An Invariant-Based Method for the Analysis of Declarative Model-to-Model Transformations

Jordi Cabot[1], Robert Clarisó[1], Esther Guerra[2], and Juan de Lara[3]

[1] Universitat Oberta de Catalunya (Spain), {jcabot,rclariso}@uoc.edu
[2] Universidad Carlos III de Madrid (Spain), eguerra@inf.uc3m.es
[3] Universidad Autónoma de Madrid (Spain), jdelara@uam.es

Abstract. In this paper we propose a method to derive OCL invariants from declarative specifications of model-to-model transformations. In particular we consider two of the most prominent approaches for specifying such transformations: Triple Graph Grammars and QVT. Once the specification is expressed in the form of invariants, the transformation developer can use such description to verify properties of the original transformation (e.g. whether it defines a total, surjective or injective function), and to validate the transformation by the automatic generation of valid pairs of source and target models.

1 Introduction

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

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 the elements of the target model should be created from elements of the source one. In declarative approaches, some kind of (visual or textual) patterns describing the relations between the source and target models is provided, rather than a program specifying how to create and link their elements. These patterns are complemented with additional information, e.g. to express relations between attributes in source and target elements, as well as to constrain when a certain relation should hold (frequently the OCL standard is used for this purpose [17]). Declarative approaches are higher-level than operational ones and they are inherently bidirectional because they do not specify any causality. Thus, they bring together in a single specification forward (i.e. source-to-target) and backward (i.e. target-to-source) transformations.

Whereas several notations have been proposed for specifying M2M transformations [1, 14, 17, 19], there is a lack of methods for analysing the correctness of declarative M2M specifications in an integral way, taking into account the relations and constraints expressed by the transformation, as well as the meta-models and their well-formedness rules.

In this paper, we propose validation and verification techniques based on the extraction of implicit transformation invariants deduced from the declarative description of the transformations. These invariants state the conditions for a valid mapping between source and target elements and we express them in OCL. We call these invariants, together with the source and target meta-models, a *transformation model* [6]. To show the wide applicability of the technique, we study how to create this transformation model from two common notations for M2M transformations: QVT [17] and Triple Graph Grammars (TGGs) [19].

Once the transformation model is automatically derived we can determine several correctness properties of the original transformation by analyzing the transformation model with any available tool for the verification of static UML/OCL class diagrams (see [2,7,9,21]). In particular, in this work we use our UMLtoCSP tool [9] for the analysis. The tool translates the transformation model into a constraint satisfaction problem that can be processed with constraint solvers to check different aspects of the model. For example, whether it is *satisfiable* (i.e. there is at least one valid pair of related source and target models), *total* (whether all valid source models have a valid related target model) or *deterministic* (whether a source model has just one valid target model).

We also use UMLtoCSP for validation, because the tool is able 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.
Paper Organization. Section 2 introduces TGGs and the method for extracting invariants. Section 3 presents such method for QVT. Section 4 shows the use of the invariants for validation and verification of transformations. Section 5 compares with related work and Section 6 ends with the conclusions.

2 From TGG Rules to OCL Invariants

Triple Graph Grammars (TGGs) [19] were proposed as a means to specify transformations between two languages (i.e. meta-models) in a declarative way. TGGs build on the notion of graph grammar [18]. A graph grammar is made of a set of rules, each having graphs in their left and right hand sides (LHS and RHS), plus an initial graph (i.e. the model to be transformed). The application of a rule to a graph is only possible if an occurrence of the LHS (a *match* morphism) is found in it. Once such occurrence is found, it is replaced by the RHS. 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 match is found for any rule. A M2M transformation by graph transformation has an operational style, as the rules specify how to build the target model assuming the source already exists or vice versa, thus being unidirectional.

TGGs are an attempt to increase the level of abstraction of a M2M specification, being more declarative and bidirectional. TGGs are made of rules working on triples graphs. These consist of two graphs called *source* and *target*, related through a *correspondence* graph. Any graph model can be used for these three graphs, from standard unattributed graphs $(V; E; s, t\colon E \to V)$ to more complex attributed graphs. The nodes in the correspondence graph (the *mappings*) have morphisms[4] to the nodes in the source and target graphs.

Definition 1 (Triple Graph). *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*.*

In the previous definition, V_{G_x} is the set of nodes of graph G_x. Morphisms cs and ct relate two nodes in the source and target graphs, such that $x \in V_{G_s}$ is related to $y \in V_{G_t}$ iff $\exists n \in V_{G_c}$ with $cs(n) = x$ and $ct(n) = y$. We often depict a triple graph by $\langle G_s \overset{cs}{\leftarrow} G_c \overset{ct}{\rightarrow} G_t\rangle$, and use $TrG|_x$ (for $x = \{s, c, t\}$) to refer to the x component of TrG. Next, we define triple graph morphisms as a triple of graph morphisms that preserve the correspondence functions cs and ct.

Definition 2 (Triple Graph Morphism). *A triple graph morphism* $f = (f_s, f_c, f_t) : TrG^1 \to TrG^2$ *is made of three graph morphisms* $f_x\colon TrG^1|_x \to TrG^2|_x$ *(with* $x = \{s, c, t\}$*) such that* $f_s|_V \circ cs^1 = cs^2 \circ f_c|_V$ *and* $f_t|_V \circ ct^1 = ct^2 \circ f_c|_V$*, where* $f_x|_V$ *is morphism* f_x *restricted to nodes.*

We use triple morphisms for three purposes: (i) to define the relation between the LHS and RHS of a TGG rule; (ii) to identify an occurrence (a match) of the LHS in the host graph and (iii) to type a triple graph.

A triple graph is typed by a *meta-model triple* [12] or TGG schema, which contains the source and target meta-models, and declares the types of mappings between the elements of both languages. Fig. 1 shows an example meta-model triple for a simplified translation between class diagrams and relational schemas. The class diagram meta-model includes the derived relation `AllParents` to navigate directly to all ancestors of a given class. The correspondence meta-model declares three classes: `C2T` is used to map classes and tables, whereas `A2Co` and `A2Ch` relate attributes and columns. This last mapping is used to relate an attribute from a parent class to a column of a table associated with a child class. The dotted arrows specify the allowed morphisms

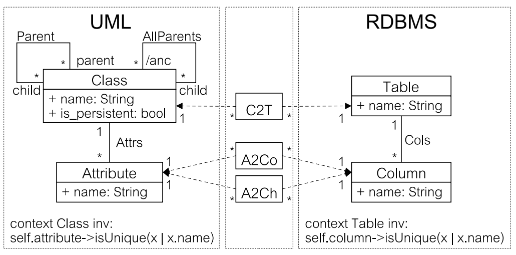

Fig. 1. Example meta-model triple.

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

from the correspondence to the source and target models. These can be treated as normal associations with cardinality 1 on the side of the source/target class and $*$ on the side of the correspondence class. The meta-model includes OCL constraints ensuring uniqueness of attribute names for each class, and similarly for tables.

A typed triple graph is 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 be so called *clan-morphisms* to take inheritance into account [12]. 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 TGG rules that describe the synchronized evolution of two models. Thus, TGG rules have triple graphs in its LHS and RHS, so as to allow manipulating both models synchronously. We use OCL in TGG rules to define attribute conditions.

Definition 3 (Declarative TGG Rule). *A declarative TGG rule* $p = (r\colon \langle L_s \xleftarrow{cs^l} L_c \xrightarrow{ct^l} L_t\rangle \to \langle R_s \xleftarrow{cs^r} R_c \xrightarrow{ct^r} R_t\rangle, ATT_{COND})$ *is made of a triple morphism* r*, and a set* ATT_{COND} *of OCL constraints expressing attribute conditions.*

In the previous definition, the rule's LHS is $\langle L_s \xleftarrow{cs^l} L_c \xrightarrow{ct^l} L_t\rangle$, and the RHS is $\langle R_s \xleftarrow{cs^r} R_c \xrightarrow{ct^r} R_t\rangle$, as r is a triple morphism, the rule is non-deleting. Fig. 2 shows three example TGG declarative rules. They are shown using a compact notation where the LHS and the RHS are presented together, the elements created by the rule (RHS-LHS) are marked as {`new`}, and the elements of the LHS which are preserved are untagged. Rule `Class-Table` declares that every time a persistent class is created, a table with the same name is created simultaneously, and vice versa. `Attribute-Column` specifies that creating an attribute of a class which is already related to a table should create a related column with the same name in that table, and vice versa. Finally, `ParentAttribute-Column` creates a new column in the table associated to a child class for all attributes in each ancestor class. Note that a M2M specification by TGGs is declarative and bidirectional as rules do not specify any direction, but synchronously create and relate source and target elements.

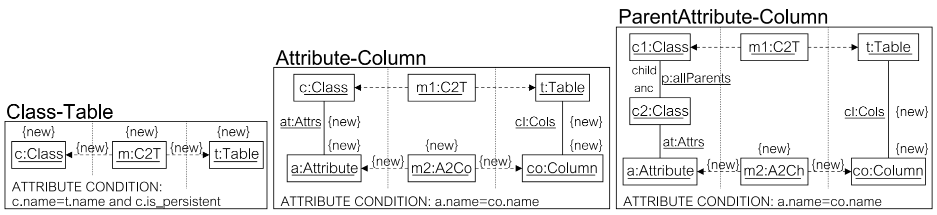

Fig. 2. Some TGG rules for the class-to-relational transformation.

From this declarative specification, an algorithm was proposed in [19] to generate low-level *operational* TGG rules to perform forward and backward transformations, and to relate two existing graphs. The LHS of a forward rule is $\langle R_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} L_t\rangle$ and its RHS is equal to the RHS of the declarative rule. For example, the forward rule for rule `Attribute-Column` has objects `c`, `m1`, `t` and `a` in its LHS, and creates the `co` object related to `t` and mapped to `a` through the new mapping `m2`. Note that a causality has to be assigned to the expressions in the attribute condition section, so that the attribute values for the created objects can be derived from the ones in the LHS. This presents practical problems, which we avoid by compiling the TGG rules into OCL invariants and using a constraint solver, as shown in the next subsection.

2.1 From Declarative TGG Rules to OCL Invariants

The invariant extraction procedure enrichs the transformation model with a set of invariants that capture the semantics embodied by the TGG rules. Intuitively, the invariants must guarantee that the target model is a valid transformation of the source according to the set of rules that can be applied on the source model, and similar for target. A rule is enabled source-to-target in any subgraph of the source model that matches the LHS of the forward operational rule and satisfies the attribute conditions for the source model, and similar for target-to-source. Rules can be fired whenever they are enabled. Thus, for each subgraph where the rule is enabled source-to-target, the invariants should make sure that (a) there is a subgraph of the target model which enables the same rule target-to-source, (b) there are mapping objects connecting both subgraphs as defined in the rule and (c) the union of both subgraphs satisfies any remaining attribute conditions. For each rule p, we call the invariant checking (a) and (b) for the source model $check_s^p$; the one checking (a) and (b) for the target model $check_t^p$; and the one checking (c) for both models $check_c^p$. The invariants capture this semantics using expressions like *"any subgraph matching the LHS of the forward/backwards rule is connected to the mapping objects"* or *"given a mapping object, the source/target models it connects must satisfy the attribute condition"*.

Next definitions 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.

Definition 4 (Invariant Extraction). *Given a declarative TGG rule* $p = (r\colon \langle L_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} L_t\rangle \rightarrow \langle R_s \overset{cs^r}{\leftarrow} R_c \overset{ct^r}{\rightarrow} R_t\rangle, ATT_{COND})$*:*

- $\forall n \in V_{R_x}$*, for* $x = \{s, t\}$*, s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $cs(m) = n$ *or* $ct(m) = n$*, add invariant* $\mathtt{check}_x^p$ *to* $type(n)$*.*
- $\forall m \in V_{R_c} - r(V_{L_c})$*, add invariant* $\mathtt{check}_c^p$ *to* $type(m)$*.*

Invariant $\mathtt{check}_s^p$ checks the existence of triple graph $TrG^s = \langle R_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} L_t\rangle$, together with terms in ATT_{COND} consisting of elements in TrG^s only. Similarly, invariant $\mathtt{check}_t^p$ checks the existence of $TrG^t = \langle L_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} R_t\rangle$, and evaluates the terms in ATT_{COND} that consider elements in TrG^t only. Note the similarity with the pre-conditions for forward/backwards operational TGG rules. In addition, `check`^p_c checks the existence of R and evaluates terms in ATT_{COND} made of elements in $R - r(L)$, not evaluated by previous constraints.

Definition 5 (check^p_s Invariant). *Given TGG rule* $p = (r\colon \langle L_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} L_t\rangle \rightarrow \langle R_s \overset{cs^r}{\leftarrow} R_c \overset{ct^r}{\rightarrow} R_t\rangle, ATT_{COND})$*, then* $\forall n \in V_{R_s} - r(V_{L_s})$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $cs(m) = n$*, the following invariant is generated:*

context *type(n)* — check^p_s invariant
inv : ***if*** $exists^p_n$ ***then*** $self.cor_1{-}{>}size() = 1$ *and ...* $self.cor_m{-}{>}size() = 1$
else $self.cor_1{-}{>}size() = 0$ *and ...* $self.cor_m{-}{>}size() = 0$ ***endif***

context *type(n)::exists^p_n()* — Helper for check^p_s: checks existence of TrG^s
body: $\left.\begin{array}{l} type(n_i) :: allInstances(){-}{>}exists(n_i|... \\ type(n_j) :: allInstances(){-}{>}exists(n_j|... \end{array}\right\} \forall n_k \in V_{TrG^s} - \{n\}$

$\left.n_i.role_j{-}{>}includes(n_j)\ and...\right\} \forall e \in E_{TrG^s}\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$
$...and\ ATT^s_{COND})...)...)...)$

where $cor_j = type(m_j)$ $(j = \{1...m\})$ *with* $m_j \in R_c - r(L_c)$ *and* $cs^r(m_j) = n$*,* $role_j$ *is the role in the meta-model allowing to navigate from* n_i *to* n_j *and* $ATT^s_{COND} \subseteq ATT_{COND}$ *is the biggest subset of constraints involving elements in* TrG^s *only. In* $exists^p_n$*, if some edge has* n *as source or target we use* $self$*.*

If association end $role_j$ has cardinality 1, then we do not use $n_i.role_j{-}{>} includes(n_j)$ but simply $n_i.role_j = n_j$ For all nodes $n \in L_s$ (instead of $R_s - r(L_s)$) that are connected to a newly created correspondence element, a similar invariant is generated with the form **if** $exists^p_n$ **then** $self.cor_1{-}{>}size() >= 1$ and $...self.cor_m{-}{>}size() >= 1$ **else** $self.cor_1{-}{>}size() = 0$ and $...self.cor_m{-}{>} size() = 0$ **endif**. This is necessary, as the node in the source graph already exists, so that it can be added further morphisms from the correspondence node. In the case of the invariant in Definition 5 it can receive just one morphism, as both nodes are created at the same time.

Invariants `check`^p_t are generated in a similar way, but considering nodes $n \in V_{R_t} - r(V_{L_t})$ and then traversing the graph $TrG^t = \langle L_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} R_t\rangle$. For nodes created in the correspondence graph, invariants are generated as follows.

Definition 6 (check^p_c Invariant). *Given TGG rule* $p = (r\colon \langle L_s \overset{cs^l}{\leftarrow} L_c \overset{ct^l}{\rightarrow} L_t\rangle \rightarrow \langle R_s \overset{cs^r}{\leftarrow} R_c \overset{ct^r}{\rightarrow} R_t\rangle, ATT_{COND})$*, then* $\forall n \in R_c - r(L_c)$ *the following invariant is generated:*

context *type(n)* — check^p_c invariant
inv : $\left.\begin{array}{l} type(n_i) :: allInstances(){-}{>}exists(n_i|... \\ type(n_j) :: allInstances(){-}{>}exists(n_j|... \end{array}\right\} \forall n_k \in V_{TrG^c} - \{n\}$

$\left.n_i.role_j{-}{>}includes(n_j)\ and...\right\} \forall e \in E_{TrG^c}\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$
$...and\ ATT^c_{COND})...)...)...)$

with $TrG^c = \langle R_s \overset{cs^r}{\leftarrow} R_c \overset{ct^r}{\rightarrow} R_t\rangle$ *and* $ATT^c_{COND} = ATT_{COND} - (ATT^s_{COND} \cup ATT^t_{COND})$ *the set of attribute conditions not evaluated by previous constraints.*

Lets consider the example rules. As all elements are created in rule `Class-Table`, an invariant is added to each one of the three involved classes. The one in the mapping checks that the source and target elements (a class and a table) it relates exist, and that the names match. The invariant for the class checks that it is persistent and, if (and only if) this is the case, a mapping object is connected to it. Finally, the invariant for tables checks that there is a mapping.

context C2T — $check_c^{Class-Table}$ *invariant*
inv : $Class :: allInstances()\text{−>}exists(c|Table :: allInstances()\text{−>}exists(t|$
$self.class = c \textit{ and } self.table = t \textit{ and } c.name = t.name))$

---

context Class — $check_s^{Class-Table}$ *invariant*
inv : **if** $self.is\_persistent$ **then** $self.c2T\text{−>}size() = 1$
else $self.c2T\text{−>}size() = 0$ **endif**

---

context Table **inv** : $self.c2T\text{−>}size() = 1$ — $check_t^{Class-Table}$ *invariant*

Note that some invariants can be simplified, e.g. the one for `C2T` is equivalent to $self.class.name = self.table.name$. From `Attribute-Column` we generate invariants for the attribute, column and `A2Co` classes, as all are created. The one in the mapping checks that the attribute and the column it relates exist, that they belong to a related class and table, and that their names match. The invariant for the attribute checks that it is mapped to the corresponding mapping element. It does not have to check that the attribute is related to a table, as this is ensured by the association cardinality constraint in the meta-model. Finally, the invariant for the column checks only its relation to a mapping.

context A2Co — $check_c^{Attribute-Column}$ *invariant*
inv : $Attribute :: allInstances()\text{−>}exists(a|Class :: allInstances()\text{−>}exists(c|$
$C2T :: allInstances()\text{−>}exists(m1|Table :: allInstances()\text{−>}exists(t|$
$Column :: allInstances()\text{−>}exists(co|$
$self.attribute = a \textit{ and } a.class = c \textit{ and } c.c2T = m1 \textit{ and } m1.table = t$
$\textit{and } self.column = co \textit{ and } co.table = t \textit{ and } a.name = co.name)))))$

---

context Attribute — $check_s^{Attribute-Column}$ *invariant*
inv : **if** $self.exists_a^{Attribute-Column}$ **then** $self.a2Co\text{−>}size() = 1$
else $self.a2Co\text{−>}size() = 0$ **endif**

---

context Attribute::exists$_a^{Attribute-Column^s}$()
body: $Class :: allInstances()\text{−>}exists(c|C2T :: allInstances()\text{−>}exists(m1|$
$self.class = c \textit{ and } c.c2T = m1))$

---

context Column — $check_t^{Attribute-Column}$ *invariant*
inv : **if** $self.exists_{co}^{Attribute-Column}$ **then** $self.a2Co\text{−>}size() = 1$
else $self.a2Co\text{−>}size() = 0$ **endif**

---

context Column::exists$_{co}^{Attribute-Column}$()
body: $Table :: allInstances()\text{−>}exists(t|C2T :: allInstances()\text{−>}exists(m1|$
$self.table = t \textit{ and } t.c2T = m1))$

3 From QVT-Relations to OCL Invariants

QVT-Relations is a declarative M2M transformation language part of the OMG QVT standard [17]. In this language, a bidirectional tranformation consists of a set of top-level relations between two models. Each relation defines two *domain patterns*, one for each model, and a pair of optional *when* and *where* arbitrary OCL predicates. These optional predicates define the link with other relations in the transformation: the *when* clause indicates the conditions 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 satisfied by all model elements in the relation.

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. However, there are differences in the matching process as patterns may contain *variables*, where some of them may be still *free* while others may be *previously bound* to model elements, e.g. resulting from the evaluation of when clauses or other expressions in the relation, depending on the execution direction of the transformation. Bound variables and constant expressions restrict the possible matches for the pattern. Instead, free variables become bound to the elements matching the pattern. These values may be used afterwards to constrain the value of further pattern expressions.

As an example, consider the rule *Class-Table* expressed as the QVT relation shown in Fig. 3. When executed in the UML→RDBMS direction the relation states that for each persistent class (i.e. each class satisfying the is_persistent condition imposed by the first domain pattern) there must exist a table with a name equal to the value of the *cn* variable. This variable has been initialized with the name of the class matching the first pattern. Additionally, to complete this part of the transformation, we require in the where clause that all attributes of c are mapped to table columns of t as stated in the relation *Attribute-Column* (inside *Attribute-Column*, variables c and t will be bound). When executed in the RDBMS→UML direction the variable *cn* takes the table name as a value and it is used to constrain the possible classes that can be a match for the first pattern. This variable binding process must be simulated by our generated invariants.

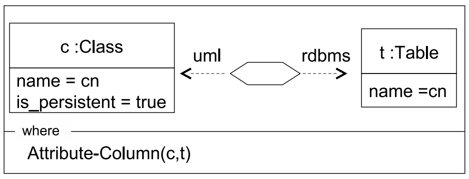

Fig. 3. Example of a QVT relation.

3.1 Extracting the OCL Invariants

Many concepts of QVT-Relations resemble the elements appearing in TGG rules (see [11] for a comparison). Therefore, the procedure for extracting the implicit invariants in a QVT-Relations transformation is similar to that explained in the previous section for TGG rules. For our purposes, the two main differences are that in QVT-Relations related elements in both models are not linked by correspondence nodes and that relationships between different relations can be made explicit by invoking them in the when and where clauses. Due to space limitations, in this section we will focus on explaining these two aspects.

The absence of correspondence nodes forces us to integrate all conditions relating elements of both models into the $\texttt{check}_s^p$ and $\texttt{check}_t^p$ invariants. In TGG rules, these two invariants were mainly used to check that if the model contained a match for the LHS pattern of the operational forward/backwards rule, the required correspondence node existed. But here, the invariants must also take care of checking that when the LHS is a match, the RHS pattern is satisfied as well. To deal with the *when* and *where* clauses of a a relation r, we rewrite r as follows: *(when_clause and* r_{source}*) implies (*r_{target} *and where_clause)*, where r_{source} and r_{target} refer to the source and target patterns in r. Which domain is the source and which the target depends on whether we are generating the $\texttt{check}_s^p$ or the $\texttt{check}_t^p$ invariant. For each different relation r' referenced in any of these clauses, we create an auxiliary OCL query operation $r'(x_1, \ldots, x_n)$ that returns true iff the objects $x_1, \ldots, x_n$ passed as arguments to the operation would satisfy the relation r'.

Following these guidelines, the two invariants generated for the previous Class-Table relation are the following:

context Class — $check_s^{Class-Table}$ *invariant*
inv : self.is_persistent = true implies
    (Table::allInstances()−>exists(t | t.name=self.name and
                                        Attribute-Column(self,t)))

context Table — $check_t^{Table-Class}$ *invariant*
inv : Class::allInstances()−>exists(c | c.name=self.name and
                                        Attribute-Column(c, self))

4 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) Validation of transformations: identifying transformations whose definition does not match the designer intent; and (ii) Verification of correctness properties of transformations: finding defects in the transformation, e.g. whether it is underspecified.

A key notion in our analysis will be the *transformation model*: the union of both the source and target metamodels, together with their integrity constraints (i.e. well-formedness rules), and 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 [2,7,9,21]: given a UML class diagram annotated with OCL constraints, decide whether there exists a legal instance of the model (i.e. an instance that satisfies all graphical and OCL constraints). Several approaches to this problem proceed constructively by automatically computing the legal instance, which is provided to designers as the 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.

4.1 Validation of Model Transformations

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

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. 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 execute the transformation without converting it into an imperative form beforehand. Intuitively, we will use a UML/OCL consistency checking tool to 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 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. For instance, in our example, if the designer wants to execute the transformation using a source model with a single persistent class called "Company" with no attributes, in our validation process we would generate this additional invariant:

```
context Class inv:
Class::allInstances()−>size() = 1 and Attribute::allInstances()−>isEmpty() and
Class::allInstances()−>exists ( cl | cl.name = "Company" and cl.is_persistent )
```

This invariant is passed to the solver along with the rest of the invariants. 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 [20]. 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.

As an example, we have used the tool UMLtoCSP [9] for UML/OCL model consistency checking to validate the TGG and QVT transformations defined as our running example. When validating different scenarios, we have identified two situations where the behavior might not be the one intended by a designer:

1. **Persistent classes without attributes:** These classes are translated into a table with no columns. Probably there should be an invariant stating that persistent classes should have at least one attribute. This scenario was found when using the following invariant as partial description of the source model:

   ```
   context Class inv:
   Class::allInstances()−>exists( x | x.is_persistent and x.attr−>isEmpty() )
   ```

2. **Persistent classes with non-persistent ancestors:** The third rule of the TGG generates a column for each attribute of each ancestor, regardless of whether they are persistent or not. Probably we should restrict columns to the attributes of persistent ancestors only. This scenario was detected using the model partially described by the invariant:

   ```
   context Class inv:
   Class::allInstances()−>exists( x | x.is_persistent and not x.parent.is_persistent
   and not x.parent.attr−>isEmpty() )
   ```

Fig. 4 (2) and (3) show the instantiation which illustrates the problems in both scenarios, as computed automatically by the tool UMLtoCSP [9].

4.2 Verification of Model Transformations

The verification of transformations answers the question "*is the transformation right?*", e.g. are there any defects in the transformation? Up to now, existing methods for the verification of model transformations have focused on lower-level model transformation definitions (see Section 5) and thus, defects are not detected until later stages of the development process.

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). Moreover, it may be desirable to avoid unnecessary invariants in the model. Failing to satisfy these criteria may be a symptom of an incomplete, over-constrained or incorrect model, something which reflects potential defects in the original M2M transformation.

We can study quality notions of transformations at two levels: considering the role of individual rules within the transformation (**R**) or considering the transformation as a whole (**T**). Some properties can be studied at both levels, depending on where we place our focus, the entire model or an individual rule. The complete list of quality properties is the following:

Satisfiable (T/R): There should be at least one source model with a target model satisfying the transformation invariants.

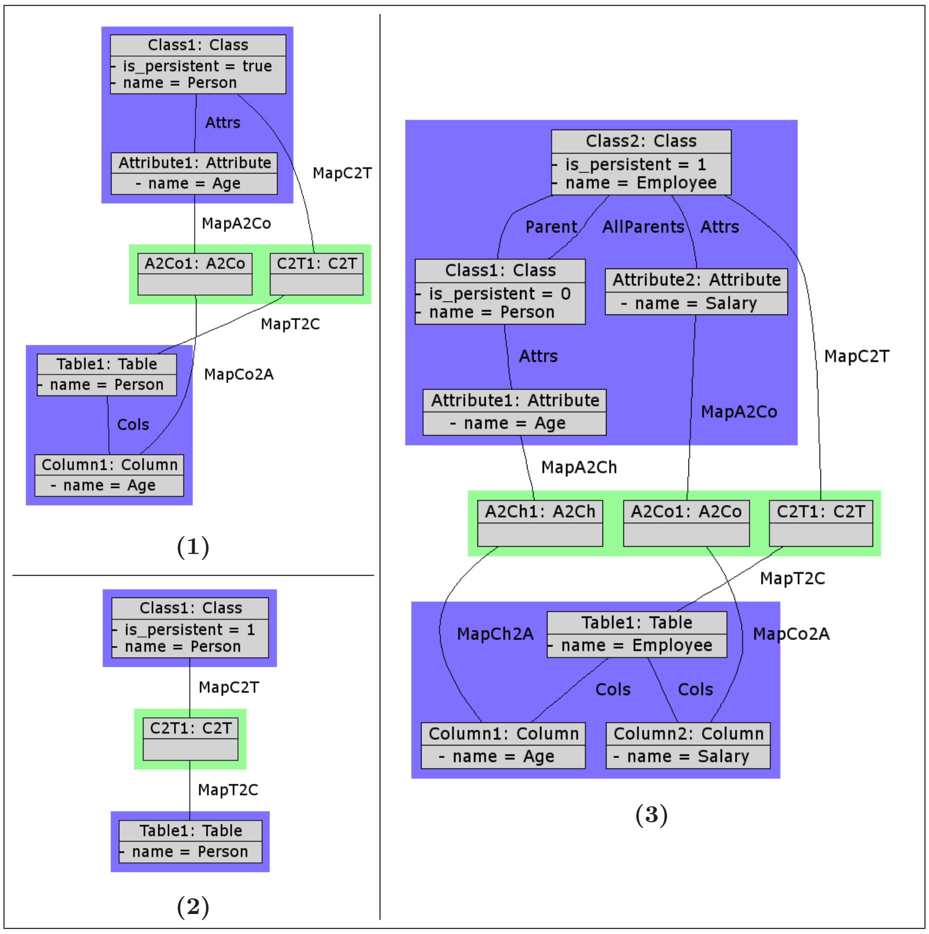

Fig. 4. Verification and validation using UMLtoCSP: (1) verifying satisfiability, (2) validating a scenario where a persistent class has no attributes, (3) validating a scenario where a persistent class has a non-persistent superclass.

Total (T): Each source model has at least one corresponding target model.
Deterministic (T/R): Each source model has at most one corresponding target model.
Exhaustive (T): It is possible to generate each target model from at least one source model.
Injective (T): It is possible to generate each target model from at most one source model.
Non-redundant (R): Given a rule, there is at least one correspondence between source/target models that cannot be fulfilled if the rule is removed.

By rewriting these properties in terms of UML/OCL consistency, we can check them using existing tools. For example, satisfiability of the transformation is directly equivalent to the consistency problem, i.e. a transformation is satisfiable iff its transformation model is satisfiable. As an example, Fig. 4 (1) illustrates the verification of satisfiability on our running example using UMLtoCSP. The tool automatically proves the property by finding a legal pair of source and target models. The tool depicts a legal instance of the metamodel `Class` which is connected through the correspondance nodes to a legal instance of the metamodel `Relational`.

Other quality 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, a legal instance of the source model with no corresponding instance in the target model. To find the counterexample, first we need to 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 all the other quality 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). On the other hand, there are other UML/OCL verification approaches which are complete, like the theorem prover HOL-OCL [7], but may not terminate so they may require user assistance to complete proofs. Designers can select the tool which better fits their needs according to this trade-off.

5 Related Work

The term *transformation model* was coined in [6] where the authors described its benefits. The work of [1] presents a similar approach based on the mathematical concept of relation between the 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 feasability 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 [8, 13, 18]) were developed for standard (i.e. operational) rules and not declarative TGG rules. This implies that we could adapt them to operational TGG rules, but we cannot analyse declarative TGG rules with them. Besides, our method also opens the door to the verification of QVT transformations.

Our analysis approach, consisting in the translation of the M2M transformation to a formal domain is similar to other approaches: [5] and [3] transform the rules into Alloy; [4] translates them into Petri graphs and [22] 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 and the input transformation language.

The emphasis of MDD is revealing an urgent need to develop validation and verification techniques especially targeted for M2M transformations. For example, the work in [23] analyses meta-model coverage (i.e., which parts of the source/target model are not transformed) similar to our analysis of total/surjective transformations. There are also some works to develop frameworks for transformation testing, like [16] or to check the semantically equivalence between the initial model and the generated code [10]. We believe that our work can be regarded as a complementary contribution to this community effort.

6 Conclusions and Future Work

We have presented a new method for the analysis of declarative M2M transformations based on the automatic extraction of 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 used our UMLtoCSP tool, to verify and validate declarative M2M transformations. This approach has the advantage that the M2M specification does not need to be operationalized for its analysis or execution.

Our method focuses on TGG and QVT but we believe it is feasible to extend it to cope with other similar transformation languages. Note that the extraction of invariants may serve as a means for the integration of different declarative M2M transformation languages, like QVT and TGGs.

In our future work, we will evaluate techniques to improve performance in the verification of complex transformations. We also plan to develop and adapt new techniques for transformation models that help us to perform an incremental execution of the model transformation and to detect and resolve inconsistencies due to simultaneous changes to both models.

Acknowledgments: Work supported by the Spanish Ministry of Education and Science, projects MOSAIC (TSI2005-08225-C07-06), MODUWEB (TIN2006-09678) and TIN2005-06053, and an 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 *MoDELS'07*, volume 4735 of *LNCS*, pages 436–450, 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 *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 *ICGT'06*, volume 4178 of *LNCS*, pages 306–320. Springer, 2006.
6. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In *MoDELS'06*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.
7. A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.
8. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Analysing graph transformation rules through OCL. In *Proc. of ICMT 2008*, volume 5063 of *LNCS*, pages 225–239. Springer, 2008.
9. J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *MoDeVVa 2008. ICST Workshop*, 2008.
10. H. Giese, S. Glesner, J. Leitner, W. Schfer, and R. Wagner. Towards verified model transformations. In *ModeVVa'06*, 2006.
11. J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In *MoDELS'07*, volume 4735 of *LNCS*, pages 16–30, 2007.
12. 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.
13. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT'02*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.
14. 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.
15. J. M. Küster. Definition and validation of model transformations. *Journal on Software and Systems Modeling*, 5(3):233–259, 2006.
16. Y. Lin, J. Zhang, and J. Gray. A framework for testing model transformations. In *Model-driven Software Development*, pages 219–236. Springer, 2005.
17. OMG. MOF 2.0 Query/View/Transformation specification, 2007.
18. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.
19. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
20. S. Sen, B. Baudry, and D. Precup. Partial model completion in model driven engineering using constraint logic programming. In *Proc. INAP'07*, 2007.
21. R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *UML'03*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.
22. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal on Software and System Modeling*, 3(2):85–113, 2004.
23. J. Wang, S.-K. Kim, and D. A. Carrington. Verifying metamodel coverage of model transformations. In *ASWEC'06*, pages 270–282. IEEE Computer Society, 2006.