Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions

Esther Guerra[1], Juan de Lara[2], and Fernando Orejas[3]

[1] Universidad Carlos III de Madrid (Spain), eguerra@inf.uc3m.es
[2] Universidad Autónoma de Madrid (Spain), jdelara@uam.es
[3] Universitat Politècnica de Catalunya (Spain), orejas@lsi.upc.edu

Abstract. Pattern-based model-to-model transformation is a new approach for specifying transformations in a declarative, relational and formal style. The language relies on patterns describing allowed or forbidden relations between two models, which are compiled into operational mechanisms to perform forward and backward transformations.
In this paper, we extend the approach for handling attribute conditions expressed in some suitable logic, adapt the operational mechanisms based on graph transformation to relax attribute handling by constraint solving, and discuss heuristics for the compilation of patterns into rules.

1 Introduction

Model-to-Model (M2M) transformations are widely used in Model-Driven Engineering, e.g. to migrate between language versions, to transform into a varification domain, or to refine a model. There are two main approaches to M2M transformation: *operational* and *declarative*. The former is based on operations that explicitly state how and when creating target elements from source ones. Instead, declarative approaches describe mappings between source and target models in a direction-neutral way. Then, operational mechanisms are generated for different scenarios, e.g. to transform a source model into a target one or vice versa (forward and backward transformations), to synchronize two models, or to signal inconsistencies between them [8].

In previous work [3] we proposed a declarative, relational and formal approach to M2M transformation based on *triple patterns* to express the relations between two models. Our patterns are similar to graph constraints [6] but for triple graphs made of two graphs plus their traceability relations. Patterns can specify positive information (the relation they declare must hold) or negative one (the relation must not hold). A pattern specification is compiled into operational mechanisms, implemented with Triple Graph Grammar (TGG) operational rules [5, 8, 14], to perform forward and backward transformations.

In this paper we extend our framework with attributes. Traditionally, attribute handling has been one of the main difficulties of declarative bidirectional languages. For example, attribute computations must be specified in a non-causal way, and therefore generating operational mechanisms involves their algebraic

manipulation for the synthesis of attribute pre-conditions and computations, which may be difficult to automate. We tackle these issues by the uniform integration of attribute computations and conditions in patterns, and by considering the manipulated models also as constraints, hence avoiding algebraic manipulation. Thus, during the transformation, attributes in models are specified by variables and formulae constraining them. When the transformation finishes, one can resort to an equation solver to obtain concrete attribute values.

The advantages of our proposal are the following. First, its relational style contrasts with declarative approaches such as TGGs, where a causality between the existing elements in the models and the ones to be created has to be given. Second, the order of pattern enforcement is deduced, contrary to approaches such as QVT, where it must be explicitly specified. Third, its formal foundation allows studying the specification in both declarative (patterns) and operational (derived rules) formats. Fourth, our patterns have a compositional style, i.e. a triple graph satisfies two patterns in conjunction if it satisfies the two patterns separately. This makes pattern-based specifications extensible. Finally, the separation of the operational mechanism from the declarative specification allows generating operational mechanisms for different purposes, as well as using different operational languages (e.g. graph grammar rules, a constraint solver, or QVT core [13]).

Paper Organization. Section 2 introduces triple graphs, our new concept of constraints, and the algebraic approach to M2M transformation. Sections 3 and 4 present our pattern-based notation and the generation of operational TGG rules, sketching some heuristics to improve their efficacy. Section 5 presents a case study. Section 6 compares with related work, and Section 7 ends with the conclusions.

2 Algebraic Approach to Model-to-Model Transformation

This section introduces triple graphs, constraint triple graphs, and triple graph transformation. Triple graphs are based on labelled graphs (called E-graphs in [6]), which are graphs allowing data in nodes and edges. Formally, an E-graph G is defined as a special kind of graph that includes an additional set of nodes D^G with the values stored in the graph, and two additional kinds of edges that are used for attribution of nodes and edges. Mappings between E-graphs (morphisms) are tuples of set morphisms – one for each set in the E-graph – such that the structure of the E-graph is preserved (for details see [6]). For the typing we use a type graph [6], similar to a meta-model, but for simplicity we omit further discussion on types.

Triple graphs are made of three graphs: source (S), target (T) and correspondence (C). Nodes in the correspondence graph relate nodes in the source and target graphs by means of two graph morphisms [5], and for technical reasons we restrict them to be unattributed (i.e. $D^C = \emptyset$). We use triple graphs to store the source and target models of a M2M transformation, as well as the transformation traces.

Definition 1 (Triple Graph and Morphism) *A triple graph* $TrG = (S \xleftarrow{c_S} C \xrightarrow{c_T} T)$ *is made of three E-graphs* S, C *and* T *s.t.* $D^C = \emptyset$, *and two graph morphisms* c_S *and* c_T *(the source and target correspondence functions).*

A triple morphism $m = (m_S, m_C, m_T) \colon TrG^1 \to TrG^2$ *is made of three E-morphisms* m_X *for* $X = \{S, C, T\}$, *s.t.* $m_S \circ c_S^1 = c_S^2 \circ m_C$ *and* $m_T \circ c_T^1 = c_T^2 \circ m_C$, *where* c_S^x *and* c_T^x *are the correspondence functions of* TrG^x *(for x={1, 2}).*

We use the notation $\langle S, C, T \rangle$ for a triple graph made of graphs S, C and T. Given $TrG = \langle S, C, T \rangle$, we write $TrG|_X$ for $X \in \{S, C, T\}$ to refer to a triple graph where only the X graph is present, e.g. $TrG|_S = \langle S, \emptyset, \emptyset \rangle$. Triple graphs and morphisms form the category **TrG**.

Example. Fig. 1 shows a triple graph relating a class diagram and a relational schema. The graph nodes are depicted as rectangles, and the data nodes in D^S and D^T as circles. We only draw the used data nodes, as they may be infinite. Graph G in Fig. 5 shows the same triple graph using the UML notation, as well as types.

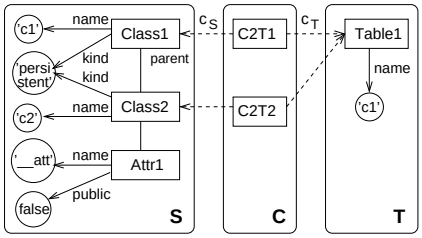

Fig. 1. Triple graph example.

In order to describe the manipulation of triple graphs by means of graph transformation rules, these rules may need to include graphs storing variables that will typically be instantiated when applying the rule. Moreover, we may need to express some properties about these variables. We have formalized this kind of graph using the new notion of *constraint triple graphs*. These are triple graphs attributed over a finite set of variables, and equipped with a formula on this set to constrain the possible attribute values of source and target elements.

Definition 2 (Constraint Triple Graph) *Given an algebra* $\mathcal{A}$ *over signature* $\Sigma = (S, OP)$, *a* constraint triple graph $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$ *consists of a triple graph* $TrG = \langle S, C, T \rangle$, *a finite set of S-sorted variables* $\nu = D^S \uplus D^T$ *(with* $\uplus$ *denoting disjoint union) and a* $\Sigma(\nu)-$*formula* α *in conjunctive or clausal form.*

Example. Fig. 2 shows a constraint triple graph. We take the convention of placing in the left compartment the terms of the formula concerning only source graph attributes; in the right compartment the terms constraining only attributes in the target; and the terms constraining both in the middle. In all cases we omit the conjunctions. Note that "=" denotes equality, not assignment. Hence, in our approach there is no attribute computation, but only attribute conditions. Finally, unused attributes are omitted in the figures, and the formula of the empty constraint is equal to **true**.

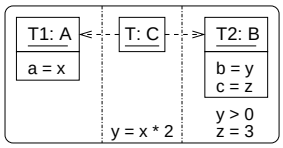

Fig. 2. Constraint.

Notice that constraint triple graphs do not store data explicitly in the graphs: the data nodes D^S and D^T are variables. Thus, if for instance we would like to store a value V on an attribute node, it is enough to label that node with some fresh variable X and include the equality $X = V$ in the associated formula.

Before defining morphisms between constraints, we need an auxiliary operation for restricting $\Sigma(\nu)-$formulae to a smaller set of variables $\nu' \subseteq \nu$. This will be useful for example when restricting a constraint triple graph to the source or target graph only. Thus, given a $\Sigma(\nu)$-formula α, its restriction to $\nu' \subseteq \nu$ is given by $\alpha|_{\nu'} = \alpha'$, where α' is like α, but with all clauses with variables in $\nu - \nu'$ replaced by **true**. Thus, for example $(x = 3) \wedge (y = 7)|_{\{x\}} = (x = 3)$.

Given a constraint $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$, we write α^S for the restriction to the source variables $\alpha|_{D^S}$, and α^T for the restriction to the target variables $\alpha|_{D^T}$. Given a variable assignment $\mathfrak{f}\colon \nu \to \mathcal{A}$, we write $\mathcal{A} \models_{\mathfrak{f}} \alpha$ to denote that the algebra $\mathcal{A}$ satisfies the formula α with the value assignment induced by $\mathfrak{f}$.

Next, we define morphisms between constraint triple graphs. These are made of a triple graph morphism and a mapping of variables (i.e. a set morphism). In addition we require an implication from the formula of the constraint in the codomain to the one in the domain, and also implications from the source and target restrictions of the formula in the codomain to the restrictions of the formula in the domain. This means that the formula in the domain constraint should be weaker or equivalent to the target (intuitively, the codomain should contain "more information").

Definition 3 (Constraint Triple Graph Morphism) *A constraint triple graph morphism $m = (m^{TrG}, m^{\nu})\colon CTrG_1^{\mathcal{A}} \to CTrG_2^{\mathcal{A}}$ is made of a triple morphism $m^{TrG}\colon TrG_1 \to TrG_2$ and a mapping $m^{\nu}\colon \nu_1 \to \nu_2$ s.t. the diagram to the left of Fig. 3 commutes, and $\forall \mathfrak{f}\colon \nu_2 \to \mathcal{A}$ s.t. $\mathcal{A} \models_{\mathfrak{f}} \alpha_2$, then $\mathcal{A} \models_{\mathfrak{f}} (\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)) \wedge (\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T)) \wedge (\alpha_2 \Rightarrow m^{\nu}(\alpha_1))$, where $m^{\nu}(\alpha)$ denotes the formula obtained by replacing every variable X in α by the variable $m^{\nu}(X)$.*

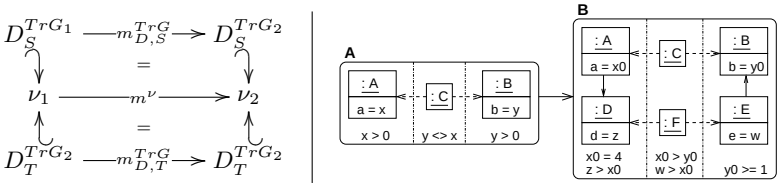

Fig. 3. Condition for **CTrG**-morphisms (left). Example (right).

Remark. Note that $\alpha_2 \Rightarrow m^{\nu}(\alpha_1)$ does not imply $\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)$ or $\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T)$. For technical reasons we require $(\alpha_2^S \Rightarrow m^{\nu}(\alpha_1^S)) \wedge (\alpha_2^T \Rightarrow m^{\nu}(\alpha_1^T))$ as will be evident in Definition 4 and its associated remark.

Example. The right of Fig. 3 shows a constraint triple graph morphism. Concerning the formula, assume some variable assignment $\mathfrak{f}\colon \nu_B \to \mathcal{A}$ satisfying α_B (i.e., s.t. $\mathcal{A} \models_{\mathfrak{f}} \alpha_B$), then such $\mathfrak{f}$ makes $\mathcal{A} \models_{\mathfrak{f}} [(x_0 = 4 \wedge z > x_0) \Rightarrow (x_0 > 0)] \wedge [(y_0 >= 1) \Rightarrow (y_0 > 0)] \wedge [(x_0 = 4 \wedge z > x_0 \wedge x_0 > y_0 \wedge w > x_0 \wedge y_0 >= 1) \Rightarrow (x_0 > 0 \wedge y_0 <> x_0 \wedge y_0 > 0)]$. That is, the formula in the morphism domain should be weaker or equivalent to the formula in its codomain.

From now on, we restrict to injective morphisms (for simplicity, and because or patterns are made of injective morphisms). Given Σ and $\mathcal{A}$, constraint triple

graphs and morphisms form the category $\mathbf{CTrG}_{\mathcal{A}}$. As we will show later, we need to manipulate objects in this category through pushouts and restrictions. A pushout is the result from gluing two objects B and C along a common subobject A, written $B +_A C$. Pushouts in $\mathbf{CTrG}_{\mathcal{A}}$ are built by making the pushout of the triple graphs, and taking the conjunction of their formulae.

Proposition 1 (Pushout in $\mathbf{CTrG}_{\mathcal{A}}$) *Given the span of $\mathbf{CTrG}_{\mathcal{A}}$-morphisms $B^{\mathcal{A}} \xleftarrow{b} A^{\mathcal{A}} \xrightarrow{c} C^{\mathcal{A}}$, its pushout is given by $D^{\mathcal{A}} = (B +_A C, \nu_B +_{\nu_A} \nu_C, c'(\alpha_B) \wedge b'(\alpha_C))$, and morphisms $c' \colon B^{\mathcal{A}} \to D^{\mathcal{A}}$ and $b' \colon C^{\mathcal{A}} \to D^{\mathcal{A}}$ induced by the pushouts in triple graphs $(B +_A C)$ and sets $(\nu_B +_{\nu_A} \nu_C)$.*

Example. Fig. 4 shows a pushout, where $\alpha_D \Rightarrow c'(b(\alpha_A)) \equiv b'(c(\alpha_A))$.

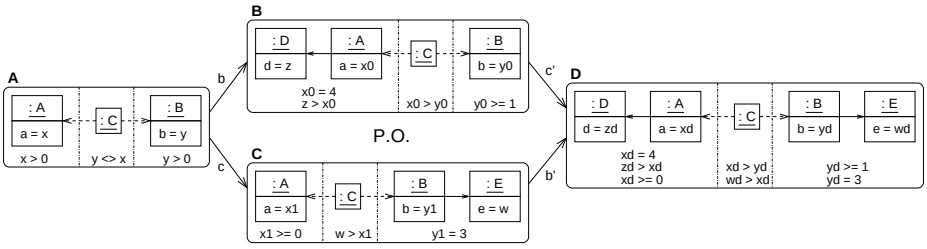

Fig. 4. Pushout example.

The source restriction of a constraint triple graph is made of the source graph and the source formula, and similarly for target. This will be used later to keep the source or target models in a constraint, when such constraint is evaluated source-to-target or target-to-source.

Definition 4 (Constraint Restriction) *Given $CTrG^{\mathcal{A}} = (TrG, \nu, \alpha)$, its source restriction is given by $CTrG^{\mathcal{A}}|_S = (TrG|_S = \langle S, \emptyset, \emptyset \rangle, D^S, \alpha|_{D^S} = \alpha^S)$. The target restriction $CTrG^{\mathcal{A}}|_T$ is calculated in an analogous way.*

Remark. The source restriction $CTrG^{\mathcal{A}}|_S$ of a constraint induces a morphism $CTrG^{\mathcal{A}}|_S \hookrightarrow CTrG^{\mathcal{A}}$. Also, given a morphism $q \colon C^{\mathcal{A}} \to Q^{\mathcal{A}}$, we can construct morphisms $q_S \colon C^{\mathcal{A}}|_S \to Q^{\mathcal{A}}|_S$ and $q_T \colon C^{\mathcal{A}}|_T \to Q^{\mathcal{A}}|_T$.

An attributed triple graph can be seen as a constraint triple graph whose formula is satisfied by a unique variable assignment, i.e. $\exists_1 \mathfrak{f} : \nu \to \mathcal{A}$ with $\mathcal{A} \models_{\mathfrak{f}} \alpha$. We call such constraints *ground*, and they form the $\mathbf{GroundCTrG}_{\mathcal{A}}$ full subcategory of $\mathbf{CTrG}_{\mathcal{A}}$. We usually depict ground constraints with the attribute values induced by the formula in the attribute compartments and omit the formula (e.g. see constraint *CTrG* to the right of Fig. 7). The equivalence between ground constraints and triple graphs is useful as, from now on, we just need to work with constraint triple graphs. In particular, triple graphs are manipulated with TGG operational rules, but seeing them as ground constraint graphs, which

offers several benefits, as we will see. The rules that we consider in this paper are non-deleting and consist of left and right hand sides (LHS and RHS) made of a constraint triple graph each, plus sets of negative pre- and post-conditions. A rule can be applied to a *host* triple graph if a constraint morphism exists from the LHS to the graph and no negative pre-condition (also called NAC) is found. Then, the rule is applied by making a pushout of the RHS and the host graph through their intersection LHS, which adds the elements created by the rule to the host graph. This step is called direct derivation. Negative post-conditions are checked after rule application, and such application is undone if they are found.

The most usual way [6, 14] of dealing explicitly with triple graphs instead of with ground constraint graphs poses some difficulties, most notably concerning attribute handling. For instance, Fig. 5 shows an example where a TGG operational rule is applied to a triple graph G. The rule creates a column for each private attribute starting by '__'. Function `LTRIM(p1,p2)` returns `p2` after removing `p1` from its beginning.

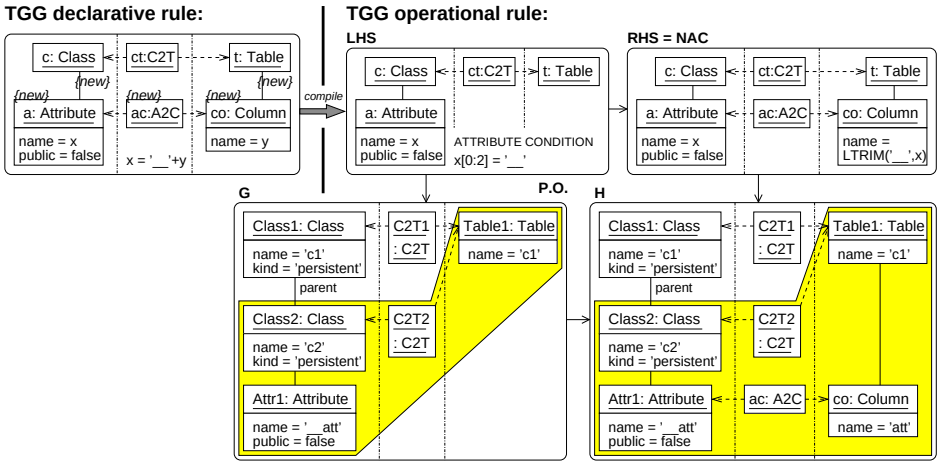

Fig. 5. Direct derivation by a non-deleting TGG operational rule.

In practice, the TGG operational rules are not specified by hand, but derived from declarative rules modelling the synchronized evolution of two models [14], as depicted in the upper part of Fig. 5. The declarative rule is shown with its LHS and RHS together, and *new* tags indicating the synchronously created elements. Of course, in declarative rules, attribute computations must be expressed in a declarative style. However, their compilation into operational rules has to assign a causality to attribute computations, which involves algebraic manipulation of formulae. Moreover, appropriate attribute conditions must be synthesized too. In the example, the condition x='__'+y has to be transformed into a computation *LTRIM('__',x)* for the created column name, and into the condition *x[0:2]*='__' as the attribute name should start by '__'. Please note that this kind of manip-

ulation is difficult to automate, since it involves the synthesis of operations and conditions. Our approach proposes a more straightforward solution. Fig. 8 shows the same example when dealing with triple graphs as ground constraints, where there is no need to synthesize attribute computations. The result of a transformation is a pair of models where their attributes are variables with values given by formulae. If needed, a constraint solver can compute concrete values.

3 Pattern-Based Model-to-Model Transformation

Triple Patterns are similar to graph constraints [6], but made of constraint triple graphs instead of graphs. We use them to describe the allowed and forbidden relations between source and target models in a M2M transformation.

Definition 5 (Triple Pattern) *Given the injective* $\mathbf{CTrG}_{\mathcal{A}}$*-morphism* $C \xrightarrow{q} Q$ *and the sets of injective* $\mathbf{CTrG}_{\mathcal{A}}$*-morphisms* $N_{Pre} = \{Q \xrightarrow{c_i} C_i\}_{i \in Pre}$, $N_{Post} = \{Q \xrightarrow{c_j} C_j\}_{j \in Post}$ *of negative pre- and post-conditions:*

- $\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j)$ *is a positive pattern (P-pattern).*
- $\overrightarrow{N}(C_j)$ *is a negative pattern (N-pattern).*

Remark. The notation $\overleftarrow{P}(\cdot)$, $\overleftarrow{N}(\cdot)$, $\overrightarrow{N}(\cdot)$ and $P(\cdot)$ is just syntactic sugar to indicate a positive pre-condition (that we call parameter), a negative pre-condition, a negative post-condition and the main constraint respectively.

The simplest P-pattern is made of a main constraint Q restricted by negative pre- and post-conditions (Pre and $Post$ sets). In this case, Q has to be present in a triple graph (i.e. in a ground constraint) whenever no negative pre-condition C_i is found; and if Q is present, no negative post-condition C_j can be found. While pre-conditions express restrictions for the constraint Q to occur, post-conditions describe forbidden graphs. If a negative pre-condition is found, it is not mandatory to find Q, but still possible. P-patterns can also have parameters, specified with a non-empty C. In such case, Q has to be found only if C is also found. Finally, an N-pattern is made of one negative post-condition, forbidden to occur (and hence C and Q are empty).

Example. The left of Fig. 6 shows a P-pattern, taken from the class to relational transformation [13]. It is made of a main constraint `C-T` with a negative pre-condition `Parent`. It maps persistent classes without parents to tables with the same name. The negative pre-condition shows only the elements that do not belong to the main constraint, and those connected to them.

The right of Fig. 6 shows a P-pattern with its parameters indicated with $\langle\langle$`param`$\rangle\rangle$. We present C and Q together, as usually the formula in C is the same as the one in Q. The pattern maps the attributes of a class with the columns of the table related to the class. As Section 4.1 will show, it is not even necessary to specify the parameters, as our heuristics are able to suggest them. In fact, a M2M specification is usually made of N- and P-patterns without parameters. For technical reasons, we assume that no P-pattern has negative post-conditions.

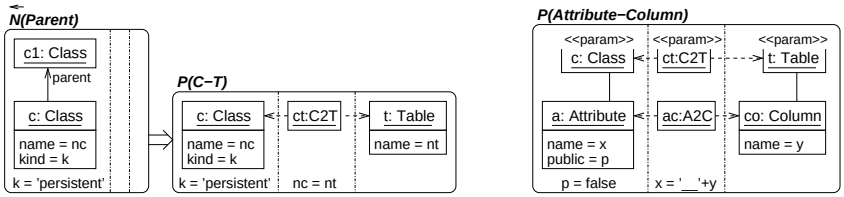

Fig. 6. P-pattern examples.

Definition 6 (M2M Specification) *A M2M specification* $SP = \bigwedge_{i \in I} P_i$ *is a conjunction of patterns, where each* P_i *can be positive or negative.*

Next we define pattern satisfaction. A unique definition is enough as N-patterns are a special case of P-patterns. We check satisfiability of patterns on constraint triple graphs, not necessarily ground. This is so because, during a transformation, the source and target models do not need to be ground. When the transformation finishes we can use a solver in order to find an attribute assignment satisfying the formulae.

We define forward and backward satisfaction. In the former we check that the main constraint of the pattern is found in all places where the pattern is source-enabled (roughly, in all places where the pre-conditions for enforcing the pattern in a forward transformation hold). The separation between forward and backward satisfaction is useful because, e.g. if we transform forwards (assuming an initial empty target) we just need to check forward satisfaction. Full satisfaction implies both forward and backward satisfaction and is useful to check if two graphs are actually synchronized.

Definition 7 (Satisfaction) *A constraint triple graph* $CTrG$ *satisfies* $CP = [\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j)]$*, written* $CTrG \models CP$*, iff:*

- CP *is* forward satisfiable*,* $CTrG \models_F CP$*:* $[\forall m^S \colon P_S \to CTrG$ *s.t.* $(\forall i \in Pre$ *s.t.* $N_i^S \ncong P_S, \nexists n_i^S \colon N_i^S \to CTrG$ *with* $m^S = n_i^S \circ a_i^S)$*,* $\exists m \colon Q \to CTrG$ *with* $m \circ q^S = m^S$*, s.t.* $\forall j \in Post \nexists n_j \colon C_j \to CTrG$ *with* $m = n_j \circ c_j]$*, and*
- CP *is* backward satisfiable*,* $CTrG \models_B CP$*:* $[\forall m^T \colon P_T \to CTrG$ *s.t.* $(\forall i \in Pre$ *s.t.* $N_i^T \ncong P_T, \nexists n_i^T \colon N_i^T \to CTrG$ *with* $m^T = n_i^T \circ a_i^T)$*,* $\exists m \colon Q \to CTrG$ *with* $m \circ q^T = m^T$*, s.t.* $\forall j \in Post \nexists n_j \colon C_j \to CTrG$ *with* $m = n_j \circ c_j]$*,*

with $P_x = C +_{C|_x} Q|_x$*,* $N_i^x = C +_{C|_x} C_i|_x$ *and* $N_i^x \xleftarrow{a_i^x} P_x \xrightarrow{q^x} Q$ $(x \in \{S, T\})$*, see left of Fig. 7.* $C +_{C|_x} Q|_x$ *is the pushout object of* C *and* $Q|_x$ *through* $C|_x$*.*

In forward satisfaction, for each occurrence of $P_S = C +_{C|_S} Q|_S$ satisfying the negative pre-conditions, an occurrence of Q must be found satisfying the negative post-conditions. A pattern is satisfied either because no m^S exists (*trivial satisfaction*), because m^S exists and some negative pre-conditions are found (*vacuous satisfaction*), or because m^S and m exist and the negative pre- and

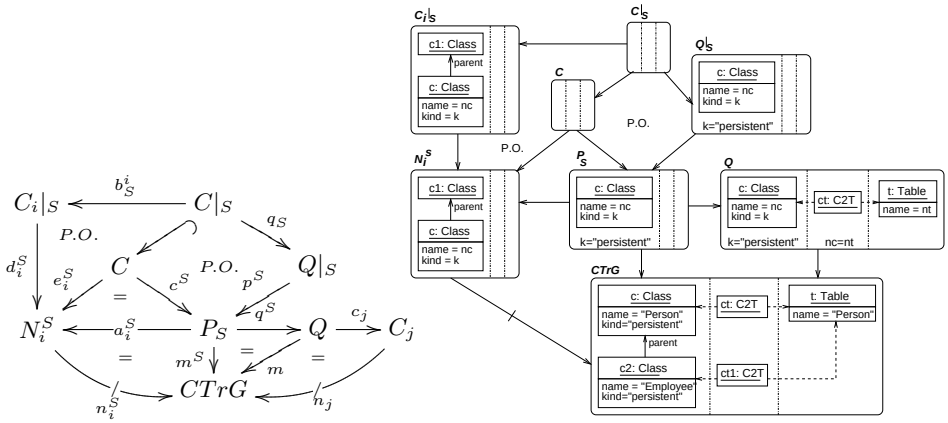

Fig. 7. Forward satisfaction (left). Example (right).

post-conditions are not found (*positive satisfaction*). Note that if the resulting negative pre-condition N_i^x is isomorphic to P_x, it is not taken into account. This is needed as many pre-conditions express a restriction in either source or target but not on both. Similar conditions are demanded for backward satisfiability.

Example. The right of Fig. 7 depicts the satisfaction of pattern `C-T` shown in Fig. 6 by the ground constraint $CTrG$. We have $CTrG \models_F C - T$ as there are two occurrences of P_S, and the first one (shown by equality of identifiers) is positively satisfied, while the second (node $c2$) is vacuously satisfied. We also have $CTrG \models_B C - T$, as there is just one m^T, positively satisfied. Hence $CTrG \models C - T$.

Given a specification $SP = \bigwedge_{i \in I} P_i$ and a constraint $CTrG$, we write $CTrG \models SP$ to denote that $CTrG$ satisfies all patterns in SP. The semantics of a specification is the language of all constraint triple graphs that satisfy it.

Definition 8 (Specification Semantics) *Given a specification SP, its semantics is given by $SEM(SP) = \{CTrG \in Obj(\mathbf{CTrG}_{\mathcal{A}}) | CTrG \models SP\}$, where $Obj(\mathbf{CTrG}_{\mathcal{A}})$ are all objects in the category $\mathbf{CTrG}_{\mathcal{A}}$.*

The semantics is defined as a set of constraint triple graphs, not necessarily ground. Given a non-ground constraint, a solver can obtain a ground constraint satisfying it, if it exists. Moreover, the specification semantics is compositional, as adding new patterns to a specification amounts to intersecting the languages of both. This fact is useful when extending or reusing pattern-based specifications.

Proposition 2 (Composition of Specifications) *Given specifications SP_1 and SP_2, $SEM(SP_1 \wedge SP_2) = SEM(SP_1) \cap SEM(SP_2)$.*

4 Generation of Operational Mechanisms

This section describes the synthesis of TGG operational rules implementing forward and backward transformations from pattern-based specifications. In forward transformation, we start with a constraint triple graph with correspondence and target empty, and the other way round for backward transformation.

The synthesis process creates one rule that contains triple constraints in its LHS and RHS from each P-pattern. In particular, $P_S = C +_{C|_S} Q|_S$ is taken as the LHS for the forward rule and Q as the RHS. The negative pre- and post-conditions of the P-pattern are used as negative pre- and post-conditions of the rule. All N-patterns are converted into negative post-conditions of the rule, using the well-known procedure to convert graph constraints into rule's post-conditions [6]. Finally, additional NACs are added to ensure termination.

Definition 9 (Derived Operational Rules) *Given specification SP and $P = [\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j)] \in SP$, the following rules are derived:*

- **Forward.** $\overrightarrow{r_P}: ((L = C +_{C|_S} Q|_S \rightarrow R = Q), pre^S(P), post(P))$,
- **Backward.** $\overleftarrow{r_P}: ((L = C +_{C|_T} Q|_T \rightarrow R = Q), pre^T(P), post(P))$,

where the sets $pre^x(P)$ (for $x = \{S, T\}$) of NACs are defined as the union of the following two sets:

- $NAC^x(P) = \{L \xrightarrow{a_i^x} N_i^x | L \ncong N_i^x\}_{i \in Pre}$ *is the set of NACs derived from P's negative pre-conditions, with $N_i^x \cong C_i|_x +_{C|_x} C$.*
- $TNAC^x(P) = \{L \xrightarrow{n_j} T_j\}$ *is the set of NACs ensuring termination, where T_j is built by making n_j injective and jointly surjective with $Q \xrightarrow{f} T_j$, s.t. the diagram to the bottom-left of Fig. 8 commutes.*

and the set $post(P)$ is defined as the union of the following two sets of negative post-conditions:

- $POST(P) = \{n_j : R \rightarrow C_j\}_{j \in Post}$ *is the set of rule's negative post-conditions, derived from the set of P's post-conditions.*
- $NPAT(P) = \{R \rightarrow D | [\overrightarrow{N}(C_k)] \in SP,\ R \rightarrow D \leftarrow C_k$ *is jointly surjective, and $(R \setminus L) \cap C_k \neq \emptyset\}$ is the set of negative post-conditions derived from each N-pattern $\overrightarrow{N}(C_k) \in SP$.*

The set $NPAT(P)$ is made of the negative post-conditions derived from the set of N-patterns of the specification. This is done by relating each N-pattern with the rule's RHS in each possible way. Moreover, the requirement that $(R \setminus L) \cap C_k \neq \emptyset$ reduces the size of $NPAT(P)$, because we only need to consider possible violations of the N-pattern due to created elements by the RHS, as we start with an empty target model.

Example. The upper row of Fig. 8 shows the operational forward rule generated from pattern `Attribute-Column`. The set NAC^S contains one constraint, equal to R. There are two NACs for termination, `TNAC2` and `TNAC1`, the latter equal to R. As a difference from Fig. 5, we do not need to do algebraic manipulation of formulae to generate the rule. The figure also shows a direct derivation where both G and H are ground constraints. Note also that we do not check in L that x starts with "_ _", but if it does not, we would obtain an unsatisfiable constraint.

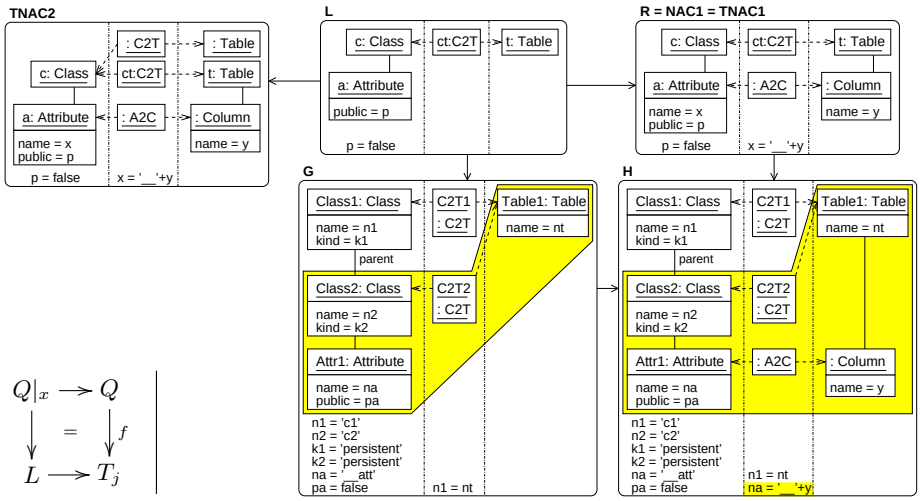

Fig. 8. Condition for $TNAC^x(P)$ (left). Example rule and derivation (right)

According to [12], the generated rules are terminating, and in absence of N-patterns, correct: they produce only valid models of the specification. However, the rules are not complete: not all models satisfying the specification can be produced. Next subsection describes a method, called *parameterization*, that in addition ensures completeness of the rules generated from a specification without N-patterns. If a specification contains N-patterns, these are added as negative post-conditions for the rules, preventing the occurrence of N-patterns in the model. However, they may forbid applying any rule before a valid model is found, thus producing graphs that may not satisfy all P-patterns (because the transformation stopped too soon). That is, in this situation the operational mechanism would not be able to find a model, even if it exists. Next subsection presents one heuristic that ensures finding models, and hence correctness, for mechanisms derived from some specifications with certain classes of N-patterns.

4.1 Parameterization and Heuristics for Rule Derivation

Applying the *parameterization* operation to each P-pattern in the specification ensures completeness of the operational mechanism: the rules are able to generate all possible models of the specification [12]. The operation takes a P-pattern and generates additional ones, with all possible positive pre-conditions "bigger" than the original pre-condition, and "smaller" than the main constraint Q. This allows the rules generated from the patterns to reuse already created elements.

Definition 10 (Parameterization) *Given* $T = \bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \wedge \bigwedge_{j \in Post} \overrightarrow{N}(C_j)$, *its parameterization is* $Par(T) = \{\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C') \Rightarrow P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j) | C \overset{i_1}{\hookrightarrow} C' \overset{i_2}{\hookrightarrow} Q, C \ncong C', C' \ncong Q\}$.

Remark. The formula $\alpha_{C'}$ can be taken as the conjunction of α_C for the variables already present in ν_C, and α_Q for the variables not in ν_C (i.e. in $\nu'_C \backslash i_1(\nu_C)$). Formally, $\alpha_{C'} = \alpha_C \wedge \alpha_Q|_{i_2(\nu_{C'} \backslash i_1(\nu_C))}$ (assuming no renaming of variables).
Example. Fig. 9 shows an example, where some of the parameters generated by parameterization are shown for a pattern like the one in Fig. 6 but without parameters. Parameterization generates 45 patterns in total. The pattern with parameter $\overrightarrow{P}(1)$ is enforced when the class is already mapped to a table, and in forward transformation avoids generating a rule that creates a table with arbitrary name. Parameter $\overrightarrow{P}(3)$ reuses a column with the same name as the attribute (but starting by '__'), possibly created by a parent class. However, $\overrightarrow{P}(2)$ is harmful as it may lead to reusing a column connected to a different table, and thus to an incorrect model.

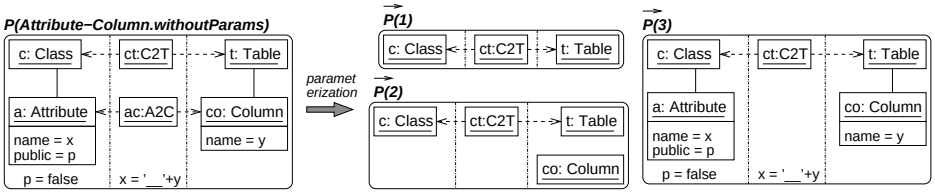

Fig. 9. Parameterization example.

As the example shows, parameterization generates an exponential number of patterns with increasingly bigger parameters, which may lead to operational rules reusing too much information. Although this ensures completeness, we hardly use it in practice, and we prefer using heuristics to control the level of reuse. However, as previously stated, generating fewer patterns can make the rules unable to find certain models of the specifications (those "too small").

We propose two heuristics in this paper. The first one is used to derive only those parameters that avoid creation of elements with unconstrained attribute values. The objective is to avoid synthesizing rules creating elements whose attributes can take several values.

Heuristic 1 *Given a pattern P, replace it by a new pattern that has as parameter all elements with attributes not constrained by any formula, and the mappings between these elements.*

Example. In Fig. 9, the heuristic generates just one pattern with parameter $\overrightarrow{P}(1)$. Thus, the generated rules avoid creating a table with arbitrary name.

Next heuristic generates only those parameters that avoid duplicating a graph S_1, forbidden by some N-pattern of the form $\overrightarrow{N}(S_1 +_U S_1)$, forbidding the duplication of a subgraph S_1. This ensures the generation of rules producing valid models for the class of specifications with N-patterns of this form (called FIP in [3]).

Heuristic 2 *Given* $[\overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q)]$, *if* $[\overrightarrow{N}(S)] \in SP$ *with* $S \cong S_1 +_U S_1$, *and* $\exists s\colon S_1 \to Q$ *and* $\nexists s'\colon S_1 \to C$ *both injective s.t.* $q \circ s' = s$, *we generate additional patterns with parameters all* C'_j *s.t.* q_1 *and* q_s *in* $C \overset{q_1}{\to} C'_j \overset{q_s}{\leftarrow} S_1$ *are jointly surjective, and the induced* $C'_j \to Q$ *is injective.*

The way to proceed is to apply heuristic 2 to each P- and N-pattern of the form $\overrightarrow{N}(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated. Next section illustrates both heuristics.

5 Example

Next we illustrate our approach with a bidirectional transformation between relational database schemas (RDBMS) and XML documents. Their meta-models are shown to the left and right of the meta-model triple in Fig. 10. Schemas contain books and subjects. A book has zero or more subjects, and those books with the same subject description are related to the same object *Subject*. On the contrary, the XML meta-model allows nested relationships, and even if two books have the same subject description, they are assigned two different objects *Subject*.

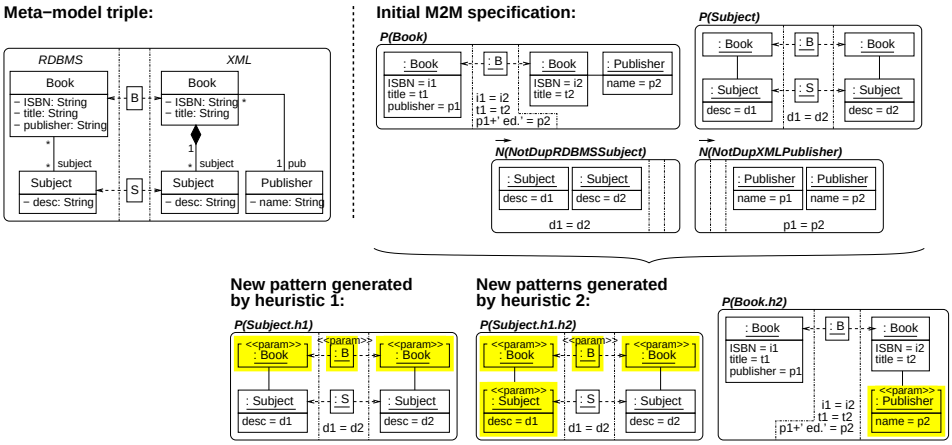

Fig. 10. Mapping relational database schemas and XML.

Fig. 10 shows the initial M2M specification, which is made of four patterns. The P-pattern `Book` states how the books in both meta-models should relate, and adds an ".ed" suffix to the publisher in the XML model. P-pattern `Subject` maps subjects in both models. Note that we need these two patterns as it is possible to have books with zero or more subjects. Should a book have exactly one subject, then only one pattern would have been enough. In addition, as the RDBMS format does not allow two subjects with the same description, we

forbid such situation by defining the N-pattern `NotDupRDBMSSubject`. Similarly, N-pattern `NotDupXMLPublisher` forbids repeating publishers in XML.

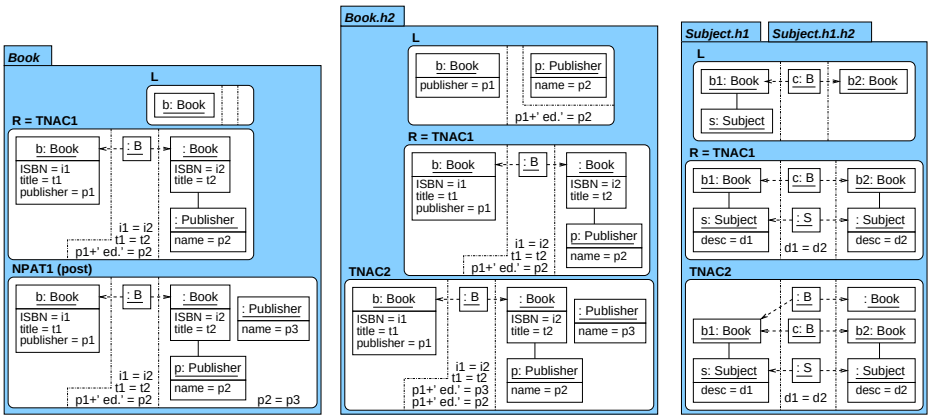

Fig. 11. Generated forward rules.

In this example we cannot use generic parameterization as it would generate patterns with parameters reusing, e.g. the *Subject*s in the XML model. Therefore we use the heuristics instead. The first one generates pattern `Subject.h1` from pattern `Subject` by defining the elements with unconstrained attributes as parameters. The new pattern replaces the old one and ensures that, when the subject is translated, the book associated to it has been translated first. The second heuristic is applied to patterns `Subject.h1` and `Book` and produces two new patterns, `Subject.h1.h2` and `Book.h2`. The first one reuses RDBMS *Subject*s so that they are not duplicated in backward transformations. The second reuses one *Publisher*, avoiding its duplication in forward transformations.

As a last step, we use patterns `Book`, `Subject.h1`, `Subject.h1.h2` and `Book.h2` and the N-patterns to generate the operational rules. Fig. 11 shows the forward ones. Rule `Book` contains a termination NAC (`TNAC1`) equal to its RHS and a negative post-condition (generated from $\overrightarrow{N}$(`NotDupXMLPublisher`)) avoiding two publishers with same name. Patterns `Subject.h1` and `Subject.h1.h2` produce equivalent rules with two termination NACs. Finally, rule `Book.h2` creates books that reuse publishers once they have been created. Note again that we do not need to perform algebraic manipulation of expressions for rule synthesis, as the LHSs and RHSs contain constraint triple graphs (where note that attributes not used in formulae are ommited, like in the LHS of rule *Book*).

Altogether, the operational mechanisms generated for this example are terminating, confluent, correct and complete even using heuristics. However, our mechanisms cannot guarantee confluence in general if we do not have a means to prefer one resulting model or another.

6 Related Work

Some declarative approaches to M2M transformation use a textual syntax, e.g. PMT [15], Tefkat [9]. All of them are uni-directional, whereas our patterns are bidirectional. There are also bidirectional textual languages, like MTF [10].

Among the visual declarative approaches, a prominent example is QVT-relational [13]. Relations in QVT may include *when* and *where* clauses that identify pre- and post-conditions and can refer to other relations. From this specification, executable QVT-core is generated. This approach is similar to ours, but we compile patterns to TGG rules, allowing its analysis [6]. Besides, we can analyse the patterns themselves. In the QVT-relations language, there is no equivalent to our N-patterns. Notice however, that our N-patterns can be used to model *keys* in QVT (e.g. elements that should have a unique identifier) as we showed in the example of the previous section with N-patterns $\overrightarrow{N}$(`NotDupXMLPublisher`) and $\overrightarrow{N}$(`NotDupRDBMSSubject`). An attempt to formalize QVT-core is found in [7].

In [1], transformations are expressed through positive declarative relations, heavily relying on OCL constraints, but no operational mechanism is given to enforce such relations. In BOTL [2], the mapping rules use a UML-based notation that allows reasoning about applicability or meta-model conformance.

Declarative TGGs [14] formalize the synchronized evolution of two graphs through declarative rules from which TGG operational rules are derived. We also generate TGG operational rules, but whereas declarative TGG rules must say which elements should exist and which ones are created, our heuristics infer such information. Moreover, TGGs need a control mechanism to guide the execution of the operational rules, such as priorities [8] or their coupling to editing rules [5], while our patterns do not need it. As in QVT, there is no equivalent to our N-patterns, however TGGs can be seen as a subset of our approach, where a declarative TGG rule is a pattern of the form $\overleftarrow{P}(L) \Rightarrow P(R)$.

In [11] the authors start from a forward transformation and the corresponding backward transformation is derived. Their transformations only contain injective functions to ensure bidirectionality, and if an attribute can take several values one of them is chosen randomly. Finally, in [4] attribute grammars are used as transformation language, where the order of execution of rules is automatically calculated according to the dependencies between attributes.

7 Conclusions and Future Work

In this paper we have extended pattern-based transformation with attributes. The resulting language allows expressing relations between models in a declarative way, leaving open the kind of logic used for attribute conditions. Typically, it can be first order predicate logic, e.g. with OCL syntax. The advantage of our approach is that it provides a formal, high-level language to express bidirectional transformations. Our language is concise, as its heuristics allow omitting the parameters in the relations. Moreover, at the operational level, we have proposed a new way of triple graph rewriting based on constraints. This idea, which can

be used in other transformation approaches, avoids manipulation of attribute conditions, one of the main difficulties of relational approaches.

We are currently working towards using this approach to formalize QVT relations. Also, we are considering other operational languages, further heuristics, devising analysis methods, and implementing a prototype tool.

Acknowledgments. Work supported by the Spanish Ministry of Science and Innovation, projects METEORIC (TIN2008-02081) and MODUWEB (TIN2006-09678). We thank the referees for their useful comments.

References

1. D. H. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *UML'02*, volume 2460 of *LNCS*, pages 243–258. Springer, 2002.
2. P. Braun and F. Marschall. Transforming object oriented models with BOTL. *ENTCS*, 72(3), 2003.
3. J. de Lara and E. Guerra. Pattern-based model-to-model transformation. In *ICGT'08*, volume 5214 of *LNCS*, pages 426–441. Springer, 2008.
4. M. Dehayni and L. Féraud. An approach of model transformation based on attribute grammars. In *OOIS*, volume 2817 of *LNCS*, pages 412–424. Springer, 2003.
5. H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *FASE'07*, volume 4422 of *LNCS*, pages 72–86. Springer, 2007.
6. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
7. J. Greenyer. A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, University of Paderborn, 2006.
8. A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148(1):113–150, 2006.
9. M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.
10. MTF. Model Transformation Framework. `http://www.alphaworks.ibm.com/tech/mtf`.
11. S.-C. Mu, Z. Hu, and M. Takeichi. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23(2):129–141, 2006.
12. F. Orejas, E. Guerra, J. de Lara, and H. Ehrig. Correctness, completeness and termination of pattern-based model-to-model transformation. Submitted, available at `http://astreo.ii.uam.es/~jlara/papers/compPBT.pdf`, 2009.
13. QVT. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.
14. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
15. L. Tratt. A change propagating model transformation language. *JOT*, 7(3):107–126, 2008.