Reconciling TGGs with QVT

Joel Greenyer and Ekkart Kindler

University of Paderborn, Department of Computer Science,
33100 Paderborn, Germany
{jgreen,kindler}@upb.de

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. 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 relation definition, a transformation engine can execute a transformation in both directions and, based on the same definition, can also propagate changes from one model to the other.

In this paper, we compare the concepts of QVT and TGGs. It turns out that TGGs and QVT have many concepts in common. In fact, fundamental parts of QVT-Core can be implemented by a TGG transformation engine. Moreover, we discuss how both technologies could profit from each other.

Keywords: MDA, model based software engineering, model transformation, model synchronization, 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 [1]. 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 and for platform and implementation specific details. In the end, the code can be generated from these models. This makes technologies for transforming, integrating, and synchronizing models a core technology within 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 basically 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. Moreover, the operational definition of a transformation makes it very hard to verify its correctness. By contrast, QVT[1] (Query/View/Transformation) [2] and TGGs (Triple Graph Grammars) [3] allow us to declaratively define the relation between two or more models. 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 have only a single definition of the relation between two classes of models, inconsistencies among the different transformation scenarios are avoided.

QVT is the transformation technology recently proposed by the OMG for the MDA. Actually, QVT has different parts: There are declarative and operational languages. Here, we focus on QVT-Core, which forms the basic infrastructure of the declarative part of QVT. 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. TGGs are at the core of FUJABA, for transforming back and forth between UML diagrams and Java code [4]. Another implementation of TGGs exists in the MOFLON tool set [5]. In addition to being declarative, QVT-Core 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 identify the essential concepts of a declarative approach toward specifying the relationship between two classes of models. Secondly, the 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 [6] and is briefly discussed in this paper. Thirdly, 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 – in particular this could provide valuable input for QVT as a standard.

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 can be mapped to TGGs, which provides an implementation of QVT based on a TGG engine. Section 4 gives a more detailed comparison of the philosophical, conceptual, and technical differences between QVT and TGGs.

2 QVT and TGG Transformation Rule Examples

In the following, we introduce QVT and TGG transformation rules along a small example. The example is picked from the ComponentTools project where component-

[1] Actually, QVT has different ways for defining transformations. Here, we refer to QVT-Core only, which is the basic infrastructure of the declarative part of QVT.

based material-flow systems can be designed and analyzed by the help of formal methods [7]. For instance, transportation or manufacturing systems can be designed by placing and connecting components, such as tracks, switches and stoppers, inside a *project*. Then, the project and its interconnected components are transformed into a formal model, for example a Petri net. Figure 1 shows how two connected Tracks in a Project should be transformed into a corresponding Petri net: A Track is represented by a Place, an Arc, and a Transition. The Connection simply corresponds to an Arc.

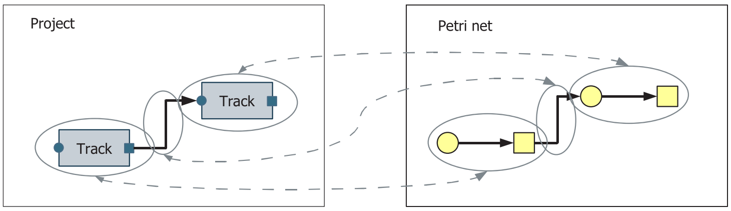

Fig. 1. An example of how two connected tracks are mapped to the corresponding Petri net

Figure 2 shows the way a rule can express how model structures correspond to each other. Here, it is specified that a Track relates to the particular Petri net construct in a certain context. The required context here is an existing relation between the parent model elements, the Project and the Petri net.

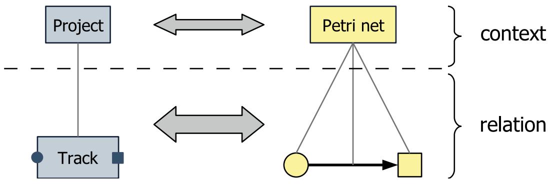

Fig. 2. A rule expressing the relation of model structures

Such rules, which express the relation between model structures, can be classified as *relational* rules and both QVT and TGG rules follow this archetype. Note that such relations can also be expressed between more than two models [8]. However, for simplicity, we focus on relations between only two models in this paper. The advantage of such rules over other transformation approaches is that they can be applied in different ways. They can be used to bidirectionally transform models, to check if two given models are equivalent, or, after an initial transformation, to incrementally propagate changes made in one model to the other. But, the rules do not describe operationally how to transform models. They are *declarative* and it is up to a transformation engine to make them operational.

For example, transforming a ComponentTools Project into its corresponding Petri net would involve the following steps. Firstly, an *axiom* or *start rule* is needed to map the Project root model object to a Petri net root model object. This provides the context necessary to apply the rule shown in Figure 2. The application of the above rule is visualized in Figure 3: For every Track in the Project, the corresponding Place-Arc-Transition-construct is created in the Petri net.

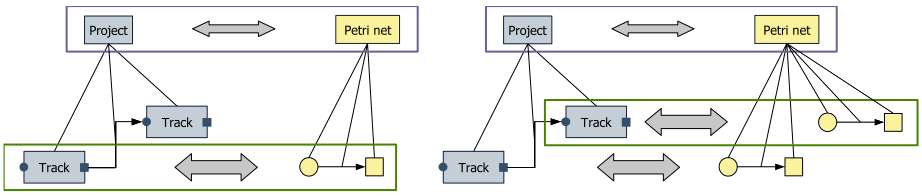

Fig. 3. Two rule applications in the example transformation

The model structures which are newly created during such rule applications provide the context for applying further rules. For example, a further rule would now be involved to transform the Connection between the components.

After introducing the concepts of relational rules and how they are used for model transformation, the following subsections will inspect the details of QVT and TGGs.

2.1 QVT-Core Mappings

The QVT specification defines two declarative transformation languages which form two layers of abstraction. Firstly, there is the more abstract and more user friendly QVT-Relations. QVT-Relations is then mapped to a more concrete language, QVT-Core, for which the semantics for performing transformations is defined in more detail. Although the concepts of QVT-Relations and its mapping to QVT-Core are very interesting, this paper focuses on QVT-Core, because it is the foundation of the declarative QVT and structurally more similar to TGGs.

In both QVT-Relations and QVT-Core, model patterns are described by OCL expressions. Although there is no graphical syntax specified for QVT-Core, Figure 4 illustrates the QVT-Core representation of the example rule from Figure 2 in a graphical way. Instead of using the concrete syntax of components and Petri nets, the model patterns are now shown in a notation similar to object diagrams. Each object node shown here represents an OCL variable. In the following, the terms *variable* and *node* are used interchangeably.

A single transformation rule in QVT-Core as shown here is called a *mapping*. The patterns in a mapping are structured in three columns[2], called *areas*, each consisting of a *guard pattern* and a *bottom pattern*. The bottom patterns represent the model elements which are actually brought into relationship by this mapping. The guard patterns specify the context which is required for this relation to hold. The outer columns, which contain the patterns belonging to the different involved models, are called the *domain areas*. In this example, the ComponentTools domain is abbreviated as `ctools` and the Petri net domain is abbreviated as `pnet`.

[2] There can be more columns in QVT-Core to specify relations between more than two models.

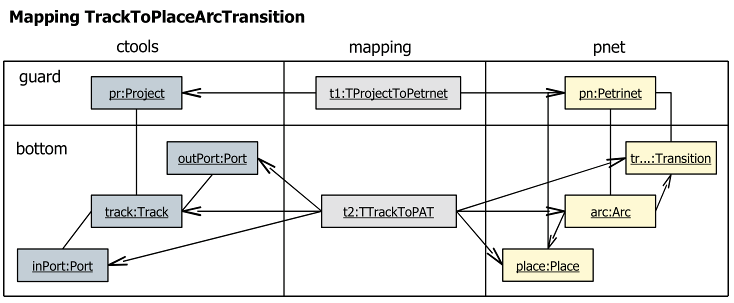

Fig. 4. A QVT-Core example rule

The center column, called the *mapping area*, contains additional elements, which embody the mapping of the involved domain patterns. In particular, there is one[3] mapping node in the middle-bottom pattern of a rule, which references all nodes in the domain-bottom patterns. In the course of a transformation, these "mapping nodes" are instantiated and keep track of the corresponding model structures. These objects are therefore called *trace objects*. In the rule, we will refer to those "mapping nodes" also as *trace nodes* or *trace variables*. In addition to the middle-bottom pattern, the middle-guard pattern can contain an arbitrary number of trace nodes, depending on the complexity of the context specified by this mapping.

Next, we have a look at the textual representation of this mapping, which is shown in Listing 1. First of all, we see the two domain areas represented by "**`check`** `ctools`" and "**`check enforce`** `pnet`". The keywords *check* and *enforce* determine whether the model patterns should just be matched in an existing model or whether model elements should also be created when they are missing. So, the mapping as shown below can just be applied in certain application scenarios, also called application *modes*: The rule can be applied to transform from a `ctools` model to a `pnet` model, but not backwards. Furthermore, if the `ctools` and `pnet` models both exist, the rule can check for a valid correspondence between the models. In particular, the enforce-keyword denotes that parts of the `pnet` model, which do not correspond to the `ctools` model, can be altered to establish a valid correspondence.

The domain areas are then structured in the way that the guard pattern is specified inside the parentheses following the domain identifier and the bottom pattern is specified in braces. The domain-guard patterns are fairly simple in this example, since they each contain just a single variable. But, in the bottom pattern, we see how the variable declaration is followed by a number of OCL expressions. These expressions describe the model structure, i.e. how the model objects should be referencing each other. Additionally, we see that there are two constraints formulated regarding a "type"-string of the Track's ports. In this example, these constraints are necessary to determine the in-Port and out-Port of the Track. For simplicity, these constraints were not reflected in the rule's graphical representation in Figure 4.

[3] QVT-Core is not restricted to just one mapping node in the bottom pattern of a rule. However, the QVT specification does not seem to intend the use of more than one.

Listing 1. The QVT-Core example rule in its textual notation

```
map TrackToPlaceArcTransition{
  check ctools(project:Project){
    track:Track, portIn:Port, portOut:Port|
    track.project = project; track.port = portIn; track.port = portOut;
    portIn.type = "in"; portOut.type = "out";
  }
  check enforce pnet(petrinet:Petrinet){
    realize place:Place, realize arc:Arc, realize trans:Transition|
    arc.arcToPetrinet := petrinet; arc.arcToPlace := place;
    arc.arcToTransition := trans; place.placeToPetrinet := petrinet;
    trans.transitionToPetrinet := petrinet;
  }
  where(t1:TProjectToPetrinet|
    t1.project=project,t1.petrinet=petrinet){
    realize t2:TTrackToPlaceArcTransition|
    t2.track := track; t2.inPort := inPort; t2.outPort := outPort;
    t2.place := place; t2.arc := arc; t2.transition := trans;
  }
}
```

Looking at the bottom pattern of the enforceable `pnet` domain, we see some differences to the previous domain. Firstly, we see that the variables are marked as *realizable*. This means that, when missing, they can be created when the rule is applied. Secondly, we see that the OCL expressions contain an assignment symbol := instead of a normal equals symbol. Since OCL is just a language to formulate constraints and queries, QVT introduces additional operations, called *assignments*, to assign reference or attribute values to model objects.

The mapping area is specified in the **`where`**-section. Here, we see the trace variables of the guard and bottom pattern. The expressions in these patterns specify how they reference the variables in the domain areas.

2.2 TGG Rules

The actual idea of Triple Graph Grammars (TGGs) is to specify how two types of graphs relate to each other. Because software models can be considered graphs, this theory can be applied to models as well. We assume that we have two types of graphs given and their structure is specified by (single) graph grammars. Then, TGG rules allow us to specify how these single graph grammar rules structurally correspond to each other. This correspondence is expressed by inserting a third graph grammar, where the nodes provide a mapping by referencing the nodes in the other two graph grammars. This is illustrated in Figure 5 for the example rule from Figure 2. The generation of graphs through the simultaneous application of these corresponding graph grammars always results in structurally corresponding graphs.

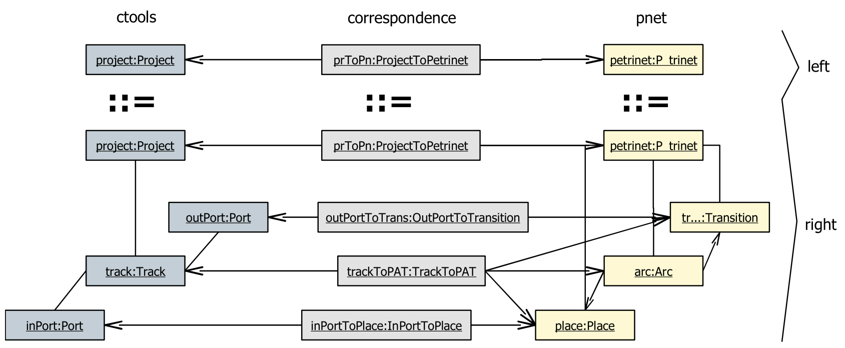

Fig. 5. The three graph grammar rules in a Triple Graph Grammar rule

Now, this formalism can be used to transform one graph into another. This is achieved by *parsing* an existing graph with the graph grammar on one side of the TGG. Then, during this process, the other graph grammar and the correspondence grammar of the TGG are applied to *create* the target graph and the correspondence graph. Another application of TGGs is to check two given graphs for their correspondence by parsing the two graphs simultaneously with the particular graph grammars in the TGG rules. In this process, the correspondence graph is built up and, in the end, represents the detailed correspondence of the nodes in the two graphs.

TGG rules are structurally very similar to QVT-Core rules. Figure 6 shows a collapsed representation of the above TGG rule. This collapsing is possible, because TGGs use only non-deleting graph grammars. This means that every element on the left-hand (top) side of the rule also appears on the right-hand (bottom) side. Note that we also introduced two attribute value constraints which were not present in Figure 7.

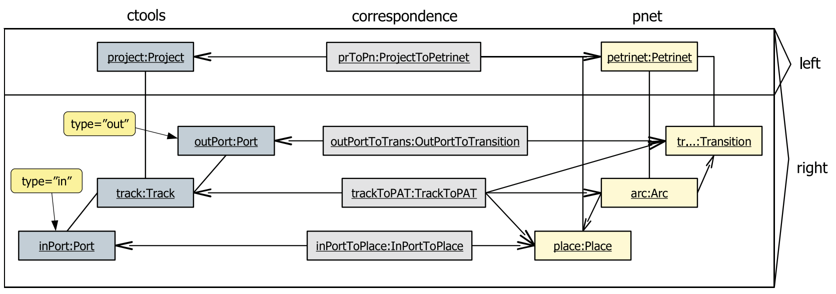

Fig. 6. A TGG example rule

Similar to QVT, there are three columns. The two columns containing the `ctools` and `pnet` patterns are called the *domains*. In fact, TGGs can also relate more than two models and may thus also be called Multi Graph Grammars (MGGs) [8]. In TGGs, the mapping nodes are called *correspondence nodes* and the middle column is therefore called the *correspondence domain*.

One difference between the above QVT-Core mappings and the TGG rules is that TGG rules have a graphical syntax. In Figure 4, we have just visualized the pattern structure made up by the OCL expressions in the QVT-Core rule. A TGG rule, in contrast, actually consists of *nodes*, representing the model objects, and *edges*, representing the references between these objects. Additionally, TGG rules may specify the attribute values of objects with *attribute value constraints*, as expressed by the rounded boxes containing labels attached to the nodes. Similar to QVT, OCL expressions can also be used for this purpose to specify literal values or values calculated from attribute values somewhere in the involved models.

After illustrating the structural similarities of QVT-Core and TGGs, the following section shows how the constructs of QVT-Core can be mapped to TGGs. The semantics of these rules are also very similar due to the fact that both QVT and TGGs are relational rules as illustrated in Figure 2. Remaining issues concerning slight differences in the semantics, or rather philosophies, are discussed in Section 4.

3 Mapping QVT to TGGs

As we have seen in the previous section, there are some apparent similarities between QVT-Core and TGGs. Therefore, a mapping can be specified from QVT-Core to TGGs. The full details have been worked out and implemented in a master thesis [6]. In the following, we informally describe the major steps of this mapping.

In the previous section, we observed that a variable in QVT-Core is essentially the same as a node in TGGs. Both a variable and a node are typed by a class in one of the involved models. Actually, in the redesigned TGG model proposed in [6], TGG nodes are also variables in terms of OCL, because the nodes should be reused as variables in OCL expressions as shown in Figure 6. Secondly, expressions which specify the reference values of objects in QVT are mapped to edges in TGGs. This holds for both OCL equality expressions, as `track.port=portIn`, and for assignments in an enforceable domain, as `arc.arcToPlace:=place`. 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 enforcement in a particular transformation scenario – but, we will come back to that in Section 4.

Apart from specifying reference values of objects, there are expressions in QVT which specify attribute values. For example equality expressions as `portIn.type ="in"` or assignments like `place.name:=track.name+"_"+portIn.name`. These expressions are not mapped to edges in TGGs, but to *attribute value constraints* as shown in Figure 6. Figure 7 summarizes the mapped constructs.

The overall structure of a single rule in QVT-Core is also quite similar to the structure of a TGG rule. The QVT-Core domain areas are mapped to TGG domain sides and, accordingly, the mapping area is mapped the correspondence column of a TGG. Then, the guard and bottom patterns in QVT-Core mappings are mapped to TGGs in such a way that the variables in the guard pattern of a QVT-Core mapping are mapped to such nodes in the TGG rule, which belong to the right-hand *and* the left-hand side. Variables which belong to the bottom pattern in the QVT-Core rule are mapped to nodes which belong to the right-hand side of the TGG rule *only*.

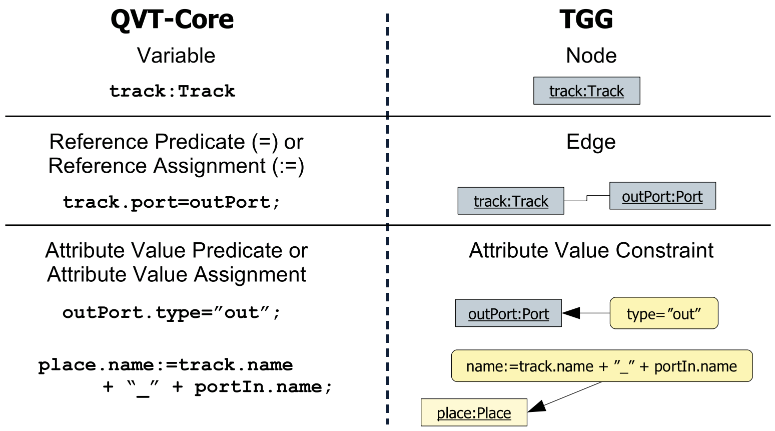

Fig. 7. Mapping the constructs of QVT and TGG transformation rules

Before designing a set of rules in QVT or TGGs, we need to specify a general transformation *setting*. This setting consists of references to the packages of the domain models involved in the transformation as well as the package of the correspondence or trace model. In the redesign of the TGG model proposed in [6], this description of a transformation setting was adopted from what is called the QVT-Base package in QVT. Therefore, technically, there is a one-to-one mapping at this level between QVT and TGGs.

We have actually specified the above mapping from QVT-Core to TGGs by a set of TGG rules. So, we use TGGs to transform QVT-Core mappings into TGG rules and, thus, we can perform model transformations specified in QVT-Core. One example of such a QVT-Core-to-TGG rule is shown in Figure 8. Here, a variable in the guard-pattern of a QVT-Core mapping is related to a node which belongs to the right-hand side and left-hand side pattern of a TGG rule. It is made sure that the QVT variable and TGG node belong to the correct domain column by referring to the corresponding CoreDomain and DomainGraphPattern in the context of the rule.

However, we do not explain the details of this rule, but rather illustrate the steps required to actually implement this transformation. Firstly, to create and transform QVT and TGG rules, both a QVT and TGG metamodel is needed. Because there are yet no implemented metamodels of the declarative QVT languages available, we implemented them anew, according to the QVT specification. A TGG metamodel and transformation engine was available from the ComponentTools project [9]. However, due to insights gained during the comparison of TGGs with QVT, we decided to conduct a redesign and reimplementation of the TGG technology. For the implementation, we chose the modeling framework of the Eclipse platform, EMF [10], as a basis. For one reason, The ComponentTools project and its TGG technology was already based on EMF. Another argument for using EMF is that it provides many useful features and that there are many interesting projects now based on EMF. One particularly interesting project is the Graphical Modeling Framework, GMF [11], where graphical editors can be generated from EMF models. Figure 8 actually shows a screenshot of a TGG rule in the generated graphical GMF-editor.

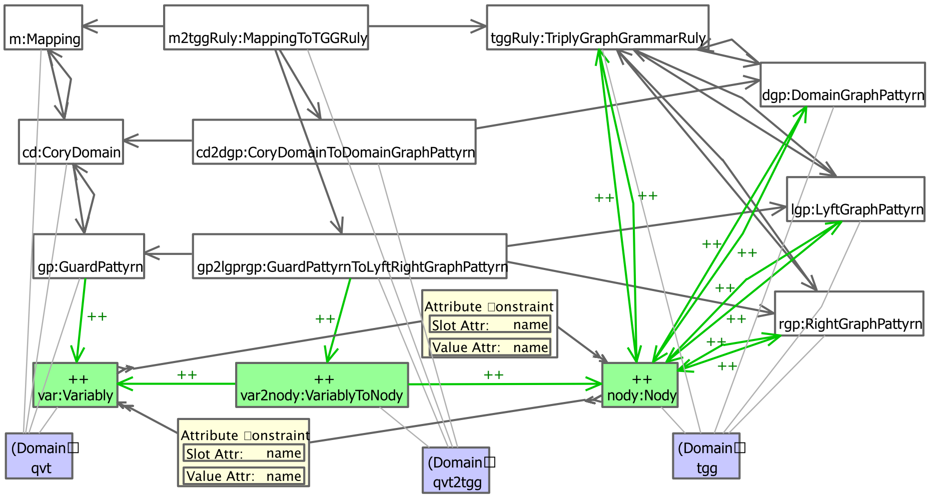

Fig. 8. Mapping the constructs of QVT and TGG transformation rules

Our TGG engine supports the interpretation of the TGG rules and is thus also called *TGG interpreter*. The integration of OCL is not yet completed, so that for now, only simple attribute value constraints are supported.

The transformation from TGGs back to QVT-Core is only possible under certain conditions. The backward mapping works if, firstly, we can assume that QVT-Core supports multiple trace variables in the bottom pattern. Secondly, because TGG rules do not specify if any domain is enforceable or not, the resulting QVT-Core would need to consider every domain area to be enforceable. A backward mapping from QVT to TGGs could nevertheless be interesting to evaluate advantages or disadvantages between the TGG interpreter and upcoming implementations of QVT-Relations or QVT-Core.

4 Comparing Concepts of QVT and TGGs

In the previous sections, we have discussed the basic concepts of QVT and TGGs. In this section, we relate the concepts of QVT and TGGs and discuss how the concepts of both technologies could benefit from each other.

The starting point of our work was that there are striking similarities and analogies between QVT and TGGs. We have seen in Section 3 that the basic structure of QVT and TGGs coincide and the rough meaning of these concepts are the same. But, there are some more or less significant differences, on the technical, the conceptual, and the philosophical level. Firstly, we discuss some philosophical differences. Then we compare the definitions of the semantics and some conceptual differences. In the end, we discuss some advanced features.

4.1 Philosophical Differences

We start with a discussion of two philosophical differences between QVT and TGGs. Firstly, there is a difference in how a QVT mapping and a TGG rule are read: QVT mappings are read in the direction from "bottom to top", whereas TGG rules are read from "top to bottom". In particular, this means that, in QVT-Core, we first have a look at the bottom patterns of the mapping, which must match on both domains when the guard pattern also match. The semantics of a TGG rules is defined in the other direction. We start with matching the left-hand side (top part) of the rule; only when a match is found, the right-hand side of both domains (bottom part) is considered. Of course, the real execution of a TGG rule is driven by the existing parts of a model (and the application scenario); but, conceptually and in the implementation, TGG rules are applied from top to bottom. Though, this might appear as a minor issue, it reflects a different way of thinking of QVT mappings and of TGG rules. In particular, this is reflected in the fact that QVT-Core may have mappings with empty guard patterns, which means that they do not need a particular context to be applied. These mappings can be used to provide the initial bindings, which serve as context for the application of further mappings. The counterpart of such QVT start mappings in TGGs is the axiom, which provides the start context for applying further rules in TGGs. The difference is that QVT can start with several start mappings in different parts of the model, whereas the matching of a TGG always starts with a single axiom.

Secondly, there is a fundamental difference in the way QVT and TGGs are applied in a concrete transformation scenario. In QVT, the application scenario, i.e. whether we perform a consistency check only or a transformation in one direction or the other, is partly encoded in the QVT mappings. The directives *check* and *enforce* within a QVT mapping state in which domain patterns will be created, deleted, or modified. Therefore, the QVT mappings are written with an application scenario in mind. By contrast, TGG rules do not refer to the application scenario. The same set of TGG rules can be used for checking consistency, for transformations in either direction, as well as for synchronizing both models after they have been changed independently of each other. The benefit is that we have a single set of TGG rules with the same semantics for all application scenarios—the only thing changing is the application scenario itself. This guarantees that the models are and remain consistent with respect to the single TGG specification, even when switching the application scenario.

This second philosophical difference has an important implication. In QVT, there is no explicit scenario which synchronizes two models that have been changed independently of each other. This can be achieved only by two subsequent incremental transformations in both directions[4]. By contrast, TGGs can be interpreted to synchronize two models in both directions in a single run at the same time.

4.2 Semantical Comparison

In Section 4.1, we have discussed the philosophical differences between QVT and TGGs already. Next, we ask a more technical question concerning the semantics of

[4] Actually, most of today's synchronization technologies use this approach; they incrementally transform the changes of one model into the other and vice versa subsequently.

QVT and TGGs. Let us assume that we have two models which are in the relation specified by the QVT mappings, resp. TGG rules. Now, we have a set of applications of these rules which prove that the models are in relation. Then, each variable or node is mapped or *bound* to the actual model elements. Now, let us consider these bindings in the opposite direction.

In TGGs, every object in the two models has exactly one binding to a *creating* node of an *applied* TGG rule (i.e. an application of a TGG rule where the node occurs on the right-hand side of this rule only). This results from the definition of the semantics of TGG rules which generates legal pairs of corresponding models. Conceptually, these pairs of models are defined by generating pairs of corresponding models starting from the axiom by applying some TGG rules: This way, both models are generated simultaneously. In the end, each object of the two generated models corresponds to exactly one creating node (right-hand side only) in an application of a TGG rule and vice versa. For more details, see [12]. In practical cases, however, we may want to describe only the relation between parts of the models. Therefore, the TGG rules will cover only these relevant parts and the rest of the models will be ignored. In these cases, there is exactly one binding of the relevant model objects to a creating node of an application of a TGG rule and there is no such binding for a model object which is not considered by the TGG.

The QVT standard, in contrast, does not make it clear whether there is at most one binding of each model object to a variable in a bottom pattern of exactly one application of a QVT mapping. Some explanations in the standard as well as the examples suggest that this is true and, thus, the interpretation seems to be very similar to the TGG interpretation. But, the mathematical formalization does not guarantee that. However, under the assumption that, at least for enforced variables, QVT also implies such a one-to-one correspondence between the relevant objects of the model and variables in the bottom patterns, the semantics of QVT-Core mappings can be mapped to TGG rules as discussed in Section 3. We consider this an open issue and a final justification of this assumption will have to be discussed.

This assumption is however supported by other considerations which concern verification. As pointed out in [13] and proved in a case study [14], TGGs can be used for verifying the semantical correctness of models that are transformed by TGGs. This kind of verification is possible, because of the one-to-one semantics of TGGs, which nicely reflects the definition of a semantics in SOS-style (structural operational semantics).

4.3 Conceptual Differences

In addition to the philosophical differences, there are some conceptual differences between QVT and TGGs. But, we will see that these can be easily aligned.

Though there is a graphical notation for QVT, QVT is conceptually more closely related to defining a set of *variables* and the definition of relations among these variables. These relations are defined in terms of OCL *expressions* and *assignments*. This makes QVT very flexible and expressive—due to the expressive power of OCL. By contrast, TGGs are graphical in nature and originate from the realm of graph grammars and graph transformations. A model is considered as a graph with *typed* *nodes* and edges between them. On the one hand, this results in a simple, precise and intuitive semantics (see Section 4.2); on the other hand, this imposes some restrictions. Some restrictions have been overcome by introducing different kinds of expressions to different variants of TGGs. Most of these extensions are straightforward, but not quite in the spirit of TGGs. However, inspired by QVT, we have shown how to introduce OCL constraints to TGGs without spoiling the "spirit of TGGs" and their graphical nature [12]. By equipping TGGs with OCL constraints, the essence of a relation between two models can be specified in a graphical way; still the expressive power of OCL is at hand when necessary. In particular, we do not need to distinguish between (querying) expressions and assignments in TGGs.

As mentioned above, QVT has the concepts of *check* and *enforce*. In fact, these keywords occur in two quite different ways in the QVT specification. So they actually constitute two different concepts. The first concepts defines the *mode* of application, i.e. it defines whether the mapping should be interpreted as a consistency check, or as a transformation in different directions, or as a synchronization. In TGGs, we call this the *application scenario*. So, *mode* and *application scenario* are just two different names for essentially the same concept. In QVT, however, *check* and *enforce* also occur within a mapping—with a similar but still different meaning. Some areas can be marked with `check` and `enforce`. The idea of check and enforce in this context is that, during a transformation, an existing object of a model can be reused in such a mapping. Only if this object does not exist, it will be created. This increases the efficiency in transformations where parts of models already exist. Though there are many different extensions of TGGs, the basic form of TGGs does not have such a concept of *reusable nodes*. A node is either required (when it occurs on the left-hand side of the rule) or it is created (when it occurs on the right-hand side only). Based on the practical experience with TGGs and inspired by QVT, we have introduced a concept of *reusable nodes* to TGGs. There meaning is that these nodes can be reused if a node with the required properties already exists. Note that reusable nodes in TGGs do allow to reuse nodes, but they do not require their reuse. If we want to force the reuse of a node, this needs to be specified by additional global constraints, which will be explained shortly. Altogether, the concept of check and enforce of QVT can be expressed in TGGs by the concepts of reusable nodes and global constraints. Since reusable elements are identified on the level of individual nodes, TGGs can express this concepts on a finer level of granularity.

As mentioned above, we also introduced the concept of *global constraints* to TGGs. A global constraint allows us to enforce that a node with specific properties exists at most once in a model. This way, the reuse of a node can be enforced. Note that a global constraint is very similar to the concept of *keys* of QVT-Relations, which is used to uniquely identify an object by some of its attributes. This guarantees that a transformation does not generate duplicates of objects that have the same key.

A last difference between QVT and TGG is of technical nature: QVT was proposed in the context of the MDA. Therefore, QVT is defined based on MOF and uses the underlying concepts. TGGs were introduced long before the existence of MOF, and there are different implementations for different technologies—independently from MOF. But, there are implementations based on the Eclipse Modeling Framework (EMF), which in turn is an implementation of MOF.

4.4 Advanced Features

In addition to the differences discussed above, there are some advanced features of QVT and TGGs that are briefly discussed here.

First of all, QVT-Core allows to nest mappings within mappings. According to the QVT-Specification [2], this nesting of mappings helps to avoid inefficient and iterated deletion and creation of objects. Thus, nested mappings do not increase the expressive power of QVT mappings, but do increase the efficiency of their application. By contrast, it is not suitable to adopt the concept of nested rules to TGG rules. As pointed out before, this would not increase the expressive power, but only the efficiency. Efficiency, however, is an implementation matter. In TGGs, efficiency might be achieve in a simpler way, due to the top down interpretation from a single start context. Still, TGGs can be designed in such way that an implementation works efficiently. One of the main concerns for efficiency is to reduce the size of each individual TGG rule (because applying a rule basically means applying a graph-matching algorithm between this rule and the models).

A useful feature in QVT-Core is the refinement of rules by others, similar to inheritance in object orientation. This is a feature which is not yet present in TGGs. We feel that this is useful for better maintaining sets of TGGs and for making them more understandable. But, this can be built on top of the concepts of TGGs and it is not necessary to make this a core feature of TGGs.

As pointed out earlier, TGGs allow many different correspondence nodes in a single TGG rule. In QVT is seems to be possible to use more than one trace node in the bottom pattern, but it does not seem to be strongly encouraged. In TGGs, multiple correspondence nodes allow us to keep track of relations between individual model elements in a very detailed way. This helps to design TGG rules in a local way, which in turn results in simpler and smaller TGG rules. We identified examples where this clearly reduces the number and size of rules. In this way, multiple correspondence nodes compensate the efficiency of nested mappings in QVT. But, they do more: they also help us to design clearer and better understandable TGG rules.

5 Conclusion

In this paper, we have discussed the similarities of QVT and TGGs. Due to the similar structure and concepts, QVT mappings can be transformed into TGG rules. This way, a TGG engine can execute transformations specified in QVT-Core.

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. The insights gained here have inspired the extension of TGGs and might provide valuable input for QVT as a standard. The goal is to have a clear and simple semantics to support a straightforward employment of model transformation technologies and to facilitate the use of validation and verification techniques.

Furthermore, it could be interesting to inspect the relation between QVT-Relations and TGGs. QVT-Relations provides more structure in order to simplify and better organize a set of transformation rules We believe that this could result in some concepts on top of TGGs, which could improve the comprehensibility of TGGs. Also we believe that there are additional concepts needed for the efficient synchronization of models, which go beyond transforming back and forth. This will need a closer investigation in the future.

References

1. Object Management Group (OMG): Model Driven Architecture - A Technical Perspective (July 2001) (last accessed 2^{nd} of April 2007), `http://www.omg.org/docs/ormsc/01-07-01.pdf`
2. Object Management Group (OMG): MOF QVT Final Adopted Specification (November 2005) (last accessed 2^{nd} of April 2007), `http://www.omg.org/docs/ptc/05-11-01.pdf`
3. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, Springer, Heidelberg (1995)
4. Wagner, R.: Developing Model Transformations with Fujaba. In: Proceedings of the 4th International Fujaba Days 2006, Bayreuth, Germany, pp. 79–82 (September 2006)
5. The MOFLON Tool Set (last accessed 30^{th} of June 2007), `http://www.moflon.org`
6. Greenyer, J.: A Study of Model Transformation Technologies - Reconciling TGGs with QVT. University of Paderborn, Department of Computer Science, Master/Diploma thesis (July 2006)
7. Gepting, A., Greenyer, J., Kindler, E., Maas, A., Munkelt, S., Pales, C., Pivl, T., Rohe, O., Rubin, V., Sander, M., Scholand, A., Wagner, C., Wagner, R.: Component Tools: A vision for a tool. In: Kindler, E. (ed.) Algorithmen und Werkzeuge für Petrinetze (AWPN) - Algorithms and Tools for Petri nets. Proceedings of the Workshop AWPN, September 30th, October 1st 2004, pp. 37–42 (2004)
8. Königs, A., Schürr, A.: MDI - a Rule-Based Multi-Document and Tool Integration Approach Special Section on Model-based Tool Integration in Journal of Software&System Modeling. Academic Press, San Diego (2006)
9. Rohe, O.: Model Transformation by Interpreting Triple Graph Grammars: Evaluation and Case Study. Bachelor thesis, University of Paderborn (January 2006)
10. The Eclipse Project: The Eclipse Modeling Framework (last accessed 2^{nd} of April 2007), `http://www.eclipse.org/emf/`
11. The Eclipse Project: The Graphical Modeling Framework (last accessed 2^{nd} of April 2007), `http://www.eclipse.org/gmf/`
12. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report, University of Paderborn, Department of Computer Science (June 2007)
13. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards Verified Model Transformations. In: MoDeV^{2}a. Proceedings of the 3rd International Workshop on Model Development, Validation and Verification, Genova, Italy, pp. 78–93. Le Commissariat à l'Energie Atomique - CEA (October 2006)
14. Leitner, J.: Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken. Diploma thesis, University of Karlsruhe/TU Berlin (March 2006)