Controlling Reuse in Pattern-Based Model-to-Model Transformations

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
Juan.deLara@uam.es
[3] Universitat Politècnica de Catalunya, Spain
orejas@lsi.upc.edu

Abstract. Model-to-model transformation is a central activity in Model-Driven Engineering that consists of transforming models from a source to a target language. Pattern-based model-to-model transformation is our approach for specifying transformations in a declarative, relational and formal style. The approach relies on patterns describing allowed or forbidden relations between two models. These patterns are compiled into operational mechanisms to perform forward and backward transformations.

Inspired by QVT-Relations, in this paper we incorporate into our framework the so-called check-before-enforce semantics, which checks the existence of suitable elements before creating them (i.e. it promotes reuse). Moreover, we enable the use of *keys* in order to describe when two elements are considered equal. The presented techniques are illustrated with a bidirectional transformation between Web Services Description Language and Enterprise Java Beans models.

1 Introduction

Model-Driven Engineering (MDE) [28] proposes the construction of software systems using models as primary artefacts. In this paradigm, models are used to specify, reason, generate code, document, test, analyse and maintain the final application. Hence, model transformation becomes a key enabling technology for MDE, and is being subject of intensive research nowadays. The Model-Driven Architecture [17] (MDA) is a particular incarnation of MDE promoted by the OMG, which proposes the use of its standard languages, like MOF for metamodelling and QVT [23] (Query/View/Transformation) for transformations.

Model-to-Model (M2M) transformation involves transforming models from a source to a target language. In the context of MDE, M2M transformations are used e.g. to migrate between language versions, to refine a model, or to transform a model into a semantic domain for analysis. Several usage scenarios can be identified. Source-to-target (resp. target-to-source) transformations assume the existence of a source (resp. target) model and create a target (resp. source) model from scratch. *Incremental* transformations optimize the former, so that

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 175–201, 2010.

if the source (resp. target) model is changed after being transformed, the target (resp. source) is updated but not regenerated. A further step is *model synchronization*, where both models can be modified at any time, and the changes are propagated to the other model to recover consistency. Hence, a sensible approach is to define a unique specification establishing when two models are consistent, and then generate specific lower-level *operational* mechanisms to solve the scenario of interest. This has the advantage that the transformation is specified only once, but it requires using a bidirectional, declarative style of specification. Moreover, the synthesis of operational mechanisms may imply complex algebraic manipulations of the declarative attribute conditions appearing in the transformation specification.

Even though many transformation languages have been proposed in the literature [23,25], there is a need for expressive, high-level, and formal languages able to precisely express the M2M consistency problem and enabling the analysis of the transformation specifications. Following the ideas of [25], but aimed at a relational style of specifications in the lines of [23,27], in [4] we developed a new approach for specifying M2M transformations. The approach is based on *patterns* describing positive or negative conditions that are to be satisfied by two models in order to be considered consistent. Patterns have a high-level (i.e. independent of the operational mechanism), algebraic semantics enabling the decision of whether two models are consistent, or to find the discrepancies with respect to the specification. These patterns are then compiled into operational mechanisms, based on triple graph grammar rules [4,12], but in which no algebraic manipulation of attribute formulae is necessary.

We believe our framework can be used to formalise other transformation languages, especially QVT-Relations (QVT-R). The purpose of this paper is to advance in this direction. With this aim, we extend our previous works [4,12,21] by bringing into our framework two concepts of QVT-R: the Check-Before-Enforce (CBE) semantics and the *keys*. CBE semantics is a way to promote element reuse in transformations, and to enable many-to-one relations between elements across models. In particular, before creating an element, it is checked whether an existing one can be reused. Keys allow specifying when two elements are considered equal. Moreover, in order to promote the use of our techniques in MDE, we propose a way to enrich the transformation specification with integrity constraints of the source and target meta-models, in particular the association cardinality constraints. Finally, we extend our patterns by allowing *abstract objects*. These features are illustrated through a transformation between Web Service Description Language [29] (WSDL) models and Enterprise Java Beans [19] (EJBs).

Paper organization. Section 2 introduces related work. Section 3 presents the case study we will use throughout the paper. Section 4 recalls the necessary background for subsequent sections. Section 5 reviews our notion of patterns, and Section 6 the generation of source-to-target and target-to-source operational mechanisms. Then, Section 7 incorporates the CBE semantics and keys into our framework. Section 8 presents further details of the case study and, finally,

Section 9 ends with the conclusions. An appendix presents some of the proofs of the main claims and propositions.

2 Related Work

Bidirectional transformation languages are receiving increasing attention in MDE, as they are able to capture consistency relations between two models in a direction-independent way. In this approach, a unique specification is used to derive operational mechanisms solving the different synchronization scenarios mentioned in the introduction (see also [16]).

A prominent example of this kind of languages is QVT-R [23], a part of the QVT family of transformation languages sponsored by the OMG. A QVT-R specification is made of relations, each consisting of two or more domains (i.e. models). Relations can be *top* or *non-top* level, and include *when* and *where* clauses that may be used to express dependencies between relations. The execution of a transformation requires that all its top-level relations hold, whereas the non-top level ones only need to hold when invoked from the *where* section of other relations. The standard specifies that QVT-R models are enforced by its compilation into QVT-core, a lower-level language. While QVT-R has no explicit notion of traces (i.e. relations between the model elements involved in the transformation), the compilation to QVT-core creates them automatically.

In [1], transformations are expressed through declarative relations made of positive patterns, 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. In our approach we can reason both at the specification and operational levels. In [6], the authors rely on completely relational transformation units and infer the order of execution by studying their dependencies. They use attribute grammars as (uni-directional) transformation language. This kind of grammars is made of textual relations where the order of execution of rules is not given, but it is automatically calculated in accordance with the dependency relations that arise between attributes. In the MTF language from IBM [18], transformations are made of textual relations expressed in RDL (the *Relations Definition Language*) that do not impose a direction of the transformation, but this is selected when invoking the transformation engine. Similar to QVT-R, MTF relations must be invoked from other relations in order to be executed, whereas in our approach we *query* the trace model.

TGGs [25] formalize the synchronized evolution of two graphs through declarative rules. The language spawned by these rules contains the pairs of models considered consistent. From this specification, low-level operational rules are derived to solve different synchronization scenarios. Interestingly, our patterns define a language of valid consistent models by means of constraints instead of rules (even though the operational mechanisms are implemented through operational rules) and hence we admit negative constraints too. The work in [15] included in this volume improves previous works on TGGs by considering TGG

schemas (meta-model triples in our jargon) with so called monotonic constraints (which if satisfied by a model, are satisfied by any submodel). These constraints are similar to our notion of N-patterns, but our N-patterns are not necessarily monotonic. Also, they propose a guiding mechanism for applying the operational rules by a so called *dangling edge condition*, which before applying one transformation rule checks if some edge in the source will not get translated. In our case, we assume that not every element in the source needs to be translated, but the fact that we generate several rules for each pattern permits obtaining all valid target models, if more than one exists [21]. An attempt to bridge TGGs and QVT-R is [10], where QVT-R is both compiled into operational TGGs (instead of using QVT-core) and translated into declarative TGGs.

An interesting issue in these languages is how to handle and express object reuse. This aspect has been tackled in QVT-R by the CBE semantics, where the operational mechanism checks which objects exist and can be reused before creating them. In order to specify when two elements are considered equal, one can set *keys* (similar to keys in databases). Reuse has to be handled explicitly in TGGs by including the objects to be reused in the left-hand side (LHS) of the declarative rules. Up to now, our patterns followed a similar approach by defining the objects to reuse as positive pre-conditions. It is interesting however to decouple the specification of the reusing policy (keys in QVT) from the specification of the transformation itself, which potentially leads to more flexible and reusable transformations. In this paper we incorporate these ideas into our framework.

3 The Example Case Study

In this section we introduce the case study that we will use throughout the paper, namely the transformation between WSDL documents and EJBs, both represented as models. WSDL [29] is an XML-based language for describing web services, endorsed by the W3C. Here we use the last version 1.1, which is the most widely used by tools. Fig. 1 shows a simplified meta-model for WSDL we have developed taking as a basis the XML syntax described in [29].

A WSDL model includes the definition of services as collections of network endpoints, or ports (class `Port` in the meta-model). Ports and messages are described in an abstract way through classes `PortType` and `Message`, independently from their concrete usage. Then, a *binding* provides the concrete information (addresses, protocols – normally HTTP – and so on) to use the services through their ports. The binding is usually done through SOAP [30], although for simplicity we have omitted the classes for the binding from the meta-model. A `PortType` defines a number of operations (similar to functions in programming languages) that the service exposes, modelled by class `Operation`. There are four types of operation, defining a protocol for exchanging messages. For example, while operations of type `OneWay` just receive one message, `RequestResponse` operations in addition send a response back. The operations refer to the messages involved,

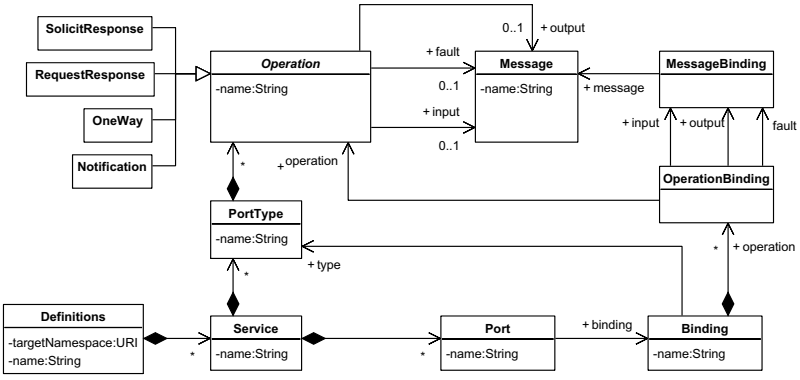


Fig. 1. WSDL meta-model (simplified)

either input, output or fault. A `Message` has a structure made of several logical parts, omitted here for simplification.

Enterprise JavaBeans (EJBs) [24] is a Java API that defines a component architecture to build server-side enterprise applications. Its specification provides a number of services commonly found in these applications, like persistence, transaction processing, concurrency control, security and exposing business methods as web services, among others. Fig. 2 shows a simplification of its meta-model we have developed taking [19] as a basis. An EJB container (class `EJBJar`) can hold a number of beans (i.e. Java components), the most important types of which are `Session` and `Entity`. The former are distributed objects that can have a state or not, depending on whether their attribute `sessionType` takes the value `stateful` or `stateless`, respectively. Stateful beans keep track of the calling process through a session, and hence a different bean instance is created for each customer. On the contrary, stateless beans enable concurrent access. Entity beans (class `Entity`) represent persistent data maintained in a database[1]. For simplicity, we have omitted the details of this kind of beans.

EJBs are deployed in an application server. Each EJB has to provide a Java implementation class, and two interfaces called *Home* and *Remote*. The meta-model in Fig. 2 contains a high-level Java meta-model that reflects the dependency of EJBs to Java.

In our case study, we are interested in specifying a bidirectional transformation between WSDL and EJB models. This is useful as, when building EJB applications, it is sometimes needed to expose them as web services, and hence to generate a WSDL file with the service description. The generated operational (backward) transformation would do this automatically. The opposite is also common, sometimes a WSDL file with the description of a service needs to be implemented. The generated operational (forward) transformation would

[1] We use the EJB1.1 specification; in the EJB3.0 specification Entity Beans were superseded by the Java Persistence API.

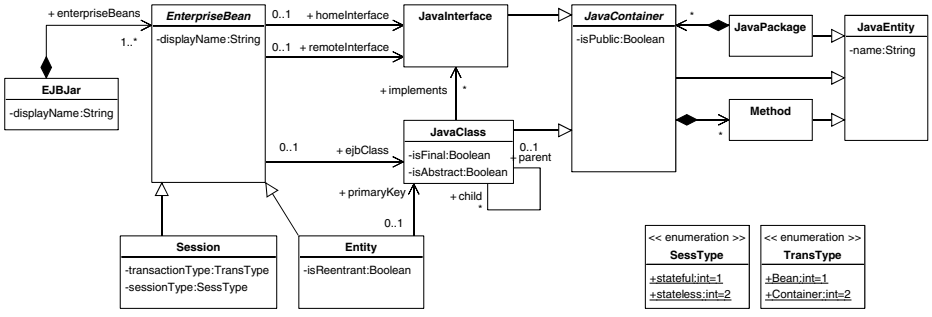


Fig. 2. EJB meta-model (simplified)

synthesize an EJB model containing skeletons of the necessary Java classes and interfaces. There are already available tools that perform these tasks. For example, the Oracle Containers for J2EE (OC4J) [20] has a tool called `wsdl2ejb` that generates an EJB from a WSDL file. Similarly, the IBM Websphere application server [13] provides the `EJB2WebService` tool to create a web service (including the WSDL file) from EJBs. Note however that both tools are not incremental and overwrite existing files. Our method has the potential to be incremental and moreover it would generate both tools starting from a single specification.

4 Preliminaries

In this section we introduce the basic theoretical concepts (triple graphs and constraint triple graphs) that we will use in our M2M specification language.

In order to perform M2M transformations, it is useful to consider structures made of a source and a target model, related through a trace model. This structure is called *triple graph* [25]. As we can provide nodes and edges in graphs with attributes and types (called E-graphs in [8]), models can be naturally encoded with graphs. An E-graph is a tuple $G = (V^G, D^G, E^G, E^G_{NA}, E^G_{EA}, (src^G_j, tar^G_j)_{j\in\{G,NA,EA\}})$, where V^G and D^G are sets of graph and data nodes, E^G is a set of graph edges, E^G_{NA} and E^G_{EA} are sets of edges modelling attributes for both nodes and edges, functions $src^G_G\colon E^G \to V^G$ and $tar^G_G\colon E^G \to V^G$ are the graph edge source and target functions, $src^G_{NA}\colon E^G_{NA} \to V^G$ and $tar^G_{NA}\colon E^G_{NA} \to D^G$ are the source and target functions for node attributes, and $src^G_{EA}\colon E^G_{EA} \to E^G$, $tar^G_{EA}\colon E^G_{EA} \to D^G$ are the functions for edge attributes. Even though we use E-graphs in our triple graphs, any other type of graph could also be used. Graphs can be typed by a type graph TG (similar to a meta-model) [8] becoming objects of the form $(G, type\colon G \to TG)$, where *type* is a typing function.

Hence, 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 [7].

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 *called 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}).*

Remark. The correspondence graph is restricted to be unattributed (i.e. $D^C = \emptyset$), but not necessarily discrete. This is so because otherwise, in general, we could not take c_S and c_T to be graph morphisms, as the conditions for attributes fail.

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. The left of Fig. 3 shows a triple graph relating a WSDL model and an EJB model. The graph nodes are depicted as rectangles, and the data nodes in D^S and D^T as rounded rectangles. We only draw the used data nodes, as they may be infinite. We have represented the types of nodes after a semicolon. In fact, a triple graph is typed by a type triple graph (or meta-model triple [11]), where the typing morphism is a triple graph morphism. The right of the same figure shows the triple graph in UML notation, which we will use throughout the paper.

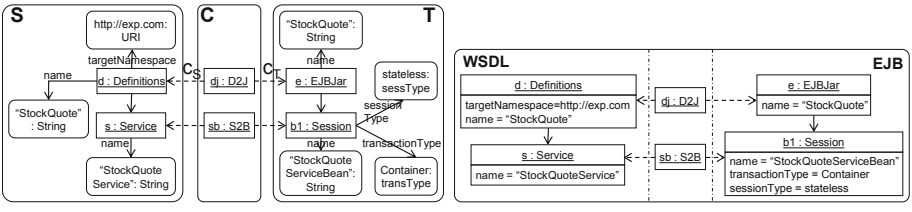


Fig. 3. Triple graph example in theoretical (left) and compact notations (right)

Next, we present the notion of *constraint triple graph* [12]. It will be used later as a building block of our patterns, as a way to express desired relations between the source and target models, and also in the left and right hand sides of the generated TGG operational rules. Constraint triple graphs are triple graphs attributed over a finite set ν of variables, and equipped with a formula on this set (i.e., a $\Sigma(\nu)-$formula, where Σ is a signature) 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.*

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 $(x = 3) \wedge \neg(y = 7)|_{\{x\}} = (x = 3)$, as we substitute $\neg(y = 7)$ by **true**.

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 $\mathfrak{f}$. Note that if $\mathcal{A} \models_{\mathfrak{f}} \alpha$, then $\mathcal{A} \models_{\mathfrak{f}} \alpha|_{\nu'}\ \forall \nu' \subseteq \nu$.

Morphisms between constraint triple graphs 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 may 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. 4 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 $m^{\nu}(X)$.*

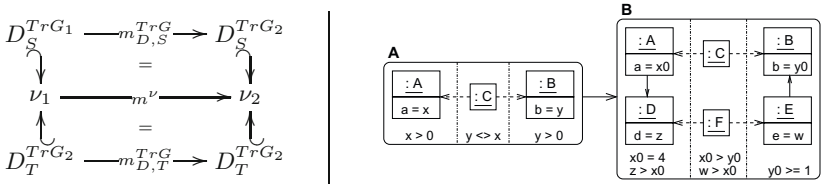


Fig. 4. Condition for **CTrG**-morphisms (left). Example (right).

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 we need to build source and target constraint *restrictions* (see below) and obtain a morphism from the restricted constraint to the full constraint.

Example. The right of Fig. 4 shows a constraint triple graph morphism. Concerning the formula, if we 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 $\mathcal{A} \models_{\mathfrak{f}} [(x_0 = 4 \wedge z > x_0) \Rightarrow (x_0 > 0)] \wedge [(y_0 >= 1) \Rightarrow (y_0 > 0)] \wedge [(x_0 = 4 \wedge z > x_0 \wedge x_0 > y_0 \wedge w > x_0 \wedge y_0 >= 1) \Rightarrow (x_0 > 0 \wedge y_0 <> x_0 \wedge y_0 > 0)]$. Thus, the formula in A (the morphism domain) is weaker or equivalent to the formula in B (the morphism codomain).

From now on, we restrict to injective morphisms (for simplicity, and because our 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 of 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 $(\nu_B +_{\nu_A} \nu_C)$.*

Proof. In appendix.

Example. Fig. 5 shows a pushout, where the pushout object D is the result of gluing the constraint triple graphs B and C along the constraint triple graph A, written $B +_A C$. In particular, the resulting constraint has the common nodes A, B and C, whereas graph B adds node D, and graph C adds node E. The formula of D α_D includes the conjunction of the formulas of graphs B and C, and note that $\alpha_D \Rightarrow c'(b(\alpha_A)) \equiv b'(c(\alpha_A))$.

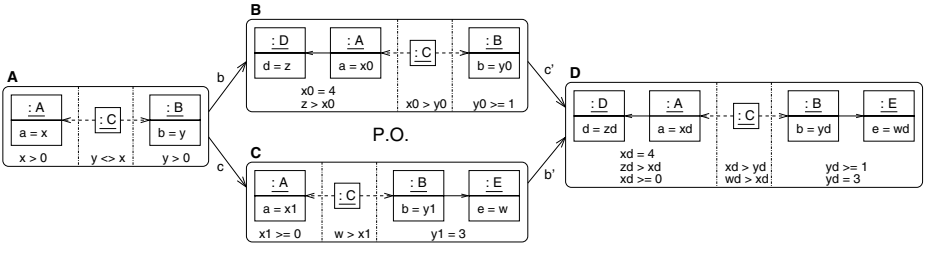


Fig. 5. Pushout example

Sometimes, we have to consider the source or the target parts of a constraint triple graph. The source restriction of a constraint triple graph $CTrG^{\mathcal{A}}$, written $CTrG^{\mathcal{A}}|_S$, is made of the source graph and the source formula, and similarly for the target restriction. Hence, $CTrG^{\mathcal{A}}|_S = (TrG|_S = \langle S, \emptyset, \emptyset \rangle, D^S, \alpha|_{D^S} = \alpha^S)$. 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 $q \colon CTrG_1^{\mathcal{A}} \to CTrG_2^{\mathcal{A}}$, we can construct morphism $q_S \colon CTrG_1^{\mathcal{A}}|_S \to CTrG_2^{\mathcal{A}}|_S$ and similarly for the target. This restriction operation will be used later to consider only the source or target models in a constraint, when such constraint is evaluated source-to-target or target-to-source.

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. The equivalence between ground constraints and triple graphs is useful as, from now on, we just need to work with constraint triple graphs.

5 Pattern-Based Model-to-Model Transformation

Now we use the previous concepts to build our M2M specification language. Specifications in our language are made of so called *triple patterns*. These are similar to graph constraints [8], but made of constraint triple graphs instead of graphs. This allows interpreting them both source-to-target and target-to-source.

We consider two kinds of pattern: positive (called P-patterns) and negative (N-patterns). While the former express allowed relations between source and target models, the latter describe forbidden scenarios. A P-pattern has a main constraint (written $P(Q)$), a (possibly empty) positive pre-condition C (written $\overleftarrow{P}(C)$), a set of negative pre-conditions (written $\overleftarrow{N}(C_i)$), and a set of negative post-conditions (written $N(C_j)$). The main constraint of a P-pattern only needs to hold when the positive pre-condition and no negative pre-condition of the pattern hold. If such is the case, then no negative post-condition of the pattern should hold. An N-pattern is a particular case of P-pattern where C and Q are empty, and there is only one negative post-condition $N(C_j)$ which is forbidden to occur (as any negative post-condition). Next definition formalises the syntax of patterns, while Definition 5 describes their semantics.

Definition 4 (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:*

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

Remark. The notation $\overleftarrow{P}(\cdot)$, $\overleftarrow{N}(\cdot)$, $N(\cdot)$ 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 for the pattern to be satisfied. In this way, while negative pre-conditions express restrictions for the constraint Q to occur, negative 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 positive parameters, specified with a non-empty C. In such a 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 C and Q are empty.

Example. Fig. 6 shows some patterns specifying the consistency between WSDL and EJB models. The P-pattern *P(Definitions-EJBJar)* declares that each `Definitions` object has to be related with an `EJBJar` with same name. This means that the service described in a WSDL document will be handled by a set of related EJBs, bundled in the same *jar* container. The P-pattern *P(Service-SessionBean)* states that each WSDL `Service` is managed by a `Session` bean, made of home and remote interfaces and an implementation class, and with methods to create and initialize the bean. The attribute condition enforces some naming conventions for these. Note that some attribute details (e.g. whether the bean is stateful or stateless) are left open. The pattern has a positive precondition C, which we show in compact notation using *param* tags.

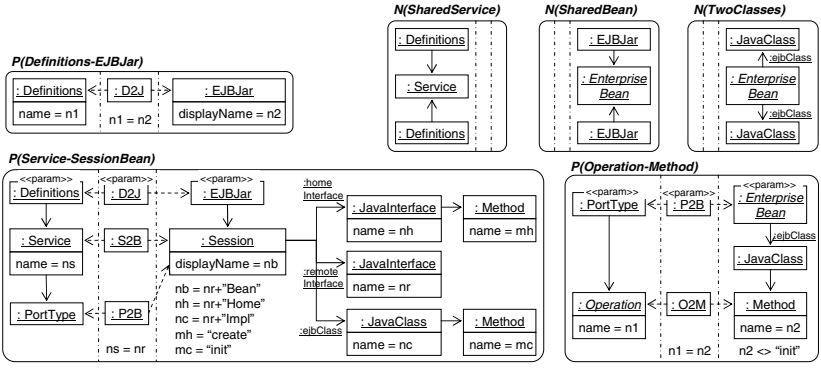


Fig. 6. Some patterns for the WSDL-EJB transformation

The P-pattern *P(Operation-Method)* declares that each `Operation` in a given port type is to be implemented as an EJB `Method` with same name in the bean implementing the port type. The target attribute condition *n2<> "init"* avoids translating the special `init` method created by pattern *P(Service-SessionBean)* back into an operation. In addition, the pattern uses *abstract objects* of types `EnterpriseBean` and `Operation`. This is allowed and, intuitively, it is equivalent to the disjunction of the eight patterns that result from the substitution of the abstract objects by all its concrete subtypes. Thus, the `PortType` may be connected either with a `Session` or with an `Entity`, and the method with any subtype of `Operation`. Finally, three N-patterns forbid `Service`s to belong to two `Definition`s, and an `EnterpriseBean` to belong to two `EJBJar`s and have two `JavaClass`es. Later we will see that in fact these N-patterns can be automatically derived from the meta-models, and also that there is no need to manually specify the parameters in the P-patterns *P(Service-SessionBean)* and *P(Operation-Method)*.

Next, we define pattern satisfaction. Since N-patterns are a special case of P-patterns, a unique definition is enough. Satisfaction is checked 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 a solver can find an attribute assignment satisfying the formulae.

We define forward and backward satisfaction. In the former we check that the main constraint of a pattern is found in all places where the pattern is source-enabled. That is, 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 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. For simplicity, we only enunciate forward satisfaction, see the full definition in [12].

Definition 5 (Satisfaction). *A constraint triple graph $CTrG$ satisfies $CP = [\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j)]$, 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, *$CTrG \models_B CP$, see [12]*

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 $C|_x$.

Remark. We use the notation $A \cong B$ to denote that A and B are isomorphic, and $A \ncong B$ to denote that A and B are not isomorphic.

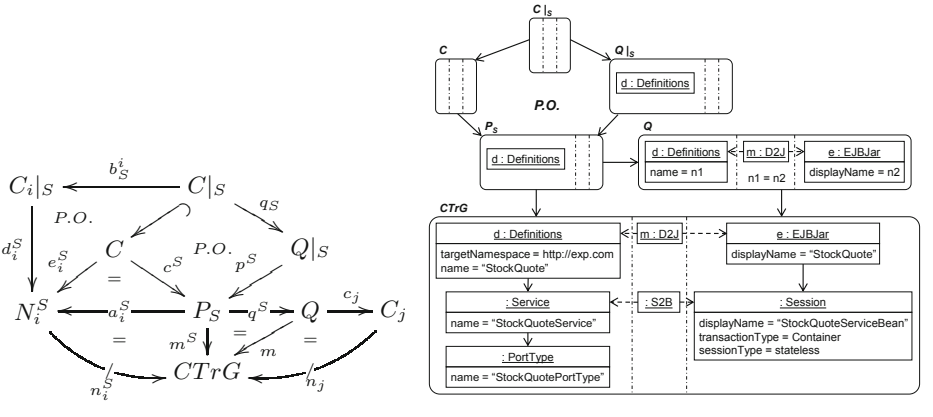


Fig. 7. Forward satisfaction (left). Example (right).

Example. The right of Fig. 7 shows an example of forward satisfaction of pattern *P(Definitions-EJBJar)* by a ground constraint triple graph *CTrG*. There is one occurrence of the source restriction of the pattern in *CTrG*, which can be extended to the whole pattern. In addition, *CTrG* also backward-satisfies the

pattern and hence it satisfies it. Note however that *CTrG* does not forward-satisfy pattern *P(Service-SessionBean)* as the session bean does not define the required java classes and interfaces. The satisfaction checking of a pattern with abstract objects is the same as that of a pattern without them, thus enabling the usual allowed substitution of abstract types by concrete ones.

We can distinguish several kinds of pattern satisfaction. In *trivial satisfaction*, a pattern is satisfied because no morphism m^S exists (i.e. there is no occurrence of the source restriction of the pattern). This is for example the case of pattern *P(Operation-Method)* in the constraint *CTrG* of Fig. 7, as there is no `Operation` object in the source of the constraint. In *vacuous satisfaction*, a pattern is satisfied because m^S exists but some of its negative pre-conditions are also found. In this case, the main constraint Q of the pattern is not demanded to occur in *CTrG*. Finally, in *positive satisfaction*, m^S and m exist and the negative pre- and post-conditions are not found. All these three cases are handled by Definition 5.

One M2M specification is a conjunction of patterns, and hence a constraint triple graph satisfies a specification if it satisfies all its patterns.

5.1 Considering the Meta-model Integrity Constraints

A transformation specification cannot be oblivious of the meta-model integrity constraints. The simplest ones are the maximum cardinality constraints in association ends. These induce N-patterns that in this paper we automatically derive and include in the transformation specification. This is useful to prevent the operational mechanisms from generating syntactically incorrect models, as N-patterns will be transformed into post-conditions of the operational rules.

The generation procedure is simple: if a class `A` is restricted to be connected to a maximum of `j` objects of type `B`, then we build an N-pattern made of an `A` object connected to `j+1` `B` objects. As an example, Fig. 6 showed three N-patterns that were derived from the WSDL and EJB meta-model constraints.

Note that additional (but restricted) forms of OCL could also be transformed, and here we can benefit from previous works on translating OCL into graph constraints [31]. Interestingly, once the meta-model constraints are expressed in the form of patterns, we can analyse their consistency with the rest of the specification. For example, if we find a morphism from some of the generated N-patterns to an existing P-pattern, then we can conclude that the transformation is incorrect, as it could try to create models violating the cardinality constraints. We plan to develop further static analysis techniques, similar to those of [22].

6 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 an initial constraint triple graph with correspondence and target empty, and the other way round for backward transformation. Moreover, we also assume that the source or target initial models

do not violate any N-pattern of the specification. Recall that some of these N-patterns are derived from the maximum association cardinality constraints in meta-models, and hence it is reasonable to assume syntactically correct starting models.

The synthesis process derives one rule from each P-pattern, made of triple constraints in its LHS and RHS. In particular, $P_S = C +_{C|_S} Q|_S$ is taken as the LHS for the forward rule, and the main constraint Q as the RHS. As an example, Fig. 8 shows the LHS and RHS of the forward rule derived from pattern *P(Definitions-EJBJar)*.

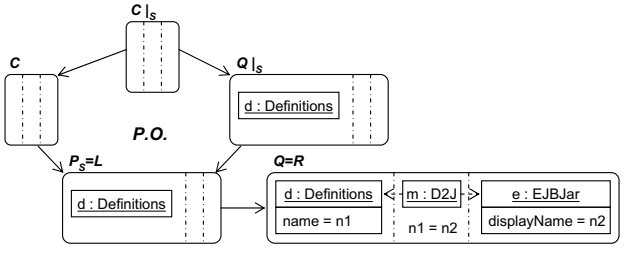


Fig. 8. Forward rule generation example

If a rule creates objects having a type with defined subtypes, we generate a set of rules resulting from substituting the type by all its concrete subtypes in the graph created by the rule, i.e. the nodes in $RHS \setminus LHS$. This substitution is not necessary in the elements of the LHS as they are not created, and it is not done in the NACs either in order to obtain the expected behaviour of disjunction. Using an optimization similar to [3], one could also work directly with abstract rules, but we would have to modify the notion of morphism and it is left for future work.

The negative pre- and post-conditions of a P-pattern are used as negative pre- and post-conditions of the associated rule(s). All N-patterns are converted into negative post-conditions of the rule(s), using the well-known procedure to convert graph constraints into rule's post-conditions [8]. Finally, additional NACs are added to ensure termination. For simplicity, we only show the generation of the forward rules, the backward rules are generated analogously [12].

Definition 6 (Derived Forward Rule). *Given specification SP and $P = [\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j\in Post} N(C_j)] \in SP$, the set of forward rules $\overrightarrow{r_P} = \{((L = C +_{C|_S} Q^n|_S \overset{r^n}{\rightarrow} R^n = Q^n), pre^n(P), post^n(P))\}_{n\in Conc(P)}$ is derived, where $\{L \overset{r^n}{\rightarrow} Q^n\}_{n\in Conc(P)}$ is the set of rules $L \rightarrow Q^n$ resulting from all valid substitutions of types by concrete subtypes in nodes belonging to $V^Q \setminus r(V^L)$. The set $pre^n(P)$ of NACs is defined as the union of the following two sets:*

- $NAC(P) = \{L \overset{a_i^S}{\to} N_i^S | L \ncong N_i^S\}_{i \in Pre}$ *is the set of NACs derived from P's negative pre-conditions, with* $N_i^S \cong C_i|_S +_{C|_S} C$. *See the left of Fig. 7, where* P_S *is* L *in this definition.*
- $TNAC^n(P) = \{L \overset{m_k}{\to} T_k\}$ *is the set of NACs ensuring termination, where* T_k *is built by making* m_k *injective and jointly surjective with* $Q^n \overset{f}{\to} T_k$, *s.t. the diagram shown below commutes.*

$$\begin{array}{ccc} Q^n|_S & \to & Q^n \\ \downarrow & = & \downarrow f \\ L & \to & T_k \end{array}$$

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

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

Remarks. In the previous definition, we have used function *Conc(P)*, which given a pattern P, calculates the set of all valid node type substitutions $\{Q^n\}$ of its main constraint Q. Slightly abusing the notation, we have used $Conc(P)$ as an index set.

The set $NPAT^n(P)$ contains the negative post-conditions derived from the 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 $(R^n \setminus L) \cap C_k \neq \emptyset$ reduces the size of $NPAT^n(P)$, because we only need to consider possible violations of N-patterns due to created elements by the RHS, as we start with an empty target model, and the source already satisfies all N-patterns.

Example. The upper row of Fig. 9 shows the operational forward rule generated from pattern *P(Service-SessionBean)*, which does not contain abstract objects. There are three NACs for termination, `TNAC1`, `TNAC2` and `TNAC3`, the former equal to R. `TNAC3` is not shown in the figure, but is like `TNAC1` with an additional node of type `D2J` in the correspondence graph, connecting nodes `d` and `e`. Note that we do not do any algebraic manipulation of formulae to generate the rule, hence demonstrating the advantages of using constraint triple graphs in our approach. The figure also shows a direct derivation where both G and H are ground constraints. Constraint H is obtained by a pushout, and hence according to Prop. 1 is calculated by a pushout on triple graphs and the conjunction of the formulae of R and G. When the transformation ends, a constraint solver can be used to resolve attribute values. We will present further generated rules in Section 8.

According to [21], 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

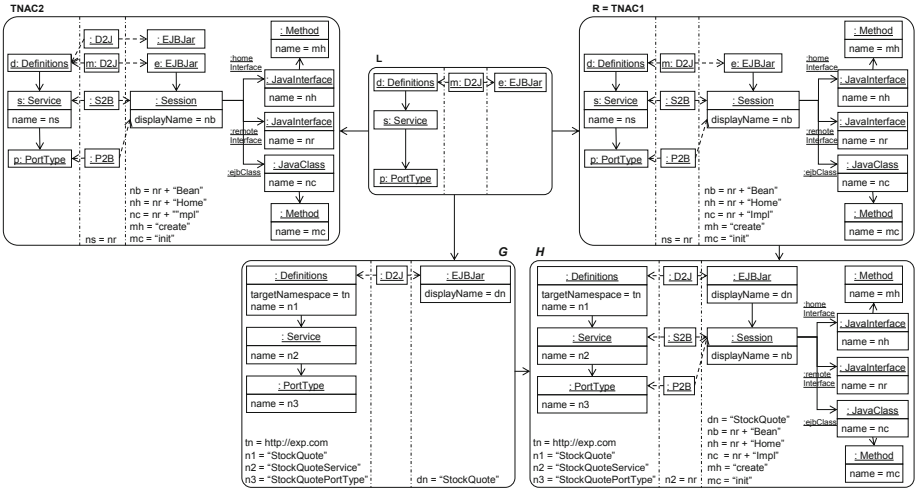


Fig. 9. Generated forward rule and derivation

produced. For example, assume we have a starting model with two `Definitions` objects with same name. Then, the synthesized forward rules are able to generate the model to the left of Fig. 10, but not the one to the right of the same figure, which also satisfies the specification. The model to the right would be generated if we could synthesize rules reusing elements created by previous applications of rules. Next subsection describes a method, called *parameterization*, that ensures completeness of the rules generated from a specification without N-patterns (and therefore it makes possible to find both solutions in the figure). The main idea is to generate additional patterns with increasingly bigger parameters, which enables the generated rules to reuse previously created elements.

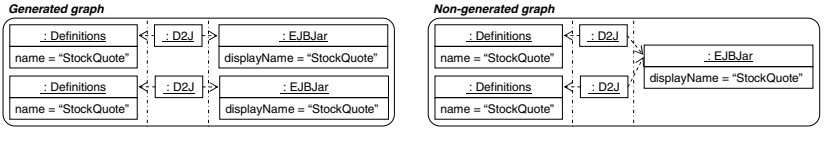


Fig. 10. Reachable (left) and unreachable (right) models for the specification without parameterization

Please note that the resulting constraint of a forward transformation forward-satisfies the specification, but does not necessarily backward-satisfies it. This is also noticed in QVT-R [23], where check-only transformations are directed as well (either forwards or backwards). Thus, the result of an *enforcing* forward transformation does not necessarily satisfy the same transformation when executed backwards in mode check-only, and vice versa.

If a specification contains arbitrary 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. In this case, some terminal graphs – to which no further rule can be applied – may not be models of the specification. Note however that if the specification admits solutions, our operational mechanisms are still able to find all of them, but in this case not all terminal models with respect to the grammar satisfy the specification.

6.1 Parameterization and Heuristics for Rule Derivation

In order to obtain completeness, we apply an operation called *parameterization* to every P-pattern in the specification. In this way, the resulting rules are able to generate all possible models of the specification [12,21]. The *parameterization* 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 7 (Parameterization). *Given* $T = \bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j)$*, its parameterization is* $Par(T) = \{\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C') \Rightarrow P(Q) \bigwedge_{j \in Post} N(C_j) | C \overset{i_1}{\hookrightarrow} C' \overset{i_2}{\hookrightarrow} Q, C \ncong C', C' \ncong Q\}$.

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. 11 shows some of the parameters generated by parameterization for a pattern like *P(Operation-Method)* in Fig. 6 but without parameters. Parameterization generates 123 patterns in total. The pattern with parameter $\overleftarrow{P}(1)$ is enforced when the port is already mapped to an EJB with a Java class, and in forward transformation avoids generating a rule that creates a bean and a Java class with arbitrary names. Parameter $\overleftarrow{P}(3)$ reuses an operation with the same name as the method, and in backward transformation allows generating just one operation from a number of methods with the same name but different number of parameters. However, $\overleftarrow{P}(2)$ is potentially harmful as it may lead to reusing a method that already belongs to a different bean, and thus to an incorrect model. Note however that this is not possible as an N-pattern generated from the maximum cardinality constraints of the meta-model forbids methods to belong to two different `JavaClass`es. This shows that including the cardinality constraints of the meta-models as N-patterns in the transformations allows controlling the level (and correctness) of reuse.

As the example shows, parameterization generates an exponential number of patterns with increasingly bigger parameters, resulting in an exponential number of rules. However one does not need to generate these rules beforehand, but they can be synthesized "on the fly". Moreover, some of these forward rules generated

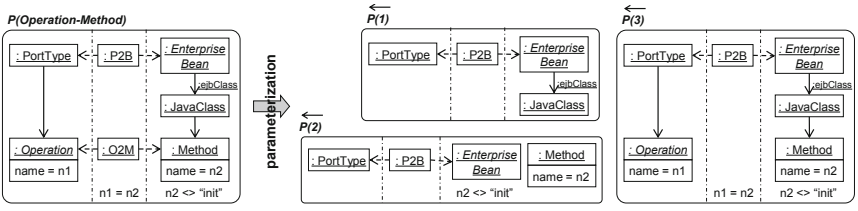


Fig. 11. Parameterization example

from the parameterized pattern will actually be equal, namely, those generated from parameters with same target and correspondence graph. Although parameterization ensures completeness, we hardly use it in practice due to the high number of generated rules, 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").

In order to reduce the number of rules, we propose two heuristics. 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 that create elements whose attributes can take any value. Instead, we prefer that these elements are generated by some other rule that assigns them a value, if it exists. Note that some transformations may not provide a unique value for each attribute thus being "loose".

Heuristic 1. *Given a pattern P, replace it by a new pattern that has as parameter all elements with some attribute not constrained by the formula in P but constrained by some other pattern, as well as the mappings and edges between these elements. We do not apply the heuristic if the obtained parameter is equal to Q.*

Example. In the pattern in Fig. 11, the heuristic generates just one pattern with parameter $\overleftarrow{P}(1)$. Thus, the generated forward rules do not create beans or classes with arbitrary names. Note that the heuristic replaces the original pattern with the generated one. This example shows that there is no need to set this parameter explicitly a priori as we did in the initial specification of Fig. 6.

In Fig. 12 we present to the left an extended version of the pattern *P(Definitions-EJBJar)* which maps WSDL definitions to a jar but also to a package. In the backward direction, a definitions object will be created for each package and its container jar (this is known because the package contains an interface that belongs to a bean inside the jar), and we use the name of the package to give value to the `targetNamespace` attribute. However, in the forward direction we want to avoid the creation of beans with undefined name, therefore we apply the presented heuristic and obtain the pattern to the right, where the positive pre-condition is annotated with the key $\langle\langle param \rangle\rangle$ and highlighted. In addition, we need to ensure that there are no two `Definitions` with same `name` and `targetNamespace`, otherwise the operational mechanisms would

create different `Definition` objects for each `JavaInterface` inside a package. This can be done using one N-pattern, or as we will see later, using the CBE semantics.

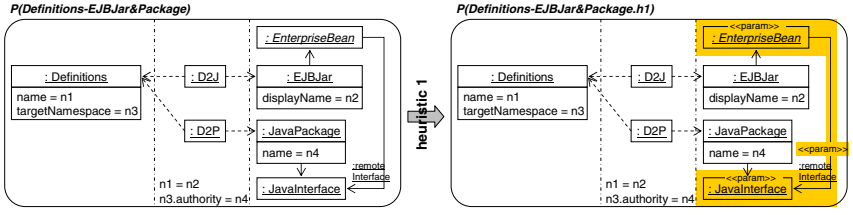


Fig. 12. Applying heuristic 1 to a pattern

The next heuristic generates only those parameters that avoid duplicating a graph S_1, forbidden by some N-pattern of the form $N(S_1 +_U S_1)$. This ensures the generation of rules producing valid models for the class of specifications with N-patterns of this form (called FIP in [4]), and which include the N-patterns generated by the maximum cardinality constraints in meta-models. The way to proceed is to apply heuristic 2 to each P- and N-pattern of the form $N(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated.

Heuristic 2. *Given a P-pattern* $[\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q)] \in SP$*, if there is an N-pattern* $[N(S)] \in SP$ *with* $S \cong S_1 +_U S_1$*, and* $\exists s\colon S_1 \to Q, u\colon U \to C$ *s.t.* $s \circ u_1 = q \circ u$ *(see left of Fig. 13), and* $\nexists s'\colon S_1 \to C$ *all injective s.t.* $q \circ s' = s$*, then we generate additional patterns with parameters all* C'_j *s.t.* q_1 *and* q_s *in Fig. 13 are jointly surjective, and the induced* $C'_j \to Q$ *is injective.*

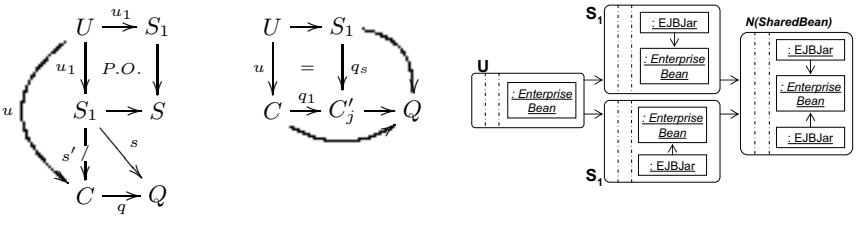


Fig. 13. Condition for heuristic 2 and generated parameters (left and center). Decomposition of N-pattern (right).

The rationale of this heuristic is that if a P-pattern has a parameter C that contains U but not S_1, and its main constraint Q contains S_1, then applying the pattern creates a new structure S_1 glued to an existing occurrence of U. This heuristic enlarges the parameter to include S_1 and thus avoid its publication. The way to proceed is to apply the heuristic for each P- and N-pattern of the

form $N(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated.

Example. The right of Fig. 13 shows that N-pattern *N(SharedBean)* satisfies the conditions demanded by heuristic 2. The pattern forbids an `EnterpriseBean` to belong to two `EJBJar`s. Fig. 14 shows the application of heuristic 2 to the P-pattern *P(Definitions-EJBJar&Package.h1)* previously obtained by heuristic 1, and to the N-pattern *N(SharedBean)* decomposed in Fig. 13. The generated parameter C'_1 includes the `EJBJar` so that it is not created in forward transformation. In this way, it avoids the creation of the model fragment forbidden by the N-pattern. Note that the initial pattern with parameter C is also kept in the specification.

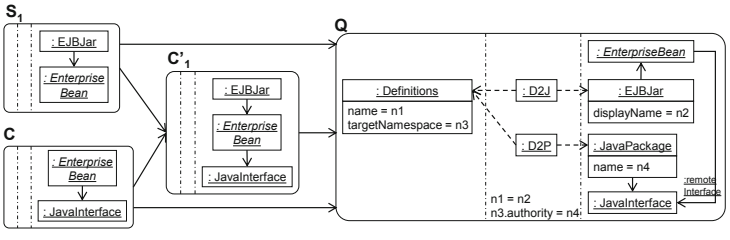


Fig. 14. Applying heuristic 2

7 Check-Before-Enforce Semantics

Even though the presented heuristics help controlling the level of reuse, in an M2M transformation it is useful to control whether an element has to be created in the generated domain or whether it already exists and can be reused. This avoids creating duplicated objects. This control mechanism has been incorporated to approaches like QVT and is called Check-Before-Enforce (CBE) semantics. In this section we incorporate it to our framework.

The idea is to generate N-patterns forbidding two objects of the same type with the same attribute values. Then, our *Heuristic 2* takes each P-pattern in the specification and generates new ones with appropriate parameters reusing the objects whenever possible.

Example. The left of Fig. 15 shows the N-pattern that the CBE semantics generates for class `Definitions`, which forbids two `Definitions` objects with same attribute values in the WSDL model. The center of the same figure presents the pattern generated by heuristic 2 from pattern *P(Definitions-EJBJar&Package.h1)* shown in Fig. 12 due to the newly introduced N-pattern. The new pattern adds a `Definitions` object to the previous parameter.

To allow for a better control of reuse, and to permit the specification of when two objects are to be considered equal, QVT includes the concept of *Key*. Keys allow us, for example, to neglect certain attributes when comparing if two

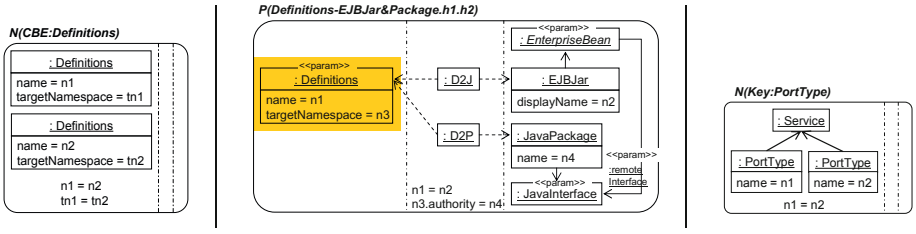


Fig. 15. N-pattern generated by CBE semantics (left). Pattern generated by heuristic 2 (center). N-pattern generated from the key of `PortType`.

objects are the same, or include further connected objects in the comparison. Again, such a concept can be easily incorporated into our framework by an appropriate generation of N-patterns. For instance, we can set that the key for `PortType`s is their name and owner service. This would be specified in QVT as `Key PortType{name, Service}`, from which our procedure generates the N-pattern to the right of Fig. 15.

8 Specification Process and Back to the Case Study

Fig. 16 summarises the steps needed to engineer a pattern-based transformation specification and obtain the operational mechanisms. The process is shown as a SPEM model [26], similar to an activity diagram, where activities are numbered in dots and represented as arrow-like icons. The model distinguishes the level at which the activity is performed (language, specification or operational) and who performs it (language engineer, transformation engineer or automated process).

First, the language engineer designs the source and target meta-models or reuse them if already available (*step 1*). Next, the transformation engineer designs the allowed traces between the elements in the source and target languages, obtaining a meta-model triple as a result (*step 2*). Once this is available, several activities can start in parallel. On the one hand, the engineer builds the transformation specification (*step 3a*) and sets the keys (*step 3b1*). On the other hand, our automatic mechanisms generate the N-patterns derived from the meta-model constraints (*step 3c*), as well as those for the CBE semantics and keys (*step 3b2*). Then, we apply the heuristics to the transformation specification and N-patterns synthesized by the previous activities (*step 4*). This results in an enriched specification that is used to generate the TGG operational rules, once the transformation direction is chosen (*step 5*).

If we apply this engineering process to our case study, the first step is to build the WSDL and EJB meta-models, which were shown in Figs. 1 and 2. The trace meta-model defines four types of nodes: (i) `D2J` connecting `Definitions` and `EJBJar` objects, (ii) `S2B` connecting `Service` and `Session` objects, (iii) `P2B` connecting `PortType` and `Session` objects, and (iv) `O2M` connecting `Operation` and `Method` objects.

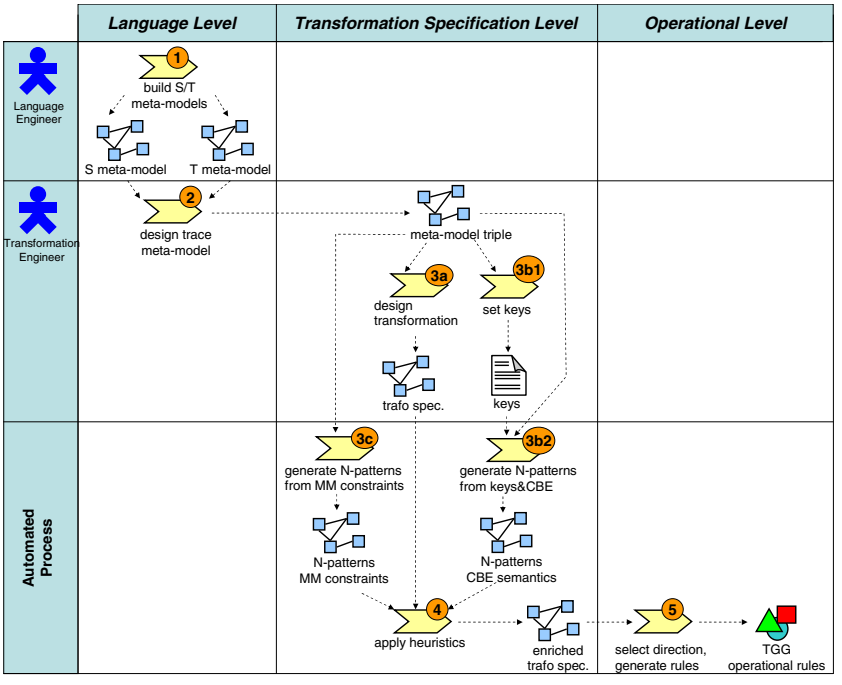


Fig. 16. Our transformation engineering process

Next, we have to design the transformation specification. This is made of the three P-patterns shown in Fig. 6: *P(Definitions-EJBJar)*, *P(Service-SessionBean)* and *P(Operation-Method)*, all of them without parameters. For simplicity we do not consider Java packages, and thus omit pattern *P(Definitions-EJBJar&Package)* shown in Fig. 12.

Meanwhile, *step 3c* generates one N-pattern for each association end in the meta-model with bounded upper cardinality. Three of these N-patterns were shown in Fig. 6. In its turn, *step 3b2* generates additional N-patterns due to the CBE semantics and according to the specified keys. This results in one N-pattern for each class in the meta-models. In case we chose the direction of the transformation first, it would be enough to generate N-patterns from one of the meta-models: the target in forward transformations and the source in backwards. Fig. 15 showed some of the N-patterns generated due to CBE semantics. Then, applying the heuristic 1 replaces some P-patterns by others with parameters, and applying the heuristic 2 adds new patterns to the specification.

Finally, we choose the operational scenario to be solved and generate the TGG operational rules. Two of the generated forward rules are shown in Fig. 17. The rule to the left is generated from the *P(Definitions-EJBJar)* pattern and creates an `EJBJar` object for each `Definitions` object in the WSDL model. The rule has one termination NAC equal to the RHS, and two post-conditions coming from the N-pattern *N(SharedBean)* that was derived by a meta-model cardinality constraint.

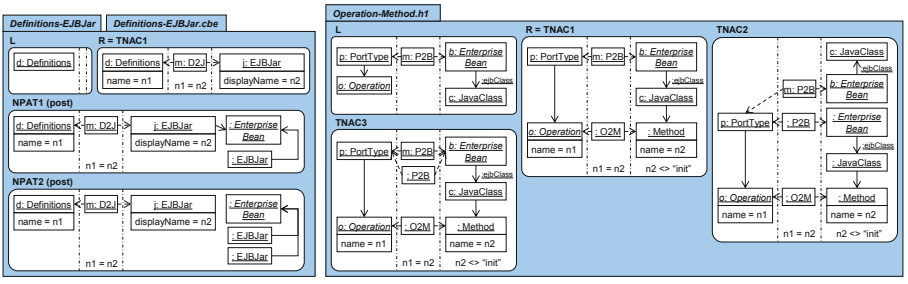


Fig. 17. Some of the generated forward rules for the case study

The right of the same figure shows the rule generated from pattern *P(Operation-Method.h1)*, pattern that replaced pattern *P(Operation-Method)* after applying the heuristic 1. The rule creates one `Method` for each `Operation` in a `PortType`. Note that objects `o` and `b` have abstract type. The rule has three termination NACs, as well as several negative post-conditions that are omitted for simplicity.

The generated forward rules can be applied to WSDL models in order to obtain the EJB model. Fig 18 shows one example where we start with a WSDL containing one `Service` owning a `PortType` with two `Operations`. After applying the four rules shown in the figure, we obtain the constraint triple graph to the right, to which no more rules can be applied. Note that in this final model not all attributes are constrained, for example the `transactionType` and the `sessionType` of the `Session` object (these attributes are not considered by the transformation specification). Hence, a constraint solver could give arbitrary values to these attributes, or the user could be asked to give one. Also, for this starting model the transformation is confluent (in the sense that we obtain a

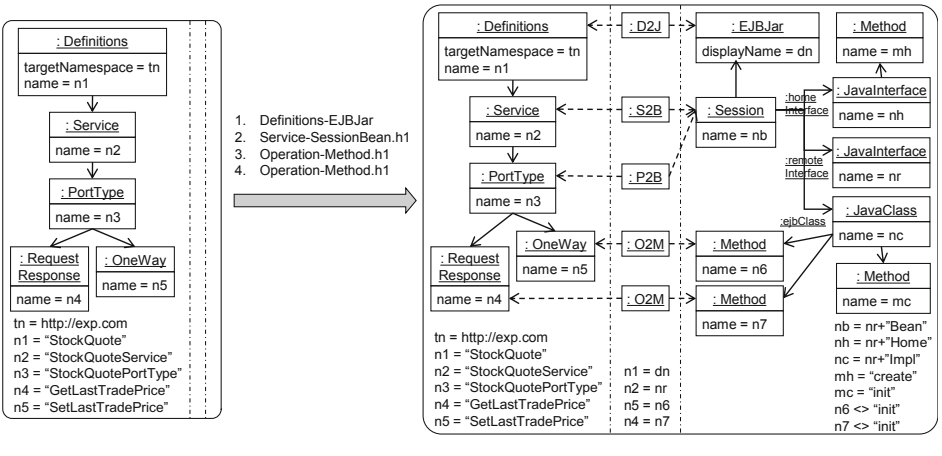


Fig. 18. Example model transformation

unique constraint triple graph, from which we can however derive several models by assigning different attribute values), but this is not necessarily so for other models, as already noted in [12,21]. This is actually a good behaviour, as one obtains all terminal models that satisfy the transformation specification.

9 Conclusions and Future Work

In this paper we have incorporated the CBE semantics and keys concepts of QVT-R into our pattern-based M2M framework in order to control object reuse in M2M transformations. This is achieved by adding N-patterns to the specification so that they forbid the existence of two objects that are considered equal, thus making the operational mechanism to reuse such objects whenever possible. We have also shown that the meta-model inheritance hierarchy and the integrity constraints have to be considered by the transformation specification. In particular, we have discussed how to generate N-patterns from the maximum cardinality constraints in associations, as well as how to handle abstract objects in patterns. Finally, we have illustrated these concepts with a transformation between WSDL and EJB models.

There are many open lines for further research. For example, one could consider the benefits of adding relations between the nodes in the correspondence graph instead of having a discrete graph there. These have been exploited in [9] for implementing incremental transformations, but note that our theory demands graph morphisms between the correspondence and the other two graphs, hence posing some restrictions. We are also starting to investigate more complex operational scenarios, like incremental transformations and model synchronization. On the theoretical side, it is worth investigating analysis methods for specifications, as well as simplifications of the current formalism. For example, in our experience, it seems possible to get rid of parameters in the initial specification, and express the restrictions with N-patterns so that one ends up with equivalent specifications. However, this is still an open question. We would also like to explore higher-level means of specifications, by (i) omitting the correspondence graph at the specification level (and automatically generating the traces at the operational level, as in [10,14]), and (ii) making possible the specification of pattern dependencies and parameter passing, similar to when or where clauses in QVT. These two steps would allow us to express the semantics of QVT-R with our framework. We also plan to perform a detailed study of the expressivity of different mechanisms for reuse of other bidirectional languages, like TGGs and QVT-R, by using realistic examples. Finally, we are also investigating other languages for the operational mechanisms, like Coloured Petri Nets, in the style of [5].

Acknowledgements. Work partially supported by the Spanish Ministry of Science and Innovation, with projects METEORIC (TIN2008-02081) and FORMALISM (TIN2007-66523), and the R&D program of the Community of Madrid (S2009/TIC-1650, project "e-Madrid"). Moreover, part of this work was done during a post-doctoral stay of the first author at the University of York, and sabbatical leaves of the second and third authors to the University of York and TU

Berlin respectively, all with financial support from the Spanish Ministry of Science and Innovation (grant refs. JC2009-00015, PR2009-0019 and PR2008-0185).

References

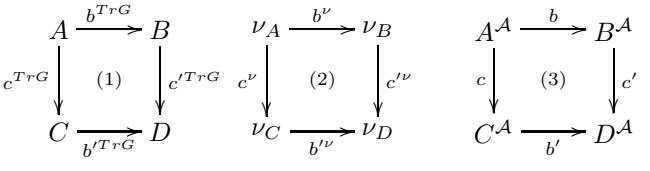
1. Akehurst, D.H., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. ENTCS 72(3) (2003)
3. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. TCS 376(3), 139–163 (2007)
4. de Lara, J., Guerra, E.: Pattern-based model-to-model transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 426–441. Springer, Heidelberg (2008)
5. de Lara, J., Guerra, E.: Formal support for QVT-Relations with coloured Petri nets. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 256–270. Springer, Heidelberg (2009)
6. Dehayni, M., Féraud, L.: An approach of model transformation based on attribute grammars. In: Konstantas, D., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 412–423. Springer, Heidelberg (2003)
7. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer, Heidelberg (2006)
9. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1), 21–43 (2009)
10. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: Implementing Query/View/Transformation with Triple Graph Grammars. Software and Systems Modeling 9(1), 21–46 (2010)
11. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. Software and Systems Modeling, special section on ICGT 2004 6(3), 317–347 (2007)
12. Guerra, E., de Lara, J., Orejas, F.: Pattern-based model-to-model transformation: Handling attribute conditions. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 83–99. Springer, Heidelberg (2009)
13. IBM WebSphere, `http://www-01.ibm.com/software/websphere/`
14. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report TR-RI-07-284, Paderborn University (2007)
15. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended triple graph grammars with compatible graph translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
16. Königs, A., Schürr, A.: Tool integration with triple graph grammars - a survey. ENTCS 148(1), 113–150 (2006)
17. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Addison-Wesley Object Technology Series (2004)
18. MTF. Model Transformation Framework, `http://www.alphaworks.ibm.com/tech/mtf`

19. OMG: Metamodel and UML profile for Java and EJB specification (2004), http://www.omg.org/cgi-bin/doc?formal/04-02-02.pdf
20. Oracle containers for J2EE, http://www.oracle.com/technology/tech/java/oc4j
21. Orejas, F., Guerra, E., de Lara, J., Ehrig, H.: Correctness, completeness and termination of pattern-based model-to-model transformation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 383–397. Springer, Heidelberg (2009)
22. Orejas, F., Wirsing, M.: On the specification and verification of model transformations. In: Palsberg, J. (ed.) Semantics and Algebraic Specification. LNCS, vol. 5700, pp. 140–161. Springer, Heidelberg (2009)
23. QVT (2008), http://www.omg.org/spec/QVT/1.0/PDF/
24. Roman, E., Sriganesh, R.P., Brose, G.: Mastering Enterprise JavaBeans, 3rd edn. Wiley, Chichester (2004)
25. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
26. SPEM (2008), http://www.omg.org/cgi-bin/doc?formal/08-04-01.pdf
27. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
28. Völter, M., Stahl, T.: Model-driven software development. Wiley, Chichester (2006)
29. W3C: WSDL v1.1. specification (2001), http://www.w3.org/TR/wsdl
30. W3C: SOAP v1.2. specification (2007), http://www.w3.org/TR/soap
31. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. ENTCS 211, 159–170 (2008)

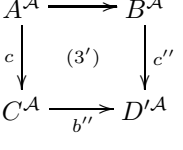
Appendix

Proof of Proposition 1.

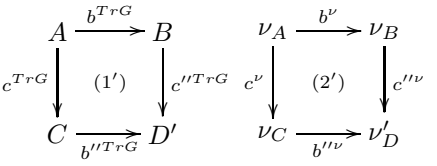
Proof. We have to prove that if diagrams (1) and (2) are pushouts then diagram (3) is also a pushout, where $D^{\mathcal{A}} = (D, \nu_D, c'(\alpha_B) \wedge b'(\alpha_C))$.



First, it may be noted that diagram (3) is indeed a diagram in $\mathbf{CTrG}_{\mathcal{A}}$, since $(c'(\alpha_B) \wedge b'(\alpha_C)) \Rightarrow c'(\alpha_B)$ and $(c'(\alpha_B) \wedge b'(\alpha_C)) \Rightarrow b'(\alpha_C)$ are tautologies, which means that b' and c' are indeed morphisms in $\mathbf{CTrG}_{\mathcal{A}}$. Moreover, we know that if diagram (3') commutes:



then also diagrams (1') and (2') commute:



which means that there are unique morphisms $e^{TrG} : D \to D'$ and $e^{\nu} : \nu_D \to \nu_{D'}$ satisfying $e^{TrG} \circ b'^{TrG} = b''^{TrG}$, $e^{TrG} \circ c'^{TrG} = c''^{TrG}$, $e^{\nu} \circ b'^{\nu} = b''^{\nu}$, and $e^{\nu} \circ c'^{\nu} = c''^{\nu}$. But this means that $e : (D, \nu_D, c'(\alpha_B) \wedge b'(\alpha_C)) \to (D', \nu_{D'}, \alpha_{D'})$ is a morphism, since if $\mathcal{A} \models \alpha_{D'} \Rightarrow c''(\alpha_B)$ and $\mathcal{A} \models \alpha_{D'} \Rightarrow b''(\alpha_C)$ then $\mathcal{A} \models (\alpha_{D'} \Rightarrow (c''(\alpha_D) \wedge b''(\alpha_C)))$. But we know that $c''(\alpha_B) \wedge b''(\alpha_C) = (e \circ c')(\alpha_B) \wedge (e \circ b')(\alpha_C) = e(c'(\alpha_B) \wedge b'(\alpha_C)))$ and this means that $\mathcal{A} \models (\alpha_{D'}) \Rightarrow e(c'(\alpha_B) \wedge b'(\alpha_C))$. Finally, if $e' : (D, \nu_D, c'(\alpha_B) \wedge b'(\alpha_C)) \to (D', \nu_{D'}, \alpha_{D'})$ is a morphism satisfying that $e' \circ b' = b''$ and $e' \circ c' = c''$ then, by the uniqueness of e^{TrG} and e^{ν}, we have that $e = e'$.