Verification and Validation of Declarative Model-to-Model Transformations Through Invariants

Jordi Cabot*[,a], Robert Clarisó[a], Esther Guerra[b], Juan de Lara[c]

[a]*Estudis d'Informàtica, Multimèdia i Telecomunicació, Univ. Oberta Catalunya (Spain)*
[b]*Computer Science Department, Universidad Carlos III de Madrid (Spain)*
[c]*Polytechnic School, Universidad Autónoma de Madrid (Spain)*

Abstract

In this paper we propose a method to derive OCL invariants from declarative model-to-model transformations in order to enable their verification and analysis. For this purpose we have defined a number of invariant-based verification properties which provide increasing degrees of confidence about transformation correctness, such as whether a rule (or the whole transformation) is satisfiable by some model, executable or total. We also provide some heuristics for generating meaningful scenarios that can be used to semi-automatically validate the transformations.

As a proof of concept, the method is instantiated for two prominent model-to-model transformation languages: Triple Graph Grammars and QVT.

Key words: Model-to-Model Transformation, Model-Driven Development, OCL, Verification and Validation, Triple Graph Grammars, QVT

1. Introduction

Model-Driven Development (MDD) is a software engineering paradigm where models are the core asset [47]. They are used to specify, simulate, test, verify and generate code for the application to be built. Many of these activities include the specification and execution of model-to-model (M2M) transformations, that is, the transformation of a model conformant to a meta-model into another one conformant to a different meta-model.

*Rbla. del Poblenou 156, E-08018 Barcelona, Spain
Email addresses: `jcabot@uoc.edu` (Jordi Cabot), `rclariso@uoc.edu` (Robert Clarisó), `eguerra@inf.uc3m.es` (Esther Guerra), `Juan.deLara@uam.es` (Juan de Lara)

There are two main approaches to M2M transformation: *operational* and *declarative*. The former is based on rules or instructions that explicitly state how and when creating the elements of the target model from elements of the source one. Instead, in declarative approaches, some kind of visual or textual patterns describing the relations between the source and target models are provided, from which operational mechanisms are derived e.g. to perform forward and backward transformations. These declarative patterns are complemented with additional information to express relations between attributes in source and target elements, as well as to constrain when a certain relation should hold. The Object Constraint Language (OCL) standard [33] is frequently used for this purpose [34].

The increasing complexity of modelling languages, models and transformations makes urgent the development of techniques and tools that help designers to assure transformation correctness. Whereas several notations have been proposed for specifying M2M transformations in a declarative way [1, 25, 34, 39], there is a lack of methods for analysing their correctness in an integral way, taking into account the relations expressed by the transformation, as well as the meta-models and their well-formedness rules.

In this paper we propose verification and validation techniques for M2M transformations based on the analysis of a set of OCL invariants automatically derived from the declarative description of the transformations. These invariants state the conditions that must hold between a source and a target model in order to satisfy the transformation definition, i.e. in order to represent a valid mapping. We call these invariants, together with the source and target meta-models, a *transformation model* [7]. To show the wide applicability of the technique, we study how to create this transformation model from two prominent M2M transformation languages: Triple Graph Grammars (TGGs) [39] and QVT [34].

Once the transformation model is synthesized, we can determine several correctness properties of the transformation by analysing the generated transformation model with any available tool for the verification of static UML/OCL class diagrams (see [2, 10, 13, 36, 43]). In particular, we have predefined a number of verification properties in terms of the extracted invariants, which provide increasing confidence on the transformation correctness. For example, we can check whether a relation or the whole transformation is applicable in the forward direction (i.e., whether there is a source model enabling a relation), forward weak executable (if we can find a pair of source and target models satisfying the relation and the meta-model constraints), forward strong executable (if a relation is satisfied whenever it is enabled), or total (whether all valid source models can be transformed). In order to illustrate this analysis, we show the use of the UMLtoCSP tool [13] to perform the verification. The tool translates the transformation model into a constraint satisfaction problem, which is then processed with constraint solvers to check different aspects of the model.

The transformation model can also be used for validation purposes. Given the transformation model, tools like UMLtoCSP can be used to automatically generate valid pairs of source and target models, or a valid target model for a given or partially specified source model. These generated pairs help designers in deciding whether the defined transformation reflects their intention, thus helping to uncover transformation defects. Additionally, we have devised heuristics to partially automate the validation process by means of generating potentially relevant scenarios (representing corner cases of the transformation) that the designer may be specially interested in reviewing.

This paper extends our preliminary work in [12]. Here, we propose a new way of handling OCL attribute conditions in TGGs which avoids algebraic manipulations; provide a new way of generating invariants, so as to make the resulting TGG and QVT invariants more uniform, easing its portability to other languages; present a detailed formalization of the extraction of invariants from QVT; provide a comprehensive list of formalized verification properties; and present a semi-automatic method for validation.

Paper organization. Section 2 introduces TGGs and our proposal for handling OCL attribute conditions. Section 3 presents the method for extracting invariants from TGGs. Sections 4 and 5 present such method for QVT. Section 6 shows the use of the invariants and UML/OCL analysis tools for the verification and validation of transformations. Section 7 compares with related work and Section 8 draws the conclusions. As running example we use a transformation between class diagrams and relational schemas [34]. The appendix includes all the invariants for the example.

2. Triple Graph Grammars

Triple Graph Grammars (TGGs) [39] were proposed by A. Schürr as a formal means to specify transformations between two languages in a declarative way. TGGs are founded on the notion of graph grammar [38]. A graph grammar is made of rules having graphs in their left and right hand sides (LHS and RHS), plus the initial graph to be transformed. Applying a rule to a graph is only possible if an occurrence of the LHS (a *match*) is found in it. Once such occurrence is found, it is replaced by the RHS graph. This is called *direct derivation*. It may be possible to find several matches for a rule, and then one is chosen at random. The execution of a grammar is also non-deterministic: at each step, one rule is randomly chosen and its application is tried. The execution ends when no rule can be applied.

Even though graph grammar rules rely on pre- and post-conditions, and on pattern matching, when used for model-to-model transformation, they have an operational, unidirectional style, as the rules specify how to build the target model assuming the source already exists. On the contrary TGGs are declarative and bidirectional since, starting from a unique TGG specifying the synchronized evolution of two graphs, it is possible to generate forward and backward transformations as well as operational mechanisms for other scenarios [26].

TGGs are made of rules working on triple graphs. These are made of two graphs called *source* and *target*, related through a *correspondence* graph. Any kind of graph can be used for these three components, from standard unattributed graphs $(V; E; s, t\colon E \to V)$ to more complex attributed graphs (e.g., E-graphs [17]). The nodes in the correspondence graph (the *mappings*) have morphisms[1] to the nodes in the source and target graphs. Triple morphisms are defined as three graph morphisms that preserve the correspondence functions. They are used to relate the LHS and RHS of a TGG rule, to identify a match of the LHS in a graph, and to type a triple graph.

Definition 1 (Triple Graph and Morphism). *A triple graph $TrG = (G_s, G_c, G_t, cs\colon V_{G_c} \to V_{G_s}, ct\colon V_{G_c} \to V_{G_t})$ is made of two graphs G_s and G_t called source and target, related through the nodes of the correspondence graph G_c.*

A triple graph morphism $f = (f_s, f_c, f_t) : TrG^1 \to TrG^2$ is made of three graph morphisms $f_x\colon G^1_x \to G^2_x$ (with $x = \{s, c, t\}$) such that the correspondence functions are preserved.

In the previous definition, V_{G_x} is the set of nodes of graph G_x. Morphisms cs and ct relate two nodes x and y in the source and target graphs iff $\exists n \in V_{G_c}$ with $cs(n) = x$ and $ct(n) = y$. We often depict a triple graph by $\langle G_s, G_c, G_t \rangle$,

[1]A morphism corresponds to the mathematical notion of total function between two sets, or in general between two structures (graphs, triple graphs, etc.)

and use TrG_x (for $x = \{s, c, t\}$) to refer to the x component of TrG. In this way, $\langle G_s, G_c, G_t \rangle_s = G_s$.

Fig. 1 shows a triple graph, taken from the class-to-relational transformation [34], which we use as a running example. The source graph is a class diagram with a package and a class, the target one is a relational schema model with one schema node, and the correspondence includes a mapping between the package and the schema. Note that "source" and "target" are relative terms, as we could also use source for the relational schema and target for the class diagram.

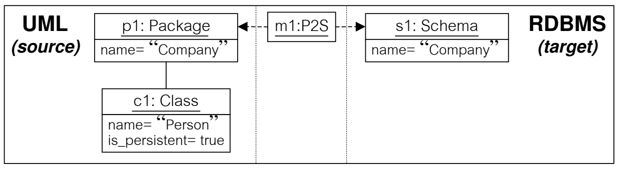


Figure 1: A triple graph example.

A triple graph is typed by a *meta-model triple* [21] or TGG schema, which contains the source and target meta-models and declares allowed mappings between both. Fig. 2 shows the meta-model triple for our running example. The correspondence meta-model declares five classes: `P2S` maps packages and schemas, `A2Co` maps attributes and columns, and `CT` and its specializations `C2T` and `C2TCh` relate classes and tables. In particular `C2TCh` is used to relate a children class with the table associated to its parent class. The dotted arrows specify the allowed morphisms from the correspondence to the source and target models, and can be treated as normal associations with cardinality 1 on the side of the source/target class. The meta-model includes OCL constraints ensuring uniqueness of attribute names for each class and table, as well as same persistence for a class and its children. As an example, the triple graph in Fig. 1 conforms to the meta-model in Fig. 2.

A typed triple graph is formally represented as $(TrG, type\colon TrG \to MM)$, where the first element is a triple graph and the second a morphism to the meta-model triple. Morphisms between typed triple graphs must respect the typing morphism and can take inheritance into account, as in [21]. For simplicity of presentation, we omit the typing in the following definitions.

Besides a meta-model triple, a M2M transformation by TGGs consists of a set of declarative rules that describe the synchronized evolution of two models. Rules have triple graphs in their LHS and RHS and may include

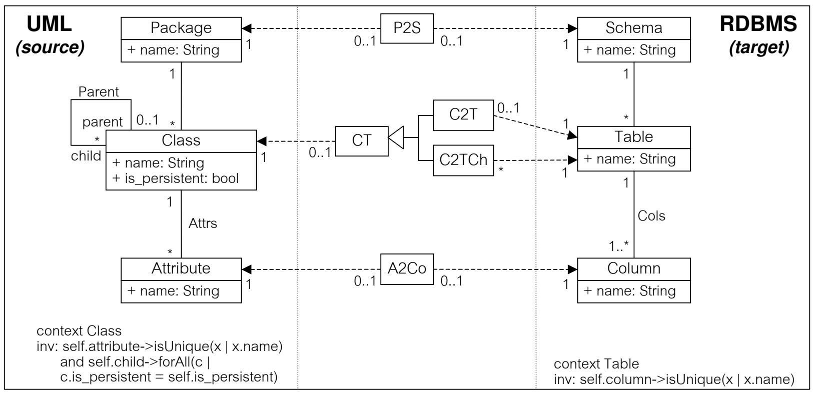


Figure 2: Example meta-model triple.

OCL attribute conditions. This contrasts with the usual approach of using attribute computations in the rules instead of conditions [17]. We use the latter as it poses some benefits that will be shown later on when operationalising the rules. Declarative rules are non-deleting because they describe how models are created, hence they are defined by an injective triple morphism.

Definition 2 (Declarative TGG Rule). *A declarative TGG rule* $p = (r\colon L \to R, ATT_{COND})$ *is made of two triple graphs,* $L = \langle L_s, L_c, L_t \rangle$ *and* $R = \langle R_s, R_c, R_t \rangle$*, an injective triple morphism* r *between* L *and* R*, and a set* ATT_{COND} *of OCL constraints over* R*, expressing attribute conditions.*

Fig. 3 shows four example TGG declarative rules using a compact notation that presents together L and R. The elements created by the rules (R-L) are marked as {`new`}, and the preserved elements are untagged. As an example, rule `Class-Table` is shown in the upper row first in extended and then in compact notation.

Rule `Package-Schema` declares that every time a package is created, a schema with the same name is created simultaneously, and vice versa. Rule `Class-Table` specifies that creating a persistent class in a package already related to a schema should create a table with the same name in that schema, and vice versa. In this case the attribute condition demands the class to have no parent. Note that we do not demand the LHS/RHS of rules to satisfy the integrity constraints of the meta-model. For example the RHS of the rule `Class-Table` is not a valid model because, according to the meta-model in Fig. 2, each table should be connected to at least one column. When

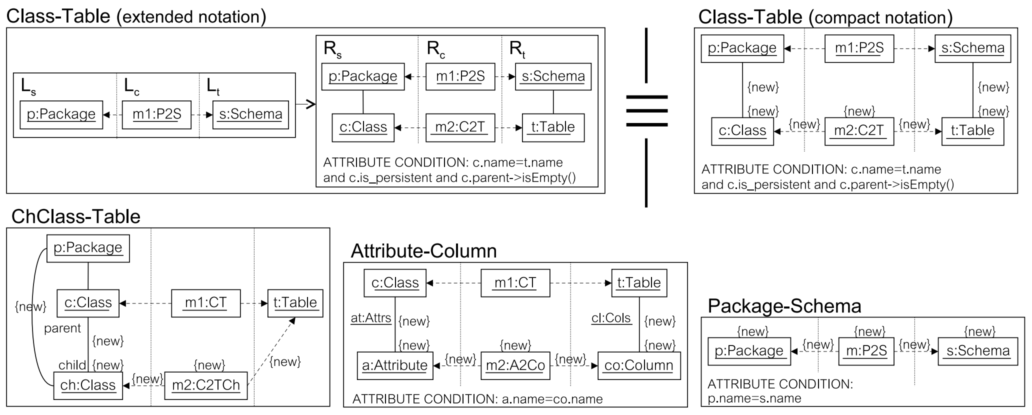


Figure 3: Some declarative TGG rules for the class-to-relational transformation.

executing a transformation it is acceptable to go through some intermediate models that are inconsistent with respect to the meta-model's constraints. What is important is that the final models are consistent.

For non top-level classes, the rule `ChClass-Table` is used instead of rule `Class-Table`. This rule specifies that creating a child class of a class already related to a table should map the child class to the same table. Finally, `Attribute-Column` synchronously creates attributes and columns for classes related to tables.

A TGG is bidirectional as rules do not specify any direction, but synchronously create and relate source and target elements. A TGG defines the language of all triple graphs that satisfy the meta-model constraints and that can be derived using zero or more applications of the grammar rules. Please note that some derived graphs may not conform to the meta-model, and hence are not part of the language.

In practice, one does not use declarative TGG rules to create source and target models at the same time, as it would require a synchronous coupling of both models. Instead, so-called operational rules are derived for different tasks, e.g. to perform forward (source-to-target) and backward (target-to-source) transformations. A forward transformation creates a set of target elements that correspond to a given set of initial source model elements, and conversely with a backward transformation. The algorithm to derive such rules was proposed in [39] (see also [26] for the description of operational rules for other purposes). Next we present an extension of the algorithm that handles OCL attribute conditions. We will use this definition in order to derive the OCL invariants in the next section.

Definition 3 (Operational TGG Rule). *Given the declarative TGG rule $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$, the following operational rules can be derived:*

- *Forward: $\overrightarrow{p} = (r'\colon \langle R_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, \overrightarrow{ATT}_{LHS}, \overrightarrow{ATT}_{RHS})$.*
- *Backward: $\overleftarrow{p} = (r'\colon \langle L_s, L_c, R_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, \overleftarrow{ATT}_{LHS}, \overleftarrow{ATT}_{RHS})$.*

where $\overrightarrow{ATT}_{LHS}$ (resp. $\overleftarrow{ATT}_{LHS}$) contains the part of the ATT_{COND} OCL expression concerning elements of the LHS of the forward (resp. backwards) operational rule only. $\overrightarrow{ATT}_{RHS}$ (resp. $\overleftarrow{ATT}_{RHS}$) contains the part of ATT_{COND} not included in $\overrightarrow{ATT}_{LHS}$ (resp. $\overleftarrow{ATT}_{LHS}$).

The operational rules enforce the pattern given by the declarative rule, thus their RHS is equal to the RHS of the declarative rule. In the forward case, the LHS assumes that the source graph already exists, whereas in the backward case the existence of the target graph is assumed. In the rest of the paper, we use L_F and L_B to refer to the LHS of the forward and backward rules. The conditions in ATT_{COND} are split in those to be checked on the LHS before rule application ($\overleftarrow{ATT}_{LHS}$ and $\overrightarrow{ATT}_{LHS}$) and those to be checked after rule application ($\overleftarrow{ATT}_{RHS}$ and $\overrightarrow{ATT}_{RHS}$).

As an example, the upper row of Fig. 4 shows the operational forward rule derived from the declarative rule `Class-Table`. The forward rule assumes that the package `p` has a class `c` and is related to a schema `s`, and then creates a new table. The figure shows the application of the rule to the graph of Fig. 1 resulting in a triple graph H that contains a newly created table in its target. Note that this resulting target graph does not satisfy the meta-model constraints because each table needs to have at least one column. Thus, one can infer that the transformation is not total, as classes without attributes cannot be transformed into a valid model. This simple example shows the necessity of providing automatic means to check properties of rules and transformations.

Our definition of declarative rule does not use attribute computations as usual in the literature [17], but declarative attribute conditions ATT_{COND}. The advantage is that no algebraic manipulation is needed when generating the operational rules, but just to split the original conditions in those to be

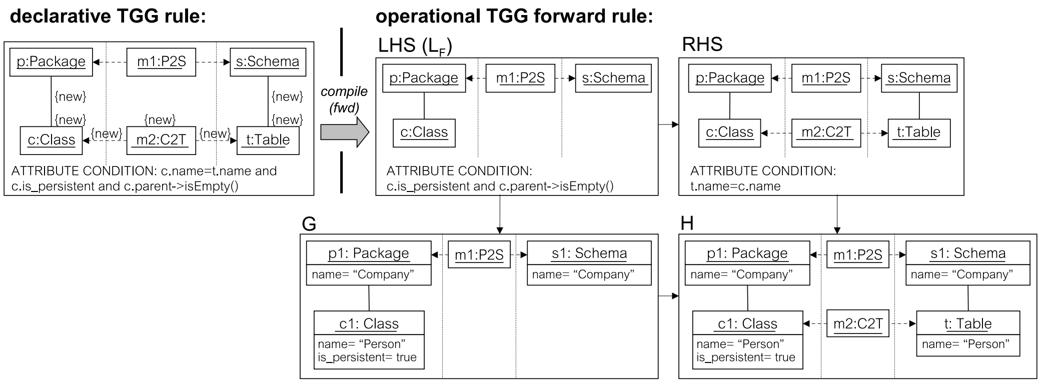


Figure 4: Derivation by operational TGG forward rule.

checked in the LHS ($\overleftarrow{ATT}_{LHS}$, $\overrightarrow{ATT}_{LHS}$) and the RHS ($\overleftarrow{ATT}_{RHS}$, $\overrightarrow{ATT}_{RHS}$). When this is implemented in practice, we can use a constraint solver to resolve attribute values. Previous approaches perform algebraic manipulation so that the attribute values for the created objects could be calculated from the ones in the LHS. For instance, in the presented example, we would have had to assign the name of the class to the name of the newly created table. Although in this case it is just an assignment, in general such algebraic manipulations present practical problems because they are difficult to automate. Note however that relying purely on constraint solving at the operational level may present computational efficiency problems in some cases. Of course, there are several tools and approaches that allow embedding OCL in normal graph grammar rules (i.e., not in TGG rules), like VMTS [29] and Fujaba [42], to express attribute conditions and computations.

Next section shows how we avoid algebraic manipulation of attribute expressions by compiling the declarative TGG rules into OCL invariants (instead of into operational rules) and using a constraint solver to actually perform and analyse the transformation. For this purpose, rules are interpreted as constraints (similar to [15]) or invariants that a pair of models should satisfy.

3. Extracting OCL Invariants from Declarative TGG Rules

Our verification method (see Section 6) is based on the analysis of the transformation model [7] derived from the transformation specification. The transformation model is made of the source and target meta-models plus the set of invariants that must hold between the source and target models in order to satisfy the transformation definition. These invariants must guarantee that the target model is a valid transformation of the source according to the set of TGG rules, and similar for the target.

In this section we present a procedure that creates invariants capturing the semantics of the TGG rules. This procedure can be regarded as a M2M transformation itself between the TGG and UML/OCL metamodels.

The invariants must ensure that each rule p is satisfied in the model. Hence, we introduce two concepts: rule *enabledness* and rule *satisfaction*. Intuitively, a declarative rule is source-enabled (resp. target-enabled) in a given graph if there exists some match of the LHS of its associated forward operational rule (resp. backward rule) in the graph.

Definition 4 (Enabledness of Rule). *Given the declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \to \langle R_s, R_c, R_t\rangle, ATT_{COND})$ *and a triple graph* G*:*

- *p is source-enabled if $\exists m\colon L_F \to G$, and $m(L_F)$ satisfies $\overrightarrow{ATT}_{LHS}$ assuming the identification of objects and links induced by m and using G as context. Given a morphism m, we write $G \vdash_{m,F} p$ if m enables p source-to-target in G.*
- *p is target-enabled if $\exists m\colon L_B \to G$, and $m(L_B)$ satisfies $\overleftarrow{ATT}_{LHS}$ assuming the identification of objects and links induced by m and using G as context. Given a morphism m, we write $G \vdash_{m,B} p$ if m enables p target-to-source in G.*

As an example, the declarative rule `Class-Table` in Fig. 4 is source-enabled in triple graph G because there is an occurrence of the LHS of its operational forward rule in G. On the contrary, the rule is not target-enabled because the LHS of the operational backward rule would have needed a table in the target graph to be matched. Hence we have $G \vdash_{m,F}$ `Class-Table` and $G \nvdash_{m,B}$ `Class-Table`. However the rule is source- and target-enabled in H.

Satisfaction of a rule involves checking both forward and backward satisfaction and is useful to ensure that two models are actually synchronized according to the rule. Intuitively, forward (resp. backward) satisfaction requires that the target (source) model satisfies the RHS of the rule for each match where the rule is source- (target-) enabled. We will use the notion of satisfaction in our algorithm to generate invariants.

Definition 5 (Satisfaction of Rule). *Given the declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \to \langle R_s, R_c, R_t\rangle, ATT_{COND})$ *and a triple graph* G*:*

- p *is forward-satisfied in* G*, written* $G \models_F p$*, if* $\forall m\colon L_F \to G$ *s.t.* $G \vdash_{m,F} p$*, then* $\exists m'\colon R \to G$ *with* $m = m' \circ r$ *s.t.* $m'(G)$ *satisfies* ATT_{COND} *assuming the identification of objects and links induced by* m *and using* G *as context.*
- p *is backward-satisfied in* G*, written* $G \models_B p$*, if* $\forall m\colon L_B \to G$ *s.t.* $G \vdash_{m,B} p$*, then* $\exists m'\colon R \to G$ *with* $m = m' \circ r$ *s.t.* $m'(G)$ *satisfies* ATT_{COND} *assuming the identification of objects and links induced by* m *and using* G *as context.*
- p *is satisfied in* G*, written* $G \models p$*, if* $G \models_F p \wedge G \models_B p$

Thus, a rule is forward-satisfied, if for each morphism where the rule is source-enabled, there is an occurrence m' of the RHS which preserves the identification of objects and links ($m = m' \circ r$) and satisfies the OCL constraints in ATT_{COND}. As an example, rule `Class-Table` in Fig. 4 is not forward-satisfied in G, but it is in H. The rule is backward-satisfied in both G and H, in the first case *trivially* because the rule is not target-enabled. As we have that $H \models_F$ `Class-Table` and $H \models_B$ `Class-Table` then we have $H \models$ `Class-Table`, or in other words, H contains two synchronized models according to `Class-Table`.

In terms of the previous definitions, the invariants to be extracted for each rule p are responsible for:

a) Locating each occurrence where p is source-enabled (see Definition 4).

b) Locating each occurrence where p is target-enabled (see Definition 4).

c) Ensuring that the elements of each occurrence found in a) and b) are connected to mapping objects according to the RHS of p.

d) Ensuring that the mapping objects connect elements that satisfy p (see Definition 5).

Next we describe our extraction procedure and the structure of the generated invariants. Our procedure makes two assumptions: (i) all rules create at least one element in the correspondence graph and (ii) each type of mapping is created by at most one rule. Given a rule, the first two steps in the procedure (see next definition) add invariants to every node n in the source or target graphs of the RHS that is connected to a newly created correspondence node m. This corresponds to the items a), b) and c) in the previous list. Step 3 in the procedure adds the invariant to every correspondence node m created by the rule (item d)).

Definition 6 (Invariant Extraction). *Given a declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$*:*

1. $\forall n \in V_{R_s}$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $cs(m) = n$*, add an invariant named* `p` *to* $type(n)$*.*
2. $\forall n \in V_{R_t}$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $ct(m) = n$*, add an invariant named* `p` *to* $type(n)$*.*
3. $\forall m \in V_{R_c} - r(V_{L_c})$*, add an invariant named* `p` *to* $type(m)$*.*

Invariant `p` in the source checks that for each occurrence where the rule is source-enabled, the rule is satisfied. According to Definition 4, the rule is source-enabled it there exists an occurrence of the LHS of the forward rule L_F satisfying the terms in $\overrightarrow{ATT}_{LHS}$. This is actually checked by a helper query operation named `p-enabled(...)`. If such query operation returns true, then the invariant `p` ensures that this occurrence is connected to all required objects needed to satisfy the rule. This is performed by a helper operation `p-mapping(...)` placed in the created correspondence node. This operation checks the object graph satisfies the structure of the RHS and the OCL constraints in ATT_{COND}, similar to Definition 5. Symmetrically, the invariant in the target ensures that each occurrence where `p` is target-enabled satisfies the rule. As both invariants are similar, we only show the structure of `p` for the source elements.

Definition 7 (Invariant for Source Elements). *Given a declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$*, then* $\forall n \in V_{R_s}$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $cs(m) = n$*, the following invariant is generated:*

```
context type(n) inv p:
 $type(n_i) :: allInstances()\text{->}forAll(n_i|$                  } $\forall n_k \in V_{L_F} - \{n\}$
    $type(n_j) :: allInstances()\text{->}forAll(n_j|...$            }
   if self.p-enabled($n_i, n_j, ...$ ) then
       $type(n_u) :: allInstances()-> exists(n_u|$                  } $\forall n_w \in V_R - r(V_{L_F})$
         $type(n_v) :: allInstances()-> exists(n_v|...$             }
            $type(m) :: allInstances()-> exists(m|...$
              $m.p\text{-}mapping(n_i, n_j, ..., n_u, n_v, ...)$ endif...))...))
```

context *type(n)::p-enabled*$(n_i :\ type(n_i),\ n_j :\ type(n_j), \ldots)$
body: $\left.\begin{array}{l} n_i.role_j{-}{>}includes(n_j) \\ and... \end{array}\right\\} \forall e \in E_{L_F}\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$
$...and\ \overrightarrow{ATT}_{LHS}$

where E_{L_F} *is the set of edges in* L_F, $role_j$ *is the role in the meta-model that allows navigating from* n_i *to* n_j. *If some edge has* n *as source or target we use the reserved word* $self$ *to refer to* n *in the expression.*

Note that the query operation `p-enabled` receives as parameters the objects in L_F, and then checks that they are connected according to L_F and that they satisfy $\overrightarrow{ATT}_{LHS}$. If association end $role_j$ has cardinality 1, then we do not use $n_i.role_j{-}{>}includes(n_j)$ but simply $n_i.role_j = n_j$. The invariant for the target elements is generated in the same way, but considering nodes $n \in V_{R_t}$ and then traversing the graph L_B. For nodes created in the correspondence graph, invariants are generated as follows.

Definition 8 (Invariant for Mappings). *Given* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$, *then* $\forall n \in V_{R_c} - r(V_{L_c})$ *the following invariant is generated:*

context *type(n)* ***inv*** *p:*
$\left.\begin{array}{l} type(n_i) :: allInstances(){-}{>}exists(n_i \mid \\ \quad type(n_j) :: allInstances(){-}{>}exists(n_j \mid ... \end{array}\right\\} \forall n_k \in V_R - \\{n\\}$
$\left. n_i.role_j{-}{>}includes(n_j)\ and... \right\\} \forall e \in r(E_L)\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$
$...and\ self.p\text{-}mapping(n_i, n_j, ...)...)...)$
context *type(n)::p-mapping*$(n_i :\ type(n_i),\ n_j :\ type(n_j), \ldots)$
body: $\left.\begin{array}{l} n_i.role_j{-}{>}includes(n_j) \\ and... \end{array}\right\\} \forall e \in E_R - r(E_L)\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$
$...and\ ATT_{COND}$

Note that the main body of the invariant checks the existence of the node in the RHS and the edges in the LHS. Then, the query operation checks the existence of the remaining edges in the RHS and the conditions in ATT_{COND}.

Let us consider the example rules in Fig. 3. From rule `Class-Table` we generate invariants for the class, the table and the `C2T` mapping, because the latter is created and connected to the class and the table. Source-enabledness is checked by the class's `Class-Table-enabled` operation, whereas actual satisfaction is checked by the `Class-Table-mapping` operation in the correspondence node. A similar invariant for the table ensures that whenever the rule is target-enabled, it is actually satisfied.

```
context Class inv Class-Table:
Package :: allInstances()->forAll(p|
  P2S :: allInstances()->forAll(m1|
    Schema :: allInstances()->forAll(s|
      if self.Class-Table-enabled(p, m1, s) then
        Table :: allInstances()->exists(t|
          C2T :: allInstances()->exists(m2|
            m2.Class-Table-mapping(p, m1, s, self, t))) endif)))
-----------------------------------------------------------------------
context Class::Class-Table-enabled(p:Package, m1:P2S, s:Schema)
body: self.package = p and m1.package = p and m1.schema = s
        and self.is_persistent and self.parent->isEmpty()
-----------------------------------------------------------------------
context C2T inv Class-Table:
Package :: allInstances()->exists(p|
  P2S :: allInstances()->exists(m1|
    Schema :: allInstances()->exists(s|
      Table :: allInstances()->exists(t|
        Class :: allInstances()->exists(c|
         m1.package = p and m1.schema = s
         and self.Class-Table-mapping(p, m1, s, c, t))))))
-----------------------------------------------------------------------
context C2T::Class-Table-mapping(p:Package, m1:P2S, s:Schema,
                                         c:Class, t:Table)
body: c.name = t.name and t.schema = s and c.package = p and self.class = c
        and self.table = t and c.is_persistent and c.parent->isEmpty()
-----------------------------------------------------------------------
context Table inv Class-Table:
Package :: allInstances()->forAll(p|
  P2S :: allInstances()->forAll(m1|
    Schema :: allInstances()->forAll(s|
      if self.Class-Table-enabled(p, m1, s) then
        Class :: allInstances()->exists(c|
          C2T :: allInstances()->exists(m2|
            m2.Class-Table-mapping(p, m1, s, c, self))) endif)))
-----------------------------------------------------------------------
context Table::Class-Table-enabled(p:Package, m1:P2S, s:Schema)
body: self.schema = s and m1.package = p and m1.schema = s
```

The generated invariant in the `Class` checks that if there is an occurrence of L_F, then there must exist a table such that the rule conditions are satisfied. The occurrence of L_F is sought by the three first nested *forAll*, which iterate to look for a package `p`, a schema `s` and a correspondence node `m1`. These elements should be connected according to L_F, and satisfy the constraints in $\overrightarrow{ATT}_{LHS}$, what is checked by the operation *Class-Table-enabled*. If such operation returns true, the invariant looks for a table `t` and a mapping `m2` connected as specified by the RHS of the rule and satisfying ATT_{COND}, what is checked by operation *Class-Table-mapping* in the mapping class `C2T`. Symmetrically, the invariant in the `Table` checks that if there is an occurrence of L_B, there is a class satisfying the rule. The invariants extracted from the other rules are shown in the Appendix.

Note that, using invariants, it is difficult to express the fact that a new table has to be created for each occurrence of L_F. Instead, we just say that a table should exist, and rely on the mapping cardinalities to ensure that indeed one is created for each class. Should we have assigned a wrong cardinality $*$ (instead of 0..1) between `C2T` and `Table`, the generated invariants would allow two classes with the same name to be related to the same table. However this is not what rule `Class-Table` expresses, which demands two different tables. By defining the right cardinality 0..1 we implement a correct translation, as each table is related to at most one `C2T` mapping, and this to exactly one class, thus ensuring that each class is related to at most one table.

We would like to remark that, both, the number of extracted invariants and the internal complexity of each invariant, are linear with respect to the number of TGG rules. Therefore, the extraction process can deal with TGGs of any size.

4. QVT-Relations

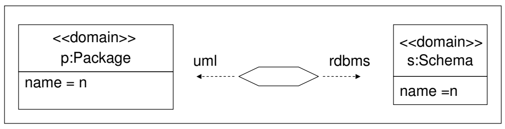
QVT-Relations is a declarative M2M transformation language part of the OMG QVT standard [34]. In this language, a bidirectional transformation consists of a set of relations between two models[2]. There are two types of relations: *top-level* and *non-top-level*. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level ones only need to hold when invoked directly or transitively from another relation.

Each relation defines two *domain patterns*, one for each model, and a pair of optional *when* and *where* OCL predicates. These optional predicates define the link with other relations in the transformation: the *when* clause indicates the constraints under which the relation needs to hold and the *where* clause provides additional conditions, apart from the ones expressed by the relation itself, that must be satisfied by all model elements in the relation.

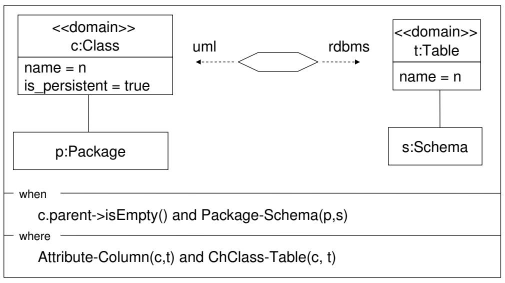
[2]Although not common, a QVT transformation could involve more than two models.

Domain patterns can be viewed as graph patterns that must be matched to a set of model elements of the appropriate type, i.e. similar to the LHS/RHS of TGGs. There are differences in the matching process, though. In TGG rules, the mappings between source and target models are explicitly represented with correspondence nodes linking elements of both models. Instead, QVT patterns may contain *variables* which can be *free* or *bound*. Bound variables and constant expressions restrict the possible matches for the pattern. Free variables become bound to the elements matching the pattern (in the execution direction of the transformation). Then, their values may be used afterwards to constrain the value of further pattern expressions. The invariants we generate will simulate this variable binding process.

Among all nodes in a domain pattern, one is marked as a root element (the one tagged with the <<domain>> stereotype). Definition of root nodes is purely for the sake of clarity (to understand a rule it is sometimes useful to pinpoint the main element in the pattern), it does not affect the semantics of the matching process. When referring to other relations in *when* or *where* clauses, parameters can be specified, and thus it is possible to pass bound variables from one relation to another. Note that the bound objects received as parameters play a similar role to the LHS of a declarative TGG rule: both are necessary preconditions to enforce the pattern.



(a) Package-Schema top-level relation.



(b) Class-Table top-level relation.

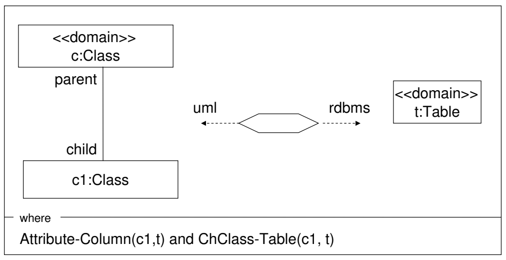
Figure 5: Top relations of the transformation.

We illustrate these concepts using the same example as in previous sections. Fig. 5(a) shows the transformation between packages and schemas. When executed in the UML→RDBMS direction, the relation states that for each package we must find a schema with the same name. Every package is a match for the source domain pattern (the pattern consists of a single node of type `Package` with no required additional links or constraints). Given a matching package p, the variable n becomes bound to the name of the package and it is used in the target domain pattern to ensure that a schema node named n exists. Likewise, when executed in the RDBMS→UML direction, the relation states that for each schema we must find a package with the same name (now n is bound to the name of the schema and restricts the packages that may satisfy the relation).

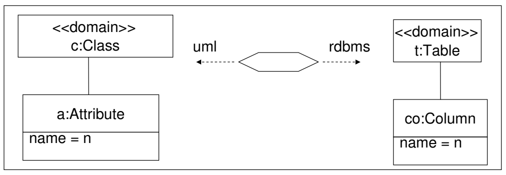
Relation `Class-Table` (Fig. 5(b)) describes the transformation between classes and tables. For each persistent class (condition $is\_persistent = true$), we need a table with the same name. The *when* clause imposes two additional constraints that restrict its application: to be a match, the class must be a root class ($c.parent \rightarrow isEmpty()$) and the package and schema of the matching class and table must satisfy the previous `Package-Schema` relation. Finally, the *where* clause imposes additional conditions to each pair of class c and table t satisfying the patterns and the *when* clause: all attributes of c must match the columns in t as expressed in the relation `Attribute-Column` (Fig. 6(b)) and children classes of c must also be mapped to t. This latter condition is defined by means of the recursive relation `ChClass-Table` (Fig. 6(a)). When executing this relation, variables c and t are bound to the argument values passed on when calling this relation from `Class-Table`. For each child class $c1$ of c, attributes of $c1$ are mapped to columns of t and, recursively, the process continues with the children classes of $c1$. This recursion stops when a class without children is reached.

Finally, relation `Attribute-Column` defines the mappings between attributes and columns. Again, variables c and t are bound to the arguments used when calling the relation from the `Class-Table` or `ChClass-Table` relations. For each attribute of c the relation requires t to have a column with the same name in the UML→RDBMS direction. When following the opposite direction, c is required to have an attribute for each column in t, with the same name.

Optionally, we can define keys (i.e. unique identifiers) for the model elements participating in a relation. For instance, *key Table* {*schema, name*}*;* declares that we cannot have two tables with the same name within the same schema. Keys are used to avoid unnecessary object creations during the transformation. Before creating a new object in the target model, the key is used to locate an existing matching object. The new object is only created if a match is not found.



(a) ChClass-Table relation.



(b) Attribute-Column relation.

Figure 6: Non-top-level relations of the transformation.

5. Extracting OCL Invariants from QVT-Relations

Many QVT concepts resemble the elements appearing in TGG rules (see [20] for a comparison). Therefore, the procedure for extracting the implicit invariants in a QVT-Relations transformation is very similar to that explained in Section 3 for TGGs. Note that it is also similar to the formalization of the QVT-Relations language in terms of the QVT-Core language defined in the QVT standard. For both TGGs and QVT, we have to check that if a rule is source-enabled (or target-enabled) the mapping conditions of the relation are satisfied. The difference is that now these mapping conditions are not specified in the correspondence nodes (since they do not exist) but must be integrated in the invariants defined for the source and target elements. A second difference is the existence of the *when* and *where* clauses in QVT. When translating the rules, the *when* clause will be part of the enabling conditions whereas the *where* clause will be added to the mapping conditions.

Definition 9 (Top-relation Invariant). *Let* p *be a top-relation with domain patterns* $S = \{root_s, s_1, \ldots, s_n\}$ *and* $T = \{root_t, t_1, \ldots, t_m\}$*, and let* $T_{when} \subseteq T$ *be the set of elements of* T *referenced in* p*'s "when" section. Then, the following invariant is generated for the* $S \to T$ *direction:*

```
context $type(root_s)$ inv p:
    $\left.\begin{array}{l} type(x_i) :: allInstances()\text{->}forAll(x_i| \\ \quad type(x_j) :: allInstances()\text{->}forAll(x_j| \ldots \end{array}\right\} \forall x_k \in (S \setminus \{root_s\}) \cup T_{when}$
        if $self.p\text{-}enabled(x_i, x_j, ...)$ then
            $\left.\begin{array}{l} type(x_u) :: allInstances()- > exists(x_u| \\ \quad type(x_v) :: allInstances()- > exists(x_v|... \end{array}\right\} \forall x_w \in T \setminus T_{when}$
                $self.p\text{-}mapping(x_i, x_j, \ldots, x_u, x_v, \ldots) \ldots))$ endif $\ldots))$
```

```
context type(root_s)::p-enabled(x_i : type(x_1), ...)
 body:  when and enabling-conditions
-----------------------------------------------------------
context type(root_s)::p-mapping(x_i : type(x_i), ...)
 body:  where and mapping-conditions
```

where the OCL expressions corresponding to the enabling and mapping conditions (i.e. the expressions ensuring that the links among the pattern objects and the attribute conditions are satisfied) are derived following the same procedure described for TGG rules.

As in the case of TGG rules, notice the distinction between nodes universally quantified (firsts part of the rule) and nodes with an existential quantifier in the template (second part). Nodes with the universal one (including the implicit one represented by the *self* variable) are used when checking matches for the rule since, for every match, we must verify that the relation holds. This is done by checking if there exists a set of nodes in the target pattern satisfying the mapping conditions (i.e. we do not need all possible nodes in the target pattern to satisfy the conditions, only one, that is why we use existential quantifiers for those nodes). Therefore, we add one universal quantifier for each node in S plus for each node in T referenced in the *when* clause. We use an existential quantifier for the rest of nodes in T. The *if* clause separating both sets of nodes ensures that the existence of the mapping nodes in the target pattern is only enforced when the relation is enabled.

Applying this definition on the `Class-Table` QVT relation in Fig. 5(b) generates the following invariants and auxiliary query operations:

```
context Class inv Class-Table:
   Package :: allInstances()->forAll(p|
     Schema :: allInstances()->forAll(s|
       if self.Class-Table-enabled(p, s) then
         Table :: allInstances()->exists(t|
           self.Class-Table-mapping(p, t, s)) endif))
-----------------------------------------------------------
context Class::Class-Table-enabled(p:Package, s:Schema)
 body:  self.package = p and self.parent->isEmpty()
        and self.is_persistent = true and p.Package-Schema-mapping(s)
-----------------------------------------------------------
context Class::Class-Table-mapping(p:Package, t:Table, s:Schema)
 body:  self.name = t.name and t.schema = s and
        self.Attribute-Column(t) and self.ChClass-Table(t)
```

```
context Table inv Class-Table:
  Package :: allInstances()−>forAll(p|
    Schema :: allInstances()−>forAll(s|
      if self.Class-Table-enabled(p, s) then
        Class :: allInstances()−>exists(c|
          self.Class-Table-mapping(c, p, s)) endif))
```

```
context Table::Class-Table-enabled(p:Package, s:Schema)
 body: self.schema = s and s.Package-Schema-mapping(p)
```

```
context Table::Class-Table-mapping(c:Class, p:Package, s:Schema)
 body: self.name = c.name and c.package = p and
       c.is_persistent = true and c.parent− > isEmpty() and
       self.Attribute-Column(c) and self.ChClass-Table(c)
```

In the $UML \rightarrow RDBMS$ direction, `Class-Table` checks that for every package-schema combination that enables the relation source-to-target there exists a table that satisfies the class mapping conditions. Note that, for this invariant, the `Schema` class is universally quantified since it is needed to evaluate the *when* clause of the rule. In fact, for this rule, the `Class-Table-enabled` operation must check not only whether the class object represented by the *self* root variable enables the relation but also that the *when* clause evaluates to true on it. Therefore, `Class-Table-enabled` checks that the class is persistent, has no parents and is related to a package satisfying the `Package-Schema` relation. The latter is checked by means of a call to the `Package-Schema-mapping` operation defined as part of the translation of the `Package-Schema` relation (see the appendix).

If a class object satisfies all these conditions then the invariants require, as stated by the `Class-Table-mapping` operation, the existence of a table t with the same name as the class (and under the schema related to the class package) and that the non-top relations `Attribute-Column` and `ChClass-Table` hold between the class and the table. The if-then condition in the invariant ensures that this is only required for matches enabling the relation source-to-target.

Non-top relations are processed similarly to top relations with two main differences: (i) non-top relations are translated as boolean operations instead of invariants, because non-top relations only need to hold when called from a top relation, and (ii) nodes passed as parameters are not quantified. For simplicity, we make the assumption that if some relation p is called from the when clause of two different relations, it receives the same parameters. Otherwise, for each different call, we would have to generate a different operation for p with different parameters.

Definition 10 (Non-top Relation Invariant). *Let* p *be a non-top relation with domain patterns* $S = \{root_s, s_1, \ldots, s_n\}$ *and* $T = \{root_t, t_1, \ldots, t_m\}$*. Let* $T_{when} \subseteq T$ *be the set of elements of* T *referenced in* p*'s "when" section and* $P = \{a_1, ..., a_k\} \subseteq S \cup T$ *the set of elements passed as parameters in the call to* p *from other relations. The following boolean operation for the* $S \rightarrow T$ *transformation direction is generated:*

```
context type(root_s)::p(a_1 : type(a_1), ..., a_k : type(a_k))
   type(x_i) :: allInstances()->forAll(x_i|                  } ∀x_k ∈ (S\{root_s}\P)∪T_when
     type(x_j) :: allInstances()->forAll(x_j| ...            }
       if self.p-enabled(x_i, x_j, ...) then
         type(x_u) :: allInstances()- > exists(x_u|          } ∀x_w ∈ T \ T_when \ P
           type(x_v) :: allInstances()- > exists(x_v|...     }
             self.p-mapping(x_i, x_j, ..., a_1, ..., a_k, x_u, x_v, ...) ...)) endif ...))
```

Notice that the operation has the same structure as the top-level invariant, but we eliminate from the body the variables passed as parameters. As an example, we provide the translation of non-top relations `Attribute-Column` and `ChClass-Table` in the appendix.

Finally, for each key defined in a relation, we generate an additional constraint ensuring that no two objects with the same key exist in the model.

Definition 11 (Key Invariant). *Given key* k *defined as Key* $X\{prop_1, \ldots, prop_n\}$ *where* X *is the type of one of the model elements participating in the relation and* $prop_i$ *is a property (attribute or association end) of* X *the following invariant is generated :*

```
context X inv k:
   X :: allInstances()->forAll(x_1, x_2|x_1 <> x_2 implies
     (x_1.prop_1 <> x_2.prop_1 or ... or x_1.prop_n <> x_2.prop_n))
```

The invariant forces two different objects to have at least a different value in one of the properties that are part of the key. Notice that we do not generate such invariant if it is already part of the meta-model constraints.

6. Analysing the Extracted Invariants

The analysis of the OCL invariants extracted from a transformation specification can reveal insightful information regarding its correctness. In this section, we show how this analysis can be applied to two problems: (i) Verification of correctness properties of transformations, that is, finding defects in them, e.g. whether they are underspecified; and (ii) Validation of transformations, that is, identifying transformations whose definition does not match the designer intent.

A key notion in our analysis will be the *transformation model*: the union of the source and target meta-models, including their integrity constraints, together with the extracted OCL transformation invariants. The goal of this representation is leveraging existing UML/OCL verification and validation tools for the analysis of model transformations. For example, there are several tools addressing the *consistency* or *satisfiability* problem for UML/OCL models [2, 10, 13, 43]: given a UML class diagram annotated with OCL constraints, decide whether there exists a legal instance of the model, satisfying all graphical and OCL constraints. Several approaches to this problem proceed constructively by automatically computing the legal instance, which is provided to designers as output of the tool. As we will discuss in this section, many interesting problems on transformations can be reformulated as consistency problems on the transformation model.

6.1. Tool support

The verification and validation of M2M transformations can be performed using existing tools for the analysis, reasoning, verification and validation of UML/OCL specifications. Candidate tools should be able to manipulate UML class diagrams, admit the definition of OCL queries and invariants, and provide some support for checking the OCL invariants (either interactively or automatically through a formal analysis or proof). There are several approaches in the literature meeting these requirements [2, 10, 13, 36]. Each tool relies on a different approach for the analysis of OCL constraints, e.g. theorem proving, SAT solving and constraint programming. No “best” approach exists as each one achieves a different trade-off in terms of *expressiveness* (set of supported UML/OCL constructs), *efficiency* (time and memory required to compute the result), *decidability* (termination of the method for any input), *completeness* (whether there are inputs for which the output is inconclusive) and *automation* (without user intervention).

In order to use any of these tools, the designer needs to define the transformation model in an input format supported by the tool. Depending on the tool, the meta-models are provided as input either in a textual notation, through a GUI for drawing the model, or an XMI file which can be exported from a UML case tool. Meanwhile, OCL invariants are described in a text file. Then, all these tools are capable of analysing a UML class diagram annotated with OCL constraints and deciding whether it is consistent, i.e. whether there is a set of objects from the classes in the diagram which satisfy the cardinality and integrity constraints. Tools based on SAT solvers (UML2Alloy [2]) or constraint solvers (UMLtoCSP [13]) prove consistency by finding a specific instance which satisfies all constraints. Meanwhile, methods based on theorem provers (HOL-OCL [10], CQC [36], Description Logics [43]) combine deduction rules of logics to construct a proof. In case the proof exists, it can also lead back to a satisfying instance. Other tools, such as the USE validation environment [19], allow defining and validating UML/OCL specifications. In USE, designers can define a (meta-)model, an instantiation of it and check whether the invariants hold on that particular instance. This testing helps the designer to detect if the meta-model is overconstrained (valid states in the domain are forbidden by the invariants) or underconstrained (invalid states are allowed by the invariants). This approach exchanges automation for interactivity.

With these inputs, verification tools provide mechanisms to automatically check the consistency of the transformation model without user intervention. Checking consistency allows the verification of the executability of the transformation and the use of all validation scenarios. Other properties checked automatically by UML/OCL analysis tools (e.g. redundancy of an invariant) lead to the verification of other properties in section 6.2 (verification).

The set of OCL expressions supported by these tools varies, e.g. some tools do not support arithmetic operations in OCL. Therefore, choosing the right tool will depend, for instance, in the type of expressions used in attribute conditions.

A problem shared by all these tools is that the analysis of UML/OCL diagrams has a high computational complexity. Reasoning on UML class diagrams is already EXP-complete without considering OCL constraints [6], and undecidable when arbitrary OCL constraints are included in the problem. Decision procedures which can support complex OCL invariants like the transformation invariants from Sections 3 and 5 have at least an exponential worst-case execution time (and also they may be undecidable and/or incomplete). This complexity places a limit on the scalability of the proposed approach, i.e. the size of the transformations that can be analyzed.

From this portfolio of tools, the examples shown in this section use the tool UMLtoCSP, which is based on a constraint logic programming solver.

However, the other tools could be used instead with a different trade-off.

The figures in this section will alternatively depict examples of our TGG and QVT running examples. As both describe the same transformation, TGGs and QVT examples will be equal with the only difference that mapping nodes are explicit in TGGs. For the sake of brevity, in some cases we will refer only to TGGs or QVT, even though the verification and validation techniques can be applied to both languages: it is only a matter of whether we consider the OCL invariants from TGGs or QVT, as they rely in the same transformation model concept.

6.2. Verification of Model-to-Model Transformations

The verification of transformations answers the question "*is the transformation right?*", i.e. are there any defects in the transformation? This verification problem can be expressed in terms of the transformation model because, like any other model, it is expected to satisfy several reasonable assumptions. For instance, it should be possible to instantiate the model in some way that does not violate any integrity constraint, including the OCL invariants of the meta-models and the transformation rules. Failing to satisfy these criteria may be a symptom of an incomplete, over-constrained or incorrect model, reflecting potential defects in the original M2M transformation.

In this section we formalize some properties that can be used to study quality notions of M2M transformations. These quality notions capture static properties of the M2M transformation, that is, they consider the application of the transformation to specific source and target models rather than studying the evolution of the model (e.g. incremental transformation or model synchronization).

We introduce a particular notation in order to keep the formalization independent of the language employed for the transformation specification and the approach used for analysis. However, the predicates that we will define have a direct correspondence with the invariants extracted in Sections 3 and 5 for TGGs and QVT.

- S and T denote a source and a target model respectively.
- $\langle S, T\rangle$ is used for a pair of related source and target models.
- r denotes a rule or relation[3], where we write $\mathsf{PRE}_\mathsf{r}^\mathsf{Fwd}$, $\mathsf{PRE}_\mathsf{r}^\mathsf{Bwd}$ to de-

[3]In the following, we use rule and relation interchangeably.

note its forward and backward pre-conditions, and $\mathsf{POST_r}$ its post-conditions. In our transformation model, $\mathsf{PRE_r^{Fwd}}$ and $\mathsf{PRE_r^{Bwd}}$ correspond to the generated OCL queries `p-enabled` presented in Definitions 7 and 9 for TGGs and QVT respectively, whereas $\mathsf{POST_r}$ corresponds to the complete generated invariant.

- TS denotes a M2M specification made of a set of rules or relations.

We also use the auxiliary function $\mathsf{OCC}(\_\,,\_)$ that returns all occurrences of the first argument (a pair of related models with a set of constraints) into the second (a pair of related models). The following predicates will be used to define verification properties, where graphs G and H used as examples can be found in Fig. 7:

- $\mathsf{INV}[S]$ holds if S is conformant to its meta-model. Similarly, $\mathsf{INV}[T]$ holds if T is conformant to its meta-model.
- $\mathsf{INV}[\langle S, T\rangle] \overset{def}{=} \mathsf{INV}[S] \wedge \mathsf{INV}[T]$.
- $\langle S, T\rangle \subseteq \langle S', T'\rangle$ holds if $\langle S, T\rangle$ is a submodel of $\langle S', T'\rangle$.
- $\langle S, T\rangle = \langle S', T'\rangle$ holds if $\langle S, T\rangle$ is isomorphic to $\langle S', T'\rangle$, i.e. both models are equal up to equality of object identifiers.

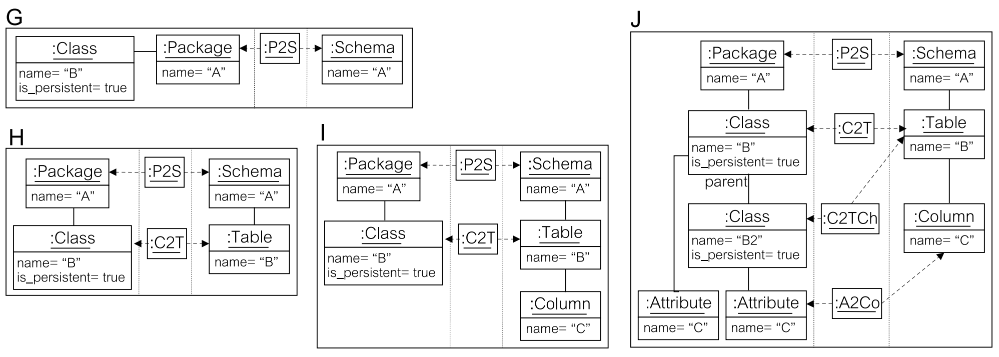


Figure 7: Example triple graphs for the verification of rule properties: (1) G is an example of forward applicability for rule `Class-Table`; (2) H is an example of forward weak executability for rule `Class-Table`; (3) I is an example of executability for rule `Class-Table`; (4) J is a counterexample of strong executability for rule `Attribute-Column`.

- $\mathsf{EN}_\mathsf{r}^\mathsf{Fwd}[\langle S,T\rangle] \overset{def}{=} \mathsf{OCC}(\mathsf{PRE}_\mathsf{r}^\mathsf{Fwd}, \langle S,T\rangle) \neq \emptyset$, i.e. r is source-enabled if there is some occurrence of its forward pre-condition. For TGG rules, this predicate corresponds to Definition 4. For example, $\mathsf{EN}_\mathsf{Class-Table}^\mathsf{Fwd}[G]$ holds because there is one occurrence of $\mathsf{PRE}_\mathsf{Class-Table}^\mathsf{Fwd}$ in G.

- $\mathsf{EN}_\mathsf{r}^\mathsf{Bwd}[\langle S,T\rangle] \overset{def}{=} \mathsf{OCC}(\mathsf{PRE}_\mathsf{r}^\mathsf{Bwd}, \langle S,T\rangle) \neq \emptyset$. For example, $\mathsf{EN}_\mathsf{Class-Table}^\mathsf{Bwd}[G]$ does not hold because there is no occurrence of $\mathsf{PRE}_\mathsf{Class-Table}^\mathsf{Bwd}$ in G, but $\mathsf{EN}_\mathsf{Class-Table}^\mathsf{Bwd}[H]$ holds.

- $\mathsf{SAT^*}_\mathsf{r}[\langle S,T\rangle] \overset{def}{=} (\forall\langle S',T'\rangle \in \mathsf{OCC}(\mathsf{PRE}_\mathsf{r}^\mathsf{Fwd}, \langle S,T\rangle)\cup\mathsf{OCC}(\mathsf{PRE}_\mathsf{r}^\mathsf{Bwd}, \langle S,T\rangle) : \exists\langle S'',T''\rangle \in \mathsf{OCC}(\mathsf{POST}_\mathsf{r}, \langle S,T\rangle) : \langle S',T'\rangle \subseteq \langle S'',T''\rangle)$. This predicate holds when a pair of models satisfies the post-conditions of a rule in all occurrences of its pre-conditions, which may be zero (trivial satisfaction). In such a case we say that the models satisfy the rule, which corresponds to Definition 5 for TGG rules. For example, $\mathsf{SAT^*}_\mathsf{Class-Table}[H]$ holds because the only occurrences of $\mathsf{PRE}_\mathsf{Class-Table}^\mathsf{Fwd}$ and $\mathsf{PRE}_\mathsf{Class-Table}^\mathsf{Bwd}$ are satisfied (i.e. included in an occurrence of $\mathsf{POST}_\mathsf{Class-Table}$). For QVT it is similar, but in addition the *when* and *where* clauses may imply the satisfaction of other relations.

- $\mathsf{SAT^*}_\mathsf{TS}[\langle S,T\rangle] \overset{def}{=} (\forall r \in TS : \mathsf{SAT^*}_\mathsf{r}[\langle S,T\rangle])$. This predicate holds if $\langle S,T\rangle$ satisfies (even trivially) all rules in the specification TS.

- $\mathsf{SAT}_\mathsf{r}[\langle S,T\rangle] \overset{def}{=} \mathsf{SAT^*}_\mathsf{r}[\langle S,T\rangle] \wedge (\mathsf{EN}_\mathsf{r}^\mathsf{Fwd}[\langle S,T\rangle] \vee \mathsf{EN}_\mathsf{r}^\mathsf{Bwd}[\langle S,T\rangle])$. This predicate holds when a pair of models satisfies r's post-conditions, but not trivially (i.e. at least one occurrence exists). For example, $\mathsf{SAT}_\mathsf{Class-Table}[H]$ holds.

- $\mathsf{SAT}_\mathsf{TS}[\langle S,T\rangle] \overset{def}{=} (\forall r \in TS : \mathsf{SAT}_\mathsf{r}[\langle S,T\rangle])$.

Once established the notation and necessary predicates, we are ready to define the list of quality properties of M2M transformations at two levels: considering the role of individual rules within a transformation, or considering the transformation model as a whole. In addition, some properties can be studied at both levels. We start with properties applicable to the level of rules. We assume the forward direction, but it should be clear that the same properties can be easily defined for the backward direction. Each property contains a description, its formula in terms of the previous notation, and an example. The graphs used as examples can be found in Fig. 7 (for QVT the examples would be similar but assuming an empty correspondence graph).

Applicable: r is *forward applicable* if there is a pair of models where r is source-enabled and the source model satisfies its meta-model constraints. We do not ask the target model to satisfy its meta-model constraints, as they may be violated during the transformation (e.g. lower cardinality constraints in associations).

Formula: $\exists \langle S, T \rangle : \mathsf{INV}[S] \wedge \mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S, T \rangle]$.

Example. Rule `Class-Table` is forward applicable in G because there is one occurrence of $\mathsf{PRE}_{\mathsf{Class-Table}}^{\mathsf{Fwd}}$ and the source graph is a valid model.

Weak Executable: r is *forward weak executable* if there exists a pair of models that satisfy r, and the source is a valid model.

Formula: $\exists \langle S, T \rangle : \mathsf{INV}[S] \wedge \mathsf{SAT}_{\mathsf{r}}[\langle S, T \rangle]$.

Example. Rule `Class-Table` is forward weak executable because H contains one occurrence of $\mathsf{POST}_{\mathsf{Class-Table}}$. Please note that the target graph of H is not a valid model, as tables need at least one column. However this condition is not demanded by the property.

Executable: r is *executable* if there exists a valid pair of models that satisfy it. Note that this property is independent of the direction.

Formula: $\exists \langle S, T \rangle : \mathsf{INV}[\langle S, T \rangle] \wedge \mathsf{SAT}_{\mathsf{r}}[\langle S, T \rangle]$.

Example. Rule `Class-Table` is executable because graph I contains one occurrence of $\mathsf{POST}_{\mathsf{Class-Table}}$ and its source and target graphs are valid models.

Strong Executable: r is *forward strong executable* if the target of every source model where r is source-enabled can be completed to satisfy r.

Formula: $\forall \langle S, T \rangle : \mathsf{INV}[\langle S, T \rangle] \wedge \mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S, T \rangle] \Rightarrow \exists T' : (\mathsf{SAT}_{\mathsf{r}}[\langle S, T \rangle] \vee$
$(\mathsf{INV}[\langle S, T' \rangle] \wedge \mathsf{SAT}_{\mathsf{r}}[\langle S, T' \rangle] \wedge \langle S, T \rangle \subseteq \langle S, T' \rangle))$.

Example. Rule `Attribute-Column` is not forward strong executable because, as the counterexample triple graph J shows, a class diagram where two classes related through inheritance define two attributes with same name cannot be translated into a valid target model. On the other hand `Package-Schema` is strong executable.

Remark. These three forms of executability demand increasing levels of satisfiability for a given rule. While for weak executability and executability r has to be existentially satisfied (in the latter by some valid target model), in the strong version it must be universally satisfied in all cases where it can be source-enabled. Note that a rule is executable if and only if it is weak executable both forwards and backwards.

Total: r is *total* if it is not trivially satisfiable in every valid source model. This is equivalent to ask r to be forward weak executable in each valid source model.

Formula: $\forall S : \mathsf{INV}[S] \Rightarrow \exists T : \mathsf{SAT_r}[\langle S, T\rangle]$.

Example. Rule `Package-Schema` is not total, because the empty model satisfies the source meta-model constraints, but there is no target model that together with it satisfies the rule. The rule would be total should we add to the meta-model a constraint asking each model to have at least one package.

Deterministic: r is *deterministic* if each valid source model can be correctly transformed in a unique way using r.

Formula: $\forall S, T, T' : \mathsf{INV}[S] \Rightarrow (\mathsf{EN_r^{Fwd}}[\langle S, T\rangle] \wedge \mathsf{EN_r^{Fwd}}[\langle S, T'\rangle] \wedge$
$\mathsf{SAT^*{}_{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT^*{}_{TS}}[\langle S, T'\rangle] \Rightarrow$
$T = T')$.

Example. All rules in our example are deterministic. In general, this property can be used to detect under-constrained transformation models. For example, should we change the cardinality of the mapping between `C2T` and `Table` from 0..1 to $*$, then rule `Class-Table` would be non-deterministic because two classes with the same name could be mapped to the same table or to two tables with the same name. A rule could also fail to be deterministic due to attribute computation (e.g. if an attribute value is set to be the square root of another), or because the target model contains elements not mentioned in the transformation. For instance, if the relational schema meta-model had e.g. foreign key nodes, as these are not considered by any rule, there could be target models with and without foreign keys associated to a unique source model.

Finally note that the formula demands $\langle S, T\rangle$ and $\langle S, T'\rangle$ to satisfy (even trivially) the whole transformation specification TS. Otherwise no rule in our example would be deterministic. See for example H and I which satisfy `Package-Schema` and have the same source model.

Functional: r is *functional* if it is total and deterministic.

Formula: $Total(r) \wedge Deterministic(r)$.

Example. No rule in our example is functional. Rule `Package-Schema` would be functional if we demand at least one package in each UML model.

Exhaustive: r is *exhaustive* if it is satisfiable in each target model. This property is the dual of property *total*, and is equivalent to ask r to be weak executable in each valid target model.

Formula: $\forall T : \mathsf{INV}[T] \Rightarrow \exists S : \mathsf{SAT}_\mathsf{r}[\langle S, T\rangle]$.

Example: No rule in the example is exhaustive. Rule `Package-Schema` is not exhaustive because there is no source model that together with an empty target model satisfies the rule. The rule would be exhaustive should we add a constraint to the target meta-model asking for at least one schema in each RDBMS model.

Injective: r is *injective* if each valid target model is a correct transformation of a unique source model. This property is the dual of *deterministic* for the source model.

Formula: $\forall T, S, S' : \mathsf{INV}[T] \Rightarrow (\mathsf{EN}_\mathsf{r}^\mathsf{Fwd}[\langle S, T\rangle] \wedge \mathsf{EN}_\mathsf{r}^\mathsf{Fwd}[\langle S', T\rangle] \wedge$
$\mathsf{SAT}^*{}_\mathsf{TS}[\langle S, T\rangle] \wedge \mathsf{SAT}^*{}_\mathsf{TS}[\langle S', T\rangle] \Rightarrow$
$S = S')$.

Example: Rule `Class-Table` is not injective. We can find the counterexample graphs K and L shown in Fig. 8 which have the same target, and one is produced from a class with two attributes, and the other from two classes related through inheritance with one attribute each. On the contrary, rule `Package-Schema` is injective.

Bijective: r is *bijective* if it is exhaustive and injective.

Formula: $Exhaustive(r) \wedge Injective(r)$.

Example. No rule in the example is bijective. Rule `Package-Schema` would be bijective should we forbid empty source and target models.

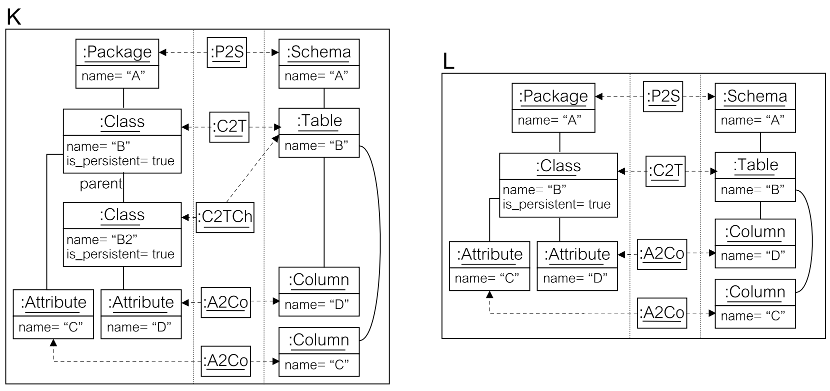


Figure 8: Example triple graphs for the verification of transformation properties: our TGG is executable but non-injective (we find two source models for the same target model).

Redundant: r is *redundant* in a specification TS if the set of pairs of models satisfying TS is exactly the same as those satisfying $TS \setminus \{r\}$.

Formula: $\forall\langle S,T\rangle : \mathsf{INV}[\langle S,T\rangle] \Rightarrow (\mathsf{SAT}^*{}_{\mathsf{TS}\setminus\{\mathsf{r}\}}[\langle S,T\rangle] \Leftrightarrow \mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S,T\rangle])$.

Example. None of the rules in the example specification are redundant. An example of redundant rule would be one like `Attribute-Column`, but applicable only to persistent classes.

Enabledness Subsumption: Given two rules r_1 and r_2, r_1 *forward subsumes* r_2, written $r_1 \leq_F r_2$, if whenever r_2 is source-enabled so is r_1. That is, the forward pre-conditions of r_1 are weaker than those of r_2.

Formula: $\forall\langle S,T\rangle : \mathsf{INV}[S] \wedge \mathsf{EN}^{\mathsf{Fwd}}_{\mathsf{r2}}[\langle S,T\rangle] \Rightarrow \mathsf{EN}^{\mathsf{Fwd}}_{\mathsf{r1}}[\langle S,T\rangle]$.

Example. We have that `Package-Schema` $\leq_F$ `Class-Table`, as whenever the latter is source-enabled, so is the former. Note that if a QVT relation r calls relations $r_1, ..., r_n$ in the where clause, we should have $r \leq_F r_1 \wedge ... \wedge r \leq_F r_n$ if the relation is to be enforced in the forward direction. Similarly, if relation r calls relations $r_1, ..., r_n$ in the when clause, we should have $r_1 \leq_F r \wedge ... \wedge r_n \leq_F r$ if the relation is to be enforced in the forward direction. Note that if the relation is meant to be bi-directional, we would have backward subsumption too.

Next, we generalize some of the presented properties to the level of transformation. In this case all properties are independent of the direction.

Executable: TS is *executable* if there is a valid pair of models satisfying it.

Formula: $\exists \langle S, T\rangle : \mathsf{INV}[\langle S, T\rangle] \wedge \mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T\rangle]$.

Example. Our example TGG transformation is executable because, e.g. graphs K and L satisfy (trivially or not) all rules. Note that we use the $\mathsf{SAT^*}_{\mathsf{TS}}[\ldots]$ predicate instead of $\mathsf{SAT}_{\mathsf{TS}}[\ldots]$ because otherwise we would be requiring at least one explicit occurrence of each rule.

Total: TS is *total* if for each valid source model there is a valid target model satisfying it.

Formula: $\forall S : \mathsf{INV}[S] \Rightarrow \exists T : \mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{INV}[T]$.

Example. Our TGG is not total as e.g. there is no valid target model such that together with the source model of graph G satisfies the specification (tables without columns are not allowed).

Deterministic: TS is *deterministic* if each valid source model can be correctly transformed in a unique way.

Formula: $\forall S, T, T' : \mathsf{INV}[S] \Rightarrow (\mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T'\rangle] \wedge$
$\mathsf{INV}[T] \wedge \mathsf{INV}[T'] \Rightarrow T = T')$.

Example. Our TGG is deterministic because its rules are deterministic.

Functional: TS is *functional* if it is total and deterministic.

Formula: $Total(TS) \wedge Deterministic(TS)$.

Example. Our specification is not functional because it is not total.

Exhaustive: TS is *exhaustive* if each target model can be produced from some source model. This property is the reciprocal of property *total*.

Formula: $\forall T : \mathsf{INV}[T] \Rightarrow \exists S : \mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{INV}[S]$.

Example: Our TGG is exhaustive as each valid relational schema can be generated from some class diagram.

Injective: TS is *injective* if each target model satisfies TS together with just one single source model. This property is the reciprocal of *deterministic.*

Formula: $\forall T, S, S' : \mathsf{INV}[T] \Rightarrow (\mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT^*}_{\mathsf{TS}}[\langle S', T\rangle]$
$\mathsf{INV}[S] \wedge \mathsf{INV}[S'] \Rightarrow S = S')$.

Example: Our TGG is not injective as graphs K and L show.

Bijective: TS is *bijective* if it is exhaustive and injective.

Formula: $Exhaustive(TS) \wedge Injective(TS)$.

Example. Our specification is not bijective because it is not injective.

All these properties can be encoded as UML/OCL consistency problems on the transformation model. For example, executability of the transformation is directly equivalent to the consistency problem, i.e. a transformation is executable iff its transformation model is satisfiable. As an example, Fig. 9 illustrates the verification of the executability property on our running example for QVT, using the tool UMLtoCSP [13] for UML/OCL model consistency. The tool automatically proves the property by finding a legal pair of source and target models satisfying the transformation model. Any other tool among those mentioned in Section 6.1 could be used instead.

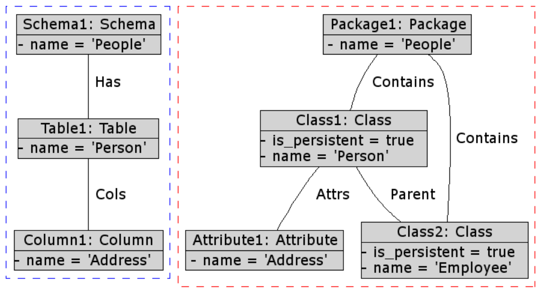


Figure 9: Verification of executability of the QVT transformation, using UMLtoCSP.

To compute this result, we proceed in the following way. First, the transformation model is modelled using a UML CASE tool capable of exporting UML class diagrams in XMI format (XML Metadata Interchange), for example Argo UML[4]). Then, the OCL transformation invariants are written in a text file. Both the XMI file and the text file are provided as the input to the UMLtoCSP tool. Next, UMLtoCSP requires the selection of the property that will be verified on the UML/OCL class diagram. In the case of executability, we are interested in checking whether there is a pair of source and target models which satisfy all meta-model and transformation invariants. Furthermore, we are probably interested in *non-empty* source and target

[4]`http://argouml.tigris.org/`

models. This property, i.e. ensuring that there exists a non-empty instance satisfying all the invariants of the model, is called *weak satisfiability* of the class diagram [13]. Weak satisfiability can be proved by computing an instance of the model which satisfies all the invariants, e.g. an example of the property. Given the input model and the invariants, the tool UMLtoCSP is able to find this legal instance (both the source and target models) automatically without any user intervention. The output provided by UMLtoCSP is a graphical representation of the instance expressed as a UML object diagram like that of Figure 9.

Other verification properties have to be decomposed into two or more consistency problems affecting either only the source model, only the target model, or the entire transformation model. For example, we can prove that a transformation is not total if we find a counterexample, i.e. a legal instance of the source model with no corresponding instance in the target model. To find the counterexample, first we generate a legal instance x of the source model. Then, we check if the entire transformation model is consistent when an additional invariant is added: the source model must be instantiated to x. If it is inconsistent, we have found our counterexample, otherwise, we keep generating new instances for x until we find our counterexample or we conclude no counterexample exists. A similar procedure can be used to check the other properties.

If we are using a bounded verification tool like UMLtoCSP to generate legal instances, the search for counterexamples is limited to a bounded space. The designer defines this space by establishing the set of possible values for attributes and upper bounds to the number of objects and links to be considered. Bounding the search space ensures that the approach terminates (the tool always provides some answer) but as a consequence it becomes *incomplete* (when no counterexample is found, the result is inconclusive: there may be a counterexample outside the bounded search space). An advantage of this approach is that there is no restriction on the constructs and operators that can be used in meta-model invariants and attribute conditions. For example, an attribute condition of a rule might require factoring a number into its list of prime factors. This type of complex attribute conditions is supported by UMLtoCSP, even though its analysis might be inefficient. UMLtoCSP does not reason on the invariants or attribute conditions directly: instead, it attempts to build an instance which satisfies all the invariants and attribute conditions. In the worst case, finding this instance might require trying all possible values for attributes in the transformation model and checking the invariants and attribute conditions for each of them.

On the other hand, there are other UML/OCL verification approaches which are complete, like the theorem prover HOL-OCL [10], but may not terminate so they may require user assistance to complete proofs. As discussed in Section 6.1, designers can select the tool which better fits their needs according to this (and other) trade-offs.

6.3. Validation of Model-to-Model Transformations

Validation tools clarify the question "*is this the right transformation?*" by allowing designers to test if the transformation behaves as expected.

Intuitively, the validation of a transformation consists in exercising the transformation in several scenarios and comparing the result with the expected outcome. Contrary to many verification approaches, validation cannot be fully automated: at some point, the intervention of the designer may be required to select relevant scenarios, to define the expected outcome or to compare between the real and expected results. However tools can provide support to designers during the validation process. In this section, we illustrate how the transformation model can be used to validate a transformation and the degree of tool support that can be achieved.

The most basic level of validation for transformations is the ability to "execute" the transformation in one direction: given a source (target) model provided by the designer, generate the corresponding target (source) model. At this level, we consider that the designer inspects the result himself and determines whether it is correct or not. This execution is not trivial because declarative transformations define *what* is the target model corresponding to a source model, without focusing on *how* it is computed. Some relevant information like the order in which individual transformation rules should be applied is generally omitted.

Thanks to the extracted invariants, it is possible to provide partial support to the execution of transformations without converting them into an imperative form beforehand. However, in order to execute the transformation, all the implicit information has to be completed by an execution engine. This means that the execution engine (i) spends execution time deciding how the computation will be performed and (ii) may follow unsuccessful branches of computation which require backtracks. Therefore, the execution of declarative transformations may be inefficient compared to the execution of imperative transformations.

Anyhow, we can use a UML/OCL consistency checking tool as our execution engine. It can find an instance of the transformation model satisfying the source and target meta-model well-formedness rules, plus the transformation invariants, plus an additional constraint: that the instantiated source model is equal to the one provided by the designer. This way, we obtain a legal instantiation of the transformation model containing the initial source model plus a valid corresponding target model.

The input model can be described as an OCL invariant that restricts the possible set of legal instances to just one, the corresponding to that specific model provided by the designer. For instance, if the designer wants to execute our transformation example using a source model with a single package called "Education" and no classes, our validation process would generate this additional invariant:

```
context Package inv:
Package::allInstances()−>size() = 1 and Class::allInstances()−>isEmpty() and
Package::allInstances()−>exists ( p | p.name = "Education")
```

This invariant is passed to the solver along with the rest of invariants of the transformation model. Note that with this alternative, current tools do not need to be extended to cope with the automatic execution of model transformations. Computing an instance that satisfies both the source model invariant and the transformation invariants will yield the corresponding target model automatically. In a similar way, designers can check which source model/s would generate a specific target model.

A second validation level is the ability to transform partially specified models. For instance, in our running example a designer might want to know whether it is possible to generate a table with three columns without having to fully define an example model, a tedious and time-consuming task [40]. To help in this matter, we can use a similar approach to the one presented so far, but using a weaker invariant to specify the designer-provided input model. In this case, the UML/OCL consistency solver is free to add new elements to the input source model when searching for a legal target model.

In addition, this validation process can be enriched to provide a description of the expected outcome of a transformation using an OCL expression. Like the input model, the outcome can be partially specified, e.g. "the target model should have at least two columns" or "the package should have as many classes as tables appear in the schema". By adding the *negation* of this OCL expression as an invariant to our transformation model, any instance found by the UML/OCL consistency solver becomes a counterexample: a scenario where the outcome of the transformation does *not* match the expected result.

For instance, a typical expected outcome of a transformation is that both source and target models satisfy the meta-model invariants. In order to validate the transformation in the forward direction, we would modify the transformation model by negating the invariants of the target meta-model: if the target meta-model has invariants inv_1, ..., inv_N we would instead impose the invariant $not(inv_1\ and\ \ldots\ and\ inv_N)$. Any instance satisfying the transformation invariants and this new target meta-model invariant would be an example of transformation on an input model producing an incorrect target model. In our running example, this technique would attempt to compute a target model where column names are not unique within the scope of one table (the negation of the OCL constraints in Fig. 2). Using UMLtoCSP we have validated the transformation in this way to produce the counterexample from Fig. 10. Notice that the designer no longer needs to compare the real and expected outcomes, as the tool directly provides a counterexample if it can find one.

Another activity where tools can assist in the validation process is in selecting relevant scenarios. In software systems, non-trivial errors usually appear in corner cases which were not properly considered. In the context of M2M transformations, it makes sense to consider validation scenarios which stress the relationship between the different rules (TGG) or relations (QVT) of a transformation. For example, it might be interesting to consider scenarios where one rule is not enabled for some reason, e.g. a RDBMS model with one schema and no tables (so only `Package-Schema` is enabled) or a UML model without inheritance (so `ChClass-Table` is not enabled). This method can be used as a heuristic to generate potentially "interesting" scenarios which the designer can review to select the significant ones.

Using the transformation invariants and a tool like UMLtoCSP, it is possible to automate the generation of such relevant scenarios. Given a set of N invariants, first we compute instances that satisfy all transformation invariants, but where rule 1 is not enabled. To this end, two additional invariants are added to the transformation model: one in the source model ensuring that the model is not source-enabled and similarly for target-enabledness. The structure of these new invariants is very similar to that of the transformation invariants, but instead of enforcing the mapping whenever the rule is enabled, they require the rule to be disabled:

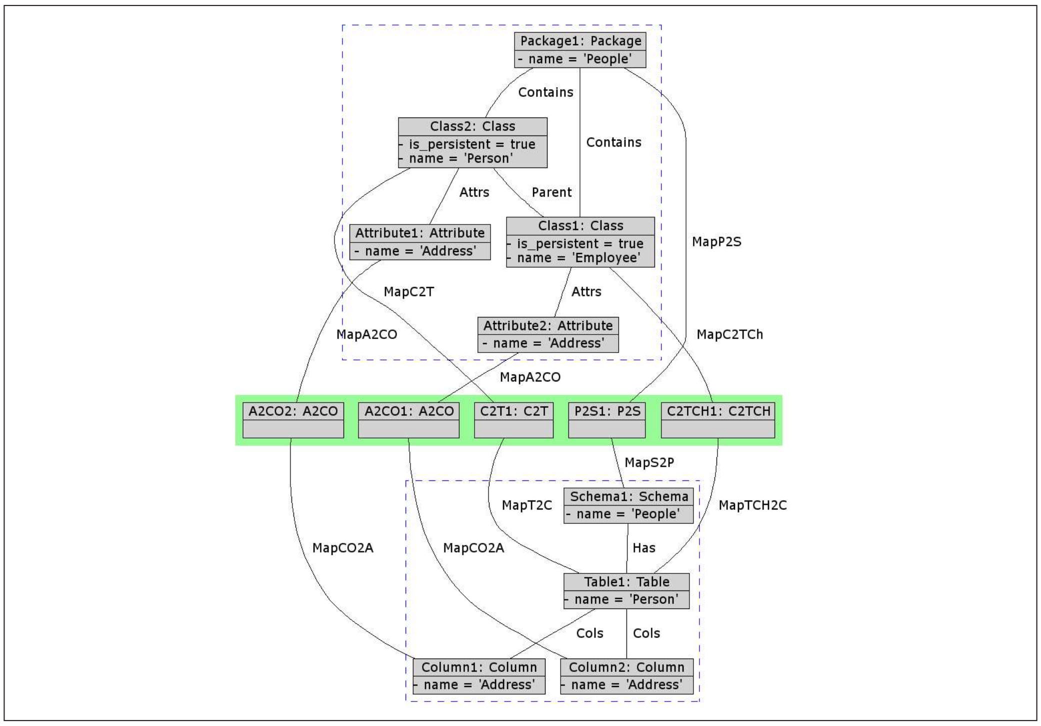


Figure 10: Scenario proving that the TGG transformation is not total, i.e. the source model cannot be transformed into a valid target model (in the target RDBMS model, column names are not unique which violates the meta-model constraints).

Transformation:	$\forall X$: **if** p-enabled(X) **then** $\exists Y$: p-mapping(X, Y) **endif**
New invariant:	$\forall X$: **not** p-enabled(X)

Similarly, we would proceed for the rest of the N rules, one by one. Then, all these scenarios can be presented to the designer, who decides which ones should be validated. As an example, Fig. 11 shows some relevant scenarios generated automatically using UMLtoCSP. Notice how they depict interesting corner cases from the point of view of the validation of the transformation: the empty model (1), an empty package/schema (2), a non-persistent class without attributes (3), a non-persistent class with attributes (4), a hierarchy of non-persistent classes (5) and a package with classes but no inheritance hierarchies (6). Some of them are generated multiple times when we consider different rules, e.g. no rule is source-enabled in the empty model. The same approach can be applied to QVT transformations, focusing on relations instead of rules.

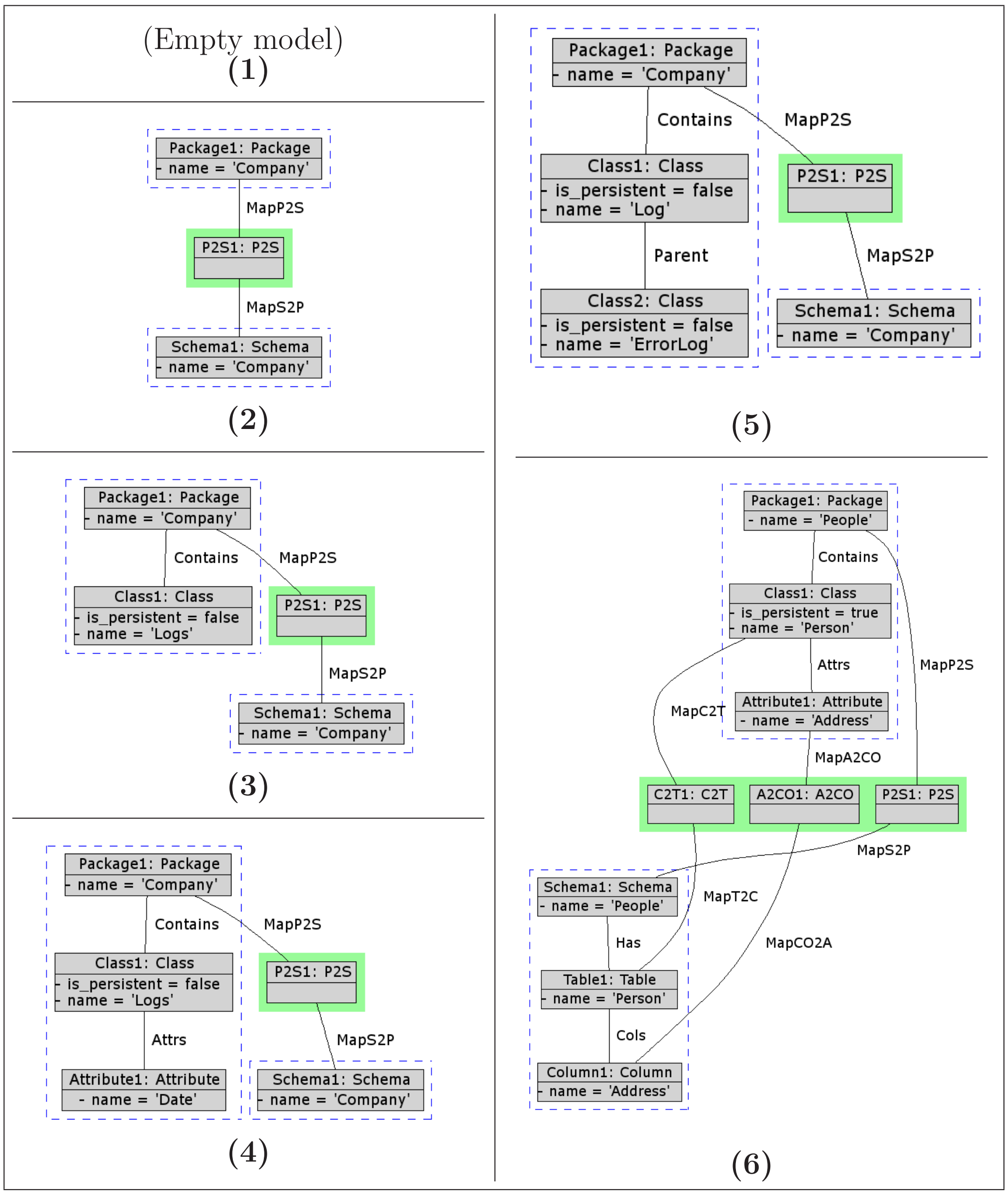


Figure 11: Sample TGG validation scenarios generated automatically, where the following rules are not source-enabled: `Package-Schema` (scenario 1), `Class-Table` (scenarios 1-5), `ChClass-Table` (scenarios 1-6) and `Attribute-Column` (scenarios 1-5).

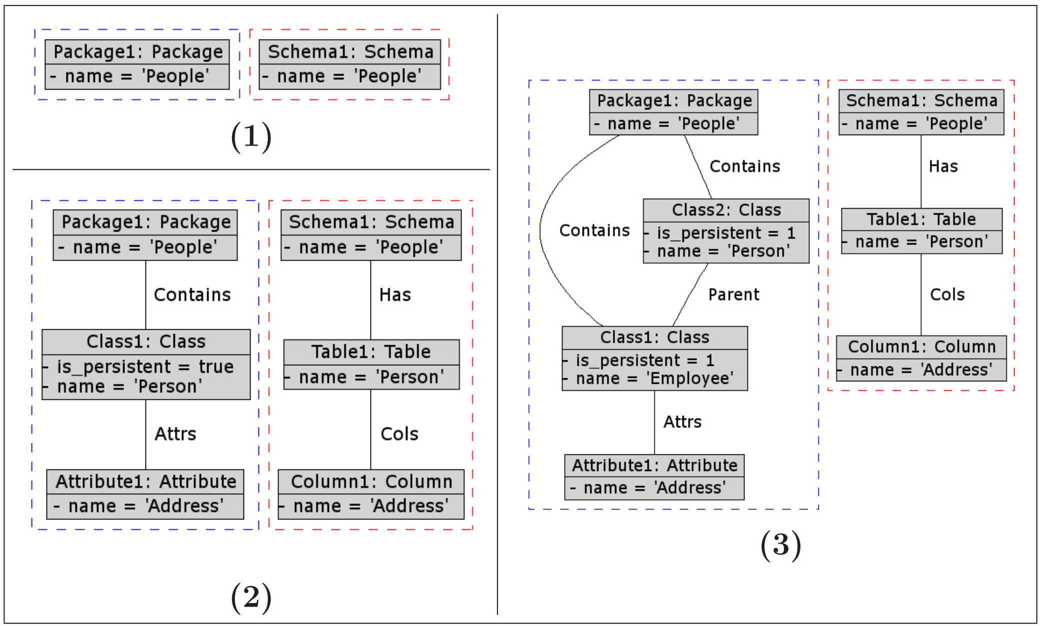


Figure 12: Sample QVT validation scenarios generated automatically, where a rule is applied exactly once: `Package-Schema` (scenarios 1-3), `Class-Table` (scenario 2), `ChClass-Table` (scenario 3), `Attribute-Column` (scenarios 2-3).

This is just one heuristic which can be used to identify interesting corner cases of the transformation, but other heuristics may be used to guide the generation of scenarios. For instance, it is possible to modify the OCL invariants in order to compute instances where a rule can only be applied *once*. The change consists in changing the quantifiers in the invariant, from `forAll` and `exists` to `one`, which enforces that the rule is only applied once. If the previous heuristic was aimed towards the generation of extreme cases of the transformation, this one computes the *base* cases, e.g. models with a single table, with a single package, with a single attribute, with a single inheritance relationship, etc. Fig. 12 depicts some sample scenarios generated with this heuristic in the QVT transformation (again, the same heuristic can be applied to TGGs by considering rules instead of relations). These scenarios highlight, for instance, that a model having tables without columns is not valid according to the meta-model invariants.

Notice that these strategies are just heuristics: there may be other interesting scenarios which are not generated using these heuristics and the tool may generate "redundant" scenarios for a given rule or relation. For instance, once we have validated the transformation of a package with three classes, the tool may suggest the validation of a package with four classes. In this sense, user intervention is still required to select interesting scenarios from the ones generated by a tool.

These automatically generated scenarios can be used in two ways. First, generating the scenarios as instances of the transformation model yields both the source of the target of the transformation. The designer can review the outcome of the transformation manually and validate that it meets his expectations. On the other hand, we can consider only the source (or the target) model of the scenario. The designer can then describe the corresponding target for that source using an invariant and rely on the tool to assess the behavior of the transformation. This second approach can be useful, for instance, if the models are too large to be reviewed by hand or if the transformation is not deterministic and there may be several possible targets for each source.

In short, combining the different techniques from this subsection, we propose the validation flow presented in Fig. 13. First, the tool suggests some validation scenarios to the designer, who selects some of them and either (a) reviews them manually or (b) defines the expected outcome using OCL invariants. Then, the tool attempts to find counterexamples which do not match the designer intent. Even though complete automation cannot be achieved, this validation flow provides tool support through all the validation steps: selection of scenarios, expressing the designer's intentions and identifying scenarios which do not correspond to these goals.

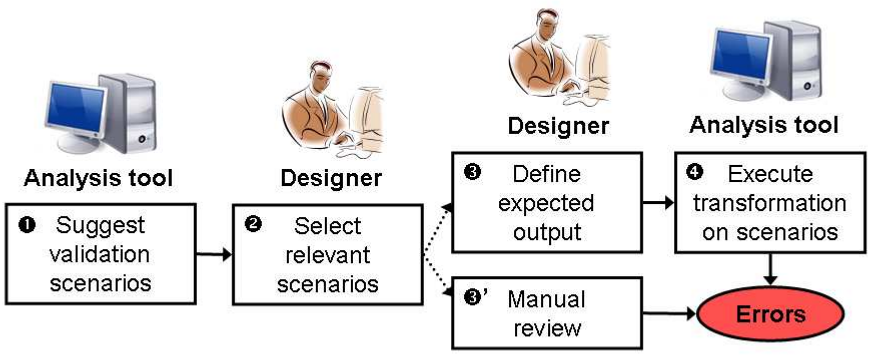

Analysis tool
❶ Suggest validation scenarios
Designer
❷ Select relevant scenarios
Designer
❸ Define expected output
❸' Manual review
Analysis tool
❹ Execute transformation on scenarios
Errors

Figure 13: Validation flow for M2M transformations using any UML/OCL analysis tool.

Any of the tools presented in Section 6.1 can be used to perform activities

1 and 4 in this validation flow. The only requirements for such tools are being able to receive a UML/OCL model as input in some format and to check the consistency of that model. Activity 1 requires checking the consistency of a transformation model where invariants have been modified according to one of the three heuristics presented in this section: (a) failure of a meta-model invariant, (b) a rule is not enabled and (c) a rule is enabled exactly once. Then, activity 4 checks the consistency of the transformation model plus an invariant modelling the source model (defined by the designer in step 2) and an invariant modelling the expected output (the product of step 3). Thus, in activities 1 and 4, the role of the designer is providing the meta-models and invariants in a suitable input format for that tool and using the consistency checking command of the UML/OCL analysis tools.

6.4. Experimental results

In this section, we discuss the experimental results for the examples described in this paper, using the tool UMLtoCSP to perform the analysis of the transformation model.

Three uses of the OCL invariants will be considered: (a) the execution of the transformation source-to-target from a given source model or target-to-source from a given target; (b) the verification of a property defined in Section 6.2; and (c) the validation of the transformation using the heuristics from Section 6.3. Table 1 displays the execution time for each of the examples used in the paper. These results have been measured on an Intel Core 2 Duo 2.53 Ghz with 2 Gb RAM. The examples in Figure 14 have been used to illustrate the execution of source-to-target and target-to-source transformations.

UMLtoCSP is a bounded verification tool and the results have been computed within a bounded search space: each class of the model can be populated with at most two objects, and each attribute has a domain with at most three distinct values. It is important to establish what the search space is because it has a large impact in the verification time: the solver used by UMLtoCSP is backtracking-based, so increasing the search space may cause an exponential blowup in the worst-case execution time. However, a helpful observation that alleviates this problem in bounded verification tools is the "small scope hypothesis" [23], i.e. a large proportion of errors in a system can be identified by considering only instances within a small scope.

In the analysis performed by UMLtoCSP, the critical resource from a complexity point of view is execution time: memory usage is not a concern. This happens because, during the search, the solver only stores in memory

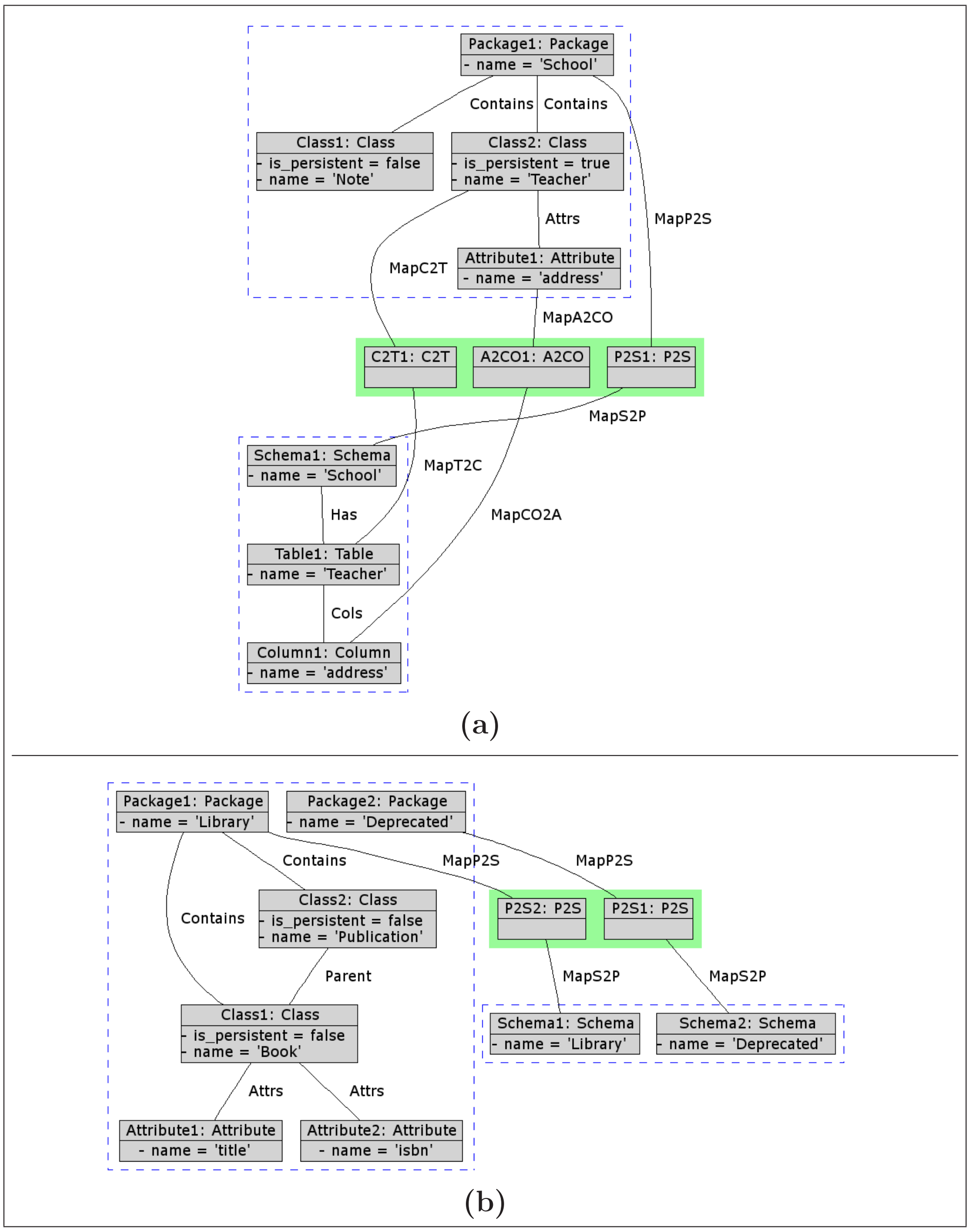


Figure 14: Sample models used to test the execution of the transformation source-to-target and target-to-source.

Execution	**Formalism**	**Example**	**Time**
Source-to-target	TGG	Fig. 14 (a)	1.84s
Target-to-source	TGG	Fig. 14 (a)	0.66s
Source-to-target	TGG	Fig. 14 (b)	0.31s
Target-to-source	TGG	Fig. 14 (b)	0.22s
Source-to-target	QVT	Fig. 14 (a)	0.19s
Target-to-source	QVT	Fig. 14 (a)	0.35s
Source-to-target	QVT	Fig. 14 (b)	0.21s
Target-to-source	QVT	Fig. 14 (b)	0.39s
Verification	**Formalism**	**Example**	**Time**
Satisfiability	TGG	Fig. 9	0.16s
Satisfiability	QVT	Fig. 9	0.02s
Totality	TGG	Fig. 10	2.41s
Validation	**Formalism**	**Example**	**Time**
`Package-Schema` is not applicable	TGG	Fig. 11-1	0.00s
`Class-Table` is not applicable	TGG	Fig. 11-2	0.05s
`Class-Table` is not applicable	TGG	Fig. 11-3	0.00s
`Class-Table` is not applicable	TGG	Fig. 11-4	0.00s
`Class-Table` is not applicable	TGG	Fig. 11-5	0.00s
`ChClass-Table` is not applicable	TGG	Fig. 11-6	0.16s
`Package-Schema` is applicable once	QVT	Fig. 12-1	0.00s
`Class-Table` is applicable once	QVT	Fig. 12-2	0.05s
`ChClass-Table` is applicable once	QVT	Fig. 12-3	0.02s

Table 1: Experimental results for the examples used in this paper.

the system of constraints and one instance of the transformation model (the candidate solution). The number of constraints is linear in terms of the size of the OCL invariants, where by "size" we mean "number of nodes of the abstract syntax tree". Moreover, before the size of the instance can become a problem (i.e. being too large to fit into physical memory), execution time would become a major issue, as the solver would have to check all smaller instances previously.

7. Related Work

The term *transformation model* was coined in [7] where the authors describe its benefits in terms of the uniformity and completeness of the transformation definition, its executability and its direction freeness. The work of

[1] presents a similar approach based on the mathematical concept of relation between source and target models. In both works, transformation models are supposed to be manually specified by the designers. Our work can be seen as a continuation of these approaches, as we derive transformation models automatically from declarative M2M transformations, and show the feasibility of such transformation models and which kinds of analysis can be done, in particular by using a constraint solver.

With respect to the analysis of transformations, our work offers new verification techniques. Current analysis techniques (especially for graph transformations [11, 22, 31, 38]) were developed for standard operational rules aimed to in-place transformations, and not for declarative TGG rules aimed at M2M transformation. Although some of these techniques can be adapted to operational TGG rules, declarative TGG rules cannot be analysed with them. Besides, our method opens the door to the verification of QVT-Relational transformations. Other related approaches, such as [49], characterize properties like stability or preservation of the underlying operational mechanism. In our case, we focus on properties of the declarative specification. In [41] the author states some requirements to determine whether a QVT-Relational transformation is bidirectional (though not necessarily bijective) but does not discuss other properties nor provides a method to automatically check bidirectionality of a given transformation. A similar work for TGG rules is described in [16].

In [35], the authors present a technique for specifying and verifying special kinds of transformations based on OCL and techniques similar to constraint solving. However, this only works for transformations that express refinements of UML class diagrams (like object decomposition or operation refinement), and hence it cannot be used with general M2M transformations.

It is also worth mentioning the approach of [24], which has been implemented in the FORMULA tool. Their approach is based on concepts from algebraic specification and logic programming. Transformations are defined by a source and a target signature, and a set of formulas, expressing the transformation constraints. The FORMULA tool implements a model finding procedure that combines abduction and techniques from SAT Modulo Theories. Hence, they can formalize for example whether a transformation is structure preserving. Except for the OCL support, we believe that such tool could be used to implement some of the analysis techniques we have presented in this paper, especially those for validation.

Our analysis approach, consisting in the translation of the M2M transformation to a formal domain is similar to other approaches: [3, 5] transform the rules into Alloy, [4] translates them into Petri graphs, and [44] into Promela for model-checking. However, again, these approaches are targeted to operational rules and/or present limitations with respect to the expressivity of the meta-models (e.g. specification of well-formedness rules as part of the meta-models definition is not allowed) and the input transformation language. In [37], we presented an approach to model-check in-place rule-based transformation based on their transformation into Maude. Again, note that this approach was for in-place transformations and that we did not take integrity constraints into account.

The emphasis of MDD in model transformation is revealing an urgent need to develop validation and verification techniques especially tailored for M2M transformations. For example, the work in [48] analyses meta-model coverage (i.e. which parts of the source/target model are not transformed) similar to our analysis of total/surjective transformations. In [32] the authors present a method to check whether the semantics of the input model is preserved in the output model of a transformation. That is, they do not try to prove the correctness of the transformation but check if the output and input model are similar in behaviour. In [18] a method is proposed to check semantical equivalence between the initial model and the generated code. In [45], the authors verify transformation correctness with respect to semantic properties by model checking the transition system of the source and target models. Hence, the approach is fully automatic, but the verification has to be done for each input model. Our techniques do not automate behaviour conformance checking, but rely on user intervention to do so.

Even though our work is not especially targeted to testing, it is also worth mentioning that there are some works aimed at developing frameworks for transformation testing [9, 30]. For example, in [9] an algorithm is presented which takes as input the meta-model of the source language and fragments of input models, and it generates appropriate source test models. In addition, our heuristics take into account the invariants extracted from the transformation, which we think can provide more suitable test models for the specific transformation under validation.

Overall, we believe that our work can be regarded as a complementary contribution to the model transformation community effort.

8. Conclusions and Future Work

We have presented a new method for the analysis of declarative M2M transformations based on the automatic extraction of OCL invariants implicit in the transformation definition. These invariants together with the definition of the source and target meta-models comprise a transformation model. Since this transformation model can be regarded as a standard UML/OCL class diagram, it can be processed with all kinds of methods and tools designed for managing class diagrams, spawning from direct application execution, to verification/validation analysis, to metrics measurement and to automatic code generation. The obtained results can then be interpreted in terms of the original transformation specification.

In particular, we have formalized a number of verification properties, both at the rule and transformation levels, and shown how the transformation model can be effectively used for its verification using our UMLtoCSP tool. This approach has the advantage that the M2M specification does not need to be operationalized for its analysis or execution. We have also presented some validation heuristics, proposed a validation process and showed how UMLtoCSP can be used to partially automate this validation process.

In this paper we have focused on TGG and QVT declarative transformations but we believe it is feasible to extend our method to cope with other similar transformation languages as ATL [25] or Tefkat [28]. Note that in this way the extraction of invariants may also serve as a means for the integration of different declarative M2M transformation languages at the lower level.

In our future work, we will evaluate techniques to improve performance in the verification of complex transformations. We are also interested in developing and adapting new techniques for transformation models that help us to perform an incremental execution of the model transformation (the analysis of the invariants helps to detect the rules that need to be reexecuted after changes on the model [14]) and to detect and resolve inconsistencies due to simultaneous changes to both models. Finally, we plan to improve our validation process by extending our list of heuristics for snapshot generation, reusing some ideas from the more mature area of test generation in object-oriented programming [8, 46].

Acknowledgements. We thank the referees for their useful comments, which helped us to improve the paper. This work is sponsored by the Spanish Ministry of Science and Innovation, under projects "MODUWEB" (TIN2006-09678), "METEORIC" (TIN2008-02081) and "Design and construction of a Conceptual Modeling Assistant" (TIN2008-00444/TIN - Grupo Consolidado), and UOC-IN3 research grant.

References

[1] D. H. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Journal on Software and System Modeling*, 2(4):215–239, 2003.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Proc. MoDELS'07*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.

[3] K. Anastasakis, B. Bordbar, and J. M. Kuster. Analysis of model transformations via Alloy. In *ModeVVa'07*, pages 47–56, 2007.

[4] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. CONCUR'01*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.

[5] L. Baresi and P. Spoletini. On the use of Alloy to analyze graph transformation systems. In *Proc. ICGT'06*, volume 4178 of *LNCS*, pages 306–320. Springer, 2006.

[6] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 2005.

[7] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In *Proc. MoDELS'06*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.

[8] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1999.

[9] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE*, pages 85–94. IEEE Computer Society, 2006.

[10] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[11] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Analysing graph transformation rules through OCL. In *Proc. ICMT'08*, volume 5063 of *LNCS*, pages 225–239. Springer, 2008.

[12] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. An invariant-based method for the analysis of declarative model-to-model transformations. In *Proc. MoDELS'08*, volume 5301 of *LNCS*, pages 37–52. Springer, 2008.

[13] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *MoDeVVa 2008. ICST Workshop*, pages 73–80, 2008.

[14] J. Cabot and E. Teniente. Incremental integrity checking of uml/ocl conceptual schemas. *Journal of Systems and Software*, In press, 2009.

[15] J. de Lara and E. Guerra. Pattern-based model-to-model transformation. In *Proc. ICGT'08*, volume 5214 of *LNCS*, pages 426–441. Springer, 2008.

[16] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Proc. FASE'07*, volume 4422 of *LNCS*, pages 72–86. Springer, 2007.

[17] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation.* Springer-Verlag, 2006.

[18] H. Giese, S. Glesner, J. Leitner, W. Schfer, and R. Wagner. Towards verified model transformations. In *ModeVVa'06*, 2006.

[19] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005. .

[20] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In *Proc. MoDELS'07*, volume 4735 of *LNCS*, pages 16–30. Springer, 2007.

[21] E. Guerra and J. de Lara. Event-driven grammars: Relating abstract and concrete levels of visual languages. *Journal on Software and System Modeling, special section on ICGT'04*, pages 317–347, 2007.

[22] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. ICGT'02*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

[23] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[24] E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Softw. Syst. Model.*, In press, 2009.

[25] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA Companion*, pages 719–720. ACM, 2006.

[26] A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148(1):113–150, 2006.

[27] J. M. Küster. Definition and validation of model transformations. *Journal on Software and System Modeling*, 5(3):233–259, 2006.

[28] M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *MTiP'02*, LNCS, pages 139–150. Springer, 2005.

[29] L. Lengyel, T. Levendovszky, and H. Charaf. Constraint validation in model compilers. *Journal of Object Technology*, 5(4):107–127, 2006.

[30] Y. Lin, J. Zhang, and J. Gray. A framework for testing model transformations. In *Model-driven Software Development*, pages 219–236. Springer, 2005.

[31] T. Mens, G. Kniesel, and O. Runge. Transformation dependency analysis - a comparison of two approaches. In *Proc. LMO'06*, pages 167–184, 2006.

[32] A. Narayanan and G. Karsai. Towards verifying model transformations. *ENTCS*, 211:191–200, 2008.

[33] Object Management Group. *UML 2.0 OCL Specification*, 2003.

[34] OMG. MOF 2.0 Query/View/Transformation specification, 2007.

[35] C. Pons and D. García. An OCL-based technique for specifying and verifying refinement-oriented transformations in MDE. In *Proc. MoDELS'06*, volume 4199 of *LNCS*. Springer, 2006.

[36] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In *ER*, volume 4215 of *LNCS*, pages 497–512. Springer, 2006.

[37] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioural semantics of visual modeling languages with maude. In *Proc. SLE'08*, volume 5452 of *LNCS*, pages 54–73. Springer, 2008.

[38] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[39] A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[40] S. Sen, B. Baudry, and D. Precup. Partial model completion in model driven engineering using constraint logic programming. In *Proc. INAP'07*, 2007.

[41] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *MoDELS*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.

[42] M. Stölzel, S. Zschaler, and L. Geiger. Integrating ocl and model transformations in fujaba. *ECEASST*, 5, 2006.

[43] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. UML'03*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[44] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal on Software and System Modeling*, 3(2):85–113, 2004.

[45] D. Varró and A. Pataricza. Automated formal verification of model transformations. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.

[46] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA'04*, pages 97–107. ACM, 2004.

[47] M. Völter and T. Stahl. *Model-driven software development*. Wiley, 2006.

[48] J. Wang, S.-K. Kim, and D. A. Carrington. Verifying metamodel coverage of model transformations. In *Proc. ASWEC'06*, pages 270–282. IEEE Computer Society, 2006.

[49] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proc. ASE'07*, pages 164–173. ACM, 2007.

Appendix

A. TGG

The invariants generated from `Package-Schema` are the following.

```
context Package inv Package-Schema:
  if self.Package-Schema-enabled() then
    Schema :: allInstances()−>exists(s|
      P2S :: allInstances()−>exists(m|
        m.Package-Schema-mapping(self, s))) endif
----------------------------------------------------------------
context Package::Package-Schema-enabled()
 body: true
----------------------------------------------------------------
context P2S inv Package-Schema:
  Package :: allInstances()−>exists(p|
    Schema :: allInstances()−>exists(s|
      self.Package-Schema-mapping(p, s)))
----------------------------------------------------------------
context P2S::Package-Schema-mapping(p:Package, s:Schema)
 body: p.name = s.name and self.package = p and self.schema = s
----------------------------------------------------------------
context Schema inv Package-Schema:
  if self.Package-Schema-enabled() then
    Package :: allInstances()−>exists(p|
      P2S :: allInstances()−>exists(m|
        m.Package-Schema-mapping(p, self))) endif
----------------------------------------------------------------
context Schema::Package-Schema-enabled()
 body: true
```

Note that the `p-enabled` operations for this rule simply return *true* since as long as a package (resp. schema) exists, the rule is enabled. The invariants generated from `ChClass-Table` are the following.

```
context Class inv ChClass-Table:
  Package :: allInstances()->forAll(p|
    Class :: allInstances()->forAll(c|
      CT :: allInstances()->forAll(m1|
        Table :: allInstances()->forAll(t|
          if self.ChClass-Table-enabled(p, c, m1, t) then
            C2TCh :: allInstances()->exists(m2|
              m2.ChClass-Table-mapping(p, c, m1, t, self)) endif))))
```

```
context Class::ChClass-Table-enabled(p:Package, c:Class, m1:CT, t:Table)
 body:  self.parent = c and self.package = p and c.package = p and
        m1.class = c and m1.table = t
```

```
context C2TCh inv ChClass-Table:
  Package :: allInstances()->exists(p|
    Class :: allInstances()->exists(c|
      CT :: allInstances()->exists(m1|
        Table :: allInstances()->exists(t|
          Class :: allInstances()->exists(ch|
            c.package = p and m1.class = c and m1.table = t and
            self.ChClass-Table-mapping(p, c, m1, t, ch)))
```

```
context C2TCh::ChClass-Table-mapping(p:Package, c:Class, m1:CT, t:Table,
ch:Class)
 body:  self.class = ch and self.table = t and ch.parent = c
        and ch.package = p
```

```
context Table inv ChClass-Table:
  Package :: allInstances()->forAll(p|
    Class :: allInstances()->forAll(c|
      CT :: allInstances()->forAll(m1|
        if self.ChClass-Table-enabled(p, c, m1) then
          C2TCh :: allInstances()->exists(m2|
            Class :: allInstances()->exists(ch|
              m2.ChClass-Table-mapping(p, c, m1, self, ch))) endif)))
```

```
context Table::ChClass-Table-enabled(p:Package, c:Class, m1:CT)
 body:  c.package = p and m1.class = c and m1.table = self
```

The invariants generated from `Attribute-Column` are the following.

```
context Attribute inv Attribute-Column:
  Class :: allInstances()->forAll(c|
    CT :: allInstances()->forAll(m1|
      Table :: allInstances()->forAll(t|
        if self.Attribute-Column-enabled(c, m1, t) then
          Column :: allInstances()->exists(co|
            A2Co :: allInstances()->exists(m2|
              m2.Attribute-Column-mapping(c, m1, t, self, co))) endif)))
---------------------------------------------------------------------
context Attribute::Attribute-Column-enabled(c:Class, m1:CT, t:Table)
 body:  self.class = c and m1.class = c and m1.table = t
---------------------------------------------------------------------
context A2Co inv Attribute-Column:
  Class :: allInstances()->exists(c|
    CT :: allInstances()->exists(m1|
      Table :: allInstances()->exists(t|
        Attribute :: allInstances()->exists(a|
          Column :: allInstances()->exists(co|
            m1.class = c and m1.table = t and
            self.Attribute-Column-mapping(c, m1, t, a, co))))))
---------------------------------------------------------------------
context A2Co::Attribute-Column-mapping(c:Class, m1:CT, t:Table,
a:Attribute, co:Column)
 body:  a.name = co.name and a.class = c and co.table = t and
        self.attribute = a and self.column = co
---------------------------------------------------------------------
context Column inv Attribute-Column:
  Class :: allInstances()->forAll(c|
    CT :: allInstances()->forAll(m1|
      Table :: allInstances()->forAll(t|
        if self.Attribute-Column-enabled(c, m1, t) then
          Attribute :: allInstances()->exists(a|
            A2Co :: allInstances()->exists(m2|
              m2.Attribute-Column-mapping(c, m1, t, a, self))) endif)))
---------------------------------------------------------------------
context Column::Attribute-Column-enabled(c:Class, m1:CT, t:Table)
 body:  self.table = t and m1.class = c and m1.table = t
```

B. QVT

The invariants generated from `Package-Schema` are the following.

```
context Package inv Package-Schema:
  if (self.Package-Schema-enabled()) then
    Schema :: allInstances()->exists(s|
      self.Package-Schema-mapping(s)) endif
------------------------------------------------------------
context Package::Package-Schema-enabled()
 body: true
------------------------------------------------------------
context Package::Package-Schema-mapping(s:Schema)
 body: self.name = s.name
------------------------------------------------------------
context Schema inv Package-Schema:
  if (self.Package-Schema-enabled()) then
    Package :: allInstances()->exists(p|
      self.Package-Schema-mapping(p)) endif
------------------------------------------------------------
context Schema::Package-Schema-enabled()
 body: true
------------------------------------------------------------
context Schema::Package-Schema-mapping(p:Package)
 body: self.name = p.name
```

The invariants generated from `ChClass-Table` are the following.

```
context Class::ChClass-Table(t:Table)
 body: Class :: allInstances()->forAll(c1|
         if (self.ChClass-Table-enabled(c1, t)) then
           self.ChClass-Table-mapping(c1, t) endif)
------------------------------------------------------------
context Class::ChClass-Table-enabled(c1:Class, t:Table)
 body: self.child->includes(c1)
------------------------------------------------------------
context Class::ChClass-Table-mapping(c1:Class, t:Table)
 body: c1.Attribute-Column(t) and c1.ChClass-Table(t)
------------------------------------------------------------
context Table::ChClass-Table(c:Class)
 body: if (self.ChClass-Table-enabled(c)) then
         Class :: allInstances()->exists(c1|
           self.ChClass-Table-mapping(c, c1) endif)
------------------------------------------------------------
context Table::ChClass-Table-enabled(c:Class)
 body: true
------------------------------------------------------------
context Table::ChClass-Table-mapping(c:Class, c1:Class)
 body: c.child->includes(c1) and self.Attribute-Column(c1)
       and self.ChClass-Table(c1)
```

The invariants generated from `Attribute-Column` are the following.

```
context Class::Attribute-Column(t:Table)
 body: Attribute :: allInstances()->forAll(a|
         if (self.Attribute-Column-enabled(a, t)) then
           Column :: allInstances()->exists(co|
           self.Attribute-Column-mapping(a, t, co)) endif)

context Class::Attribute-Column-enabled(a:Attribute, t:Table)
 body: self.attribute->includes(a)

context Class::Attribute-Column-mapping(a:Attribute, t:Table, co:Column)
 body: co.name = a.name and t.column->includes(co)

context Table::Attribute-Column(c:Class)
 body: Column :: allInstances()->forAll(co|
         if (self.Attribute-Column-enabled(a, co)) then
           Attribute :: allInstances()->exists(a|
           self.Attribute-Column-mapping(c, a, co)) endif)

context Table::Attribute-Column-enabled(a:Attribute, co:Column)
 body: self.column->includes(co)

context Table::Attribute-Column-mapping(c:Class, a:Attribute, co:Column)
 body: co.name = a.name and c.attribute->includes(a)
```