Refinement Transformation Support for QVT Relational Transformations

Thomas Goldschmidt

FZI Research Center for Information Technology

Haid-und-Neu-Str. 10-14

76131 Karlsruhe

Germany

goldschmidt@fzi.de

Guido Wachsmuth

Humboldt-Universität zu Berlin

Unter den Linden 6

10099 Berlin

Germany

guwac@gk-metrik.de

Abstract

Model transformations are a central concept in Model-driven Engineering. Model transformations are defined in model transformation languages. This paper addresses QVT Relations, a high-level declarative model transformation language standardised by the Object Management Group. QVT Relations lacks support for default copy rules. Thus, transformation developers need to define copy rules explicitly. Particular for refinement transformations which copy large parts of a model, this is a tremendous task. In this paper, we propose generic patterns for copy rules in QVT Relations. Based on these patterns, we provide a higher-order transformation to generate copy rules for a given metamodel. Finally, we explore several ways to derive a refinement transformation from a generated copy transformation.

1. Introduction

Model transformations. In Model-driven Engineering, model transformations are a central concept. They are used to translate source models to target models, e.g. platform-independent models into platform-specific ones. Furthermore, model transformations can be instrumented to translate models into text, e.g. in an executable language like Java [1]. With Query/View/Transformation (QVT) [2], the Object Management Group provides a standard for model-to-model transformations. Actually, QVT defines three model transformation languages: *QVT Relations* and *QVT Core* are declarative languages at two different levels of abstraction. The *QVT Operational Mappings* language is an imperative language. In this paper, we focus on QVT Relations, the high-level declarative language.

One can distinguish two kinds of model-to-model transformations: *Exogenous transformations* translate models expressed in a certain source language into models expressed in a different target language. For *endogenous transformations*, source and target models are expressed in the same language. The target model can be either derived by changing the input model *in-place* or by creating an entire new model.

Model refinements. Often, the target model of a transformation is simply a *refinement* of the source model, that is, the transformation preserves large parts of the source. Refinements might be either endogenous, e.g. an optimisation, or exogenous, e.g. migration to a new language version. Though in-place transformations are particular useful to describe refinements in a compact way, there are several reasons to prefer the creation of a new model: First, the source model is preserved. Second, traces between the source and target model become explicit. Finally, with QVT Relations, exogenous refinements need to be described explicitly since in-place transformations are restricted to endogenous transformations.

A transformation realising a refinement needs to copy large parts of a model. Since QVT Relations does not support default copies, a refinement definition needs to specify copies explicitly. In this paper, we investigate copies in QVT Relations. First, we propose generic patterns for copy rules. Second, we provide a way to generate the definition of a copy transformation for a given metamodel. The generation is specified as a higher-order transformation. Finally, we explore several ways to derive a refinement from a generated copy transformation.

Structure of the paper. In Section 2, we discuss related work. In Section 3, we examine generic patterns in copy rules. In Section 4, we present a higher-order transformation for the generation of copy transformations. In Section 5, we discuss the derivation of refinement transformations out of generated copy transformations. The paper is concluded in Section 6.

2. Related Work

In contrast to QVT Relations, QVT Operational Mappings provide a *deep copy* operation that can be used within imperative mapping rules. This operation creates a copy of a given substructure. However, it is not possible to specify exceptions for elements that occur within the substructure that is copied. Therefore, this approach is only partially useful for a refinement transformation.

The Atlas Transformation Language (ATL) [3] supports a special mode that allows the transformation programmer to specify that a transformation should be run as a refinement transformation. This means that all elements are copied by default whilst those elements that are matched by transformation rules within the actual transformation are not. Those elements are instead treated as specified by the given transformation rules.

Another model transformation approach is the Triple Graph Grammar (TGG) [5] approach. Within this approach the standard execution mechanism is described as being an in-place transformation where transformation rules are applied as long as there are rules that still match patterns within the model. So, TGGs can naturally be used for endogenous, in-place transformations and do not need special support for copy rules.

3. Generic Patterns for Copy Rules

In this section, we will discuss generic patterns for copy rules. We illustrate these patterns by copy rules for Petri net models.

Example: Petri net models

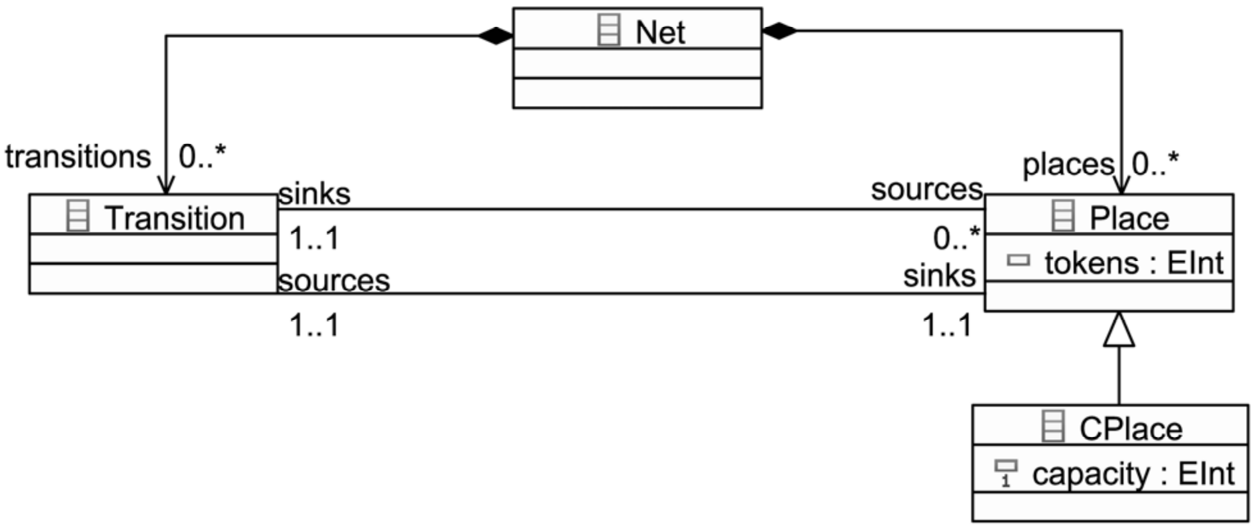

Figure 3.1: Example metamodel for Petri nets

Figure 3.1 shows a metamodel for Petri net models: A *Net* consists of *places* and *transitions*. A *Transition* has several places as *sources* and as *sinks*. In the same way, a *Place* has several transitions as *sources* and as *sinks*. Furthermore, a place is marked by an optional number of *tokens*. A *CPlace* is additionally constrained by a maximal capacity.

Copying Petri nets reveals several issues:

(i) Nets, places, and transitions need to be copied.

(ii) For places, we need to preserve their kind (*Place* vs. *CPlace*).

(iii) For places, we need to preserve the number of marking tokens. The same holds for capacities.

(iv) Links between nets, places, and transitions need to be copied.

}

Copying instances

The first two issues concern copies of metaclass instances. For a non-abstract metaclass, we define a *copy relation* to specify copies of its instances. In the source model, this relation simply matches an instance of the metaclass. In the target model, the relation enforces a corresponding instance of the same metaclass. Declaring the relation as top ensures that the relation is applied to every instance of the metamodel in the source model. Listing 3.1 shows such a relation *copyNet* for the metaclass *Net*.

```
top relation copyNet {

      checkonly domain orig netO: Net {};
      enforce domain copy netC: Net {};
      where { copiedNet(netO, netC); }
}

relation copiedNet {
      checkonly domain orig netO: Net {};
      checkonly domain copy netC: Net {};
}
```

Listing 3.1 Copy and marker relations for metaclass *Net*.

In the *where* clause of this relation, another relation *copiedNet* is called with the original net and its copy as arguments. This *marker relation* is shown as well in Listing 3.1. Marker relations are non-top relations that simply match their arguments in the source and target model. We use such relations to indicate which elements have already been copied. Once an element is copied, we call the corresponding marker relation from the *where* clause.

```
top relation copyPlace {
      checkonly domain orig placeO: Place {};
      enforce domain copy placeC: Place {};
      when { not copiedCPlace(placeO, placeC); }
      where { copiedPlace(placeO, placeC); }
}
top relation copyCPlace {
      checkonly domain orig placeO: CPlace {};
      enforce domain copy placeC: CPlace {};
      where { copiedCPlace(placeO, placeC); }
}
relation copiedPlace {
      checkonly domain orig placeO: Place {};
      checkonly domain copy placeC: Place {};
}
relation copiedCPlace {
      checkonly domain orig placeO: Place {};
      checkonly domain copy placeC: Place {};
      where { copiedPlace(placeO, placeC); }
}
```

Listing 3.2 Copy and marker relations for metaclasses *Place* and *CPlace*.

Listing 3.2 shows the copy and marker relations for *Place* and *CPlace*. It illustrates how marker relations for subclasses call the marker relations for their superclasses, that is, *copiedCPlace* calls *copiedPlace* from its *where* clause. This way, we can indicate that an instance was already copied by a more specific rule. The *when* clause of *copyPlace* ensures this. The relation is only executed, if *copiedCPlace* is never called for a given pair of places in the source and target model. To make this pattern work for any given class hierarchy, we need marker relations for abstract meta-classes as well.

Copying slots and links

The last two issues concern copies of slots and links. Both can be achieved in a similar way. To copy slots instantiating an attribute defined by a metaclass, we define a top relation for this attribute. In the source model, the relation matches an instance of the metaclass as well as the value stored in the slot corresponding to the attribute. In the target model, another instance of the metaclass is matched. The original value is copied to the corresponding slot in the target instance. The when clause of the relation needs to ensure that the target instance is a copy of the source instance by calling a marker relation. Listing 3.3 shows the copy relation for *Place.tokens*. Since the marker relation *copiedCPlace* calls *copiedPlace*, this relation will copy tokens slots for instances of *CPlace* as well.

```
top relation copyPlace_token {
      tokenO: Integer;
      checkonly domain orig placeO: Place { tokens = tokenO };
      enforce domain copy placeC: Place { tokens = tokenO };
      when { copiedPlace(placeO, placeC); }
}
```

Listing 3.3 Copy relation for attribute *Place.tokens*.

To copy links instantiating a reference between metaclasses, the pattern is quite similar. We define a top relation for the reference matching instances in the source and target model. In both models, the relation matches two instances connected by a link. The *when* clause ensures that instances in the target model are copies of the instances in the source model. Listing 3.4 shows the copy relations for *Place.sources* and *Place.sinks*.

```
top relation copyPlace_sources {
      checkonly domain orig placeO: Place {
            sources = transO: Transition {}
      };
      enforce domain copy placeC: Place {
            sources = transC: Transition {}
      };
      when {
            copiedPlace(placeO, placeC);
            copiedTransition(transO, transC);
      }
}
```

```
top relation copyPlace_sinks {
      checkonly domain orig placeO: Place {
            sinks = transO: Transition {}
      };
      enforce domain copy placeC: Place {
            sinks = transC: Transition {}
      };
      when {
            copiedPlace(placeO, placeC);
            copiedTransition(transO, transC);
      }
}
```

Listing 3.4 Copy relations for references *Place.source* and *Place.sink*.

For a pair of bidirectional references, only one copy relation for one of the references is needed.

4. Generation of Copy Transformations

The patterns presented in Section 3 can be used to specify a copy transformation in QVT Relations for arbitrary Ecore metamodels. Obviously, these patters are that generic that we can generate a copy transformation directly and automatically from a given metamodel. In this section, we investigate a *higher-order transformation* for this purpose. This higher-order transformation is written in QVT Relations itself and captures the patterns discussed in the preceding section. It is based on the Ecore meta-metamodel, on the OCL standard library (for negation), and on the QVT Relations metamodel. After executing this higher-order transformation, a complete model of the copy transformation is available. This can then either directly be used in its abstract syntax or a simple pretty printer can be used to print it in its textual concrete syntax.

4.1. Overall Structure

The overall structure of the higher-order transformation is shown in Listing 4.1. Basically, the transformation works analogously to the patterns of Section 3. For each package in the metamodel, a copy transformation is generated (c.f. `Package2Transformation`). The name of this transformation is derived from the package name. The copy transformation declares two domains *source* and *target*, both typed by the metamodel package.

```
transformation Ecore2copyQVT (
      mm: ecore, oclstdlib:ecore, qvt: QVTRelation)  {
      top relation Package2Transformation {

            n:String;
            checkonly domain mm ePackage: ecore::EPackage { name = n   };
            enforce domain qvt t: QVTRelation::RelationalTransformation {
              name = 'Copy' + n,
              modelParameter = sourceMM: QVTBase::TypedModel {
                        name = 'source',
                        usedPackage = uPackage: ecore::EPackage{}  },
              modelParameter = targetMM: QVTBase::TypedModel {
                        name = 'target',
                        usedPackage = uPackage: ecore::EPackage{}  }
            };
            when {  ePackage.eContainer().oclIsUndefined();}
            where { uPackage = ePackage;
                    MarkTypedModel(sourceMM, targetMM);
                    MarkTransformation(t); }
      }
      relation MarkTypedModel { ... }
      relation MarkTransformation { ... }

      top relation Class2CopyRelation { ... }
      top relation SubClass2MarkerCallInWhen { ... }
      top relation Class2MarkerRelation { ... }
      top relation Attribute2Relation { ... }
      top relation Reference2Relation{ ... }
      top relation MarkBooleanType { ... }

      relation Class2Domain { ... }
      relation Attribute2Template { ... }
      relation Reference2Template { ... }
      relation Class2MarkerCall { ... }
      relation Class2MarkerCallInPattern { ... }
}
```

Listing 4.1 Overall structure of the higher-order transformation *emf2copyQVT*.

The remaining relations of the higher-order transformation generate the relations of the copy transformation:
(i) For each non-abstract metaclass, a copy relation is generated (c.f. `Class2CopyRelation`).
(ii) For each subclass, a negated call to the corresponding marker relation is added to the *when* clause of a copy relation (c.f. `SubClass2MarkerCallInWhen`).
(iii) For each metaclass, a marker relation is generated (c.f. `Class2MarkerRelation`).
(iv) For each attribute, a copy relation is generated (c.f. `Attrib-ute2CopyRelation`).
(v) For each relation, a copy relation is generated (c.f. `Refer-ence2CopyRelation`).

We will discuss the details of the generation in the remainder of this section.

4.2. Generating copy relations from metaclasses

Listing 4.2 and Listing 4.3 show relations for the generation of copy relations from metaclasses. For each non-abstract metaclass within a metamodel package, a copy relation is created (c.f. `Class2CopyRelation`). The pattern for creating the relation looks very complicated due to the heavily nested abstract syntax. During the creation, a *where* clause is generated that contains a `RelationCallExp` to call the corresponding marker relation. The actual contents of this call are generated in the `Class2MarkerCall` relation.

```
top relation Class2CopyRelation {
      sourceMM, targetMM : QVTBase::TypedModel;
      checkonly domain mm eClass: ecore::EClass {
            ePackage = ePackage: ecore::EPackage {},
            name = n : String{},
            abstract = false
      };
      enforce domain qvt rel: Relation {
            name = 'Copy' + n,
            isTopLevel = true,
            variable = sourceVar: ocl::ecore::Variable {},
            variable = targetVar: ocl::ecore::Variable {},
            _domain = sourceDom: QVTRelation::RelationDomain {
                  isCheckable = true },
            _domain = targetDom: QVTRelation::RelationDomain {
                  isEnforceable = true },
            _transformation = transfo : RelationalTransformation {},
            _where = wherePattern: QVTBase::Pattern {
                  predicate = pred: QVTBase::Predicate {
                        conditionExpression =  markerCall:RelationCallExp{}
                  }
            }
      };
      when {
            Package2Transformation(rootPackage(ePackage), transfo) or
                  MarkTransformation(transfo);
            MarkTypedModel(sourceMM, targetMM);
      }
      where {
            Class2Domain(sourceMM,eClass,'source',sourceVar,sourceDom);
            Class2Domain(targetMM,eClass,'target',targetVar,targetDom);
            Class2MarkerCall(eClass,sourceVar,targetVar,markerCall);
      }
}
```

Listing 4.2 Generating a copy relation from a metaclass.

The domains that are used as source and target within e.g., `Cass2CopyRelation` c(see Listing 4.3) call `Class2Domain` relation in their where clauses to construct the `DomainPattern` that is used to match the source and construct the target domain of the copy relations. ObjectTemplateExpressions are used to match the corresponding class of the model elements that are going to be copied.

```
relation Class2Domain {
      checkonly domain qvt mm: QVTBase::TypedModel {};
      checkonly domain mm eClass: ecore::EClass { name = className: String {} };
      primitive domain prefix: String;
      enforce domain qvt var: ocl::ecore::Variable {
            name = prefix + className, eType = eClass };
      enforce domain qvt dom: QVTRelation::RelationDomain {
            typedModel = mm,
            name = prefix + className,
            rootVariable = var,
            pattern = p: QVTRelation::DomainPattern {
                  templateExpression = expr: ObjectTemplateExp {
                        referredClass = eClass,
                        eType = eClass,
                        bindsTo = var: ocl::ecore::Variable {}
                  },
                  bindsTo =  var: ocl::ecore::Variable {}
            }
       };
}
```

Listing 4.3 Generating domains for relations.

The *when* clauses of copy relations are generated by a separate relation (c.f. `SubClass2MarkerCallInWhen`). For each direct subclass, a negated `RelationCallExp` calling the marker relation of this subclass is generated.

```
top relation SubClass2MarkerCallInWhen {
      checkonly domain mm subClass: ecore::EClass {
            eSuperTypes = superClass: ecore::EClass {}
      };
      checkonly domain oclstdlib notOp: EOperation { ... };
      enforce domain qvt rel: Relation {
            _when = whenPattern: QVTBase::Pattern {
                  predicate = pred: QVTBase::Predicate {
                        conditionExpression = notCall: OperationCallExp {
                              source = markerCall: RelationCallExp {},
                              referredOperation = notOp,
                              eType = booleanType
                        }
                  }
            }
      };
      when { Class2CopyRelation(superClass, rel); }
      where {
Class2MarkerCall(subClass,rel.variable->at(1),rel.variable->at(2),markerCall);
      }
}
```

Listing 4.4 Generating negated marker calls from subclasses.

4.3. Generating marker relations from metaclasses

Listing 4.4 shows relations for the generation of marker relations. Marker relations are generated for each metaclass including abstract ones (c.f. `Class2MarkerRelation`). For each direct superclass, a call to the marker relation of this superclass is added to the *where* clause. Again, this is achieved by a separate relation (c.f. `Class2MarkerCallInPattern`) similar to the one for negated calls in *when* clauses.

```
top relation Class2MarkerRelation {
      sourceMM, targetMM: QVTBase::TypedModel;
      checkonly domain mm eClass: ecore::EClass {
            ePackage = ePackage: ecore::EPackage {},
            name = n : String{}
      };
      enforce domain qvt rel: Relation {
            name = 'Mark' + n,
            isTopLevel = false,
            variable = sourceVar: ocl::ecore::Variable {},
            variable = targetVar: ocl::ecore::Variable {},
            _domain = sourceDom: QVTRelation::RelationDomain {
                  isCheckable = true
            },
            _domain = targetDom: QVTRelation::RelationDomain {
                  isCheckable = true
            },
            _transformation = transfo: QVTRelation::RelationalTransformation {},
            _where = wherePattern: QVTBase::Pattern {}
      };
      when {Package2Transformation(rootPackage(ePackage), transfo) or
                  MarkTransformation(transfo);
            MarkTypedModel(sourceMM, targetMM);
      }
      where {
            Class2Domain(sourceMM, eClass, 'source', sourceVar, sourceDom);
            Class2Domain(targetMM, eClass, 'target', targetVar, targetDom);
            eClass.eSuperTypes -> forAll(st | Class2MarkerCallInPattern(st,
      sourceVar, targetVar, wherePattern));
      }
}
```

Listing 4.5 Generating marker relations from metaclasses.

4.4. Generating copy relations from attributes and references

Finally, Listing 4.5 shows a relation for the generation of copy relations from attributes. This relation generates a copy relation for each attribute in the metamodel. An analogue relation (c.f. `Reference2CopyRelation`) is used to create copy relations for references. Again, we omit patterns already mentioned in other listings.

```
top relation Attribute2Relation {
      sourceMM, targetMM: QVTBase::TypedModel;
      checkonly domain mm attribute: ecore::EAttribute {
            name = attrName: String {},
            eType = attrType: ecore::EDataType {},
            eContainingClass = eClass: ecore::EClass {
                  ePackage = ePackage: ecore::EPackage {},
                  name = className: String {}
            }
      };
      checkonly domain oclstdlib notOp: EOperation { ... };
      enforce domain qvt rel: Relation {
            name = 'CopyAttribute_' + className + '_' + attrName,
            isTopLevel = true,
            variable = attrVar: ocl::ecore::Variable {
                  name = 'local_' + attrName + 'Value', eType = attrType
            },
            variable = sourceVar: ocl::ecore::Variable {},
            variable = targetVar: ocl::ecore::Variable {},
            variable = attrSourceVar: ocl::ecore::Variable {},
            variable = attrTargetVar: ocl::ecore::Variable {},
            _domain = sourceDom: QVTRelation::RelationDomain {
                  isCheckable = true },
            _domain = targetDom: QVTRelation::RelationDomain {
                  isEnforceable = true },
            _transformation = transfo: QVTRelation::RelationalTransformation {},
            _when = whenPattern: QVTBase::Pattern {
                  predicate = pred: QVTBase::Predicate {
                        conditionExpression =  markerCall: RelationCallExp {}
                  }
            }
      };
      when {Package2Transformation(rootPackage(ePackage), transfo) or
                  MarkTransformation(transfo);
            MarkTypedModel(sourceMM, targetMM);
      }
      where {Class2Domain(sourceMM, eClass, 'source', sourceVar, sourceDom);
            Attribute2Template(attribute, 'source', attrVar, attrSourceVar,
sourceDom.pattern.templateExpression);
            Class2Domain(targetMM, eClass, 'target', targetVar, targetDom);
            Attribute2Template(attribute, 'target', attrVar, attrTargetVar, tar-
getDom.pattern.templateExpression);
            Class2MarkerCall(eClass, sourceVar, targetVar, markerCall);
      }
}
```

Listing 4.6 Generating copy relations from attributes and references.

To match the values of the attributes that are copied `Attribute2Relation` calls the `Attribute2Template` relation. For this an a variable is generated that is then checked against the source and target attributes within a `PropertyTemplateItem` of the copy relation. See Listing 4.7 for details on this part of the copy relation.

```
relation Attribute2Template {
      attributeOwningClass : ecore::EClass;
      checkonly domain mm attribute: ecore::EAttribute {
            name = attrName: String {},
            eType = attrType: ecore::EDataType {}
      };
      primitive domain prefix: String;
      checkonly domain qvt attrValVar: ocl::ecore::Variable {};
      enforce domain qvt attrVar: ocl::ecore::Variable {
            name = prefix + '_' + attrName, eType = attrType };
      enforce domain qvt expr: QVTTemplate::ObjectTemplateExp {
            part = attrTemplate: QVTTemplate::PropertyTemplateItem {
                  referredProperty = attribute,
                  value = featureExp: ocl::ecore::VariableExp {
                        referredVariable = attrValVar,
                        eType = attrType
                  }
            },
            eType = attributeOwningClass
      };
      where {attributeOwningClass = attribute.eContainingClass;}
}
```

Listing 4.7 Generating attribute templates.

5. Writing Refinement Transformations

There are two possibilities to specify a refinement transformation based on a generated copy transformation. First, by writing a new transformation manually that overwrites specific rules of the generic copy transformation. The rules of the refinement transformation are then called instead of the overwritten ones within the copy transformation. This enables us to handle those parts of the model that should be treated other then a mere copy. Second, it could be possible to define a transformation that only contains the exception rules and then use a higher-order transformation to weave these rules into a generated copy transformation.

Manually Extended Transformation

According to the QVT Relations specification [2], it is possible to define *extension* transformations. Within an extension, it is possible to define rules that conditionally *override* rules from the extended transformation. However, the exact semantics of the override mechanism is not defined within the standard. For example, it is not mentioned if this conditional override means that for elements where the overriding pattern does not match the overridden rule is used instead or if the element is not matched at all. Furthermore, QVT Relations engines like mediniQVT [4] currently also do not support this feature. Nevertheless, we propose extension as a natural way to specify refinement transformations with help of a generated copy transformation.

The transformation developer has to make sure that only *complete* rules should be provided by the extension. *Complete* in this case means that for each manually specified refinement rule a complementary rule needs to be specified that matches all elements that are *not* matched by the refinement rule. Only then it can be ensured that all model elements not handled by the refinement rules are normally copied. An easy way to ensure this is to call the refinement transformation as well as the corresponding generic copy rule using an exclusive or within the *where* statement of a third rule that has the same source pattern as the copy rule.

Another problem with extension is that the user has to make sure that the corresponding marker relations are called within refinement rules. Otherwise all dependent generated copy rules will not execute. An example for this manual extension can be seen in listing 5.1. Note that the call of the original copy relation (`copyPN::copyNet`) symbolizes that the original transformation rule should be called here. As the standard does not make a statement on how to do this it might also be necessary to manually copy this rule as a non-toplevel rule to the refining transformation to be able to call it within the where clause of `refinedCopyNetCall`.

```
transformation refinePN extends copyPN (orig: petri, copy: petri) {
      top relation refinedCopyNetCall overrides copyPN::copyNet{
            checkonly domain orig netO: Net {};
            enforce domain copy netC: Net {};
            where { if (refinedCopyNet(netO, netC)) then true
                     else copyPN::copyNet(netO, netC) endif; }
      }
      relation refinedCopyNet {
            toks, newToks : Integer;
            checkonly domain orig netO: Net {
                  p = refP : Place{token = toks}
            };
            enforce domain copy netC: Net {
                  p = refP : Place{token = newToks}
            };
            when {toks = 0;}
            where { newToks = toks+1; copyPN::copiedNet(netO, netC); }
      }
}
```

Listing 5.1 Using manual extension to refine the copy transformation

Automatic Weaving

The second possibility to create the desired refinement transformation is based on a transformation weaving technique. As depicted in figure 5.1 the rules that resemble the intended refinements are specified as standard QVT Relations transformation (c.f. *$T(M_2M_2)$*). As a next step, this transformation is tagged as being a refinement transformation. This tagging can, for instance, be done using either comments on the textual concrete syntax of the transformation or as a non-intrusive decorator model on the model (abstract syntax) of the transformation.

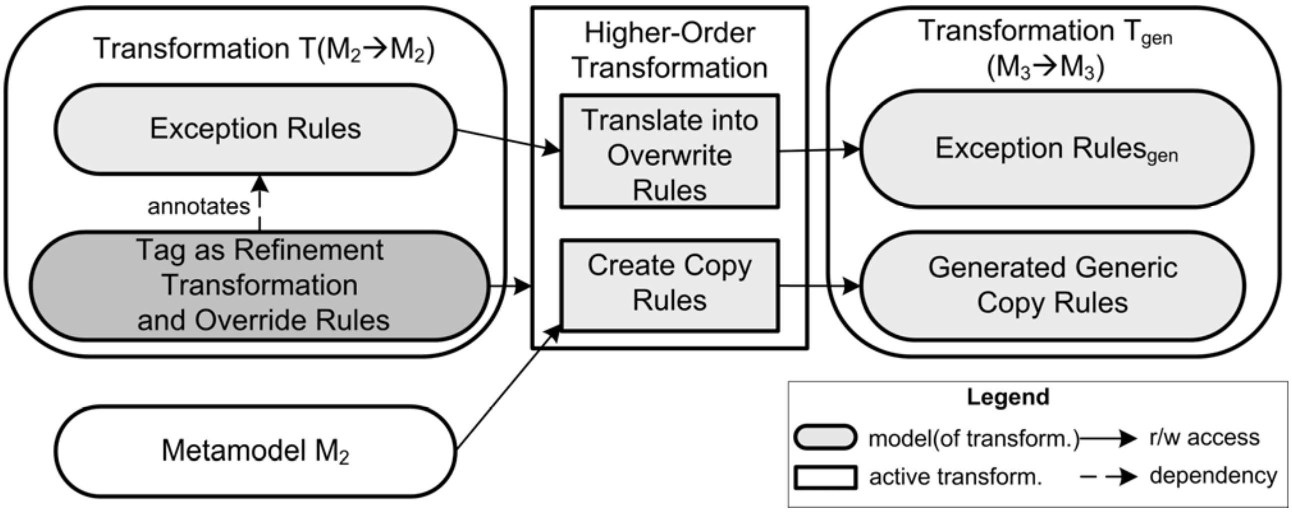
Transformation T(M₂→M₂)
Exception Rules
annotates
Tag as Refinement Transformation and Override Rules
Metamodel M₂
Higher-Order Transformation
Translate into Overwrite Rules
Create Copy Rules
Transformation T_gen (M₃→M₃)
Exception Rules_gen
Generated Generic Copy Rules
Legend
model(of transform.)
r/w access
active transform.
dependency

Figure 5.1 Automatic Weaving of Exception Rules

The tagged transformation is then transformed by a higher-order transformation (c.f. *HOT*). This transformation translates the refinement rules in a way that they conform to the expected calls of the marker relations (see Section 4). Furthermore, it generates copy rules (as defined in section 4) for those patterns that are not handled by the refinement. This step is based on the metamodel M_2 that is used as input and output parameter of the refinement transformation. As the refinement rules may only match a part of the elements that were normally matched by their corresponding copy rules, a similar construct has to be introduced as it is shown in section 5.1.

Using the tagging and weaving approach the transformation developer does not need to care about ensuring the consistency of the refinement rules with the generated copy rules. However, as the developer does not necessarily see the generated transformation which is executed, debugging is more complicated in this case. Additional support for tracing the debug information within the generated transformation back to the original transformation would be required. We prefer to use model-to-model over model-to-text transformations for the higher-order transformations as tracing is included in most of such engines. These traces can then easily be used during debugging to easily navigate from the generated, actually debugged transformation elements to their original, manually specified elements.

A similar refinement technique is described in [6] called *superimposition*. The superimposing transformation loosely overrides rules of the superimposed transformation with the same name. A higher-order transformation is then used to merge both transformations into a third one. A similar approach is used here when weaving the explicit refinement rules with generated copy rules. However, in this case there is no explicit superimposed transformation, as the copy rules (which would be superimposed by the refinement rules) are also generated during the weaving process and do not exist before.

6. Concluding Remarks

We presented an approach that allows transformation developers to circumvent the lack of default copy rules within QVT Relations. We presented generic patterns for copying instances, slots, and links. Based on these patterns, we provided a higher-order transformation for generating a copy transformation from a metamodel. The higher-order transformation is written in QVT Relations. Finally, we proposed two different possibilities on how the approach can be integrated into the development of refinement transformations. Future work will address the support for exogenous refinement transformations.

References

1. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. OOPSLA (2003)
2. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) http://www.omg.org/docs/formal/08-04-03.pdf.
3. Mens, T, Gorp, P.V.: A taxonomy of model transformation. GraMoT (2005).
4. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming, Special Issue on Second issue of experimental software and toolkits (EST) 72(1-2) (2008) 31–39
5. ikv++: medini QVT. http://www.ikv.de/ Last retrieved 2008-10-30.
6. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, volume 903 of Lecture Notes in Computer Science (LNCS), pages 151–163,Heidelberg, 1994. Springer Verlag.
7. Wagelaar, D.: Composition Techniques for Rule-Based Model Transformation Languages, 1st International Conference on Model Transformation - Theory and Practice of Model Transformations, volume 5063 of Lecture Notes in Computer Science (LNCS), pages 152-167, Heidelberg, 2008. Springer Verlag.