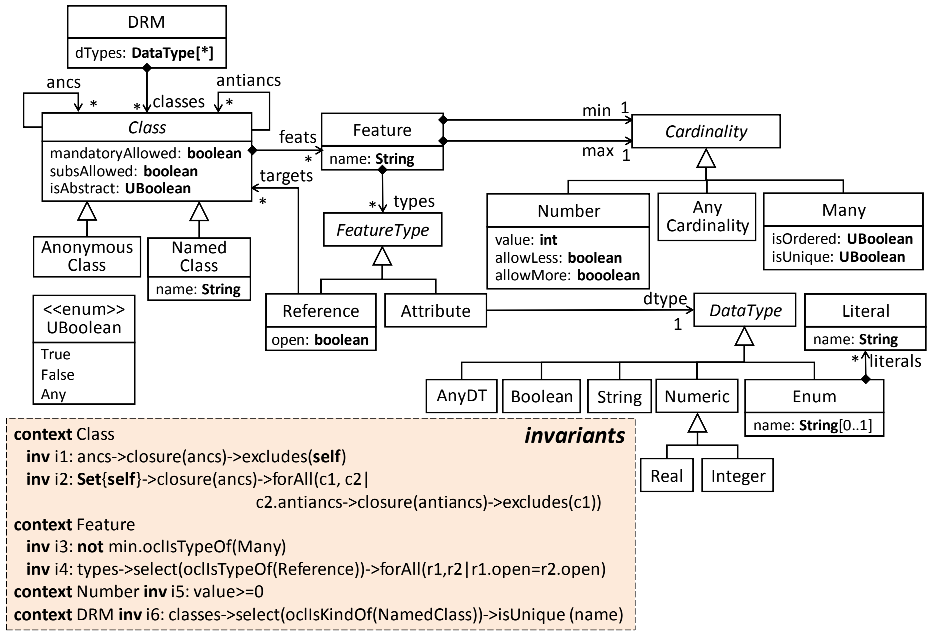
We say that operation q can be used as if defined over the DRM (denoted q_{DRM}), as for every model M typed over MM, we can find a typing $type^*$ from M to the DRM such that $q_{DRM}(M) = q(M)$. This is so as any instance model M of a meta-model MM that conforms to a DRM can be typed over the DRM. The existence of this typing $type^*$, which makes the triangle (1) in the left of Figure 5 commute, provides the semantics of our notion of refinement between a meta-model and a DRM.

The right part of Figure 5 depicts the correctness property that is expected from the DRM of an operation q, namely, q is well-defined over any meta-model MM' refining the DRM.

In the remainder of this section, we introduce the notion of DRM. Refinement between DRMs (relation $refinedBy$ in Figure 5) is presented in Section 5.1. Section 5.2 defines extra conditions enabling the composition of atomic refinements and provides a catalogue of refinement operations. This catalogue is useful to developers aiming at transformation reuse via querying (Figure 4) to increasingly refine an initial DRM. Then, in Section 5.3, we make the observation that meta-models can be seen as a special case of DRM. Finally, Section 5.4 gives a semantics of refinement in terms of instantiation, and proves the existence of the $type^*$ relation in the diagram of Figure 5.

4.2. Defining DRMs

We use the meta-model in Figure 6 to represent structural requirements for single meta-models. Its instances, called DRMs, resemble meta-models but some decisions can be left open if they are irrelevant for the transformation at hand, like class names, attribute types, the target of references, or the cardinality of features. This way, a potentially infinite set of meta-models may refine a DRM.



```
context Class
  inv i1: ancs->closure(ancs)->excludes(self)
  inv i2: Set{self}->closure(ancs)->forAll(c1, c2|
                          c2.antiancs->closure(antiancs)->excludes(c1))
context Feature
  inv i3: not min.oclIsTypeOf(Many)
  inv i4: types->select(oclIsTypeOf(Reference))->forAll(r1,r2|r1.open=r2.open)
context Number inv i5: value>=0
context DRM inv i6: classes->select(oclIsKindOf(NamedClass))->isUnique (name)
```

Fig. 6: Meta-model to define domain requirements models.

We consider two kinds of classes: named and anonymous. While the former have a name (which is assumed unique, cf. constraint i6), the name of the latter is irrelevant, meaning that the class is allowed to have any name. Classes have a flag isAbstract with a three-valued enum type UBoolean, which allows stating whether a class is abstract, concrete or any of both. A class defines a collection of features. The flag mandatoryAllowed permits a class to have more mandatory features than those indicated in collection feats, while there is no constraint on the number of extra non-mandatory features. 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 antiancs. More precisely, if B $\in$ A.antiancs, then we reject meta-models in which B is an ancestor of A, or both share a common (direct or indirect) subclass. The meta-model invariants on Class ensure that ancs is acyclic; and there is no class with two ancestor classes one of which is antiancestor of the other. The latter constraint also entails that ancs and antiancs are disjoint, and antiancs irreflexive.

Features have minimum and maximum cardinality, which can be either a number (class Number), an indication of many cardinality (Many), or we might allow any cardinality (AnyCardinality). If the maximum is many, features can also indicate whether they are ordered or unique using UBoolean values. For the case of a number, it can be specified whether the cardinality is allowed to be lower (allowLess) or higher (allowMore) than this number. Features always have a name, and optionally, they may have a set of compatible types, which can be References, Attributes or both.

Attributes can specify their data type, or it can be left open using the AnyDT class.

References can indicate the admissible compatible target types (collection targets), some of which can be anonymous classes. The flag open permits defining whether a reference has open or closed semantics, being the difference that open references can have a larger set of targets than specified in collection targets, while closed ones can have less targets. This flag is useful to distinguish when a reference is being read (closed) or written (open) by the transformation. We require each compatible Reference of a given Feature to have the same semantics (invariant i4 in Figure 6). Section 5.4 will provide more details on the semantics of open/closed references.

We have adopted a specific concrete syntax to denote the different characteristics of DRMs (see, e.g., Figure 7). Specifically, the upper-right corner of a class shows whether *i)* it can be either abstract or concrete (AC), only abstract (A), or only concrete (C); *ii)* it can defer the conformance checking to its subclasses (encircled inheritance-like triangle); and *iii)* it forbids extra mandatory features (crossed-out circle ⊘). The antiancestor relation is shown as a crossed-out red inheritance relation. The arrowhead of references is filled and closed for closed references, and open for open references.

Example. Figure 7 shows three examples of DRM, instances of the meta-model in Figure 6, using the described concrete syntax. DRM (a) has been extracted from the source domain of the transformation in Listing 1 (Section 7 will describe the extraction procedure). The DRM requires two classes named ClassDeclaration and MethodDefinition, which cannot inherit from each other. 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 can be any. In its turn, ClassDeclaration requires a feature methods which can be an attribute or a reference (we use a "?" prefix to denote that both possibilities are allowed). The DRM also demands four anonymous classes for which only certain features are required. These classes could be matched by the same or different

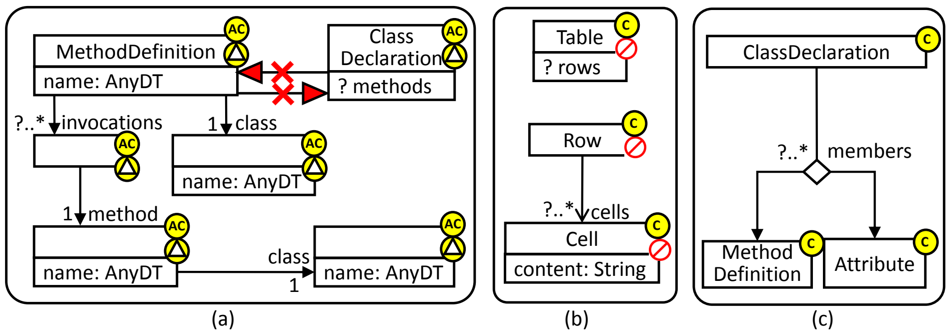


Fig. 7: (a) Source DRM of Listing 1. (b) Target DRM of Listing 1. (c) Multiple compatible reference targets.

DRM (b) has been extracted from the target domain of Listing 1. It requires three concrete named classes. Class Table requires a feature rows, which can be an attribute or a reference. As Section 6 will show, 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. The rationale is that, as the transformation creates objects of these classes, should the classes had more mandatory attributes, the transformation would not initialize them, producing ill-formed models. Reference Row.cells is open because the target domain of an ATL transformation is write-only.

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.

In subsequent sections, we use a semi-formal treatment of DRMs. Hence, we represent them using an algebraic structure derived from the meta-model in Figure 6. In the following, we use Str to denote the set of all possible strings, $Bool = \{true, false\}$ for the possible boolean values, $UBoolean = Bool \cup \{any\}$ for the three-valued boolean values, $Numeric = \{Real, Integer\}$ for the supported numeric types, and $DataType = \{AnyDT, String, Boolean, Numeric, Real, Integer\}$ for the supported data types.

Definition 4.1 (*DRM*). A domain requirements model (DRM) is a tuple

$$\begin{aligned} RM = \langle & NC, AC, F, R, A, Number, Many, \\ & ancs, antiancs, feats, targets, types, \\ & name, mandatoryAllowed, subsAllowed, isAbstract, \\ & min, max, value, allowLess, allowMore, dtype, open\rangle \end{aligned}$$

made of:

— Sets NC of named classes, and AC of anonymous classes. We define the set $C = NC \cup AC$.
— Sets F of features, R of reference types, and A of attribute types. We define the set $FT = R \cup A$.
— Sets $Number$ of numeric cardinalities, and $Many$ of multiple cardinalities. We define the set $Card = Number \cup Many \cup \{AnyCardinality\}$.
— Relations $ancs \subseteq C \times C$; $antiancs \subseteq C \times C$; $feats \subseteq C \times F$; $targets \subseteq R \times C$; and $types \subseteq F \times FT$. Relation $ancs$ must be acyclic, and relations $feats$ and $types$ are restricted to be injective (left definite).
— Functions $name\colon NC \cup F \to Str$; $mandatoryAllowed, subsAllowed\colon C \to Bool$; $isAbstract\colon C \to UBoolean$; $min, max\colon F \to Card$; $value\colon Number \to \mathbb{N}_0$; $allowLess, allowMore\colon Number \to Bool$; $dtype\colon A \to DataType$; and $open\colon R \to Bool$. We demand $name|_{NC}$ to be injective.

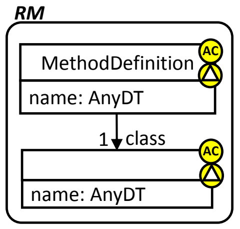
such that the following conditions hold:

(1) $\forall c \in C, \forall c_1, c_2 \in c.ancs^* \bullet c_1 \notin c_2.antiancs^+$
(2) $\forall f \in F \bullet min(f) \notin Many$
(3) $\forall f \in F, \forall r_1, r_2 \in f.types \bullet \{r_1, r_2\} \subseteq R \implies r_1.open = r_2.open$

The previous definition omits enums and the ordered and uniqueness features of Many for simplicity. Analogously to the meta-model of Figure 6, the definition requires ancs to be acyclic (as invariant i1 in the meta-model) while conditions (1-3) in the definition are equivalent to invariants i2, i3, and i4 in the meta-model. Condition 1 implies that antiancs is irreflexive and does not overlap with ancs. The injectivity of $name|_{NC}$ (the restriction of function $name$ to NC) results in unique names for named classes (as required by constraint i6 in the meta-model).

Given a tuple RM, we will use RM_C to refer to set C, and similarly for the other sets. In the following, we use an "object-oriented" notation for functions and relations. For example, given an element $c \in NC$, we use $c.name$ instead of $name(c)$. Moreover, given a relation or a function, we use $^+$ to denote its transitive closure, and * for its reflexive-transitive closure. For example, $c.ancs^*$ denotes the set of all direct and indirect ancestors of c, and c itself. The encoding of all definitions and theorems in the paper using Alloy is available at http://miso.es/trms.

Example. Figure 8 shows the encoding of a DRM RM using Definition 4.1. The DRM has one



RM = ⟨ NC = {MD}, AC = {AC1}, F = {f1, f2, f3}, R = {r1}, A = {a1, a2}, Number = {one}, Many = {},
ancs = {}, antiancs = {}, feats = {(MD, f1), (AC1, f2), (MD, f3)}, targets = {(r1, AC1)},
types = {(f1, a1), (f2, a2), (f3, r1)},
name = {(MD, "MethodDefinition"), (f1, "name"), (f2, "name"), (f3, "class")},
mandatoryAllowed = subsAllowed = {(MD, true), (AC1, true)},
isAbstract = {(MD, any), (AC1, any)},
min = {(f1, one), (f2, one), (f3, one)}, max = {(f1, one), (f2, one), (f3, one)},
value = {(one, 1)}, allowLess = {(one, false)}, allowMore = {(one, false)},
dtype = {(a1, AnyDT), (a2, AnyDT)}, open = {(r1, false)} ⟩

Fig. 8: Formal encoding of an excerpt of the DRM in Figure 7(a).

5. REFINEMENT, COMPOSITION AND SEMANTICS OF DOMAIN REQUIREMENT MODELS

Once we have defined DRMs, next we introduce a refinement relation between them (Section 5.1), and then strengthen this relation to make it composable, presenting a catalogue of useful refinements (Section 5.2). Then, we show that meta-models can be seen as a special kind of DRM (Section 5.3), and finish presenting a semantics of refinement in terms of typing (Section 5.4).

5.1. Refinement of domain requirement models

Next, we introduce a notion of refinement between DRMs. This is useful to understand the modifications that can be done to a DRM to obtain a more refined DRM, as well as to be able to narrow the (possibly infinite) set of meta-model pairs that can be used with a given transformation, as the scenario described in Section 3.2 requires (cf. Figure 4).

Refinement between DRMs is a special kind of mapping $ref\colon RM \to RM'$ from a DRM RM to a more refined one RM'. The more general notion of mapping between two DRMs maps classes to classes, features to features, references to references, and attributes to attributes. We consider two kinds of mappings for classes, m_C and m_S, which will impose different conditions in refinement mappings. Classes must be mapped using either one of the mappings, and so $m_C \cup m_S$ (the union of both mappings) is a total function. Features should be mapped only if they belong to classes mapped through m_C. The mappings for references (m_R) and attributes (m_A) are also partial. This is necessary as a feature in RM may declare several feature types, and we do not demand all of them to be mapped, but finding just a compatible one in RM' is enough.

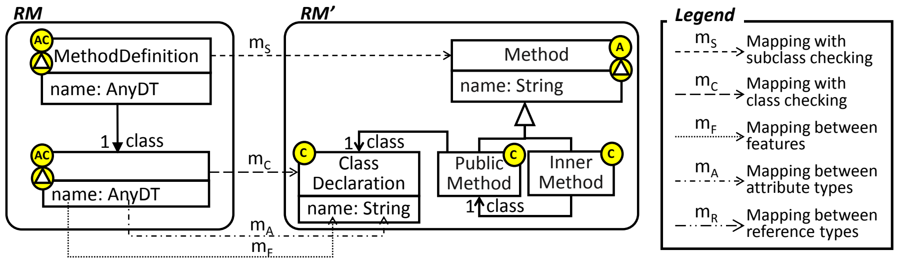
Definition 5.1 (*DRM mapping*). A ***DRM mapping*** $m\colon RM \to RM'$ from a DRM RM to a DRM RM' is a tuple $m = \langle m_C, m_S, m_F, m_R, m_A\rangle$ made of:

— Two partial functions $m_C, m_S\colon RM_C \nrightarrow RM'_C$ mapping classes to classes, such that:
  (1) their union $m_{CS} = m_C \cup m_S$ is a total function
  (2) $\forall C \in RM_C \bullet\ m_S(C)\ is\ defined \implies C.subsAllowed = true \wedge \forall C_s \in C.subs \bullet m_S(C_s)\ is\ defined$
— A partial function $m_F\colon RM_F \nrightarrow RM'_F$ mapping features to features, where:
  (1) $\forall C \in RM_C \bullet\ m_C(C)\ is\ defined \implies \forall f \in C.feats \bullet m_F(f)\ is\ defined$
— A partial function $m_R\colon RM_R \nrightarrow RM'_R$ mapping reference types to reference types
— A partial function $m_A\colon RM_A \nrightarrow RM'_A$ mapping attribute types to attribute types

As we will see in Definition 5.2, the mapping m_C will be used to check refinement of a class C by the mapped class $m_C(C)$, while m_S will be used to check refinement of a class C by each concrete subclass of $m_S(C)$. The latter checking requires that C permits subclasses ($C.subsAllowed = true$), and that all subclasses of C are mapped via m_S (we use predicate $subs$ to denote the set of direct and indirect subclasses). Classes mapped via m_C can have any value for $subsAllowed$.

Example. Figure 9 shows a DRM mapping example. Function m_S maps class MethodDefinition in RM to Method in RM', while function m_C maps the anonymous class in RM to the named class ClassDeclaration in RM'. Since m_{CS} (the union of m_C and m_S) must be a total function, the domain of m_S and m_C cannot overlap (as we would obtain a relation instead of a function), and all classes in RM must be mapped either by m_S or m_C (otherwise m_{CS} would not be total). The definition of a DRM mapping comprises the mapping of both features (m_F) and feature types (m_A for attribute data types, and m_R for reference targets). In the figure, we depict m_F as links between the names of the mapped features, m_A as links between attribute datatypes, while m_R is empty in this case. m_F only needs to map features of classes that have been mapped through m_C (i.e., feature name from the anonymous class in the example). This is so as classes mapped via m_S will be checked structurally against a set of subclasses (in the figure, MethodDefinition against PublicMethod and InnerMethod), and we want to avoid mapping features multiple times (reference class in RM to both references class in RM').

A DRM refinement is a special kind of mapping, which satisfies a number of conditions, as we show in next definition.



RM
RM'
Legend
MethodDefinition
name: AnyDT
1 class
name: AnyDT
AC
AC
Method
name: String
A
Class Declaration
name: String
1 class
Public Method
Inner Method
1 class
C
C
C
m_S
m_C
m_A
m_F
m_S Mapping with subclass checking
m_C Mapping with class checking
m_F Mapping between features
m_A Mapping between attribute types
m_R Mapping between reference types

Fig. 9: DRM mapping example.

Definition 5.2 (*DRM refinement*). Given two DRMs RM and RM', a DRM mapping $m = \langle m_C, m_S, m_F, m_R, m_A \rangle \colon RM \to RM'$ is a refinement if the following predicate holds:

$$
\begin{aligned}
refinement(m) \triangleq \\
& name\_refinement(m) \ \wedge abstract\_refinement(m) \wedge \\
& ancs\_preservation(m) \ \wedge antiancs\_preservation(m) \wedge \\
& mand\_allowed\_refinement(m) \ \wedge feature\_refinement(m) \wedge \\
& feature\_type\_commut(m) \ \wedge feature\_type\_refinement(m) \wedge \\
& subs\_refinement(m)
\end{aligned}
$$

If $m \colon RM \to RM'$ is a DRM refinement, then we say that RM' refines, or is a refinement of, RM.

The definition relies on nine predicates that we introduce next. In all of them, we assume that a DRM mapping $m = \langle m_C, m_S, m_F, m_R, m_A \rangle \colon RM \to RM'$ between DRMs RM and RM' is given. The predicates are defined over the DRM structure presented in Definition 4.1.

▷ *class name refinement:* this predicate requires that each named class C in RM that is mapped through m_C, is mapped to a class in RM' with the same name. The name of the class an anonymous class is mapped to does not matter.

Predicate: Class name refinement

$$
\begin{aligned}
& name\_refinement(m) \triangleq \\
& \quad \forall C \in RM_{NC} \bullet m_C(C)\ is\ defined \implies m_C(C).name = C.name
\end{aligned}
$$

Example. The mapping m in Figure 9 satisfies $name\_refinement(m)$ because the only class for which m_C is defined is anonymous, and hence, it can be mapped to a class with any name.

▷ *abstractness refinement:* this predicate requires mapping abstract classes to abstract classes, and concrete classes to concrete ones. Classes where *isAbstract* is *any* can be mapped to either abstract or concrete classes.

Predicate: Abstractness refinement

$$
\begin{aligned}
& abstract\_refinement(m) \triangleq \\
& \quad \forall C \in RM_C \bullet C.isAbstract = any \vee m_{CS}(C).isAbstract = C.isAbstract
\end{aligned}
$$

Example. The mapping m in Figure 9 satisfies $abstract\_refinement(m)$ as both classes in RM have *any* as required abstractness.

▷ *ancestor preservation:* this predicate requires preserving the structure of the ancestor relationship: if C_a is an ancestor of the class C, then C_a should be mapped to an ancestor of the class C is mapped to, or be this latter class.

Predicate: Ancestor preservation

$$ancs\_preservation(m) \triangleq$$
$$\forall C, C_a \in RM_C \bullet C_a \in C.ancs^+ \implies m_{CS}(C_a) \in m_{CS}(C).ancs^*$$

Example. The mapping in Figure 10 satisfies the predicate $ancs\_preservation$ because class MethodDefinition is an ancestor of PublicMethod in RM, and this relation is preserved by m_C. The predicate woul

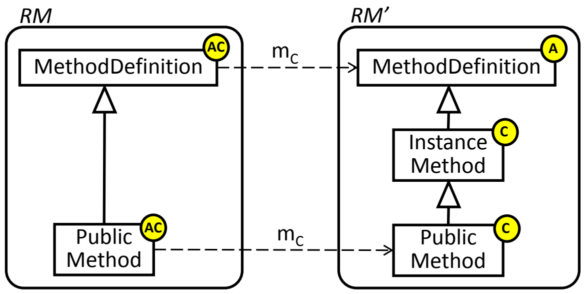


Fig. 10: DRM mapping example illustrating *ancestor preservation*.

▷ *antiancestor preservation:* similar to the previous predicate, this one demands preserving any forbidden ancestor relationship: if C_a cannot be an ancestor of C (i.e., $C_a \in C.antiancs^+$), then C_a should be mapped to an antiancestor of the class mapped to C.

Predicate: Antiancestor preservation

$$antiancs\_preservation(m) \triangleq$$
$$\forall C, C_a \in RM_C \bullet \quad C_a \in C.antiancs^+ \implies m_{CS}(C_a) \in m_{CS}(C).antiancs^+$$

The antiancs relation enables a fine-grain control of the types of objects that transformation rules can match, as a rule from class C_a, where $C_a \in C.antiancs^+$, is applicable to C_a objects but not to C objects. If a refinement maps classes C_a and C to C'_a and C' such that C'_a is an ancestor of C', the rule would become applicable to the instances of both C'_a and C', hence not preserving the rule behaviour. This predicate avoids this problem. As Section 7 will show, DRMs extracted from ATL for source meta-models typically contain antiancs relations between the source classes of the rules.

Example. Figure 11 shows a mapping that violates the predicate $antiancs\_preservation$. It does not preserve the antiancs relation because MethodDefinition in RM' is an ancestor of FieldDeclaration

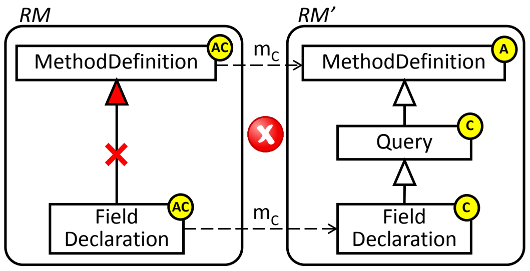


Fig. 11: Mapping violating *antiancestor preservation*.

▷ *mandatory features refinement:* if a class C does not permit more mandatory features than the ones it already defines or inherits (i.e., $C.mandatoryAllowed = false$), then this predicate ensures that the class C is mapped to has the same number of mandatory features as C. A feature is mandatory if its minimum cardinality is bigger than 0. This condition only applies to classes mapped by m_C.

Predicate: Mandatory features refinement

$$mand\_allowed\_refinement(m) \triangleq$$
$$\forall C \in RM_C \;\bullet m_C(C)\; is\ defined \wedge C.mandatoryAllowed = false \implies$$
$$|\{f \mid f \in C.feats^* \wedge isMand(f)\}| = |\{f \mid f \in m_C(C).feats^* \; \wedge isMand(f)\}|$$

with

$$isMand(f : Feature) \triangleq \; f.min \in Number \wedge$$
$$f.min.value > 0 \; \wedge f.min.allowLess = false$$

Typically, for ATL, classes in DRMs of source meta-models have $mandatoryAllowed = true$ as their objects are read-only, while classes in DRMs of target meta-models have $mandatoryAllowed = false$ as their objects are write-only. If we allowed target classes with more mandatory features than those specified in the DRM, the transformation may produce ill-formed models.

Example. Figure 12 shows a DRM mapping that violates the predicate $mand\_allowed\_refinement$. The reason is that class Table in RM has no mandatory features and does not allow any (indicated by the crossed-out circle); however, class Table in RM' has one mandatory feature (name) inherited from NamedElement

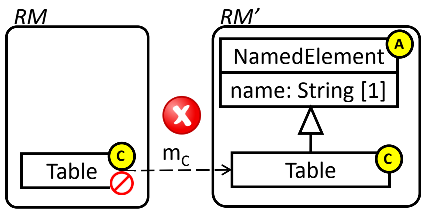


Fig. 12: DRM mapping example illustrating a violation of *mandatory features refinement*.

▷ *feature refinement:* this predicate requires each feature defined in a class C to be mapped to an equally named feature owned or inherited by the class C is mapped to, provided that C is mapped through m_C. The cardinality interval of the mapped feature in RM' should be the same or a refinement of the cardinality interval of C's feature, if the latter cardinality is specified.

Predicate: Feature refinement

$$feature\_refinement(m) \triangleq$$
$$\forall C \in RM_C \;\bullet\; m_C(C)\; is\ defined \implies$$
$$\forall f \in C.feats \;\bullet\; m_F(f) \in m_C(C).feats^* \wedge$$
$$f.name = m_F(f).name \wedge refines_{card}(f, m_F(f))$$

with

$$\begin{aligned}
refines_{card}(f, f' : Feature) &\triangleq refines_{min}(f.min, f'.min) \wedge refines_{max}(f.max, f'.max) \\
refines_{min}(c, c' : Card) &\triangleq c = AnyCardinality \vee refines_{num}(c, c') \\
refines_{max}(c, c' : Card) &\triangleq c = AnyCardinality \vee refines_{num}(c, c') \vee refines_{many}(c, c') \\
refines_{num}(c, c' : Card) &\triangleq c \in Number \wedge \\
&\quad ((c' \in Number \wedge \\
&\quad\quad (c.value = c'.value \vee \\
&\quad\quad (c.value > c'.value \wedge c.allowLess = true) \vee \\
&\quad\quad (c.value < c'.value \wedge c.allowMore = true))) \vee \\
&\quad (c' \in Many \wedge c.allowMore = true)) \\
refines_{many}(c, c' : Card) &\triangleq c \in Many \ \wedge c' \in Many
\end{aligned}$$

Predicate $refines_{card}$ receives two features – f and f' – and yields $true$ if the minimum and maximum cardinality of f' refine those of f. This is so if either the minimum (resp. maximum) cardinality of f is $AnyCardinality$, if both f and f' have the same minimum (resp. maximum) cardinality, or if the minimum (resp. maximum) cardinality of f' is different but it respects the flags of f $allowLess$ (i.e., it can be smaller) and $allowMore$ (i.e., it can be bigger). Checking the maximum cardinality admits a fourth possibility that arises when f has cardinality $Many$.

Example. Given a feature f with cardinality $1..10$ and flags $allowLess = false$ and $allowMore = true$ for both the minimum and maximum cardinality values, predicate $feature\_refinement$ allows its mapping to a feature f' with the same name and cardinalities $1..10$, $1..*$, or $2..20$, but not to a feature with cardinality $1..1$ or $0..*$.

▷ *feature type commutativity:* this is a well-formedness condition for the partial mapping of feature types. It states that if an attribute or reference type is mapped, then it is mapped to one of the types of the mapped feature.

Predicate: Feature type commutativity

$$\begin{aligned}
&feature\_type\_commut(m) \triangleq \\
&\quad \forall f \in RM_F, \forall ft \in f.types \bullet \ m_A(ft)\ is\ defined \implies m_A(ft) \in m_F(f).types \wedge \\
&\quad\quad m_R(ft)\ is\ defined \implies m_R(ft) \in m_F(f).types
\end{aligned}$$

▷ *feature type refinement:* this predicate checks that feature types are refined correctly, taking into account that a feature can declare several potential compatible types, either attribute types or reference types (see meta-model in Figure 6). In case of attribute types (predicate $refines_{Attr}$), $AnyDT$ can be refined by (i.e., mapped to) any type, $Numeric$ can be refined by $Real$ and $Integer$, and otherwise the data type must be preserved. In the case of reference types (predicate $refines_{Ref}$), there are two semantics depending on whether the reference types are *open* or *closed*. Let r be a reference type in RM that is mapped to a reference type r' in RM'. Then, if r is open, all target classes of r must be mapped into targets of r'. This means that r can be refined to include more compatible classes. If r is closed, then all target classes of r' must receive a mapping from some target of r. This means that r can be refined to a more restricted set of admissible target classes.

Predicate: Feature type refinement

$$
\begin{aligned}
&feature\_type\_refinement(m) \triangleq \\
&\qquad \forall f \in RM_F \bullet m_F(f)\ is\ defined \implies ((f.types = \emptyset) \vee \\
&\qquad\quad (m_F(f).types \neq \emptyset\ \wedge \forall ft' \in m_F(f).types\ \exists ft \in f.types \bullet \\
&\qquad\qquad (m_A(ft) = ft' \wedge refines_{Attr}(ft, ft')) \vee \\
&\qquad\qquad (m_R(ft) = ft' \wedge refines_{Ref}(ft, ft'))))
\end{aligned}
$$

with

$$
\begin{aligned}
refines_{Attr}(a, a' : FT) \triangleq\ & a \in A \wedge a'\ is\ defined \wedge \\
& (a.dtype = AnyDT \vee a.dtype = a'.dtype \\
& \vee (a.dtype = Numeric \wedge a'.dtype \in Numeric)) \\
refines_{Ref}(r, r' : FT) \triangleq\ & r \in R \wedge r'\ is\ defined \wedge \\
& (r.open = true \implies \\
& \quad \forall c \in r.targets \bullet m_{CS}(c) \in r'.targets) \wedge \\
& (r.open = false \implies \\
& \quad \forall c' \in r'.targets \bullet \exists c \in r.targets \bullet m_{CS}(c) = c')
\end{aligned}
$$

Example. Figure 13(a) illustrates the refinement of closed references. We depict m_F as links between the names of the mapped features, and m_R (i.e., the mapping of reference types) as links between the reference arrow ends. The reference members in RM declares two reference types, though only one of them is mapped (the one pointing to Attribute). This is possible because mappings m_R and m_A can be partial. Given the reference members in RM, the predicate $feature\_type\_refinement$ iterates on all targets of the reference members in RM' (only class Attribute in this case) and checks that they are mapped from some target of the reference members in RM. In this case, Attribute in RM' is mapped from Attribute in RM, therefore, this is a valid refinement. This predicate is useful to chara[illegible]ce of RM[illegible]

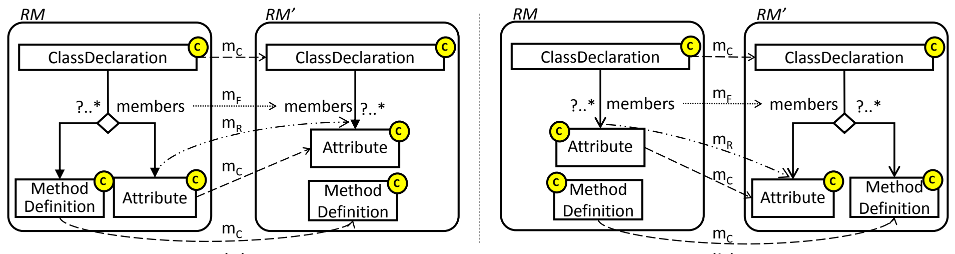


Fig. 13: DRM mapping examples illustrating *feature type refinement*. (a) Closed references. (b) Open references.

Figure 13(b) illustrates the semantics of open references, which is the converse. In this case, the predicate iterates on all target classes of the reference in RM, and checks that the reference mapped to it in RM' has compatible target classes for them. The refined reference may declare more compatible classes, as is the case in the figure (see MethodDefinition in RM'). This is useful to characterize write-only target meta-models.

▷ *subclass refinement:* this predicate requires that a class mapped by m_S is refined by all concrete subclasses of the class is mapped to. Recall from Definition 4.1 that a class can only be mapped by m_S if its flag $subsAllowed$ is $true$.

Predicate: Subclass refinement

$$\begin{array}{l} subs\_refinement(m) \triangleq \\ \quad \forall C \in RM_C \bullet\ m_S(C)\ is\ defined \Rightarrow |\ conc\_subs(m_S(C))\ | > 0\ \wedge \\ \quad\quad \forall C' \in conc\_subs(m_S(C)) \bullet \\ \quad\quad\quad mand\_allowed\_refinement(C, C') \wedge \\ \quad\quad\quad \forall f \in C.feats\ \exists f' \in C'.feats^* \bullet feats\_refinement(f, f') \end{array}$$

with:

$$\begin{aligned} conc\_subs(c : Class) &\triangleq \{c' \in c.subs \cup \{c\} \mid c'.isAbstract \neq true\} \\ feats\_refinement(f, f' : Feature) &\triangleq feature\_refinement(f, f') \wedge \\ &\quad\ feature\_type\_commut(f, f') \wedge \\ &\quad\ feature\_type\_refinement(f, f') \end{aligned}$$

The predicate checks that every non-abstract subclass of class C is mapped to refine the features of C. It also ensures that $m_S(C)$ has a non-empty set of non-abstract subclasses, or is itself non-abstract. The actual class refinement is assessed by predicates *mand_allowed_refinement(_,_)*, *feature_refinement(_,_)*, *feature_type_commut(_, _)*, and *feature_type_refinement(_, _)*. These predicates are analogous to the previous versions which receive a mapping as a parameter, but performing the checkings over particular pairs of classes or features.

Example. The mapping in Figure 14 satisfies $subs\_refinement$. This is so as m_S maps NamedElement to Element, and hence, the predicate checks that every non-abstract subclass of Element refines NamedElement regarding its mandatory features (predicate $mand\_allowed\_refinement$), features ($feature\_refinement$) and feature types ($feature\_type\_commut$ and $feature\_type\_refinement$). These predicates hold because both ClassD

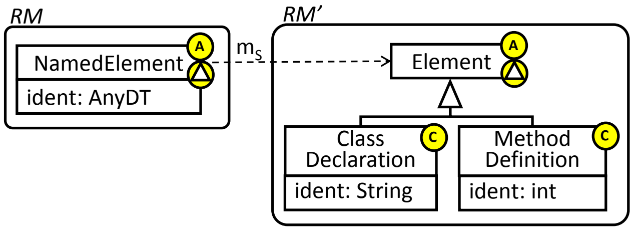


Fig. 14: DRM mapping example illustrating *subclass refinement*.

Once we have defined the predicates involved in DRM refinements, we illustrate this notion through an example.

Example. Figure 15 shows examples of correct (a, b, c) and incorrect (d) DRM refinements with respect to DRM (a) in Figure 7 (repeated on top of Figure 15 for convenience). In the figure, we have indicated the mapping of classes using equal numbers, while the mapping of features can be deduced by the equality of their names.

The mapping to DRM (a) is a refinement because it correctly maps the named classes MethodDefinition and ClassDeclaration (demanded by the $name\_refinement$ predicate). None of these two classes is an ancestor of the other, and the antiancs relation is preserved ($antiancs\_preservation$ predicate). Both MethodDefinition and ClassDeclaration have features with the same name and cardinality as the DRM classes they refine ($feature\_refinement$ predicate). Among these features, the attribute name is inherited from an anonymous class, and its type is refined from $AnyDT$ to $String$ ($feature\_type\_refinement$ predicate). In addition, MethodInvocation in DRM (a) also refines one of the anonymous classes in the DRM, while MethodDefinition refines another anonymous class, and

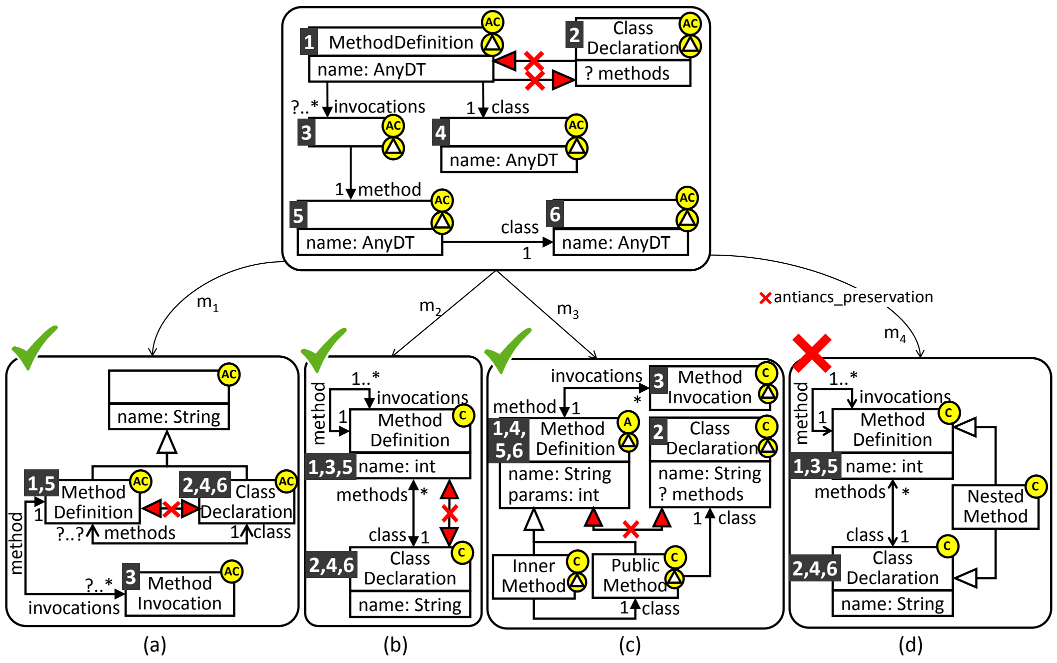


Fig. 15: Correct and incorrect DRM refinements with respect to DRM (a) in Figure 7.

ClassDeclaration refines two of them. Feature ClassDeclaration.methods in the DRM on top, which can be either a reference or an attribute, is refined by reference ClassDeclaration.methods in DRM (a).

The mapping to DRM (b) is also a refinement. All the classes in DRM (b) are concrete, as the classes they refine allow any abstractness (permitted by predicate $abstract\_refinement$). In this case, the name attribute is directly owned by the classes and has different types, both of which are proper refinements of the more general $AnyDT$. The four anonymous classes in the DRM on top are refined by just two classes. Some reference cardinalities in the refined and refining DRMs are different, such as the minimum cardinality of invocations that has been refined from $AnyCardinality$ to 1, which is allowed by predicate $feature\_refinement$.

The mapping to DRM (c) is a refinement because all concrete subclasses of the abstract class MethodDefinition in DRM (c), structurally conform to the class MethodDefinition in the upper DRM (as required by predicate $subs\_refinement$). This abstract class defines the mandatory attribute params that does not appear in the upper DRM, but predicate $mand\_allowed\_refinement$ allows this because class MethodDefinition on top allows extra mandatory attributes in the refining classes.

Mapping (d) is not a refinement because the antiancestor relation is not preserved (predicate $antiancs\_preservation$ fails). Class NestedMethod inherits from both MethodDefinition and ClassDeclaration, and so with reference to the transformation in Figure 1, NestedMethod objects would be matched by rules Table and MethodDefinition, causing a runtime error.

5.2. Refinement composition

Transformation reuse via transformation querying requires being able to perform a stepwise refinement of a DRM, likely concatenating simple refinement steps (see scenario in Section 3.2). However, the predicate $refinement(m)$ in Definition 5.2 is too weak to ensure compositionality of refinements. We need to strengthen it to ensure the value of flags *mandatoryAllowed*, *allowLess*, *allowMore* and *open* is preserved. The next three predicates capture this.

Predicate: Mandatory allowed preservation

$$mand\_allowed\_preservation(m) \triangleq$$
$$\forall C \in RM_C \bullet C.mandatoryAllowed = false \Rightarrow m_{CS}(C).mandatoryAllowed = false$$

Predicate: Cardinality preservation

$$\begin{aligned}&card\_preservation(m) \triangleq \forall C \in RM_C, \forall f \in C.feats \bullet\\ &\quad m_F(f)\ is\ defined \Rightarrow same\_bounds(f, m_F(f)) \wedge\\ &\quad m_F(f)\ not\ defined \Rightarrow \forall C' \in conc\_subs(m_S(C)),\ \exists f' \in C'.feats^* \bullet\\ &\qquad\qquad feats\_refinement(f, f') \ \wedge\ same\_bounds(f, f')\end{aligned}$$

with:

$$\begin{aligned}&same\_bounds(f, f' : Feature) \triangleq f.min \in Number \wedge f'.min \in Number \implies\\ &\quad f.min.allowLess = f'.min.allowLess \wedge\\ &\quad f.min.allowMore = f'.min.allowMore \qquad \text{(and similar for } f.max\text{)}\end{aligned}$$

Predicate: Reference semantics preservation

$$\begin{aligned}&ref\_sem\_preservation(m) \triangleq \forall f \in RM_F,\ \forall rt \in f.types \bullet\\ &\quad m_R(rt)\ is\ defined \Rightarrow rt.open = m_R(rt).open \wedge\\ &\quad m_R(rt)\ not\ defined \Rightarrow m_S(f.owner)\ is\ defined \implies\\ &\qquad \forall C' \in conc\_subs(m_S(f.owner)),\ \exists f' \in C'.feats^* \bullet\\ &\qquad\quad feats\_refinement(f, f') \ \wedge\ \forall rt' \in f'.types \bullet\\ &\qquad\quad refines_{Ref}(rt, rt') \Rightarrow rt.open = rt'.open\end{aligned}$$

Predicates $card\_preservation$ and $ref\_sem\_preservation$ have two parts. The first one checks cardinality (resp. reference semantics) preservation for those features mapped via m_F, which means their owner classes were mapped through m_C. The second part checks the same conditions for those features owned by classes mapped via m_S. As those features are not explicitly mapped, the checks need to use predicate $feats\_refinement$.

Next, we introduce the notion of strong refinement as a special type of refinement.

Definition 5.3 (*DRM strong refinement*). Given two DRMs RM and RM', the mapping $m = \langle m_C, m_S, m_F, m_R, m_A\rangle : RM \rightarrow RM'$ is a strong refinement if the following predicate holds:

$$\begin{aligned}srefinement(m) \triangleq\ & refinement(m) \wedge mand\_allowed\_preservation(m) \wedge\\ & card\_preservation(m) \ \wedge ref\_sem\_preservation(m)\end{aligned}$$

Example. In Figure 15, refinements (b) and (c) are strong. Refinement (a) is not strong because reference class does not preserve its semantics (it is closed in the DRM on top, and open in DRM (a)). Refinements (b) and (c) are strong because they preserve the semantics of references and *mandatoryAllowed*. Please note that strong refinements are not required to preserve $subsAllowed$.

The following definition describes when two DRM mappings are composable and how to compose them. Then, Lemma 5.5 states that composing two DRM mappings yields a well-formed DRM mapping. Finally, Theorem 5.6 defines the conditions under which the composition of two strong refinements leads to a strong refinement.

Definition 5.4 (*DRM mapping composition*). Given two DRM mappings $m^1 = \langle m^1_C, m^1_S, m^1_F, m^1_R, m^1_A\rangle : RM \rightarrow RM'$ and $m^2 = \langle m^2_C, m^2_S, m^2_F, m^2_R, m^2_A\rangle : RM' \rightarrow RM''$ such that:

$$\begin{aligned}&\forall C \in RM_C \bullet m^2_S(m^1_C(C))\ is\ defined \Rightarrow (C.subsAllowed = true \wedge\\ &\qquad \forall C_s \in C.subs \bullet m^1_S(C_s)\ is\ defined \ \vee\ m^2_S(m^1_C(C_s))\ is\ defined)\end{aligned}$$

the composition $m^2 \circ m^1$ yields the DRM mapping $m = \langle m_C, m_S, m_F, m_R, m_A \rangle \colon RM \to RM''$ defined componentwise as follows:

— $m_X = m^2_X \circ m^1_X$ for $X \in \{C, F, R, A\}$
— $m_S = m^2_S \circ m^1_S \cup m^2_S \circ m^1_C \cup m^2_C \circ m^1_S$

The composition of DRM mappings is calculated componentwise. The composability condition in the definition ensures on the one hand that the resulting function m_S is only defined for classes with $subsAllowed = true$, as required by the definition of DRM mapping (cf. Definition 5.1). On the other hand, it also ensures that if the resulting m_S is defined for a class C, it is also defined for all its subclasses C_s, as also required by the definition of DRM mapping.

Example. Figure 16 shows an example of DRM mapping composition. In the composed mapping, ClassDec
are map

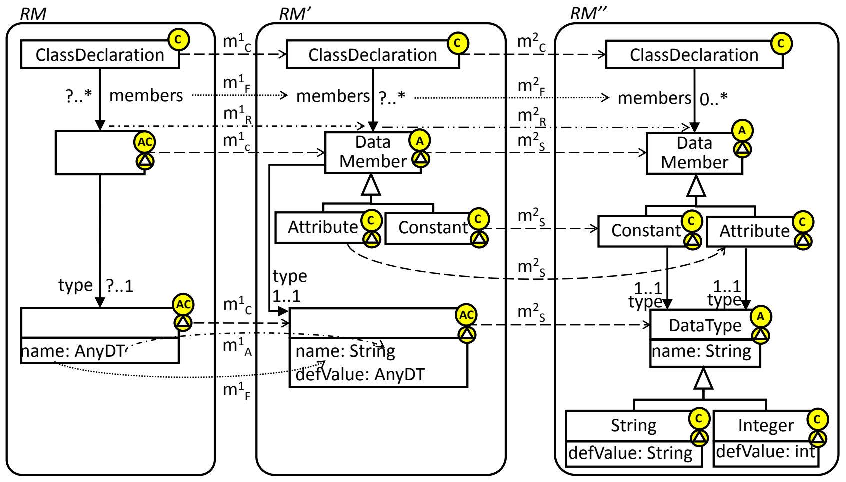


Fig. 16: Example of DRM mapping composition.

Before proving properties of refinement compositions, we deal with a more basic property of composed DRM mappings: that they are well-formed (i.e., they fulfil the conditions in Definition 5.1).

LEMMA 5.5 (COMPOSITION OF DRM MAPPINGS IS WELL-FORMED).
Given two composable DRM mappings $m^1 \colon RM \to RM'$ and $m^2 \colon RM' \to RM''$, then $m^2 \circ m^1 \colon RM \to RM''$ is a well-formed DRM mapping.

PROOF. In Appendix A.1. □

Finally, we are ready to characterize when the composition of two strong refinements yields a strong refinement. We use *cod(m)* to refer to the codomain of the mapping m (i.e., the target), and *dom(m)* for its domain (i.e., the source). This theorem is important as it enables the stepwise refinement of DRMs.

THEOREM 5.6 (COMPOSITION OF STRONG REFINEMENTS IS STRONG REFINEMENT).
Given two composable DRM mappings $m_1 \colon RM \to RM'$ and $m_2 \colon RM' \to RM''$ such that

$$cod(m^1_S) \cap dom(m^2_{CS}) = \emptyset$$

we have that

$$srefinement(m_1) \wedge srefinement(m_2) \Rightarrow srefinement(m_2 \circ m_1)$$

PROOF. In Appendix A.2. □

Example. Mappings m^1 and m^2 in Figure 16 are strong refinements, and no class in RM' belongs to both the codomain of m^1_S and the domain of m^2_{CS}; hence, their composition $m = m^2 \circ m^1$ is a strong refinement. This means that the predicate $name\_refinement$ holds for m because the name of both ClassDeclaration and $m_C($ ClassDeclaration$)$ is the same; class ClassDeclaration is concrete in both RM and RM'', and the abstractness of both anonymous classes in RM is any; $ancs\_preservation$ and $antiancs\_preservation$ hold because $ancs$ and $antiancs$ are empty in RM; $mand\_allowed\_refinement$ and $mand\_allowed\_preservation$ hold because all classes have the flag *mandatoryAllowed* set to *true*; $feature\_refinement$ holds because members in RM is mapped to a feature named analogously in RM'', and their owner classes are mapped as well; $feature\_type\_commut$ and $feature\_type\_refinement$ hold because the type of members is preserved; $subs\_refinement$ holds because the subclasses of DataMember and DataType in RM'' refine the anonymous classes in RM and their features type and name; $ref\_sem\_preservation$ holds because all references have close semantics; and $card\_preservation$ holds because $allowLess$ and $allowMore$ are preserved (even though Figure 16 does not represent this information visually).

Example. Figure 17 shows the composition of two strong refinements that does not fulfil the precondition to obtain a strong refinement, as DataType in RM' belongs to both the codomain of m^1_S and th
DataTyp
that do

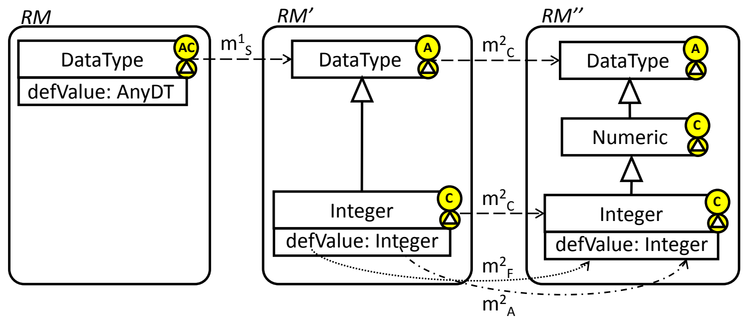


Fig. 17: Composition of strong refinements that violates the precondition to yield a strong refinement ($cod(m^1_S) \cap dom(m^2_{CS})$ is not empty).

5.2.1. Refinement operations. In the following, we present a catalogue of refinement operations that can be successively applied to a given DRM, and ensure that the resulting DRM is a strong refinement of the initial one. This is relevant for the scenario described in Section 3.2, where TRMs are refined to reduce the number of retrieved transformations. The proposed catalogue is not exhaustive, but other refinements are possible and can be added to our catalogue if proved to be strong refinements. Figure 18 gives minimal examples of each refinement operation in the catalogue, while the appendix contains the proofs of correctness for each of them.

(1) *Adding new class*. Given a DRM RM, adding a new class C' yields a modified DRM RM' which strongly refines RM.

PROOF. In Appendix A.4. □

(2) *Assigning fresh name to anonymous class*. Given a DRM RM with an anonymous class C, assigning a fresh name to the class yields a modified class C' (which therefore becomes a named class) and a modified DRM RM', which strongly refines RM.

PROOF. In Appendix A.5. □

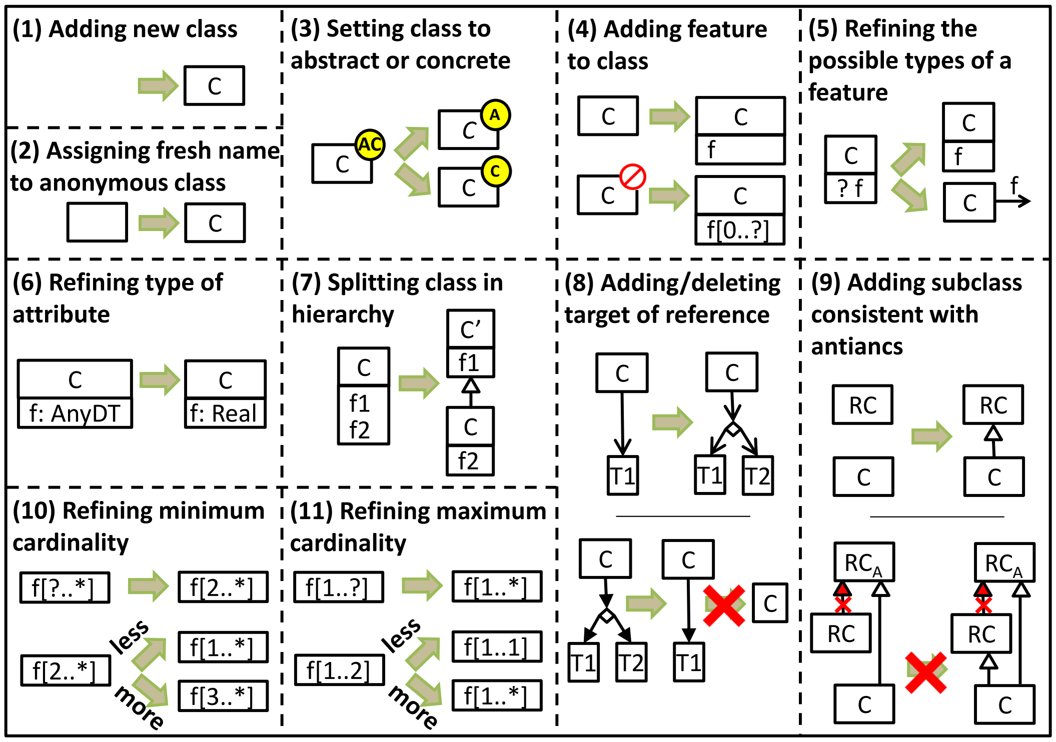

(1) Adding new class
C
(2) Assigning fresh name to anonymous class
C
(3) Setting class to abstract or concrete
C
AC
C
A
C
C
(4) Adding feature to class
C
C
f
C
C
f[0..?]
(5) Refining the possible types of a feature
C
? f
C
f
C
f
(6) Refining type of attribute
C
f: AnyDT
C
f: Real
(7) Splitting class in hierarchy
C
f1
f2
C′
f1
C
f2
(8) Adding/deleting target of reference
C
T1
C
T1
T2
C
T1
T2
C
T1
C
(9) Adding subclass consistent with antiancs
RC
C
RC
C
RC_A
RC
C
RC_A
RC
C
(10) Refining minimum cardinality
f[?..*]
f[2..*]
f[2..*]
less
more
f[1..*]
f[3..*]
(11) Refining maximum cardinality
f[1..?]
f[1..*]
f[1..2]
less
more
f[1..1]
f[1..*]

Fig. 18: Cheat sheet with refinement operations.

(3) *Setting class to abstract or concrete, provided it allows any abstractness.* Given a DRM RM with a class C such that $C.isAbstract = any$, changing $isAbstract$ to $true$ or $false$ yields a modified class C' and a modified DRM RM', which strongly refines RM.

PROOF. In Appendix A.6. □

(4) *Adding feature to class, which must be optional if the class does not allow mandatory features.* Given a DRM RM and a class C, adding a new feature f to $C.feats$ yields a modified class C' and a modified DRM RM', which strongly refines RM. If $C.mandatoryAllowed = false$, then $f.min$ must be a $Number$ n with $n.value = 0$ and $n.allowMore = false$.

PROOF. In Appendix A.7. □

(5) *Refining the possible types of a feature.* Given a DRM RM with a feature f such that $|f.types| = 0$ (meaning that f can have any type), adding a feature type to $f.types$ yields a modified DRM RM', which strongly refines RM. This effectively implies that f is refined into an attribute or a reference (see Figure 18). Conversely, if $|f.types| > 1$, then deleting one element from $f.types$ yields a modified DRM RM', which strongly refines RM.

PROOF. In Appendix A.8. □

(6) *Refining type of attribute.* We consider the refinement of untyped and numeric attributes. In the first case, given a DRM RM with an attribute f such that $f.dtype = AnyDT$, the operation changes $f.dtype$ to a datatype from the set $\{String, Boolean, Numeric, Real, Integer, Enum\}$, yielding a modified attribute f' and a modified DRM RM', which strongly refines RM. In the second case, $f.dtype = Numeric$, and the operation changes the datatype by one from the set $\{Real, Integer\}$, which also yields a strong refinement.

PROOF. In Appendix A.9. □

(7) *Splitting class in hierarchy*. Given a DRM RM with a class C, creating a new class C_{super} which is added to C's ancestors yields a modified class C' and a modified DRM RM', which strongly refines RM. Optionally, any feature $f \in C.feats$ can be pushed up to C_{super}, yielding a modified feature f'. As an example, Figure 18 (7) shows a class C that is split, and its feature f1 is moved to the new superclass.

PROOF. In Appendix A.10. □

(8) *Adding/deleting target of reference*. This effectively implies refining the type of a reference. Given a DRM RM with a reference r such that $r.open = true$, adding a new or existing class to $r.targets$ yields a modified reference r' and a modified DRM RM', which strongly refines RM. Conversely, given a reference r such that $r.open = false \wedge |r.targets| > 1$, removing a class from $r.targets$ also yields a strong refinement. Even though the condition for closed references to have more than one target is not needed according to predicate $feature\_type\_refinement$, we require it because closed references without targets cannot be further refined, and hence, no meta-model (in which all references have exactly one target) would "conform" to the DRM.

PROOF. In Appendix A.11. □

(9) *Adding subclass consistent with antiancs*. Given a DRM RM with classes RC and C s.t.

$$\forall C_a \in C.ancs^* \bullet (\nexists C_s \in C_a.antiancs^* \bullet C_s \in RC.ancs^* \wedge \\ \nexists C_s \in RC.ancs^* \bullet C_a \in C_s.antiancs^*)$$

adding RC to $C.ancs$ yields a DRM RM' which strongly refines RM. If the subclass C is new, we also obtain a strong refinement. Figure 18 (9) shows two examples of subclassing, the upper one corresponding to a valid refinement, and the second one not because the subclass C inherits from a class RC_A that belongs to $RC.antiancs$.

PROOF. In Appendix A.12. □

(10) *Refining minimum cardinality*. Given a DRM RM with a feature f, any of the following changes yields a modified feature f' and a modified DRM RM' that strongly refines RM: if $f.min = AnyCardinality$, assigning to $f.min$ any $Number\ n$; if $f.min = Number \wedge f.min.allowLess = true$, assigning a lower value to $f.min.value$; and if $f.min = Number \wedge f.min.allowMore = true$, assigning a higher value to $f.min.value$.

PROOF. In Appendix A.13. □

(11) *Refining maximum cardinality*. Given a DRM RM with a feature f, any of the following changes yields a modified feature f' and a modified DRM RM' that strongly refines RM: if $f.max = AnyCardinality$, assigning to $f.max$ any $Number\ n$ or $Many$; if $f.max = Number \wedge f.max.allowLess = true$, assigning a lower value to $f.max.value$; if $f.max = Number \wedge f.max.allowMore = true$, assigning either a higher value to $f.max.value$ or $Many$ to $f.max$.

PROOF. In Appendix A.14. □

5.3. Meta-models as domain requirement models

Regular meta-models can be considered a special case of DRM. This observation permits unifying the notions of DRM refinement and conformance of meta-models to DRMs, by a unique relationship between DRMs. The following definition enumerates the features that an encoding of meta-models as DRMs yields.

Definition 5.7 (*Meta-model*). A DRM RM is called a meta-model if:

(1) classes have a name: $AC = \emptyset$

(2) classes are either abstract or concrete: $\forall c \in C \bullet \neg c.isAbstract = any$
(3) features are either references or attributes: $\forall f \in F \bullet |f.types| = 1$
(4) features have a concrete cardinality value: $\forall f \in F \bullet \{f.min, f.max\} \subseteq Number \cup Many$
(5) attributes have a concrete type: $\forall a \in A \bullet \neg a.dtype = AnyDT \wedge \neg a.dtype = Numeric$
(6) references have exactly one target type: $\forall r \in R \bullet |r.targets| = 1$
(7) references are closed: $\forall r \in R \bullet r.open = false$
(8) cardinalities are strict: $\forall n \in Number \bullet n.allowLess = false \wedge n.allowMore = false$
(9) all possible antiancestors are declared: $\forall c_1, c_2 \in C \bullet c_1 \notin c_2.ancs^* \wedge c_2 \notin c_1.ancs^* \implies ((c_1 \in c_2.antiancs \wedge c_2 \in c_1.antiancs) \vee \exists c' \in C \bullet \{c_1, c_2\} \subseteq c'.ancs^*)$

These conditions are necessary to faithfully capture the instantiation semantics of meta-models, which for example requires named classes and features with a defined type. The last condition reflects the fact that, implicitly, meta-models declare all possible antiancestors (i.e., those not conflicting with the $ancs$ relation, so that the meta-model is a valid DRM according to Definition 4.1). This is because, in the standard meta-model semantics, an object o cannot be typed by two classes, unless one is subclass of the other. The $antiancs$ relation is used in this case to make explicit this restriction. As we will see in Section 5.4, general DRMs admit objects typed by several classes not related by inheritance, provided the $antiancs$ relation is not violated. In meta-models, references are closed, as a link typed by a certain reference can only contain instances of the target classes of the reference. The definition does not state any conditions on *subsAllowed* or *mandatoryAllowed*, as these express expectations on other refining DRMs, and hence any value is allowed.

Example. In Figure 15, only DRM (b) is a meta-model. DRM (a) is not a meta-model because it contains an anonymous class, some classes have no defined abstractness, some references are open, and some cardinalities are not concrete (i.e., min or max have the value $AnyCardinality$). DRM (c) is not a meta-model because several $antiancs$ relations are missing. DRM (d) is not a meta-model either because it contains open references.

5.4. Semantics of refinement

Similarly to the instantiation relationship of models with respect to meta-models ($type : M \rightarrow MM$), we can type models with respect to DRMs ($type^* : M \rightarrow DRM$). This way, a transformation q defined over a DRM can be applied to any model typed by the DRM (see left of Figure 5). This typing is in general non-constructive due to the variability that DRMs entail. Moreover, whereas the typing of models by meta-models is normally total, their typing with respect to DRMs is partial, as DRMs have open-world semantics. This is so as models are allowed to contain objects that are not typed by any DRM class, and objects can assign a value to features not declared by the object's class. Some typing rules for DRMs are also less strict than those for meta-models. For instance, in DRMs, the typing of objects is nominal for named classes but structural for anonymous ones, the checking of cardinalities is not performed for features that admit any cardinality, the validity of reference and attribute values is not checked unless the DRM specifies their type (recall that the DRM can omit the type of features if it is unimportant for the associated transformation), and objects may be typed by an abstract class. The latter is needed as an abstract class in a DRM may abstract away a hierarchy of classes via m_S.

Example. Figure 19 shows a model M typed by a meta-model MM (encoded as a DRM), and its retyping $type^*$ with respect to a DRM RM of which MM is a refinement. Object m is retyped by the named class MethodDefinition and provides a value to its attribute name. The type of such attribute is String in the type to MM, but becomes AnyDT in the typing to RM. Object c is retyped by both anonymous classes, AC1 and AC2, because ClassDeclaration is mapped from both classes. Object c provides a value to both attributes name using a single slot which is typed by both name attributes. Object p gets no type from RM because class Package is not mapped from any class in RM.

Next, we capture this intuition formally. First, we define a simple notion of model.

Definition 5.8 (*Model*). A model is a tuple $M = \langle O, F, L, slots, target \rangle$ made of:

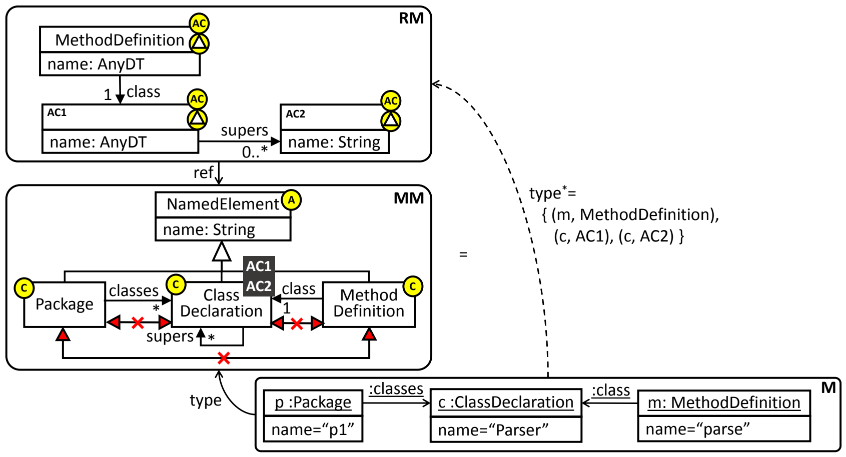


Fig. 19: Retyping a model with respect to a DRM.

— A set O of objects.
— Sets F and L of fields and links. We use $S = F \cup L$ to denote the set of slots.
— An injective relation $slots \subseteq O \times S$ assigning slots to objects.
— A function $target\colon L \to O$ assigning target objects to links.

For simplicity, the previous definition abstracts away field values, as they are not essential for our purposes. Given a model M, we write M_O to refer to its set of objects (and similar for sets F, L, and S). We use the auxiliary function $owner\colon S \to O$ to return the object owning a given slot. As before, we sometimes use an "object-oriented" notation for functions and relations (e.g., $o.owner$ instead of $owner(o)$).

Next, we define the typing of a model by a DRM.

Definition 5.9 (*Model typing*). Given a model M and a DRM RM, a model typing $type\colon M \to RM$ is a mapping $type = \langle type_{objs}, type_{slots}\rangle$ made of two relations:

— $type_{objs} \subseteq M_O \times RM_C$ mapping objects to classes
— $type_{slots} \subseteq M_S \times RM_F$ mapping slots to features

This definition permits objects with zero, one or multiple typings. The latter is needed when an object is typed by two or more classes, each describing a different object facet. For example, object c in Figure 19 receives two types from RM because ClassDeclaration in MM is mapped from two classes in RM (AC1 and AC2). Similarly, slots may receive no typing, one, or many. In the figure, slot c.name is typed both by the name attribute of the two anonymous classes in RM.

Next, we provide the well-formedness conditions for model typings.

Definition 5.10 (*Well-formed model typing*). A typing $type\colon M \to RM$ is *well-formed* iff:

(1) Objects are not typed by two classes such that one class is antiancestor of any ancestor of the other:

$$\forall o \in M_O,\ \forall c_1, c_2 \in RM_C \bullet \{c_1, c_2\} \subseteq type_{objs}(o) \implies \nexists c_1' \in c_1.ancs^* \bullet c_1' \in c_2.extantiancs$$
$$with\ extantiancs(c : Class) \triangleq \{c' : Class \mid \exists d \in c.ancs^* \bullet c' \in d.antiancs^+\}$$

(2) Slots are typed by features owned or inherited by some of the object types:

$$\forall s \in M_S,\ \forall f \in type_{slots}(s) \bullet \exists c \in type_{objs}(s.owner) \bullet f \in c.feats^*$$

(3) The type of every slot is not contradictory with that of the feature:

$$\begin{aligned}\forall s \in M_S,\ \forall f \in type_{slots}(s) \bullet\ & s\ \in M_F \implies f.types = \emptyset \vee \exists a \in f.types\ \bullet\ a \in RM_A \wedge \\ & s\ \in M_L \implies f.types = \emptyset \vee \exists r \in f.types\ \bullet\ r \in RM_R\end{aligned}$$

(4) Slots obey the minimum cardinality of their types:

$$\begin{aligned}&\forall o \in M_O,\ \forall c \in type_{objs}(o),\ \forall f \in c.feats^* \bullet \\ &\qquad f.min \in Number \wedge f.min.allowLess = false \implies \\ &\qquad\qquad |\{s \in slots(o) \mid f \in type_{slots}(o)\}| \geq f.min.value\end{aligned}$$

(5) Slots obey the maximum cardinality of their types:

$$\begin{aligned}&\forall o \in M_O,\ \forall c \in type_{objs}(o),\ \forall f \in c.feats^* \bullet \\ &\qquad f.max \in Number \wedge f.max.allowMore = false \implies \\ &\qquad\qquad |\{s \in slots(o) \mid f \in type_{slots}(o)\}| \leq f.max.value\end{aligned}$$

(6) Every link target is coherent with its type:

$$\begin{aligned}&\forall f \in RM_F, \forall l \in M_L\ \bullet f \in type_{slots}(l) \implies \\ &\qquad \forall r \in f.types\ \bullet r.open = false \implies \\ &\qquad\qquad \forall c' \in type_{objs}(l.target)\ \bullet\ c'.ancs^* \cap r.targets \neq \emptyset\end{aligned}$$

A typing is called *write well-formed* if in addition it satisfies the following condition:

(7) Objects do not have an abstract typing:

$$\forall o \in M_O,\ \forall c \in RM_C\ \bullet\ c \in type_{objs}(o) \implies c.isAbstract \neq true$$

Next, we show how a model typed by a DRM RM' can be retyped with respect to another DRM RM that is refined by RM'. The scheme to the left of Figure 5 showed the usefulness of this retyping in our context: a model is typed by a meta-model over which a transformation q is defined, and then it gets retyped by the DRM extracted from the transformation.

Definition 5.11 (*Retyping by refinement*). Given a DRM refinement $m\colon RM \to RM'$ and a well-formed model typing $type\colon M \to RM'$, we can retype M w.r.t. RM by applying a function $back$ to $type$, yielding $back(type) = type^*\colon M \to RM$, where $type^* = \langle type^*_{objs}, type^*_{slots}\rangle$ is calculated as follows:

$$\begin{aligned}type^*_{objs} = \{(o,c) \in objBck(type,m) \mid\ & (m_C(c)\ is\ defined \implies \\ &\quad \forall\ d\ \in RM_C\ \bullet (o,d) \in objBck(type,m)\ \wedge \\ &\quad\quad c \in d.ancs^+ \implies m_S(d)\ is\ defined) \wedge \\ & (m_S(c)\ is\ defined \implies \\ &\quad (\nexists d \in RM_C\ \bullet (o,d) \in objBck(type,m)\ \wedge \\ &\quad\quad d \in c.ancs^+ \wedge m_C(d)\ is\ defined) \wedge \\ &\quad (\nexists d \in RM_C\ \bullet (o,d) \in objBck(type,m)\ \wedge \\ &\quad\quad c \in d.ancs^+ \wedge m_S(d)\ is\ defined))\}\end{aligned}$$

$$\begin{aligned} type^*_{slots} = \ & \{(s,f) \mid s \in M_S \wedge f \in RM_F \wedge m_F(f) \ is \ defined \wedge \\ & \quad m_F(f) \in type_{slots}(s) \wedge \exists d \in type^*_{objs}(s.owner) \wedge f.owner \in d.ancs^*\} \cup \\ & \{(s,f) \mid s \in M_S \wedge f \in RM_F \wedge m_F(f) \ not \ defined \wedge \\ & \quad \exists da \in type^*_{objs}(s.owner) \bullet f.owner \in da.ancs^* \wedge \\ & \quad \exists t \in type_{objs}(s.owner) \ \bullet \ m_S(f.owner) \in t.ancs^* \wedge \\ & \quad \exists f' \in t.feats^* \ \bullet \ (f' \in type_{slots}(s) \ \wedge feature\_refinement(f,f') \wedge \\ & \quad feature\_type\_refinement(f,f') \ \wedge feature\_type\_commut(f,f')) \wedge \\ & \quad (f.max \in Number \wedge f.max.allowMore = false) \implies \\ & \quad |\{s \in o.slots \mid type^*_{slots}(s) = f\}| < f.max.value\} \end{aligned}$$

with:

$$\begin{aligned} objBck(type, m) \triangleq \ & m_C^{-1} \circ type_{objs} \cup \\ & \{(o,c) \mid o \in M_O \wedge c \in RM_C \wedge o \notin dom(m_C^{-1} \circ type_{objs}) \wedge \\ & \ \exists d' \in type_{objs}(o) \ \bullet \ m_{CS}(c) \in d'.ancs^*\} \end{aligned}$$

The retyping of objects uses the auxiliary function $objBck$. This function obtains the classes in RM_C by inverting the m_C map, and for those objects that remain untyped, it adds the classes mapped to an ancestor of any of the object's types. Then, to build $type^*_{objs}$ for a given object o, from all candidate classes in $objBck$, we take either a class c mapped via m_C for which all its subclasses (compatible with o) are mapped via m_S, or else we take a class c mapped via m_S having no compatible ancestor mapped via m_C and no compatible subclass mapped via m_S. This means that m_C has preference over m_S to build $type^*_{objs}$, and then, we take the most specific m_C or m_S.

For slots, we consider two cases. The first one is for slots typed by features of classes mapped through m_C; in such cases, m_F should be defined, and the mapping traverses it back. The second one is for slots typed by features of classes mapped through m_S; in such cases, there is no explicit mapping of features, so we reconstruct it using the predicates required by $subs\_refinement$ ($feature\_refinement$, $feature\_type\_refinement$, $feature\_type\_commut$) and ensuring proper maximum cardinality.

Example. Figure 20 illustrates how a model M typed by a DRM RM' is retyped to a more abstract DRM RM. The figure depicts the refinement $m\colon RM \to RM'$ using dashed arrows, and the typing $type\colon M \to RM'$ as a set of tuples, the first two belonging to $type_{objs}$ and the rest to $type_{slots}$. The objects in M also show the type received by the typing $type$. The typing $type^*\colon M \to RM$ is calculated by applying the function $back$ of Definition 5.11. For slots, it amounts to "following" $type_{slots}$ to RM', and then the arrows from RM_F to RM' backwards. In addition, both objects get retyped by the anonymous class (identified by "a"). This is so as the refinement maps both classes in RM to ClassDeclaration in RM', and then, the retyping assigns the most specific one (which is the anonymous class) as the type of the objects.

Retyping through arbitrary refinements does not necessarily yield a well-formed typing. Instead, as the following theorem states, we require the refinement to be strong, or the target DRM to be a meta-model. This result is exactly the one we need as, in practice, we will start with a model typed by a meta-model, and then extract a DRM of which the meta-model is a refinement. In order to retype the model using more abstract DRMs, strong refinements are required.

THEOREM 5.12 (RETYPING IS WELL-FORMED). *Given a DRM refinement $m\colon RM \to RM'$ and a write well-formed model typing $type\colon M \to RM'$, $back(type)\colon M \to RM$ is well-formed if:*

(a) RM' is a meta-model, **or**
(b) $srefinement(m)$ holds

PROOF. In Appendix A.15. □

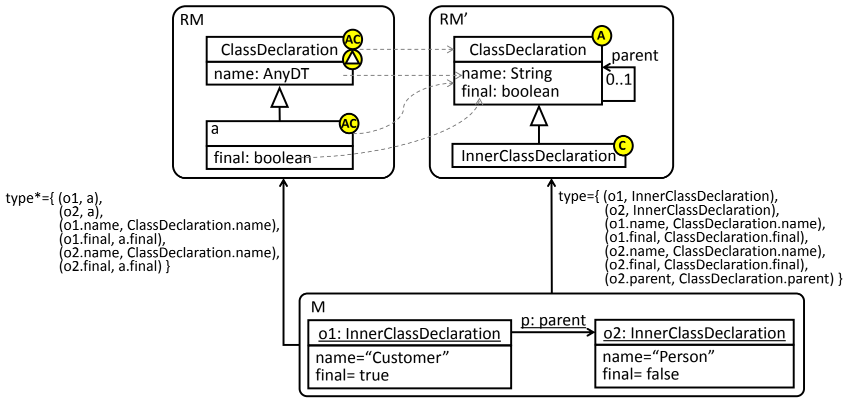

RM
ClassDeclaration
AC
name: AnyDT
a
AC
final: boolean
RM'
ClassDeclaration
A
parent
name: String
final: boolean
0..1
InnerClassDeclaration
C
type*={ (o1, a),
(o2, a),
(o1.name, ClassDeclaration.name),
(o1.final, a.final),
(o2.name, ClassDeclaration.name),
(o2.final, a.final) }
type={ (o1, InnerClassDeclaration),
(o2, InnerClassDeclaration),
(o1.name, ClassDeclaration.name),
(o1.final, ClassDeclaration.final),
(o2.name, ClassDeclaration.name),
(o2.final, ClassDeclaration.final),
(o2.parent, ClassDeclaration.parent) }
M
o1: InnerClassDeclaration
p: parent
o2: InnerClassDeclaration
name="Customer"
final= true
name="Person"
final= false

Fig. 20: Retyping a typed model to a more abstract DRM.

6. TRANSFORMATION TYPING REQUIREMENTS MODELS

A DRM describes the possible choices for a meta-model (or more generally another DRM) to satisfy the source/target typing requirements of a transformation. However, a choice for an open element of the source (resp. target) DRM may forbid some options of the target (resp. source) DRM in case such options break the syntactic correctness of the transformation. For instance, in the transformation shown in Figure 1, the binding rows $\leftarrow$ s.methods constrains the possible types of the rows and methods fields to those that yield a non-faulty execution.

Hence, we gather the 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 dependencies between leaf nodes so that the occurrence of a leaf node forces the presence of the dependent nodes.

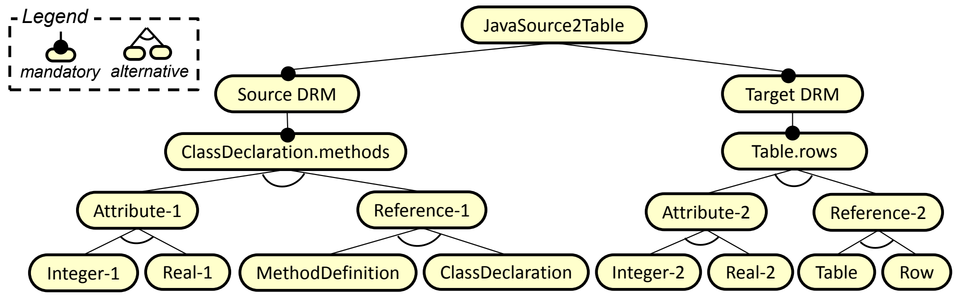
For this purpose, we first provide the definition of a feature model.

Definition 6.1 (*Feature model*). A feature model $FM = (F, \phi)$ consists of a set of features $F = \{f_1, ..., f_n\}$ and a propositional formula ϕ that defines relations between them.

Example. The upper part of Figure 21 shows an excerpt of the compatibility model for the running example using the classical feature diagram notation [Kang et al. 1990], and the bottom includes the propositional formula ϕ implied by the model. The compatibility model 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 12 of the transformation in Figure 1 as part of a binding. If it is an attribute, then it can have any data type (the figure only shows Integer and Real for simplicity). 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. Such dependencies are visually depicted as compatibility rules in the feature diagram, and formally expressed as terms in the formula ϕ. As the compatibility rules illustrate, dependencies also work from target to source.

A feature model permits selecting configurations of features. Each configuration is a subset of features that makes the formula ϕ $true$. Moreover, configurations can be partial, that is, not all independent features of the model may have been selected.

Definition 6.2 (*Feature configuration*). A valid feature configuration ρ of a feature model FM is a subset of its features that satisfies ϕ, i.e., ϕ evaluates to $true$ when each variable $f \in \phi$ is substituted by $true$ when $f \in \rho$, and by $false$ otherwise. We use $P = \{\rho_i\}_{i \in I}$ to denote



```
Compatibility rules (source choices)
  Integer-1 requires Integer-2 or Real-2
  Real-1 requires Real-2
  MethodDefinition requires Row
  ClassDeclaration requires Table
```

```
Compatibility rules (target choices)
  Integer-2 requires Integer-1
  Real-2 requires Integer-1 or Real-1
  Table requires ClassDeclaration
  Row requires MethodDefinition
```

Formula implied by the feature model:

```
ϕ = JavaSource2Table ∧ Source DRM ∧ ClassDeclaration.methods ∧ Target DRM ∧ Table.rows
    ((Attribute-1 ∧ ¬Reference-1) ∨ (¬Attribute-1 ∧ Reference-1)) ∧
    ((Attribute-2 ∧ ¬Reference-2) ∨ (¬Attribute-2 ∧ Reference-2)) ∧
    ((Integer-1 ∧ ¬Real-1) ∨ (¬Integer-1 ∧ Real-1)) ∧
    ((MethodDefinition ∧ ¬ClassDeclaration) ∨ (¬MethodDefinition ∧ ClassDeclaration)) ∧
    ((Integer-2 ∧ ¬Real-2) ∨ (¬Integer-2 ∧ Real-2)) ∧
    ((Table ∧ ¬Row) ∨ (¬Table ∧ Row)) ∧
    (Integer-1 ⇒ (Integer-2 ∨ Real-2)) ∧ (Real-1 ⇒ Real-2) ∧
    (MethodDefinition ⇒ Row) ∧ (ClassDeclaration ⇒ Table) ∧
    (Integer-2 ⇒ Integer-1) ∧ (Real-2 ⇒ (Integer-1 ∨ Real-1)) ∧
    (Table ⇒ ClassDeclaration) ∧ (Row ⇒ MethodDefinition)
```

Fig. 21: Excerpt of the compatibility model for the running example.

the set of all valid configurations. A feature configuration ρ is called a partial configuration if $\rho \notin P \ \wedge \ \exists \, \rho_i \in P \ \bullet \ \rho \subseteq \rho_i$.

Example. The excerpt of the feature model in Figure 21 admits the following five configurations (we only show leaf features, i.e., with no children): $P = \{\langle$ Integer-1, Integer-2$\rangle$, $\langle$ Integer-1, Real-2$\rangle$, $\langle$ Real-1, Real-2$\rangle$, $\langle$ MethodDefinition, Row$\rangle$, $\langle$ ClassDeclaration, Table$\rangle\}$. These configurations represent the type choices for ClassDeclaration.methods and Table.rows that yield a well-typed transformation.

We have defined the meta-model shown in Figure 22 by following the previous definitions of *feature model* and *feature configuration*. It allows expressing the kind of feature models our approach needs, which produces compatibility rules of the form $F^1 \implies F^2 \vee ... \vee F^n$ (see Figure 21). Class CompatibilityRule in the meta-model permits declaring this kind of dependencies among source and target features, like " MethodDefinition requires Row" (given by the formula MethodDefinition $\implies$ Row) , which specifies th [illegible]
the source DRM,

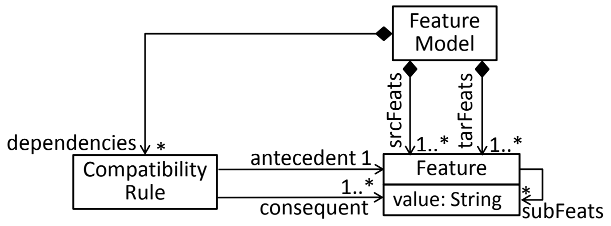


Fig. 22: Compatibility meta-model.

Each valid configuration identifies a consistent set of choices among the open options in the source and target DRMs of a transformation. To ensure the transformation remains syntactically correct, each choice can be implemented as a refinement operation, like those proposed in Section 5.2.1. This relation between choices and corresponding refinements is captured by the notion of typing requirements model (TRM). This is made of a feature model, source and target DRMs, and a specification of the refinement operation to be executed on the source or target DRM when a certain feature is selected.

Definition 6.3 (*Typing requirements model*). A typing requirements model is a tuple $TRM = \langle FM, RM_s, RM_t, \mu_s, \mu_t\rangle$ made of:

— A feature model $FM = (F, \phi)$,
— Two DRMs RM_s and RM_t called source and target,
— Two functions $\mu_s\colon F \rightarrow OP$ and $\mu_t\colon F \rightarrow OP$ from features of FM to strong refinement operations over RM_s and RM_t respectively. Each strong refinement operation in μ_s (resp. μ_t) must be independent of each other.

A TRM assigns a refinement operation to each feature in the compatibility model. If selecting a feature should not produce any effect on a DRM, the identity refinement operation (which does nothing) can be assigned to that feature. This approach is conceptually similar to delta modelling [Clarke et al. 2015], a transformative approach to product lines. In our setting, the transformations that the product line applies to our products (the DRMs) are strong refinements.

Our notion of TRM can be easily generalized to capture requirements of transformations with multiple input and output models by enabling a set of DRMs (instead of two) and a set of functions μ (instead of two).

Example. The TRM for the running example contains a feature model of which Figure 21 is an excerpt, and functions μ_s and μ_t assign the following refinement operations to its features:
— Integer-1, Integer-2: "*refining type of attribute to integer*" ∘ "*assigning attribute type to open feature*"
— Real-1, Real-2: "*refining type of attribute to real*" ∘ "*assigning attribute type to open feature*"
— MethodDefinition, ClassDeclaration, Table, Row: "*assigning reference type to open feature*"
— rest of features: "*identity refinement*"

For illustration, we have associated the composition of two refinement operations (numbers 5 and 6 in our catalogue, see Figure 18) to features Integer-1, Integer-2, Real-1 and Real-2. This is allowed as, by Theorem 5.6, the composition of two strong refinements is a (strong) refinement. Regarding non-leaf features, they typically (but not necessarily) imply the identity refinement, not modifying the DRMs.

Given a TRM and a feature configuration, we apply the refinement operations associated with the selected features in order to synchronously refine both DRMs in the TRM in a consistent way. This is called a *configuration-based TRM refinement*. The selected feature configuration can be partial (i.e., leaving open choices). The DRMs in a TRM can also be refined independently from each other, but in that case, the performed refinements cannot interfere with the choices captured in the compatibility model. This is called a *free TRM refinement*. In both cases, the resulting DRMs have reduced open choices, hence contributing to the transformation querying scenario presented in Section 3.2, which requires the ability to refine TRMs (see Figure 4).

Definition 6.4 (*TRM refinement*). Given a $TRM = \langle FM = (F, \phi), RM_s, RM_t, \mu_s, \mu_t\rangle$, a *TRM refinement* TRM' through a refinement step $TRM \overset{ref}{\Longrightarrow} TRM'$, where $ref = \langle ref_s\colon RM_s \rightarrow RM'_s, ref_t\colon RM_t \rightarrow RM'_t\rangle$ is a tuple of strong DRM refinements, is built in one of the following two ways:

(1) *free TRM refinement*: $TRM' = \langle FM, RM'_s, RM'_t, \mu_s, \mu_t\rangle$, where $ref_s\colon RM_s \rightarrow RM'_s$ and $ref_t\colon RM_t \rightarrow RM'_t$ are two DRM refinements obtained by applying to RM_s and RM_t strong refinement operations that are not in conflict with those in μ_s and μ_t.
(2) *configuration-based TRM refinement*: given a (possibly partial) configuration $\rho \in P$, we build $TRM' = \langle FM' = (F, \phi \wedge \bigwedge_{f\in\rho} f), RM'_s, RM'_t, \mu'_s, \mu'_t\rangle$, where RM'_s is obtained by applying the operations associated to each feature $f_0, ..., f_n \in \rho$ in sequence: $RM_s \overset{\mu_s(f_0)}{\Longrightarrow} ... \overset{\mu_s(f_n)}{\Longrightarrow} RM'_s$,

and ref_s is built by concatenating the DRM strong refinements produced by these operations. RM'_t and ref_t are obtained analogously. $\mu'_s(f)$ is the identity refinement for $f \in \rho$, and $\mu_s(f)$ otherwise. $\mu'_t(f)$ is defined analogously.

We write $TRM \overset{ref^*}{\Longrightarrow} TRM'$ to denote zero or more refinement steps.

While configuration-based TRM refinements perform the coupled evolution of DRMs in order to deal with dependent open choices, free TRM refinements permit refining each DRM independently from the other. A configuration-based refinement not only modifies the DRMs, but in addition, it modifies the formula of the feature model by conjoining the name of the features in the configuration. This forces the selection of those features in any valid configuration of the refined TRM. Moreover, those features get assigned the identity refinement in place of the originally associated operations, as they have just been applied.

Note that we model the selection of features by a (partial) configuration ρ as the conjunction of each feature in the set with the formula ($\phi \wedge \bigwedge_{f \in \rho} f$). This is enough for the type of feature models we produce (a tree of alternative choices, where exactly one choice is needed per open type). For more general feature models, we should also conjoin the negation of the features not selected by the configuration. For clarity of presentation, we have opted for the simpler formulation.

Example. Figure 23 shows two TRM refinements for excerpts of the source and target DRMs of the running example (Figure 7 contains the complete DRMs). First, the user has selected the configuration ⟨ MethodDefinition, Row⟩ in the feature model, and hence, its associated refinement operations are triggered. These assign a reference type to features methods and rows in the source and target DRMs, the former with MethodDefinition as target, and the latter targeting class Row. Since this is a configuration-based refinement, the formula of the feature model is conjoined $MethodDefinition \wedge Row$, and its features MethodDefinition and Row get assigned the identity refinement (represented as having no associated operation). The figure shows a second refinement, which is free, where the user has refined the source DRM by making its classes concrete and assigning a concrete type to attribute name.

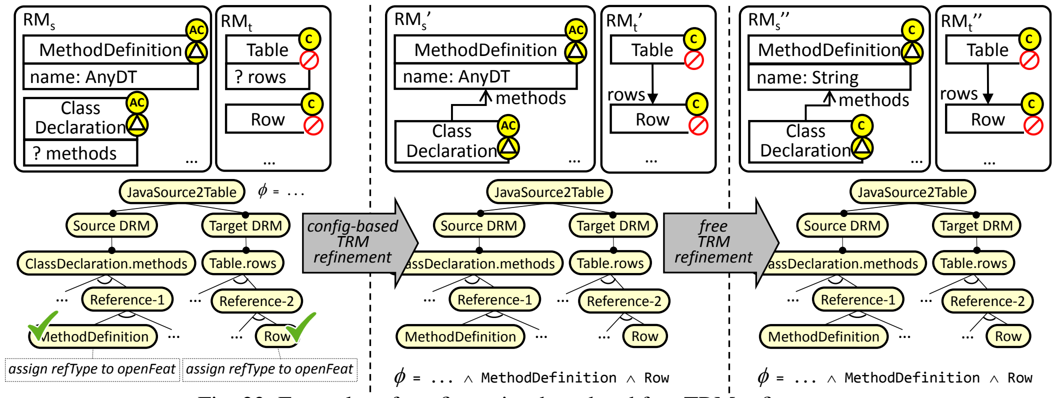


Fig. 23: Examples of configuration-based and free TRM refinements.

We say that a TRM is *ground* if no configuration-based refinement step is possible because all features in its feature model have been selected. Since Definition 6.4 models feature selection by conjoining the feature to the formula, a TRM is ground if $\nexists f \in F \bullet SAT(\phi \wedge f) \wedge SAT(\phi \wedge \neg f)$, with SAT a predicate that holds if the formula is satisfiable. This means that there is no feature left for which we can either select it or not, and hence the selected configuration is total.

A TRM defines a language, which consists of all TRMs that can be obtained from it by zero or more TRM refinement steps.

Definition 6.5 (*TRM language*). Given a transformation requirements model TRM, the language generated by TRM is given by $L(TRM) = \{TRM' \mid TRM \overset{ref}{\Longrightarrow}^{*} TRM'\}$.

The meta-model querying scenario presented in Section 3.1 and Figure 3 requires the ability to determine whether two meta-models (or more generally two DRMs) conform to the TRM extracted from a transformation. For this purpose, next we define the notion of conformance between TRMs, which states that a TRM conforms to another if the former TRM belongs to the language of the latter.

Definition 6.6 (*TRM conformance*). Given TRMs TRM^1 and TRM^2, we say that TRM^1 conforms to TRM^2 iif $TRM^1 \in L(TRM^2)$.

Assessing if an arbitrary DRM pair $\langle DRM_s, DRM_t\rangle$ conforms to a given TRM TRM (as the scenario in Section 3.1 demands) amounts to checking whether there is a ground $TRM' \in L(TRM)$ and a strong refinement pair $\langle ref_s\colon RM'_s \to DRM_s, ref_t\colon RM'_t \to DRM_t\rangle$. For technical reasons, we cannot check conformity by building a TRM $TRM'' = \langle FM = (\emptyset, true), DRM_s, DRM_t, \{\}, \{\}\rangle$ out of the DRM pair, and then assessing $TRM'' \in L(TRM)$, because in general the feature models of TRM and TRM'' may not be related as required by Definition 6.4.

7. EXTRACTING TYPING REQUIREMENTS MODELS FROM ATL TRANSFORMATIONS

So far, we have presented our method to define and use TRMs independently from any transformation language. This section explains the procedure for extracting TRMs out of existing ATL transformations, hence demonstrating the applicability of the method in practice (Section 7.1). Then, in Section 7.2, we discuss how to generalize the extraction to other transformation languages.

7.1. Extracting TRMs from ATL

To describe the extraction of TRMs from ATL transformations, we rely on the Attribute Grammar formalism, which is an elegant and powerful mechanism to describe computations over syntax trees [Slonneger and Kurtz 1995]. 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 tree 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 pass from a node to its parent by means of *synthesized attributes* (SAs), or from a node to its children by means of *inherited attributes* (IAs). Attribute values can also be assigned, modified, and checked at any node in the syntax tree.

Viewing an ATL transformation as a parse tree, AG can pass values from a node to its parent using a SA, or from the current node to a child using an IA. In addition to passing attribute values up or down the parse tree, they can be assigned, updated, and checked at any node in the derivation tree.

Table I shows a fragment of the ATL attribute grammar (AG_{ATL}) we have developed to create TRMs while traversing the syntax tree of an ATL transformation. It is important to remark that we show a simplification of the real grammar to give a flavour of how the proposed extraction mechanism works, without compromising the readability of the explanation. Anyhow, the developed tool available online[6] considers all the productions defined for the actual AG_{ATL}, which implements all the concepts presented in Sections 4 and 6.

For each production rule in Table I, we add attributes whose value is set using values of the parent or children nodes. The table shows some productions and their associated attribute computation rules. The computations infer the value of the attribute *type* for the parsed elements and update the

[6]http://github.com/MDEGroup/totem

#	**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⟩;	*leftFeature* ← *createFeature*(*name*(ID), ⟨binding⟩.parent) *rightFeature* ← *type*(⟨oclExpression⟩) *type*(⟨binding⟩) ← *addClassToTargetDRM*(*leftFeature*.parent) *addCompatibilityAlternatives*(*leftFeature*, *rightFeature*)
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*⟩))

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

TRM (DRMs and compatibility model) accordingly. The attribute *type* behaves both as inherited and synthesized, thus it is initialized during a top-down phase, and updated during a subsequent bottom-up phase. When the parsing is complete, a pair of source and target DRMs together with a compatibility model linking them is produced.

For explanatory purposes, Figure 24 shows a sketch of a graph transformation specification [Ehrig et al. 2006] describing the productions in Table I. Rules show objects of the parse tree of the ATL transformation with coloured background, and elements of the TRM in white. Elements created by the rules are tagged as new, and modified attributes are tagged as modified. The type attribute is represented as a link between the parse tree objects and the TRM elements. Overall, rule p7 creates classes, rules p3 and p5 insert such classes in either the source or the target DRM, p6 creates features given a binding, and p9 creates features given a NavigationOrAttributeCallExpression.

Figure 25 shows a fragment of the AG_{ATL} parse tree corresponding to the rule Table of the transformation in Figure 1. Each node of the tree is decorated with the corresponding attribute computation rules according to the grammar in Table I. The Figure also shows the effects of execution of the computations (createClass, addClassToSourceDRM, addClassToTargetDRM) on the source (RM_s) and target (RM_t) DRMs being built.

The computation rules make use of the following auxiliary functions, which create and update elements (conforming to the DRM and compatibility meta-models shown in Figures 6 and 22) 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 DRM where the created class should be included is 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* to the source and target DRM, respectively. The type of the class is inferred from the non-terminal ⟨oclModelElement⟩, and is added to the source or target DRM depending on whether the non-terminal appears in an ⟨inPattern⟩ or an ⟨outPattern⟩. Accordingly, the production *p3* uses *addClassToSourceDRM* to manage the non-terminal ⟨InPatternElement⟩, like JavaSource!ClassDeclaration, and *p5* uses *addClassToTargetDRM* to manage ⟨OutPatternElement⟩, like Table!Table. In both cases, the

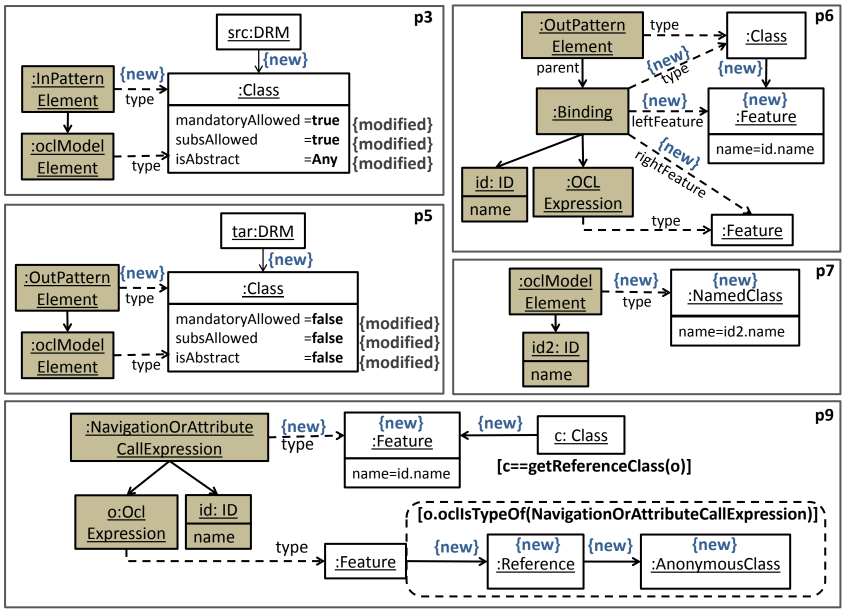


Fig. 24: Sketch of graph transformation rules representing the behaviour of the attribute grammar in Table I.

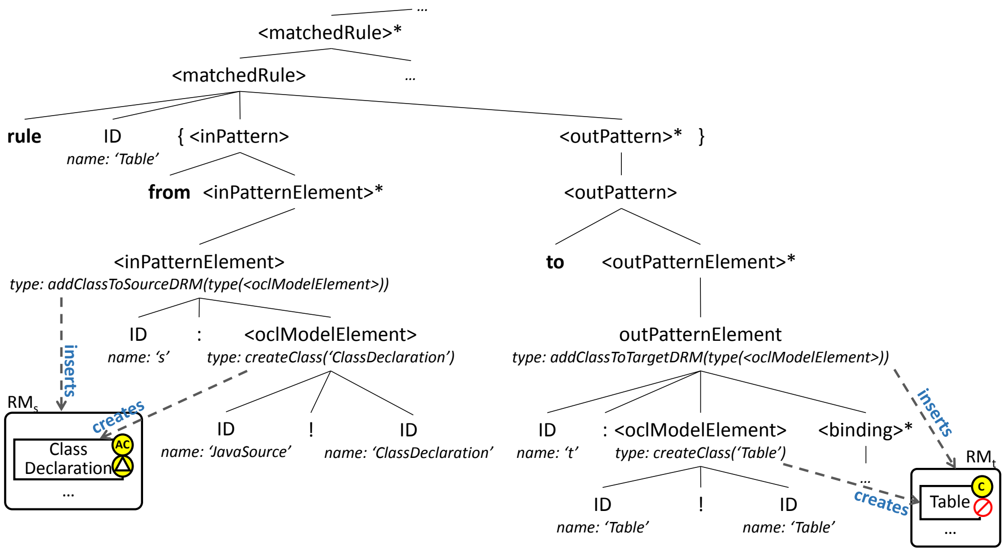


Fig. 25: A sample AG_{ATL} parse tree.

classes previously created by the function *createClass* (e.g., ClassDeclaration and Table) are added to the corresponding DRM. Attributes mandatoryAllowed and subsAllowed are $true$ for classes added to the source DRM, and $false$ for classes in the target DRM. The attribute isAbstract is Any for classes in the source DRM, and $false$ otherwise. The antiancs relation is set between any two named classes of the source DRM created by the production *p2* from input patterns with 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.

▷ *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 that 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 in the input class *c*. It is used in the productions *p6* and *p9*. The former production manages the non-terminal ⟨binding⟩, like rows ← s.methods at line 12 in Figure 1. The latter production handles the non-terminal ⟨NavigationOrAttributeCallExp⟩, like i.method.name at line 7. In the case at line 12, a new feature named rows is added in the target DRM. Its possible types are inferred from the type of the OCL expression s.methods, which is handled by operation *addCompatibilityAlternatives*. In the case at line 7, *p9* matches i.method with ⟨oclExpression⟩ and name with ID. Since name is not an operator, a new feature named name is created in the class referred by i.method. Concerning the cardinality of the created feature, when a Number element is created, its attribute allowMore is $true$ if it is a min cardinality of the source DRM or a max cardinality of the target DRM, while its attribute allowLess is $true$ if it is a max cardinality of the source DRM or a min 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 is Many.

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

▷ *addCompatibilityAlternatives(left: Feature, right: Feature)*: it is used in the production *p6* for building the compatibility model, assigning refinements to compatible types to left and right features of a binding. It does a case analysis between the left and right features, checking compatibility issues like cardinality consistency or types of resolving rules. Then, it creates the corresponding choices in the compatibility model. The compatibility model conforms to the meta-model shown in Figure 22, which allows defining several alternative configurations for the source and target DRMs. For instance, as the compatibility model in Figure 21 shows, the alternatives for feature ClassDeclaration.methods in the source DRM are either being an attribute or being a reference with MethodDefinition or ClassDeclaration as target.

7.2. Extracting TRMs from other transformation languages

The procedure just described targets ATL, one of the most used transformation languages nowadays. However, other transformation languages exist, like the Epsilon Transformation Language (ETL) [Kolovos et al. 2008] or languages of the QVT family [Object Management Group 2016].

ETL is a language similar to ATL. It is rule-based and supports imperative constructs. ETL rules are similar to ATL rules, but their source pattern can only contain one element, and several rules can be applied to the same object. There is no implicit binding resolution, but it must be explicitly invoked using the equivalent() operation or the special assignment ::=. Any model involved in an ETL transformation can be read or written. These features imply the following considerations in the extraction of the TRM:

— *Imperative creation of objects*. ETL can include imperative constructs to create objects in the source model. Hence, the corresponding class in the DRM need to set the flags mandatoryAllowed, subsAllowed and isAbstract to $false$.
— *Antiancs relation*. ETL permits applying more than one rule on the same object. This means that no antiancs relations have to be created.
— *Reference compatibility*. Bindings can make use of the operator ::= or the operation equivalent() to compute the compatible target classes for the references involved in the binding. This explicit binding resolution is essentially the same as for ATL, but taking into account that source objects can be transformed by several rules.
— *Read and written references*. A reference can be both read and written by an ETL program. While our DRMs support either open or closed references, we would need *both* at the same time to consider this case. Our approach does not support this currently, but for refinement, it would just imply preservation of the target classes (i.e., neither adding nor deleting target classes would be refinements).
— *Cardinality of read and written features*. A feature can be both read and written by an ETL program. If the feature has a Number cardinality, then neither allowMore nor allowLess can be true.

Overall, on the one hand, the possibility of having both read and written features restricts the possibilities to reuse (i.e., of finding refinements) because the cardinalities and the reference semantics are more restricted. On the other, the fact that no antiancs relations are created enhances the reuse possibilities using our approach.

Other languages, like QVT relational, can provide more detailed typing information. For example, relations in QVT can declare typed variables, which then can be assigned to object features; and there is no binding resolution neither implicit nor explicit, but explicit parameter passing between relations. Both characteristics facilitate type inference. We leave further investigation about the generalization of our approach to other languages to future work.

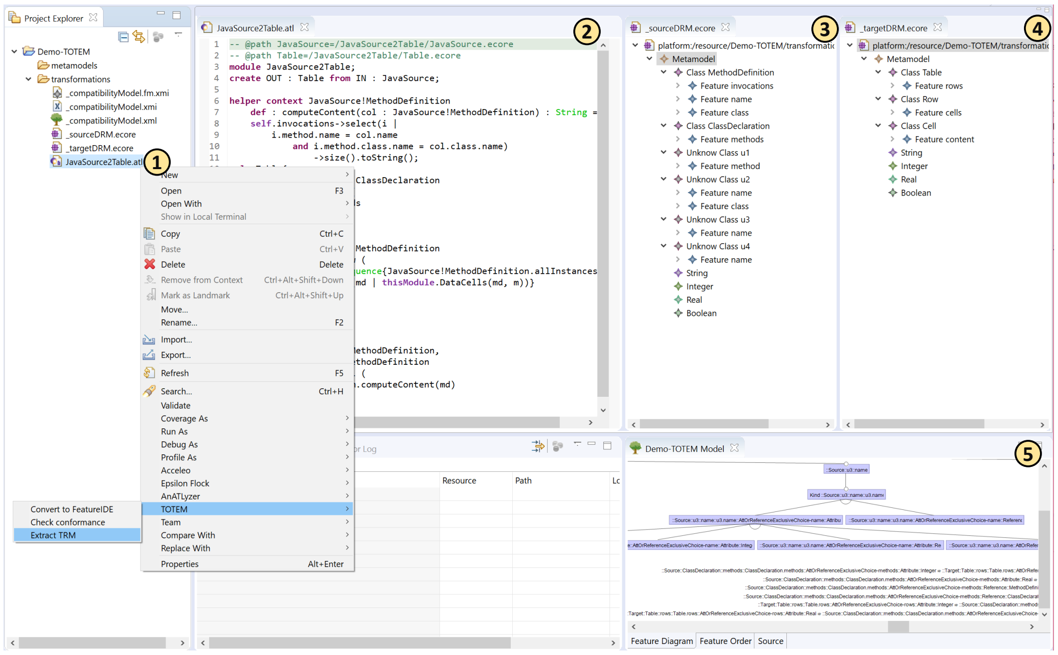
8. TOOL SUPPORT

The presented approach is fully supported by an Eclipse-based tool called TOTEM. The tool is freely available at http://github.com/MDEGroup/totem, including the source code, a screencast showing the tool at work, and the results of the evaluation presented in the next section. In the following, we provide details of the supported functionality in Section 8.1, and describe the tool architecture in Section 8.2.

8.1. Tool functionality

TOTEM offers support for the following functionalities:

— *Extraction of TRMs from ATL transformations.* Developers having an ATL transformation can use TOTEM to automatically extract its TRM as explained in Section 7. For this purpose, they have to select the ATL transformation of interest and trigger the action *"TOTEM → Extract TRM"*. In the example shown in Figure 26, the developer has selected the transformation JavaSource2Table.atl (labels 1 and 2). Subsequently, the source and target DRMs (panels with labels 3 and 4) and the compatibility model linking them (panel with label 5) are generated. Technology-wise, DRMs are EMF models conformant to the meta-model in Figure 6, and can be visualized either using a tree editor (upper-right in Figure 26) or a user-friendly graphical concrete syntax similar to the one used throughout the paper (Figure 27).
— *Conformance checking of a meta-model to a DRM.* TOTEM permits checking whether a given meta-model refines a DRM. The DRM may have been extracted from a transformation as explained above, or it may have been specified manually. Checking refinement entails inspecting all conditions presented in Section 5, and the outcome of the analysis is reported to the user.
— *Graphical editing and free refinement of DRMs.* TOTEM includes a graphical editor to visualize, specify and modify DRMs (see Figure 27). The editor provides the catalogue of refinement operations presented in Section 5.2.1, via contextual menus. This permits performing free strong



```
-- @path JavaSource=/JavaSource2Table/JavaSource.ecore
-- @path Table=/JavaSource2Table/Table.ecore
module JavaSource2Table;
create OUT : Table from IN : JavaSource;

helper context JavaSource!MethodDefinition
    def : computeContent(col : JavaSource!MethodDefinition) : String =
    self.invocations->select(i |
        i.method.name = col.name
            and i.method.class.name = col.class.name)
                ->size().toString();
```

Fig. 26: Extraction of a TRM from an input ATL transformation.

refinements either on the source or the target DRM of a TRM (cf. Definition 6.4). For this purpose, the developer must select the element to be refined, and the editor filters the refinement operations by showing only those that can be applied to the selected context. For instance, Figure 27 shows the graphical representation of the source DRM extracted from the JavaSource2Table.atl transformation, as well as a contextual menu with the refinements that can be applied on the selected anonymous class u3.

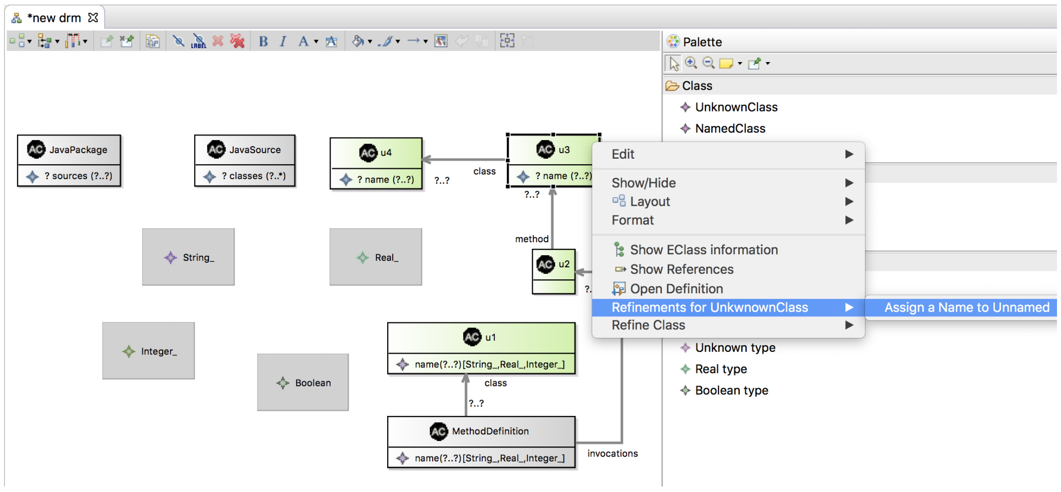


Fig. 27: Visualization and refinement of the input DRM using TOTEM.

— *Visualization of compatibility models as feature models, and configuration-based TRM refinement.* TOTEM supports the visualization of compatibility models in terms of feature models, as panel 5 of Figure 26 illustrates. In addition, users can select feature configurations in the compatibility model, and consequently, perform the associated configuration-based TRM refinements as described in Definition 6.4. In this way, both the source and target DRMs of a TRM can be synchronously refined in a consistent way.

8.2. Tool architecture

TOTEM is an Eclipse plugin, and Figure 28 shows its component-based, layered architecture. The existing components that TOTEM relies on are depicted in white, whereas those that we have developed are depicted in grey. In particular, the graphical *DRM editor* enabling the visualization and editing of DRMs is based on the Eclipse Sirius project [Sirius 2018]. The editor permits interacting with the *TRM extractor* and *conformance checker* components, which have been implemented atop the Eclipse Modeling Framework (EMF) [Steinberg et al. 2008]. The *conformance checker* also relies on some languages of the Epsilon family (in particular, on the Epsilon Transformation Language
(E o
ge h
re n
A

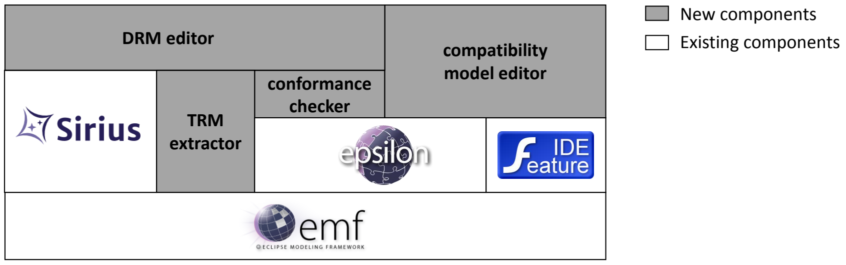


Fig. 28: Layered architecture of TOTEM.

9. EVALUATION

In this section, we evaluate our approach with the aim of answering the research questions **RQ1** (*"Is the TRM extraction mechanism from ATL transformations correct and complete?"*) and **RQ2** (*"To what extent does the variability encoded in the extracted TRMs enable transformation reuse?"*) introduced in Section 1.

To answer **RQ1**, a formal proof of correctness and completeness of the TRM extraction method would be desirable; however, ATL is an unformalised language. Even though some efforts exist to express the *execution* semantics of ATL by compilation into Maude [Troya and Vallecillo 2011], formal typing rules for ATL, including OCL, are unavailable. Therefore, we opt for an empirical evaluation based on seven transformations developed by third parties. 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.). Other aspects of the proposal, like the correctness of the refinement catalogue (Section 5.2.1), the composability of strong refinements (Theorem 5.6), and the correctness of model retyping through refinement relations (Theorem 5.12), are proved formally in the appendix.

In the following, we describe the common evaluation setup in Section 9.1, report on the evaluation of the research questions in Sections 9.2 and 9.3, and discuss threats to validity in Section 9.4. The materials used for the evaluation, including the used transformations, extracted TRMs, mutants and raw data, are available at http://miso.es/trms/eval.html.

9.1. Evaluation setup

Our evaluations consider the following seven ATL transformations: 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), Ant2Maven (a mapping between tasks of different build systems), Class2Relational (a simple object-relational mapping), HSM2FSM (a flattening of hierarchical state machines), and UML2Intalio (a transformation from UML Activity Diagrams to Intalio BPMN).

The selection criterion was to choose transformations written by a third-party (except UML2Intalio), with no typing errors (or very easily fixable ones) to avoid introducing a bias. The first five transformations are publicly available in the ATL Zoo, HSM2FSM is used as a case study in [Cheng et al. 2018] to validate a verification method for model transformations, and UML2Intalio is used in [Sánchez Cuadrado et al. 2018] to assess the validity and completeness of a catalogue of quick fixes for ATL. These transformations cover 71% of the ATL constructs, measured by extracting the footprint of the transformation definitions with respect to the underlying ATL meta-model. The main missing features are action blocks (the imperative part of ATL), called rules, tuples and maps.

Table II summarizes the main features of the transformations used in the evaluation. We have used the number of nodes in the abstract syntax tree as a measure of the complexity of bindings, filters, and helpers. For example, an expression like self.classes→ isEmpty() counts as two nodes.

	JavaSource2Table	HSM2FSM	PetriNet2PNML	KM32EMF	Ant2Maven	Class2Relational	UML2Intalio
Classifiers (src/tar)	6/4	6/6	9/13	15/20	48/59	6/5	248/20
Attributes (src/tar)	1/1	3/3	3/4	10/33	93/98	3/1	106/14
References (src/tar)	7/3	6/8	12/14	17/48	28/35	6/5	481/31
ATL matched rules	3	7	10	10	30	7	9
ATL lazy rules	2	0	0	0	0	1	0
ATL rule filters	1	5	2	2	2	5	5
Bindings	8	18	29	40	98	20	14
ATL context helpers	1	0	0	1	0	2	6
ATL global helpers	1	0	0	0	0	1	0
Lines of code	68	81	96	128	269	101	98
Avg binding complexity	6.38	1.78	29	2.28	2.04	4.95	2.64
Avg filter complexity	4.00	11.20	0	9.50	9.50	6.60	11.20
Avg helper complexity	17.50	0	0	24.00	0.00	5.00	8.50

Table II: Testbed transformations in the evaluation and their characteristics.

9.2. RQ1: Correctness and completeness of the ATL extraction mechanism

To answer **RQ1**, we consider the seven transformations abovementioned, together with their source and target meta-models. First, we use TOTEM to extract the TRM of each transformation (i.e., source and target DRMs and compatibility model). Then, we generate first-order mutants of the source and target meta-models by systematically applying the meta-model modifications identified in [Cicchetti et al. 2008] (cf. Table III). Our aim is to generate many slightly different variants of the original meta-models, so that some break the transformation, while others do not. Finally, we assess whether a meta-model mutant is a refinement of the extracted TRM when the transformation can use it safely, and it is not a refinement otherwise. To determine if a meta-model mutant can be safely used with a transformation, we use the ANATLYZER [Sánchez Cuadrado et al. 2017] ATL static type checker as an oracle of the typing relation between the mutated meta-model and the transformation. If ANATLYZER does not report any errors, then it means that the mutant does not break the transformation, and it should be a refinement of the extracted TRM. Otherwise, the mutant breaks the transformation, and it should not be a refinement of the TRM.

Altogether, for each meta-model mutant, we may obtain one of the following results: *i)* the mutant refines the TRM and does not break the transformation (*true positive, TP*); *ii)* the mutant refines the TRM but breaks the transformation (*false positive, FP*); *iii)* the mutant does not refine the TRM and breaks the transformation (*true negative, TN*); or *iv)* the mutant does not refine the TRM but does not break the transformation (*false negative, FN*). Then, we compute *precision* (an indicator of correctness) as $\frac{\#TP}{\#TP+\#FP}$, and *recall* (an indicator of completeness) as $\frac{\#TP}{\#TP+\#FN}$.

Table IV summarizes the obtained results. There are no false negatives, and thus recall is 100%, signifying that the extracted TRMs correctly exclude the meta-models that cannot be used with the transformations (i.e., such meta-models are not refinements of the TRM). There are some false positives though, meaning that some meta-models are refinements of the extracted TRM, but the transformation may raise runtime errors if executed with them. Nevertheless, the overall precision is

Kind	**Meta-model modification**
Additive	Add obligatory / non-obligatory metaclass
	Add obligatory / non-obligatory metaproperty
	Generalize metaproperty
	Pull metaproperty
	Extract abstract / non-abstract superclass
Subtractive	Eliminate metaclass
	Eliminate metaproperty
	Push metaproperty
	Flatten hierarchy
	Restrict metaproperty
Updative	Rename metaelement
	Move metaproperty
	Extract/inline metaclass

Table III: Meta-model modification operators used in the evaluation.

still high (97%). An example of false positive occurs in the expression i.method.name of the running example (line 7). In the original meta-model, the attribute name is compulsory, but one meta-model mutant relaxes its cardinality to 0..1. The extracted TRM does not put any cardinality restriction to this attribute, however ANATLYZER signals this typing problem, and thus it is reported as a false positive. We have observed that the reported false positives are due to limitations in the TRM extraction process, though they are not a shortcoming of the general method itself. 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. We will discuss in more detail some limitations of the TRM extraction method in Section 9.2.1.

	JavaSource2Table	HSM2FSM	PetriNet2PNML	KM32EMF	Ant2Maven	Class2Relational	UML2Intalio	Total
Mutants	144	314	305	2,480	18,535	256	4,753	26,787
True positives	70	148	169	1,751	6,986	103	4,437	13,664
True negatives	66	154	131	690	11,254	139	270	12,704
False positives	8	12	5	39	295	14	46	419
False negatives	0	0	0	0	0	0	0	0
Precision	90%	93%	97%	98%	96%	88%	99%	97%
Recall	100%	100%	100%	100%	100%	100%	100%	100%
Reusable MMs	70	148	169	1,751	6,986	103	4,437	13,664
Non-reusable MMs	74	166	136	729	11,549	153	316	13,123
Incompatible	4	11	18	6	400	4	13	456

Table IV: Evaluation results.

To analyse whether the evaluation thoroughly tested all aspects of our refinement relation, we have manually revised the extracted TRMs and some mutants. We found several interesting cases. For instance, PetriNet2PNML exercised the subsAllowed flag (illustrated in Figure 14 for the running example), since some features of an abstract class Arc were located in all subclasses. Meta-model modifications like *pull metaproperty*, *push metaproperty*, *inline metaclass* and *flatten hierarchy* generate mutants which require structural typing in the refinement. All these cases were correctly handled by our refinement checking mechanism.

To analyse the effect of the mutations, the second and third last rows of Table IV show the number of meta-models for which the transformation is reusable, or it is not. In this way, the row "reusable MMs" is calculated as *TP+FN*, while "non-reusable MMs" is *TN+FP*. Notably, there is a high number of meta-models – different from the ones used to develop the transformations – for which the transformations are reusable (13,664) and our method identifies all of them. The transformations are not reusable with more than 13,000 meta-models, and our method identifies 97% of these cases. Non-reusability can be discovered either because there is no DRM refinement, or because the conditions implied by the compatibility model fail. The last row of the table details how many meta-models individually refine the DRMs but do not satisfy the compatibility model (456 in total). This shows the usefulness of the compatibility model to increase the precision of the refinement relation by ruling out configurations which would lead to ill-behaved transformations.

9.2.1. Limitations of the TRM extraction algorithm. The evaluation shows that the extraction algorithm is complete (no false negatives), in the sense that the extracted TRMs gather all the information needed to guarantee that if a meta-model does not refine the TRM, then it will break the transformation. This is because the extraction algorithm over-approximates the actual meta-model used by the transformation (i.e., the typing is as flexible as possible). However, this is at the cost of some precision loss, as in a few cases, there may be meta-models which break a transformation even if they are valid refinements of its TRM. In the evaluation, this happened in 3% of cases. Most of these false positives could be solved by using information of the meta-models (if available) during the TRM extraction process. We leave this improvement for future work and, in the following, discuss the most relevant limitations found in the Class2Relational case study. Figure 29 shows excerpts of this transformation and its source meta-model.

```
 1 module Class2Relational;
 2 create OUT : Relational from IN : Class;
 3
 4 helper context Class!NamedElt def: nameOrEmpty : String =
 5   if self.name.oclIsUndefined() then ''
 6   else self.name endif;
 7
 8 helper context Class!Attribute def: multiValuedOrFals
 9   if self.multiValued.oclIsUndefined() then false
10   else self.multiValued endif;
11
12 rule Class2Table {
13   from c : Class!Class ( not c.isAbstract )
14   to out : Relational!Table (
15     name <- c.name,
16     col <- Sequence {key}->union(
17            c.atts->select(e | not e.multiValuedOrFals
18     ...
19     ), key : Relational!Column ( ... )
20 }
21
22 rule MultiValuedClassAttribute2Column {
23   from a : Class!Attribute (
24     a.type.oclIsKindOf(Class!Class) and a.multiValued
25     to t : Relational!Table (
26       name <- a.owner.nameOrEmpty + '_' + a.name,
27       ...
28     )
29 }
```

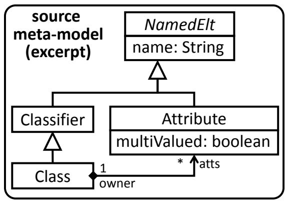


Fig. 29: Excerpts of the Class2Relational transformation (left) and its source meta-model (right).

A first limitation is due to the fact that ATL can emulate meta-model attributes by transformation helpers (e.g., nameOrEmpty in line 4, multiValuedOrFalse in line 8). Since ATL helpers are not added to the DRMs, some problems may arise. As an example, Figure 30 shows an excerpt of the source DRM of the transformation, and two meta-models MM1 and MM2 that refine the DRM but cannot be used safely with the transformation. MM1 is similar to the source meta-model, but changing the name of class NamedElt to BaseElt. MM1 refines the DRM because NamedElt in the DRM matches all subclasses of BaseElt in MM1; however, using the transformation with MM1 yields an error because line 4 declares a helper over class NamedElt, which does not exist in MM1. Similarly, MM2, where the type of Class.atts is NamedElt, is a refinement of the DRM; however, using the transformation with MM2 leads to an error in line 17 because the helper multiValuedOrFalse is invoked on NamedElt objects, but the helper is defined over class Attribute. The extraction algorithm could avoid these problems being more conservative. For example, it could detect these scenarios in order to set the flag subsAllowed of NamedElt in the DRM to false. However, this would lead to a false negative illustrated in MM3, where the attribute name has been moved from NamedElt to its subclasses. If NamedElt had subsAllowed set to false, no class in MM3 would refine it (i.e., structural typing would not apply), and MM3 would not be

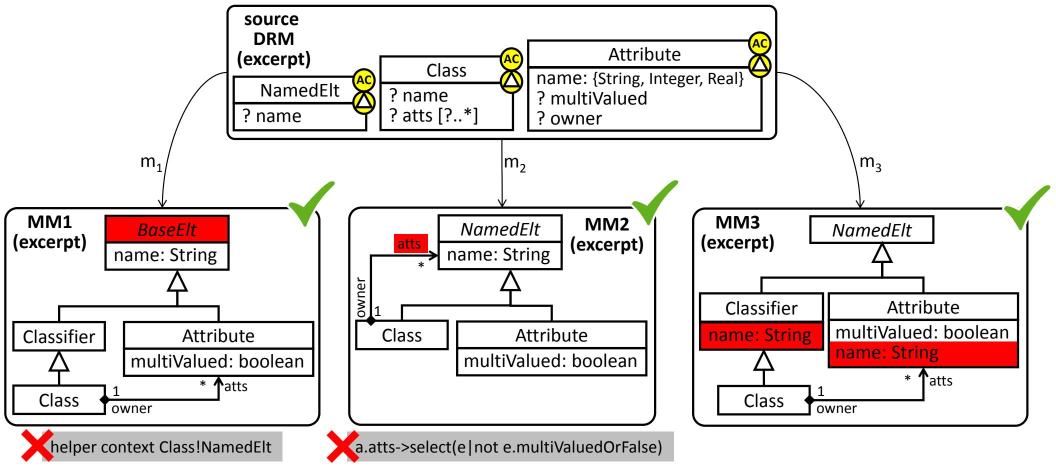


Fig. 30: Examples of false positives in the Class2Relational transformation.

As previously mentioned, a second limitation of the TRM extraction mechanism concerns the precision of cardinalities. For example, the target DRM includes a class Table with a feature keys of unknown cardinality. A meta-model containing Table.keys with cardinality 1..* would be a refinement of this DRM, but would make rule MultiValuedDataTypeAttribute2Column incorrect as it creates a Table with no key (mandatory in the meta-model).

In summary, the TRM extraction algorithm is complete according to our experiments (100% recall), and has high precision (97% precision) up to some limitations regarding cardinalities and ATL helpers. Improving precision, e.g., using meta-model information, is up to future work.

9.3. RQ2: Variability support in TRMs

A TRM gathers requirements that meta-models need to fulfil to be able to use them safely with a given transformation. To reuse a transformation with as many meta-models as possible, its TRM needs to represent the allowed meta-model variability. In this experiment, we look at the power of TRMs to encode this variability. For this purpose, we analysed the TRMs of the seven transformations, measuring the elements helping in expressing variability. For the DRMs, we measured the ratio of anonymous vs. named classes, and the ratio of untyped features (which can become either attributes or references) vs. typed fields (actual attributes or references). In the compatibility models, we computed the size of the feature model (both total number of features and leaf features), and the number of valid configurations. Table V summarizes the results.

Transformation	Source DRM		Target DRM		Compatibility model		
	Anonymous/ Total classes	Untyped fields/ Total fields	Anonymous/ Total classes	Untyped fields/ Total fields	Total Features	Leaf Features	Configs
JavaSource2Table	5/8 [62.5%]	5/10 [50%]	0/4 [0%]	2/4 [50%]	174	99	$>$ 1,402,192
HSM2FSM	0/6 [0%]	6/10 [60%]	0/4 [0%]	8/8 [100%]	307	192	$>$ 325,647
PetriNet2PNML	12/16 [75%]	12/12 [100%]	0/8 [0%]	21/22 [95%]	637	410	$>$ 953,959
KM32EMF	1/11 [9%]	31/32 [97%]	0/10 [0%]	32/34 [94%]	1,576	1,125	$>$ 666,127
Ant2Maven	1/32 [3%]	79/80 [99%]	0/32 [0%]	83/85 [98%]	7,447	6,304	$>$ 133,701
Class2Relational	0/5 [0%]	9/10 [90%]	0/4 [0%]	6/8 [75%]	266	157	$>$ 1,610,020
UML2Intalio	4/15 [27%]	10/13 [77%]	0/6 [0%]	5/8 [62.5%]	306	187	$>$ 1,086,476
Average	23/93 [24.7%]	152/167[91%]	0/68 [0%]	157/169 [93%]	1,530.4	1,210.6	$>$ 355,787

Table V: Indicators of the variability in the TRMs.

First, we notice that anonymous classes only appear in source DRMs. This is expected as the target domain is write-only and ATL lacks navigation expressions for the target domain. Most transformations (5 out of 7) have source DRMs with anonymous classes, and overall around 25% of classes are anonymous. Anonymous classes are common in source DRMs because most transformations contain some navigation expression in the rule filters, helpers or bindings. Anonymous classes help in encoding variability as they can be matched to classes with an arbitrary name. If we compare with the original source meta-models (see Table VI), we can see that DRMs enhance flexibility because they reduce significantly the number of named classes. This reduction depends on the meta-model percentage that a transformation uses. An extreme case is UML2Intalio, which only requires a small fragment of UML (the part of activity diagrams) and so the resulting DRM has 11 named classes, while the meta-model has 248 classes. Typically, source DRMs also achieve a reduced number of named classes compared to meta-model footprint techniques, which derive a meta-model slice containing only those elements touched by the transformation [Sen et al. 2009; Burgueño et al. 2015; Jeanneret et al. 2011]. The reason is that DRMs do not enforce class hierarchies but just express structural requirements, sometimes using anonymous classes; while sliced meta-models may contain intermediate classes in inheritance hierarchies. Still, meta-model footprints are normally used in combination with other reuse techniques, e.g., based on concepts [de Lara and Guerra 2011; Sánchez Cuadrado et al. 2014a] or model types [Steel and Jézéquel 2007; Guy et al. 2012]. Anyhow, as Table VI shows, such footprints are typically less flexible than DRMs regarding classes.

Transformation	Named classes in DRM	Classes in Meta-model	Reduction DRMs vs. MMs	Classes in Footprint	Reduction DRMs vs. Footprints
JavaSource2Table	3	6	50%	6	50%
HSM2FSM	6	6	0%	6	0%
PetriNet2PNML	4	9	55,5%	7	42,8%
KM32EMF	10	15	33,3 %	14	28,6%
Ant2Maven	31	48	35,4%	45	31,1%
Class2Relational	5	6	16,7%	6	16,7%
UML2Intalio	11	248	95,6%	32	65,6 %

Table VI: Named classes in DRMs, meta-models and footprints for the source domain, and reduction ratio of named classes when using DRMs.

Table V also shows the ratio of fields that have a type vs. those that do not have it. Fields with no type can be matched to both attributes and references in a meta-model, thus helping to express variability. Most fields in the source and target DRMs are untyped, providing great reuse opportunities. In contrast, the type of all fields in meta-models and footprints is fixed (i.e., it is a particular reference type or data type). The compatibility model ensures that the variability in the types that source and target features may take is coherent. Table V shows that the feature models generally admit a large number of configurations (calculated using FeatureIDE's best approximation with 1 hour timeout). The number of features in the model is typically correlated with the number of untyped fields. The substantial number of configurations shows that TRMs encode very high variability.

Overall, we can conclude that TRMs extracted from ATL transformations encode a large variability, which ensures a flexible reuse. This can be stated of individual DRMs compared to meta-models and footprints, and of TRMs including their compatibility model.

9.4. Threats to validity

In this section, potential threats to validity associated with the performed evaluation are discussed. In particular, we distinguish threats among construct, internal, and external validity as follows.

Construct validity. It pertains to any factor that can compromise the validity of inferences that observations or measurement tools actually represent or measure the construct being investigated. One possible threat to construct validity is that the set of meta-model modification operators used to generate the meta-model mutants might be not complete, potentially preventing exercising all features of our refinement relation. To minimise the impact of this threat, we manually assessed that our mutants exercise all features of our refinement checks, like subsAllowed, mandatoryAllowed, and the different semantics of references. Another threat to construct validity can be related to the adoption of ANATLYZER for determining if a meta-model mutant can be safely used with a transformation. Even though this might represent a bias since anATLyzer was created by three co-authors of this paper, it is important to remark that anATLyzer has been used as was presented in the original paper [Sánchez Cuadrado et al. 2017] without operating any specific change or adaptation that might have introduced inferences in the experiments presented in this paper.

Internal validity. Threats to internal validity concern any confounding factor that could influence our results. We have used a relatively low number of transformations. The reason is that our experiment requires the seed transformations to be free of typing errors. Unfortunately, as we showed in our previous work [Sánchez Cuadrado et al. 2017], most freely available third-party ATL transformations contain errors, which prevents their use in our evaluation. Hence, to ascertain to what extent our evaluation is complete, we have measured the coverage of ATL constructs in our setup with respect to the whole language, and we cover a good part of them (around 71% of the constructs). As for the ATL constructs not covered in our experiment, according to [Selim et al. 2017], they are not among the most common in practice (called rules are found in around 19% of transformations of the ATL zoo, and imperative blocks in around 23% of transformations). Besides, the number of generated meta-model mutants is quite high (more than 26,000) which means that our TRM extraction algorithm has been exhaustively tested. Another threat to internal validity is that the evaluation has used anATLyzer as an oracle to well-typedness. Although anATLyzer has been reported to have high precision and recall [Sánchez Cuadrado et al. 2017; 2018], it is not infallible. To avoid distortions on the evaluation results due to possible errors in our oracle, we have manually revised the dubious cases and have not found any incorrect result.

External validity. Threats to external validity refer to the extent to which the results of our study can be generalized. The proposed TRM formalism and the refinement operators have been implemented and exercised on ATL model transformations only. However, in order to ensure the applicability of the approach also to other model transformation languages, we conceived the TRM formalism and the corresponding refinement operators in a technology agnostic manner. The conceived theorems and the proofs have been defined by discarding any technology specific aspects, and the theory implementation has been done at-posteriori.

10. RELATED WORK

Over the last years, several techniques for model transformation reuse have been conceived [Kusel et al. 2015]. These are generally classified as *intra-transformation* or *inter-transformation* reuse approaches. The former techniques support reuse within a single transformation by means of rule inheritance [Wimmer et al. 2012], rules with variability [Strüber et al. 2018], module superimposition [Wagelaar et al. 2010] and internal composition mechanisms like phases [Sánchez Cuadrado and Molina 2009], hooks [Sánchez Cuadrado and Molina 2008], localized transformations [Etien et al. 2015] and unit combinators [Kleppe 2006].

In this paper, we are more interested in inter-transformation reuse, which consists of reusing transformations that were developed for a meta-model, with a different one (i.e., reuse across meta-models). Compared with intra-transformation reuse, inter-transformation techniques aim at a more coarse-grained reuse – typically complete transformations – and require mechanisms to check that the reused transformation can work properly with a different meta-model, and even automatically adapt the transformation to the new context.

One of the distinctive aspects of inter-transformation reuse is the form of the interface for reuse, as this is the artefact that developers need to deal with when confronted with a new reuse [Bruel et al. 2018]. This interface can follow a black-box or a white-box style. In white-box approaches, the reuse interface is the transformation itself, sometimes expressed in a high-level format, e.g., design patterns [Lano and Rahimi 2014; Lano et al. 2014]. This is the case of DelTa [Ergin et al. 2016], a high-level language to express and compose transformation design patterns. These patterns can be adapted for particular meta-models by mapping the pattern elements to meta-model types, and it is possible to generate transformation code from the patterns.

Black-box approaches are more common. In this case, the reuse interface describes the typing requirements that a transformation demands from its source/target meta-models. Then, the reuser needs to specify how particular meta-models satisfy those requirements. In most cases, the typing requirements are expressed in the form of a meta-model that is developed on purpose for each reusable transformation, and that needs to be mapped to the concrete meta-model the transformation is to be reused for [Steel and Jézéquel 2007; Guy et al. 2012; de Lara and Guerra 2011; Sánchez Cuadrado et al. 2014a; de Lara and Guerra 2017; de Lara et al. 2015]. However, a meta-model is a concrete realization of some design concern, and we have seen in this paper that different meta-models may realize a specific concern in different ways (e.g., as an attribute or as a reference). Hence, some researchers have investigated more expressive means to describe typing requirements for transformations, for example, based on logics [Zschaler 2014]. Next, we analyse existing black-box reuse approaches, indicating how our work improves the state of the art.

Inter-transformation approaches that use meta-models as reuse interface include *model types* [Steel and Jézéquel 2007; Guy et al. 2012], *concepts* [de Lara and Guerra 2011; Sánchez Cuadrado et al. 2014a], *a-posteriori typing* [de Lara and Guerra 2017], *facet-oriented modelling* [de Lara et al. 2018] and *multi-level modelling* [de Lara et al. 2015].

Inspired by generic programming [Gregor et al. 2006], *concepts* have been proposed as the reuse interface of transformations [Sánchez Cuadrado et al. 2014a; 2014b]. In this context, concepts are meta-models whose elements (classes, attributes, references) are interpreted as variables that need to be bound to the elements of concrete meta-models. This binding produces an adapted transformation that is directly applicable to the specific meta-models.

Instead of genericity, some approaches profit from polymorphism for transformation reuse. As an example, *model types* [Steel and Jézéquel 2007; Guy et al. 2012] are based on establishing a subtyping relationship or binding between the transformation meta-model and the meta-model where the transformation is to be reused. By means of this subtyping relation, the transformation becomes applicable to the specific meta-model. In the same vein, Boronat [Boronat 2017] proposes a method to discover subtyping (subsumption) relations between two meta-models, considering OCL constraints and supporting structural subtyping.

Multi-level modelling [Atkinson and Kühne 2002; de Lara et al. 2014; de Lara and Guerra 2018] permits modelling using an arbitrary number of meta-levels. This enables the definition of families of meta-modelling languages (e.g., for domain-specific process modelling) which can be successively refined by instantiation. This way, transformations defined for the meta-meta-model of a language family can be reused for any of the languages of the family [de Lara et al. 2015]. Hence, multi-level modelling enables transformation reuse by exploiting the instantiation relation.

Another way to reuse a transformation is by retyping arbitrary models with respect to the transformation meta-models [de Lara and Guerra 2017; de Lara et al. 2018]. Model retyping enables transformation reuse because it permits seeing models as instances of the meta-models a transformation is defined on.

Overall, the common theme of these approaches is the use of a meta-model as reuse interface. Instead, we propose TRMs as reuse interface as they are more expressive than plain meta-models to convey transformation requirements, for two reasons. First, DRMs can express variability. For example, it is possible to indicate that a class needs a certain feature without prematurely stating that it should be an attribute or a reference; leave the cardinality bounds open; specify several possible target types for a reference; or omit the name of classes. Moreover, DRMs make explicit further expectations required for a correct typing, like the possibility for classes to have or not extra mandatory features or the identification of classes that cannot inherit from each other. This expressivity is not possible with a plain meta-model. Second, TRMs include a compatibility model expressing dependencies between open options of source and target meta-models. These dependencies cannot be captured with just two meta-models, and therefore we use a feature model for this purpose.

Different from meta-model based approaches, ours does not require binding the specific meta-models to the transformation reuse interface; instead, a transformation can be reused with any meta-model pair that satisfies the TRM. This brings increased usability to the approach (reuse is automatic), and enables automatic discovery of reusable transformations for given meta-models or TRMs. However, an explicit binding language (e.g., like [Sánchez Cuadrado et al. 2014a]) is useful to bridge heterogeneities between the reuse interface and the meta-models. Therefore, we plan to investigate how to combine our approach with binding languages in future work.

Another issue is the provision of techniques to extract the reuse interface of a transformation. Meta-model based approaches can profit from techniques to obtain the meta-model footprint [Burgueño et al. 2015; Jeanneret et al. 2011]. This is the part of the input and output meta-models accessed by the transformation, which is itself a meta-model. Specific methods have been developed to co-evolve the extracted footprint and the transformation, in order to obtain a simpler, more understandable interface [Sánchez Cuadrado et al. 2015]. While these works rely on the actual transformation meta-models, our technique to extract the TRM does not need the meta-models. This is useful when the meta-models are not available. Moreover, the end result is a TRM, which is more expressive than a meta-model to specify variability and typing requirements.

Instead of using meta-models to express the reuse interface, Zschaler uses logic to express meta-model requirements extracted from in-place transformations [Zschaler 2014]. Similar to our approach, the goal is having more expressive means to capture typing requirements, and being able to check if particular meta-models satisfy those requirements. However, while this can be considered the seminal work in this area, it was not fully elaborated. For example, there are no means to specify that features can be attributes or references, or to require anonymous classes. Instead of logics, we opted for creating a meta-model able to express model-to-model requirements for a given transformation and included a compatibility model because we target model-to-model transformations. Moreover, extracting the requirements from ATL transformations is more challenging as we need to deal with OCL expressions, automated binding resolution, and meta-model dependencies.

Other approaches focus on checking the correctness of transformation reuse. In this regard, transformation *intents* [Lúcio et al. 2016; Salay et al. 2016] describe semantical properties that ensure a correct reuse according to the designer expectations. In our case, we aim at ensuring syntactical correctness, but it would be interesting to incorporate intents into our framework in the future.

Typing model transformations is important to ensure their correct chaining. In [Vignaga et al. 2013], a type system for mega-models assigns to transformations the types of their source and target meta-models. Our TRMs could be integrated into this type system, with the benefit of achieving more abstract types for transformations (the TRMs), facilitating a more flexible transformation chaining. In [Vallecillo and Gogolla 2012], the authors argue on the need for adding *behavioural types* to transformations (abstract descriptions of their behaviour), in addition to structural types. For this purpose, they propose a contract-based approach called Tracts. Our TRMs focus purely on the structural part. It is up to future work to combine TRMs with behavioural types.

Regarding our techniques, we have created our own meta-model to express transformation requirements. Instead, we could have used uncertainty annotations [Famelis and Chechik 2019] over standard meta-models. However, expressing concerns specific to typing requirements, like antiancestor relations between classes, or the fact that a class cannot have further mandatory features, may be difficult to express using generic annotations.

Concerning refinement relations between models, several works define well-formedness rules that can be used to check whether a UML diagram is a valid refinement of another. Their goal is either being able to transition from more abstract views of a system to detailed representations that preserve the semantics, or abstracting concrete models. For instance, [Faitelson and Tyszberowicz 2017] defines a set of refinement rules for class diagrams and use case diagrams. They have a relational semantics and interpret the subtype relation (in class diagrams) and the includes/extends relations (in use case diagrams) as subsetting relations. This branch of works assumes a diagram and its refinement are complete, in the sense that there is no uncertainty or variability on their elements (e.g., a feature must be either an attribute or a reference, this cannot be underspecified). More importantly, their goal is different to ours, as they do not focus on (and their refinement rules do not ensure) safe model transformation reuse.

Overall, our work advances the state of the art by proposing a more expressive means to capture the typing requirements of model-to-model transformations. This is a novel reuse interface for model transformations that enables automated reuse (i.e., does not require a binding) and ensures syntactic reuse correctness.

11. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new approach for model transformation reuse based on TRMs. TRMs are automatically extracted from model transformations and contain a compatibility model constraining the possible open options in the source and target meta-models. As meta-models can be encoded as DRMs, we have provided a theory of TRM refinements, distinguished between different types of refinement, and proposed a catalogue of refinement operators. We have implemented prototype tool support for ATL and presented an evaluation showing good precision and recall of the ATL TRM extraction procedure, and confirming that the extracted TRMs encode high variability that enables flexible reuse.

In the future, we would like to incorporate the notion of *binding* into our refinement relationship to improve reusability. Such bindings may resolve heterogeneities (e.g., class renaming) between the TRMs and the meta-models, and would induce a transformation adaptation, in the style of [Sánchez Cuadrado et al. 2014a]. We also plan to explore heuristics for automatic meta-model generation from TRMs. For this purpose, we may exploit techniques for optimal product selection [Hierons et al. 2016] from the feature model of dependencies specified in the TRM. As our checks are syntactical, we aim at incorporating a notion of *transformation intent* and *behavioural type* into our approach. Another research line is the creation of extraction procedures for other transformation languages, like ETL or QVT. At the technical level, we plan to integrate TOTEM with model transformation repositories [Di Rocco et al. 2016] or general-purpose repositories like GitHub, to allow powerful means to query, discover and chain reusable transformations. Finally, we plan to exploit the notion of DRM as a device to capture meta-model requirements in other applications, like bottom-up and example-based modelling [Kästner et al. 2018; López-Fernández et al. 2015].

Acknowledgements. Work partially funded by the R&D programme of the Madrid Region (project FORTE, S2018/TCS-4314), the Spanish Ministry of Science (project MASSIVE, RTI2018-095255-B-I00), the Spanish MINECO (project RECOM, TIN2015-73968-JIN, AEI/FEDER/UE), a Ramón y Cajal 2017 grant, and the European Union Horizon 2020 research and innovation programme through the Polyglot and Hybrid Persistence Architectures for Big Data Analytics (TYPHON) project (#780251). We thank the anonymous referees for their useful comments, which helped us improving the paper.

REFERENCES

Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12, 4 (2002), 290–321.

Francesco Basciani, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2014. Automated chaining of model transformations with incompatible metamodels. In *MODELS (LNCS)*, Vol. 8767. Springer International Publishing, 602–618.

Artur Boronat. 2017. Structural model subtyping with OCL constraints. In *SLE*. ACM, 194–205.

Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jezequel, Joerg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. 2018. Model transformation reuse across metamodels: A classification and comparison of approaches. In *ICMT (LNCS)*, Vol. 10888. Springer, 92–109.

Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. 2015. Static fault localization in model transformations. *IEEE Trans. Software Eng.* 41, 5 (2015), 490–506.

Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. 2016. Perspectives of model transformation reuse. In *IFM (LNCS)*, Vol. 9681. Springer, 28–44.

Zheng Cheng, Rosemary Monahan, and James F. Power. 2018. Formalised EMFTVM bytecode language for sound verification of model transformations. *Software and Systems Modeling* 17, 4 (2018), 1197–1225.

Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *EDOC*. IEEE Computer Society, 222–231.

Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. 2015. Abstract delta modelling. *Mathematical Structures in Computer Science* 25, 3 (2015), 482–527.

Juan de Lara and Esther Guerra. 2011. From types to type requirements: genericity for model-driven engineering. *Software and Systems Modeling* 12, 3 (2011), 453–474.

Juan de Lara and Esther Guerra. 2017. *A posteriori* typing for model-driven engineering: Concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.* 25, 4 (2017), 31:1–31:60.

Juan de Lara and Esther Guerra. 2018. Refactoring multi-level models. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 17:1–17:56.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46.

Juan de Lara, Esther Guerra, Joerg Kienzle, and Yanis Hattab. 2018. Facet-oriented modelling: Open objects for model-driven engineering. In *SLE*. ACM, 147–159.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. Model-driven engineering with domain-specific metamodelling languages. *Software and Systems Modeling* 14, 1 (2015), 429–459.

Juan de Lara, Juri Di Rocco, Davide Di Ruscio, Esther Guerra, Ludovico Iovino, Alfonso Pierantonio, and Jesús Sánchez Cuadrado. 2017. Reusing model transformations through typing requirements models. In *FASE (LNCS)*, Vol. 10202. Springer, 264–282.

Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, Jesús Sánchez Cuadrado, Juan de Lara, and Esther Guerra. 2016. Using ATL transformation services in the MDEForge collaborative modeling platform. In *ICMT (LNCS)*, Vol. 9765. Springer, 70–78.

Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2012. Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In *ICGT (LNCS)*, Vol. 7562. Springer, 20–37.

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.

Huseyin Ergin, Eugene Syriani, and Jeff Gray. 2016. Design pattern oriented development of model transformations. *Comput. Lang. Syst. Struct.* 46, C (2016), 106–139.

Anne Etien, Alexis Muller, Thomas Legrand, and Richard F. Paige. 2015. Localized model transformations for building large-scale transformations. *Software and Systems Modeling* 14, 3 (2015), 1189–1213.

David Faitelson and Shmuel S. Tyszberowicz. 2017. UML diagram refinement (focusing on class- and use case diagrams). In *ICSE*. IEEE / ACM, 735–745.

Michalis Famelis and Marsha Chechik. 2019. Managing design-time uncertainty. *Software and System Modeling* 18, 2 (2019), 1249–1284.

Douglas P. Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: linguistic support for generic programming in C++. In *OOPSLA*. ACM, 291–310.

Clement Guy, Benoît Combemale, Steven Derrien, Jim Steel, and Jean-Marc Jézéquel. 2012. On model subtyping. In *ECMFA (LNCS)*, Vol. 7349. Springer, 400–415.

Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. 2016. SIP: Optimal product selection from feature models using many-objective evolutionary optimization. *ACM Trans. Softw. Eng. Methodol.* 25, 2 (2016), 17:1–17:39.

Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press. http://alloytools.org/.

Cédric Jeanneret, Martin Glinz, and Benoit Baudry. 2011. Estimating footprints of model operations. In *ICSE*. ACM, 601–610.

Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Science of Computer Programming* 72, 1-2 (2008), 31–39.

Frédéric Jouault and Jean Bézivin. 2006. KM3: A DSL for Metamodel Specification. In *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*. Springer-Verlag, 171–185.

Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Andreas Kästner, Martin Gogolla, and Bran Selic. 2018. From (Imperfect) Object Diagrams to (Imperfect) Class Diagrams: New Ideas and Vision Paper. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*. ACM, 13–22.

Anneke Kleppe. 2006. MCC: A model transformation environment. In *ECMDA-FA (LNCS)*, Vol. 4066. 173–187.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. The Epsilon Object Language (EOL). In *ECMDA-FA (LNCS)*, Vol. 4066. Springer, 128–142. See also https://www.eclipse.org/epsilon/doc/eol/.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. The Epsilon Transformation Language. In *ICMT (LNCS)*, Vol. 5063. Springer, 46–60. See also https://www.eclipse.org/epsilon/doc/etl/.

Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. 2015. Reuse in model-to-model transformation languages: are we there yet? *Software and Systems Modeling* 14, 2 (2015), 537–572.

Kevin Lano and Shekoufeh Kolahdouz Rahimi. 2014. Model-transformation design patterns. *IEEE Trans. Software Eng.* 40, 12 (2014), 1224–1259.

Kevin Lano, Shekoufeh Kolahdouz Rahimi, Iman Poernomo, Jeffrey Terrell, and Steffen Zschaler. 2014. Correct-by-construction synthesis of model transformations using transformation patterns. *Software and Systems Modeling* 13, 2 (2014), 873–907.

Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Software and System Modeling* 14, 4 (2015), 1323–1347.

Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. 2016. Model transformation intents and their properties. *Software and Systems Modeling* 15, 3 (2016), 647–684.

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer. See also https://featureide.github.io/.

Object Management Group. 2005. UML 2.0 OCL Specification. (2005). https://www.omg.org/spec/UML/2.0/.

Object Management Group. 2016. MOF Query/View/Transformation (QVT). (2016). https://www.omg.org/spec/QVT/.

Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. 2015. Pattern-based development of domain-specific modelling languages. In *MODELS*. IEEE, 166–175.

Rick Salay, Steffen Zschaler, and Marsha Chechik. 2016. Correct reuse of transformations is hard to guarantee. In *ICMT (LNCS)*, Vol. 9765. Springer, 107–122.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2014a. A component model for model transformations. *IEEE Trans. Software Eng.* 40, 11 (2014), 1042–1060.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2014b. Reverse engineering of model transformations for reusability. In *ICMT (LNCS)*, Vol. 8568. Springer, 186–201.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Reusable model transformation components with bentō. In *ICMT (LNCS)*, Vol. 9152. Springer, 59–65.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2017. Static analysis of model transformations. *IEEE Trans. Software Eng.* 43, 9 (2017), 868–897.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2018. Quick fixing ATL transformations with speculative analysis. *Software and Systems Modeling* 17, 3 (2018), 779–813.

Jesús Sánchez Cuadrado and Jesús García Molina. 2008. Approaches for model transformation reuse: Factorization and composition. In *ICMT (LNCS)*, Vol. 5063. Springer, 168–182.

Jesús Sánchez Cuadrado and Jesús García Molina. 2009. Modularization of model transformations through a phasing mechanism. *Software and Systems Modeling* 8, 3 (2009), 325–345.

Douglas C. Schmidt. 2006. Guest editor's introduction: Model-driven engineering. *Computer* 39, 2 (2006), 25–31.

Gehan M. K. Selim, James R. Cordy, and Juergen Dingel. 2017. How is ATL really used? Language feature use in the ATL zoo. In *MoDELS*. IEEE Computer Society, 34–44.

Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. 2009. Meta-model pruning. In *MODELS (LNCS)*, Vol. 5795. Springer, 32–46.

Shane Sendall and Wojtek Kozaczynski. 2003. Model transformation: The heart and soul of model-driven software development. *IEEE software* 20, 5 (2003), 42–45.

Sirius. last accessed in 2018. https://eclipse.org/sirius/. (last accessed in 2018).

Kenneth Slonneger and Barry L Kurtz. 1995. *Formal syntax and semantics of programming languages*. Vol. 340. Addison-Wesley Reading.

Jim Steel and Jean-Marc Jézéquel. 2007. On model typing. *Software and Systems Modeling* 6, 4 (2007), 401–413.

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional. See also http://www.eclipse.org/modeling/emf/.

Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-based model transformation: formal foundation and application. *Formal Asp. Comput.* 30, 1 (2018), 133–162.

Javier Troya, Sergio Segura, José Antonio Parejo, and Antonio Ruiz Cortés. 2018. Spectrum-based fault localization in model transformations. *ACM Trans. Softw. Eng. Methodol.* 27, 3 (2018), 13:1–13:50.

Javier Troya and Antonio Vallecillo. 2011. A rewriting logic semantics for ATL. *Journal of Object Technology* 10 (2011), 5: 1–29.

Antonio Vallecillo and Martin Gogolla. 2012. Typing Model Transformations Using Tracts. In *ICMT (Lecture Notes in Computer Science)*, Vol. 7307. Springer, 56–71.

Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices ()* 35, 6 (2000), 26–36.

Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. 2013. Typing artifacts in megamodeling. *Software and System Modeling* 12, 1 (2013), 105–119.

Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. 2010. Module superimposition: a composition technique for rule-based model transformation languages. *Software and Systems Modeling* 9, 3 (2010), 285–309.

Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. 2014. The state of practice in model-driven engineering. *IEEE Software* 31, 3 (2014), 79–85.

Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris S. Kolovos, Richard F. Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. 2012. Surveying rule inheritance in model-to-model transformation languages. *Journal of Object Technology* 11, 2 (2012), 3: 1–46.

Steffen Zschaler. 2014. Towards constraint-based model types: A generalised formal foundation for model genericity. In *VAO*. ACM, New York, NY, USA, Article 11, 8 pages.

A. APPENDIX

This appendix contains the proofs of the main results in the paper. The Alloy specifications used to support these proofs are available at http://miso.es/trms/.

A.1. Lemma 5.5: Composition of DRM mappings is well-formed (Section 5.2)

PROOF. Given two composable mappings $m^1\colon RM \to RM'$ and $m^2\colon RM' \to RM''$, we need to prove that their composition $m = m^2 \circ m^1\colon RM \to RM''$ (as in Definition 5.4) is well-formed. According to Definition 5.1, m is well-formed if it fulfils the following conditions:

— m_{CS} should be a total function. We have that $m_{CS} = m_C \cup m_S = m^2_C \circ m^1_C \cup (m^2_S \circ m^1_S \cup m^2_S \circ m^1_C \cup m^2_C \circ m^1_S) = m^2_C \circ (m^1_C \cup m^1_S) \cup m^2_S \circ (m^1_S \cup m^1_C) = (m^2_C \cup m^2_S) \circ (m^1_C \cup m^1_S)$. Since $m^2_C \cup m^2_S$ and $m^1_C \cup m^1_S$ are total functions, so is m_{CS}.

— $\forall C \in RM_C \bullet m_S(C)\ is\ defined \implies C.subsAllowed = true$. As we have $m_S = m^2_S \circ m^1_S \cup m^2_S \circ m^1_C \cup m^2_C \circ m^1_S$, we proceed by cases:
  — For those $C \in RM_C$ s.t. $m^2_S \circ m^1_S(C)$ or $m^2_C \circ m^1_S(C)$ is defined, this condition holds because it holds for m^1_S.
  — For those $C \in RM_C$ s.t. $m^2_S \circ m^1_C(C)$ is defined, it holds because the composability condition states: $\forall C \in RM_C \bullet m^2_S(m^1_C(C))\ is\ defined \implies C.subsAllowed = true$.

— $\forall C \in RM_C \bullet m_S(C)\ is\ defined \implies \forall C_s \in C.subs \bullet m_S(C_s)\ is\ defined$. Again, we proceed by cases on m_S:
  — For those $C \in RM_C$ s.t. $m^2_S \circ m^1_S(C)$ or $m^2_C \circ m^1_S(C)$ is defined, this condition holds because it holds for m^2_S and m^1_S.
  — For those $C \in RM_C$ s.t. $m^2_S \circ m^1_C(C)$ is defined, it holds because the composability condition states: $\forall C \in RM_C \bullet m^2_S(m^1_C(C))\ is\ defined \implies \forall C_s \in C.subs \bullet m^1_S(C_s)\ is\ defined \vee m^2_S(m^1_C(C_s))\ is\ defined$. This means that, given a $C_s \in C.subs$, if $m^1_S(C_s)\ is\ defined$, then $m_S(C_s)$ is defined; while if $m^2_S(m^1_C(C_s))$ is defined, then $m_S(C_s)$ is defined.

— $\forall C \in RM_C \bullet m_C(C)\ is\ defined \implies \forall f \in C.feats \bullet m_F(f)\ is\ defined$. This holds because $m_C = m^2_C \circ m^1_C$, and both m^2_C and m^1_C satisfy this condition.

□

A.2. Theorem 5.6: Composition of strong refinements is strong refinement (Section 5.2)

PROOF. Given two composable strong refinements $m^1\colon RM \to RM'$ and $m^2\colon RM' \to RM''$, we need to prove that $m = m^2 \circ m^1$ is a strong refinement. We proceed by cases, checking that every predicate in $srefinement$ holds for m:

(1) $name\_refinement$ demands $\forall C \in RM_{NC} \bullet m_C(C).name = C.name$. According to the definition of DRM mapping composition (Definition 5.4), we have that $m_C = m^2_C \circ m^1_C$. As both m^1 and m^2 are strong refinements, we have that $\forall C \in RM_{NC} \bullet C.name = m^1_C(C).name$, and $\forall C \in RM'_{NC} \bullet C.name = m^2_C(C).name$. Hence, we have $\forall C \in RM_{NC} \bullet C.name = (m^2_C \circ m^1_C)(C).name = m_C(C).name$ as required.

(2) $abstract\_refinement$ demands $\forall C \in RM_C \bullet C.isAbstract = any \vee m_{CS}(C).isAbstract = C.isAbstract$. According to Definition 5.4, $m_{CS} = (m^2_C \cup m^2_S) \circ (m^1_S \cup m^1_C)$. For those classes

C with $isAbstract \neq any$, $isAbstract$ is preserved through all $m^1_S, m^1_C, m^2_S, m^2_C$. Hence, for each class C, either $C.isAbstract = any$, or $m_{CS}(C).isAbstract = C.isAbstract$.

(3) $ancs\_preservation$ demands $\forall C, C_a \in RM_C \bullet C_a \in C.ancs^+ \implies m_{CS}(C_a) \in m_{CS}(C).ancs^*$. As $ancs^*$ is preserved by both m^1_{CS} and m^2_{CS}, it is preserved by m_{CS}.

(4) $antiancs\_preservation$ demands $\forall C, C_a \in RM_C \bullet C_a \in C.antiancs^+ \implies m_{CS}(C_a) \in m_{CS}(C).antiancs^+$. As $antiancs^+$ is preserved by both m^1_{CS} and m^2_{CS}, it is preserved by m_{CS}.

(5) $mand\_allowed\_refinement$ demands:

$$\begin{aligned}&\forall C \in RM_C \bullet m_C(C)\ is\ defined \wedge C.mandatoryAllowed = false \implies\\ &\quad |\{f \mid f \in C.feats^* \wedge isMand(f)\}| = |\{f \mid f \in m_C(C).feats^* \wedge isMand(f)\}|\end{aligned}$$

Predicate $mand\_allowed\_refinement$ holds for m^1 and m^2 as both are refinements. As they also are strong refinements, predicates $mand\_allowed\_preservation(m^1)$ and $mand\_allowed\_preservation(m^2)$, which demand preserving the flag $subsAllowed$, hold as well. Therefore, given a class $C \in RM_C$ with $C.mandatoryAllowed = false$, then $m^1_C(C).mandatoryAllowed = false = m^2_C(m^1_C(C)).mandatoryAllowed$. This means that $|\{f \mid f \in C.feats^* \wedge isMand(f)\}| = |\{f \mid f \in m^1_C(C).feats^* \wedge isMand(f)\}| = |\{f \mid f \in m^2_C(m^1_C(C)).feats^* \wedge isMand(f)\}|$, and so $mand\_allowed\_refinement(m^2 \circ m^1)$ holds.

(6) $feature\_refinement$ demands:

$$\begin{aligned}&\forall C \in RM_C \bullet m_C(C)\ is\ defined \implies\\ &\qquad \forall f \in C.feats \bullet m_F(f) \in m_C(C).feats^* \wedge\\ &\qquad f.name = m_F(f).name \wedge refines_{card}(f, m_F(f))\end{aligned}$$

Assuming that $feature\_refinement$ holds for m^1 and m^2, it is easy to see that the previous predicate, up to $f.name = m^2_F(m^1_F(f)).name$, holds for $m^2 \circ m^1$. In addition, predicate $refines_{card}(f, m^2_F(m^1_F(f)))$ holds because $refines_{num}$ and $refines_{many}$ are transitive. This is so as predicates $card\_preservation(m^1)$ and $card\_preservation(m^2)$ hold because m^1 and m^2 are strong refinements. This ensures equality of the $allowLess$ and $allowMore$ flags.

(7) $feature\_type\_commut$ demands:

$$\begin{aligned}\forall f \in RM_F, \forall ft \in f.types \bullet\ & m_A(ft)\ is\ defined \implies m_A(ft) \in m_F(f).types \wedge\\ & m_R(ft)\ is\ defined \implies m_R(ft) \in m_F(f).types\end{aligned}$$

which holds by transitivity because $m_A = m^2_A \circ m^1_A$, $m_R = m^2_R \circ m^1_R$, and $m_F = m^2_F \circ m^1_F$.

(8) $feature\_type\_refinement$ demands:

$$\begin{aligned}&\forall f \in RM_F \bullet m_F(f)\ is\ defined \implies ((f.types = \emptyset) \vee\\ &\quad (m_F(f).types \neq \emptyset \wedge \forall ft' \in m_F(f).types\ \exists ft \in f.types \bullet\\ &\qquad (m_A(ft) = ft' \wedge refines_{Attr}(ft, ft')) \vee\\ &\qquad (m_R(ft) = ft' \wedge refines_{Ref}(ft, ft'))))\end{aligned}$$

The predicate holds because, given $f \in RM_F$, if $f.types = \emptyset$, then $feature\_type\_refinement(m^2)$ holds. Otherwise, if $f.types \neq \emptyset$, we need to show that $refines_{Attr}(\_,\_)$ and $refines_{Ref}(\_,\_)$ are transitive. Predicate $refines_{Attr}(\_,\_)$ is transitive due to the transitivity of equality. Predicate $refines_{Ref}(\_,\_)$ is transitive as well because predicate $ref\_sem\_preservation$ (which holds because both mappings are strong refinements) forces references to be mapped with same *open* value. For the case of *open* references, mappings can monotonically increase the allowed targets. For *closed* references, mappings can monotonically decrease the allowed targets. Both cases lead to a correct composition.

(9) $subs\_refinement$ demands:

$$\forall C \in RM_C \bullet m_S(C)\ is\ defined \implies |\ conc\_subs(m_S(C))\ | > 0 \wedge \\ \forall C' \in conc\_subs(m_S(C)) \bullet \\ mand\_allowed\_refinement(C, C') \wedge \\ \forall f \in C.feats\ \exists f' \in C'.feats^* \bullet feats\_refinement(f, f')$$

Mapping m_S can be obtained by composing $m_S^2 \circ m_C^1$ ($m_C^2 \circ m_S^1$ and $m_S^2 \circ m_S^1$ are empty according to the condition for composition). Hence the predicate holds, because it holds for m_S^2.

(10) $mand\_allowed\_preservation$ and $card\_preservation$ hold in m due to the transitivity of element inclusion and equality.

(11) The first part of predicate $ref\_sem\_preservation$ (where m_R is defined) holds due to the transitivity of equality. For the second part, if m_S^2 of the owner classes are defined, the predicate holds, as it holds for m_S^2.

□

Note that the predicates for strong refinement are required for proving compositionality of the refinement predicates. Hence, $mand\_allowed\_preservation$ is required for $mand\_allowed\_refinement$, $card\_preservation$ for $feature\_refinement$, and $ref\_sem\_preservation$ for $feature\_type\_refinement$. Other properties of DRMs, like $subsAllowed$, are not required to be preserved.

A.3. Identity refinement

Given a DRM RM, we introduce an identity refinement $id_{RM}\colon RM \to RM$, which is a strong refinement. This identity refinement will be used to prove in Appendices A.4 to A.14 that each refinement in the catalogue presented in Section 5.2.1 are strong refinements.

LEMMA A.1 (IDENTITY REFINEMENT). *Given a DRM RM, the identity refinement $id_{RM}\colon RM \to RM$ defined as $id_{RM} = \langle id_C\colon RM_C \to RM_C, id_S = \emptyset, id_F\colon RM_F \to RM_F, id_R\colon RM_R \to RM_R, id_A\colon RM_A \to RM_A\rangle$ is a valid DRM mapping that satisfies $srefinement(id_{RM})$.*

Remark. For simplicity, we refrain from introducing other possible identity refinements, e.g., mapping through id_S the subset of classes C such that $C.subsAllowed = true$, and satisfying $mand\_allowed\_refinement(C, C_{sub})$ for each concrete subclass C_{sub} of C.

PROOF. We need to prove that the identity mapping $id_{RM}\colon RM \to RM$ is a valid DRM mapping and is a strong refinement. According to Definition 5.1, id_{RM} is a valid DRM mapping if:

— $id_{CS} = id_C \cup id_S$ is a total function (holds as id_C is defined on the whole RM_C, and id_S empty).
— $\forall C \in RM_C \bullet id_S(C)\ is\ defined \implies$ ... (holds because id_S is empty).
— $\forall C \in RM_C \bullet id_C(C)\ is\ defined \implies \forall f \in C.feats \bullet id_F(f)\ is\ defined$ (holds because id_F is defined on the whole RM_F).

To prove strong refinement we proceed by cases, checking that id_{RM} satisfies every predicate in $srefinement$ (cf. Definition 5.2):

(1) $name\_refinement$ and $abstract\_refinement$ hold because classes are mapped using the identity mapping ($id_C \cup id_S$).
(2) $ancs\_preservation$ and $antiancs\_preservation$ hold because of the identity mapping of classes, and because relations $ancs$ and $antiancs$ do not change.
(3) $mand\_allowed\_refinement$ holds because the number of features in every class does not change, and each feature is mapped to itself via the identity mapping id_F.
(4) $feature\_refinement$ holds because of the identity mapping id_F, and noting that the auxiliary predicates $refines_{num}$ and $refines_{many}$ hold if both $Card$ arguments are equal.

(5) $feature\_type\_commut$ holds because of the identity mappings id_R and id_A.
(6) $feature\_type\_refinement$ holds because of the identity mappings id_R and id_A, and noting that the auxiliary predicates $refines_{Attr}$ and $refines_{Ref}$ hold for two equal arguments of type A or R.
(7) $subs\_refinement$ holds because id_S is empty.
(8) $mand\_allowed\_preservation$ holds because of the identity mapping of classes.
(9) $card\_preservation$ and $reference\_sem\_preservation$ hold because of the identity mapping of features and reference types.

Cases (1-7) are for refinement, and cases (8-9) are for strong refinement. □

A.4. Refinement operation: *Adding new class* (Section 5.2.1)

PROOF. Given a DRM RM, the identity mapping $id_{RM}\colon RM \to RM$ is a strong refinement (cf. Appendix A.3). We construct a new DRM RM' by adding a named class with a fresh name to RM_{NC}, so that $RM' = \langle RM_{NC} \cup \{C'\}, \_\rangle$[7]. The mapping id_{RM} also applies if RM' is in the codomain, and is still a strong refinement as $\nexists C \in RM_C$ s.t. $id_{CS}(C) = C'$. Therefore, the evaluation of each predicate used by $srefinement$ does not change. The same reasoning follows when adding a new anonymous class to RM_{AC}. □

A.5. Refinement operation: *Assigning fresh name to anonymous class* (Section 5.2.1)

PROOF. Given a DRM RM, we construct a new DRM by giving a fresh name "X" to an anonymous class: $RM' = \langle RM_{NC} \cup \{C\}, RM_{AC} \setminus \{C\}, name \cup \{(C, \text{"X"})\}, \_\rangle$. The identity mapping id_{RM} applies when RM' is in the codomain, and it satisfies $srefinement$ because predicate $name\_refinement$ only applies to named classes in RM but not in RM'. □

A.6. Refinement operation: *Setting class to abstract or concrete* (Section 5.2.1)

PROOF. Given a DRM RM and a class $C \in RM_C$ with $C.isAbstract = any$, we create a new DRM $RM' = \langle \_, isAbstract{+}{+}\{(C, true)\}, \_\rangle$ where C is abstract (we use "++" to denote function overriding). The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $abstract\_refinement$ holds because $C.isAbstract = any$ in RM. The same reasoning applies if we set C to concrete in RM'. □

A.7. Refinement operation: *Adding feature to class* (Section 5.2.1)

PROOF. Given a DRM RM and a class C s.t. $C.mandatoryAllowed = false$, we create a new DRM RM' by adding a new optional feature f to C: $RM' = \langle RM_F \cup \{f\}, feats \cup \{(C, f)\}, Number \cup \{n\}, value \cup \{(n, 0)\}, allowLess \cup \{(n, false)\}, allowMore \cup \{(n, false)\}, min \cup \{(f, n)\}, \_\rangle$ (where f can have any max cardinality). The identity mapping id_{RM} is applicable when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $mand\_allowed\_refinement$ holds because $isMand(f)$ is false and the set of mandatory features of C in RM and RM' stays the same.

If the feature f is added to a class C s.t. $C.mandatoryAllowed = true$, then predicate $mand\_allowed\_refinement$ holds regardless of whether f is mandatory or not. □

A.8. Refinement operation: *Refining the possible types of a feature* (Section 5.2.1)

PROOF. Given a DRM RM and a feature f s.t. $|f.types| = 0$, we create a new DRM RM' by adding an attribute type a to f: $RM' = \langle A \cup a, types \cup \{(f, a)\}, \_\rangle$ (where $a.dtype$ is unimportant). The identity mapping id_{RM} also applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $feature\_type\_refinement$ holds because $|f.types| = 0$ in RM. The same reasoning holds if we add a reference type to f, or a set of attribute and reference types.

[7]For simplicity, this and the following proofs only show the parts of the DRM tuple that change, representing the unchanged parts as _.

Given a DRM RM, a feature f s.t. $|f.types| > 1$, and a feature type $t \in f.types$, we create a new DRM RM' by deleting t from $f.types$: $RM' = \langle types \setminus \{(f,t)\}, \_\rangle$. The identity mapping id_{RM} needs to be modified by removing the mappings from id_R or id_A to f, and it is still a strong refinement. This is so as predicate $feature\_type\_refinement$ holds because $f.types$ is not empty in RM', and for every remaining feature type $ft \in f.types$, $refines_{Attr}(ft, ft)$ or $refines_{Ref}(ft, ft)$, while the other predicates hold, as they hold for id_{RM}. □

A.9. Refinement operation: *Refining type of attribute* (Section 5.2.1)

PROOF. Given a DRM RM and a feature f with an attribute type a s.t. $a.dtype = AnyDT$, we create a new DRM RM' by changing the type of a to any element $t \in DataType$: $RM' = \langle dtype + +\{(a,t)\}, \_\rangle$. The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $feature\_type\_refinement$ holds because $refines_{Attr}(RM.a, RM'.a)$ holds (with $RM'.a$ the attribute type a in RM') since $a.dtype = AnyDT$ in RM. The proof when changing $Numeric$ by $Real$ or $Integer$ is analogous. □

A.10. Refinement operation: *Splitting class in hierarchy* (Section 5.2.1)

PROOF. Given a DRM RM and a class $C \in RM_C$ s.t. $F_1 \subseteq C.feats$, we create a new DRM RM' by adding a superclass C_{sup} with fresh name to C and moving the features in F_1 to C_{sup}: $RM' = \langle RM_C \cup \{C_{sup}\}, ancs \cup \{(C, C_{sup})\}, feats \cup \{(C_{sup}, f_i)\}_{f_i \in F_1} \setminus \{(C, f_i)\}_{f_i \in F_1}, \_\rangle$. The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $ancs\_preservation$ holds because the $ancs$ relation is extended but existing values are not modified. Predicate $feature\_refinement$ also holds because the features of C in RM can be mapped to features owned or inherited by $id_{RM}(C)$. □

A.11. Refinement operation: *Adding/deleting target of reference* (Section 5.2.1)

PROOF. Given a DRM RM and a reference r s.t. $r.open = true$, we create a new DRM RM' by adding a new target class $C \in RM_C$ s.t. $C \notin r.targets$ to $r.targets$: $RM' = \langle targets \cup \{(r, C)\}, \_\rangle$. The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $feature\_type\_refinement$ holds because $refines_{Ref}(RM.r, RM'.r)$ (with $RM'.r$ the reference type r in RM') holds, since each target of r in RM is a target of r in RM'.

Given a DRM RM with a reference r s.t. $r.open = false$ and $|r.types| > 1$, we create a new DRM RM' by deleting a target class $C \in r.targets$ from $r.targets$: $RM' = \langle targets \setminus \{(r, C)\}, \_\rangle$. The identity mapping id_{RM} applies when RM' is in the codomain, and is still a strong refinement. This is so as $feature\_type\_refinement$ holds because $refines_{Ref}(RM.r, RM'.r)$ holds, since each target of r in RM' is a target of r in RM (as they are mapped via an identity mapping). □

A.12. Refinement operation: *Adding subclass consistent with antiancs* (Section 5.2.1)

PROOF. Given a DRM RM with two classes $\{C, RC\} \subseteq RM_C$ s.t.

$$\begin{aligned}\forall C_a \in C.ancs^* \bullet (\nexists C_s \in C_a.antiancs^* \bullet C_s \in RC.ancs^* \wedge \\ \nexists C_s \in RC.ancs^* \bullet C_a \in C_s.antiancs^*)\end{aligned}$$

we create a new DRM RM' by adding C as a subclass of RC: $RM' = \langle ancs \cup \{(C, RC)\}, \_\rangle$. The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $ancs\_preservation$ holds because the $ancs$ relation is extended but existing values are not modified, and $antiancs\_preservation$ holds because $antiancs$ is not modified. If C is a new class added to RM', we do not need to make any assumption on the relation between C and RC, but $ancs\_preservation$ holds because $ancs$ is only extended.

Please note that RM' is a valid DRM, since condition (1) of Definition 4.1 holds: given any $C_1 \in RC.ancs^*$ and any $C_2 \in C.ancs^*$, the refinement condition ensures $C_1 \notin C_2.antiancs^*$ and vice versa, as required by condition (1). □

A.13. Refinement operation: *Refining minimum cardinality* (Section 5.2.1)

PROOF. Given a DRM RM with a feature f s.t. $min(f) = AnyCardinality$, we create a new DRM RM' by assigning to f any $n \in Number$ as cardinality: $RM' = \langle min + +\{(f,n)\}, Number \cup \{n\}, \_\rangle$ (where the value of n is unimportant). The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $feature\_refinement$ holds because $refines_{min}(RM.f.min, RM'.f.min)$ holds. Generally, the predicate holds when $f.min$ is $AnyCardinality$, and it also supports the following refinements when $f.min$ is a $Number$: a smaller numeric value when $allowLess = true$, and a bigger numeric value when $allowMore = true$. $\square$

A.14. Refinement operation: *Refining maximum cardinality* (Section 5.2.1)

PROOF. Given a DRM RM with a feature f s.t. $max(f) = AnyCardinality$, we create a new DRM RM' by assigning to f any $n \in Number$ as cardinality: $RM' = \langle max + +\{(f,n)\}, Number \cup \{n\}, \_\rangle$ (where the value of n is unimportant). The identity mapping id_{RM} applies when RM' is in the codomain, and it is still a strong refinement. This is so as predicate $feature\_refinement$ holds because $refines_{max}(RM.f.max, RM'.f.max)$ holds. Generally, the predicate holds when $f.max$ is $AnyCardinality$, and it also supports the following refinements when $f.max$ is a $Number$: a smaller numeric value when $allowLess = true$, and a bigger numeric value or $Many$ when $allowMore = true$. $\square$

A.15. Theorem 5.12: Retyping is well-formed (Section 5.4)

PROOF. Given a refinement $m\colon RM \to RM'$ and a write well-formed typing $type\colon M \to RM'$, we need to show that $back(type)\colon M \to RM$ (in Definition 5.11) is well-formed if RM' is a meta-model or $srefinement(m)$ holds. We proceed by cases, showing that each condition for typing well-formedness holds:

(1) *Objects are not typed by two classes such that one class is antiancestor of any ancestor of the other.* As m is a refinement, $antiancs\_preservation(m)$ holds, and so m_{CS} cannot map two antiancestor classes to classes that are not antiancestors. Assume an object typed by two classes $C_1, C_2 \in RM'_C$ in the codomain of m_C. Then, function $objBck$ calculates the typing $type'$ using m_C^{-1}. The only way for an object to obtain a typing from RM made of two antiancestor classes is that C_1 and C_2 are neither ancestors nor antiancestors. However, $antiancs\_preservation$ forbids this.
Now assume that C_1, C_2 are not in the codomain of m_C, but their ancestors receive a mapping via m_{CS}. Because $type$ is well-formed, any ancestor of C_1, C_2 cannot be antiancestor of each other. As $antiancs\_preservation$ holds, the resulting typing classes in RM_C cannot be antiancestors of each other.

(2) *Slots are typed by features owned or inherited by some of the object types.* The typing $type^*_{slots}$ assigns a typing in two ways. First, using m_F backwards on objects typed by classes that receive a mapping with m_C. In this case, function $back$ ensures that a feature f is selected as the type for slot s s.t. $\exists d \in type^*_{objs}(s.owner) \wedge f.owner \in d.ancs^*$, hence satisfying the property. Similarly, for those features for which m_F is not defined, $back$ selects a feature f as the type for s s.t. $\exists da \in type^*_{objs}(s.owner) \bullet f.owner \in da.ancs^*$, hence satisfying the property.

(3) *The type of every slot is not contradictory with that of the feature.* Given a field s typed by a feature f', then either $f'.types$ is empty, or it contains an attribute a'. Now, if $f' = m_F(f)$, $back$ assigns f as the new type. However, as m is a refinement, then $feature\_type\_refinement$ holds, and so either $f.types$ is empty, or it contains some attribute a s.t. $refines_{Attr}$ holds. Similarly, if $\nexists f \bullet m_F(f) = f'$, then $back$ selects a feature f s.t. $feature\_type\_refinement(f, f')$ holds. An analogous reasoning can be done if s is a link.

(4) *Slots obey the minimum cardinality of their types.* This property demands the number of slots s owned by an object o and typed by a feature f s.t. $f.min.allowLess = false$, to be greater or equal than $f.min.value$. If RM' is a meta-model, all features have $allowLess = false$, which

means this property is checked for every $f' \in RM'_F$. Now, assume we have some $f \in RM_F$ s.t. $m_F(f) = f'$. Then, $f.min.allowLess$ can be true or false. If it is true, then the minimum cardinality is not checked on the feature. If it is false, according to $feature\_refinement$, we have $f.min.value = f'.min.value$, and so the property holds for f, because it holds for f'. If $f.min.allowMore$ is true, according to $feature\_refinement$, we may have $f.min.value \leq f'.min.value$, in which case the property holds, as the minimum value for f' is lower.
If instead of RM' being a meta-model, we have $srefinement(m)$, then $card\_preservation(m)$ holds. This means that for any $f \in RM_F \bullet f.min.allowLess = m_F(f).min.allowLess$ (and similar for $allowMore$). Hence, if the minimum cardinality of any $f' \in RM'_F$ with $f'.min.allowLess = false$ is satisfied in every object, so it is satisfied for f (with $m_F(f) = f$), because by predicate $feature\_refinement$, we have $f.min.value = f'.min.value$. If f' is not in the codomain of m_F, the condition still holds, because $card\_preservation$ ensures preservation of $allowLess$ on any feature f that can be mapped to f' s.t. $features\_refinement(f, f')$. As function $back$ only selects as type features f' with $features\_refinement(f, f')$, then the condition holds.

(5) *Slots obey the maximum cardinality of their types*. This property demands the number of slots s owned by an object o and typed by a feature f s.t. $f.max.allowMore = false$, to be less or equal than $f.max.value$. If RM' is a meta-model, all features have $allowMore = false$, which means this property is checked for every $f' \in RM'_F$. Now, assume we have some $f \in RM_F$ s.t. $m_F(f) = f'$. Then, $f.max.allowMore$ can be true or false. If it is true, then the maximum cardinality is not checked on the feature. If it is false, according to $feature\_refinement$, we have $f.max.value = f'.max.value$, and so the property holds for f, because it holds for f'. If $f.max.allowLess$ is true, according to $feature\_refinement$, we may have $f.max.value \geq f'.max.value$, in which case the property holds, as the maximum value for f is higher.
For the case of $srefinement(m)$, we have $card\_preservation(m)$. This means that for any $f \in RM_F \bullet f.max.allowMore = m_F(f).max.allowMore$ (and similar for $allowLess$). Hence, if the maximum cardinality of any $f' \in RM'_F$ with $f'.max.allowMore = false$ is satisfied in every object, so it is satisfied for f (with $m_F(f) = f$), because by predicate $feature\_refinement$, we have $f.max.value = f'.max.value$. However, a class in RM may have several features compatible with f', and $back$ would try to type the slot by all of them, with the risk of violating the maximum cardinality. To avoid overstepping this, $back$ adds the condition $|\{s \in o.slots \mid type^*_{slots}(s) = f\}| < f.max.value$, hence ensuring that the maximum cardinality holds. This fact effectively allows several valid retypings.

(6) *Every link target is coherent with its type*. This property demands that the objects pointed by links typed by closed references conform to some of the reference targets. If RM' is a meta-model, all its references are closed, and therefore, this property holds for every typed link. Then, we may have $r \in RM_R$ mapped to r' in RM'_R. As m is a refinement, we have $refines_{Ref}(r, r')$. If r is open, the property does not need to be checked for r. If r is closed, then r' may have less targets than r, so any link end compatible with targets in r' will be compatible with targets in r. If $srefinement(m)$, then we need to have $r.open = r'.open$ due to predicate $ref\_sem\_preservation$. Therefore, if r' is closed, r is also closed and the same reasoning applies.

Note that we require $type$ to be write well-formed, to avoid typing any object o by an abstract class C'. Assume $m_S(C) = C'$. The mapping m_S only requires conformance for non-abstract subclasses of C' (or by C' if it is non-abstract). This means that C may have features that are not matched in C' (as C' is abstract), hence producing incorrect retypings due to the violation of minimum cardinalities. However, starting with a typing that is write well-formed avoids this problem. □