Fundamenta Informaticae XXI (2010) 1–33
IOS Press

An Algebraic Semantics for QVT-Relations Check-only Transformations

Esther Guerra, Juan de Lara

Computer Science Department. Universidad Autónoma de Madrid (Spain)

{*Esther.Guerra, Juan.deLara*}*@uam.es*

Abstract. QVT is the standard for model transformation defined by the OMG in the context of the Model-Driven Architecture. It is made of several transformation languages. Among them, QVT-Relations is the one with the highest level of abstraction, as it permits developing bidirectional transformations in a declarative, relational style. Unfortunately, the standard only provides a semiformal description of its semantics, which hinders analysis and has given rise to ambiguities in existing tool implementations.

In order to improve this situation, we propose a formal, algebraic semantics for QVT-Relations check-only transformations, defining a notion of satisfaction of QVT-Relations specifications by models.

Keywords: Model-Driven Engineering, Model Transformation, QVT-Relations, Category Theory.

1. Introduction

Model-Driven Engineering (MDE) is a software engineering paradigm where models are no longer passive documentation, but they are the principal assets of the development process and play an active role. Among other activities, models are used to specify, simulate, test, generate code and maintain the final application. In this way, as models are more abstract than code, the development process is shortened, the productivity is increased, and higher levels of quality and standardization are easier to achieve [28].

A central activity in MDE is the manipulation of models. This is not performed using general-purpose languages like Java, but using domain-specific languages which are more adequate for the task of model transformation. QVT [22] (for Query/View/Transformation) is a standard promoted by the OMG comprising languages to specify model-to-model transformations. These are a special kind of model manipulations that in the simplest case take one input model of a given language (e.g. Statecharts), and produce one output model of a different language (e.g. Petri nets). Model-to-model transformations are pervasive in MDE, as models frequently have to be refined or abstracted during the development process, and may be transformed into different formal languages for analysis.

The QVT standard comprises three languages: QVT-Relations (QVT-R), QVT-Core (QVT-C) and QVT-Operational (QVT-O). QVT-R is the one with highest level of abstraction. It has a declarative, relational, bidirectional style. This means that we can use a single QVT-R specification operationally to transform from a source to a target language (called forward transformation) and from target to source (called backward transformation) either in batch (the source or target models are created from scratch) or update modes. We can also use it in check-only mode to verify the conformance of two models with respect to the transformation specification. QVT-C and QVT-O are both operational languages used to implement forward and backward transformations. While QVT-C is a trace-based language similar to triple graph grammars [24], QVT-O is lower level and includes imperative constructions.

Even though QVT-R is not having the same impact as other OMG standards like the UML [17], there are already several tools giving support for its execution [15, 16]. However, one drawback is its lack of formal semantics. The standard describes a compilation into QVT-C for solving the transformation scenario, but it only gives an informal procedure for solving the check-only scenario. However, the check-only scenario is more fundamental than the transformation one, as it permits checking the correctness of the result of a transformation. Hence, a detailed, formal semantics for the check-only case is urgently needed by tool builders, users and researchers in order to be able to evaluate whether two given models are synchronized, if one is a correct translation of the other, or to identify parts of the specification that the models do not satisfy.

This work is a contribution towards solving this gap. In particular, we present a formal semantics for the QVT-R check-only scenario based on algebraic specification and category theory. Our semantics provides a formalization of QVT relations and their dependencies, and defines a notion of forward and backward satisfaction of specifications by models. Our proposal also generalizes some details of the standard. First, relations are formalized as bidirectional constraints based on equations instead of on assignments. Second, our mechanism for parameter passing is more general and flexible than the one described in the standard as, for instance, it permits passing different sets of parameters to a given relation.

The first formal semantics of QVT-R was given in Stevens' seminal paper using game theory [27], but to the best of our knowledge, ours is the first formalization attempt that captures essential elements like the structure of a relation (when, where clauses, declaration of formal parameters) and the binding. Moreover, we also provide a validation of the proposed semantics with respect to existing tools [16] and the formalization given in [27].

The rest of the paper is organized as follows. Section 2 gives an overview of QVT-R, especially the check-only scenario. Section 3 introduces the basic building blocks of our algebraic formalization, which is presented next in Section 4. Section 5 compares our semantics with that implemented by tools and other proposals in the literature. Section 6 reviews related research and, finally, Section 7 concludes.

2. Introduction to QVT-Relations

QVT-R is the highest-level of abstraction language of the OMG standard for Query/View/Transformation [22]. It has a declarative nature and a dual graphical/textual syntax. A QVT-R specification is made of relations with two or more domains (usually two). Domains are described by patterns similar to UML object diagrams, and with a flag indicating whether they are *checkonly* or *enforced*. Models of enforced domains may be modified in order to satisfy the relations, whereas models of checkonly domains are just inspected to check for disagreements, but cannot be modified.

Transformations are given a direction when they are applied. If the direction is a domain tagged as enforced, the models of this domain may be modified to obtain a model that, together with a given model from the other domain, satisfies the transformation specification. If a transformation is applied in the direction of a checkonly domain, then the execution engine must report the locations where the model does not conform to the transformation, but cannot modify the model.

Relations may contain *when* and *where* clauses. The former expresses conditions under which a relation needs to hold. They usually refer to other relations, to which they may pass objects appearing in the current relation. *Where* clauses may call other relations, and are similar to function calls in traditional programming. In addition, relations can be *top* or *non-top* level. After executing a transformation all top-level relations should hold, whereas the non-top level ones only need to hold when invoked from the *where* section of other relations.

Throughout the paper we will use an example transformation from a subset of UML class diagrams into relational database schemas. In order to ease understanding, the example is a simplification of the one given in the QVT standard [22]. The meta-models for both languages are shown in Figure 1. The UML meta-model to the left declares `Packages` made of `Classes` with typed attributes. `Packages` and `Classes` inherit a `persistent` attribute, used to mark whether they are persistent and therefore they should be reflected in the RDBMS schema. The RDBMS meta-model to the right defines `Schemas` with `Tables`, and these contain `Columns` of a given type.

Figure 2 shows the QVT-R specification relating both languages. It defines three relations, two of them being top-level. When interpreted in the $uml \to rdbms$ direction, relation *PackageSchema* demands that for each persistent package in a UML model, there is a schema with the same name as the package, but prefixed by “S_”. Relation *ClassTable* states that for each persistent class there must be

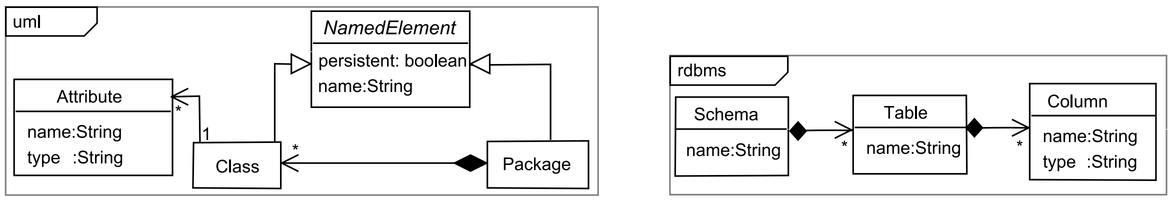


Figure 1. The meta-model of the two languages used in the example transformation.

a table with same name, but prefixed by "T_". Nonetheless, the *when* section demands this relation to hold only if relation *PackageSchema* holds for the package and schema containing the class and table. In addition, the *where* clause asks the *AttributeColumn* relation to hold for the class and the table. Finally, relation *AttributeColumn* requires that for each attribute of a class, there is a column with the same name and suitable type. The type of the column is calculated by the OCL expression in the *where* section.

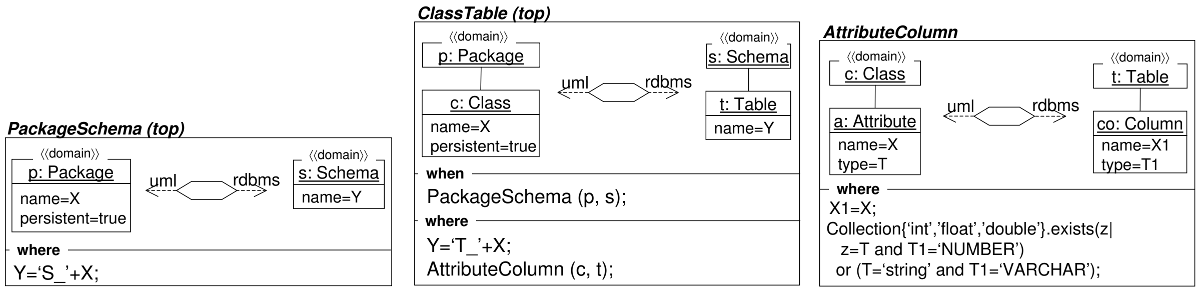


Figure 2. An example QVT-R transformation.

Relations indicate their signature by means of the $\langle\langle$*domain*$\rangle\rangle$ keyword. In this way, the elements tagged as $\langle\langle$*domain*$\rangle\rangle$ should be passed as parameters when the relation is invoked in the *when* or *where* clauses of any relation (including itself). For instance, p and s are marked as *domain* in relation *PackageSchema*, and whenever the relation is invoked from the *when* section of relation *ClassTable*, a package and a schema are passed as actual parameters.

In checkonly mode, if we provide a UML and a RDBMS models and execute the transformation in the $uml \rightarrow rdbms$ direction, we obtain whether the RDBMS model is consistent with the UML model interpreting all relations in this direction, and the models are not modified. Similarly, we can execute the transformation in the $rdbms \rightarrow uml$ direction to inspect whether the UML model is consistent with the RDBMS model, interpreting in this case all relations in the $rdbms \rightarrow uml$ direction. If both models are consistent with each other in each direction, we say that the models are synchronized with respect to the specification.

The satisfaction checking procedure demands the satisfaction of all top-level relations in a specification. Roughly, a relation is satisfied by two models in a given direction if, for each occurrence of its source pattern in the source model, we can find an occurrence of its target pattern in the target model being a correct binding for the relation. To simplify the terminology, we speak of *forward* transformation when the source domain is the one depicted to the left in the relation ($uml \rightarrow rdbms$ in our example), whereas we speak of *backward* transformation if it is the one shown to the right ($rdbms \rightarrow uml$). Hence, we define the concept of *forward pre-condition* as the source objects needed to enable a relation, and the backward pre-condition is defined analogously. As we will show later, building this pre-condition given a relation is slightly more complex. In particular, it has to include the source objects, the objects used by the relations appearing in its *when* clause, as well as those appearing in invocations to the actual relation from the *where* clause of a caller relation.

As an example, the left of Figure 3 shows two models where we want to check forward satisfaction, i.e. check whether for each occurrence of the forward pre-conditions, there are elements in the target domain that make such occurrence satisfy the relation. Relation *PackageSchema* is enabled in object `p`, and is actually satisfied because there is a schema (object `s`) that is a correct binding for the relation. The models also satisfy relation *ClassTable* forwards. This relation is enabled at objects {`c, p, s`}, and there is a table that completes the binding for the relation. Please note that `p` and `s` belong to the forward pre-condition of *ClassTable* because they are passed to *PackageSchema* in the *when* clause. Moreover, *ClassTable* has a *where* dependency with relation *AttributeColumn*, which should be satisfied as well. The forward pre-condition for *AttributeColumn* is made of objects {`c, a, t`} (i.e. the objects in the UML domain, and those received by the invocation from *ClassTable*). The models contain one occurrence of these objects that satisfies the relation because object `col` is a correct binding. Hence, altogether, we conclude that the RDBMS model is consistent with the UML one, when interpreting the transformation forward.

On the contrary, the UML model is not consistent with the RDBMS one when interpreting the transformation backwards, for two reasons. First, relation *PackageSchema* is backward-enabled twice, at schemas `s` and `s1`. However, it is not satisfied at schema `s1` because there is no suitable package for it. Second, relation *ClassTable* is not satisfied because its *where* dependency is not satisfied. This is so because there is no suitable translation for column `co`. The right of Figure 3 shows two models that satisfy the specification both forwards and backwards, and therefore these models are synchronized.

Actually, two synchronized models do not need to have an isomorphic structure. This depends on the transformation specification, and also on the fact that by default, elements in the source and target models can be used several times to satisfy the relations. As an example, Figure 4 depicts two synchronized models that are not structurally isomorphic. This is possible as the same table can be used to forward-satisfy relation *ClassTable* for classes `c` and `c1` in the source model. Moreover, the models also satisfy the transformation backwards, as the table has two possible binding classes in the source model.

The standard does not provide a formal means to check the conformance of models with respect to QVT-R specifications, but just an informal procedure. In the following, we develop such a formalization.

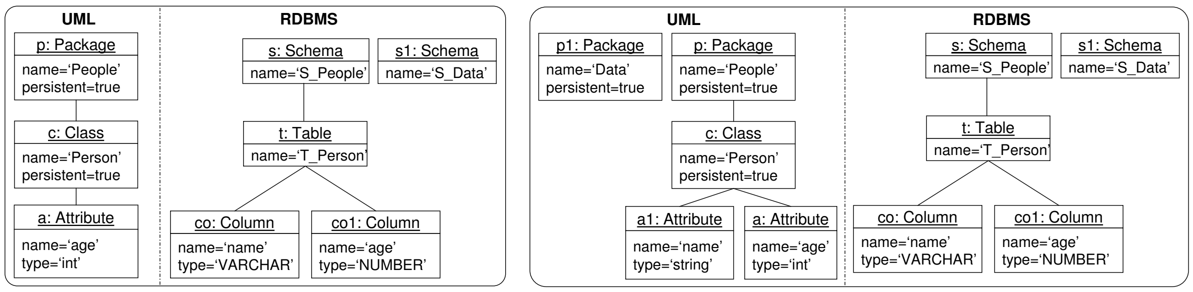


Figure 3. Example of forward satisfaction (left). Example of synchronized models (right).

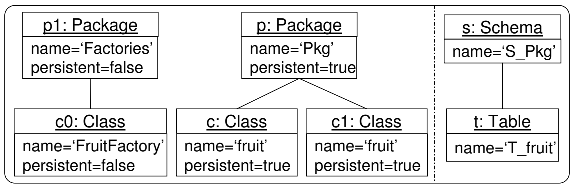


Figure 4. Two synchronized models that are not structurally isomorphic.

3. Formalising Model Pairs: Symbolic Tuples

Before presenting our formalization of QVT-R, we start by defining the notion of *symbolic tuple*, a structure that we will use to represent pairs of models like those of Figures 3 and 4, as well as the pair of domains inside a QVT relation, like those in Figure 2. The definition of each model in the structure is based on E-graphs [8, 9], a kind of edge and node labelled graph. E-graphs have a component set V_D that stores all possible values (or labels) for the attributes. An attribute is represented as an edge from a node or an edge to its value in V_D. While the attributes for nodes are stored in the set E_{NA}, the attributes for edges are stored in the set E_{EA}.

Definition 1. (E-graph (from [9]))
An E-graph is a tuple $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_i, target_i)_{i\in\{G,NA,EA\}})$, where V_G is a set of graph nodes, V_D is a set of data nodes, E_G is a set of graph edges, E_{NA} is a set of "node attribution" edges, E_{EA} is a set of "edge attribution" edges, $source_G \colon E_G \to V_G$ and $target_G \colon E_G \to V_G$ are the source and target functions for the graph edges, $source_{NA} \colon E_{NA} \to V_G$ and $target_{NA} \colon E_{NA} \to V_D$ are the source and target functions for the "node attribution" edges, and $source_{EA} \colon E_{EA} \to E_G$ and $target_{EA} \colon E_{EA} \to V_D$ are the source and target functions for the "edge attribution" edges.

An E-graph morphism h is made of five set morphisms (h_{V_G}, h_{V_D}, h_{E_G}, $h_{E_{NA}}$, $h_{E_{EA}}$) such that the structure given by the source and target functions is preserved (see [8, 9] for details).

In MDE, models are typed by meta-models defining the node and edge types that can appear in the models. Similarly, we can add typing to E-graphs by working with tuples $(G, type \colon G \to TG)$, where

TG is the type graph or meta-model G is conformant to [8, 9] (that is, we use objects in the slice category of E-graphs over TG). In the following formalizations, we usually omit the typing for simplicity. Please note that meta-models may actually contain inheritance relations, which can be handled through clan-morphisms [5, 8], but for clarity, we omit a treatment of inheritance in this paper.

E-graphs store all admissible attribute values in a possibly infinite set V_D. Following the idea of *symbolic graphs* [19], we replace such possibly infinite set by a finite set of variables together with a formula α constraining their value.

Definition 2. (Symbolic Graph and Morphism (adapted from [19]))
A symbolic graph over the data $\Sigma-$algebra A is a tuple $C = \langle G, \alpha \rangle$, where G is an E-graph whose set of data nodes V_D is a finite set of variables, and α is a first-order Σ-formula over the variables in V_D and over the values in A.

Given symbolic graphs $\langle G_1, \alpha_1 \rangle$ and $\langle G_2, \alpha_2 \rangle$ over the same data algebra A, a symbolic graph morphism $h\colon \langle G_1, \alpha_1 \rangle \rightarrow \langle G_2, \alpha_2 \rangle$ is an E-graph morphism $h\colon G_1 \rightarrow G_2$ such that $A \models (\alpha_2 \Rightarrow h^{\#}(\alpha_1))$, where $h^{\#}(\alpha_1)$ is the formula obtained by replacing in α_1 every variable x_i in the set of labels V_D of G_1 by $h_{V_D}(x_i)$.

Example. Figure 5 shows a symbolic graph morphism between two symbolic graphs. The graphs are depicted using the standard notation of UML object diagrams to show the typing, whereas the formulas are depicted below. The symbolic graph to the left can be interpreted as a constraint or pattern that we want to search in the symbolic graph to the right. It represents a model with a persistent package and a persistent class, equally named. The morphism a identifies objects p and c in the two graphs and, as required by the definition of symbolic graph morphism, the formula to the right implies the formula to the left. On the contrary, the morphism cannot identify objects c in G_1 and c1 in G_2 because in that case there is no implication for the formulas, as the value of variables M and Z is different.

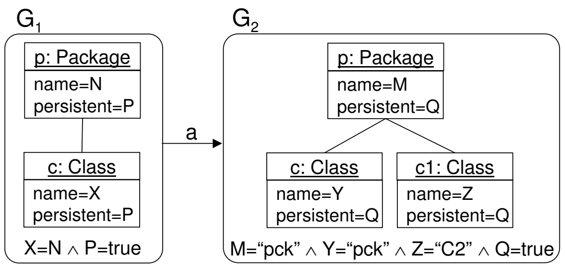


Figure 5. Example of symbolic graph morphism.

As we have seen, a symbolic graph can be used to represent both constraints (left graph in Figure 5) and models (right graph of Figure 5). We call a symbolic graph *ground* if its formula restricts the variables used for attribution to take exactly one value, as it is the case of G_2 in Figure 5. In the following, we usually depict ground symbolic graphs by placing the unique attribute values in the compartment of the attributes (just like in UML object diagrams) instead of showing the formula. Moreover, given a symbolic graph C, we sometimes use $D(C)$ to refer to its set of variables.

In our context, we are interested in manipulating pairs of models which may represent the two domain patterns in a QVT relation, or two models for which we want to check conformance with respect to a QVT-R specification. Hence, we define symbolic tuples made of two symbolic graphs and an additional formula expressing constraints that involve variables of the two graphs.

Definition 3. (Symbolic Tuple)
A symbolic tuple over the data $\Sigma-$algebra A is a tuple $P = \langle C_S, C_T, \beta \rangle$, where C_S and C_T are symbolic graphs over the data $\Sigma-$algebra A, and β is a first-order Σ-formula (sometimes called "relating formula") over the variables in $D(C_S) \uplus D(C_T)$ and over the values in A ($\uplus$ denotes disjoint union).

Given P, we use the notation $\gamma = \beta \wedge \alpha_S \wedge \alpha_T$ for the formula β in the context of the formulas α_S and α_T of C_S and C_T.

Example. Figure 6 shows two symbolic tuples (actually, it also shows a morphism between them, which we will explain later). The left tuple is not ground. It represents a pattern made of a package and a schema with same name (except for the prefix "S_"). We take the convention of placing the formula β that relates variables of the two graphs in the middle of the symbolic tuple, whereas the formulas α_S and α_T in C_S and C_T are placed in their corresponding graph compartment. If an attribute is assigned a variable which does not appear in the formulae of the symbolic tuple, we usually omit the attribute for simplicity. The symbolic tuple to the right is ground, and represents a concrete pair of models. Its relating formula is *true* and therefore we do not show it. As stated before, we do not show the formula of ground tuples but place the attribute values in the attribute compartments instead.

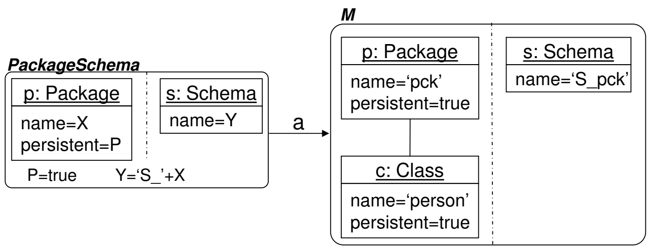


Figure 6. Example of symbolic tuple morphism.

Later, we will need to restrict a symbolic tuple to its source or target part. Hence, given a tuple $P = \langle C_S, C_T, \beta \rangle$, its source restriction is given by $P|_S = \langle C_S, \emptyset, true \rangle$, whereas its target restriction is given by $P|_T = \langle \emptyset, C_T, true \rangle$.

Definition 4. (Symbolic Tuple Morphism)
Given two symbolic tuples $P^i = \langle C_S^i, C_T^i, \beta^i\rangle$ (i=1,2) over the data $\Sigma-$algebra A, a symbolic tuple morphism (short S-morphism) $a = (a_S, a_T)\colon P^1 \to P^2$ is made of two symbolic graph morphisms $a_S\colon C_S^1 \to C_S^2$, $a_T\colon C_T^1 \to C_T^2$ such that $A \models (\gamma^2 \Rightarrow a^{\#}(\gamma^1))$, where $a^{\#}(\gamma^1)$ is the formula obtained by replacing in γ^1 every variable x_i in the first symbolic tuple by $a_{S,V_D}(x_i) \cup a_{T,V_D}(x_i)$.

Remark. $a^{\#}$ accounts for the renaming of variables, and a_{S,V_D} and a_{T,V_D} are the function component of the E-graph morphisms that map the sets of data nodes V_D. Abusing of notation, we sometimes use a instead of $a^{\#}$ for the replacement of variables when no confusion is possible.

Example. Figure 6 shows an S-morphism that identifies the elements with same identifier in the two tuples. The source and target formulas of tuple M imply the source and target formulas of tuple *PackageSchema*, respectively. These implications are demanded by the two symbolic graph morphisms a_S and a_T that make the symbolic tuple morphism a. Moreover, the conjunction of the source, target and relating formulas of tuple M (γ^M) implies the conjunction of the source, target and relating formulas of tuple *PackageSchema* ($\gamma^{PackageSchema}$). The latter implication is demanded by Definition 4.

Please note that two symbolic tuples $P^i = \langle C_S^i, C_T^i, \beta^i\rangle$ (i=1,2) are isomorphic ($P^1 \cong P^2$) if both the source and target symbolic graphs are isomorphic ($C_S^1 \cong C_S^2$, $C_T^1 \cong C_T^2$) and the relating formulae β^i are equivalent.

Symbolic tuples and S-morphisms over a data algebra A form the category **SymbTuple**$_A$. It is easy to see that if we have an S-morphism $a\colon P^1 \to P^2$, then we also have S-morphisms from the source and target restrictions of P^1 to P^2 (i.e. we have $P^1|_S \to P^2$ and $P^1|_T \to P^2$), as well as S-morphisms from the source and target restrictions of P^1 to the source and target restrictions of P^2 respectively (i.e. we have $a|_S\colon P^1|_S \to P^2|_S$ and $a|_T\colon P^1|_T \to P^2|_T$). Finally, given a symbolic tuple P, we also have the inclusion S-morphisms $P|_S \hookrightarrow P$ and $P|_T \hookrightarrow P$.

Our formalization of QVT-R will make use of pushouts and pullbacks of symbolic tuples. Roughly, the former consist on gluing two tuples through a common intersection, and is calculated by performing the pushout of the two symbolic graphs making the tuples, and taking the conjunction of the β formulas. In its turn, the pushout of two symbolic graphs is built by taking the pushout of the graph components and the conjunction of their formulas [18]. Pullbacks model the intersection of two tuples. They are calculated by performing the pullback of the symbolic graphs in the tuples, and taking the disjunction of the source, target and relating formulas. The details and proofs of the properties of these constructions are in Appendix A.

Proposition 1. (Pushouts and Pullbacks of Symbolic Tuples)
Given the span of S-morphisms $\langle C_S^1, C_T^1, \beta^1\rangle \xleftarrow{f} \langle I_S, I_T, \beta^0\rangle \xrightarrow{g} \langle C_S^2, C_T^2, \beta^2\rangle$, the pushout $\langle C_S^1, C_T^1, \beta^1\rangle \xrightarrow{d} \langle P_S, P_T, \beta^1 \wedge \beta^2\rangle \xleftarrow{e} \langle C_S^2, C_T^2, \beta^2\rangle$ is built component-wise, where P_S and P_T are the pushout objects of $C_S^1 \xleftarrow{f_S} I_S \xrightarrow{g_S} C_S^2$ and $C_T^1 \xleftarrow{f_T} I_T \xrightarrow{g_T} C_T^2$ in symbolic graphs.

Given the co-span of S-morphisms $\langle C_S^1, C_T^1, \beta^1\rangle \xrightarrow{f} \langle I_S, I_T, \beta^0\rangle \xleftarrow{g} \langle C_S^2, C_T^2, \beta^2\rangle$, the pullback $\langle C_S^1, C_T^1, \beta^1\rangle \xleftarrow{d} \langle P_S, P_T, \theta\rangle \xrightarrow{e} \langle C_S^2, C_T^2, \beta^2\rangle$ is built component-wise, where P_S and P_T are the pullback objects of $C_S^1 \xrightarrow{f_S} I_S \xleftarrow{g_S} C_S^2$ and $C_T^1 \xrightarrow{f_T} I_T \xleftarrow{g_T} C_T^2$ in symbolic graphs, and θ is the disjunction of β^1 and β^2, which are existentially quantified over the variables of β^1 and β^2 not present in P_S and P_T.

Proof: in Appendix A.

Example. The left of Figure 7 shows a pushout example. In particular, M' is the pushout object of $M \leftarrow F \rightarrow PackageSchema$, calculated by taking the pushout of the two graphs in M and $PackageSchema$ and the conjunction of their formulae. The right of the same figure shows a pullback example, where the relating formula in the pullback object F is the disjunction of the relating formulas in M and $PackageSchema$, each with an existential quantification of the variables that are not present in F. The formulas in the left and right graphs are computed in a similar way.

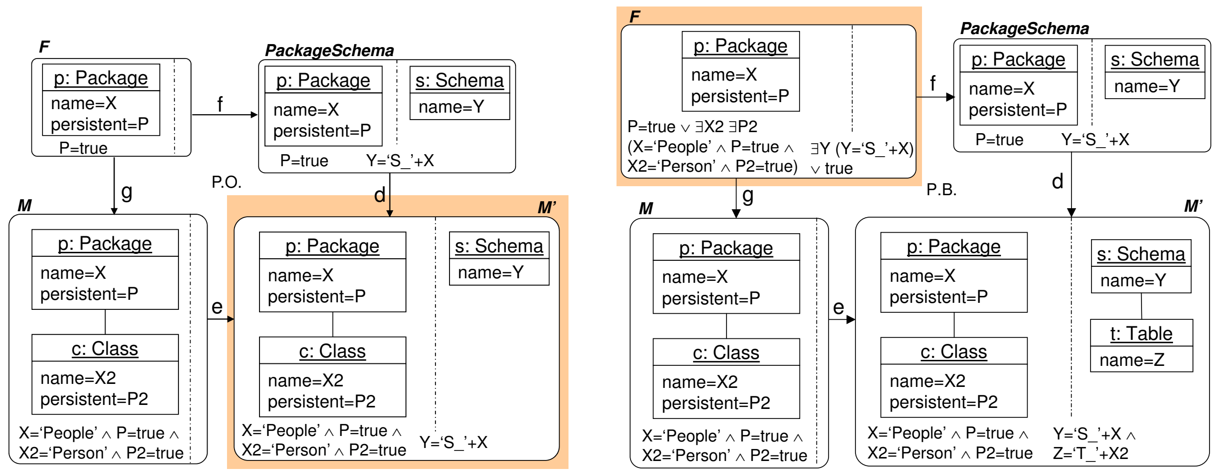


Figure 7. Pushout example (left). Pullback example (right).

The pushout and pullback constructions are straightforward to generalize to finite co-limits and limits, taking the conjunction and disjunction of the formulas respectively [18, 20].

We will use a symbolic tuple to model the source and target domain patterns in a QVT relation. In order to check whether a pair of models is consistent with a given relation in a certain domain, we need to look at symbolic tuples either source-to-target (forwards) or target-to-source (backwards). In this way, given a QVT relation, we construct the so-called forward (backward) pre-condition, which in the simplest case is made of the objects in the source (target) domain. This pre-condition is useful because, when checking the forward (backward) satisfaction of a relation, we will demand that each occurrence of its forward (backward) pre-condition can be extended to an occurrence of the whole symbolic tuple. In case a relation defines a dependency with other relations in its *when* section, or is invoked from the *where* section of other relations, the forward (backward) pre-condition consists of the amalgamation of the source (target) objects and the parameters in the invocations.

For this reason we introduce a construction called the forward and backward interpretation of a given S-morphism or set of S-morphisms. The co-domain of the S-morphism represents a QVT relation. Its domain may represent the parameters received by the relation in a *where* invocation, or those passed from a *when* invocation in the relation. We need to consider sets of S-morphisms because a relation may have several *when* dependencies and be called from different *where* sections of other relations.

The forward interpretation of an S-morphism $D \rightarrow R$ (with D representing the *when* or *where* parameters and R the QVT relation) is calculated by making the pushout of $D \hookleftarrow D|_S \overset{d|_S}{\rightarrow} R|_S$. The forward interpretation of a set of S-morphisms with same co-domain generalizes the previous pushout construction by taking the amalgamation of all S-morphisms in the set. The backward interpretation is similar but taking the target restriction of D and R. From now on, we just present the different concepts for the forward direction, as those for backwards are analogous.

Definition 5. (Forward Interpretation of S-morphisms)

1. Given an S-morphism $d\colon D \rightarrow R$, its forward interpretation is given by $F_d(R) = u\colon F(d) \rightarrow R$ as the construction to the left of Figure 8 shows, where $F(d)$ is the pushout object, and u exists due to the pushout universal property.

2. Given a symbolic tuple R and a non-empty set of S-morphisms $DS = \{D_j \overset{d_j}{\rightarrow} R\}_{j \in J}$, its forward interpretation is given by $F_{DS}(R) = u\colon F(DS) \rightarrow R$ as the center of Figure 8 shows, with I the limit of $\{p_j\}$, $F(DS)$ the colimit of $\{i_j\}$, and u exists due to the co-limit universal property.

3. If the set DS is empty, then by definition we consider an S-morphism from the empty symbolic tuple to R, $F_\emptyset(R) \overset{def}{=} F(\emptyset \rightarrow R) \cong R|_S \hookrightarrow R$, as the right of Figure 8 shows. An empty symbolic tuple is made of two empty graphs and the *true* formula.

$$\begin{tikzcd}
 & D|_S \arrow[dl, hook'] \arrow[dr, "d|_S"] & \\
D \arrow[dr, "d^S"] \arrow[ddr, bend right, "d"'] & \text{P.O.} & R|_S \arrow[dl, "p^S"'] \arrow[ddl, hook', bend left] \\
 & F(d) \arrow[d, dotted, "u"] & \\
 & R &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & I \arrow[dl, "i_0"'] \arrow[dr, "i_n"] & \\
F(d_0) \arrow[dr, "e_0"] \arrow[ddr, "p_0"'] & \dots & F(d_n) \arrow[dl, "e_n"'] \arrow[ddl, "p_n"] \\
 & F(DS) \arrow[d, dotted, "u"] & \\
 & R &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \emptyset \arrow[dl, hook'] \arrow[dr, "\emptyset"] & \\
\emptyset \arrow[dr, "\emptyset"] \arrow[ddr, bend right, "\emptyset"'] & \text{P.O.} & R|_S \arrow[dl] \arrow[ddl, hook', bend left] \\
 & R|_S \arrow[d, dotted, "u"] & \\
 & R &
\end{tikzcd}$$

Figure 8. Forward interpretation of an S-morphism d (left), of a non-empty set $\{d_j\}$ of S-morphisms (center), and of an empty set $\emptyset$ of S-morphisms (right).

Example. Figure 9 shows to the left the forward interpretation of the S-morphism $when \rightarrow ClassTable$. It amalgamates the symbolic tuple *when* together with the source symbolic graph of *ClassTable*. Due to the pushout universal property, we obtain a unique morphism from $F(when \to ClassTable)$ to *ClassTable*. The center of the figure shows two S-morphisms from tuples $D1$ and $D2$ to another tuple R, and the right shows their forward interpretation. In particular, the latter is built by taking the forward interpretation of each S-morphism separately ($F(d1)$ and $F(d2)$), calculating their limit together with R (the result is I), and making the colimit of I and the forward interpretations (obtaining $F(DS)$).

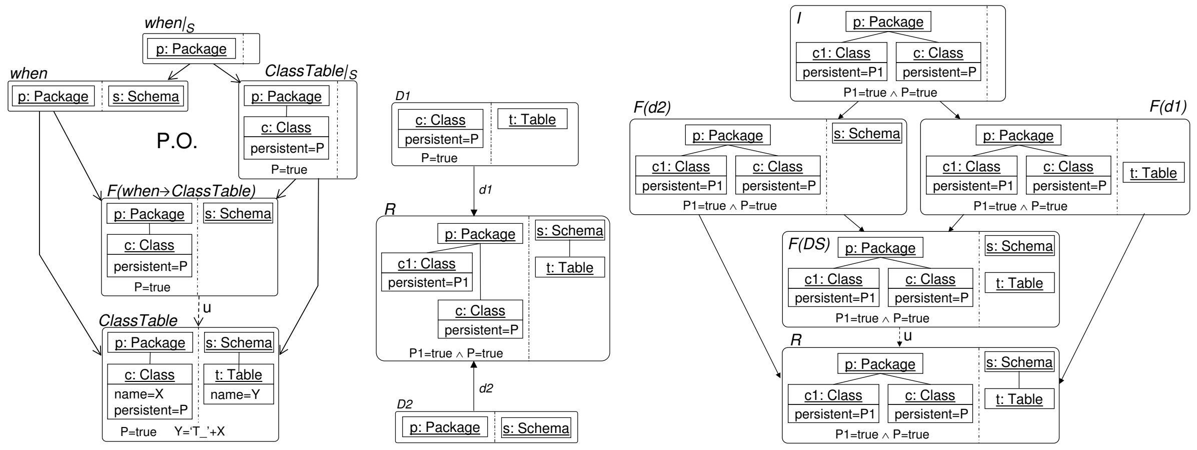


Figure 9. Example of forward interpretation of an S-morphism (left). Example of a set of two S-morphisms (center) and their forward interpretation (right).

Once we have formalised pairs of models and some useful constructs, we are in the position to formalise the check-only semantics of QVT-R.

4. An Algebraic Semantics for QVT-Relations

A QVT-R specification is made of a set of symbolic tuples, some of which are top. The *when* and *where* dependencies are formalized as two sets of spans of S-morphisms. In this way, if the *when* section of relation R_0 has a dependency to relation R_1, we model it with the span: $R_0 \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_1$, where the symbolic tuple D contains the parameters passed from R_0 to R_1. We restrict to the case in which QVT relations have exactly two domains.

Definition 6. (QVT-R Specification)
A QVT-R specification $S = \langle r, top \subseteq r, when, where \rangle$ is made of:

1. a set $r = \{R_i\}$ of symbolic tuples, called QVT relations or simply relations;
2. a set *top* $\subseteq r$ of top-level relations;
3. a set $when = \{R_i \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_j\}$ of *when* dependencies, with $R_i, R_j \in r$, $R_j \in top$, and $src\colon D \to R_i$, $tar\colon D \to R_j$ two S-morphisms. Given a relation R_i, we use the notation

$when(R_i) = \{R_i \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_k | R_i \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_k \in when\}$ for the set of *when* dependencies for R_i;

4. a set $where = \{R_i \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_j\}$ of *where* dependencies, with $R_i, R_j \in r$, $R_j \notin top$, and $src\colon D \rightarrow R_i$, $tar\colon D \rightarrow R_j$ two S-morphisms. Given a relation R_i, we use the notation $where(R_i) = \{R_i \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_k | R_i \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R_k \in where\}$ for the set of *where* dependencies for R_i

such that S does not contain cycles of *where* or *when* dependencies. S contains a cycle of dependencies if there is a set $dep = \{R_i \overset{src_i}{\leftarrow} D_i \overset{tar_j}{\rightarrow} R_j, R_j \overset{src_j}{\leftarrow} D_j \overset{tar_k}{\rightarrow} R_k, \ldots, R_n \overset{src_n}{\leftarrow} D_n \overset{tar_i}{\rightarrow} R_i\} \subseteq when \uplus where$.

Remark 1. Our formalization defines the *when* and *where* dependencies globally, whereas in the standard notation, a relation R_i has associated a *when* section $when(R_i)$ and a *where* section $where(R_i)$. In this way, our formalization provides more flexibility as relations do not encode which of their elements are formal parameters (variables marked as *domain*), but instead, a relation can be invoked in different ways from different relations.

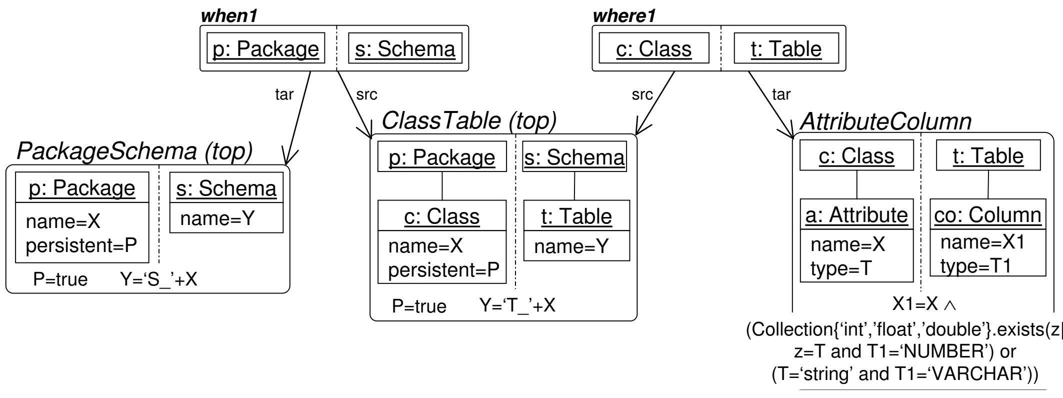
Remark 2. We impose two conditions in the *when* and *where* dependencies. First, a *when* dependency should refer to top-level relations (item 3, condition $R_j \in top$) only. Second, a top-level relation cannot be invoked from a *where* section (item 4, condition $R_j \notin top$). This corresponds well to practice, and the available QVT-R tools also implement these restrictions (see Section 5 for a brief discussion on tooling issues).

Remark 3. As we will notice later, the satisfiability of a relation is a procedure inherently recursive. Definition 6 forbids cycles of dependencies to avoid infinite recursion. In Section 4.1 we will discuss potential issues arising if no restriction is given.

Example. Figure 10 depicts the example QVT-R specification in Figure 2 using our formalization. Relation *ClassTable* is the only one defining dependencies to other relations in the *when* and *where* clauses. These dependencies are tuples containing the elements passed as parameters to the other relations, together with the corresponding S-morphisms. The OCL expressions in the original relations have been added to the formula of the left/right symbolic graph if they only contain variables from them, or to the relating formula β in the middle if they relate variables of both graphs. The relations contain an indication of whether they are *top* or not. The explicit representation used in the figure is only to illustrate the formal definition, but we can use the standard visualization shown in Figure 2.

Given a specification S and one of its relations R, we use the notation $when|_R = \{R \overset{src}{\leftarrow} D | R \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R' \in when\}$ to denote a set like $when(R)$ but containing only the first S-morphism src in the spans that have R as the co-domain. An analogous notation is used for the set *where*.

Next, we define the forward pre-condition of a relation as the forward interpretation of its *when* dependencies, using the construction shown in Definition 5. The forward pre-condition contains the part



UML_RDBMS= { **r**={PackageSchema, ClassTable, AttributeColumn}, **top**={PackageSchema, ClassTable},
when = { ClassTable←when1→PackageSchema }, **where** = { ClassTable←where1→AttributeColumn }}

Figure 10. The example QVT-R specification using our theoretical notation.

of the relation that we will look for in the models in order to consider that the relation is enabled in the models and should be satisfied.

Definition 7. (Forward Pre-condition of QVT Relation)
Given a QVT specification S and a relation $R \in r$, its forward pre-condition $Pre^F(R)$ is the domain of the forward interpretation $F_{when|_R}(R)$.

Remark. If the set $when|_R$ is empty, then $F_{when|_R}(R)$: $(Pre^F(R) = R|_S) \rightarrow R$ according to Definition 5.

Example. The left of Figure 9 shows the forward pre-condition $Pre^F(ClassTable) = F(when \rightarrow ClassTable)$ for relation *ClassTable*, which is built by making the pushout of the tuple containing the elements appearing in its *when* dependency, together with the source restriction of its symbolic tuple. Should *ClassTable* have defined more dependencies in its set *when*, we should have taken the forward interpretation for all of them, according to Definition 5.

Next we define the conditions for a relation R to be forward-enabled in a tuple M, where M contains the models where we want to check the satisfaction of R. In particular, a relation is forward-enabled if we find one occurrence of its forward pre-condition $Pre^F(R)$, for which each *when* dependency is satisfied. As we will see later, our semantics demands each forward-enabled top relation to be satisfied. Satisfaction is checked using a predicate, presented in Definition 11. Hence, as the general notions of enabledness and satisfaction involve a mutual recursion, we will start with the simpler case of enabledness of relations with empty *when*, then continue with the satisfaction of relations with empty *when* and *where* (which are the base cases of the recursion), and finally present the general cases of enabledness and satisfaction.

A relation R with empty *when* is forward-enabled at any match of its forward pre-condition.

Definition 8. (Forward Enabledness of QVT Relation with Empty *When*)
Given a QVT specification S and a relation $R \in r$ such that $when(R) = \emptyset$, we say that R is forward-enabled in a symbolic tuple M if there is a match $m^S \colon Pre^F(R) \to M$ of its forward pre-condition, written $M \vdash_{m^S,F} R$. See the left of Figure 11.

$$\begin{tikzcd}
Pre^F(R) \arrow[r, "u"] \arrow[d, "m^S"'] & R \\
M &
\end{tikzcd}
\qquad
\begin{tikzcd}
Pre^F(R) \arrow[r, "u"] \arrow[d, "m^S"'] \arrow[dr, phantom, "="] & R \arrow[dl, bend left, "m"] \\
M &
\end{tikzcd}$$

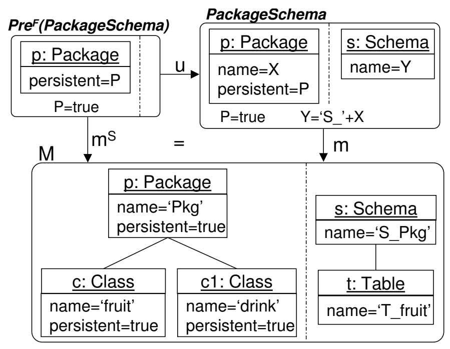


Figure 11. Forward enabledness of a relation with empty *when* (left). Satisfaction of a relation with empty *when* and *where* (center). Example of enabledness and satisfaction (right).

Example. The right of Figure 11 shows a tuple M where the relation *PackageSchema* is forward-enabled. The relation has an empty set *when*, so according to Definition 5, the forward pre-condition is calculated using the empty dependency $\emptyset \to R$, which yields a pre-condition equal to the source restriction of the tuple ($PackageSchema|_S$). As there is an occurrence of the forward pre-condition in M, the relation is forward-enabled in M at such occurrence.

Next we present the satisfaction of a relation with empty *when* and *where*. In such a case, we only need to find a commuting S-morphism with that of the pre-condition, as shown in the central diagram of Figure 11. However, in a QVT-R specification, the satisfaction of a relation sometimes needs to be checked with respect to a calling relation. That is, taking into account some parameters passed from the calling relation, so that the occurrence of the called relation is checked in a certain location in M. In this way, we define satisfaction using a general predicate SAT^F which receives three parameters: (i) the relation R to be checked for satisfaction, (ii) an S-morphism $m_D \colon D \to M$ indicating the actual parameters received by R from an invoking relation (i.e., the place in the model M where the binding is to be sought), and (iii) a dependency $d \colon D \to R$ defining the formal parameters of such an invocation. In the simplest case of a relation R with empty *when* and *where*, a tuple M forward-satisfies R if $SAT^F(R, D \to M, D \to R)$ evaluates to true.

Definition 9. (Forward Satisfaction of QVT Relation with Empty *When* and *Where*)
Given a QVT specification S and a relation $R \in r$ such that $when(R) = where(R) = \emptyset$, we say that R is forward-satisfied if predicate $SAT^F(R, m_D \colon D \to M, d \colon D \to R)$ holds.

Predicate SAT^F holds for R (with $when(R) = where(R) = \emptyset$) and the given S-morphisms, if $\forall m^S \in \{m^S \colon F(d) \to M \mid m_D = m^S \circ e,\ with\ M \vdash_{m^S,F} R, D \overset{e}{\to} F(d)\}$, $\exists m \colon R \to M$ s.t. (1) and

(2) commute in Figure 12.

$$\begin{tikzcd}
F(d) \arrow[r, "u"] \arrow[dr, "m^S"'] & R \arrow[d, "m"] \arrow[l, phantom, shift right=6, "(1)"] \arrow[r, phantom, shift right=6, "(2)"] & D \arrow[l, "d"'] \arrow[dl, "m_D"] \arrow[ll, bend right, "e"', "="] \\
& M &
\end{tikzcd}$$

Figure 12. Forward satisfaction of a relation with empty *when* and *where* dependencies.

Remark. As we will see later, the satisfaction of top relations requires the predicate SAT^F to hold for the empty S-morphisms $m_D\colon D = \emptyset \to M$ and $d\colon D = \emptyset \to R$, which make any m commute with m_D in Figure 12. In such a case, as d is empty, $F(\{\emptyset \to R\})$ is the source restriction of R. Hence, for top relations without dependencies, the diagram in Figure 12 is equivalent to the one in the center of Figure 11, where all possible m^S morphisms forward-enabling R are sought.

Example. The right of Figure 11 shows a tuple M where the relation *PackageSchema* is forward-satisfied. This is so as there is only one match m^S at which the relation is forward-enabled, and for which we find a commuting match m of the whole relation, making the relation forward-satisfied.

Next we generalize the definition of forward enabledness for any QVT relation with an arbitrary set *when* (Definition 8 was the base case of the recursion). In the general case, in addition to demand a match m^S from the forward pre-condition to M, we ask for the satisfaction of its *when* dependencies in a match commuting with m^S. We use the predicate SAT^F with appropriate parameters to check this satisfaction.

Definition 10. (Forward Enabledness of QVT Relation)
Given a QVT specification S and a relation $R \in r$, we say that R is forward-enabled in a symbolic tuple M at match $m^S\colon Pre^F(R) \to M$ of its forward pre-condition, written $M \vdash_{m^S,F} R$, if $\forall R \overset{src}{\leftarrow} D \overset{tar}{\rightarrow} R' \in when(R)$, $SAT^F(R', m^S \circ e \circ g, tar)$ holds. See Figure 13, and Definitions 9 and 11 for the formulation of predicate SAT^F.

Example. The right of Figure 13 shows an example where we test the enabledness of relation *ClassTable*, which has a non-empty *when* set. In this case, in addition to finding a match m^S from the pre-condition $Pre^F(ClassTable)$ to M, we need to check the satisfaction of all relations in the *when* set commuting with m^S. For this latter case we invoke predicate SAT^F with the relation *PackageSchema*, the match where this relation should be satisfied (objects p and s and the corresponding match $m^S \circ e \circ g$), and the dependency $when1 \overset{tar}{\rightarrow} PackageSchema$ which is used to build the pre-condition of *PackageSchema*. According to Definition 9, the predicate holds for these parameters, meaning that *PackageSchema* is satisfied at the passed match as we find a commuting S-morphism $PackageSchema \overset{m'}{\rightarrow} M$. Altogether, *ClassTable* is forward-enabled in M.

$$\begin{tikzcd}
F(d) \arrow[d, "e"] & & D \arrow[ll, "g"'] \arrow[dl, "src"'] \arrow[dr, "tar"] & \\
Pre^F(R) \arrow[r, "u"] \arrow[d, "m^S"] & R & & R' \\
M & & &
\end{tikzcd}$$

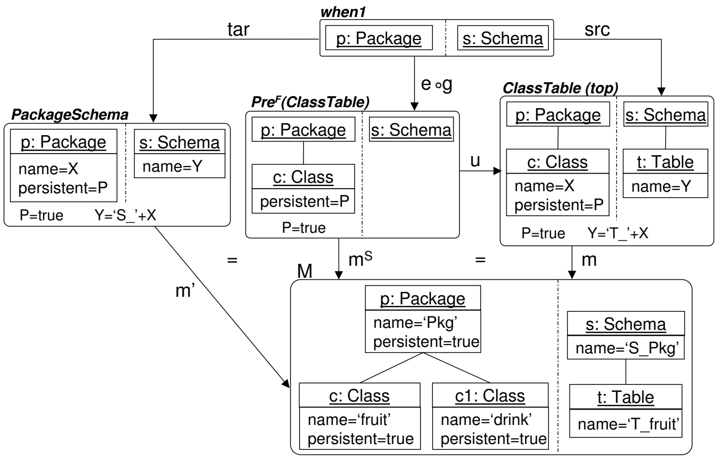


Figure 13. Forward enabledness of QVT relation (left). Example (right).

Next, we present the general case for satisfaction of a relation in a given span of S-morphisms, i.e. in the context of a certain invocation from either a *when* or a *where* clause. This is checked through predicate SAT^F with three parameters: (i) the relation R to be checked, (ii) an S-morphism $D \to M$ with the actual parameters received from the invocation, and (iii) a dependency $D \to R$ defining the formal parameters of the invocation. The second parameter of the predicate is used to restrict the occurrences of the forward pre-condition $Pre^F(R)$ in M that need to be satisfied, whereas the third parameter is treated as an additional pre-condition in the *when* clause of the relation. In its turn, the predicate may demand the satisfaction of other relations at certain matches if they appear in the *when* or *where* clauses of R. This is handled by recursive calls to the predicate, as stated in Definitions 10 and 11. Thus, our definition takes into account that the satisfaction of a particular relation is checked in the context of a specification S, and it depends on how it is actually invoked from other relations. In this way, a specification can require a certain relation to be satisfied under different conditions if the SAT^F predicate is checked with different second and third parameters, meaning that the relation is invoked from different places or with different sets of parameters.

Definition 11. (Forward Satisfaction of a QVT Relation R at a Span $R \overset{d}{\leftarrow} D \overset{m_D}{\rightarrow} M$)
Given a QVT specification S, a relation $R \in r$, and a span s of S-morphisms $R \overset{d}{\leftarrow} D \overset{m_D}{\rightarrow} M$, we say that R is forward-satisfied at s if predicate $SAT^F(R, m_D\colon D \to M, d\colon D \to R)$ holds.

Predicate SAT^F holds for R and $R \overset{d}{\leftarrow} D \overset{m_D}{\rightarrow} M$ if $\forall m^S \in \{m^S\colon F(\{d\} \cup when|_R) \to M \mid m_D = m^S \circ e,\ with\ M \vdash_{m^S,F} R, D \overset{e}{\rightarrow} F(\{d\} \cup when|_R)\}$, $\exists m\colon R \to M$ s.t.:

i) (1) and (2) commute to the left of Figure 14, and

ii) $\forall R \overset{src}{\leftarrow} D' \overset{tar}{\rightarrow} R' \in\ where(R),\ SAT^F(R', m \circ src, tar)$ holds (see the right of Figure 14).

$$\begin{tikzcd}
F(\{d\} \cup when|_R) \arrow[r, "u"] \arrow[dr, "m^S"'] \arrow[r, phantom, "(1)", shift right=4ex] & R \arrow[d, "m"] \arrow[r, phantom, "(2)", shift right=4ex] & D \arrow[l, "d"'] \arrow[dl, "m_D"] \arrow[ll, bend right, "e"', "="] \\
& M &
\end{tikzcd}
\qquad
\begin{tikzcd}
R \arrow[d, "m"] & D' \arrow[l, "src"'] \arrow[r, "tar"] & R' \\
M & &
\end{tikzcd}$$

Figure 14. Forward satisfaction (left). Forward satisfaction of the *where* dependencies (right).

Example. Relation *ClassTable* is forward-enabled at two occurrences in the tuple M to the right of Figure 13. The first one with objects $\{$`p, s, c`$\}$ satisfies the relation because there is an occurrence of the relation that includes these objects and, in addition, the *where* dependencies are satisfied (*AttributeColumn* is trivially satisfied as `c` has no attributes). However, the second one at objects $\{$`p, s, c1`$\}$ does not satisfy the relation because we do not find one occurrence of the relation for the objects (i.e. a table with suitable name is missing).

The forward satisfaction of a QVT-R specification by a model pair M demands that M forward-satisfies all top-level relations in the specification. For this purpose we use the predicate SAT^F.

Definition 12. (Forward Satisfaction of QVT-R Specification)
Given a QVT-R specification $S = \langle r, top \subseteq r, when, where\rangle$ and a symbolic tuple M, we say that M forward-satisfies S, written $M \models_F S$, if $\forall R \in top$, $SAT^F(R, \emptyset \rightarrow M, \emptyset \rightarrow R)$.

Example. Our example specification has two top-level relations: *PackageSchema* and *ClassTable*. If we consider the model pair M to the right of Figure 13, we see that relation *PackageSchema* is forward-satisfied because there is only one occurrence of its forward pre-condition in M, which is satisfied. However, relation *ClassTable* is not forward-satisfied because a suitable table for class `c1` is missing. As a result, M does not forward-satisfy the specification. More in detail, our formalization performs the following steps to check the satisfaction:

1. check satisfaction of top-level relation *PackageSchema*, which is evaluated through predicate SAT^F $(PackageSchema, \emptyset \rightarrow M, \emptyset \rightarrow PackageSchema)$. The relation does not have dependencies, therefore the predicate does not perform any recursion, but directly evaluates to $true$.

2. check satisfaction of top-level relation *ClassTable*, which is evaluated through predicate SAT^F $(ClassTable, \emptyset \rightarrow M, \emptyset \rightarrow ClassTable)$ (Definition 11). There are two matches of the forward pre-condition of *ClassTable* in M: m_1^S identifying the class in the relation with `c` in M, and m_2^S identifying it with `c1` (see Figure 15(a)).

   (a) retain the matches that forward-enable relation *ClassTable* (Definition 10). Both m_1^S and m_2^S forward-enable the relation as *PackageSchema* is satisfied at both matches or, more precisely, the predicates $SAT^F(PackageSchema, m_1^s \circ e \circ g, tar)$ and $SAT^F(PackageSchema, m_2^s \circ$

$e \circ g, tar)$ hold (see Figure 15(b) for building morphism $m_D = m_1^s \circ e \circ g$, and Figure 15(c) for the satisfaction of the first predicate).

(b) check if there is a morphism m from relation *ClassTable* to M commuting with m_1^S and m_D, which exists.

(c) check if there is a morphism m from relation *ClassTable* to M commuting with m_2^S and m_D, which does not exist (see Figure 15(d)). Therefore relation *ClassTable* is not satisfied.

A model pair is synchronized with respect to a specification if it satisfies the specification both forwards ($\models_F$) and backwards ($\models_B$). In this section we have presented enabledness and satisfaction for the forward case. The backward case is similar but taking the target restrictions of relations. Actually, forward and backward satisfaction are symmetrical exchanging source and target models.

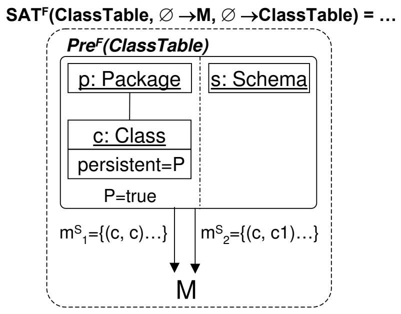
Definition 13. (Synchronization)
Given a QVT-R specification S and a symbolic tuple M, we say that M is synchronized with respect to S, written $M \models S$, iff $M \models_F S$ and $M \models_B S$.

Example. Figure 16 shows two model pairs that are synchronized with respect to our specification, as both top relations *PackageSchema* and *ClassTable* are forward- and backward-satisfied at all occurrences. In the model pair to the right, the two occurrences that forward-enable *ClassTable* are satisfied with the same table.

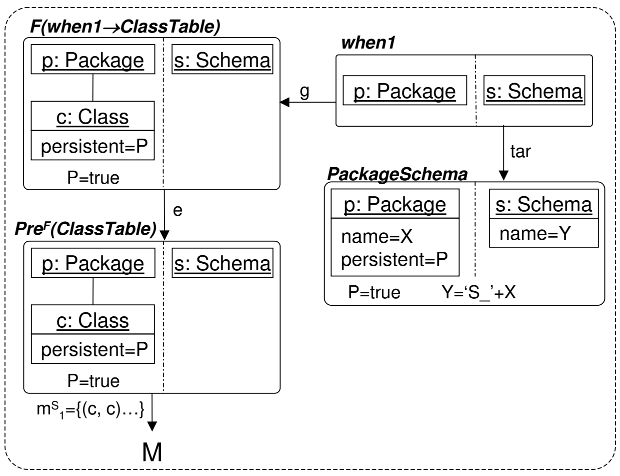
4.1. Recursion

According to the standard [22], a *when* clause in a relation R specifies the conditions under which R needs to hold. This definition is inherently recursive as, in order to check the enabledness of R, first we need to check the satisfaction of the relations referenced in its *when* clause. Moreover, for R to hold, all conditions in its *where* clause need to hold. This condition is again recursive as in order to check the satisfaction of R, we need to check the satisfaction of all relations invoked in its *where* clause as well. For this reason our SAT^F predicate is recursive, and the base case is the evaluation of a relation with empty *when* and *where* (Definition 9). Definition 6 forbids cycles of *when* or *where* dependencies, hence avoiding any problematic situation due to infinite recursion.

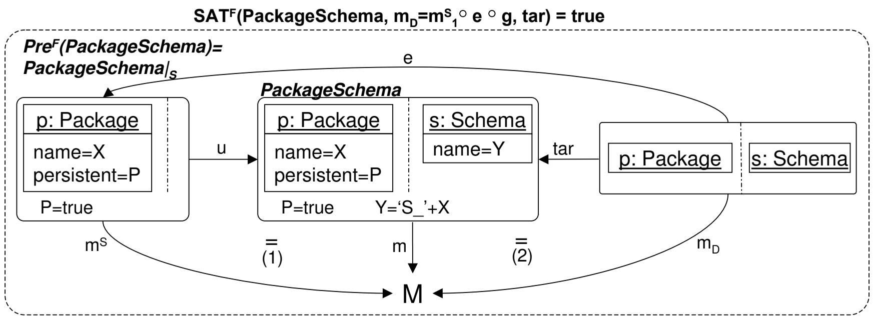
Should we refrain from imposing restrictions on the *when* and *where* dependencies, paradoxes could arise in the evaluation of satisfaction. For example, assume a specification made of three top relations R_1, R_2 and R_3 in which there is a cycle of *when* dependencies: R_1 includes R_2 in its *when* clause, R_2 includes R_3 in its *when* clause, and R_3 includes R_1 in its *when* clause. Let us suppose that we have a tuple M representing a pair of models where we want to check if they (forward) satisfy our specification. Let us assume that each R_i have a match $m_i^S\colon Pre^F(R_i) \to M$, so that whether they are enabled or not depends on the satisfaction of their *when* dependencies. Finally, let us suppose that if R_1 is enabled



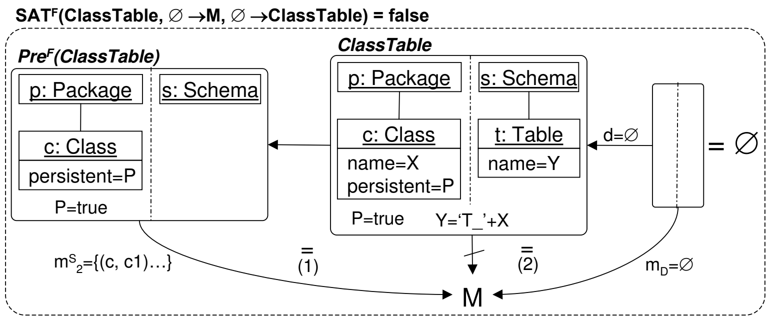
(a) Two pre-condition matches in M.



(b) Building morphism $m_D = m_1^S \circ e \circ g$.



(c) Checking satisfaction of relation *PackageSchema* at a given span.



(d) No commuting match for m_2^S.

Figure 15. Evaluating the satisfaction of *ClassTable*.

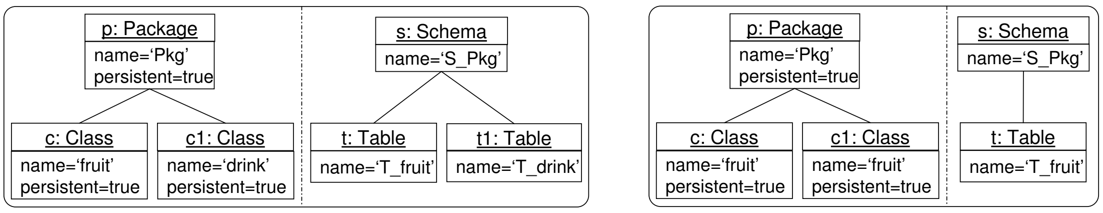


Figure 16. Example model pairs synchronized with respect to the specification.

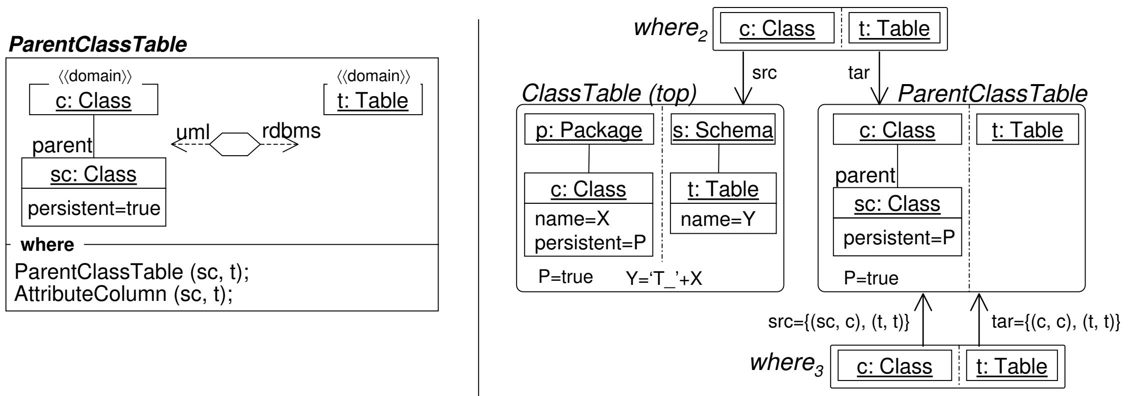
for the given match then it is not satisfied by M (there is no match $m\colon R_1 \to M$) and similarly for R_2 and R_3. The question is, does M satisfy R_1 (or R_2 or R_3)? We can check that any answer leads to a contradiction:

- Assume that M satisfies R_1 for the given match. This means that R_1 cannot be enabled, since otherwise – according to the initial assumption – R_1 would not be satisfied. This means that its *when* dependency R_2 is not satisfied by M. But if R_2 is not satisfied by M, then R_2 is enabled, and this means that R_3 is satisfied. But if R_3 is satisfied, then it is not enabled, which means that R_1, its *when* dependency, is not satisfied, leading to a contradiction.

- Assume then that M does not satisfy R_1 for the given match. This means that R_1 is enabled, and hence its *when* dependency R_2 is satisfied. Now, if R_2 is satisfied, it is not enabled, which means that R_3 is not satisfied. But if R_3 is not satisfied it means that it is enabled, which means that R_1, its *when* dependency, is satisfied. This is also a contradiction.

In order to avoid these paradoxes, we need some kind of restriction to forbid certain types of circularities. The restrictions posed in this paper (i.e. the absence of cycles of *when* and *where* dependencies) are however too restrictive in practice. For instance, Figure 17 shows an increment of the QVT specification in Figure 10 with one additional non-top relation *ParentClassTable* (shown to the left in QVT-R syntax, and to the right using our formalization). The resulting specification is similar to the example presented in the Appendix of the standard [22]. The new relation is invoked from the *where* section of relation *ClassTable*, and then it invokes itself in its *where* section to ensure that the table associated to a class contains columns for the attributes of the class's parents (checked by relation *AttributeColumn*). Attributes of indirect parents are handled recursively. Thus, the specification contains a cycle of dependencies given by $ParentClassTable \overset{src}{\leftarrow} where_3 \overset{tar}{\rightarrow} ParentClassTable$.

This cycle is however not problematic unless the model under test contains a cycle in the inheritance relation, in which case a program that simply evaluates the recursion (depth first) following the dependencies will not terminate. Instead, the program may be provided with a *cache*, which prevents evaluating satisfaction twice with same parameters in a chain of dependency evaluation. For example, Figure 18 shows a pair of models, where the one to the left has a cycle in the inheritance relation (we assume this



UML_RDBMS=
{ **r**={PackageSchema, ClassTable, AttributeColumn, ParentClassTable},
top={PackageSchema, ClassTable},
when = { ClassTable$\xleftarrow{}$when$_1$$\xrightarrow{}$PackageSchema },
where = { ClassTable$\xleftarrow{}$where$_1$$\xrightarrow{}$AttributeColumn,
ClassTable$\xleftarrow{}$where$_2$$\xrightarrow{}$ParentClassTable,
ParentClassTable$\xleftarrow{}$where$_3$$\xrightarrow{}$ParentClassTable,
ParentClassTable$\xleftarrow{}$where$_4$$\xrightarrow{}$AttributeColumn}}

Figure 17. Adding another relation to the example: A cycle.

is allowed by the meta-model). *PackageSchema* is satisfied in M since $SAT^F(PackageSchema, \emptyset \rightarrow M, \emptyset \rightarrow PackageSchema)$ holds as there is a suitable schema s for the only package p. Then, a program might evaluate $SAT^F(ClassTable, \emptyset \rightarrow M, \emptyset \rightarrow ClassTable)$ because *ClassTable* is also top. The relation is enabled at two matches m_1^S and m_2^S identifying classes c1 and c2, for which find two commuting matches m_1 and m_2 identifying the corresponding tables t1 and t2 (condition (i) of Definition 11). Then, the program would need to evaluate SAT^F on the two *where* dependencies of $ClassTable$. In order to evaluate the first one (relation *ParentClassTable*) at the match for class c1, the program would need to check the satisfaction of relation *ParentClassTable* for class c2, and the other way round, obtaining a recursive cycle. Nonetheless, as just stated, avoiding this kind of cycles can be done if the program does not evaluate the predicate SAT^F on the same relation at the same match twice in a sequence of recursive calls. Instead, the second time the predicate is not evaluated, but only the other dependencies are checked. In our example, this means that the program would continue evaluating the other *where* dependencies of *ParentClassTable*, namely the dependency of *AttributeColumn*, which is satisfied. Therefore, the program would find that the specification is satisfied in M.

This idea of a *cache* can be used by tool builders to break certain cycles. However, finding the right kind of restrictions in Definition 6 to avoid paradoxes but allowing non-problematic circularities is an open question for us.

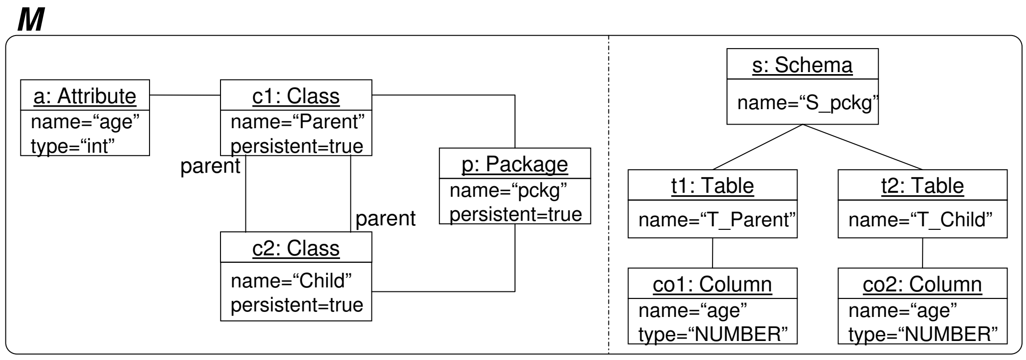


Figure 18. A model tuple with a cycle.

5. Validating the Formalization

In this section, we show some validation results of the proposed semantics using as a reference existing tools and other semantics proposed in the literature.

ModelMorf [16] is the only tool we are aware of that implements the checkonly scenario. In this tool, QVT transformations are written using textual syntax, therefore we encoded the transformation shown in Figure 2 in textual format. Attribute computations were an issue, as ModelMorf does not support bidirectional constraints like the one in relation *AttributeColumn*, but only attribute assignments. Thus, we had to transform the constraint in this relation into an assignment expression that explicitly assigned a value to *T1*. Note that if the transformation is meant to be used both forwards and backwards, we need to synthesize assignment expressions for the attributes in both domains. Hence, in ModelMorf, we have to provide the forward and backward version of the β formula in our symbolic tuples. Apart from this, the checkonly execution of the transformation over the model pairs shown in the figures of the paper returned the expected results.

We also tested how the tools handle some cases of infinite recursion due to dependency cycles among relations. For the example presented in Section 4.1 (a recursive call to relation *ParentClassTable* evaluated on the model in Figure 18), ModelMorf returns *false* not only in the check-only scenario, but surprisingly, also in the transformation scenario. However the example model does satisfy the specification. The same example executed in the MediniQVT [15] tool for the transformation scenario (as this tool does not support checkonly) causes a non-terminating execution. We also experimented with other cyclic transformations, obtaining the same results: while MediniQVT entered in a non-terminating loop, ModelMorf always returned false. Other proposed semantics, as Stevens' game semantics for QVT-R currently forbids cycles of when and where dependencies [26], just like our Definition 6, so could not be used for comparison. Therefore we discovered that different tools implement heterogeneous approaches to recursive cycles. The procedure that we sketched before would prevent checking the satisfaction of a relation for the same elements twice, so that the checking procedure terminates and, in this particular example, it returns *true*.

We have also applied our formalization to the transformation given in [26], where the author proposes a semantics for QVT-R checkonly scenarios based on game theory. Figure 19 shows the transformation using our formalization. It uses the same meta-model for the source and target, and consists of three relations. Relation *ContainersMatch* is top. It demands connected *Container* and *Inter* objects in source and target, and invokes relation *IntersMatch* in its *where* section using both *Inter* objects as parameters. Relation *IntersMatch* demands one connected *Thing* object in both domains, and invokes relation *Things-Match* using both *Thing* objects. Finally, relation *ThingsMatch* asks the attributes of the *Thing* objects to have the same value.

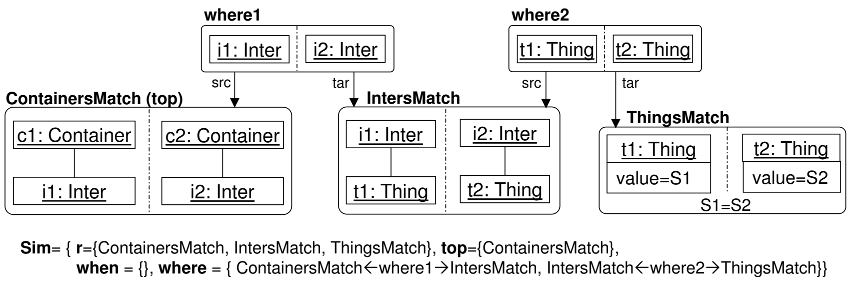


Figure 19. Example QVT-R transformation, adapted from [26].

Figure 20 illustrates the process of forward satisfaction checking, over a model pair M borrowed from [26]. M satisfies the transformation both forwards and backwards. The top of the figure depicts the pre-condition Pre^F of the three relations, and how parameters are passed. The forward satisfaction of the specification is evaluated through $SAT^F(ContainersMatch, \emptyset \to M, \emptyset \to ContainersMatch)$, as this is the only top relation. The pre-condition for *ContainersMatch* is made of the source objects, for which there is only one match m_C^S in M shown by equality of identifiers. The satisfaction of predicate SAT^F demands the existence of one match for *ContainersMatch* commuting with the pre-condition and satisfying the *where* dependencies. As it can be observed, there are two matches of *ContainersMatch* commuting with m_C^S: {`c1, i1, c2, i2`} (depicted $m_{C,i2}$) and {`c1, i1, c3, i3`}. However, only the first one satisfies the *where* dependencies. The parameters passed through *where1* demand the satisfaction of *IntersMatch* in the same *Inter* objects identified by the match of relation *ContainersMatch*: `i1` and `i2`. The pre-condition of *IntersMatch* has two matches: $m_{I,tc1}^S$ with objects {`i1, i2, tc1`}, and $m_{I,td1}^S$ with objects {`i1, i2, td1`}. For each of them, there is a commuting match of the whole relation at matches $m_{I,tc1}$ and $m_{I,td1}$, respectively. Finally, relation *ThingsMatch* is invoked, which checks equality of values.

Altogether, our evaluation forward and backward coincides with that of [26] and the one given by ModelMorf (up to cycles). We have tested our semantics with other examples, including the ones given

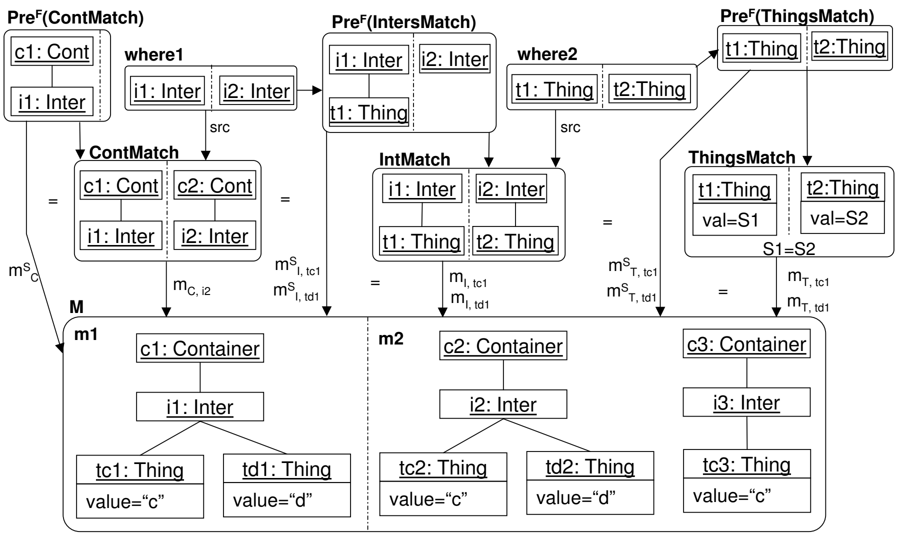


Figure 20. Forward transformation checking.

in [26]. This does not prove that our semantics capture the idea the authors of the QVT standard had in mind, but it provides further evidence in favour of such a hypothesis.

6. Related Work

Even though QVT is the standard language for model transformation in the MDA, very few attempts to formalize its semantics can be found in the literature, and even less address the formalization of the check-only scenario. The only approach we are aware of is the seminal work of Stevens in [26]. This is the first formal semantics for QVT-R checkonly transformations, and is based on game theory. The semantics is described as a game between a refuter and a verifier, which at each state of the game pick bindings in the source and target domains of the relations under evaluation. This semantics is high-level in the sense that some concepts, like the binding, are left as an oracle. In contrast, our semantics relies on an explicit formalization of bindings (S-morphisms) and provides constructions for the pre-conditions (parameter passing). In this way, both semantics focus in different aspects: while ours explains how parameters are passed, how pre-conditions are built, and where they have to be sought, Stevens' semantics is focused on the $\forall$-$\exists$ alternation when bindings are sought in source and target domains. Nonetheless, the parametric aspects of Stevens' semantics, like the binding or the parameter passing, could be formalized with the approach given here. As we have seen in the previous section, both semantics seem to agree, except for the fact that in [27], cycles are not currently considered.

Regarding the formalization of operational scenarios, the authors in [2, 14] formalize QVT-R transformations by using rewriting logic and Maude. In [12], inspired by the standard compilation of QVT-R into QVT-C, the authors translate QVT-R into triple graph grammars which in this case play the role of QVT-C. In [6], we compiled QVT-R specifications into coloured Petri nets for execution, debug and analysis. Finally, the authors in [11] formalize both QVT-R and QVT-O by using theory of problems, being able to verify whether a QVT-O specification is correct with respect to a QVT-R specification. None of these works consider the check-only scenario, which however is a fundamental aspect of the QVT-R semantics. In this sense, it is difficult to evaluate whether the semantics provided here (for checkonly scenarios) is compatible with that described in these works (for operational scenarios). However, we believe that our algebraic formalization of symbolic tuples is somewhat close to the ones used by triple graph grammars.

There are also works that follow semi-formal or non-formal approaches. For instance, [10] uses OCL to represent the static semantics of relations, and Alloy for the dynamics. This approach is similar to our previous work in [3], where we translated QVT-R into OCL and used a constraint solver for execution and analysis. In [23], the authors translate QVT-R into QVT-O, so that the resulting models can be executed in tools supporting the QVT operational mappings, like SmartQVT [25]. However, the approach only supports the transformation scenario. A similar experiment is proposed in [13], where QVT-R is transformed into the input language of the ATL virtual machine.

Concerning tools, MediniQVT [15], ModelMorf [16], MOMENT-QVT [1] and TROPIC [29] are tools that support QVT-R. However, only ModelMorf supports check-only scenarios. One detail is of interest though. Relations in our approach contain formulas, and therefore 'X=Y' is interpreted as an equality and not as an assignment. As a consequence, the obtained semantics is completely bidirectional, as specifications can be interpreted forwards and backwards without any need to do algebraic manipulation of formulas. On the contrary, current tools do not support this approach, but they are based on assignments to variables, which hinders the use of specifications for bidirectional transformations.

Thus, the work presented here contributes to the understanding of the QVT-R check-only transformations, and can serve as a blueprint for tool implementations, as well as a foundation for model transformation activities.

7. Conclusions

In this paper we have presented a formal, algebraic semantics for QVT-R check-only transformations, hence contributing to the foundations of model-to-model transformations in MDA. Our semantics is based on a formalisation of QVT relations by model pairs and dependencies, taking into consideration the binding of relations to models by means of S-morphisms.

The advantages of our semantics are the following. First, it explains the construction of enabling pre-conditions for the relations and parameter passing. It does so by conceptually constructing a network of symbolic tuples (the relations), from which commuting S-morphisms to the model pairs under evaluation should be found. This relates our approach to the theory of graph constraints [8], especially nested constraints and application conditions, developed by the community of graph transformations [7]. Hence, in the future, we plan to use the theoretical results for nested constraints to check the satisfiability of specifications, obtain equivalent ones or simplify them. For example, we plan to investigate the conditions needed to transform a *where* invocation into a *when* dependency of the invoked relation and obtain an equivalent specification.

Second, the explicit network of constraints can be used to build valid models of the specifications by calculating the co-limit of the network. This is particularly useful for the generation of test cases for transformations, in particular white-box testing, where one selects the relations that are to be exercised (i.e. how many occurrences of each relation we want in our test case). It must be noted that the model obtained through the co-limit construction may not satisfy all meta-model constraints, and hence should be extended with further elements. However, such model could be fed into a constraint solver like UMLtoCSP [4] to complete it with the needed elements using techniques similar to those in [3].

Third, our semantics generalizes several aspects of the standard and current practice. In particular, the formulae attached to our symbolic tuples are inherently bidirectional and do not need to be manipulated when the tuples are interpreted forwards or backward. Our formalization also generalizes the parameter passing mechanism of QVT-R, as it permits invoking a relation with different number of parameters without the need of duplicating the relation, as it should be done in the standard. Moreover, our parameters may include both objects and links, being symbolic tuples in their own right, and may even include formulae constraining those objects as well. Further exploration of this feature is left for future work.

Finally, our formalization enables the verification of transformations using algebraic techniques developed in the graph transformation community. For example, one could use symbolic tuples to express verification properties that should be satisfied (or not) by the transformation specification. For instance, in the UML to RDBMS example, one may like to check that the transformation always stores every attribute of a child class in the same table as the parent class. This property may not be stated explicitly in the transformation specification, but we may want to verify it. For this purpose we could use the refutation techniques developed in [21] to check whether the transformation verifies this property.

In the future, we plan to extend our formalization as follows. First, in this work we have modelled the *when* and *where* dependencies as sets, whereas in the standard the dependencies can be embedded in arbitrary OCL expressions. Also, the standard foresees relations with more than two domains to attack problems concerned with multi-modelling. We also plan to investigate analysis techniques for QVT-R specifications based on this formalization. Finally, we are investigating the right kind of restrictions in Definition 6 to avoid paradoxes but allowing non-problematic circularities. We will tackle these issues in a future contribution.

Acknowledgements. We would like to thank the referees for their detailed comments that helped us improving previous versions of this paper. We are particularly grateful to "referee 1" for providing us with the paradoxical evaluation example shown in Section 4.1. This work has been supported by the Spanish Ministry of Science and Innovation with projects METEORIC (TIN2008-02081) and *Go Lite* (TIN2011-24139), and by the R&D program of the Community of Madrid with project "e-Madrid" (S2009/TIC-1650).

References

[1] A. Boronat. *MOMENT: A formal framework for MOdel manageMENT*. PhD thesis, Universitat Politècnica de Valencia, 2007. See also `http://moment.dsic.upv.es/content/view/34/75/`. Last accessed: November 2010.

[2] A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *FASE'06*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.

[3] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[4] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming. In *ASE'07*, pages 547–548. ACM, 2007.

[5] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.

[6] J. de Lara and E. Guerra. Formal support for QVT-Relations with coloured Petri nets. In *MoDELS'09*, volume 5795 of *LNCS*, pages 256–270. Springer, 2009.

[7] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae*, 74(1):135–166, 2006.

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

[9] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In *ICGT'04*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004.

[10] M. García. Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In *MDSD today*, pages 21–30. Shaker Verlag, 2008.

[11] R. S. Giandini, C. Pons, and G. Pérez. A two-level formal semantics for the QVT language. In *CIbSE'09*, pages 73–86, 2009.

[12] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: Implementing Query/View/Transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[13] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *SAC'06*, pages 1188–1195, 2006. See also `http://www.eclipse.org/m2m/atl/usecases/QVT2ATLVM/`. Last accessed: November 2010.

[14] F. J. Lucas and J. A. T. Álvarez. Model transformations powered by rewriting logic. In *CAiSE Forum*, volume 344 of *CEUR Proc.*, pages 41–44, 2008.

[15] MediniQVT. `http://projects.ikv.de/qvt/`. Last accessed: November 2010.

[16] ModelMorf. `http://www.tcs-trddc.com/trddc_website/scripts/project_detail.php?lab=SWRD&project_id=44`. Last accessed: November 2010.

[17] OMG. UML homepage. `http://www.uml.org/`. Last accessed: November 2010.

[18] F. Orejas. Attributed graph constraints. In *ICGT'08*, volume 5214 of *LNCS*, pages 274–288. Springer, 2008.

[19] F. Orejas. Symbolic attributed graphs for attributed graph transformation. In *Proc. International Colloquium on Graph and Model Transformation*, Electronic Communications of the EASST, pages 26–55, 2010. Available from `www.lsi.upc.edu/~orejas/papers/Gramot1.pdf`.

[20] F. Orejas. Symbolic graphs for attributed graph constraints. *J. Symb. Comput.*, 46(3):294–315, 2011.

[21] F. Orejas and M. Wirsing. On the specification and verification of model transformations. In *Semantics and Algebraic Specification*, volume 5700 of *LNCS*, pages 140–161. Springer, 2009.

[22] QVT. `http://www.omg.org/spec/QVT/1.1/PDF/`. Last accessed: September 2011, 2005.

[23] R. Romeikat, S. Roser, P. Müllender, and B. Bauer. Translation of QVT relations into QVT operational mappings. In *ICMT'08*, volume 5063 of *LNCS*, pages 137–151, 2008.

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

[25] SmartQVT. France Telecom R&D. `http://sourceforge.net/projects/smartqvt/`. Last accessed: November 2010.

[26] P. Stevens. A simple game-theoretic approach to checkonly QVT relations. In *ICMT'09*, volume 5563 of *LNCS*, pages 165–180. Springer, 2009.

[27] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

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

[29] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger. A Petri net based debugging environment for QVT relations. In *ASE'09*, pages 3–14. IEEE Computer Society, 2009. See also `http://www.modeltransformation.net/`. Last accessed: November 2010.

Appendix A: Proofs

Here we provide the proofs of the characterization of pushouts and pullbacks given in Proposition 1. Actually, these proofs are adaptations of those of [19] for symbolic graphs to our symbolic tuples.

Proposition 2. (Pushout of Symbolic Tuples (adapted from [19]))
Given the pushouts (1) and (2) in symbolic graphs, then square (3) is a pushout in **SymbTuple**$_A$ (see Figure 21), where $\beta_P = d(\beta_C) \wedge e(\beta_D)$ and $x = (x_S, x_T)$ for $x = \{d, e, f, g\}$.

$$\begin{tikzcd}
I_S \arrow[r, "g_S"] \arrow[d, "f_S"'] \arrow[dr, phantom, "(1)"] & D_S \arrow[d, "e_S"] & I_T \arrow[r, "g_T"] \arrow[d, "f_T"'] \arrow[dr, phantom, "(2)"] & D_T \arrow[d, "e_T"] & \langle I_S, I_T, \beta_I \rangle \arrow[r, "g"] \arrow[d, "f"'] \arrow[dr, phantom, "(3)"] & \langle D_S, D_T, \beta_D \rangle \arrow[d, "e"] \\
C_S \arrow[r, "d_S"] & P_S & C_T \arrow[r, "d_T"] & P_T & \langle C_S, C_T, \beta_C \rangle \arrow[r, "d"] & \langle P_S, P_T, \beta_P \rangle
\end{tikzcd}$$

Figure 21. Pushouts in **SymbTuple**$_A$.

Proof: The first step is to show that if (1) and (2) are pushouts, then $e = (e_S, e_T)$ and $d = (d_S, d_T)$ are actually S-morphisms. This is straightforward since e_S, e_T, d_S and d_T are symbolic graph morphisms, and $\gamma_P = [d(\beta_C) \wedge e(\beta_D)] \wedge [d_S(\alpha_C^S) \wedge e_S(\alpha_D^S)] \wedge [d_T(\alpha_C^T) \wedge e_T(\alpha_D^T)] \Rightarrow d(\gamma_C) = d(\beta_C \wedge \alpha_C^S \wedge \alpha_C^T)$ and $\gamma_P = [d(\beta_C) \wedge e(\beta_D)] \wedge [d_S(\alpha_C^S) \wedge e_S(\alpha_D^S)] \wedge [d_T(\alpha_C^T) \wedge e_T(\alpha_D^T)] \Rightarrow e(\gamma_D) = e(\beta_D \wedge \alpha_D^S \wedge \alpha_D^T)$ are tautologies.

The second step is to show that if $e'\colon \langle D_S, D_T, \beta_D \rangle \to \langle P'_S, P'_T, \beta'_P \rangle$ and $d'\colon \langle C_S, C_T, \beta_C \rangle \to \langle P'_S, P'_T, \beta'_P \rangle$ are two S-morphisms with $e' \circ g = d' \circ f$, then there is a unique S-morphism $u\colon \langle P_S, P_T, \beta_P \rangle \to \langle P'_S, P'_T, \beta'_P \rangle$ s.t. $e' = u \circ e$ and $d' = u \circ d$, see Figure 22. First we proof existence, and then uniqueness.

$$\begin{tikzcd}
\langle I_S, I_T, \beta_I \rangle \arrow[r, "g"] \arrow[d, "f"'] \arrow[dr, phantom, "(3)"] & \langle D_S, D_T, \beta_D \rangle \arrow[d, "e"] \arrow[ddr, bend left, "e'"] & \\
\langle C_S, C_T, \beta_C \rangle \arrow[r, "d"] \arrow[drr, bend right, "d'"'] & \langle P_S, P_T, \beta_P \rangle \arrow[dr, dotted, "u"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & {} \\
 & {} & \langle P'_S, P'_T, \beta'_P \rangle
\end{tikzcd}$$

Figure 22. Condition for Pushouts in **SymbTuple**$_A$.

Existence. Since (1) and (2) are pushouts, then there are symbolic graph morphisms $u_S\colon P_S \to P'_S$ and $u_T\colon P_T \to P'_T$. But $u = (u_S, u_T)\colon \langle P_S, P_T, \beta_P \rangle \to \langle P'_S, P'_T, \beta'_P \rangle$ is an S-morphism, because if $A \models \gamma'_P \Rightarrow e'(\gamma_D)$ and $A \models \gamma'_P \Rightarrow d'(\gamma_C)$, then $A \models \gamma'_P \Rightarrow (e'(\gamma_D) \wedge d'(\gamma_C))$. But we know that $e'(\gamma_D) \wedge d'(\gamma_C) = (u \circ e)(\gamma_D) \wedge (u \circ d)(\gamma_C) = u(e(\gamma_D) \wedge d(\gamma_C))$. Hence $A \models \gamma'_P \Rightarrow u(\gamma_P)$.
Uniqueness. If $u'\colon \langle P_S, P_T, \beta_P \rangle \to \langle P'_S, P'_T, \beta'_P \rangle$ is an S-morphism such that $u' \circ e = e'$ and $u' \circ d = d'$, then by the universal pushout property in symbolic graphs, we have that $u' = u$. □

Next, we prove the characterization of pullbacks. Similar to [19], we require the variables in $D(I_S) \uplus D(I_T)$ to be disjoint with those of $D(D_S) \uplus D(D_T)$ and $D(C_S) \uplus D(C_T)$ (see Figure 23), as it notably simplifies the proof. However, this is not really a restriction because, if we are given a symbolic tuple $\langle C_S, C_T, \beta_C \rangle$ and rename its variables in $D(C_S) \uplus D(C_T)$, we obtain an isomorphic symbolic tuple

$\langle C'_S, C'_T, \beta'_C \rangle$. Hence, in order to calculate the pullback of two symbolic tuples, we can rename their variables (if needed) and then use the next proposition.

Proposition 3. (Pullbacks of Symbolic Tuples (adapted from [19]))
Given the pullbacks (1) and (2) in symbolic graphs, then square (3), where $\beta_I = (\exists D(D_S)\, \exists D(D_T)(\beta_D \wedge eq(g))) \vee (\exists D(C_S)\, \exists D(C_T)(\beta_C \wedge eq(f)))$ is a pullback in **SymbTuple**$_A$ (see Figure 23), with $x = (x_S, x_T)$ for $x = \{d, e, f, g\}$, $\exists D(D_S)$ denoting an existential quantification over the variables in D_S (and similar for D_T, C_S and C_T), and $eq(g)$ denoting the conjunction of equalities $\bigwedge_{x \in D(I_S) \uplus D(I_T)} x = g(x)$[1] (and similar for $eq(f)$).

$$\begin{tikzcd}
I_S \arrow[r, "g_S"] \arrow[d, "f_S"'] \arrow[dr, phantom, "(1)"] & D_S \arrow[d, "e_S"] \\
C_S \arrow[r, "d_S"] & P_S
\end{tikzcd}
\qquad
\begin{tikzcd}
I_T \arrow[r, "g_T"] \arrow[d, "f_T"'] \arrow[dr, phantom, "(2)"] & D_T \arrow[d, "e_T"] \\
C_T \arrow[r, "d_T"] & P_T
\end{tikzcd}
\qquad
\begin{tikzcd}
\langle I_S, I_T, \beta_I \rangle \arrow[r, "g"] \arrow[d, "f"'] \arrow[dr, phantom, "(3)"] & \langle D_S, D_T, \beta_D \rangle \arrow[d, "e"] \\
\langle C_S, C_T, \beta_C \rangle \arrow[r, "d"] & \langle P_S, P_T, \beta_P \rangle
\end{tikzcd}$$

Figure 23. Pullbacks in **SymbTuple**$_A$.

Proof: The first step is to show that if (1) and (2) are pullbacks, then g and f are actually S-morphisms. For this purpose, we need to prove that $A \models \gamma_C \Rightarrow f(\gamma_I)$ and $A \models \gamma_D \Rightarrow g(\gamma_I)$.

Lets assume that $\sigma_D \colon D(D_S) \uplus D(D_T) \to A$ is a valuation, that is, an assignment of values to the variables in $D(D_S) \uplus D(D_T)$ such that $A \models \sigma_D(\gamma_D)$. We have to show that $A \models \sigma_D(g(\gamma_I))$, or equivalently that $A \models \sigma_I(\gamma_I)$ for $\sigma_I = \sigma_D \circ g$. Since

$$\begin{aligned}
\gamma_I = {} & [(\exists D(D_S)\, \exists D(D_T)(\beta_D \wedge eq(g))) \vee (\exists D(C_S)\, \exists D(C_T)(\beta_C \wedge eq(f)))] \\
& \wedge [(\exists D(D_S)\, (\alpha^S_D \wedge eq(g_S))) \vee (\exists D(C_S)\, (\alpha^S_C \wedge eq(f_S)))] \\
& \wedge [(\exists D(D_T)\, (\alpha^T_D \wedge eq(g_T))) \vee (\exists D(C_T)\, (\alpha^T_C \wedge eq(f_T)))]
\end{aligned} \tag{1}$$

then it is enough to prove that

$$A \models \sigma_I((\exists D(D_S)\, \exists D(D_T)(\beta_D \wedge eq(g))) \wedge (\exists D(D_S)\, (\alpha^S_D \wedge eq(g_S))) \wedge (\exists D(D_T)\, (\alpha^T_D \wedge eq(g_T)))) \tag{2}$$

But this is equivalent to show that there are substitutions $\sigma'_D \colon D_S \uplus D_T \to A$, $\sigma'^S_D \colon D_S \to A$, $\sigma'^T_D \colon D_T \to A$ for the variables in $D_S \uplus D_T$ (existentially quantified) such that $A \models \sigma'_D(\sigma_I(\beta_D \wedge eq(g))) \wedge \sigma'^S_D(\sigma_I(\alpha^S_D \wedge eq(g_S))) \wedge \sigma'^T_D(\sigma_I(\alpha^T_D \wedge eq(g_T)))$. Since $\beta_D, \alpha^S_D, \alpha^T_D$ do not involve any variable from

[1]g(x) means the variable in D_S or D_T that is the image of x through $g = (g_S, g_T)$.

$D(I_S) \uplus D(I_T)$, we have that

$$\sigma'_D(\sigma_I(\beta_D \wedge eq(g))) \wedge \sigma'^S_D(\sigma_I(\alpha^S_D \wedge eq(g_S))) \wedge \sigma'^T_D(\sigma_I(\alpha^T_D \wedge eq(g_T))) =$$
$$[\sigma'_D(\beta_D) \wedge \sigma'^S_D(\alpha^S_D) \wedge \sigma'^T_D(\alpha^T_D)] \wedge [\sigma'_D(\sigma_I(eq(g))) \wedge \sigma'^S_D(\sigma_I(eq(g_S))) \wedge \sigma'^T_D(\sigma_I(eq(g_T)))]$$

Now, if we take $\sigma'_D = \sigma_D, \sigma'^S_D = \sigma_D|_S$ and $\sigma'^T_D = \sigma_D|_T$, we have that $A \models \sigma_D(\gamma_D) = \sigma_D(\beta_D \wedge \alpha^S_D \wedge \alpha^T_D)$ by assumption. Moreover, if $x \in D(I_S) \uplus D(I_T)$, then by definition $\sigma_I(x) = \sigma_D \circ g(x) = \sigma_D(g(x))$, which means that $A \models \sigma_D(\sigma_I(x = g(x)))$ for every $x \in D(I_S) \uplus D(I_T)$ and therefore $A \models \sigma_D(\sigma_I(eq(g) \wedge eq(g_S) \wedge eq(g_T)))$. Hence, $A \models \gamma_D \Rightarrow g(\gamma_I)$. By the same reasoning, f is also an S-morphism.

The second step is to show that if $g' \colon \langle I'_S, I'_T, \beta'_I \rangle \to \langle D_S, D_T, \beta_D \rangle$ and $f' \colon \langle I'_S, I'_T, \beta'_I \rangle \to \langle C_S, C_T, \beta_C \rangle$ are two S-morphisms with $e \circ g' = d \circ f'$, then there is a unique S-morphism $u \colon \langle I'_S, I'_T, \beta'_I \rangle \to \langle I_S, I_T, \beta_I \rangle$ s.t. $g' = g \circ u$ and $f' = f \circ u$, see Figure 24. First we proof existence, and then uniqueness.

$$\begin{tikzcd}
\langle I'_S, I'_T, \beta'_I \rangle \arrow[rrd, bend left, "g'"] \arrow[rd, dotted, "u"] \arrow[rdd, bend right, "f'"'] & & \\
& \langle I_S, I_T, \beta_I \rangle \arrow[r, "g"] \arrow[d, "f"'] \arrow[rd, phantom, "(3)"] & \langle D_S, D_T, \beta_D \rangle \arrow[d, "e"] \\
& \langle C_S, C_T, \beta_C \rangle \arrow[r, "d"] & \langle P_S, P_T, \beta_P \rangle
\end{tikzcd}$$

Figure 24. Condition for Pullbacks in **SymbTuple**$_A$.

Existence. First, as (1) is a pullback in symbolic graphs, we have that there is a unique symbolic tuple morphism $u_S \colon I'_S \to I_S$ such that $g_S \circ u_S = g'_S$ and $f_S \circ u_S = f'_S$. Similarly, as (2) is a pullback in symbolic graphs, there is a unique $u_T \colon I'_T \to I_T$ such that $g_T \circ u_T = g'_T$ and $f_T \circ u_T = f'_T$. We first prove that $u = (u_S, u_T) \colon \langle I'_S, I'_T, \beta'_I \rangle \to \langle I_S, I_T, \beta_I \rangle$ is actually an S-morphism. If $\sigma_I \colon D(I_S) \uplus D(I_T) \to A$ is a valuation for the variables in $\langle I_S, I_T, \beta_I \rangle$ such that $A \models \sigma_I(\gamma_I)$, we have to prove that $A \models \sigma_I(u(\gamma'_I))$. Given the definition of γ_I in equation 1, assume $A \models \sigma_I(\gamma_I)$ holds. Then, transforming equation 1 into disjunctive form, one of the terms in the disjuntion has to satisfy the valuation σ_I. Without loss of generality, lets assume that the term satisfying the valuation is that of equation 2, which means that there are valuations $\sigma_D \colon D(D_S) \uplus D(D_T) \to A$, $\sigma^S_D \colon D(D_S) \to A$, $\sigma^T_D \colon D(D_T) \to A$ such that $A \models \sigma_D(\sigma_I(\beta_D \wedge eq(g))) \wedge \sigma^S_D(\sigma_I(\alpha^S_D \wedge eq(g_S))) \wedge \sigma^T_D(\sigma_I(\alpha^T_D \wedge eq(g_T)))$ and hence $A \models \sigma_D(\beta_D) \wedge \sigma^S_D(\alpha^S_D) \wedge \sigma^T_D(\alpha^T_D) \wedge \sigma_D(\sigma_I(eq(g))) \wedge \sigma^S_D(\sigma_I(eq(g_S))) \wedge \sigma^T_D(\sigma_I(eq(g_T)))$. Since $g' \colon \langle I'_S, I'_T, \beta'_I \rangle \to \langle D_S, D_T, \beta_D \rangle$ is an S-morphism and $A \models \sigma_D(\gamma_D)$, we have that $A \models \sigma_D(g'(\gamma'_I))$. But as $g' = g \circ u$, we have that $A \models \sigma_D(g \circ u(\gamma'_I))$, or equivalently $A \models \sigma'_D(u(\gamma'_I))$, where $\sigma'_D = \sigma_D \circ g$. On the other hand, since $A \models \sigma_D(\sigma_I(eq(g)))$ for every variable $x \in D(I_S) \uplus D(I_T)$, we have $\sigma_D(\sigma_I(x)) = \sigma_D(\sigma_I(g(x)))$. Since σ_D and σ_I are defined over disjoint sets, we have that $\sigma_D(\sigma_I(x)) = \sigma_I(\sigma_D(g(x)))$, and since x is not in $D(D_S) \uplus D(D_T)$, $g(x)$ is not in $D(I_S) \uplus D(I_T)$, and $\sigma'_D = \sigma_D \circ g$, we have that $\sigma_I(x) = \sigma'_D(x)$. Therefore $A \models \sigma_I(u(\gamma'_I))$.

Uniqueness. Uniqueness is derived from the universal pullback property in symbolic graphs. Hence, if $u' = (u'_S, u'_T) \colon \langle I'_S, I'_T, \beta'_I \rangle \to \langle I_S, I_T, \beta_I \rangle$ is an S-morphism satisfying $g \circ u' = g'$ and $f \circ u' = f'$, by the universal pullback property in symbolic graphs (so that u_S and u_T are unique), we have that $u = u'$. $\square$

Proof of Proposition 1: Direct consequence of the Proofs of Propositions 2 and 3.