A Study of Model Transformation Technologies:
Reconciling TGGs with QVT

Diploma Thesis by Joel Greenyer
(jgreen@mail.uni-paderborn.de)

supervised by Dr. Ekkart Kindler
and Robert Wagner

submitted to Dr. Ekkart Kindler
and Prof. Dr. Gregor Engels

Version date: July 15, 2006
# Table of Contents

1 Introduction

1.1 Redesign of the TGG Transformation Technology

1.1.1 Redesign of the TGG Metamodel

1.1.2 Redesign of the TGG-Interpreter

1.2 Mapping QVT to TGGs

2 Comparing QVT and TGGs

2.1 The example

2.2 Comparing QVT and TGG rules

2.2.1 Patterns and Contexts

2.2.2 Start Rules vs. Start Context

2.3 First Comparison Summary

3 QVT

3.1 Transformations

3.2 Relations and Mappings

3.2.1 Execution Direction

3.2.2 When- & Where-Clause

3.2.3 Top-level Relations and Top-level Mappings

3.2.4 Relations to Mappings

3

7

8

9

10

12

13

17

17

20

21

23

23

24

25

26

29

29
3.2.4.1 Schematic Structure of a Relation (QVT-Relational) ................................................. 29
3.2.4.2 Schematic Structure of a Mapping (QVT-Core) .................................................... 30
3.2.4.3 Mapping Relation Patterns to Mapping Patterns .................................................. 31
3.2.5 Patterns and Predicates .............................................................................................. 32
  3.2.5.1 Domain Patterns in QVT-Relational ................................................................. 34
  3.2.5.2 Transformation of Checkonly Domain Patterns to QVT-Core ...................... 35
  3.2.5.3 Transformation of Enforceable Domain Patterns To QVT-Core ................. 36
  3.2.5.4 Keys ...................................................................................................................... 37
  3.2.5.5 Transforming Relation Calls to QVT-Core Mappings .................................... 38
3.3 A model transformation processing scenario ................................................................. 40
3.4 Summary ....................................................................................................................... 44

4 TGGs ................................................................................................................................ 45
  4.1 Introduction to Triple Graph Grammars ................................................................... 45
  4.2 TGGs and Model Transformations ........................................................................... 50
    4.2.1 Transformations of Object Graphs ................................................................. 50
    4.2.2 Attribute value transformation ......................................................................... 51
    4.2.3 Check & Enforce ............................................................................................... 53
    4.2.4 Matching Instances of Subclasses ................................................................. 54
    4.2.5 Multi Graph Grammars .................................................................................... 54
    4.2.6 Start nodes ........................................................................................................ 55
    4.2.7 Further TGG Features ...................................................................................... 55
      4.2.7.1 Negative and Optional Pattern Elements ................................................... 55
      4.2.7.2 Complete and Partial Grammars ............................................................... 57
    4.2.8 Verification and Transformation Synthesis ..................................................... 59
  4.3 A model transformation processing scenario ............................................................... 59
  4.4 Comparing the QVT and TGG processing scenario .................................................. 62
  4.5 Summary ....................................................................................................................... 63

5 TGG-Interpreter Redesign ............................................................................................... 65
  5.1 TGG Model Redesign ................................................................................................. 65
    5.1.1 Graph patterns and MOF models .................................................................... 65
      5.1.1.1 Typed Graphs Prior to the Redesign ............................................................ 67
    5.1.2 The TGG rule pattern structure ....................................................................... 68
    5.1.3 Specifying Transformations with Triple Graph Grammars ................................ 69
      5.1.3.1 TGG Transformations Prior to the Redesign .............................................. 70
    5.1.4 Attribute Constraints ......................................................................................... 70
    5.1.5 TGGs and OCL .................................................................................................. 73
1 Introduction

Single software tools are typically developed for a certain application domain and, thus, are working on structured data, their model, that suits the requirements of this domain best. When integrating software tools or information systems that work on proprietary domain models, transformations between these models become necessary. Today, software development is increasingly based on technologies where conceptual and platform independent models and metamodels play a centralized role. Such model driven technologies are utilized not only to develop single software components, but to integrate these components into more complex software systems. The Object Management Group (OMG, [OMG]) presents an approach to separate the conceptual application model from the underlying technologies, called Model Driven Architecture (MDA, [MDA]). Developing software at this more conceptual level has many advantages as the domain models become more and more complex. But, this also raises the requirements to specify transformations between these models at the more abstract modeling level. Therefore, the OMG is developing the QVT specification (Query/Views/Transformations, [QVT]) to standardize transformations of models [MOF]. The finalization of this specification can be expected during this year, 2006.

At the University of Paderborn, in the context of two projects of the Software Engineering Group, Fujaba Tool Suite [Fujaba] and ComponentTools [AWPN04][KRW05], model transformation technologies have been developed based on Triple Graph Grammars (TGGs, as introduced by Schürr [Sch94]). Triple Graph Grammars are a convenient formalism to specify model transformations. For example is it possible to specify bidirectional transformations and the graph based approach as well as the graphical syntax are a native way to specify transformations between object graphs. Furthermore, the graph grammar formalism provides access to a rich theory of verification methods like the analysis of the confluence of transformation rules. Ongoing work at the University of Paderborn is, for example, also investigating how a Triple
Graph Grammar can be synthesized from a set of related example source and target models [Geb06].

In the context of MDA, the QVT specification is the result of comparing, selecting, and merging different model transformation methods and technologies, considering a number of mostly practical requirements. Amongst others, these are performance, usability and conformance to or use of other MDA related standards. To investigate which improvements and extensions to existing TGG transformation technologies are necessary for a more efficient use in practice, this thesis is comparing TGGs with the upcoming QVT specification. QVT specifies an imperative as well as two declarative languages. As TGGs are a declarative transformation formalism, only the declarative languages will be considered in this thesis.

Although the formalism suggested by QVT is conceptually different from TGGs, a first rough comparison displays many similarities between TGGs and QVT. These similarities and the fact that QVT is yet rarely supported by tools leads to the consequent question to which extend the QVT specification can be implemented by an improved TGG transformation technology (as already suggested by Küster, Sendall and Wahler [KSW04]). A mapping from QVT to TGGs would therefore not only leverage improvements to Triple Graph Grammars, but would also provide QVT with an implementing technology and a graphical notation. Furthermore, formal methods from graph grammar theory would then become accessible to QVT.

Therefore, this thesis is addressing the first key question, how to improve the TGG technology, by a redesign of the existing solutions at the University of Paderborn [Fujaba][KRW05]. The second key question, how QVT can be mapped to TGGs, is then answered by providing a formal mapping between QVT and TGGs in form of a TGG transformation implemented in the previously redesigned technology.

1.1 Redesign of the TGG Transformation Technology

Transformation technologies based on Triple Graph Grammars have been developed in the context of two projects at the University of Paderborn. The first project, the Fujaba Tool Suite [Fujaba], is a CASE tool that utilizes TGGs to transform models to models and models to code. In Fujaba, the transformation rules are compiled to Java code that performs the transformation.

The second project, ComponentTools [AWPN04][KRW05] uses TGGs to transform [Roh06] an abstract component model into formal models . These formal models are then used to perform further tasks like the verification or simulation of the component system. For example, Petri nets are used as a basis for simulation that is visualized in 3D [KP04]. In another application, finite automata are used for controller synthesis [Sch06] or verification through model checking. Different from Fujaba, ComponentTools performs the transformation by interpreting, rather than compiling the Triple Graph Grammars.
1.1.1 Redesign of the TGG Metamodel

Both TGG transformation tools mentioned above were considered to be improved and extended in order to map QVT to TGGs. To permit this mapping, there are a few criteria that will have to be met. Primarily, just as QVT, TGGs will have to specify transformations of models compliant to the *Model Object Facility* specification (MOF [MOF]).

MOF is another OMG specification that standardizes the format of models and metamodels as a foundation of MDA. It is crucial for the further reading of this thesis to understand the different meta levels of MOF. Thus, these meta levels are briefly explained in the following (see figure 1): The instance data is found at the instance- or M0-level. Here, for example the in-memory objects of a Java program or database entries are located. As the instance data can also be considered as a model in a broader sense, this thesis is referring to a model at this level as the *instance model*. Models at the M1-level can be, for example, class- or ER diagrams. They are referred to simply as *models* or *class models*. Then, models of modeling languages as UML, which are at the M2-level, will be called *metamodels*. On top of this, yet another meta level to describe modeling languages themselves is the M3-level. This is actually the self-specifying *meta-metamodel* or the *MOF model*. Additionally, every model at any level that can be described by MOF is generally called a *MOF model*.

![Figure 1: The MOF meta levels](image)

Now, when QVT transformation rules are specified with respect to MOF models, the same has to apply to TGG transformation rules. In detail, the Fujaba and ComponentTools model transformers and their underlying TGG models were reviewed under the following aspects:

1. In order to transform MOF compliant instance models, the corresponding MOF class models are referenced by the TGG rules.
2. The TGG model is extendable to add further language constructs in the future.
3. The model allows a performant rule processing.
4. The model provides an adequate structure to map QVT transformations to it.
Both existing tools are not providing sufficient flexibility and extendability considering these aspects. Though the Fujaba TGG model is based on the quasi MOF compliant Fujaba metamodel, the TGG model was not extendable enough and did not provide an adequate structure to provide a mapping from QVT to it. The ComponentTools TGG metamodel did neither have rules that were referencing MOF compliant models, nor did this TGG model provide an extendable and structurally appropriate starting point. The most evident structural limitation to these two TGG models is that they only allowed to specify transformation between two models (1:1), as originally intended by TGGs. But, this limitation is not necessary and an approach closer oriented on QVT rather needs a concept that allows the specification of n:m model transformations. Consequently, this thesis provides a new TGG metamodel as a redesign of the previously mentioned TGG model from the ComponentTools projects. A major improvement is that now transformations can be specified between multiple models. So, TGGs are actually extended to \textit{Multi Graph Grammars}, a term recently introduced by Königs and Schürr [KS06a].

The MOF standard mentioned above is again an implementation independent specification. A MOF implementation will have to be chosen as a basis for the TGG model. There are a few technologies implementing MOF or certain parts of it. The most popular and widespread MOF implementation today is The Eclipse Modeling Framework (EMF [EMF]) with its ECore model. This implementation is chosen, in addition to its popularity, for the following reasons: Firstly, EMF provides many convenient utilities for model driven applications. Secondly, it is chosen because many other projects already exist that facilitate model driven development tasks through EMF. In this thesis, for example, GMF [GMF] was used to deploy a convenient graphical TGG editor in a very short time. A third argument for EMF is that many projects at the University of Paderborn, for example the ComponentTools project, are integrated in Eclipse and based on EMF models.

1.1.2 Redesign of the TGG-Interpreter

This TGG model redesign also implicates the redesign of the existing TGG model transformer implementation. The TGG-Interpreter from the ComponentTools project was used as an archetype, because the interpreted approach, though potentially slower, provides a more lightweight software component that can more easily be integrated in the future. There are many arguments concerning the interpreted vs. compiled transformation approaches that shall not be discussed here. A long term vision would be to provide both the interpreted and compiled solution which could be exchanged for different areas of application.

In the following, figure 2 shows an abstract overview of the architecture of the redesigned TGG interpreter.
Figure 2: The redesigned TGG-Interpreter architecture when transforming MOF models

Figure 2 shows the TGG interpreter when transforming one MOF instance model into another. To describe how one instance model relates to another or how it shall be transformed into another, rules have to be specified in terms of their class models. Without this meta description of the data, it would not be possible to generically talk about model structures or their relationships. So, the TGG rule references the class models and then, the interpreter transforms the instance models according to the supplied TGG rules.

A valuable feature of the TGG-Interpreter from the ComponentTools project is that arbitrary non-MOF Java instance models, even without providing a proper (meta-)model, can be transformed by implementing specific model adapters [FD04]. In the case where other kinds of non-EMF Java models need to be transformed by the redesigned interpreter, adapters will have to be used to wrap these instance models as EMF models which the interpreter will be able to interpret (see figure 3). Also, the extraneous model, which may be available or not, needs to be mapped to an EMF model to provide a basis to design the TGG transformation rules upon.

Figure 3: The TGG-Interpreter architecture when adapting arbitrary Java models
1.2 Mapping QVT to TGGs

After the redesign of the TGG model, this thesis shows how QVT can be mapped to TGGs. QVT provides two declarative languages which are candidates for this mapping. The first language is the more abstract and user friendly QVT-Relational. Transformations specified in QVT-Relational will be transformed into the second language, QVT-Core, which is more specific and then used to perform the transformation. Both languages, QVT-Relational and QVT-Core can be mapped to TGGs as this thesis will show. The primary focus, however, will be the mapping from QVT-Core to TGGs. Figure 4 illustrates the mapping relationships.

![Figure 4: Mapping QVT to TGGs](image)

The left hand side of the figure shows, how QVT-Relational is mapped to QVT-Core. This mapping is provided by QVT, so that mapping QVT-Relational to TGGs can be considered as transitively accomplished by mapping QVT-Core to TGGs. However, this thesis also provides an approach how TGG-Relational can be mapped to TGGs directly. A further idea, which is discussed as an outlook of this thesis, is to add another, more abstract TGG language on top of TGGs, here referred to as TGG++. This could provide a more user friendly abstraction layer in the future, similar to QVT-Relational for QVT.

Both declarative languages of QVT use the Object Constraint Language (OCL [OCL]) to specify the relation of object structures and additional constraints. OCL allows to formulate complex expressions and, therefore, QVT is more expressive than the prior TGG specifications as well as the redesigned TGGs in this thesis. Thus, only parts of QVT will be mapped to TGGs. Many expressions are not originally intended to be supported by TGGs, because they may violate the bidirectionality of a transformation. Furthermore, complex constructs will not be verifiable anymore by methods from graph grammar theory. However, an approach is presented how OCL could be integrated into TGGs to obtain an equal expressive power.

This thesis is structured as follows. Firstly, the next chapter introduces and compares QVT and TGG transformation rules along a simple example. Then, the structure and semantics of QVT and TGGs are explained in chapter 3 and 4. Based on this description and the comparison of QVT and TGGs, the redesigned TGG model, the graphical TGG rule editor and the architecture of the implemented TGG interpreter is explained in chapter 5. At last, chapter 6 is presenting a transformation from QVT-Core to TGGs as well as an approach to transform QVT-Relational to TGGs.
2 Comparing QVT and TGGs

This chapter briefly compares QVT and TGGs with the help of a simple example to display the major similarities and differences of the two model transformation approaches. A popular example from the MDA context is the transformation from UML class diagrams to relational database schemas. However, this thesis introduces another example from an application of the ComponentTools project. There, abstract components of a material flow system are transformed into formal models, for example Petri nets. Further applications like simulation or verification of the component system are then performed based on these formal models.

In the following, firstly, the example transformation case is introduced. Then, the QVT and TGG rules for this transformation are compared, characterizing the apparent similarities and differences. A detailed discussion of the language constructs and their processing, is then given in chapters 3 and 4.

2.1 The example

The example presented here is a transformation of a component model into a Petri net (see figure 5). In ComponentTools, a component system is drawn by placing and connecting components in a Project. The left side of figure 5 schematically shows a small component system which consists of two components, simple Tracks in this case.
These Tracks and the Connection that connects the components’ incoming and outgoing Ports is transformed into the corresponding Petri net construct shown on the right side. In the Petri net, there are Places and Transitions which are connected by Arcs.

Now, generically a rule to transform a single Track component into its corresponding Petri net construct is schematically shown in figure 6.

Such a rule is called a relation because model elements are related to each other. In this sense, both QVT and TGG rules can be considered relations. This relation states that a Track with its two Ports relates to a Place, Arc and Transition, provided that the specified context is given. Here, the context requires that the Track is a child element of a Project and that the Place, Arc and Transition are children of a Petri net. A context is typically provided by other relations or certain axioms.

Now, these relations can be applied as transformation rules. Figure 7 shows how the first Track from the above example component system (figure 5) is transformed according to the relation in figure 6.
Figure 7: First application of a rule to transform a Track

Here, a Place Arc and Transition were created as child objects of the Petri net. The same rule will be applied a second time to transform the second Track as shown in figure 8.

Figure 8: Second application of a rule to transform a Track

Then, there would be another rule, which is omitted for now, to transform the connection. In addition to the scenario above, there are further uses for such relations. They can also be applied to transform in the backward direction or to check if there is a valid correspondence between two given models. For now, however, only the above transformation scenario shall be considered.

Both QVT and TGG rules can be considered relations. QVT and TGG transformation rules are presented and characterized in the following, but before, the example domain models are more closely examined.

Firstly, the simplified component model (called ctools) is shown in figure 9.
The ctools model has the Project class as its root that contains Components and Connections. The Component may have a number of Ports which serve as incoming or outgoing attachment points to the Connections. One specific component is the Track. Not included in this diagram is the common superclass NamedElement, which provides a String attribute "name" to all shown subclasses. Also note that all associations in this model are bidirectional.

The simplified Petri net model (called pnet) shown in figure 10 has the Petrinet class as its root that contains Places, Transitions as well as Arcs. Arcs can connect Places with Transitions, but in this simplified model, not even a direction can be specified. The associations between the Petrinet class and its contained classes are bidirectional, but the Places and Transitions do not know about their connecting Arcs.

Figure 11 shows two object diagrams of the related domain model elements according to the class models displayed in figures 9 and 10.
Both QVT and TGG transformation rules specify and relate patterns similar to object diagrams. These patterns in the transformation rules are therefore also called *object patterns*. The QVT and TGG transformation rules are compared in the following.

## 2.2 Comparing QVT and TGG rules

As previously mentioned, QVT specifies two declarative languages at two levels of abstraction. QVT-Relational is more elementary and user friendly and is mapped to the more technically specific language QVT-Core. Both QVT-Relational and QVT-Core are text based languages, but QVT also specifies a graphical syntax for QVT-Relational. For an easier recognition of the object patterns from the example above, this section introduces graphical notation of QVT-Relational first. Following sections of this thesis will then rather refer to the more concrete textual representation.

### 2.2.1 Patterns and Contexts

To display the similarities between QVT and TGG, the transformation rule as previously introduced is compared in both formalisms. A QVT *relation*, as a single transformation rule is called in QVT-Relational, is show in the following figure 12.
As seen in this figure, a QVT relation specifies how parts in the ctools and pnet instance model relate to each other. Objects that reference each other in the instance model can be represented as graphs and the graphical notation that is used here to represent object patterns is similar to UML object diagrams. The overall semantic of this relation is that when a pattern on one side is found in the one model, then the other pattern has to exist in the other model.

The patterns are described starting from a single, central domain element. The following listing of the textual representation will illustrate the pattern structure and notation:

```
top relation TrackToPlaceArcTransition{
    enforce domain ctools track:Track {
        componentToProject=project:Project{},
        componentToPort=inPort:Port{},
        componentToPort=outPort:Port{}
    }
    enforce domain pnet arc:Arc {
        arcToPetrinet=petrinet:Petrinet{},
        arcToPlace=place:Place{placeToPetrinet=petrinet},
        arcToTransition=transition:Transition{transitionToPetrinet=petrinet}
    }
    when{
        ProjectToPetrinet(project, petrinet);
    }
}
```

This listing shows that the patterns are hierarchically structured starting from the domain element (Track in ctools and Arc in pnet). These patterns are described through predicates that use OCL to formulate object pattern conditions. The use of OCL allows QVT to become fairly expressive.

In QVT-Relational, relations can refer to other relations. In this example, the TrackToPlaceArcTransition relation is referring to the ProjectToPetrinet relation through a when clause, handing the project and petrinet pattern elements as

---

1 A more detailed explanation on the semantics of QVT is given in chapter 3.
2 This listing is slightly simplified. See the issue on QVT constructs discussed in section 6.3, pp. 109.
parameters. The ProjectToPetrinet relation, for example is a start rule that relates a Project to a Petrinet at the top level. So, when applying the TrackToPlaceArcTransition rule to existing model instances, the Project and Petrinet model elements also need to satisfy the relation conditions specified by the ProjectToPetrinet relation.

![Figure 13: The QVT relation ProjectToPetrinet](image)

Before giving a further description of the QVT relations, the corresponding TGG transformation rule is shown in figure 14.

![Figure 14: The TGG rule TrackToPlaceArcTransition](image)

On first sight, the Triple Graph Grammar rule looks quite similar to the QVT relation. Again, the rule relates one pattern in the one domain model to another pattern in the other domain model. Actually, all pattern elements seen in the QVT relation reappear, though differently arranged, in this rule. In contrast to the OCL pattern descriptions in QVT, TGG rules establish the patterns by an actual graph model with nodes and edges. Roughly, this TGG rule can be understood in the following way: Whenever a Project and Petrinet object were matched by a previous rule, then, for every Track of this Project, there has to exist a corresponding Petri net construct in the Petrinet object.

Essentially different, as shall be shortly mentioned here, are the following two characteristics. Firstly, there are no explicit references to other TGG rules, in contrast to the when-clause of the QVT relation. Here, these references are rather implicit. In terms of grammars, there are nodes that belong to the context side, or left-hand side, of the rule. For these left-hand side nodes, previous matches from other rules have to exist in order to apply the current rule. In this regard, another notation for the same TGG rule is the following (see Figure 15).
In this notation, the context nodes are filled white and the nodes that are intended to be translated by this rule are marked with "++" in addition to a green filling. This notation is a special way to write Triple Graph Grammars. The coloring of the nodes make the rules easier to read when they become more complex. The detailed semantics of these rules will be discussed in chapter 4.

The second different characteristic compared to QVT relations are the correspondence nodes that provide an explicit mapping between the domain pattern nodes. For example, the correspondence nodes seen in the rule above provide a fine-grained mapping between the track, its ports and the corresponding Petri net constructs. A counterpart to these correspondence nodes cannot be found in QVT-Relational, but does exist in QVT-Core. There, trace classes provide a very similar functionality. Chapter 3 provides a more thorough description of both QVT-Relational and QVT-Core and also discusses the mapping between these two language levels.

### 2.2.2 Start Rules vs. Start Context

Both the QVT and TGG transformation rules shown above refer to a certain context in which the model elements effectively relate to each other. In the case of the above QVT relation shown back in figure 12, another relation, ProjectToPetrinet, is explicitly referenced. This relation is shown back in figure 13.

This simple rule relates the root model objects and does not define any dependencies to other rules.

```plaintext
top relation ProjectToPetrinet   // map each project to petrinet
{
    enforce domain ctools project:Project {}; 
    enforce domain pnet petrinet:Petrinet {}; 
}
```

Therefore, this rule can be seen as a starting rule of the transformation. Triple Graph Grammars, however, as formal grammars in general, need a start symbol or start context prior to the rule application to start the transformation. Figure 16 schematically shows the example of a start context where a Project-, correspondence- and Petrinet node are already bound to their respective model objects (indicated by the underlying gray boxes).
2.3 First Comparison Summary

A first comparison of QVT and TGGs shows many similarities in the structure of the single rules as well as in how the rules correlate to describe a transformation.

- **Patter/graph based**: QVT-Relational and QVT-Core as well as TGGs describe object patterns, in source and target domain models.
- **Declarative**: QVT and TGG rules declaratively relate domain model patterns to each other.
- **Contextual**: Both QVT and TGG rules specify certain contexts under which the relation is valid.
- **Trace classes vs. correspondence nodes**: TGGs relate model patterns with explicit correspondence nodes. Such constructs do not exist in QVT-Relational, but the similar concept of trace classes is found in QVT-Core.

There are also apparent differences that could be observed. Although there are many structural similarities, the representation of object patterns is essentially different. Also, some language features and the overall rule processing approach seems different. The most important differences are the following:

- **OCL predicates vs. graph model**: QVT uses predicates in OCL to describe the domain model patterns opposed to a native graph model with nodes and edges in TGGs.
- **Relation calls vs. context nodes**: QVT and TGG rules refer to a previous context. But, QVT relations call other relations with parameter variables. TGG rules, in contrast, mark certain nodes as context nodes.
- **Start rule vs. start context**: In QVT, there are rules that do not refer to a previous context. These rules can be used to start a transformation. Such rules do not exist in TGGs. Rather, a start context needs to be provided as the starting point of a transformation.

Despite the differences, the similarities listed above allow the assumption that a mapping between QVT and TGGs, or certain subsets of these languages, is possible. This mapping will be thoroughly illustrated in chapter 6. But, before, a more detailed description is given of the declarative QVT languages in chapter 3 and TGGs in chapter 4.
3 QVT

In the Final Adopted version [QVT], QVT specifies two declarative transformation languages, QVT-Relational and QVT-Core. Also, an imperative language is specified, QVT-Operational. Primarily significant for the purpose of this thesis is an understanding of the syntax and semantics of the declarative languages. These two languages form two layers of abstraction. QVT-Relational, already introduced in the preceding chapter, is a user friendly language that is translated to QVT-Core, a more technically specific language used to finally process the transformation. This section will summarize both language specifications and sketch how the transformation from QVT-Relational to QVT-Core is performed. Finally, an example transformation scenario is explained in section 3.3 to show how the rules in QVT-Core are processed.

The example introduced in chapter 2 is referenced throughout this section. To understand the transformation examples, recall the example domain models (see pp. 16).

3.1 Transformations

Transformations in both QVT-Relational and QVT-Core consist of a set of rules. But, they also globally declare the domain models that shall be transformed and with this, the packages where these models are defined.

```
transformation ctools2pnet (ctools:com.example.ctools, pnet:com.example.pnet) {
```

3 This summary is solely based on the QVT Final Adopted Specification ([QVT]). Presumably, there will still be modifications towards the finalization of the standard. Also, the specification is not always absolutely clear about both the syntax and semantics of the languages. The summary here is given in the best knowledge concerning the available sources.

4 See [QVT], pp.13, 119.
These domain models and the classes defined therein are then referenced by the rules. This is schematically shown in figure 17.

![Diagram showing QVT rules referencing domain models](image)

**Figure 17: QVT rules referencing the domain models**

This figure shows how the QVT rules reference the domain models. The instance models are transformed by applying QVT rules.

Transformations may furthermore be executed in a certain *mode*. If a source instance model exists and a target instance model shall be produced by the transformation, it's said that the transformation is executed in *enforcement mode* towards the target model. When a transformation is executed, it follows a *check&enforce*-semantic. This means that existing parts in the target model are matched against the rule pattern. More specifically:

- Model elements that are successfully matched remain in the model.
- Model parts that do not exist are created.
- Parts that do not fully satisfy the pattern predicates of the rules are modified and
- All model parts that are not successfully matched by a rule are removed from the target model

The transformation can furthermore be executed in *checking mode*, which means that it is checked if existing domain models are valid with respect to the relations defined by the transformation.

### 3.2 Relations and Mappings

Transformation rules in QVT-Relational, called *relations*, were already introduced in the preceding chapter. A transformation rule in QVT-Core, called a *mapping*, is conceptually similar
to a relation in QVT-Relational. Often, there is a one-to-one transformation from a relation to a mapping, but this is not always the case as section 3.2.2 will explain shortly.

But firstly, the example from the previous chapter is revisited. The following listing shows the TrackToPlaceArcTransition relation in its textual notation (see its graphical notation back on page 18):

```plaintext
top relation TrackToPlaceArcTransition{
    checkonly domain ctools track:Track {
        componentToProject=project:Project{},
        componentToPort=inPort:Port{},
        componentToPort=outPort:Port{}
    };
    enforce domain pnet arc:Arc {
        arcToPetrinet=petrinet:Petrinet{},
        arcToPlace=place:Place{placeToPetrinet=petrinet},
        arcToTransition=transition:Transition{transitionToPetrinet=petrinet}
    };
    when{
        ProjectToPetrinet(project, petrinet);
    }
}
```

In the subsections below, the constructs in this QVT-Relational rule are described and how they are translated into QVT-Core. Both languages are explained step by step until the complete QVT-Core mapping corresponding to this relation is listed on page 39.

### 3.2.1 Execution Direction

In addition to the overall execution mode of a transformation explained above, the single domain sides of the rules in QVT-Relational can be marked with `checkonly` or `enforce`. This determines which execution direction of the transformation is intended. A rule as displayed in the relation listed above would therefore just allow one execution direction. In case all domains are enforceable, the rules allow a bidirectional transformation. When all domains are just checkable, only the `checking` execution mode is available.

Now, when translating relations to QVT-Core mappings, there is a mapping generated for each enforcement domain in the relation. Each of these mappings has marked one domain as enforceable whereas the others are marked as checkonly. So, for example, if the TrackToPlaceArcTransition had two enforceable domain sides to specify a bidirectional transformation, two mappings result form this relation:

```plaintext
map TrackToPlaceArcTransition{
    check ctools...
}
```

---

5 See [QVT], p. 15

6 See [QVT], pp. 139, 140 mapping rule 4.1, 4.2
The rule in the example above specifies a 1:1 (1-to-1) transformation. Technically, a mapping could also be used to transform n models to one enforceable target model, a n:1 transformation. However, the QVT specification does not provide sufficient detail of how an n:m model transformation is processed with such a set of mapping rules.  

### 3.2.2 When- & Where-Clause

Before looking at the notation of the patterns, the when- and where-clauses are examined a little closer. The relation listing above shows that another relation is called in the when-clause (see relation ProjectToPetrinet, section 2.2.2 on page 20). The when-clause states that this relation is only valid if a certain previous condition is satisfied. This precondition might be an additional pattern condition or a reference to a condition specified by another relation, a relation call.

In addition to the when-clause, QVT relations can also declare where-clauses. These where-clauses can be used to state that further conditions or succeeding relations need to hold. The following listing shows an example:

```plaintext
check enforce pnet...
}
map TrackToPlaceArcTransition{
    check enforce  ctools...
    check pnet...
}
```

7 Here, the QVT specification is not very clear. There are examples of QVT-Core mappings listed that have more than one enforceable domain side. See [QVT], pp. 178, 179
The `PortToPlaceName` relation would, for example, look like this:

```plaintext
relation PortNameToPlaceName{
    checkonly domain ctools port:Port{
        name = n;
    }
    checkonly domain pnet place:Place{
        name = n;
    }
}
```

Here, the name of a Port is mapped to the name of the respective Place. So, schematically, the references between these rules could be seen as shown in Figure 18.

![Figure 18: Schematic dependencies between the example QVT relations](image)

This means that when the `TrackToPlaceArcTransition` relation is applied, also the `PortNameToPlaceName` and `PortNameToTransitionName` relations need to be applied on the model elements that are passed onto these relations as parameters (see the listings above). Now, through these relation dependencies, QVT-Relational offers a lot of flexibility to the transformation rule designer. There might exist any kind of when- and where-references between a set of rules. For instance, the `PortNameToPlaceName` and `PortNameToTransitionName` relation might yet be referenced from a where clause of another relation (see figure 19).

![Figure 19: Extended schematic relation dependencies](image)

This means that both relations, `TrackToPlaceArcTransition` and `SignalToPlaceArcTransitions` additionally require the relations `PortNameToPlaceName` and `PortNameToTransitionName`.
and PortNameToTransitionName to hold for the respective domain pattern elements that are passed on to them. This way, certain conditions that occur repeatedly in multiple relations or with multiple pattern elements inside a single relation can be sourced out to separate relations.

These relation calls in a where clause may leave the impression that relations in QVT-Relational specify a certain control flow. But, these relation calls are translated to a flat set of mappings in QVT-Core where no actual rule invocations occur. In QVT-Core, rules also refer to a previous context. Here, the when-clauses are mapped to a conceptually comparable construct. But, in contrast to QVT-Relational, QVT-Core mappings do not reference succeeding relations directly: Relation calls from the where clause are rather mapped to when-like context references of the succeeding mapping as shown in figure 20.

![Figure 20: Translating when and where relation calls to QVT-Core mappings](image)

But, since relations can be invoked by multiple preceding rules, QVT-Core mappings will have to be unfolded so that each "invoked relation maps to a separate mapping for each invoker-invoked combination". The relations shown to the right in figure 21 represent the relation invocation structure from figure 19. A more abstract representation is shown for brevity. These relations are translated to an unfolded set of mappings. If relation E is called from the where clause of relation B and relation C, then there will be two mappings E_B and E_C in QVT-Core where one references B in its context and the other references C.

![Figure 21: Unfolding of where-relationships](image)

This leads to a multiplication of mappings with each invoker-invoked combination. When invoked relations yet invoke further relations, this could lead up to a quadratic quantity of

---
8 See [QVT], p. 138, first mapping rule
mappings compared to the original relations. There could be even more mappings, depending on how often relations are invoked by others from their where clause.

### 3.2.3 Top-level Relations and Top-level Mappings

To start a transformation, QVT-Relational allows to mark certain relations as top-level. These relations may call other relations through relation calls in their when and where clauses. So, in QVT-Core, it is not always the directly corresponding mapping that is used to start the transformation, because another mapping may be yet needed to provide a proper context. This leads to a different meaning of top-level mappings in QVT-Core. Mappings in QVT-Core are called top-level mappings when they can be applied without any further context. These semantics of top-level are not exactly the same which leads to further implications in the translation of QVT-Relational to QVT-Core. The QVT specification is not very clear about the relationships between these two meanings of top-level. Because details are not particularly relevant for the purpose of this thesis, they will be omitted here.

### 3.2.4 Relations to Mappings

The previous section showed how the relation invocations are translated from QVT-Relational to QVT-Core. Now, the following question is, inspecting the rules a little closer, how the domain patterns of the relations are translated from relations to mappings. Therefore, the schematic structure of relations and mappings is introduced in the following

#### 3.2.4.1 Schematic Structure of a Relation (QVT-Relational)

Figure 29 shows the schematic structure of a QVT relation. In this figure, there is a relation with two domains (L and R) and its when and where-clauses. The two outer circles represent the set of variables that are used in the patterns.

![Figure 22: Schematic structure of a QVT relation](image)

Now, two kinds of variables can be distinguished in a domain side of a relation (represented by the included circles):

1. Variables that occur in the when-clause or in both the when-clause and the domain pattern
2. Variables that occur in the domain pattern or in the where-clause or in both the where-clause and the domain pattern

Before showing how these variables and the pattern expressions are translated to QVT-Core, the patterns of QVT-Core mappings are introduced.

### 3.2.4.2 Schematic Structure of a Mapping (QVT-Core)

A mapping from QVT-Core, compared to QVT-Relational, has a different schematic representation as shown in figure 21.

![Figure 23: Schematic structure of a QVT-Core mapping](image)

In the structure of a QVT-Core mapping as shown here, again the two domain patterns occur (L and R). The columns in this structure are called *areas*. In addition to the L- and R-domain areas, there is a *middle area*. The middle area is added during the transformation. It contains the *trace class* which maintains the mapping between the domain pattern elements and can be compared to the correspondence nodes in TGGs (see section 2.2.1 p. 20). When applying this rule, instances of this trace class, the *trace objects* maintain the mapping structure of the instance model patterns (more details will follow in section 3.3, pp. 40).

Each area is divided in two patterns, the *guard pattern* and the *bottom pattern*. The guard pattern contains variables and predicates that describe a certain context that has to hold in order to apply the rest of the rule. For each variable in the guard pattern, there has to exist a valid variable binding in the previously processed instance models and, furthermore, all the predicates in the guard pattern have to be valid. If the context described in the guard patterns is valid, then the bottom patterns can be applied. This will be explained shortly in section 3.3 *A model transformation processing scenario* on page 40. Before it is explained in the following how the relation patterns from QVT-Relational will be transformed into these mapping patterns.

---

9 See [QVT], p. 120
3.2.4.3 Mapping Relation Patterns to Mapping Patterns

In the domain sides of a relation, as shown in figure 22 (see p. 29), two different types of pattern variables have been distinguished: Variables that occur in the when-clause and variables that occur in the domain pattern and/or the where-clause. Now, the when-clause in a relation is describing a certain precondition that has to be valid in order for the rest of the relation to apply. This corresponds to the guard pattern of the QVT-Core mapping. The other variables, in contrast, are part of the actual pattern relationships described by the relation. Therefore, these variables belong to the bottom pattern. Figure 24 illustrates this mapping\(^{10}\).

![Diagram showing mapping of pattern variables from QVT-Relational to QVT-Core](image)

**Figure 24: Mapping pattern variables from QVT-Relational to QVT-Core**

This mapping is shown in the case of the TrackToPlaceArcTransition example relation in figure 23. The graphical representation of this rule was already introduced on page 18. Here, the elements have been rearranged a little to visualize the variable mapping.

---

\(^{10}\) See also [QVT], p. 138, mapping rule 2
In this example, the Project and Petrinet variables also appear in the when clause of the relation, namely in the relation call expression. So, they are translated into guard pattern variables in the mapping. The remaining variables are translated into bottom pattern variables as they represent the actual transformation patterns. Also shown here are the trace classes that reside in the middle domain and reference the domain pattern variables.

Along with the variable mappings, the predicates that occur in the when-clause become predicates of the guard pattern and predicates from the domain and/or where-clause become predicates in the bottom pattern. One exception here are the relation invocation calls in the where-clause. They will not be transformed into the current QVT-Core mapping, but will be “reflected in the mapping corresponding to the invoked relation”\(^\text{11}\) (also see section 3.2.2 When- & Where-Clause, pp. 26). The following section will describe the transformation of the predicates in more detail.

### 3.2.5 Patterns and Predicates

In QVT-Relational, the pattern elements like `track:Track` are typed variables which are bound to model elements as model patterns are checked or enforced during the rule application. Patterns of these variables are expressed as nested OCL predicates which are boolean

---

\(^{11}\) See [QVT], p. 138, mapping rule 2.6.1
expressions that state conditions over these variables. The example listing above just shows simple equality expressions, but the full expressive power of QCL is available in QVT. So, for example, also expressions like string concatenation as shown here would be possible:

```plaintext
checkonly domain ctools track:Track {
    componentToProject=project:Project{},
    componentToPort=inPort:Port{name = s1},
    componentToPort=outPort:Port{name = s2},
};

enforce domain pnet arc:Arc {
    arcToPetrinet=petrinet:Petrinet{},
    arcToPlace=place:Place{
        placeToPetrinet=petrinet,
        name="port_" + s1
    },
    arcToTransition=transition:Transition{
        transitionToPetrinet=petrinet,
        name="port_" + s2
    }
};

when{
    ProjectToPetrinet(project, petrinet);
}
```

OCL certainly allows a wide variety of expressions and operations. Also, OCL allows operations to be combined with further operations to form complex terms. But, this raises two problems: Firstly, OCL is a query language. So, OCL expressions can be evaluated, but the pattern conditions described by OCL statements can not always be enforced in a transformation. A simple example is a greater-or-equal operation (\(\text{inPort.capacity} \geq \text{place.maxTokens}\)). This conditions can be evaluated, but no specific enforcement operation can be derived from this statement (see subsection 3.2.5.3, p. 36 on enforcement operations). The QVT-Specification is not formulating clear restrictions to the kinds of OCL expressions that can be used, but it appears that only equality predicates have corresponding enforcement operations when creating target object patterns.

A second problem that results from the use of OCL is that certain constructs cause the bidirectionality of the rule to be difficult or impossible to maintain. In the example above, there is a string concatenation expression. In case of a backward transformation from pnet to ctools, there is no string subtraction operation defined that could act as an inverse of the concatenation operation. In the above example, only one transformation direction is indicated by the checkonly and enforce specifiers, so the string concatenation operation causes no problem in this case. In general, however, there has to be an inverse to each operation in bidirectional transformation rules (see Figure 26).
Now, there is no inverse to each operation in OCL, so the rule designer has to be aware of the implications of exploiting the full expressive power of OCL in the rules. Unfortunately, the QVT specification is not addressing this issue.

3.2.5.1 Domain Patterns in QVT-Relational

In QVT-Relational, the predicates describing the domain patterns are nested expressions, structured in braces. The following listing shows the ctools domain pattern of the TrackToPlaceArcTransition rule.

```plaintext
checkonly domain ctools track:Track {
    componentToProject=project:Project{},
    componentToPort=inPort:Port{name = s1},
    componentToPort=outPort:Port{name = s2},
};
```

In QVT-Relational, expressions of the form `variableIdentifier:variableType{...}`, for example `track:Track{...}`, are called `ObjectTemplateExpressions`. Variables have a type that, as in this case, can be a class or a data type. Inside such ObjectTemplateExpressions, there can be expressions of the form `propertyName=OCLExpression`. These expressions are called `PropertyTemplateExpressions`. For example, `name=s1` is a simple PropertyTemplateExpression where the property is `Port.name` and the value OCL expression is simply the variable `s1` (VariableExpression). Another example of a PropertyTemplateExpression is `componentToProject=project:Project{}`. Here, the property is `Component.componentToProject` and the value OCLExpression is again an ObjectTemplateExpression. These are the instruments to describe domain patterns in QVT-Relational.

However, the QVT specification does not provide precise information concerning the handling of set-valued properties. In the ctools example, the componentToPort reference is actually set-valued and an equality expression as shown in the listing above may not be the completely correct expression here. Probably, if a set-valued property has to be set, an expression like the following would be possible.

```plaintext
checkonly domain ctools track:Track {
    componentToProject=project:Project{},
    componentToPort=Set(Port){
        inPort:Port{name = s1},
        componentToPort=outPort:Port{name = s2},
    }
};
```
For simplicity, the QVT rules in this thesis do not use set-valued expressions. Section 6.3 on page 109 discusses this with respect to transforming QVT to TGGs. After explaining the domain pattern structure in QVT-Relational, their corresponding constructs in QVT-Core will be introduced.

### 3.2.5.2 Transformation of Checkonly Domain Patterns to QVT-Core

When the nested predicates (ObjectTemplate- and PropertyTemplateExpressions) are translated from QVT-Relational to QVT-Core, they undergo transformations depending on the enforcement mode of the containing domain pattern of the rule. Firstly, the transformation of a checkonly domain pattern will be described.

The ctools domain from the above listing is marked as checkonly:

```plaintext
ccheckonly domain ctools track:Track {
  componentToProject=project:Project{},
  componentToPort=inPort:Port{name = s1},
  componentToPort=outPort:Port{name = s2},
};
```

This relation will look like this in QVT-Core:

```plaintext
ccheck ctools(project:Project){
  track:Track, portIn:Port, portOut:Port|
  track.project = project;
  portIn.track = track;
  portOut.track = track;
  portIn.name = s1;
  portOut.name = s2;
}
```

In this listing, the guard and bottom patterns as introduced in section 3.2.4.2, p. 30 can be seen. The guard pattern is enclosed by brackets and the bottom pattern is enclosed by braces as shown here.

```plaintext
ccheck ctools{
  <<guard pattern>>
} {
  <<bottom pattern>>
}
```

As the listing of the complete relation TrackToPlaceArcTransition on page 25 shows, the project variable is used in the when-clause and therefore appears in the guard pattern of the QVT-Core mapping. The remaining variables do not appear in the when-clause and therefore only appear in the bottom pattern of the mapping. Also the domain pattern predicates from the relation reappear in the guard and bottom patterns. In this case, there is no predicate expression in the guard pattern, but they appear in the bottom pattern. Here, the predicates are simply listed in a flattened sequence of expressions. So, in a QVT-Core pattern, the structure from the nested TemplateExpressions in QVT-Relational may get lost, but only as the patterns are transformed...
from a checkonly relation domain. In the transformation of enforce relation patterns, additional transformation conditions are applied as described in the following.

### 3.2.5.3 Transformation of Enforceable Domain Patterns To QVT-Core

In the case where the relation pattern is not marked as checkonly, but as enforceable, the pattern is differently translated to QVT-Core. The following listing is again showing the ctools pattern from above, but now marked as enforceable:

```plaintext
enforce domain ctools track:Track {
    componentToProject=project:Project{},
    componentToPort=inPort:Port{name = s1},
    componentToPort=outPort:Port{name = s2},
};
```

The pattern structure in the relation pattern remains the same, but the QVT-Core mapping patterns will be transformed from this pattern in another way. Now, each variable in the bottom pattern will become a realized variable. This is primarily a way to distinguish a variable that may be created in a target pattern. In addition, each predicate will become an assignment. Assignments are the counterpart to equality predicates\(^\text{12}\). As already discussed in section 3.2.5, only equality predicates have the assignment as their corresponding enforcement operation when creating target object patterns. Assignments look similar to the equality predicates, but will use the assignment operator instead (\(:=\)). So, the corresponding QVT-Core mapping pattern will look like this:

```plaintext
check enforce ctools(project:Project){
    realize track:Track, realize inPort:Port, realize outPort:Port|
    track.componentToProject := project;
    track.componentToPort := inPort;
    track.componentToPort := outPort;
}
```

Compared to the QVT-Core mapping pattern in the previous checkonly case, this pattern is missing the two port name predicates:

```plaintext
inPort.name = s1;
outPort.name = s2;
```

These predicates originate from nested ObjectTemplateExpressions in the relation pattern and are treated differently when an enforceable pattern is transformed. These nested expressions are decomposed\(^\text{13}\) into nested mappings. The following listing shows an example:

```plaintext
map TrackToPlaceArcTransition{
    ...
    check enforce ctools(project:Project){
        realize track:Track, realize inPort:Port, realize outPort:Port|
        track.componentToProject := project;
        track.componentToPort := inPort;
        track.componentToPort := outPort;
    }
}
```

---

\(^{12}\) See [QVT], p. 138, mapping rule 2.4

\(^{13}\) See [QVT], p. 140, mapping rule 4.3
The nesting of mappings in the enforceable object patterns serves two purposes. Firstly, it is a way to maintain the navigation structure that exists in the nested predicates of QVT-Relational. The navigation structure is not of so much concern when existing model elements have to be checked. But, during the enforcement of domain patterns, the appropriate order of creating model objects and references is maintained by sequentially applying the nested mappings. Another purpose of nested mappings is related to the concept of keys, which will be explained in the following.

### 3.2.5.4 Keys

The concept of keys plays a crucial role in the check&enforce semantics of QVT. Keys\(^{14}\) are sets of attributes that can be declared in QVT-Relational to aid in the specific identification of domain model elements. In object oriented programming in general, there might be two objects that are structurally equal, which means that all property values are equal. Often one attribute is selected to serve as a key property to uniquely identify an object. For example, EMF [EMF] allows to mark an attribute of a class to be its unique identifier.

Semantically, not just one property may determine the identity of an object in a model, but rather a set of properties. Similarly, relational databases allow to specify certain sets of columns as (secondary) keys of tables. It is important to be able to express such identities during a transformation where certain target objects may exist. The check&enforce semantics of QVT, as explained in section 3.1 (pp. 23), will check existing model objects before creating new ones. Now, there have to be identities to determine if an existing model object shall be modified in a transformation step or if rather a new object shall be created (then existing objects are potentially deleted). This is particularly important because an object may hold other relevant data or references to further model parts that are not subject of the transformation.

The following listing shows the header of a relation that specifies keys.

```plaintext
...  
map{  
    check enforce ctools(){  
        portIn.name := s1;  
        portOut.name := s2;  
    }  
}  
}
```

Here, two keys are declared. For Tracks, there is only one key attribute name. But Ports may be uniquely identified through their type and name.

---

\(^{14}\) See [QVT], pp. 16, 17
Now, keys may be declared in QVT-Relational, but they do not explicitly exist in QVT-Core. In QVT-Core, the keys are rather reflected in the structure of the nested mappings. If a predicate from the relation (PropertyTemplateExpression) is an assertion over a non-identifying variable, the corresponding assignment will be located inside a nested mapping. But, if the PropertyTemplateExpression is an assertion over an identifying variable, the asserting will be located in the current level mapping. In the sequential processing of the nested mappings, this has the effect that primarily all identifying variable assignments can be checked with an existing target model object. If this primary check is successful, this means that the identity has been confirmed. Then, the processing of the nested mapping will modify all non-identifying property values if necessary. If this primary identity check is not successful, then a new model object is created according to the nested target pattern.

Thus, with the example key declarations above, the example mapping from section 3.2.5.3 (see p. 3.2.5.3) will not exclude the Port name assignments in a nested mapping.

```plaintext
map TrackToPlaceArcTransition{
  ...
  check enforce ctools{project:Project}{
    realize track:Track, realize inPort:Port, realize outPort:Port|
    track.componentToProject := project;
    track.componentToPort := inPort;
    track.componentToPort := outPort;
    portIn.name := s1;
    portOut.name := s2;
  }
}
```

This way, when there may exist a Track with its Ports in the ctools model, the port name is checked at the top level to determine if the checked model objects shall be modified or if rather a new Track with its Ports shall be created.

### 3.2.5.5 Transforming Relation Calls to QVT-Core Mappings

After showing how the pattern variables and predicate expressions are translated from QVT-Core to QVT-Relational, this subsection shows how relation calls from the when- or where clauses of the relations are translated. Figure 20 (back on page 28) already schematically showed how a relation call in a when- and where clause is transformed into a corresponding QVT-Core mapping. It was shown how a relation call from a where-clause is translated into a rather when-like construct in the mapping corresponding to the invoked relation. Both cases are translated into mappings that reference the context of previously processed mapping.

Such a when-like dependency between the mappings is expressed by listing a mapping's trace class in the guard pattern of the another mapping. This is schematically shown in figure 25.

---

15 See [QVT], p. 140, mapping rule 4.3 bullets 2 and 3
The dotted arrows in this schema represent the pattern dependencies in the mappings. Variables from one pattern are visible to each referencing patterns. So, by referencing another mapping's trace class in the mapping's guard pattern, a context that was previously processed by another mapping can be referenced. An example was actually already shown previously in figure 23 (p. 30).

The QVT-Core mapping resulting from such a when-clause relation invocation is listed below. Derived from the rule names, the trace classes in this example are called TProjectToPetrinet and TtrackToPlaceArcTransition.

```ml
map TrackToPlaceArcTransition{
    check ctools(project:Project){
        track:Track, portIn:Port, portOut:Port|
        track.project = project;
        track.port = portIn;
        track.port = portOut;
    }

    check enforce pnet(petrinet:Petrinet){
        realize place:Place, realize arc:Arc, realize transition:Transition|
        arc.arcToPetrinet := petrinet;
        arc.arcToPlace := place;
        arc.arcToTransition := transition;
    }

    where(t1:TProjectToPetrinet|
        project=t1.project,
        petrinet=t1.petrinet){
        realize t2:TTrackToPlaceArcTransition|
        t2.track := track;
        t2.inPort := inPort;
    }
}
```
Here, firstly the two domain patterns for the eTools and pnet domains are shown. The nested mapping expression actually belongs to the pnet side pattern. The where clause here represents the middle domain and shall not be mistaken with the where clause in QVT-Relational. In the guard pattern of the middle domain, the trace class from the precondition mapping TrackToPlaceArcTransition is listed. This way, the two variables project and petrinet that were previously bound by the ProjectToPetrinet mapping become accessible here. The bottom pattern of the middle domain creates the trace object for this mapping and assigns the references to all participating domain pattern variables.

Now, this chapter on QVT has summarized the most important characteristics of QVT-Relational and QVT-Core and how a transformation from QVT-Relational and QVT-Core is performed. The QVT-Core mapping listed above is also the complete counterpart to the initial relation TrackToPlaceArcTransition on page 25.

### 3.3 A model transformation processing scenario

Remaining in this chapter on QVT is a short description on how a transformation is finally performed based on a set of QVT-Core mappings. The QVT specification itself does not provide details about transformation algorithms since it rather focuses on the implementation independent language. But, this section explains how a simple QVT algorithm may process a transformation.

For this purpose, the small example of a component system that was introduced in chapter 2 will be explained in more detail here. Figure 28 repeatedly shows the example domain models. Here, two Track components and one connection in a Project have to be translated into the corresponding Petri net constructs.
Until now, two QVT-Rules for the ctools-to-pnet example have been introduced. A rule that translates the root objects, Project and Petrinet (ProjectToPetrinet). The second rule specifies how a Track with its incoming and outgoing Ports can be transformed into a Place, Arc and Transition (TrackToPlaceArcTransition). For this example scenario, now a third rule is finally added that transforms a Connection in ctools to an Arc in a Petri net (ConnectionToArc). Figure 29 lists the three QVT-Core rules in an abstract graphical notation. This graphical notation of QVT-Core mappings is just used to ease the understanding of QVT-Core, it is not specified by QVT.

**Figure 28: The transformation example**

**Figure 29: Abstract representation of the three example QVT-Core mappings**
The two rule ProjectToPetrinet and TrackToPlaceArcTransition should be familiar from the previous examples. It can be seen here that the precondition patterns are placed in the guard pattern, the bottom pattern contains the patterns that shall be checked or enforced by the rule. The third rule, ConnectionToArc is used to transform a Connection that connects an outgoing and incoming Port of two Track components. Corresponding to a connection, there has to be an Arc in the Petri net that connects the Place and Transition corresponding to the outgoing and incoming Port. In this rule, the guard pattern is a little more complex and demonstrates a case where a context provided by several previous rules is required.

In the following, the transformation of a small example ctools model will be explained step by step. Figure 30 shows the example where a Project with two Tracks is already given as a source model. To start the transformation, a top mapping, which does not specify any context in its guard pattern, is matched in the source instance model. Then, the target model elements are enforced as well as the trace object. As shown here, the first transformation rule has already been applied that created the Petrinet object in the pnet instance model.

Next, the source instance model is searched for a pattern that occurs in a bottom pattern of a QVT-Core mapping in the rule set. The bottom pattern from the ConnectionToArc mapping might be matched, but the required context is not yet available to apply this rule. Instead, the TrackToPlaceArcTransition rule is applied. As shown in figure 31, the required precondition pattern is found and then, the Place, Arc, Transition objects are created as well as the trace object.
The same rule, `TrackToPlaceArcTransition`, can be applied a second time to translate the second ctools Track. Then, there is enough context available to apply the third rule, `ConnectionToArc` on the connection object left in the source model. This is shown in the next figure 32.

After this fourth transformation step, there are no unbound source model objects left and the transformation is finished. This simple processing scenario shows the graph pattern matching and creation that takes place for each rule. Attribute value transformations, omitted in the example, typically take place at the end of a rule application or as soon as all necessary model objects are bound.
3.4 Summary

This chapter introduced the two declarative languages, QVT-Relational and QVT-Core, of OMG's new QVT (Query/Views/Transformation) specification. It was shown how they form two layers of abstraction to specify relations between model patterns. QVT-Relational is more abstract and user friendly and allows to explicitly specify the sequence of rule applications through the when&where clauses of the single relations. To process a transformation, the QVT-Relational rules are then transformed into QVT-Core rules. The principles of this transformation have been explained. For example, how invocations between relations are unfolded or how patterns are transformed from QVT-Relational to QVT-Core. Last, a small example scenario explained the major steps in a transformation based on a set of QVT-Core mappings.

However, a few language constructs have not been covered in this chapter, because they are not relevant for the further contents of this thesis. For example can transformations and single rules be extended by other transformations and rules. This is a practical issue because it allows to flexibly specify an extendable hierarchy of transformations. Although this is not thoroughly covered by the QVT specification. Another practical issue is that QVT-Core mappings can specify default assignments. Default assignments can be used to execute operations prior to the rule execution to initialize participating model elements.

The next chapter will introduce Triple Graph Grammars (TGGs). Some principles of TGGs are quite similar to QVT. Chapter 6 will then show how a mapping from QVT to TGGs is accomplished.
In object oriented software, data is structured as linked objects which can be considered as graphs. Triple Graph Grammars (TGGs, as introduced by A. Schürr [Sch94]) describe relations between two sets of graphs and provide a convenient formalism to specify the correspondence between two models. TGGs can be used to declaratively describe bidirectional model transformations [Kön05][KRW05] or to describe how the consistency between two models can be maintained [KS06a][GW06].

In this thesis, TGGs are used for QVT-like MOF model transformations. Different approaches or existing tools for model transformation that use TGGs as their underlying formalism may interpret TGGs differently. This is because the application area or the technologies used are different. This chapter discusses some of these issues and explains the overall semantics of TGGs. The next chapter then presents the implementation of TGGs in more detail as well as a model transformer that interprets TGGs.

### 4.1 Introduction to Triple Graph Grammars

Graph grammars, similar to string grammars, describe certain sets of graphs through a number of graph production rules. Triple Graph Grammars (TGGs) are an extension to Pair Grammars [Pra71]. They relate two sets of graphs by relating the single graph production rules. In TGG rules a third graph grammar is introduced to describe how the other two structurally relate to each other. Figure 33 shows an example of a TGG rule.
In this quite simple rule, the single production rules for graph A and graph B produce differently structured graph patterns. The correspondence nodes in the middle provide specific mapping information and can be used to relate one or more elements on both sides.

In this example, the node from the left side of each single graph grammar rule reappears in the right side of the rule. Here, “left” and “right” refer to the usually horizontal reading direction of grammar production rules which are now aligned vertically. Graph grammars that do not discard nodes in their production rules are called non-deleting rules. For the use of TGGs for model transformation, only non-deleting rules are of concern. Deleting grammars (graph grammars as well as string grammars) increase the complexity of checking, for example, which sequence of rule applications produces a given phrase or graph structure. Not only is this is a computationally complex problem, but it may also make the resulting language of the grammar hardly predictable for a rule designer. However, model transformation or model manipulation can be achieved with just relying on non-deleting rules. This allows a more compact notation of the above rule as shown in figure 34. Here, basically only the right side graph patterns are shown and the nodes that appear in the left side as well appear in the dedicated top section. The latter nodes are also referred to as context- or precondition nodes.

Figure 33: The three production rules of a TGG rule

Figure 34: A TGG rule in compact notation
The notation shown in figure 34, however, is only useful if the graph pattern structure allows a readable arrangement of the nodes. Another alternative notation is shown in figure 35.

![Figure 35: Alternative TGG rule notation](image)

In this notation, used for example by Fujaba [Fujaba], the white nodes represent nodes that appear on both the left and the right side of the graph grammar rule. The green nodes, also marked with a “++”, are those that are additionally produced in the rule.

Now, a Triple Graph Grammar can be used to transform one graph into another or, furthermore, to check if two given graphs structurally correspond to each other. Depending on the application, a TGG can be interpreted differently. Figure 36 schematically shows an application scenario where one graph is given and shall be transformed into another graph.
In this scenario, the existing graph A on the left is parsed by the respective side of the Triple Graph Grammar. Whenever a graph grammar rule on this side can be applied to match a certain pattern in graph A, then the other two sides of the graph grammar rule are applied to create the graph patterns for the correspondence graph and graph B. One rule application step is shown in this figure: A start context for the graph transformation was given and serves as the context to apply this first rule. Then a pattern in graph A is successfully matched and, thus, the graph B and correspondence graph patterns are created. The common context that has to exist before matching or parsing any right graph grammar patterns always provides a previously created correspondence. So, succeeding graph patterns are always created in the correct corresponding locations.

As mentioned above, the forward transformation is just one possible application for TGGs. Figure 37 shows the three different ways to interpret a TGG rule, depending on the transformation direction.
During a forward transformation from a one source graph to a target graph, as shown in the example scenario above, firstly the existing source graph pattern is matched with the source side of the rule. Then, target and correspondence patterns are created. In the backward transformation scenario, the former target side of the TGG is matched and the former source side and correspondence patterns are created accordingly. In the last transformation scenario, there is actually no model transformed, but the TGG rules are applied to calculate a valid correspondence between two existing graphs. For this purpose, both source and target graph grammar sides are matched with the existing graph elements. Then, the correspondence graph pattern is created. If such a correspondence transformation is performed successfully, the created correspondence nodes represent a fine-grained structural mapping of the two given graphs.

In this introductory example, TGGs were introduced for simple graphs to show the pure graph based nature of this formalism. The following section explains how the introduced concepts are used for model transformation and which additional requirements arise in this area of application.
4.2 TGGs and Model Transformations

To transform (MOF) models with Triple Graph Grammars, TGGs express how object graphs relate to each other. Furthermore, a number of additional TGG language constructs are necessary. For example, TGGs have to be extended to express attribute value transformations. The characteristics of TGGs for model transformation and additional constructs are introduced in the following.

4.2.1 Transformations of Object Graphs

To transform object models, TGG rules relate object graph patterns from the source and target domain models. The following figure repeatedly displays an example TGG rule from the introductory example in chapter 2 (pp. 13).

![Figure 38: An example TGG rule for model transformation](image)

According to the previously introduced transformation directions (see figure 37), a TGG rule is applied, for example as shown in figure 39, to perform a transformation step from a ctools model to a pnet model. Firstly (1), the context (left side) pattern of the TGG rule has to be matched with previously processed model elements. Secondly (2), an existing, unprocessed pattern in the source model has to be parsed by the source graph grammar. This means that there has to be an unprocessed ctools object pattern that needs to be matched with the right ctools TGG pattern. Then, finally (3), the target and correspondence object patterns are created.

![Figure 39: TGG rule applied in a forward transformation](image)
The terms *object patterns* or *object graphs* have been already been used in the preceding chapters. From the view of graph grammar theory, an object graph is a *typed* graph. Each object node in the graph has a class as its type. An object graph is defined by a class diagram that acts as its *graph schema* or *type graph*. Also the edges in an object graph are typed as they represent an instance of a reference between two classes. In this sense, *matching* a node in the TGG rule means that the class of the TGG node must be equal to the class of the matched model object.

Figure 40 shows how the TGG rules reference the domain models as well as a correspondence model.

The correspondence model is the graph schema for the correspondence graph where each type of correspondence node has to be specified by a class. The TGG rules can then be used by a transformer to perform a transformation between instances of these models. This is very similar to QVT (see p. 23) where domain model patterns consist of typed variables.

In the following, additional TGG constructs are introduced that are necessary to use TGGs for model transformation.

**4.2.2 Attribute value transformation**

In addition to references to other objects, objects may hold primitive attribute values. Therefore, from the graph theory point of view, an object graph is *attributed*. The attribute values that can be held by an object are specified by the class model where classes can specify typed attributes. The TGGs introduced so far are able to match and transform objects and references. But, to handle attribute values, further constructs are needed. In this thesis, these constructs are called *attribute constraints*. Figure 41 gives an example of how attribute constraints can be used.
In this example, there are two kinds of attribute constraints. One kind of attribute constraint, a literal constraint, compares an attribute value with a given literal constant. In this example, the Ports from the ctools model need to have a certain “type” value. During pattern matching, these attribute constraints formulate additional criteria specifying when to match and object and a TGG node. When creating target objects, such literal constraints specify which attribute values need to be assigned after creation. The other kind of constraint shown here is an attribute equality constraint. These constraints can either express attribute value equality inside a source model pattern. So, a source object pattern can only be matched if the equality constraint is satisfied. When in contrast specifying an object constraint across the domain models as shown here, then the attribute value from the source model object is assigned to the target model attribute value after it is created. Alternatively, a more graphical notation of attribute constraints can be seen in the figure 42.

Here, the constraints are represented as rounded rectangles. The double arrows were chosen as the notation to mark the node that the attribute constraint is referring to. Because the attribute equality constraint is specified for both the forward and backward transformation direction, there are two constraints necessary. This graphical notation can be especially useful to visualize the
attribute constraints between two nodes. With many attribute constraints in the rule, on the other hand, this notation may become too confusing.

The constraints introduced so far only utilized the equality operator. But, further operations might be needed in practice. For example, this could be the concatenation of string values, addition of integers, boolean operators or further comparison operators like “greater than”. However, there are two problems with such additional operations. Firstly, verification tasks might become more difficult or impossible (see section 4.2.8, pp.59). Secondly, such operations might violate the bidirectionality of the transformation. As mentioned in the chapter on QVT (section 3.2.5, pp. 32), to maintain bidirectionality, there has to be an inverse to every attribute value assignment operation in the rule (see figure 43). For example, string concatenation may seem trivial, but the reverse operation might not always be applicable.

![Figure 43: Attribute value assignment operations and their inverse](image)

The next chapter (Chapter 5, pp.65) will describe how OCL statements can be used in TGGs to flexibly formulate more complex constraints.

The general concepts of TGGs have now been introduced. The following subsections present further extensions to TGGs or slightly enhanced applications scenarios for model transformation. At the end of this chapter, general concepts of TGG model transformations are summarized with an example rule processing scenario (see section 4.3, pp. 59).

### 4.2.3 Check & Enforce

In practice, there are often transformation scenarios where parts of the target model already exist. QVT handles this by also checking existing parts of the target model and only creating new objects if the target object does not conform to an identity specified by key attributes (see pp. 37). Existing TGG transformation applications from Fujaba or ComponentTools do not or not yet support the matching of existing target objects. It is planned for the TGG interpreter developed in this thesis, to allow to switch between the check&enforce semantic (according to QVT, see pp. 23) and a “enforce-only”-semantic. However, so far only the enforce-only semantic are supported. See Chapter 5 (pp. 65) for further details.

TGGs may not only be used to transform one model into another as one finite process. When integrating software tools or information systems, it is necessary to keep the different domain models of the single software components permanently synchronized. Changes may take place at different places in different models and these changes have to be propagated to other models. Transforming parts in two existing models is called *incremental model transformations* [GW06]. In such a scenario, sometimes objects are added and sometimes existing model elements are
modified or deleted. For incremental model transformation it is crucial that existing model
elements can be matched.

4.2.4 Matching Instances of Subclasses

A node is matched to an object if the object is an instance of the same class as type class of
the node. But, in object orientation, there may be an inheritance relationships between classes. So
classes may be superclasses or subclasses of others. There are certain cases in transformations
where rather an instanceof comparison (like in object oriented programming) between the model
object and a node's type class is desired.

For example, in the ctools model, the Track class inherits from the Component class (see the
ctools class diagram on page 16). Now, assuming that there are further subclasses of Component
and there might be rules that transform certain model elements in the context of all kinds of
Components in general. Then, it would be convenient to just have one rule with a generalized
“Component” node instead of writing a separate rule for each subclass of Component. However,
there are two problems with this gain of convenience. Firstly, a superclass might also be an
abstract class or even an interface. Then, it is no problem to match existing objects with this
superclass. But, when creating the target pattern, there is no information of what to create
exactly. This information was omitted along with the generalization. So, the use of nodes that
check for an instanceof relationship with a model object is only safe for the context nodes of the
rule or in a source model pattern (when no bidirectionality is intended).

The second problem when allowing to match subclasses is that the rule set might become non-
deterministic: There might be potentially two rules that can be applied in a transformation step
and one is matching for a more general superclass. A strategy can be invented to solve this
ambiguous situation: If two or more rules conflict in this way, always the more general rule has
to be applied. But, however, this would violate the pure declarative nature of TGGs and would
also increase the complexity of the matching algorithm.

4.2.5 Multi Graph Grammars

Triple Graph Grammars specify a relation between two graphs and, thus, can be used for 1-to-1
model transformations. In practice, however, it might be necessary to transform two, three or
sometimes even more models into one or more other models. 1-to-1 transformation rules could
be specified to express a transformation between each pair of models. But, not only is this
inefficient, it is even impossible when cross-model relationships strongly influence the result of a
transformation. It is possible with QVT to transformation n-to-m models. There can be any
number of source and target domain model sides in a transformation rule. Also, the TGGs
redesigned in this thesis allow to specify multiple domain model sides and can thus be rather
considered as Multi Graph Grammars. A. Königs and A. Schürr recently introduced Multi Graph
Grammars in their multi-document integration approach [KS06a]. In MGGs, several graphs can
be integrated by one correspondence model as shown in figure 44.
Formally, there is no problem to extend TGGs to MGGs. In a concrete n-to-m transformation scenario, the n models on one side and the m models on the other side can be subsumed to one source and one target graph. Then, a Multi Graph Grammar rule can again be seen as a simple Triple Graph Grammar rule.

### 4.2.6 Start nodes

Some approaches or technologies implementing TGGs transformations have restrictions of which initial node matches may trigger the application of a TGG rule (see the processing scenario in section 4.3, pp. 59). Sometimes, certain nodes have to be explicitly marked as start nodes or, at least, there has to be one node from where the whole directed graph pattern can be reached. But conceptually, in TGGs, such start nodes are not necessary for pattern matching. Even directed references in the model can be navigated backwards with little additional effort. The TGGs in this thesis will not need any start nodes nor are there any restrictions to the nature of the graph patterns other than that the whole TGG rule graph must be connected.

A different issue is that starting a pattern matching from some nodes may be more efficient than starting from others. For large rule sets, it could be useful to apply heuristics to find such nodes. However, this is only a matter of how efficiently TGG rules are interpreted by a transformation tool. It is not a characteristic of the TGG itself.

### 4.2.7 Further TGG Features

There are a few more issues or features of TGGs that are briefly discussed in the following. Often, further constructs are introduced to formal methods that seem practically useful for certain applications. But, on the other hand, many extensions may destroy the valuable characteristics of the formalism.

#### 4.2.7.1 Negative and Optional Pattern Elements

To extend the expressive power or the usability of TGG rules, one thing that seems useful are negative or optional pattern elements. Figure 56 illustrates an example.
For example, two similar patterns may result in the same target model pattern. In one case, there might be an additional object present, in the other case not. This can be expressed in a rule that allows optional elements. But, the general nature of this problem leaves the question of what model elements to create in a backward transformation. In most applications, such optional constructs are not necessary or can be avoided.

Negative pattern elements formulate constraints that a rule must not be applied when certain pattern elements are present. This may in some cases be necessary. But, there are different semantics of negative elements: Either negative pattern elements may refer to a local non-existence of model objects or they may specify that certain patterns must not exist in a model pattern at all. The latter is a rather global interpretation of negative pattern elements. Checking for such negative elements globally may increases the runtime of a matching algorithm because all possible forbidden patterns have to be searched before the rule can be applied.

One rather local and computationally feasible solution would be to specify that certain reference values are NULL or that attribute values are not equal to a certain value. Figure 46 shows an example.

In this example pattern, that may appear in the domain side of a TGG rule, one Port of a Track component is connected to a NULL node. The edge, in this case is labeled with the respective type reference name “outgoingConnection”. This means that this Port node can only be matched
if the Port object has no outgoing Connections. Furthermore, the “type” string of the Port object must not be equal to string constant “in”.

The TGGs implementation in this thesis provides a prototypical implementation of such NULL-nodes. However, optional edges or nodes are not supported.

### 4.2.7.2 Complete and Partial Grammars

Sometimes, only parts of a model are relevant and need to be transformed. Then, the TGG rules should only be designed for the relevant parts of the transformation. If the less relevant model parts, however, influence the transformation, they might appear as context nodes in the TGG rules. Consider the following example from the component model shown in figure 47.

![Figure 47: Extended component model example](image)

Here, the component kind (whether it is a Track or another type of component) is not expressed by a concrete subclass, but by a ComponentType. The ComponentType itself is never translated directly, but strongly influences the result of the transformation. Often, only parts of a model are subject to a transformation and others just supply additional information. But, as mentioned previously, the context nodes need to be matched with previously processed model objects. That means that the TGGs, as introduced so far, need to specify a complete grammar for all participating model parts. The TGGs as implemented by Fujaba, in contrast, also allow to formulate a partial grammar on a model. There, the context nodes do not necessarily need to be previously processed by another rule. The TGGs in this thesis will distinguish such constraint nodes from the normal context nodes as shown in figure 48. Here, the constraint node is colored gray.
But, the constraint nodes may lead to problems when transforming in the backward direction. In case of the example shown above in figure 48, the correct ComponentType, if it exists at all, cannot be easily resolved in the target model. To avoid this, enough context to discover such pattern elements has to be provided (see figure 49).

Here, another constraint node is inserted that allows to navigate to the ComponentType constraint node starting from a context node. So, when creating such a pattern in the target model, the reference from the Component to its ComponentType can be set. See the provided example that is linked in the Appendix in section 8.2.3 (pp. 118).

The Triple Graph Grammars implemented in this thesis will allow to differentiate between context nodes that need to be produced by a previous TGG rule and the constraint nodes that might be matched with existing model elements without this restriction.
4.2.8 Verification and Transformation Synthesis

The issue of verification and transformation synthesis shall be addressed here shortly. Verification was previously emphasized as a key advantage of TGGs over other model transformation approaches. In general, there are many possible verification tasks and a lot of theory from the field of graph grammars or others. Such verification tasks could become or already are interesting topics of further investigation on TGGs. To give examples, two verification techniques that are particularly interesting for TGGs are the following.

Firstly, a set of TGG rule can be analyzed to see if it produces an unequivocal result. There may be rule sets where the source model can be matched in different ways by the source side grammar of the TGG. Then, when the target side grammars produce different target patterns, the transformation is then not deterministic and the transformation result is not predictable. If this is the case, the TGG is not confluent. Heckel, Küster, and Taentzer [HKT02b] described a way to check for confluence in a typed, attributed graph grammar. With such mechanisms, a TGG rule set can be verified to ensure a deterministic transformation result. However, the correctness of a transformation result is not yet verified.

Another sort of verification was recently presented in a diploma thesis by J. Leitner [Lei06]. Here, a transformation with TGGs is formalized as a set of theorems. These theorems can be verified by the theorem prover Isabelle/HOL. So, for a certain transformation, the result can be verified in terms of the theorems. Here, the TGG rule set itself is not verified, but rather the produced transformation result. There is an automated mechanism to check the correctness each transformation output.

In addition to verification, TGG rules can also be synthesized from a set of example source and model instance models. This is subject of the ongoing research at the University of Paderborn [Geb06].

4.3 A model transformation processing scenario

The previous section explained the semantics of Triple Graph Grammar rules. Now, to illustrate how the actual pattern matching and creation is performed, a short processing scenario is shown. This scenario is similar to the QVT processing scenario (see pp. 40), but also reveals some major differences between QVT and TGGs. However, it shows that TGGs can be used as an implementation for QVT.

Firstly, figure 50 shows the start context as well as two example TGG rules from the ctools-to-pnet example. As already mentioned in section 2.2.2 (p. 20), rather than providing a start rule, TGGs need a start context to start a transformation with.
The start context and the TrackToPlaceArcTransition rule have already been introduced. Also, the second rule, ConnectionToArc, is known from the similar QVT mapping in the QVT-Core processing scenario that was presented in section 3.3 (pp. 40).

Now, the following figure 51 shows the initial ctools model where a Project with two Tracks and a Connection is given. Here, the start context already supplies the first correspondence between the Project and the Petri net.

---

**Figure 50: Example TGG start context and rule set**

**Figure 51: ctools to pnet TGG transformation example - Step 1 (start context)**
Now, when a context is given, the TGG rules can be applied. According to the graph grammar semantics, firstly, the context pattern of a TGG rule has to be matched with a pattern in the previously processed model. Then, the right side of the source pattern has to be matched with unprocessed elements in the source instance model. If these two patterns have been successfully matched, the right target and correspondence patterns are created. In this example, the TrackToPlaceArcTransition rule can be applied (see figure 52).

This rule application step seems quite similar to the previous QVT example. Indeed, the result of the transformation step is the same. But, the QVT processing algorithm started by matching the rules' bottom pattern to existing, unprocessed model elements. In contrast, the TGG processing algorithm started by matching the rules' context side with previously processed model elements. These are basically two different ways to approach the rule pattern matching that produce the same result.

The TrackToPlaceArcTransition rule can be applied a second time to translate the second Track. Then, there is sufficient context to translate the Connection (see figure 53).
After the application of the ConnectionToArc rule, transformation terminates. All source model elements have been successfully processed and translated.

### 4.4 Comparing the QVT and TGG processing scenario

Comparing the QVT and TGG processing scenarios shows that in a QVT and TGG transformation, the rules are declaratively applied to a constellation of processed and unprocessed model elements. Even though the matching of each single rule is performed in a different way, the overall transformation steps and the transformation results are the same. Also structurally, QVT-Core mappings and TGG rules are quite similar: Variables in the mapping are represented as nodes in the TGG rule and, furthermore, the guard and bottom patterns in the mapping directly correspond to the left (context) and right sides of the TGG rules.

However, the correspondence nodes in TGGs allow to specify a more fine-grained correspondence between the domain pattern elements than the trace classes in QVT-Core. This advantage can be seen in the above processing scenario: In the QVT-Core mappings (on page 41), there is just one trace class between the bottom patterns of a rule. The trace class for the TrackToPlaceArcTransition mapping is specific for the Track transformation. Then, it reappears in the context of the ConnectionToArc mapping. This means that the ConnectionToArc mapping can only be used for Connections that connect the Ports of Tracks. Now, other Components may be introduced as subclasses (see the ctools model on page 16). In QVT, this would consequently lead to a large number of transformation rules. In contrast, the above TGG rule, ConnectionToArc (see figure 50), does not reference any component-specific context and, thus, can be used for any further components that may exist in the model.
4.5 Summary

This chapter introduced the most important concepts of Triple Graph Grammars and how they are used for model transformation. From this area of application, there emerge further requirements for TGGs. It was shown how constraints on attribute values or attribute value transformations are additionally specified in TGG rules. Further practically relevant issues were discussed, like the matching of subclasses or the matching of existing target model objects. But, some of such further features may violate valuable characteristics of TGGs, like their bidirectionality.

Furthermore, important similarities and differences between QVT and TGGs were shown. Patterns that are described via OCL in QVT are similarly expressed by edges between nodes or by attribute constraints. Also the patterns of the rules are quite similar. And even though the general rule processing approach is different, it was shown in an example scenario that the overall transformation result is equal for corresponding QVT and TGG rule sets.

Chapter 6 will show a formal mapping between QVT and TGGs, but before, the next chapter addresses the redesigned implementation of a TGG interpreter that allows to perform model transformations as described in this chapter.
5 TGG-Interpreter Redesign

The first key objective of this thesis is to implement an improved Triple Graph Grammar interpreter for the convenient transformation of MOF models. The second objective is to align the concepts of TGGs and QVT in such a way that QVT, or rather the declarative parts thereof, can be implemented by TGGs.

The previous chapter already introduced the concepts of TGGs. Now, this chapter is presenting the implementation of a TGG interpreter which results from a redesign of the TGG interpreter used by the ComponentTools project [AWPN04][KRW05] and a Bachelor Thesis by O. Rohe [Roh06]. In the following, a redesign of the TGG model is presented that allows TGGs to reference the MOF domain models of the particular transformation. Afterwards, the redesign of the TGG interpreter is shown. The next chapter then describes how a mapping between QVT and TGGs allows to implement QVT by TGGs.

5.1 TGG Model Redesign

This section introduces the design of the TGG model which is used by the TGG interpreter. Firstly, the elementary graph structure is explained and, then, the pattern structure of the TGG rules. It follows a description of how attribute value transformations can be specified and how this model is kept extensible to add further language constructs in the future.

5.1.1 Graph patterns and MOF models

To specify a transformation between (MOF) models, TGG rules need to be specified with respect to these particular models. Figure 54 illustrates the relationships between the TGG rules and the domain models as introduced in section 4.2 (pp. 50).
A TGG rule is basically an object graph. Figure 55 shows a graph model for object graphs that reference another class model. In object graphs, every node references a certain class as its type and the type of an Edge is a reference from the class model respectively.

However, referencing a class model and performing a transformation on the instance models does not work generically for any programming model. A meta-modeling facility needs to provide both an accessible instance model and an accessible class model. The patterns in the TGG rules reference the class models and it is necessary to operate on the instance model in terms of these meta patterns. MOF-like technologies like EMF, JMI or Fujaba allow to specify a class model and automatically generate accessible code for it. In many other cases, the class model is not easily accessible or it is not even available. Many applications just provide legacy code and is not generically accessible for a transformation application. In such cases, adapters or wrappers will need to be implemented to make foreign models accessible. Adapters are described more thoroughly in 5.3.7 (p. 89).

The TGG model in this thesis, as might have been noticed in figure 54, is based on EMF¹⁶ (The Eclipse Modeling Framework [EMF]). EMF is providing the ECore model, a MOF-like metamodeling facility that is the basis of almost all model driven applications in Eclipse. EMF is also providing many additional features like the serialization of models as XMI [OMG05a] or editor

¹⁶ EMF models can be represented in a UML-like notation as used in this thesis. Especially associations may not conform precisely to the UML notation. A filled diamond, for example, represents a containment in EMF that corresponds to an aggregation in UML.
generation facilities. Further Eclipse projects provide other interesting technologies based on EMF. For example, GMF (The Graphical Modeling Framework [GMF]) allows the automatic generation of graphical editors. GMF was used in this thesis to implement a graphical TGG editor that is presented in section 5.2 (pp. 74). Another argument for the use of EMF is that many projects at the University of Paderborn, like ComponentTools, are integrated in Eclipse.

5.1.1.1 Typed Graphs Prior to the Redesign

The TGG model of the interpreter from the ComponentTools project is not directly referencing class models. Here, the TGG graph model is rather referencing a more abstract sort of type graph as shown in figure 56. A type graph simply consists of node types and edge types. Compared to a MOF model, this graph type has advantages and disadvantages. MOF is already a pretty sophisticated (meta) model. There can be packages, classes, generalization of classes, operations, parameters and much more. A simple graph type with nodes, edges and attributes (omitted here) is less complicated and any model can be reduced to this less complicated model. This has an advantage when other, non-MOF models need to be transformed. Then, model adapters (p. 11) need to make such foreign models accessible to the transformer. They wrap another model in an interface that is generically accessible to find and create object patterns. These adapters may, in some cases, be more simple when their interface has to reflect a simple graph model instead of a MOF model.

However, the reduction to a simple graph model always makes an extra mapping step necessary. And there are further tradeoffs The mapping is often complicated and it will restrict the expressiveness of the transformation rules. The reason is mainly because most object oriented models make use of generalization and then, the simple graph model would have to be extended to become again more MOF-like. It is, in fact, the fundamental idea of MOF to provide a suitable meta model for object orientation. This is another argument why the redesign of the TGG model in this thesis is now directly referencing ECore models. An idea of how model adapters may be designed in this case is given in section 5.3.7 (p. 89).

17 This is not an exact representation of the ComponentTools TGG interpreter model. It is simplified for the comparison in this chapter. See [Roh06] for details.
5.1.2 The TGG rule pattern structure

As mentioned above, a TGG rule is basically a graph where certain nodes (and edges) belong to certain sub patterns. Figure 57 shows the sub patterns in a TGG rule and how they overlap.

![Figure 57: The sub patterns of a TGG rule](image)

Because the TGGs in this thesis are non-deleting grammars (see back on page 46), practically all rule nodes belong to the right side graph pattern. Then, there are the context nodes that also belong to the left side. In addition, there can be three domain sides or even more, since the TGG model in this thesis will actually be a Multi Graph Grammar (MGG) model (see section 4.2.5, p. 54).

The EMF model according to this pattern structure is shown in figure 58.

![Figure 58: Model of the sub graph pattern of a TGG rule](image)

Here, firstly, the GraphGrammarRule is a special kind of graph that has a left and a right graph pattern. A GraphPattern references the graph elements that belong to it, nodes and edges. Secondly, a TripleGraphGrammarRule is a special kind of GraphGrammarRule that subdivides the graph with further graph patterns, the DomainGraphPatterns. Because there can be any number of DomainGraphPatterns, a TGG rule is actually a Multi Graph Grammar rule.
Although, to avoid confusion, the original term Triple Graph Grammar is preferably used in this thesis.

### 5.1.3 Specifying Transformations with Triple Graph Grammars

In addition to the single transformation rules, it is necessary to generally specify which models in which packages are subject of the transformation. A look at the QVT-Base model shown in figure 59 shows how this can be achieved.

![Figure 59: The QVT-Base model](image)

Here, it is specified that a transformation consists of rules and *typed models*. A typed model is the class model that is referenced by the domain sides of the transformation rules. Because a model can be spread over several packages, these packages are combined in a typed model. Furthermore, the QVT base model arranges for transformations and rules to be extended or overridden. QVT does not clearly specify the semantics of such constructs.

The QVT-Base model is the fundamental model for all languages of the QVT specification. Both QVT-Core and QVT-Relational inherit from this model. In this thesis, the QVT-Base model that is specified in the QVT specification is implemented as the base model of TGG transformations as well. The following figure 60 shows how the TGG model is extending the QVT-Base model.

---

18 See [QVT], pp. 26
Now, the TripleGraphGrammarRule inherits from the QVT-Base Rule class. Also, the DomainGraphPattern now inherits from the QVT-Base Domain and, thus, the association between Rule and Domain is inherited this way (see that the particular association as shown back in figure 58 is now omitted). At the root in the TGG model, there is the TripleGraphGrammar. Here, it is a subclass of the QVT-Base Transformation class. The TripleGraphGrammarRules are all contained in a TripleGraphGrammar by the association inherited from QVT-Base.

5.1.3.1 TGG Transformations Prior to the Redesign

The TGG model used by the ComponentTools TGG interpreter does not directly reference any participating domain models, since they are reduced to an abstract type graph. Other domain models are rather referenced by the particular model adapters. Here, however, this architecture shall not be discussed in further detail.

5.1.4 Attribute Constraints

So far, the graph patterns may specify nodes and edges which correspond to objects and references in the domain instance model. But, not yet expressed in this object graph structure is a representation of attribute constraints. Attribute constraints were previously introduced in section 4.2.2 (see pp. 51). In a TGG transformation rule, it is necessary to express additional node matching constraints or attribute value assignments. Figure 61 is repeatedly shows a TGG rule with attribute constraints.
As shown for the “type” attribute of the Ports on the ctools model side, most simply an attribute constraint can check or assign a literal constant value. Also an attribute equality can be checked or transformed as shown for the “name” attributes. Generally, an attribute constraint specifies a value to a slot attribute of a slot node. Figure 62 shows how the slot nodes are marked by the double arrow of the rounded constraint rectangles.

Figure 61: A TGG rule with attribute constraints

Figure 62: Graphical notation of an attribute constraint

The model for such constraints is displayed in figure 63.
As shown here, there is an inheritance hierarchy of different constraint classes. At the top of this hierarchy, there is the Constraint class. This abstract class does not yet specify any attribute values that are subject to the constraint. It is the base class for any further constraints that may be integrated in the future. Because such a constraint does not necessarily need to be associated with a node, the TripleGraphGrammarRule class is the container for the constraints. Also, a constraint may be associated to a graph pattern of the rule, because this determines when this constraint will need to be checked or enforced. Now, the first subclass of the Constraint is the TGGConstraint which references a slot node and a slot attribute that is subject of the constraint. Also, a TGGConstraint may be marked as checkonly. This expresses that a certain value may only be checked during pattern matching, but shall not be assigned during pattern creation. As further subclasses of the TGGConstraint, there are a number of PrimitiveLiteralConstraints that specify a certain constant literal value that an attribute value of a model object must confirm to. Furthermore, to express the equality of two attribute values, there is the EqualsAttributeConstraint. The equality of two attribute values may, however, only be one sort of constraint between two attributes. There might be other operations that define a more complex attribute-to-attribute mapping. Therefore, to keep this model extensible, an abstract superclass, the AttributeConstraint, is introduced. An AttributeConstraint is referencing a certain value attribute of a certain value node. Any sort of constraint between two attributes can be added by extending the AttributeConstraint class.

The constraints shown in this model just provide the most basic attribute value constraint expressions. For the use in practice, however, the expressiveness will have to be increased by providing further constraints. One valuable extension here would be to integrate expressions of the Object Constraint Language (OCL) as explained in the following.
5.1.5 TGGs and OCL

In QVT, the Object Constraint Language (OCL [OCL]) is used to specify pattern matching constraints or attribute value assignments. OCL is a comprehensive query language through which QVT becomes fairly expressive. Expressions in OCL allow to query all sorts of values and conditions in an object model and, thus, an integration of QCL into TGGs is a valuable extension. An implementation would have exceeded the scope of this thesis. However, the foundation for this integration is included in the TGG model.

In QVT-Core, value assignments may look like the following statement.

```plaintext
... place.name := "port_" + inPort.name;
...```

Here, `place` and `inPort` are two variables that represent Place and Port objects. The assignment specifies that the name attribute of the Place object shall be set to a string literal concatenated with the name attribute of the Port object. This statement is similar to the equality attribute constraint in figure 61, but here, there is also a string concatenation operation involved. But, this is just a simple example of an OCL expression. OCL expressions may contain variables, attribute values, constant literal values, operations, if-statements and so on. Typically the subject of such expressions in QVT are the variables. As mentioned before, these variables in QVT conceptually correspond to the nodes in TGGs. So, if nodes could be used as variables in OCL statements, then the integration of OCL is possible. Therefore, the Node class inherits from the OCL Variable class as shown in figure 64.

![Diagram](https://via.placeholder.com/150)

**Figure 64: A TGG Node inherits from the OCL Variable class**

This diagram shows that the Variable class from the ocl.expressions package is itself a TypedElement. So now, the Node is not referencing a type class itself (see figure 55), but the TypedElement from the ocl.uml package references an EClassifier. The class EClass is a subclass of EClassifier and thus, this is no restriction in the model.

Now, OCL expressions can use TGG nodes as variables. A TGG OCL constraint may look like shown in figure 65.
The new OCLConstraint is now another subclass of the TGGConstraint class. So, this constraint specifies a slot node and slot attribute and now any kind of OCLExpression can be used to specify a value for the object's attribute.

The OCL packages introduced here are part of the EMFT project (Eclipse Modeling Framework Technologies [EMFT]). This project supplies, among other things, an OCL parser and validator for ECore models. To fully integrate the support of OCL constraints in TGGs, the EMFT OCL parser/validator would need to be integrated. As mentioned before, this integration could not be implemented during this thesis, but is an interesting feature to be added in the future.

Now, the integration of OCL surely poses a great extension of the expressive power of TGGs. But many expressions that are possible in OCL make it difficult to maintain the bidirectionality of the transformation rules. So, for now only the most basic attribute constraints are implemented as shown in figure 63.

So far, the most important TGG model features have been introduced. In the following, a graphical editor for this TGG model is presented which was developed in the scope of this thesis.

### 5.2 Graphical TGG Editor

The Eclipse Graphical Modeling Framework (GMF [GMF]) allows to flexibly generate graphical editors from ECore models by mapping model elements to graphical nodes, connections and labels. There are many other graphical features that can be specified as well as palette tools and specific editing constraints. Here, GMF will not be explained in detail, but a short overview is given over the TGG rule editor that was generated with GMF.

Figure 66 shows a screenshot of an example TGG rule in the TGG diagram editor.
Figure 66: The TGG diagram editor in the Eclipse workbench

Here, the Eclipse workbench is shown with a TGG diagram editor. On the left, the file navigator shows the projects that contain the TGG rule model files and a corresponding diagram file (the file that is selected in this screenshot). The rule diagram editor as shown is currently opened to edit the known example TGG rule TrackToPlaceArcTransition. For this editor, the alternative Fujaba-like notation for TGG rules was chosen (see pp. 19, 47) because it is more convenient for complex patterns. The toolbar on the right side of the diagram editor shows a list of tools that allow to draw nodes and edges as well as attribute constraints. At the bottom of the workbench window, below the editor, there is the Properties View where further properties of the currently selected elements can be edited.

To get an impression of the editor, a short editing step is described in the following. The online help of the provided software, however, contains a more thorough user guide. The editor software is supplied along with this thesis and can be installed and tested (see the appendix on page 117). Here, figure 67 shows a new TGG node (green) that was connected with a DomainGraphPattern (blue). A DomainGraphPattern represents the domain side of a TGG rule. It may seem awkward to see the DomainGraphPattern represented as a graphical node where it should rather represent a certain pattern area. Indeed, GMF would allow to specify a container figure instead that contains other nodes. However, when the pattern affiliation is expressed as shown, it is much easier to draw the patterns when the rules become more complex.
As shown here, when a node is created with the according tool from the tool palette, a name can be entered and the TGG node has to be connected to a certain DomainGraphPattern node. This then allows to select the desired type class from the particular domain class model. Further settings of the node can then be specified in the Properties View, for instance if the node should be a context/left side node or if an instance object of a subclass shall be matched or not.

Similarly, an edge can be drawn from one TGG node to another as shown in figure 68.

Figure 67: Creating a node in a TGG rule
Firstly, the edge creation tool has to be selected from the palette, then an edge can be drawn from a source node to a target node. The edge automatically belongs to the same DomainGraphPattern as the source node. When drawn, the desired type reference can be selected in the properties view. The selection depends on the type classes of the source and target nodes. Typically, there is only one or not more than a few references that exist between two classes. In the ctools example model, as shown in this screenshot, there is only one type reference that can be selected for this edge that connects a Project node with a Track node.

There are a few more editor features that will not be explained here in any more detail. Now, when the rule set is completely designed, the TGG can be used by the TGG interpreter to perform transformations on the instance models. The architecture, overall mechanisms and usage of the TGG interpreter is explained in the following.

**Figure 68: Drawing an edge between TGG nodes**
5.3 Interpreter Redesign

After introducing the TGG model and the graphical GMF editor, this section now shows how the TGG interpreter software is designed, how a transformation is processed and how the application is conveniently integrated into Eclipse.

The TGG interpreter presented here is a redesign of the interpreter that originated from the ComponentTools project. Previously, this chapter discussed some major changes and improvements to the TGG model. Here, however, the prior interpreter design shall not be discussed. A more detailed comparison to the prior application will be omitted to concentrate on the more important issues.

5.3.1 Transformation processing steps

A TGG processing scenario was shown in section 4.3 (pp. 59). These transformation processing steps need to be performed by the TGG interpreter and can be summarized as follows (see the schematic illustration in figure 69):

![Figure 69: Matching a TGG rule with processed and unprocessed model elements](image)

While there are model elements left to transform in the source model, the interpreter iterates over the TGG rule set and checks if a rule can be applied. A rule can be applied when its context (left side) pattern and source model pattern can be successfully matched. Then, the target model as well as the correspondence patterns can be created (see figure 70).
The transformation terminates either if there are no more source model elements left to transform or if there are no more rules that can be applied.

Now, there is one important optimization that significantly reduces the pattern matching time. When applying a TGG rule, firstly the context (left) side is matched with the previously processed model elements. But, not all source model elements are of concern to start the pattern matching with. In a transformation step, some source elements may have been processed long before and don't need to be considered anymore. So, there are model objects which can be isolated to start pattern matching with (see figure 71).

These objects are namely those processed source objects that have unprocessed neighbor objects. This set of objects is called front, illustrated in figure 71. Here, the edges are highlighted to show that these source model objects are connected to unprocessed objects.

5.3.2 The Interpreter

In the following, the interpreter design is presented. Figure 72 shows the most important parts of the TGG interpreter model.
Firstly, there is the central Interpreter class. To start a transformation, it supplies the method `applyGraphTransformation()`. Furthermore, the Interpreter class is responsible to keep track of the globally matched and transformed model elements. Then, when iterating through the TGG rule set as describes above, a RuleProcessor is initialized for each rule to perform the rule processing. The two methods, `checkPrecondition()` and `applyRule()`, trigger the pattern matching and pattern creation in two steps. After each rule application, the FrontManager is responsible to update the set of source model objects in the front. To determine which models shall be transformed and which TGG shall be used, the Configuration supplies the necessary configuration information. All these interpreter model parts and further model elements are described in the following sections.

The Interpreter is the central class in the TGG interpreter model. During the transformation process, the Interpreter keeps track of the already matched or transformed model elements. Such model elements are either an object or a reference between two objects. Now, when an object is matched by a TGG rule node or created based upon such one, the object is bound to this node. Accordingly, a matched or created reference is bound to its respective edge. These edge- and node bindings have to be maintained throughout the transformation process. They are used to mark the matched or created model elements as processed and thus determine the context that is available for the application of further TGG rules. To maintain these node and edge bindings, the interpreter has a NodeBindingManager and an EdgeBindingManager as shown in figure 73:
The NodeBindingManager and EdgeBindingManager maintain a number of Node- or EdgeBindings and specify a number of type safe convenience access methods. Furthermore, the Node- and EdgeBindings are referenced through qualified associations. A NodeBinding is simply the tuple of a model object that is bound to a TGG rule node. So, the qualified association allows to retrieve a NodeBinding by the NodeBinding's object. This is particularly convenient, because this way, when matching objects in the model, the corresponding NodeBinding, if it exists, can be easily retrieved. The according EdgeBinding, which binds and edge to a reference, may look a bit more complicated. The reason is that, in an object model, a reference does not explicitly exists. A reference in an object model is just a property value of one object that points to another object. Thus, a reference in an object model is determined by the source and target objects and the particular property that holds the pointer. In ECore, as shown in the diagram above, the according EReference from the class model can be referenced to specify the particular property of the object. This triple that symbolizes a reference between two objects is called VirtualModelEdge. Here, the EdgeBindingManager references the EdgeBindings through a qualified association has the VirtualModelEdge as a key. This way, when navigating in the object model, it can be retrieved directly from the EdgeBindingManager to which TGG edge the reference is bound.

One technical restriction in EMF is that qualified associations are not supported or, rather, they can be designed through a workaround. Admittedly, the diagram in figure cheated a little for simplification reasons. The EMF workaround to this problem is a technical issue that shall not be described here in further detail.
Additionally shown in figure 73 is the Rule Processor which also contains a NodeBindingManager and EdgeBindingManager. The RuleProcessor also needs to maintain a number of Node- and EdgeBindings, but only for the scope of one rule application. So, there are two sorts of bindings that may be distinguished. There are global bindings, maintained by the Interpreter for the scope of the whole transformation, and local or rule specific bindings, maintained by the RuleProcessor. After the successful application of a TGG rule, all local bindings become global bindings. The RuleProcessor is introduced in the following.

5.3.3 The RuleProcessor

During a transformation, as mentioned above, the Interpreter iterates over the TGG rule set and initializes a RuleProcessor for each particular TGG rule to process this rule. The parts of the interpreter model associated with the RuleProcessor are shown in figure 74.

![Diagram of RuleProcessor and related parts](image)

Figure 74: The RuleProcessor and related parts in the interpreter model

The RuleProcessor is responsible for the processing of one rule. Because of this, one particular TripleGraphGrammarRule from the TGG model is referenced by the RuleProcessor. Furthermore, it is the responsibility of the RuleProcessor to match and create the graph patterns of the TGG rule. Therefore, as mentioned above, it keeps track of the node and edge bindings through its NodeBindingManager and EdgebindingManager. Then, the RuleProcessor also needs to check or enforce certain constraint conditions formulated in the rules. These constraints are handed to the ConstraintsProcessor which will be explained shortly. Before, the pattern matching is briefly explained in the following.

5.3.3.1 Pattern matching

The processing of a TGG rule involves the pattern matching and pattern creation as described in section 4.2 (pp. 50). The problem of subgraph pattern matching is computationally quite complex and can lead to a runtime which is not acceptable in some areas of application. Some issues on the complexity and optimizations are discussed in the following.
The subgraph pattern matching problem is an NP-complete problem. For every potential node match, each neighbor in one pattern has to be matched with a neighbor in the other pattern. Then again, for each potential node match found, the same has to be checked recursively. When matching object graphs, however, the graphs are typed (edges as well as nodes). Through their types, the graphs are more (or sometimes less) structured. Compared to other areas of application, for example visual pattern recognition, this results in a significant reduction of potential matches. Also, most rule patterns are usually quite small so that the recursion depth is quite low as well.

TGG transformations typically become slow, when there exist many objects in the front that can be matched with a rule's source context nodes. This is illustrated in figure 75.

![Figure 75: Find a start match for pattern matching](image)

To start the pattern matching, firstly, one object in the front has to be matched with a precondition node in the source model side of the TGG rule. See the encircled patterns in figure 75. This leads to \( n \) times \( m \) potential start matches for just one rule. This number is multiplied by \( k \) if there are \( k \) many rules. As said, once a successful start match is found, the pattern matching time is usually acceptable for small rules. But, there is a lot of potential to reduce the transformation time if the retrieval of initial matches could be optimized. Here, different model structures allow different optimization strategies. Different heuristics could be tested and implemented in the future.

### 5.3.3.2 Maintaining Node and Edge Bindings

A RuleProcessor maintains node and edge bindings during the processing of one rule to memorize the matched nodes and edges. After the successful processing of the rule, the rule specific node and edge bindings are copied to the global node and edge bindings that are maintained by the Interpreter. Then, practically the whole RuleProcessor and its node and edge bindings might be disposed to free memory. However, it is possible to maintain the rule specific node and edge bindings. This could be particularly interesting when performing incremental model transformations (see p. 53): In a scenario where a number of models need to be permanently synchronized, it could be necessary to undo and redo a certain number of
transformation steps. This is possible by maintaining this information in the particular RuleProcessors.

5.3.4 The ConstraintsProcessor

The attribute constraints in the TGG model were already introduced in section 5.1.4 (pp. 70). In particular, there are constraints that expressed a constant string-, integer- or boolean literal attribute value. Also, there exists a constraint that specifies the equality of two attribute values. Furthermore, an approach was presented how this rather basic set of attribute constraints could be extended by constraints that can specify an attribute value via OCL. The mechanisms to process such constraints are presented in the following.

Figure 76 shows the architecture of the ConstraintsProcessor. As mentioned above, each RuleProcessor has a ConstraintsProcessor which is responsible for the processing of all constraints that are formulated in the rule.

As shown in this diagram, the ConstraintsProcessor contains a number of ConstraintProcessors. Similar to a RuleProcessor, which is initialized for each particular TGG rule, there is a ConstraintProcessor initialized for each constraint in a rule. For this reason, a ConstraintProcessor is associated with one Constraint from the TGG model. The ConstraintsProcessor and the ConstraintProcessor have a number of methods to trigger the checking of the constraints. During pattern matching, the constraints in the rule only need to be checked. Then, secondly, there are methods processConstraints() and processConstraint() that trigger the enforcement of the constraints. This now means that attribute values are assigned to the slot model objects. In this process, depending on the nature of the TGG rule, there can be
different values assigned to an attribute multiple times. This is not a priori forbidden in the TGG rules, but such value clashes pose an illegal operation in the transformation and an exception will be thrown\(^\text{19}\).

Now, there are two subclasses of the ConstraintsProcessor. One is the TGGConstraintsProcessor class that works with the subclasses of the TGGConstraintProcessor to process the previously mentioned constraints. There is one concrete TGGConstraintProcessor subclass responsible for handling StringLiteralConstraints, IntegerLiteralConstraints, BooleanLiteralConstraints and EqualsAttributeConstraints. The second subclass of the ConstraintsProcessor is the ExtendableConstraintsProcessor. In general, this architecture may be extended by adding further ConstraintsProcessor and ConstraintProcessor subclasses. The ExtendableConstraintsProcessor, however, represents an approach to allow other specialized ConstraintProcessor to be plugged into the interpreter. So, when the TGG is extended by further sorts of constraints, another Eclipse plug-in may deliver a special ConstraintProcessor for these extended constraints. (This way, also black box\(^\text{20}\) operations, arbitrary Java code, can be integrated into TGGs.) The ExtendableConstraintsProcessor is then responsible to assign each TGG constraint to the correct ConstraintsProcessor. The plug-in dependencies in this case are illustrated in figure 77.

![Figure 77: The plug-in dependencies when extending the TGG constraints and the according processors](image)

Shown on the left in this figure is the TGG Model and how it could be extended by certain constraint constructs. On the right side, there is the TGG interpreter that interprets the TGGs. Here, an extension to the ConstraintProcessor is accordingly possible that can be delivered to the TGG interpreter through the Eclipse plug-in mechanism.

This way, also an OCL constraint processor for OCL constraints could be plugged into the interpreter. As mentioned before in section 5.1.5 (pp. 73), it could be possible to formulate QCL attribute constraints with OCL expressions provided by EMFT [EMFT]. The parser and validator from the EMFT project could then be integrated in a constraint processor plug-in.

---

\(^{19}\) This corresponds to the enforcement semantics of QVT. See [QVT] p. 20

\(^{20}\) See [QVT] p. 9
5.3.5 The FrontManager

After each TGG rule processing, there are new nodes and edges bound to the domain model elements. All these new node and edge bindings may be of concern when processing the next rule, because they may represent an adequate context for that next rule. But, as mentioned previously, to start the pattern matching, only the node bindings in the front are important. These are the source model node bindings who's model objects have unbound (unprocessed) neighbors. Therefore, all the new node bindings from a successfully terminated rule application are handed to the FrontManager. The FrontManager then performs an update on the new node bindings and those that are currently in the front. Then, after the minimization of the front, the next initial match of a rule node with a node binding in the front can be searched.

In some cases, the front may become quite large. Depending on the structure of the source model, an intelligent sorting of the front node bindings may significantly reduce the matching time. It was previously mentioned in section 5.3.3.1 that optimized initial node matches could be found through heuristics. So, the improvement of the FrontManager represents another interesting task for further investigation.

5.3.6 The Configuration

The Configuration of an interpreter provides the details of the transformation that shall be performed. Schematically, a TGG interpreter configuration shown in figure 78.

![Figure 78: The parts of a TGG interpreter configuration](image)

A Configuration mainly specifies two things: Firstly, it references a Triple Graph Grammar which shall be used for a particular transformation. Secondly, it provides the start context of the transformation. This provides an access to all the instance models which are subject of the transformation. Additionally, a Configuration also specifies which domain models are source and target of the particular transformation setting.
Figure 79 displays the Configuration class in the interpreter model architecture.

As shown in this diagram, a Configuration references one TripleGraphGrammar and it contains a number of DomainModels. DomainModels represent the particular instance models that shall be transformed. Because a TGG transformation always needs a start context to start the transformation (see section 2.2.2, pp. 20), the root object(s) of the target model(s) need to be present as well as the root object(s) of the source model(s). Furthermore, a start context needs to include a valid node and edge binding for each object and reference in the model. Through the target- and sourceDomainModel references, the Configuration specifies which of the DomainModels is a source or target models of the current transformation setting. Also, there is always one CorrespondenceModel that needs to be specified.

Manually specifying a start context for each transformation configuration may be a costly task. In contrast, this would be the first transformation step in QVT. The matching process of the TGG interpreter would yet need some modifications to process such start rules that do not contain any context. This will be subject of further improvements to the software. But, this thesis provides utility methods that facilitate the creation of the start context. There is an Eclipse wizard available to generate a configuration step by step. This New Configuration Wizard can be used in the following scenario, see figure 80 and 81.
EMF allows to serialize the ECore models in XML files. In the figure above, a screenshot shows the model files from the ctools example (see section 8.2.1, p. 117 in the appendix). Here, there are the two source and target model files as well as the TGG rules.

Now, when the source and target models are stored as such files somewhere in the Eclipse workbench, an interpreter configuration with the start context can be persisted alongside as XML as well. The New Configuration Wizard will support the creation of an interpreter configuration with the following step shown in figure 81.

Firstly, the user can select the source and target model files. In the third step, the TGG file can be specified. Next, one or several possible correspondence nodes are displayed which could exist between the source and target root model elements. The user may select an appropriate correspondence node. Then, the output of this wizard is a correspondence instance model file and the interpreter configuration file. Now, when the start context is serialized like this, it can be loaded into the interpreter to start a transformation. The following screenshot (see figure 82) shows the created configuration file as well as the context menu action to start the transformation.
The Eclipse UI support shown here, however, is just one way of integrating the TGG interpreter. The whole interpreter architecture may be flexibly integrated into other Eclipse-based applications. Then, if the in-memory domain models can be transformed directly.

5.3.7 Model Adapters

EMF and the ECore model provide a convenient meta modeling facility. In this implementation, the transformation rules can directly reference the domain class models and the interpreter can generically operate on the generated code. It was mentioned previously that this does not work generally with any Java application model. However, model adapters can be implemented that would allow to operate on other models. The basic approach and possible further work will be explained in the following.

ECore provides a *reflective API* for its models that allows an application to operate generically on these models through a set of generic methods. Figure 83 shows the central classes from the ECore model that display this interface.
The EPackage is the root element in any ECore model and is always associated with an EFactory. Now, this EFactory, or the respective implementation for the particular generated code, is able to create a model object with its create() method. As a parameter, this method needs an EClass from the EPackage. The create() method can also initialize EDataTypes and and load data values from or save values to strings, but this shall not be of further concern here. The EObject is the superclass of any model class in ECore. Here, most important to navigate and operate reflectively in the model, are the eGet()- and eSet() methods. These methods take an EReference or EAttribute from the class model as parameters. So this way, with the structural information supplied with the class model, it is possible to perform any operation on the model. There are further important methods in this interface, but these shall not be described here in further detail.

Now, the TGG interpreter can operate on the ECore reflective API, but, when integrating non-EMF based tools, it cannot access any Java model. In this case, as mentioned previously, model adapters will have to be implemented. One solution is apply the adapter (or wrapper) design pattern so that there is an ECore adaptor class for every foreign adaptee class. Schematically, this is illustrated in figure 84.
Figure 84: ECore model adapters for foreign Java code

Here, the foreign Java code is shown on the right. There might be a corresponding foreign class model of some sort. But, sometimes this class model is not available, or the Java code was written by hand. So, if this foreign class model exists, it may be transformed into an EMF model. For example can EMF convert Rational Rose models be into ECore. If the foreign class model does not exist, it has to be reverse engineered from the provided code. Then, when the according EMF class model is created, the corresponding EMF code can be generated for this class model. This code now has to be modified so that the generated classes form adaptors for the foreign model Java classes. An adaptation of the EMF code generation facilities could facilitate this programming work.

With this approach, there might be an additional adaptor object in the memory for each model object. This could become a memory problem for large instance models. To avoid this, other, more lightweight adapter mechanisms are possible. For example could adaptor objects be instantiated only for the model parts where the interpreter is currently operating.

The Model Transformation Framework (MTF [MTF]) is another model transformation facility for EMF models. MTF is actually adapting foreign models through the mechanisms presented above. In addition, the EMF reflective API is extended by the MTF. For example may foreign Java classes require special constructor parameters. Then, extra methods exist to handle the object initialization.

5.4 Summary

In this chapter, the software components were described which have been implemented in the scope of this thesis. The TGG interpreter as well as the TGG model provide a flexible and extendable software design. Possible extensions and further optimizations for this tool were
presented, for example how the TGG interpreter could be extended by an OCL processor. The integration of efficient heuristics for the pattern matching and rule application are further important tasks in the future.

Furthermore, a rich graphical TGG editor was presented. Together with the Eclipse UI support, like the New Configuration Wizard, these components already form an efficiently usable tool for the design and application of TGG model transformations in Eclipse.

The next chapter will now show how this technology implements significant parts of the QVT specification.
6 Reconciling TGGs with QVT

Until now, this thesis described and compared QVT and TGGs and presented the implementation of a TGG interpreter for MOF model transformations. In this tool and the underlying TGG model, already some QVT-like concepts were integrated. This prepared for the final step towards implementing QVT by TGGs. This chapter now presents a transformation from QVT rules into TGG rules which can then be applied by the TGG interpreter.

This transformation, however, does not yet provide a complete mapping of all the QVT language constructs. For example, the full expressive power of OCL which is available in QVT cannot yet be expressed by TGGs. But, approaches are presented how these issues can be resolved and do not pose any general restrictions to this mapping.

Now, there are two candidates for this mapping, QVT-Relational and QVT-Core. Because QVT-Core is structurally more similar to TGGs, it is the primary objective to specify a transformation of QVT-Core to TGGs. QVT provides a mapping from QVT-Relational to QVT-Core and therefore the mapping from QVT-Relational to TGGs is achieved transitively. However, the approach of a direct mapping from QVT-Relational to TGGs shows that such a transformation is possible as well.

6.1 Mapping QVT-Core to TGGs

The overall rule structure of QVT-Core mappings, already introduced in chapter 3, is similar to the structure of TGG rules. In the following, firstly a mapping of the overall rule patterns in QVT-Core and TGGs is shown. Then the transformation of the single pattern expressions is specified. Here, object diagrams of the QVT-Core and TGG rule structures are showing how the transformation rule models relate to each other. Then, some of the actual TGG rules that transform QVT-Core to TGGs are explained.
This section, however, does not cover every transformation step in detail. The Appendix links to the transformation examples which are provided with the software (see section 8.2.2, pp. 117). There, the complete list of QVTCove-to-TGG transformation rules can be found.

One general issue that should be kept in mind is that the QVT-Core to TGG mapping is rather surjective and the transformation not fully bidirectional. In QVT-Core, the domain pattern sides can each be marked either as checkable or enforceable. So, depending on the transformation direction, the QVT-Core patterns will also contain different pattern constructs. However, such a distinction is not made in TGGs and different transformation directions that are indicated by a QVT-Core mapping will have no influence on the resulting TGG rule.

6.1.1 The Rule Pattern Structure Transformation

Mappings as well as TGG rules specify context patterns and patterns that are matched or created in the source and target model. Schematically, the comparison of a QVT-Core mapping and a TGG rule structure is illustrated in figure 85.

![Figure 85: Schematically comparing the QVT-Core and TGG rule structure](image)

As shown in this figure, the domain sides in QVT-Core, called areas, correspond directly to the domain graph patterns in the TGG rule. The guard and bottom patterns that exist inside each area in QVT-Core correspond to an intersection of several patterns in a TGG rule: A guard pattern corresponds to the TGG rule pattern that is the intersection of the left-, the right- as well as the
particular domain graph pattern. Accordingly, the bottom pattern corresponds to the right and non-left part of the particular domain graph pattern.

To show this correspondence on the model level, a closer look at the QVT-Core model is necessary. The QVT-Base model, which is a foundation of the QVT-Core model, was already introduced in section 5.1.3 (pp. 69). As well as the QVT-Base model, the QVT-Core was implemented with EMF as a basis for the mapping shown here. At the time of developing this mapping, no thorough model of any other QVT transformation tools was available. But, the QVT-Relational to TGG mapping will, in contrast, make use of a third party rule model. The details are presented in section 6.2 (pp. 93). Figure 86 now shows the rule pattern structure in the QVT-Core class model.

Figure 86: The Mapping structure in the QVT-Core model

Here, the Transformation and Rule form the QVT-Base package are implemented by the CoreTransformation and the Mapping in QVT-Core. The domain sides in a Mapping are implemented by CoreDomains that inherit from the Area class. This class supplies a GuardPattern and a Bottom pattern. Also, the Mapping class inherits from Area and thus has a GuardPattern and a BottomPattern. A closer look on the pattern model in QVT-Core will be taken shortly, but before, the QVT-Core mapping structure will be compared to the TGG rule structure at the model level.

Now, the rule pattern correspondence shown figure 85 as an object model look like shown in figure 87.
As seen here, the CoreDomain corresponds one to one to the DomainGraphPatterns. Additionally shown is that each CoreDomain and accordingly each DomainGraphPattern reference a TypedModel as they both inherit from the Domain class in the QVT-Base package (see section 5.1.3, pp. 69). In QVT-Base, the TypedModels are defined at the top level along with the Transformation. So, the Domains from each rule reference the same particular TypedModels in the Transformation. The containment hierarchy is additionally displayed in the notation of this object diagram for a better understanding. When looking at the remaining patterns, this diagram shows that there are multiple Guard- and BottomPatterns, one for each domain side. These correspond to just one LeftGraphPattern and RightGraphPattern in the TGG rule. To see the correspondences in detail, the TGG transformation rules for this rule structure mapping are explained in the following.

Before explaining the transformation rules, figure shows the start context that need to be set prior to the transformation.

Figure 87: The QVT-Core and TGG rule structure object models

---

**Start context**

Figure 88: Start context of the QVT-Core to TGG transformation
In the following, the TGG transformation rules are presented with diagrams drawn with the previously introduced TGG editor (see section 5.2, pp. 74). The first transformation rule that is shown transforms the QVT-Core TypedModel objects into TypedModel objects in the TGG model (see figure 89).

![Figure 89: The TypedModelToTypedModel transformation rule](image)

This is a rather simple rule. The TypedModels are contained by the CoreTransformation or TripleGraphGrammar respectively which are provided here as context nodes. During the transformation of each single TypedModel, their names are also transformed. This is done by the equal-attribute constraints represented by the rounded rectangles. There are equal-attribute constraints in both the forward and backward direction, even through this transformation is not fully bidirectional. However, both transformation directions are prepared for, because there might be future application scenarios where, after a complete transformation, model changes have to be propagated by incremental transformations (see section 4.2.3, p. 53). In this case, single transformation steps may have to be transformed in both directions.

Notably furthermore is that also the correspondence nodes are connected here. In fact, the whole correspondence model in this example provides a thorough containment hierarchy. This has the advantage that also the correspondence model can be conveniently persisted after the transformation by the mechanisms provided by EMF.

The next rule translates the EPackages that are specified by the TypedModel. This rule may seem a bit odd compared to other TGG rules, but is actually a valid TGG rule with two source sides and one target side (see figure 90).
As seen here, the two TypedModels from QVT-Core and TGG both reference the same domain model. Because of this, the ECore class models are treated as a separate source domain side in this rule. When the rule is applied, the context nodes are matched as well as the EPackage node. The gray EPackage node shown here is a constraint node as explained in section 4.2.7.2 (see pp.57). This is because this Triple Graph Grammar does not specify a complete grammar for the domain class models: The EPackage node is neither a node that shall be created, nor is it a node that was previously processed by another rule. When the rule is applied, only the reference from the TGG-TypedModel and the EPackage is created.

Now, the QVT-Core mappings have to be translated to TGG rules. This is specified next in the MappingToTGGRule rule (see figure 88).

Here, if the start context is given, each Mapping in the CoreTransformation is translated into a TripleGraphGrammarRule. Also the Left- and RightGraphPatterns of the TGG rule are already created here. In contrast to guard and bottom patterns, there only exists one left and right graph pattern in a TGG rule and, thus, they are created right away. In the next transformation rule, the
Left- and RightGraphPatterns reappear as context nodes in the next rule that translates QVT-Core CoreDomain to the TGG DomainGraphPattern.

![Diagram of CoreDomainToDomainGraphPattern rule]

Figure 92: The CoreDomainToDomainGraphPattern rule

In this diagram, the edge labels are deactivated for more clarity. Here, the CoreDomain in a QVT-Core Mapping is mapped to the DomainGraphPattern of a TGG rule. The GuardPattern in the CoreDomain is set in correspondence with the existing Left- and RightGraphPattern in the TGG rule and the BottomPattern is set in correspondence only with the right graph pattern in the TGGRule. These correspondences are necessary to transform the pattern elements into the correct TGG patterns later on. Further context nodes here are the previously transformed TypedModels. These have to be referenced by the CoreDomain and the DomainGraphPattern to specify which domain model side they represent.

These rules presented above perform the mapping of the QVT-Core and TGG rule structure as shown in figure 87. The transformation of the pattern variables and simple expressions is explained in the following.

### 6.1.2 Transformation of Pattern Elements and simple Expressions

Now, the QVT-Pattern elements are translated in their corresponding TGG pattern constructs. Primarily, variables are translated into nodes and predicates and assignments are translated into...
edges or attribute constraints. As an example, figure 93 shows an excerpt of a QVT-Core mapping and the corresponding TGG rule construct.

![Diagram](image)

**Figure 93: Comparing a simple pattern in QVT-Core and TGG**

Here, a simple pattern from the ctools example (see the ctools model on page 16) is showing a Project in the context side of the QVT and TGG rules. In the bottom pattern of the mapping and the right side of the TGG respectively, there is a Track which is contained in the Project. In the textual representation, this QVT-Core pattern looks like this:

```java
map TrackTo...
{
  check enforce ctools(pr:Project){
    realize tr:Track|
    tr.componentToProject := pr;
    ...
  }
  ...
}
```

QVT-Core distinguishes between checkonly and enforceable patterns. Checkonly patterns may contain *variables* and *predicates* to express certain pattern conditions. Here, for example, the variable `pr:Project` is just a (checkonly) variable. But, enforceable bottom patterns typically contain *realized variables* and *assignments* (see section 3.2.5.3, pp. 36). The realized variables specify the instance model elements that shall be created and assignments specify object references or attribute values that shall be set.

Before showing the above schematic comparison on the object diagram level, further parts of the QVT-Core model have to be explained. Figure 94 shows the guard and bottom patterns in the QVT-Core model.
Figure 94: The variables and expressions in the Guard- and BottomPattern

All the Patterns in a QVT-Core mapping inherit from the Pattern class that is specified in the QVT-Base package. A Pattern may contain Variables from the QCL Expressions package as well as Predicates which contain an OCL expression with a boolean return value. Now, BottomPatterns also contain Variables and Predicates, but may also contain RealizedVariables and Assignments. RealizedVariables are simply a specialized form of Variables that may be created in an enforceable pattern. Assignments are the enforceable counterparts of Predicates and, in the textual notation, always have the following form (also compare the previous listing):

\[
\text{<slotObjectExpression>.<property> := <valueExpression>;}
\]

Assignment have a *slot expression* which specifies a certain model object as the subject of an assignment. Then, the property specifies a property of this model object that shall be assigned. In the ECore metamodel, the class EStructuralFeature represents properties. At last, the value which shall be assigned is expressed by another OCL expression. This expression has to return a value which conforms to the particular property.

In the following the transformation of variables and realized variables to nodes is explained. Then, an example is given of how assignments and predicates are transformed into TGG edges or attribute constraints.

### 6.1.2.1 Variables to Nodes

In the above mapping fragment listed on page 100, there are two variables. The variable \( pr \) is representing a Project an is located in the guard Pattern. The other variable \( tr \) is a realized variable in the bottom pattern which is representing a Track. The object structure of this pattern fragment is shown in the left part of figure . In addition, figure 95 shows the corresponding
pattern fragment in a TGG rule and the domain model classes that both the variables and nodes reference.

![Diagram of QVT-Core Mapping and TGG rule](image)

**Figure 95: Variables and Nodes**

The right side of this figure also illustrates how a variable in a guard pattern corresponds to a node in the TGG rule. Here, both the left and right graph patterns reference the Project node whereas the Track node is not referenced by the left graph pattern as it corresponds to the realized variable in the bottom pattern.

Now, figure 96 shows a transformation rule that translates a guard pattern variable into a TGG node.

![Diagram of GuardPatternVariableToNode transformation rule](image)

**Figure 96: The GuardPatternVariableToNode transformation rule**

Here, the Variable is a child of the GuardPattern according to the object diagram in figure 95. The Variable is translated into a Node in the TGG rule. The context nodes in this rule provide the appropriate TGG graph patterns that this Node shall be added to. Firstly, this is the DomainGraphPattern that corresponds to the CoreDomain in the mapping. Furthermore, there
are the LeftGraphPattern and RightGraphPattern that were previously associated with the GuardPattern.

To complete the transformation of a variable into a node, the reference to the particular type class has to be set. This rule is similar to the usedPackage rule back on page 98 and will be omitted here for brevity.

### 6.1.2.2 Assignments to Edges

Assignments in QVT-Core assign values to certain properties. These properties may be either references to other objects or data values. Therefore, assignments may be either transformed into edges, which represent references in TGGs, or attribute constraints. Figure 97 shows an extended version of the object diagram in figure 95.

This figure corresponds to the mapping fragment shown on page 100. Now, the assignment in the listing

```
tr.componentToProject := pr;
```

is shown as a child of the BottomPattern of the Mapping. The slot Variable `tr` is referenced through an OCL VariableExpression. Then, the property that shall be assigned is the EReference `componentToProject` from the domain class model. Last, the value of this Assignment is expressed by another VariableExpression that references the `pr` Variable. This reference assignment directly corresponds to an edge in a TGG rule. This transformation is achieved by a TGG rule shown in figure 98.
Here, two Variables that were previously transformed into Nodes are supplied as the context. Then, the assignment construct with its source and target VariableExpressions is transformed into an Edge. The EReference which is specified as the property if the Assignment is again a constraint node, similar to the EPackage node in the usedPackage rule on page 98. In the TGG rule, this EReference is the then set as the type reference of the edge. In addition to the Nodes, the RightGraphPattern is provided as context. In this case of translating an assignment which is in the bottom pattern of a mapping, the corresponding edge will belong on the right graph grammar side of the TGG rule.

In addition to this rule, there are rules that translate edges from predicates in the case that there is just a checkonly mapping pattern to be is transformed. Furthermore, in some cases, an assignment or predicate actually corresponds to two edges. In figure 93, there are two edges drawn as a counterpart for the assignment in QVT-Core, because this reference in ctools is a bidirectional reference (see the ctools model on page 16). In QVT, it is sufficient to just specify one assignment, but in TGGs it is (currently) necessary to draw both a forward and backward edge. Here, these transformation rules will not be discussed any further. The Appendix in section 8.2.2 (p. 117) links to the respective transformation rules can be found in the supplied software.

6.1.2.3 Assignments to Attribute Constraints

The above transformation rule translates such assignments that assigned object references. In contrast, if an assignment assigns a data value to an attribute, this is translated into an attribute constrain in TGGs. Firstly, the following listing shows an extended mapping fragment (see page 100) with an example string assignment.

```plaintext
map TrackTo... {
```
This mapping fragment might be part of a rule where it is particularly important to state that the Track is a "startTrack". In this ctools example, this condition is expressed by the name attribute. The corresponding object diagram to this string value assignment is shown in figure 99.

**QVT-Core Mapping**

```
  Mapping
  |-----CoreDomain
  |       |-----GuardPattern
  |       |         |-----BottomPattern
  |________|
```

**TGG rule**

```
  TripleGraphGrammarRule
  |-----DomainGraphPattern
  |       |-----LeftGraphPattern
  |       |-----RightGraphPattern
  |     |-----Node
  |     name="tr" type
  |     slotExpression
  |     valueExpression
```

**Figure 99: A string assignment and the corresponding string attribute constraint**

Here, the assignment again references the RealizedVariable tr with its slot VariableExpression. But the value expression now is an OCL StringLiteralExpression that specifies a string value. The property that is now referenced by the Assignment is an EAttribute that specifies the name value of the Track class. In the ctools model, this attribute is actually inherited from a superclass of Track, but this shall not be of concern in this example. On the TGG side in figure 99, there is again the Node that corresponds to the RealizedVariable tr. This node is now the slot node of a StringLiteralConstraint that is the counterpart of the Assignment. The StringLiteralConstraint similarly specifies a stringValue and references the same Attribute from the domain class model.

Figure 100 shows the TGG rule that is used to transform a string attribute assignment into the StringLiteralConstraint.
In this rule, the StringLiteralConstraint corresponds to the Assignment as described above. Similar to the reference assignments the EAttribute is also referenced by the StringLiteralConstraint. The AttributeEqualConstraints in this rule are responsible to transform the respective string value that the StringLiteralExpression specifies. Furthermore, this rule requires that a mapping's BottomPattern was previously matched with a TGG rule's RightGraphPattern. The StringLiteralConstraint, which corresponds to the Assignment as described above, is contained by the TripleGraphGrammarRule and is part of the RightGraphPattern (see the TGG model in section 5.1.4, pp. 70).

Now, all the major transformation steps in the QVT-Core to TGG transformation were explained. For some of the above transformation steps there is more than one rule involved to handle special conditions that were not discussed here. The provided example can be consulted to inspect the complete set of transformation rules (see 8.2.2, p.117). In the remaining chapter, an approach is presented that shows how QVT-Relational can also be transformed directly into TGGs.

6.2 Mapping QVT-Relational to TGGs

In addition to the QVT-Core-to-TGG transformation, QVT-Relational can also be transformed directly into TGGs. The provided software (see the appendix section 8.2.4, p.118) contains an example that implements the first steps of this transformation. Especially interesting here is that a QVT model is used from a tool that was developed lately by the ModelWare consortium [ModelWare] [MWQVT06]. In addition to an EMF-based QVT-Relational model, this tool also
provides a text parser based on the Universalis project [Bel99]. The QVT tool was recently published under the Eclipse Public License (EPL) and will presumably be available shortly in the context of the Eclipse Model Driven Development integration project (MDDi [MDDI]). In the following, an approach of this transformation is explained.

In the transformation from QVT-Relational directly into TGGs, the transformation of the pattern expression is not much different from the transformation of the QVT-Core statements as presented previously. The main additional challenge in this transformation is that the correspondence model needs to be derived from the QVT relations. This is shown in figure 101 which compares a relation pattern to a TGG pattern.

Here, the variable in the relation is not referenced by an correspondence node or trace class as in QVT-Core. In a transformation to TGGs, this Variable is translated into a Node which needs to be referenced by a correspondence node. In this case, there is only one correspondence node created per TGG rule.

The first two steps in this transformation are described in the following. However, further issues in this transformation, like the unfolding of where-clause relation invocations (see section 3.2.2, pp. 26), are not covered in here.

The transformation from QVT-Relational into TGGs is achieved by transforming from a given relational transformation into a Triple Graph Grammar and, simultaneously, into the according correspondence Package. Thus, the transformation will actually be a 1:2 model transformation. Figure 102 schematically presents the first transformation rule.

**Figure 101: Comparing a QVT-Relational and TGG pattern**

The first two steps in this transformation are described in the following. However, further issues in this transformation, like the unfolding of where-clause relation invocations (see section 3.2.2, pp. 26), are not covered in here.
This rule shows how a Relation is transformed into a TGG rule which contains the single correspondence node right away. Simultaneously, the according correspondence EClass is created, which is the type class of the correspondence node. The context in this rule, actually the start context of the transformation, contains the root elements of the three participating domain models: There is firstly the RelationalTransformation which contains the relations. This corresponds directly to the TripleGraphGrammar which contains the TGG rules. Also, an initial correspondence-EPackage is provided which contains all the type classes of the correspondence model. Omitted here are for example the graph patterns, references to the typed models.

Next, when transforming the single variables that occur in the relation domain patterns, they are translated into TGG nodes. This alone is quite similar to the QVT-Core-to-TGG transformation. But, also the corresponding edge from the correspondence node will have to be created (see again figure 101). The according transformation rule is shown schematically in figure 103.

This rule shows how, in the second step, a variable is translated into a TGG node. Also, an edge is created that connects the correspondence node with the domain node that corresponds to the variable. Simultaneously, there is an EReference created for the correspondence class which serves a the type reference for the TGG edge.
6.3 Transforming further Pattern Constructs

There are further constructs in QVT-Relational and QVT-Core that need to be transformed to complete the QVT to TGG mapping. As mentioned previously, there are some unresolved issues that are briefly mentioned here.

Firstly, there is the issue of how many-valued references are handled by QVT. For the purpose in this thesis, previous examples of QVT relations avoided the use of set-value expressions. This listing shows a fragment of the example relation listed on page 18 and 25.

```plaintext
top relation TrackToPlaceArcTransition{
    checkonly domain ctools track:Track {
        componentToProject=project:Project{},
        componentToPort=inPort:Port{},
        componentToPort=outPort:Port{}
    };
    ...
}
```

This fragment shows the ctools pattern description of the TrackToPlaceArcTransition relation. In this pattern, the Track component has two Ports through its many-valued componentToPort reference (see the ctools model on page 16). Therefore the two equality predicates should rather look like shown here:

```plaintext
top relation TrackToPlaceArcTransition{
    checkonly domain ctools track:Track {
        componentToProject=project:Project{},
        componentToPort=Set(Port){
            inPort:Port{},
            outPort:Port{}
        }
    };
    ...
}
```

Here, the value assigned to the componentToPort is a Set which contains the two Ports. However, the QVT specification just provides insufficient examples on the usage of such constructs and, furthermore, it is not clear how these set-valued constructs are translated from QVT-Relational into QVT-Core. Therefore, the examples in this thesis rely on the representation shown in the first previous listing, even though this might not be the notation intended by QVT. When the finalization of this standard clarifies the usage of many-valued property expressions, it will pose no problem to also translate these constructs into TGGs.

The second yet uncovered issue in QVT into TGG mapping is that OCL provides an expressive power which is not yet available in TGGs. The transformation so far only allows to translate the most important pattern expressions into TGGs. However, section 5.1.5 (pp. 73) and section 5.3.4 (p. 85) presented an approach on how OCL can be integrated into TGGs. With this integration of OCL, all domain pattern expressions could be translated into TGGs as well. TGGs would then provide a comparable expressive power.
6.4 Summary

This chapter presented how a transformation from QVT to TGGs could be achieved. The basic transformation steps in the QVT-Core to TGG transformation were explained. Because the QVT specification provides a mapping from QVT-Relational to QVT-Core, transitively also a mapping from QVT-Relational to TGGs is covered. However, also an approach was presented that showed that a direct transformation from QVT-Relational to TGGs is possible.

The transformation presented here covered the major parts of QVT. But, there are a number of QVT language constructs which were not yet covered by the transformation. But, most of the issues remaining just require some technical extension to TGGs and do not represent a general inconsistency between the QVT and TGG concepts. It was, for example explained which extensions to TGGs and the TGG interpreter would allow to provide an equal expressive power. Also, in the rule processing of QVT and TGGs, the starting point of a transformation is different. It was mentioned in section 2.2.2 (p. 20) that there is a conceptual difference between a start rule and a start context. However, the alignment of these concepts is just a technical issue. To overcome this conceptual difference, the TGG interpreter has to be extended so that also rules without any context can be matched with the model patterns.
7 Conclusion

Triple Graph Grammars are a convenient formalism to specify model transformations and are well suited for the model driven integration of software systems. Existing TGG transformation tools, however, do not provide enough flexibility or sufficient expressive power for the use in practice. Therefore, it was the primary objective of this thesis to improve the existing TGG interpreter from the ComponentTools project. For this purpose, the QVT (Query/Views/Transformations) specification was compared with TGGs. QVT is an upcoming standard by the OMG which specifies model transformations in the context of MDA. The first review of the QVT specification showed many similarities to TGGs. It was therefore the secondary objective of this thesis to investigate to which extent the QVT specification could be implemented by the improved TGG-based model transformation tool.

7.1 Summary

After the first comparison of QVT and TGGs in chapter 2, both formalisms were presented in more detail. In chapter 3, the syntax and semantics of the declarative languages QVT-Relational and QVT-Core were explained. Then TGGs were introduced in chapter 4 and it was described how they are applied for model transformations. Many concepts of QVT and TGGs were shown to be quite similar and in chapter 4, some of the principles of QVT were already aligned to those of TGGs. For example, it was indicated how the check&enforce semantics of QVT can be adopted or how n-to-m model transformations can be expressed by TGGs.

Chapter 5 then presented the improved TGG interpreter that was implemented in the scope of this thesis. Firstly, a TGG model was described which integrated a number of QVT principles. In fact, this TGG model is based on the QVT-Base package which is specified by QVT. In the design of this TGG model, it was the objective to provide as much flexibility as possible. So, the
model may be extended by further expressions or operations. For example, an approach was presented to integrate OCL in the future.

Secondly, it was shown in section 5.2 how a rich graphical editor was implemented for this TGG model. For this purpose, the latest technologies from the Eclipse Graphical Modeling Framework (GMF) were used. This editor allows to conveniently specify TGG rules and aids to design only such graph patterns which are syntactically correct according to the referenced domain models.

The last part of chapter 5 then introduced the design of the TGG interpreter which performs the transformation of ECore models. Here, its modular design was presented which provides a flexible and extendable software architecture. So, it is possible to extend this interpreter in the future, for example by plugging in an OCL processor or custom application-specific operations. Through the use of the EMF facilities, it is furthermore possible to store and load the interpreter's configuration settings or to persist the correspondence models which are created during the transformation. In fact, this interpreter is already efficiently used in a project by C. Lohmann at the University of Tampere to translate UML activity diagrams into process descriptions, for example BPEL [Loh06].

After describing the implemented software components, chapter 6 presented a structural mapping from QVT to TGGs. Primarily, it was shown that the major language constructs of QVT-Core can be transformed into TGGs. For this purpose, the QVT-Core model was implemented. A TGG transformation was then shown which transforms QVT-Core mappings into TGG rules. In the next step, these TGG rules could be executed to perform the desired transformation. This shows that significant parts of QVT can be implemented by TGGs. Once OCL will be integrated into TGGs, an equal expressive power will allow a yet more thorough implementation of QVT. Through this mapping of QVT-Core to TGGs, also QVT-Relational was implemented transitively, because a mapping from QVT-Relational to QVT-Core is provided by the QVT specification. Additionally, this thesis presented an approach to transform QVT-Relational directly into TGGs. For this purpose, a QVT-Relational model was used which will presumably be contributed to the Eclipse Model Driven Development integration project (MDDi [MDDI]) in the near future.

7.2 Outlook

Now, there are further interesting subjects of future research. Firstly, to obtain a more complete implementation of QVT, an integration of OCL, as mentioned above, would be a valuable extension to TGGs. Secondly, there are a number of interesting verification techniques which could be applied to TGGs to check the correctness of the transformation rules or to verify the correctness of the transformation result. Some interesting approaches were mentioned in section 4.2.8. But, TGGs cannot be verified when any arbitrary extension may be added to the formalism. So, a study and clear classification of TGGs language constructs would be useful.

Similar to the two abstraction levels of QVT-Relational and QVT-Core, there could also be a yet more user friendly version of TGGs. It was mentioned in the introduction that a “TGG++” could
be developed as a simplified version of TGGs. The transformation of QVT-Relational to TGGs, for example, showed that the correspondence nodes can be derived from QVT relations. This, however, leads to always just one correspondence node per TGG rule. So, a more intelligent synthesis of correspondence nodes would be necessary, possibly through concepts like “TGG by example” [Geb06]. A construct like the where-clauses in QVT-Relational could be a further improvement of TGGs. Especially in such cases where equal model patterns occur in different contexts, the specification of multiple similar TGG rules could be abbreviated. Alternatively, TGG rules could “inherit” from other TGG rules. Then, the extending rules could add certain pattern elements or could override those of the super-rule.

Although EMF is a popular and wide spread meta modeling framework, it should also be possible to transform other non-EMF models. For this purpose, the concept of model adapters was presented. It would significantly extend the application area of the TGG interpreter if also the Fujaba meta model or JMI can be made accessible.

Concluding, QVT provides a number of useful and interesting concepts. Through this reconciliation of TGGs and QVT, some of these concepts could be successfully adopted by TGGs. But, it was shown that TGGs yet offer a number of advantages over the approach presented by the QVT specification. TGGs allow a more flexible design of the transformation rules and there is a number of verification techniques that could be applied to this graph grammar based technology. Especially the latter issue will become increasingly important in model driven software development.
8 Appendix

The TGG interpreter software and the TGG editor which was implemented during this thesis is provided on the contained CD. In the following, a short description of the software and the installation steps is given. In addition, a number of examples are supplied that show different application scenarios and features of the supplied tools.

8.1 Provided Software

The provided material on the CD allows to set up an eclipse developing environment. So, the software and examples can be tested and, if needed, the code and models can be inspected. It is assumed that a Java SDK (1.4.2) is installed.

Firstly, the 3.2 Release of Eclipse and GMF 1.0 need to be installed. The ZIP files for an installation on a Windows XP system are included in the CD's "/Archives" folder. Alternatively, GMF 1.0 can be installed via the Callisto update site as explained in the GMF tutorial (http://wiki.eclipse.org/index.php/GMF_Tutorial#Setup).

Now, the CD contains two eclipse workspace directories that have to be copied into a local folder:

- eclipse-SDK-3.2-workspace
- eclipse-SDK-3.2-runtime-workspace

Then, the installed eclipse can be started. During startup, the location of the above eclipse-SDK-3.2-workspace has to be specified as the workspace to be used. A number of plug-in projects should be visible in the navigator on the left side. Then, the runtime workbench has to be configured. In the main menu, select Run->Run... and specify the correct location of the runtime
workspace in the launch configuration. This is shown in the following screenshot in figure 104. In the configuration shown here, there is a launch configuration called "New_configuration".

**Figure 104: configuring the runtime workbench**

If it should not exist, a new launch configuration has to be created. Then, the location of the copied runtime workspace has to be correctly specified. The runtime workbench can be launched next by pressing the "Run"-button. The launched workspace should look like shown in figure 105.

**Figure 105: Screenshot of the runtime workbench window**
Here, there are a number of plug-in projects that belong to the examples which will be explained shortly. In the navigator menu, there is a working set specified for each example. This will filter out only the relevant projects for the particular example.

There is also an online user guide available. Figure 106 shows how it can be opened from the main menu of the runtime workbench by selecting "Help"->"Help Contents".

8.2 Provided Examples

8.2.1 The simple ctools2pnet example

The ctools2pnet example is a very simple transformation example which was used throughout this thesis. Here, components of a component system are transformed into a corresponding Petri net. The online user guide provides a thorough documentation of this example to get started.

8.2.2 The QvtCore2TGG example

This example provides the models an settings to perform a QVT-Core to TGG transformation as explained in section 6.1 (pp. 93). Here, a simple QVT-Core transformation specifies a ctools-to-pnet transformation as in the above example. This QVT-Core transformation can be transformed...
into a Triple Graph Grammar. This TGG can then again be used to perform the ctools-to-pnet transformation.

### 8.2.3 The ComponentTools2PNK example

This is an example where the TGG interpreter is integrated into another application. Here, a transformation is performed from a ComponentTools component system into a Petri net. This example can be considered the real-life counterpart of the simple ctools-to-pnet example above. Furthermore, this application prototypicaly shows the model adapter concept presented in section 5.3.7 (pp. 89). Because here, a non-ECore Petri net model from the Petri Net Kernel of the Humbold University of Berlin is used [KD96][KO98]. Figure shows a screenshot:

![Starting a transformation within ComponentTools](image)

**Figure 107: Transforming a component system into a Petri net within ComponentTools**

### 8.2.4 The QvtRelational2TGG example

This example shows an approach of transforming QVT-Relational directly into TGGs as described in section 6.2 (pp. 106). Here, the QVT model of the QVT Frontend Tool by ModelWare is used [ModelWare][MWQVT06]. Presumably, this model will be contributed to the Eclipse Model Driven Development integration project [MDDI]. See the online help for details.
Bibliography


[Lei06] Johannes Leitner: *Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken (Verification of Model Transformations Based on Triple Graph Grammars).* March, 2006. Diploma thesis supervised by Prof. Dr. Sabine Glesner, University of Karlsruhe/TU Berlin

Diploma/Master thesis supervised by Prof. T. Systä, Tampere University of Technology; Prof. U. Kelter, Siegen University


Eidesstattliche Erklärung


________________________________

Paderborn, den 15. Juli 2006