Noname manuscript No.
(will be inserted by the editor)

Comparing Relational Model Transformation Technologies – Implementing QVT with Triple Graph Grammars

Joel Greenyer[1]*, **Ekkart Kindler**[2]

[1] Software Engineering Group, University of Paderborn, Warburger Str. 100, D-33098 Paderborn, Germany
e-mail: jgreen@upb.de

[2] Informatics and Mathematical Modelling, Technical University of Denmark (DTU), DK-2800 Kgs. Lyngby, Denmark
e-mail: eki@imm.dtu.dk

The date of receipt and acceptance will be inserted by the editor

Abstract The Model Driven Architecture (MDA) is an approach to develop software based on different models. There are separate models for the business logic and for platform specific details. Moreover, code can be generated automatically from these models. This makes transformations a core technology for MDA and for model-based software engineering approaches in general. QVT (Query/View/Transformation) is the transformation technology recently proposed for this purpose by the OMG.

TGGs (Triple Graph Grammars) are another transformation technology proposed in the mid-nineties, used for example in the FUJABA CASE tool. In contrast to many other transformation technologies, both QVT and TGGs declaratively define the relation between two models. With this definition, a transformation engine can execute a transformation in either direction and, based on the same definition, can also propagate changes from one model to the other.

In this paper, we compare the concepts of the declarative languages of QVT and TGGs. It turns out that TGGs and declarative QVT have many concepts in common. In fact, QVT-Core can be mapped to TGGs. We show that QVT-Core can be implemented by transforming QVT-Core mappings to TGG rules, which can then be executed by a TGG transformation engine that performs the actual QVT-transformation. Furthermore, we discuss an approach for mapping QVT-Relations to TGGs.

Based on the semantics of TGGs, we clarify semantic gaps that we identified in the declarative languages of QVT and, furthermore, we show how TGGs can benefit from the concepts of QVT.

* supported by the International Graduate School Dynamic Intelligent Systems.

Key words MDA, model-based software engineering, model transformation, Query/View/Transformation (QVT), Triple Graph Grammar (TGG).

1 Introduction

In the recent years, several approaches to *model-based software engineering* have been proposed. One of the most prominent approaches is the *Model Driven Architecture (MDA)* of the OMG [**?**]. The main idea of all these approaches is that software should no longer be programmed, but developed by a stepwise refinement and extension of models. In the MDA, the focus is on separating the models for the business logic from the models for platform and implementation specific details. In the end, the code can be generated from these models. Because of this, technologies for transforming, integrating, and synchronizing models are at the core of model-based software engineering.

Today, there are many different technologies for transforming one model into another. Most of these technologies are defined in a more or less operational way; i. e. they define instructions how the elements from the source model must be transformed into elements of the target model. This implies that forward and backward transformations between two models are defined more or less independently of each other. By contrast, *QVT* (*Query/View/Transformation*) [**?**] and *TGGs* (*Triple Graph Grammars*) [**?**] allow us to declaratively define the relation between two or more models by rules. Such a declarative definition of a relation can be used by a transformation engine in different ways, which we call *application scenarios*: Firstly, there are the *forward* and *backward transformations* of one model into another. Secondly, once we have transformed one model into another, the engine can keep track of changes in either model and propagate those changes to the other model and change it accordingly. This is

called *model synchronization* and is one of the most crucial application scenarios in round-trip engineering. Since QVT and TGGs use one definition of the relation between two classes of models for all these application scenarios, inconsistencies among the different scenarios are avoided. Note that declarative QVT and TGG transformations are not always deterministic, but this problem is not in the focus of this paper. For a detailed discussion of bidirectional model transformations, we refer to Stevens [**?**]. For details on algorithms for realizing the model synchronization scenarios we refer to Giese and Wagner [**?**,**?**].

QVT is the transformation technology recently proposed by the OMG for the MDA. QVT has different parts: There are declarative and operational languages. In this paper, we focus on the declarative languages QVT-Relations and QVT-Core. TGGs were introduced in the mid-nineties, and are now used in the FUJABA Tool Suite, which is a CASE tool supporting round-trip engineering. In Fujaba, TGGs are for example used for transforming back and forth between UML diagrams and Java code [**?**]. Other implementations of TGGs exist in the MOFLON tool set [**?**], the AToM3 meta-modeling tool [**?**], and PROGRES [**?**,**?**]. To be able to transform EMF (Eclipse Modeling Framework) models in our Eclipse-based projects and for this comparison of QVT and TGGs, we have implemented another TGG engine.

In addition to being declarative, QVT and TGGs have many concepts in common and—upon closer investigation—have striking similarities. In this paper, we investigate these similarities for several reasons. Firstly, the common concepts of QVT-Core and TGGs represent the essential concepts of a declarative approach toward specifying the relationship between two classes of models. Secondly, we want to analyze the similarities of QVT and TGGs in order to extract the essential differences of the two technologies. Our analysis shows that QVT-Core can be mapped to the concepts of TGGs so that QVT-Core can be implemented by an engine for executing TGG transformations. This mapping was worked out in a master thesis [**?**] and is discussed in this paper. This mapping supports the most important constructs of QVT-Core with some limitations on the expressive power of attribute constraints (see Sect. 2.3 for details). Here, in addition to our prior conference contribution [**?**], we discuss the QVT-Core-to-TGG mapping in more detail and we also discuss an approach for mapping QVT-Relations to TGGs directly in order to clarify the semantics of QVT-Relations and to foster a common understanding of relational transformation languages. Finally, the differences between QVT and TGGs are analyzed and we discuss how both technologies can benefit from the concepts provided by the other technology. This will help to improve both transformation technologies. Foremost, we suggest to be more explicit about the binding of rules to the model elements in the interpretation of QVT. Moreover, we suggest to introduce reusable nodes in TGGs (see Sect. 5).

This paper is structured as follows: Section 2 introduces the main concepts of QVT and TGGs with the help of an example. Section 3 identifies the similar concepts and shows how QVT-Core can be mapped to TGGs, which provides an implementation of QVT based on a TGG engine. We have implemented this mapping through a TGG transformation and document a number of representative TGG rules of this mapping. Section 4 discusses an approach for mapping QVT-Relations directly to TGGs. Section 5 discusses some differences between QVT and TGGs and their implications for the interpretation of QVT and for extensions of TGGs.

2 TGG and QVT Transformations by Example

As pointed out in the introduction, QVT and TGGs have in common that they are both relational. In this section, we illustrate the general idea of relational transformation approaches, and we introduce QVT and TGGs by the help of an example. The example is a simplified version of a transformation taken from the ComponentTools project [**?**]. ComponentTools is a tool for engineers to design, analyze, and verify systems by using different mathematical models and formal methods. One core feature of ComponentTools is the possibility to define transformations between formalisms. Our example shows how to transforms a component-based material flow system contained inside a project plan into a Petri net, which defines its dynamic behavior.

2.1 A relational definition of a transformation

Here we discuss a simple class of systems where a production line is built from components such as tracks, switches, and stoppers. The structure is defined in a project plan, which we call *project* for short. The left-hand side of Fig. 1 shows a very simple project, which consists of two track components that are connected to each other, indicated by an arrow from the out-port of the first track to the in-port of the other. The right-hand side of Fig. 1 shows a Petri net that models the behavior of this production line. This Petri net can then be used for simulating, visualizing, and verifying the behavior of that production line.

The ellipses and the dashed arrows in Fig. 1 indicate how the different parts of the project correspond to the different parts of the Petri net. A track component corresponds to a Petri net part with a place (circle), a transition (square), and an arc (arrow) in-between. More precisely, the place corresponds to the in-port of the project, and the transition corresponds to the out-port. The connection of the project corresponds to an arc of the Petri net, which connects the respective transition and place.

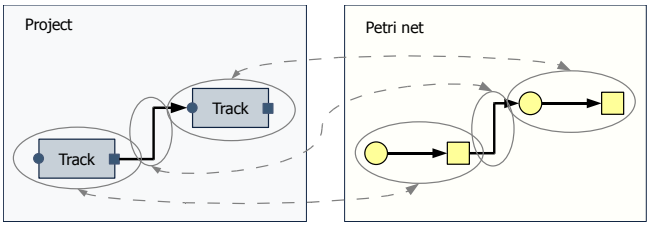

Fig. 1 A project and the corresponding Petri net

The idea of this transformation can be captured in two rules, which are shown in Fig. 2 and 3. Let us have a look at Fig. 2 first. The rule consists of two parts: the

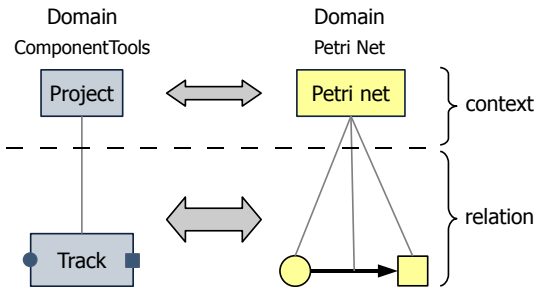

Fig. 2 Relation between a track and its Petri net

context (top) and the *relation* (bottom). Its meaning is that when we have a project and a corresponding Petri net already, we can insert a track in the project and the corresponding place, transition, and arc in the Petri net. Since the project and the corresponding Petri net need to exist in order to apply the rule, they are called context. Likewise, the rule from Fig. 3 says that if we have an out-port and an in-port with their corresponding transition and place, we can insert a connection to the project along with the corresponding arc in the Petri net. Both rules

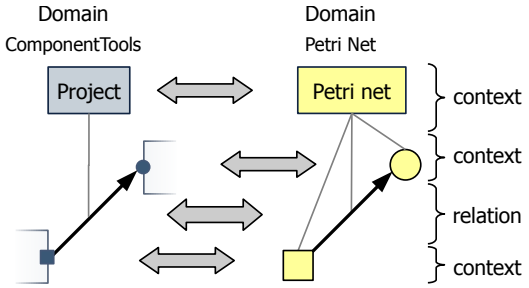

Fig. 3 Relation between a connection and the Petri net (arc)

together express the relation between the elements of a project and the elements in the Petri net. Basically, they say how a project and a Petri net can be built up by applying the rules on the project domain and the Petri net domain in parallel. Note that these rules do not have a direction, and we have not yet discussed how to use them for transforming a project into a corresponding Petri net. The rules just define when a pair of models is in relation by creating the two models in parallel.

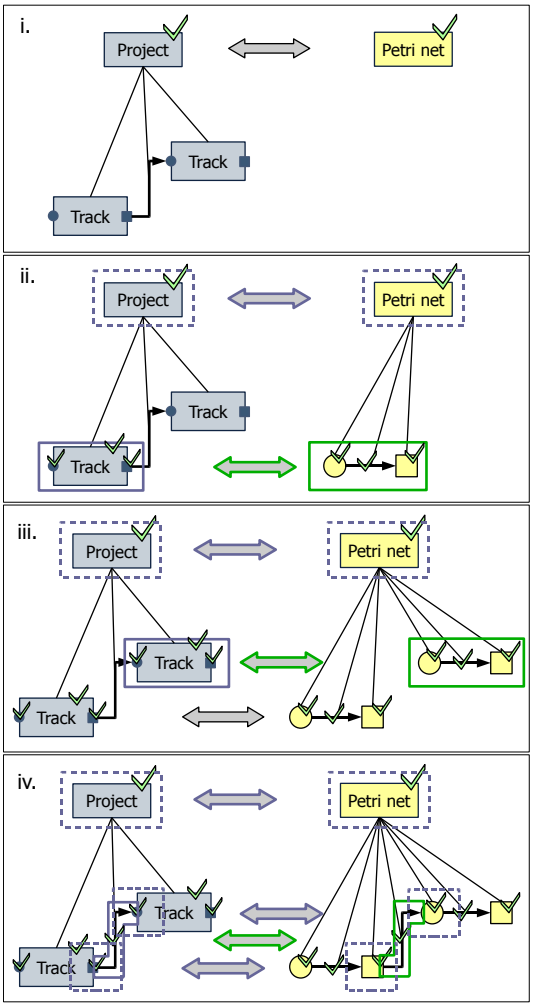

Fig. 4 Example rules: A forward transformation

Next, we will show how these rules can be used for transforming a project into a Petri net, so that the project and the Petri net are related as defined by the rules above. The top (i.) left-hand side of Fig. 4 shows our example project again, which we are going to transform into a Petri net. In order to transform the project, we have added the Petri net root element to which we want to add all the Petri net elements. Since the project and the Petri net are the start situation for applying rules (in graph grammars often called a start graph or axiom, or a start rule in QVT), we bind the project and the Petri net to this fixed start situation. Graphically, this binding is indicated by the two ticks. Now, we start with matching the rules to the project. This is done in row (ii.), where we apply the rule from Fig. 2. This rule fits for the following reason: The context of the rule is matched to the already bound elements (the project and the Petri net), indicated by the dashed boxes. The relation part of the rule in the

project is matched to the previously unbound track at the bottom (indicated by the outlined boxes), and the missing elements of the rule for matching the rule on the Petri net side are created (indicated by the outlined boxes). These newly created elements will now also be indicated as bound elements. Next, we can match the other track of the project in the same way as shown in row (iii.) and bind the respective elements. At last, we can match the rule for the connection as shown in row (iv): The context of that rule is matched to the elements bound by the earlier rule applications, the connection is bound, and the corresponding arc in the Petri net is created and bound. Now all elements of the project are bound and the corresponding Petri net is created. Its elements were created alongside binding the elements of the project to the rules. Note that every element of the project was exactly bound once when it was matched with the relation side of a rule; later it could be used only in the context of a matching rule.

The above example outlines the basic idea of applying relational rules for performing transformations. The essence is that, in the rules, we distinguish between the *context* of a rule and the *relation*: the context of a rule is always matched to already bound elements. The elements of the relation of a rule are matched to unbound elements of the source of the transformation, or are bound to the newly created elements of the target of the transformation. In the further course of this paper, we will see that an essential difference between QVT and TGGs is that the relation part of declarative QVT rules may also bind to already bound nodes.

Common to relational rules is that they explicitly refer to *domains*, which are the "columns" of the rule. In our example, the two domains *project* and *Petri net* have two completely different meta-models. But, it is also possible that both domains have the same meta-model; this could for example be a transformation from UML to UML. In principle, this idea is not even restricted to two domains. There can be any number of domains. In this paper, however, we confine ourselves to two domains.

The rules do not define a transformation direction. The direction of a transformation is part of what we call *application scenario*: it defines which domain is the source and which domain is the target of the transformation.

2.2 Meta-models for the example

In the subsequent sections, we discuss some more details of TGGs and QVT. In order to do that, we need to introduce the meta-models for our two domains: ComponentTools projects and Petri nets. These meta-models are shown in Fig. 5.

The meta-model for the ComponentTools project is called ctools. It shows that a project consists of *components* and *connections*. For now, the only concrete components are *tracks*. A component can have *ports*, where each port has a type, which is "in" or "out". And a connection can connect two ports. Note that a meta-model would include many more restrictions to exactly capture the syntactically correct projects, which we did not include here.

The meta-model for Petri nets is pnet. It shows that a Petri net consists of places, transitions, and arcs. For the purpose of this paper, we made a slight simplification: arcs actually do not have a direction, they just have a transition end and a place end.

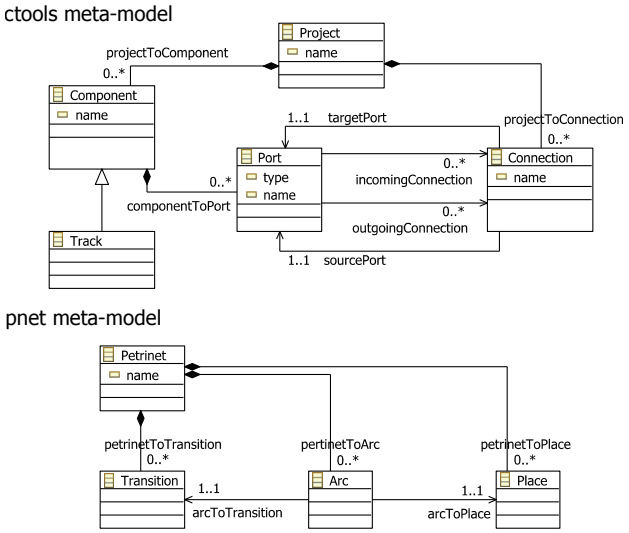

Fig. 5 The ctools and pnet example meta-models

2.3 TGG Rules

Figure 6 shows the TGG version of the rule from Fig. 2, which defines the relation between the track of a project and its corresponding Petri net. There are two main differences: First of all, the rule is not shown in the concrete graphical syntax for projects resp. Petri nets anymore; rather it is shown in the abstract syntax based on the meta-models. This, however, is a presentation issue only. Secondly, the correspondences between the elements of the different models are made more specific now. The elements in the middle part, which are called *correspondence nodes*, have explicit references to the elements that are related by the rule. For example, we can now exactly see that the out-port of a track component is related to the transition of the Petri net. And the track component itself is related to all elements of the Petri net.

Technically, a TGG rule can be considered to be a graph grammar rule. If we have a graph or a model that contains the left-hand side of the graph grammar rule, we can replace that part by the right-hand side of the graph grammar rule. In our rules, the right-hand side

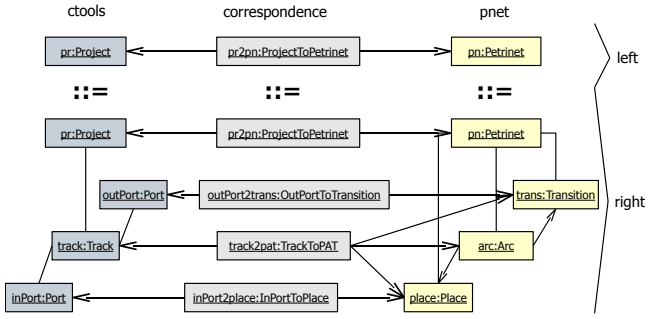

Fig. 6 The TGG rule for tracks

always contains all elements of the left-hand side; so the application of the rule, always extends the existing graph or model. And, it will extend both domains in parallel, which is exactly the meaning of a relational rule as discussed in Fig. 4. If we start from a situation as shown in Fig. 7 by applying the graph grammar rules, we will exactly get the pairs of models that correspond to each other. In graph grammars, this starting point is called an *axiom*. When existing model elements are bound to an axiom, such as shown in Fig. 4 (i), we call that the *start context*.

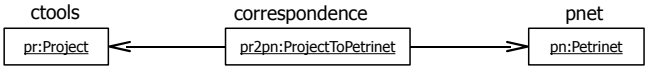

Fig. 7 The start context for the example transformation

This graph grammar semantics exactly captures the relation between two sets of models, by generating them in parallel from some axiom or start context. The only difference to classical graph grammars is that, in TGGs, there are two distinguished domains and a correspondence domain, and every element of a rule is associated with exactly one of these domains. Since there are three domains, they are called Triple Graph Grammars. As mentioned earlier, this can be extended to more than two domains: Multi-Graph Grammars as proposed in [**?**], which however are beyond the scope of this paper.

As explained above, the graph grammar semantics of TGG rules is defined by generating the elements of all domains in parallel. In order to perform a transformation from one domain to the other, we can use the same idea as in Fig. 4 again. Before explaining that, we introduce a slightly less verbose notation for TGG rules: As mentioned already, all our TGG rules contain all elements from the left-hand side also on their right-hand side. Therefore, the elements of the left-hand side occur twice. We avoid that and draw all elements only once, but the elements occurring on the right-hand side only, will be drawn in green and marked with '++', since they are the *produced* elements. All other elements (not labelled with '++') correspond to what we called context in our general schema (see Fig. 2); elements labelled with '++' correspond to the relation elements.

Figure 8 shows a TGG rule in this more compact notation. In addition, it introduces a new concept, which is called *attribute constraints* (in rounded boxes). In this example, the attribute constraints formalize the restriction that the in- and out-ports are actually in- and out-ports. See in Fig. 5 that the ports have a "type" attribute of type string. Our current TGG implementation supports simple attribute constraints that allow to assign or check string, integer, or boolean literal values or to enforce or check an attribute to be equal to another object's attribute of the same type (see Section 3.2 for examples).

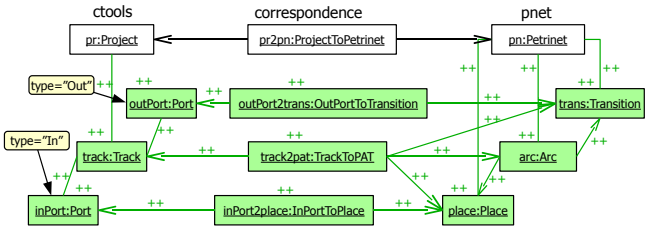

Fig. 8 The TGG rule in its compact notation

Figure 9 illustrates the different application scenarios of a TGG. We start with explaining the first scenario (i), which is called forward transformation. The schema shows the rule with the context elements at the top and the relation elements at the bottom. During the transformation, the rules will be applied, which means that the elements of the rule will be matched with and bound to the elements of the actual models. To explain the details, we need to distinguish three categories of elements in a rule, which are processed in different ways: We call these categories *context*, *bind*, and *create*. The graphical representation of these elements in the schema is shown on the right-hand side of Fig. 9. In order to apply a rule in a forward transformation, the elements of the context of the rule must be mapped to elements that are bound already (context: map to bound element). For produced nodes of a TGG rule, there are two categories: The elements of the source domain of the transformation will be mapped and bound to yet unbound elements (produce: bind to unbound element) of the source domain. When these elements are successfully mapped, they will be bound. The other category are the elements of the correspondence domain and the target domain: they will be created once the elements of the source domain were successfully mapped and bound. Once these elements are created, they will be also bound (produce: create and bind to new element). This exactly reflects the transformation as explained in Fig. 4: starting from the bound elements in the start context, the relation elements of the rule will be bound resp. created and bound until there is nothing left for being bound anymore. When everything

is bound, the newly created nodes in the target domain constitute the result of the transformation. Note that the matching, the creation and, thus, the binding mechanism applies not only to nodes, but also to edges.

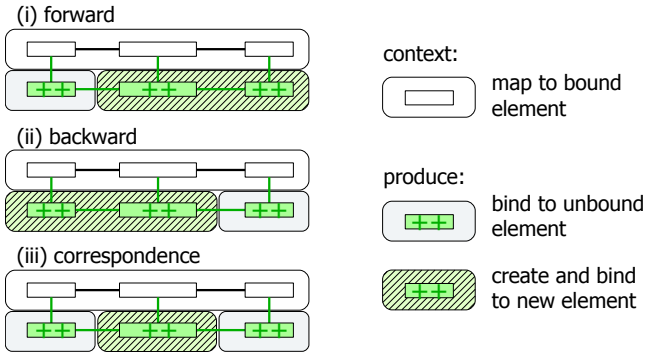

Fig. 9 The different application modes of a TGG rule and the interpretation of context/produced nodes

For a backward transformation, we just need to change two categories of processing the produced nodes: This is shown in row (ii) of Fig. 9. Note, that we do not need to change the rules for that; we just change the way in which the different elements are processed (i. e. create and bind, and bind unbound). The last scenario (iii) in Fig. 9 shows how the correspondences between two models can be created.

Actually, there is another, more general scenario which subsumes all the others: Let us assume that we have two models that, at some point in time, were consistent to each other according to the TGG rules. Then both models are changed independently of each other. In that case, both models need to be changed to bring them into correspondence again. We call that model *synchronisation*. Note that all other scenarios are a special case of synchronisation. For example, let us consider a forward transformation. Since the target model does not exist in this case, we first create the start situation as shown in Fig. 4(i) and assume that the synchronized pair that we started with is nothing more than the start context. Then, we start the synchronisation from there, which does the forward transformation.

Note also that the synchronisation scenario is more tricky than the other scenarios, because there are many different ways of coming up with two corresponding models. The easiest way would be to delete everything from both models such that nothing but the start situation remains as shown in Fig. 7. Though these two models perfectly correspond to each other, this is not what we want, since the two corresponding models do not contain any information of the two models we wanted to synchronize. And there are many other ways to come up with two corresponding models. The question is which pair we really want and what the "best" pair is. This very much depends on when, how and by whom the different changes on the two models have been made. And it might also depend on the role of the two models in some process. Technically, the "best" fitting pair could, for example, be captured by some kind of metric on a minimal distance to the pair we started from. But, we do not cover this topic in this paper.

In Fig. 4, we have discussed already how rules can be applied to transform one model into the other. In this example, it is always clear which rule can be applied in the forward direction. And this is typically the case for most practical applications of TGGs. But, in principle it could happen that different rules are applicable to the same part of the source model. In that case, we have non-determinism and it could happen that, dependent on the non-deterministic choices of rules, we cannot bind all the elements of the source model in the end. In that case, the transformation is not valid and we need to backtrack in order to try out other possible combination of rules. This is technically possible, but the backtracking makes the transformations much more inefficient. That is why, in practice, one chooses the rules in such a way that the application of the rules is deterministic. What is missing yet, however, is a theory when rules are deterministic and how to make rules deterministic (in analogy to classical theory of parsing deterministic grammars). But, we do not go into the details of that here.

2.4 QVT-Relations

QVT provides two declarative model transformation languages, QVT-Relations and QVT-Core. Both languages rely on two other OMG standards, namely the Meta Object Facility (MOF) [**?**] as a meta-modeling infrastructure, and the Object Constraint Language (OCL) [**?**] for describing model structures. QVT-Relations is the more user-friendly language whereas QVT-Core, in contrast, is slightly more verbose, intended to be interpreted by tools or to be compiled to executable code. The QVT specification provides a mapping from QVT-Relations to QVT-Core by means of a QVT-Relations transformation. The semantics of QVT-Relations is described independently of QVT-Core and there exists a tool today which directly operate on QVT-Relations (MediniQVT [**?**]). However, the QVT specification additionally defines the semantics of QVT-Relations by mapping to QVT-Core; the semantics of QVT-Core is described more formally in predicate logic.

Both a textual and graphical syntax is specified for QVT-Relations. Figure 10 shows the previously introduced example transformation in graphical QVT-Relations. The corresponding textual version is shown in Listing 1.

The relation `TrackToPlaceArcTransition` represents the relation corresponding to the TGG rule

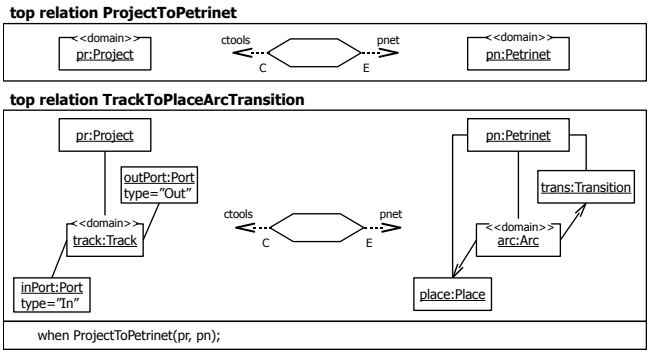

Fig. 10 The example transformation in QVT-Relations

shown in Figure 8, which looks very similar. However, there are some differences to the previously introduced TGGs. One difference to TGGs is that QVT-Relations uses OCL to describe model patterns. Expressions called *template expressions* specify the properties of model objects that shall be matched or created. For example, the expression `ctools::Port{ type = 'In' }` is an *object template expression*, which describes an instance of class "Port" with a "type" attribute that is equal to the string "In". This object template expression is used in the expression `componentToPort = portIn : ctools::Port{ type = 'In' }`, which is called a *property template expression*. This property template expression states that a valid match of the object template expression results in a binding of the variable `portIn` to a matched object. Then this property template expression specifies a property of an object described by another object template expression. In this example, this object template expression describes an instance of the class "Track". Three property template expressions specify values for its references `componentToProject` and `componentToPort` (refer back to the example meta-models in Figure 5.) The reader may observe that property template expressions do not differ between cases where a single-valued or set-valued property is specified. In the set-valued case, these expressions are interpreted such that at least one element in the set has to match the object template expression. In this nested structure of template expressions, the top-most expression contains the whole specification of the domain model pattern. It is therefore called the *domain pattern*. The (root) variable that it is assigned to is referred to as the *domain*. In the graphical syntax, such domain variables are annotated with `<<domain>>`. Reference values in object template expressions may also be specified by variable expressions which contain previously bound variables, and, apart from binding objects, variables can bind attribute values which can be used in variable expressions of other object template expressions, for example `ctools::Project{ name = n }`. In the Listing below, `n` is a variable that, in principle, could take any value. Since this variable occurs twice in this rule, in conditions `name = n` for the project and the Petri net, this rule expresses that a project and a corresponding Petri net shall have the same name `n`.

Listing 1 (QVT-Relations Code)

```
top relation ProjectToPetrinet {
   checkonly domain ctools pr : ctools::Project{
      name = n
   };
   enforce domain pnet pn : pnet::Petrinet {
      name = n
   };
}

top relation TrackToPlaceArcTransition {
   checkonly domain ctools track : ctools::Track{
      componentToProject = pr : ctools::Project{},
      componentToPort = portIn : ctools::Port{
         type = 'In'
      },
      componentToPort = portOut : ctools::Port{
         type = 'Out'
      },
   };
   enforce domain pnet arc : pnet::Arc {
      arcToPetrinet = pn : pnet::Petrinet{},
      arcToPlace = place : pnet::Place{
         placeToPetrinet = pn
      },
      arcToTransition = trans : pnet::Transition{
         transitionToPetrinet = pn
      }
   };
   when {
      ProjectToPetrinet(pr, pn);
   }
}
```

Another difference between QVT-Relations and TGGs is that relations may contain explicit references to other relations. We see that the relation `TrackToPlaceArcTransition` contains a reference to the relation `ProjectToPetrinet` inside the so-called *when clause*. Such a when clause states that the relation shall hold in the context of a successful binding of the referenced relation. Here, the variables `pr` and `pn` are the parameters of the relation call. The signature of a relation is defined by the domain variables.

More specifically, this means the following in a forward transformation scenario, putting ourselves in the position of a transformation engine for a moment: We start with some relation, for example `TrackToPlaceArcTransition`, and find a set of bindings for the source domain pattern in the source instance model. We call this set of possible bindings the *binding candidates*. We have to evaluate all expressions in the domain pattern as well as those in the when clause. In this case, we discover that variable `pn` is restricted by the relation `ProjectToPetrinet`. So we have to process this relation now. For efficiency, we might have handled the referenced relations first, after analyzing

the relation dependencies. But this does not change the result. The `ProjectToPetrinet` relation is similar to the TGG axiom shown in Figure 7, with the difference that this relation may be applied several times, e.g. to several Project objects in the source model. This relation has no when clause with further restrictions and, thus, whenever their source pattern can be bound to the source model structures, the target pattern is created accordingly. Thus, after matching the Project and after creating the Petrinet object, we can return to the `TrackToPlaceArcTransition` relation and refine the previously established binding candidates according to the bindings from the `ProjectToPetrinet` relation for variables `pr` and `pn`. I.e. if we have found tracks with no parent Project, then those binding candidates are removed from the set of binding candidates. For each valid binding of the source pattern in the `TrackToPlaceArcTransition` relation, we can now create the corresponding target model structures.

We see that the domain patterns in the relations are marked by the keywords `check` and `enforce`. This describes whether the patterns can only be matched in existing models or whether model structures can be modified when necessary. According to the check and enforce modifiers, the example relations shown here allow only a transformation in the direction of the Petri net domain. QVT explains when target elements shall be created by the *check-before-enforce* semantics: When there are already existing elements in the target model that can be exactly matched by the domain pattern of a relation, QVT has the principle to bind existing model structures first before creating new ones. Existing model elements are those elements that are created by a previous rule application or elements that exist because we are in an application scenario of updating an existing target model. Target elements that existed before the transformation and remain unbound after a transformation will be deleted. Sometimes we would like to force the reuse of particular existing model elements although they do not exactly match in a relation's domain pattern. This is achieved by declaring one or more properties that uniquely identify these objects. These identifying sets of attributes are called *keys*.

Note that in TGG rules, we distinguish between nodes that may be bound only to yet unbound objects and nodes which may be bound only to already bound objects (see Fig. 9). The declarative QVT languages do not specify such a binding semantics for variables. We consider this a major semantic gap in QVT, which may lead to unintentional transformation results. We explain that in Section 5.3 and 5.4 and argue that we need a specific binding semantics as in TGGs for certain variables in QVT-Relations and QVT-Core. We also discuss keys and the check-before-enforce semantics in more detail in Section 5.4.

Another difference to TGGs is that QVT-Relations does not contain variables which explicitly express the correspondences between the domain model elements. The hexagon in the middle of each relation as shown in Figure 10 is just part of the graphical syntax and is not reflected in the code. Rather, the hexagon expresses that the relation itself *is* the correspondence between the variables in the domain patterns. In Section 2.5, we show that the QVT-Core rules which correspond to the relations contain one central correspondence variable (or *trace* variable in QVT terms).

We discuss further features of the QVT-Relations language, for example the *where clause* or *top* and *non-top* relations, when discussing their relationship to the concepts of TGGs in Section 4 and Section 5.

2.5 QVT-Core Mappings

In the following, we introduce QVT-Core and its semantics along our running example. As mentioned before, the QVT specification provides a mapping from QVT-Relations to QVT-Core. QVT-Relations is more user-friendly, whereas QVT-Core is a comparatively simple, but more verbose language. On the one hand, QVT-Core is simpler because in QVT-Core rules, called *mappings*, model structures are described by a flat set of variables and expressions. The nested template expressions of enforceable domains of QVT-Relations are mapped to *nested mappings* in QVT-Core, but we will omit these details here. On the other hand, QVT-Core mappings are more verbose because an extra domain is introduced, which contains explicit variables and expressions to describe the *trace model* between the transformed domains. The trace model is essentially the same concept as the correspondence model in TGGs. The simplicity of QVT-Core and the analogous concepts of the trace and correspondence model are the reasons why we base our mapping from the relational QVT to TGGs on QVT-Core, as shown in Section 3. In Section 4, we sketch an idea of how to map QVT-Relations to TGGs directly.

Figure 11 schematically shows the QVT-Core mapping which corresponds to the two relations shown in Figure 10. We improvise a graphical syntax here, because the QVT specification defines only a textual syntax for QVT-Core. Listing 2 shows the corresponding textual representation.

Similar to TGGs, we see three columns in this mapping, which are called *areas* in QVT-Core. The outermost areas represent the domain models of the transformation and therefore these areas are called *domain areas* or *domains* according to the terminology in QVT-Relations. `check ctools(...)...` and `check enforce pnet(...)...` are examples of such domains in the textual syntax. The middle column is called the *middle area* and it contains the variables and expressions to describe the explicit trace model structure between the domain model elements. The trace (class) model is generated

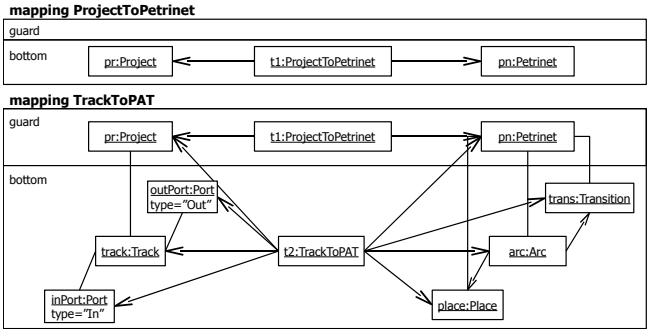

Fig. 11 The example transformation in QVT-Core

during the transformation from QVT-Relations to QVT-Core (see Section 2.6 or the QVT-Specification [**?**] for details). In the textual representation, the middle area is described in the `where(...){...}`-statement. (The where statement of QVT-Core must not be confused with the where clause in QVT-Relations, see Section 4.) The `check` and `enforce` keywords are used again here, similar to QVT-Relations to distinguish such domains which may only be matched and such which may also be created or modified. We explain the semantics of a mapping more closely after explaining further parts of the mapping structure.

Each area consists of two *patterns*: a *guard* pattern (enclosed in parenthesis) and a *bottom* pattern (enclosed in braces). The guard patterns of a mapping represent the context of a mapping. This means that, when there is a valid match of the guard patterns of all domains, a mapping needs to *hold*. A mapping holds, when we can find a valid match for either all bottom patterns or none of the bottom patterns. The pattern descriptions consist of variable declarations and expressions, as for example in this bottom pattern shown in the listing {`track:Track, ...| track.componentToProject = pr; ...`}. The expressions in the patterns are either *predicate expressions* or *assignments*. Predicate expressions constrain properties of the objects to be matched, i. e. they specify attribute values of the objects, for example `portIn.type = 'In';` and they constrain the references among the objects to be matched, for example `track.componentToPort = portIn;`. Assignments are similar to predicate expressions, but they express object properties which may be assigned in enforceable bottom patterns. Instead of the equality sign in predicate expressions, assignments use the assignment operator, for example `arc.arcToPlace := place;`. The variables in enforceable bottom patterns can be marked as *realized variables*, which represent objects that may be created or deleted when enforcing the pattern and for which property values may be assigned.

Listing 2 (QVT-Core Code)

```
map ProjectToPetrinet{
  check ctools(){
    pr:Project
  }
  check enforce pnet(){
    pn:Petrinet
  }
  where(){
    t1:TProjectToPetrinet|
    t1.pr := pr; t1.pn := pn;
  }
}

map TrackToPlaceArcTransition{
  check ctools(pr:Project){
    track:Track, portIn:Port, portOut:Port|
    track.componentToProject = pr;
    track.componentToPort = portIn;
    track.componentToPort = portOut;
    portIn.type = 'In'; portOut.type = 'Out';
  }
  check enforce pnet(pn:Petrinet){
    realize place:Place, realize arc:Arc,
    realize trans:Transition|
    arc.arcToPetrinet := pn;
    arc.arcToPlace := place;
    arc.arcToTransition := trans;
    place.placeToPetrinet := pn;
    trans.transitionToPetrinet := pn;
  }
  where(t1:TProjectToPetrinet|
    t1.pr=pr,t1.pn=pn){
    realize t2:TTrackToPlaceArcTransition|
    t2.pr := pr; t2.pn := pn;
    t2.track := track; t2.inPort := inPort;
    t2.outPort := outPort; t2.pl := pl;
    t2.arc := arc; t2.trans := trans;
  }
}
```

Operationally, the semantics of a QVT-Core mapping is that, if there exists a valid binding of all guard patterns and there exists a valid binding of a bottom pattern of at least one domain, we have to make sure that there exist bindings for all other bottom patterns: The bindings of the other bottom patters are checked first, which means that, for checkonly and enforced bottom patterns, existing elements are bound when an exact match of the pattern can be found. When no valid match can be found for an enforceable bottom pattern, we create this pattern. In the case that we have a valid match of at least one enforced bottom pattern, but fail to find a match for a checkonly bottom pattern, then we have to delete the matched objects from the enforceable domains. In the case that we can find a valid match for at least one checkonly bottom pattern, but fail to find a valid match for another bottom pattern, the mapping is violated.

As mentioned in the previous section, QVT does not specify whether certain variables may be bound only to yet unbound objects or only to already bound objects. We discuss that in Section 5.3.

2.6 Mapping QVT-Relations to QVT-Core

The QVT specification provides a mapping from QVT-Relations to QVT-Core in order to specify the semantics of QVT-Relations in more detail. Furthermore, QVT proposes to use QVT-Core as a basis for executing transformations that are specified in QVT-Relations. We do not discuss the details of this mapping here, but we want to give a short, example-driven summary of how the QVT-Core mappings in Listing 2 are obtained from the relations in Listing 1.

A QVT-Core transformation specification that is obtained from a QVT-Relations transformation specification consists of a set of QVT-Core mappings where each mapping corresponds to a relation. The domain areas of a mapping correspond to the domains of a relation. The template expressions in the domains of a relation are mapped to sets of flattened predicate expressions in the domain areas of the mapping. Variables in the guard patterns of the mapping are those which are involved in a relation call of a when clause. For each relation in the QVT-Relations transformation, a class for the explicit trace model is created. For each variable in a relation, the trace class has a reference to a domain class which is the type of this variable. In QVT-Core, this trace class is the type for the variable appearing in the middle bottom pattern of a mapping (see `realize t1:TProjectToPetrinet` and `realize t2:TTrackToPlaceArcTransition` in Listing 2). In the middle bottom pattern, all the reference property values are assigned for the trace variable, pointing to the other variables of the mapping (`t2.pr := pr; t2.pn := pn; t2.track := track; ...`). The trace variables which appear in the middle guard pattern are typed over the trace class belonging to the relation invoked in the when clause relation call (see for example `where(t1:TProjectToPetrinet|...)`) and predicates expression link the trace variable(s) to the domain guard pattern variables which were passed on as parameters in the when relation call (e.g. `t1.pr=pr,t1.pn=pn`).

The translation of enforceable domain patterns from QVT-Relations to QVT-Core is actually a bit more complicated than shown here: The nested template expressions in the relations' enforceable domains result in a *nesting of mappings*. However, we do not discuss this in more detail in this paper.

After having introduced TGGs, QVT-Relations and QVT-Core in this section, we show how QVT-Core can be mapped to TGGs in the following section. In Section 4, we return to QVT-Relations. There, we clarify the semantics of the where clause, which we omitted in this section. Also we describe the relationship between the explicit rule invocations in QVT-Relations and the implicit rule relationships in QVT-Core and TGGs.

3 Mapping QVT-Core to TGGs

As we have seen in the previous section, there are some apparent similarities between QVT and TGGs. In this section we show that a mapping can be specified from large parts of QVT-Core to TGGs. Semantic differences that persist beyond this structural mapping between QVT-Core and TGGs are discussed in Section 5.

We have specified a TGG transformation to describe this mapping and, in the following, we introduce a number of representative rules of this mapping. The full details have been worked out and implemented in a master thesis [**?**]. The rules and models shown in this section are diagrams taken from this implementation, which we explain here in more detail.

We have implemented a TGG engine as an Eclipse plug-in that allows to transform ECore models from the Eclipse Modeling Framework (EMF). Different from the implementation of TGG transformation engines in Fujaba [**?**] and MOFLON [**?**], where TGG rules are compiled to Java code, we interpret the rules directly. Although the rule interpretation may be slower, we experience that the interpreter logic is more easily maintainable and extensible this way.

Before explaining the transformation from QVT-Core to TGGs, we introduce the meta-models involved in the transformation.

3.1 QVT-Core and TGG Meta-Models

We implemented the QVT-Core meta-model for our mapping as it is described in the QVT specification. Since our implementation is based on EMF, we reused EMF's Ecore model as the implementation of the *Essential MOF* (EMOF), a subset of MOF [**?**], which is referenced by the QVT specification. Furthermore, we reused the OCL model used by EMF as the implementation of the *Essential OCL*, a subset of OCL [**?**], which is also referenced by QVT. Following the QVT specification, we implemented the *QVT-Base* package, which contains some base meta-classes that describe the fundamental parts of a transformation specification. The QVT-Core meta-model extends those base meta-classes. We considered some concepts of the QVT-Base package a suitable base for the TGG meta-model as well. Therefore, our TGG meta-model also extends QVT-Base, although not all the concepts of QVT-Base are reused in TGGs. The package dependencies are summarized in Figure 12.

Let us inspect the QVT-Base meta-classes first. Some details of the meta-models are omitted here, for example that some meta-classes extend classes from the EMOF package (see the QVT specification [**?**] for details). Furthermore, role names of the associations are omitted where they are equal to the name of the class of the association end.

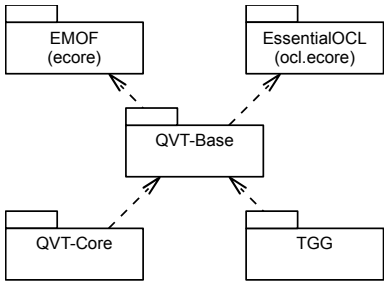

Fig. 12 The QVT-Core and TGG package dependenies

The QVT-Base package is shown in Figure 13. As shown here, a transformation consists of a number of rules, each being composed of a number of domains. A domain can be checkable and it can be enforceable. A domain is associated with exactly one *typed model* that references one or more packages. These packages contain the classes of the domain models involved in this transformation specification. The domain patterns of the rules are typed over these class models represented by the typed model. The QVT-Base package provides the basic notion of a pattern. A pattern consists of a number of predicates, which contain exactly one OCL expression (which shall evaluate to true or false). A pattern also consists of a number of OCL variables, which bind to model elements when the pattern is matched to a domain instance model. Each variable is typed by exactly one class.

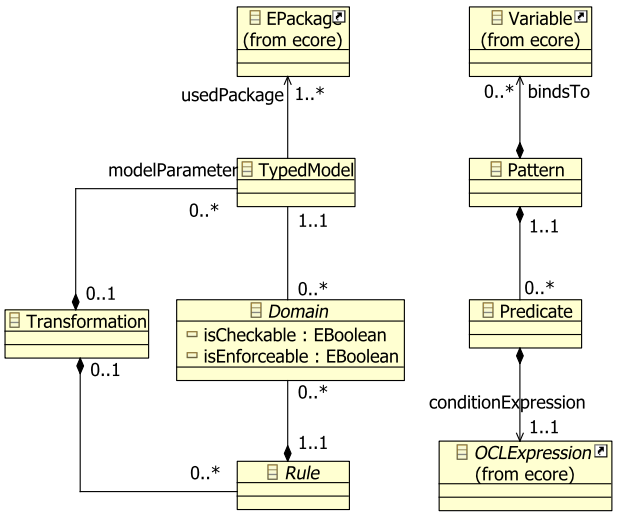

Fig. 13 The QVT-Base package

In the QVT-Core meta-model, shown in Figure 14, a *core transformation* extends the transformation meta-class from QVT-Base and a mapping in QVT-Core extends the basic rule meta-class from QVT-Base. As described previously, a domain in a QVT-Core mapping is also referred to as an *area*, which consists of a guard and bottom pattern. The *core domain* inherits the two concepts of the domain and the area. The guard and bottom patterns are each a *core pattern*, which extends the basic pattern from QVT-Base. Thus, each guard and bottom pattern may contain variables and predicates. These variables and predicates are typed over the typed model of the core domain that they are contained in. In addition to variables and predicates, a bottom pattern may consist of *realized variables* and *assignments*. As stated before, the realized variables and assignments are used to define creatable parts of the bottom patterns of enfoceable domains. Assignments are formed as follows: The *value expression* is an OCL expression which represents the value which shall be assigned to the property of some object in the enforced domain model. For example, the value expression of the assignment `portIn.type = 'In';` is a *string literal expression* representing the string `'In'`. The *slot expression* is typically an OCL *variable expression* that is referring to a variable in the same pattern, e.g. `portIn`. This variable binds the object to which the property value shall be assigned. Finally, the information about which property of the object shall be assigned with the value is specified by the target property of the assignment. It points to the structural feature (meaning an attribute or reference) of the type class of the slot variable, for example the `type`-attribute of the class `Port`.

Our TGG meta-model shown in Figure 15 also extends some concepts of the QVT-Base package. A Triple Graph Grammar, for example, is a transformation and a Triple Graph Grammar rule extends the rule from QVT-Base. Predicates and OCL expressions are currently not extended in the TGG meta-model. It would be desirable to use OCL in TGGs for expressing constraints on attribute values, but up to now only simple attribute value constraints are supported by our TGG interpreter implementation. For brevity, we omit our custom meta-classes for expressing these simple attribute value constraints. The reader may refer to [**?**] for details. Instead of using OCL to describe model patterns, we use a graph model: Each TGG rule is a graph, which consists of nodes which may have outgoing and incoming edges. Each TGG rule is also a (single) *graph grammar rule*, which has a left graph pattern and a right graph pattern (refer back to Figure 6). These patterns reference the nodes and edges which form part of that pattern (following the idea of the compact notation as shown in Figure 8). A Triple Graph Grammar rule is a graph grammar rule, which is separated into three or more parts (columns) representing the participating domains and the correspondence domain. Thus, a Triple Graph Grammar rule owns domain graph patterns and (not shown in the meta-model) every node in the rule is required to belong to exactly one domain graph pattern. Omitted in Figure 15 is that nodes are typed over classes from the typed model of the domain graph pattern they are in. Accordingly, edges are typed over references in the typed model of their domain graph pattern.

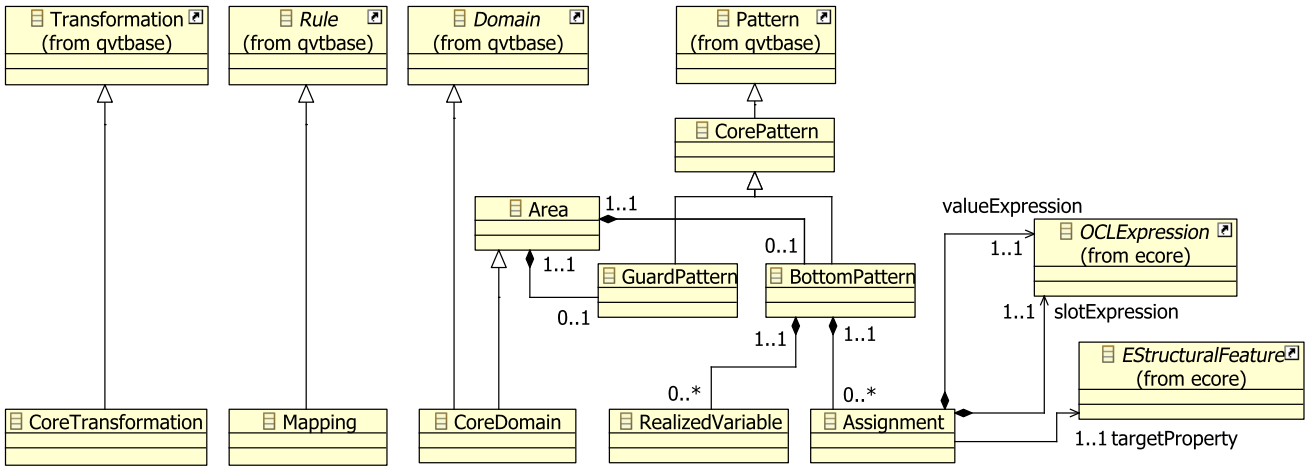

Fig. 14 The QVT-Core Meta-Model

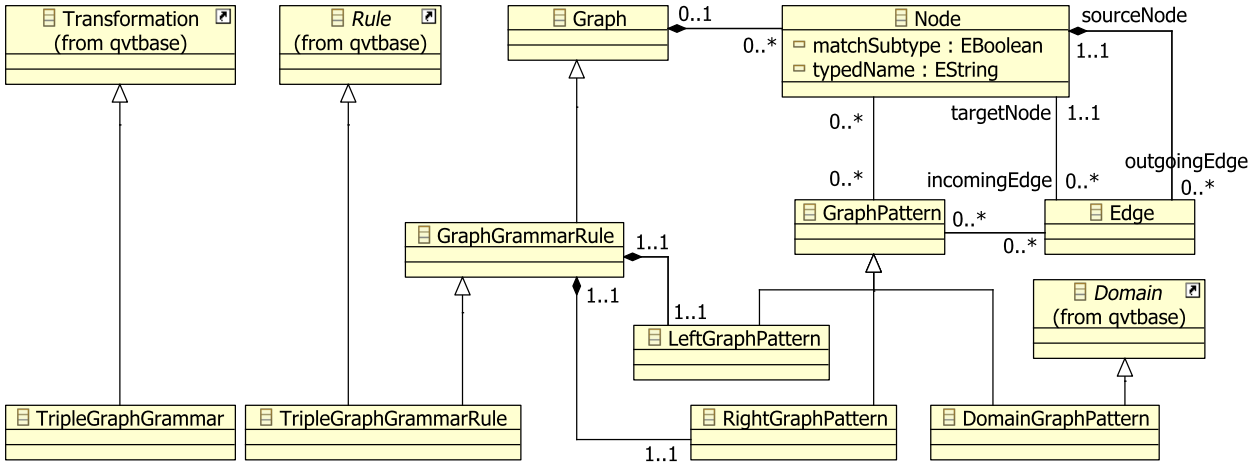

Fig. 15 The TGG Meta-Model

3.2 The QVT-Core to TGG transformation rules

In the following, we describe a number of representative TGG rules for translating a QVT-Core transformation specification into a TGG. The first TGG rules describe how the transformation structure and rule structure is mapped. Then we show how to translate a guard pattern variable in a QVT-Core mapping to a context node in a TGG rule and we explain how an enforceable assignment of a reference value is translated into an edge. At last, we explain how to map a string value assignment to the corresponding TGG constraint.

The start context of the QVT-Core to TGG transformation shown in Figure 16 states that a core transformation corresponds to a TGG. The rule shown next in Figure 17 depends on this start context and states that whenever a mapping is created in a QVT-Core transformation, then a TGG rule with a left and right graph pattern has to be created in the TGG. In the following, we may also read the rules in such a way that "a mapping corresponds to a TGG rule with a left and right graph pattern", but most of the following transformation rules we explain with the QVT-Core to TGG transformation direction in mind. Then, we may say that "a mapping is transformed into a TGG rule with a left and right graph pattern". The rule diagrams shown here are exported from the TGG rule editor that we implemented in Eclipse. The edges are annotated with the name of their type reference. E.g. a core transformation contains mappings by the composition relationship "rule" that is inherited from the QVT-Base package. The edge pairs represent bidirectional links. For readability of the diagrams, we hide the type name annotations on the edges whereever we see no risk of confusion. The domain graph patterns that we previosly presented as the "columns" of a TGG rule are now represented by the nodes at the bottom of the rule diagrams which are labelled with "(Domain)". The dashed lines leading from the domain graph pattern node to the rule nodes express which nodes belong to which domain. The yellow, rounded rectangles

in the rule diagrams represent *attribute equality constraints*, which state that certain attribute values of objects have to be equal. In the rule shown in Figure 17 it is required that the name of the TGG rule is equal to the name of the QVT-Core mapping and vice versa. The attribute equality constraints have a direction in which they are interpreted, indicated by the arrows on the connection ends. Currently, our TGG interpreter requires two such statements for each transformation direction. In the future, however, we would like to express such bidirectional constraints with only one statement.

When translating QVT mappings, we actually need to distinguish between translating regular mappings and *nested mappings* (see also Section 2.5), but we omit further details here.

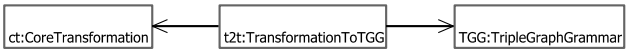

Fig. 16 The transformation start context

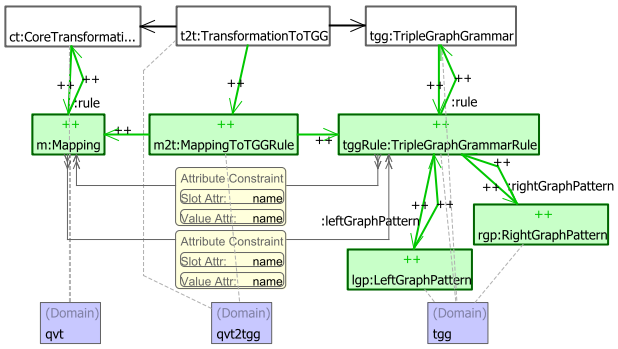

Fig. 17 Rule: MappingToTGGRule

Based on the axiom, also the TypedModelToTypedModel rule shown in Figure 18 can be applied to transform the typed model of a core transformation to a typed model of a TGG transformation. Next, rule UsedPackageToUsedPackage in Figure 19 states that any packages referenced by a typed model of a QVT-Core transformation is also referenced by the corresponding typed model of a TGG. This rule may indeed seem odd compared to the other TGG rules that we presented so far, because there are no produced (++) nodes, but instead a *gray node* annotated with "##" that is belonging to yet another domain (ecore). The gray (##) node or *reusable node* as we call it, is a concept which we have not explained so far. It allows us to cover two distinct cases in one rule. To explain this, we read the rule as a single graph production rule: For two corresponding typed models, we may (1) produce a new package and create a "usedPackage"-link to this package from both typed models. However, (2) we may want to reference an already created package as a used package from the two typed models. In this case, only the "usedPackage"-links shall be created to this existing package. Thus, using the reusable (##) node, is an abbreviation for writing two rules: (1) One rule where the package node is a produced (++) node and (2) another rule where the package node is a context node. In our QVT-Core to TGG transformation, the ecore domain is a source domain and, therefore, the used packages are only matched and never created. Thus, we do not need to qualify further, which packages will be reused; we match a package multiple times when it is referenced as a used package multiple times by different typed models. In Section 5.3 and 5.4, we explain why distinguishing these different types of nodes (context nodes, produced nodes and reusable nodes) is important.

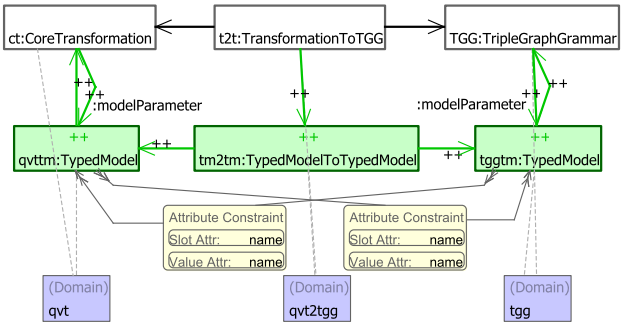

Fig. 18 Rule: TypedModelToTypedModel

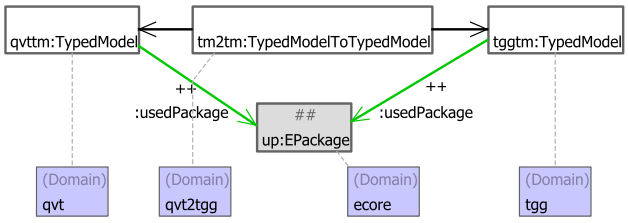

Fig. 19 Rule: UsedPackageToUsedPackage

Next, let us consider the GuardDomainToDomainGraphPattern rule shown in Figure 20, which translates a core domain of a QVT-Core mapping into a domain graph pattern of a TGG rule. This rule requires that the core mapping was already mapped to a TGG rule by the rule MappingToTGGRule (Figure 17). The guard and bottom patterns of the core domain are matched and correspondences are created between the guard and bottom pattern of the core domain and the left and right graph pattern of the TGG rule. Here we see that each core domain has a guard and a bottom pattern whereas there exists only one left graph pattern and one right

graph pattern per TGG rule. A guard pattern specifically corresponds to the left and right graph pattern and a bottom pattern corresponds to the right graph pattern only. This expresses the fact that guard pattern variables correspond to context nodes and bottom pattern variables correspond to produced nodes (as explained shortly). Furthermore, the typed model of the core domain is required to be previously translated into a typed model of the domain graph pattern (see the TypedModeToTypedModel rule in Figure 18). When the CoreDomainToDomainGraphPattern rule is applied, the domain graph pattern is set to reference the typed model corresponding to the typed model of the core domain.

Now we have introduced all rules which provide the context for translating variables in QVT-Core mappings to nodes in TGG rules. We explain the GuardPatternVariableToContextNode rule in Figure 21. We see that a variable of a guard pattern is translated into a node. The node, however, is required to be associated with a number of other elements: The node firstly needs to be contained in the Triple Graph Grammar rule. Secondly, the node belongs to the domain graph pattern which correponds to the variable's core domain. Thirdly, the node is required to be a context node, which is expressed though its association with the left and right graph patterns of the TGG rule. Variables in the bottom patterns of a mapping are translated to produced nodes in TGG rules similarly. The difference is that no links are created between the node and the left graph pattern. We also do not distinguish between translating variables or realized variables, because in TGGs we do not distinguish between enforceable and patterns and such which are only checked. We discuss this in more detail in Section 5.1.

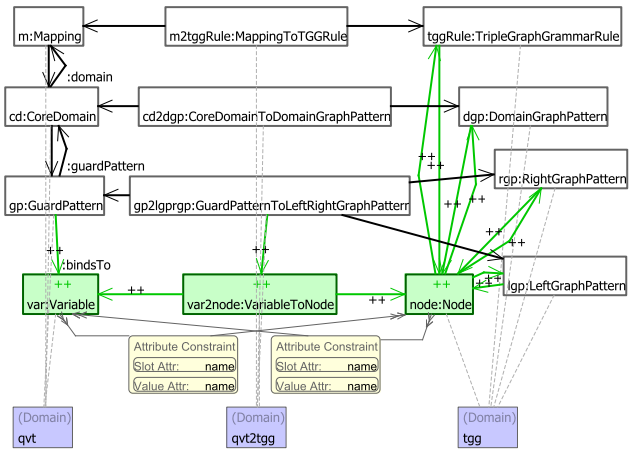

Fig. 21 Rule: GuardPatternVariableToNode

The rule for ensuring that the node's type class is equal to the type class of its corresponding variable is shown in Figure 22. The principle of this rule is similar to the UsedPackageToUsedPackage rule in Figure 19 – the variables'/nodes' type classes may be produced or reused when the classes serve as type classes for multiple variables/nodes.

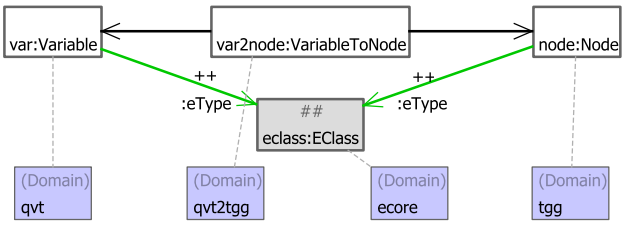

Fig. 22 Rule: VariableTypeClassToNodeTypeClass

The rest of the QVT-Core to TGG translation consists of rules for translating predicates and assignments into edges or attribute value constraints. We distinguish between such predicates and assignments which refer to reference properties and those which refer to attribute (data) values. The former are translated into edges of the TGG rule, the latter are transformed into attribute value constraints. As we currently only support simple attribute value constraints in TGGs, we can only map expressions where string, integer or boolean values are constrained or assigned or where the equality of two attribute values is compared.

The ReferencePredicateToEdge rule in Figrure 23 shows how to translate a guard pattern predicate with an OCL expression over a reference value to a context edge. The pattern which we see on the QVT side corresponds to the abstract syntax of OCL for an expression like `track.componentToPort = portIn;`. The "equals" corresponds to an OperationCallExpression (named "EqualsOperationCallExpression", but we don't check this is this rule), which performs an equality check between a value given by a source OCL expression and an argument OCL expression. The source expression is the property call expression in front of the equals sign (e.g. `track.componentToPort`). The property call expression references a source expression which is pointing to an object, in this case through a variable expression (e.g. refering to the `track` variable). The argument of the property call expression is refering to a property. In this rule for translating a predicate to an edge, we require the referred property to be a reference (which is an "EReference" in the ECore meta-model). Furthermore, in this rule we require the reference to be unidirectional. ECore expresses bidirectional references in such a way that an EReference may be paired with another "opposite" EReference. The NULL node in this rule expresses that the "eOpposite" reference value of the referred EReference is NULL and, thus, we are dealing with a unidirectional reference. In case of a bidirectional reference, we need to create two edges in TGGs for representing each link separately. This is a technicality required by our implementation of the

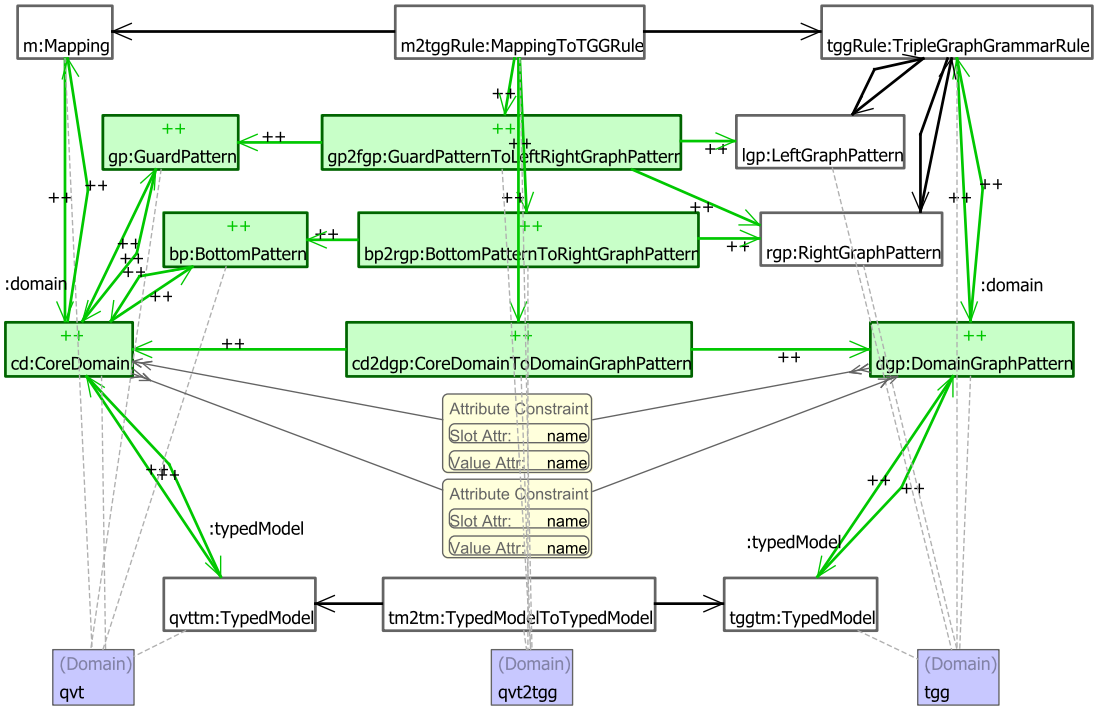

Fig. 20 Rule: CoreDomainToDomainGraphPattern

TGG interpreter, which complicates the QVT to TGG mapping – surely, there is room to improvement.

Note that the edge leading to the NULL node is marked by "##". This indicates that this edge is reusable, where reusable has the same meaning as for reusable nodes that were mentioned before: it can be either used as a context edge of the rule or as a produced edge (see Sect. 5.4 for more details). In combination with NULL nodes, however, marking the edges is meaningless, since there shall not (and does not) really exist any link to be bound to such an edge. We use the reusable edge for a merely technical reason: the interpreter algorithm does not check for any existing link bindings when it is matching reusable edges.

The argument of the operation call expression is a variable expression pointing to the variable which is compared with the property (e.g. `portIn`). We see that this variable corresponds to the target node of the edge whereas the variable in the source expression corresponds to the source node of the edge. Since we are translating a guard pattern predicate, the edge is also associated with the left and right graph pattern of the TGG rule, which implies that it is a context edge.

With the last rule shown next, we want to explain two points which differ from the previous rule. Firstly, we show that expressions over object properties which are not reference properties, but attribute (data value) properties are mapped not to edges, but to attribute value constraints. Certainly, these mappings are specific to our TGG model and may be realized differently when OCL expressions or other query mechanisms are supported by TGGs. Secondly, we show how assignments which may appear in bottom patterns differ from OCL predicate expressions. Figure 24 shows the rule for translating a string literal assignment into a string literal constraint. The pattern on the QVT side of the rule describes the abtract syntax for an assignment expression like `portIn.type := 'In';`. The slot expression points to a variable expression, representing the object to which a value shall be assigned. The property which shall be assigned is specified by the target property which is referenced by the assignment. As just explained, we need to specify in particular that the target property is not an EReference, but an EAttribute – for the other case, there is a rule for translating a (produced) edge. Furthermore, the assignment references an expression which represents the value which shall be assigned to the objects property. In this case, it is an OCL string literal expression. The value represented by this expression is stored inside the expression's string symbol attribute, which is translated to the string value attribute of the string literal constraint on the TGG side. We omit further details about how attribute value expressions form part of the TGG meta-model – it is just important to note that these constraints refer to a slot node and a slot attribute to determine on which object which property should be checked or assigned.

Note that in TGGs, there is no distinction between an expression which is enforceable and one which is not. It is up to the transformation engine to decide the en-

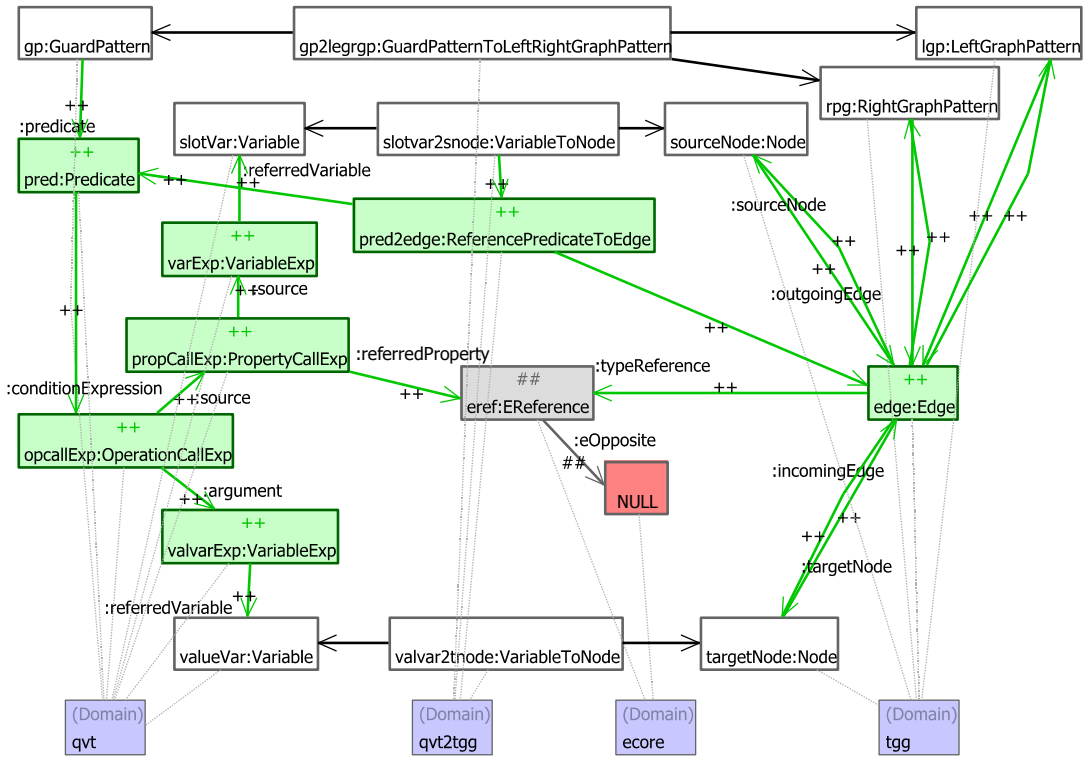

Fig. 23 Rule: ReferencePredicateToEdge

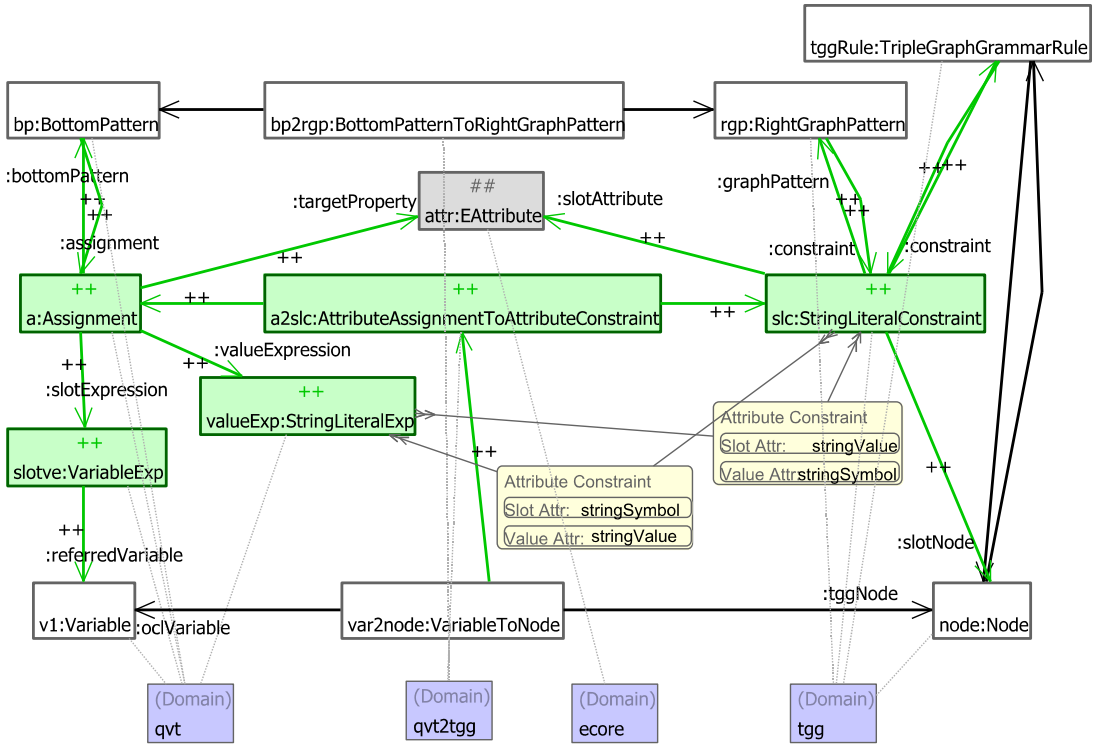

Fig. 24 Rule: StringLiteralAssignmentToStringLiteralConstraint

forcement of edges and attribute value constraints in a particular application scenario – we will come back to this in Section 5.

4 Relationship between the QVT-Relations and TGGs

In this section, we compare QVT-Relations and TGGs in more detail. We clarify the semantics of the where clause in QVT-Relations by explaining how it is mapped to QVT-Core. Furthermore, we explain how the explicit rule invocations through the when&where clause in QVT-Relations relate to the implicit rule dependencies in TGGs. Based on this relationship, we show how the mapping from QVT-Relations to QVT-Core could be improved in terms of efficiency. A direct mapping from QVT-Relations to TGGs furthermore provides a better understanding of QVT-Relations. The QVT specification is in some parts unclear about the exact semantics of QVT-Relations. Even today's QVT-Relations tools, ModelMorf[**?**] and MediniQVT[**?**], produced different results when we tested them with our example transformation (we explain details in Section 5.3).

Figure 25 shows a slightly extended version of the example transformation shown in Figure 10. New are the relations `PortToNode` and `ConnectionToArc` (the latter we have already shown schematically in Figure 3). `PortToNode` is a relation that maps a port to a node and translates the attached property objects into annotation objects of the nodes. (We have introduced an abstract class Node for this purpose in our Petri net metamodel that is the superclass of the classes Place and Transition.) In this rule, the value of each property's description attribute is translated into the text attribute of the corresponding annotation. We see that this relation is referenced twice in the so-called *where clause* of the `TrackToPlaceArcTransition` relation. This relation call in the where-clause means that the relation `PortToNode` has to hold for variables which are passed on to it as parameters. In this case, it is the pair of the in-port and the place and the pair of the out-port and the transition. Note that, every relation except for the invoked relation `PortToNode` is marked by the keyword *top*. This means that the relation `PortToNode` may not map all ports to Petri net nodes right away, but can only map those model elements that are passed on to the rule by the invoking relation.

Besides the transformation of the track component, the full ComponentTools to Petri net transformation will later on handle many more components, such as curves, switches, joins, stoppers, etc. The reader may peek at Figure 28 to see how a join component is translated into its Petri net logic. For each of these rules, in- and out-ports are mapped to particular places and transitions in the Petri net. Thus, each relation translating a component will invoke the `PortToNode` relation once for each port. The `PortToNode` relations are furthermore needed as a precondition when the connections that may connect the ports of components are translated to arcs between the Petri net nodes that correspond to the components' ports. This is done by the relation `ConnectionToArc`.

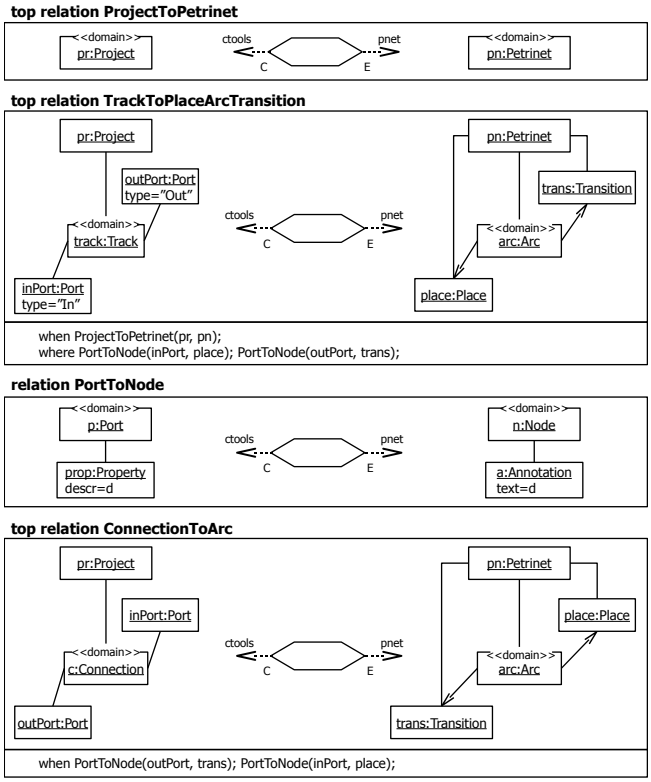

Fig. 25 The extended example transformation in QVT-Relations

The obvious difference between QVT-Relations and TGGs is that QVT-Relations explicitly formulates the dependencies of transformation rules in the when and where clauses. However, in TGGs these dependencies do also exist, but only implicitly by specifying that rules can only be applied when the necessary context patterns were previously produced by another rule.

In the mapping from QVT-Relations to QVT-Core, we have shown that variables in a relation that are involved in a relation call of the when clause (as parameters) are mapped to guard pattern variables in the QVT-Core mapping. (see Section 2.6). Furthermore, these variables are connected by a trace variable of the same type as the trace variable corresponding to the referenced relation. As we explained in Section 3, guard pattern variables are always translated to context nodes. Thus, we can infer that all variables of a relation involved in a when relation call correspond to context nodes in TGGs.

But, how is the where clause mapped to QVT-Core and how can we map it to TGGs? The translation

of a relation call in the where clause to QVT-Core is shown in Figure 26 for the example of rules `TrackToPlaceArcTransition` and `PortToNode` from Figure 25 (also refer to the QVT specification [**?**], Chapter 10). Firstly, nothing in particular happens to the variables in the mapping which corresponds to the invoking relation. But, in the mapping which corresponds to the invoked relation, the referenced variables are mapped to variables in the guard pattern. Those variables are referenced by an additional trace variable which is added to the guard pattern and which has the same type as the trace variable of the mapping which corresponds to the invoking relation. This far, this is the same principle as translating a relation call in the when clause that goes in the opposite direction. However, since a relation may be referenced by a where relation call by several other relations, the question arises which type to choose for the trace variable? QVT solves this by creating a separate mapping each time a relation is invoked. Each of these mappings are equal except for the type of the trace variable in the guard pattern and the expressions which link this trace variable to the domain variables. In our example in Figure 26, we show this "unfolding" of mappings. The relation which is referenced by two relation calls from the `TrackToPlaceArcTransition` variable is "unfolded" into two mappings. In our example, the trace variables do not differ, because the two relation calls originate from the same relation, but the domain variables are referenced through expressions refering to another reference. In the `TrackToPlaceArcTransition` mapping in Figure 26, we labeled the edges from the trace variable to all domain variables, which correspond to these expressions over different references. See how they reappear in the guard of the two mappings `PortToNode_TrackToPlaceArcTransition_1` and `PortToNode_TrackToPlaceArcTransition_2`.

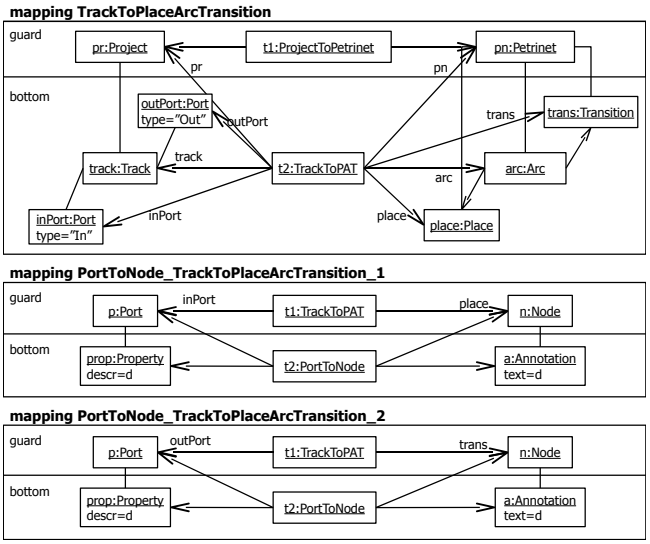

Fig. 26 Unfolding of PortToNode mappings in QVT-Core

Such an unfolding of mappings, however, may produce a large set of rules. In this example transformation, many more relations may be added to transform further components. Every time the `PortToNode` relation is called from these relations, a separate mapping is created in QVT-Core. This seems highly redundant and may possibly slow down the transformation process.

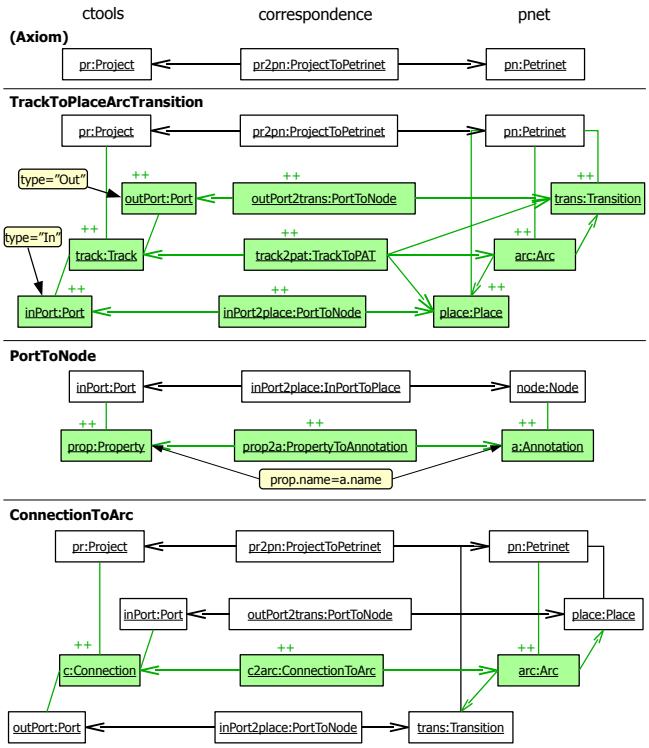

Fig. 27 The extended example transformation in TGGs

Figure 27 shows the TGG rule set that corresponds to the relations in Figure 25. The dependencies that are explicitly specified in QVT-Relations by when&where are implicit dependencies given by the required contexts of the rules. Different from the mapping to QVT-Core, however, we see that the TGG rule `TrackToPlaceArcTransition`, in addition to its central correspondence node, contains two other correspondence nodes that are typed by the correspondence class PortToNode. This is the same correspondence class which is the type of the correspondence nodes in the context of the `PortToPlace` TGG rule. This shows how we can express where relation calls in TGGs: for every occurrence of a where relation call in a relation, we insert a produced correspondence node with a distinct type into the corresponding TGG rule. This correspondence node has produced edges to the (domain) nodes which correspond to the variables involved in the where relation call. In the TGG rule which corresponds to the referenced relation, we insert a context correspondence node of the same type along with context edges referencing the nodes which

correspond to the domain variables of the relation. This strategy of mapping where relation calls to TGGs avoids the previously explained "unfolding" of rules. We feel that adapting this principle to the translation of QVT-Relations to QVT-Core would result in a much smaller and more efficient set of mappings.

We currently elaborate this direct mapping from QVT-Relations to TGGs. From the insights gained in this mapping, consistency constraints and a guideline for specifying TGGs can be formulated [**?**]. We see this complementary to the work by Klar et al. about structuring and modularizing model transformations [**?**].

5 Comparing QVT and TGGs

In the previous sections, we have discussed the main concepts of declarative QVT and TGGs, and we have shown how QVT-Core as well as QVT-Relations can be mapped to TGGs. This way, our TGG interpreter implements a transformation engine for QVT.

The QVT standard, however, leaves some room for interpretations, which becomes evident when comparing different implementations of QVT-Relations (see Sect. 5.3 for an example). In this section, we compare QVT and TGGs and discuss semantic gaps which currently exist in the QVT specification.

5.1 Application Scenarios

One fundamental difference between QVT and TGGs is that QVT encodes the direction of a transformation in the rules by the directives *check* and *enforce*, whereas there is no indication of a transformation direction in TGG rules. For TGGs, the transformation direction is provided only when a transformation is started, which we call the *application scenario*. We believe that it is one of the strengths of TGGs that the relation between two classes of models can be defined without explicitly referring to a transformation direction [**?**]. There may be transformations which may not generally be applicable in the backward direction, because the target domain does not contain the information needed to (re)create source domain models from it. Even in that case, we should not restrict the rules to only one transformation direction. Firstly, in some cases the rules can still be used to propagate changes in the target model back to the source model. For example, after an initial transformation from the source to the target model, the correspondence/trace information is maintained and therefore, in some cases, changes can still be synchronized between both models (see Section 2.3) in both directions deterministically. Secondly, in case that the source model and the correspondence/trace information is lost (or never existed), even not fully bidirectional transformations can be combined with reverse-engineering techniques to re(create) some possible source model. But, that is a different issue that is not in the scope of this paper.

5.2 Rules and their coordination

Our QVT-Core to TGG mapping shown in Section 3 and the approach for mapping QVT-Relations to TGGs in Section 4 shows that QVT and TGG rules are very similar. Using a graph model to describe a model pattern instead of OCL is a matter of taste, convenience, or personal opinion. Still, OCL is more powerful than simple graph patterns, in particular for expressing calculations on attribute values. This is why TGGs come with many different extensions. We propose to extend the graph model in TGGs by introducing constraints that may contain arbitrary OCL-expressions. In [**?**], we have shown how this can be implemented in TGGs.

By our mapping from QVT to TGGs, we have shown that the coordination of the rule applications is quite similar in QVT and TGGs. Even the explicit rule invocations in QVT-Relations through the relation calls in the when and where clauses can be mapped to corresponding (implicit) rule dependencies in TGGs (see Section 4).

One remaining issue in the coordination of the application of different rules is the question where to start the application of rules. In QVT, there are the start rules: For QVT-Relations, the start rules are such top-level rules that do not have any relation calls in their when-clause. For QVT-Core, these are the mappings with an empty guard. QVT can start matching any of these start rules, wherever they fit. This is different in TGGs: There is one axiom, which is a graph pattern, not a rule[1] ; the nodes of the axiom will be bound to the existing objects of the model. This initial binding is called the start context. From there, other rules will be applied. Thus, TGGs are very explicit where to start, which increases the efficiency of a transformation, but we may loose flexibility.

In practice, this difference between QVT and TGG does often not play an important role. Many QVT transformations are designed in such a way that a single start rule can be mapped in exactly one way to an existing model or a pair of related models initially—at the so-called roots of the respective models. This would be exactly the start context that we would provide to a TGG engine. Allowing multiple starting points for a QVT transformation furthermore raises the question of what happens to a QVT transformation when start rules

[1] Sometimes, this pattern is considered to be the right-hand side of a rule. Though this is technically correct, the axiom is not applied as a rule; therefore, calling an axiom a rule is conceptually misleading.

could be bound to the model in different ways. Are the bindings starting from different places allowed to overlap? This is a questions of how model objects may be bound to (bottom pattern) variables of applied rules, which is subject of the next section.

5.3 Bindings

It is very important for a correct application of the transformation rules to specify in which way objects in the model may be bound to nodes or, respectively, variables in a rule when the rule is applied. The semantics for mappings in QVT-Core (see Section 2.5 of this paper or Section 9.7 and 9.8 of the QVT specification [**?**]) is that when there is a valid binding of all guard pattern variables in a mapping and there is furthermore a valid binding of at least one bottom pattern, then valid bindings have to exist for all other bottom patterns. This semantics corresponds to the intuitive idea of a relation—either all "columns" of a relation should match or none. But the QVT specification does not specify whether model objects may be bound to at most one bottom pattern variable, i. e. whether bindings of model patterns of different rule applications may overlap in the instance model. However, this restriction is important because, if not the case, the intuitive sounding one-to-one relation between the bottom patterns cannot be ensured in the instance model where we see the results. Stevens [**?**] also raises this question—in the course of matching existing models with a QVT rule, may bindings found for the bottom pattern variables overlap? In our opinion, QVT should distinguish between variables where bindings may overlap and variables where bindings must not overlap.

In TGGs, we require that, in the end, every object in the models is bound to exactly one produced node of an application of a TGG rule (see Section 2.3). In the following, we refer to this semantics as the *bind-exactly-once semantics*. Note, that the model objects may of course be bound to several context nodes of different applications of TGG rules. In our mapping from QVT-Core to TGGs as presented in Sect. 3, we map each bottom pattern variable of a QVT-Core mapping to a produced node of a TGG rule. Thus, our interpretation of the QVT standard is that the same bind-exactly-once semantics applies to bottom pattern variables. Therefore, the bind-exactly-once sementics applies to such variables in QVT-Relations that are not part in a when relation call (see Sect. 2.6) and not referenced by a where relation call of another relation (see Sect. 4).

A particular part of the bind-exactly-once semantics in TGGs is that, in the end, all model objects must be bound. If there are left-over model objects that are not bound, the transformation is incomplete. In order to define when a transformation is successful in the case that we only intend to transform part of the model, we have to specify a *view* of the model to which the transformation is applied. Then, we require that all objects in this view are bound exactly once to a produced node. In [**?**], we call the parts outside the view of a model the model's *context*. The context may be relevant in constraints, but these objects are not bound at all, and can therefore be ignored by the TGG engine. In our implementation, we consider the view simply to be those objects and links in the model which are instances of the classes and references that we use as node and edge types in the TGG rules. This is, however, not a general strategy. Our interpretation of the QVT-standard is that the same bind-exactly-once semantic applies. However, QVT does not specify whether a transformation is unsuccessful when objects remain unbound. It is only stated that objects which remain unbound when transforming in the direction of an enforced domain shall be deleted (see Sect. 2.4).

Experiments with MediniQVT [**?**] and ModelMorf [**?**], which both implement QVT-Relations, show that there are other interpretations of how the objects may be bound in QVT-Relations. These interpretations may yield unintended results. Figure 28 shows a slightly more complicated TGG rule from the ComponentTools project—for ease of understanding, it is presented in the concrete syntax of the two domains. It transforms a join component into a corresponding Petri net logic. We encoded the rule in Figure 28 as a QVT-Relations rule in MediniQVT and ModelMorf. The result when testing MediniQVT with this example was that the rule was applied four times for a single join component. The number 4 comes from the four different ways that the in-ports of the instance model can be bound to the in-ports of the bottom pattern of the rule. See Fig. 29 for an illustration of the output. (The two in-ports of the join component are called "PortIn1" and "PortIn2", the out-port is called "PortOut". Fig. 29 displays the names of the corresponding places and transitions.) The Petri net element, however, was only created once. The reason for that seems to be that MediniQVT has a distinct semantics for domain variables, which was the Petri net in this case. The domain variable on the ComponentTools side was the join component. We tried to specifiy the rule differently, in order to have not the Petri net as the domain variable. But that was not possible since, firstly the Petri net element is the only element from where we can navigate the whole Petri net structure and, secondly, there is no single Petri net element that relates to the join component—it is simply the whole structure.

ModelMorf shows a different effect. It obviously has a mechanism for choosing non-overlapping bindings for rules. But, in other examples, when we translated a join component connected with other components, we observed redundant nodes and arcs being created from the connections. But, thus far we can only second guess about the reasons for that. Admittedly, model structures with such symmetries may not occur in many model transformation, but neither is it an artificial example.

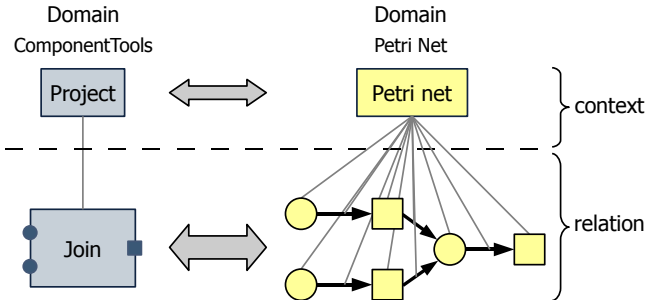

Fig. 28 Relation of a join component and its Petri net structure

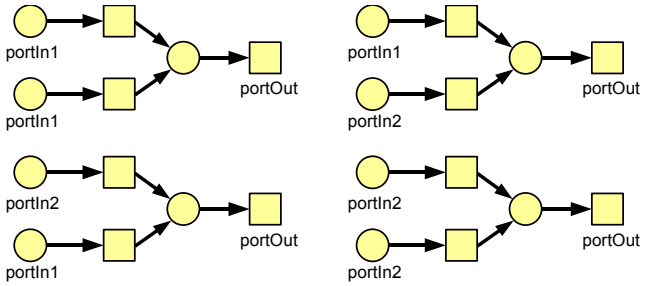

Fig. 29 The result of a MediniQVT transformation of a join component

The main point here is that there seem to be different interpretation of legal bindings in QVT and it is not possible to argue whether any of the implementations is correct or not based on the specification. From a TGG perspective, the bind-exactly-once interpretation for variable bindings makes the most sense.

5.4 Reusable nodes and global constraints

Actually, the bind-exactly-once-semantics of TGGs is sometimes a bit restrictive. The first reason for that is the following: As explained in the previous subsection, in order for a transformation to be successful, we defined that every object of a source model of a transformation must be mapped—even if it is not relevant for generating the target model. But, this is not a serious problem, because such parts could be considered not to belong to the model. Thus, we may imagine that the transformation engine works on a restricted view.

Still, the bind-exactly-once-semantics of TGGs sometimes forces us to make two different TGG rules for one and the same relation (see discussion of the rule of Fig. 19). Sometimes, we need one rule for the situations in which a particular object was bound already (then it would be in the context part of the rule); and we need another rule for a situation in which the object was not yet bound (then it would be in the production part of the rule). These two rules are basically identical, except that this node is marked with a ++ (produced node) in one rule and not marked with ++ (context node) in the other. And if we had several of such nodes within the same rule, the number of variants of that rule would grow exponentially with that number. This is why we introduced the concept of *reusable* nodes in [**?**]—since in a coloured representation the node is either a white or a green node, we graphically represent these nodes in gray and sometimes even call them *gray* nodes. The semantics of these reusable nodes is that, at the discretion of the TGG engine, they can be either considered to be in the context part or in the production part of the rule. Then, the way of dealing with nodes from the model's context and of dealing with reusable nodes is actually the same; both ideas can be captured by the same concept. We have already described an example of a reusable node in Section 3, see Figure 19 and its explanation.

The concept of reusable nodes serves a similar purpose as the check-before-enforce semantics of variables in QVT (see Sect. 2.4). The idea of check-before-enforce in QVT is the following: before creating the object, it is checked whether such an object exists already in the target model. If it exists, it will not be created another time; instead, the existing object will reused, i. e. the rule variable will be bound to the existing object. If it does not exist, it will be created. (Actually objects are only reused when a valid match of a complete domain pattern can be found, see Sect. 2.4.) These two cases resemble the two different choices for reusable nodes in TGGs. In TGGs, however, the reuse of an existing object is not given priority: it can be reused if it exists in the target model already, but it could also be created another time. To reuse existing elements or to create them anew is at the discretion of the TGG engine. It may be practical in many cases to give priority to reuse, as QVT does, but creating an element anew still leads to a valid correspondence of the models. The problem is that the concept of reusable nodes and the semantics of variable bindings in QVT introduces non-determinism. However, giving a priority to reuse does not completely eliminate this non-determinism again. The transformation result may still differ depending on the sequence of rule applications. Therefore, we prefer to keep the concepts of reusable nodes, which introduces non-determinism, separate from techniques to eliminate non-determinism.

In some situations, it might be acceptable to create an object several times; in other situations, we may not want that. Then, this can be defined by *global constraint* on the model. A global constraint is expressed in the meta-model class diagram and possibly additional OCL-constraints. Such a global constraint could be a 0..1 association or constraints like "an automaton may have only one state marked as the initial state". Such constraints could also require, for example, the uniqueness of some attribute value. The keys in QVT, which define the uniqueness of objects by one attribute value or by a combination of attribute values, are thus a special kind of global constraint.

In summary: the check-before-enforce concept of QVT combines the TGG concepts of reusable nodes

with a general priority on the reuse of elements in case of an exact match of a rule pattern and a reuse that is forced by global constraints. In TGGs, we prefer to keep the concept of reusable nodes that introduces non-determinism separate from concepts for restricting the non-determinism again.

6 Related Work

There exist a number of approaches for interpreting the semantics of QVT or for implementing QVT by a mapping to another domain.

Rensink and Nederpel provide a graph transformation semantics for MTL, one of the model transformations languages that were initially proposed for the QVT standard [**?**]. MTL is a declarative, unidirectional language where rules specify source and target patterns by simple query expressions. Rensink and Nederpel worked out a mapping from MTL into single graph transformation rules. In contrast to Rensink and Nederpel, our approach covers the declarative, bidirectional transformation languages that were finally included in the QVT standard.

Lengyel et al. have worked out a mapping from QVT-Relations into graph rewriting rules instead of bidirectional TGGs [**?**]. The when and where relationships among the relations are translated into a control structure in which graph transformation rules corresponding to the relations are executed. However, they do not discuss possible implications that their mapping has on the interpretation of the semantics of QVT-Relations.

Romeikat et al. provide a translation from QVT-Relations to QVT Operational Mappings [**?**]. QVT Operational Mappings is another language of the QVT specification for expressing transformations imperatively. In Operational Mappings, rules describe how to translate certain input objects into output objects. Rule dependencies in QVT-Relations that are expressed by when and where clauses are mapped to invocations of operational mappings from other mappings as well as *resolveIn*-expressions that allow mappings to refer to elements mapped by previously executed mappings. They also describe how the check-before-enforce semantics and the key mechanism of QVT-Relations may be realized in QVT-Operational Mappings, but they do not discuss in detail the implication of their mapping approach to the interpretation of QVT-Relations. The authors furthermore admit that their approach is limited to certain "easy" cases in QVT-Relations, but they do not specify their limitations in detail.

Instead of mapping QVT to another domain, Stevens provides a formalization of bidirectional model transformations [**?**]. She clarifies issues like the reversibility of transformations and the exact meaning of bindings as we have also discussed in Section 5.3.

7 Conclusion

In this paper, we have discussed the similarities of QVT and TGGs. Due to the similar structure and concepts, relational QVT can be transformed into TGG rules. We have shown how QVT-Core transformations can be mapped to TGGs and, this way, a TGG engine can execute transformations specified in QVT-Core. We also presented an idea on how QVT-Relations can be directly mapped to TGGs; but the exact mappings need yet to be worked out in detail.

In addition, we have discussed the differences in the philosophy and concepts between QVT and TGGs. This improves our understanding of how model transformations and model synchronizations work with relational rules. With the help of the semantics of TGGs, we clarified some semantic gaps which we find in the specification of QVT today. The insights gained from QVT have furthermore inspired some extensions of TGGs.

Acknowledgements The authors would like to thank Patrick Könemann for helpful discussions, Jörg Kiegeland for his support on MediniQVT as well as Oleg Travkin for the discussions and the work on his Bachelor project.

Author's Biography

Joel Greenyer received his master degree in Computer Science in 2006 from the University of Paderborn. He is currently a Ph.D. student of the International Graduate School Dynamic Intelligent Systems at the University of Paderborn, working in the Software Engineering Group of Prof. Dr. Wilhelm Schäfer. His main area of research is the model-based design of mechatronic systems, with the focus on the modeling and synthesis of interaction specifications as well as the consistency management among models, model transformation and model synchronization.

Ekkart Kindler is currently an associate professor in Computer Science and Engineering at the Technical University of Denmark (DTU). He received his masters and his PhD in Computer Science from the Technische

Universtität München in 1990 and 1995, resp. He received his Habilitation in Computer Science from the Humboldt-Universität zu Berlin in 2001. After his Habilitation, he was visiting professor in theoretical as well as in practical computer science at different German Universities and was an assistant professor (Hochschuldozent) in Software Engineering at Paderborn University from 2002 to 2007.

His research interests include formal methods and their application in software and systems engineering and business process management. Currently, he is working on formalizing and unifying the concepts of business process modelling and on techniques and tools for the automatic analysis and verification of system and process models. He is also working in the area of Model-based Software Engineering and techniques that make use of models in the software development process for getting rid of low-level programming. This includes techniques for interpreting models, for automatically generating code from models, as well as for model transformation in general.