Using the AgileUML metamodel matching and transformation synthesis tools

K. Lano

May 25, 2021

1 Introduction

The AgileUML toolset provides techniques for deducing matchings of classes and features between metamodels. These matchings can be used to derive model transformations in UML-RSDS, ATL, ETL, QVT-O and QVT-R.

The latest version of the tools can be obtained from: https://www.agilemde.co.uk or from https://projects.eclipse.org/projects/modeling.agileuml.

2 Metamodel matching

Metamodels should be loaded using the File menu option *Recent* (this loads the file output/mm.txt) or *Load metamodel*. Classes in the metamodel(s) should be marked as *source*, ie., with this stereotype, if they are in the source metamodel of the matching, and as *target* if they are in the target metamodel. Unmarked classes are assumed to be shared (in both metamodels and mapped to themselves). It is convenient to use the *Import* facility to separate source and target metamodels into separate files, eg:

```
Import:
classmm.txt

Import:
relationalmm.txt
```

in mm.txt.

Figure 1 shows the visual representations of the metamodels of the ATL Class2Relational transformation case from the ATL zoo (www.eclipse.org/atl/atlTransformations). The source metamodel MM_1 is *Class*, on the LHS, the target metamodel MM_2 is *Relational*, on the RHS.

In KM3 text format the source metamodel is written as:

```
package Class {
```

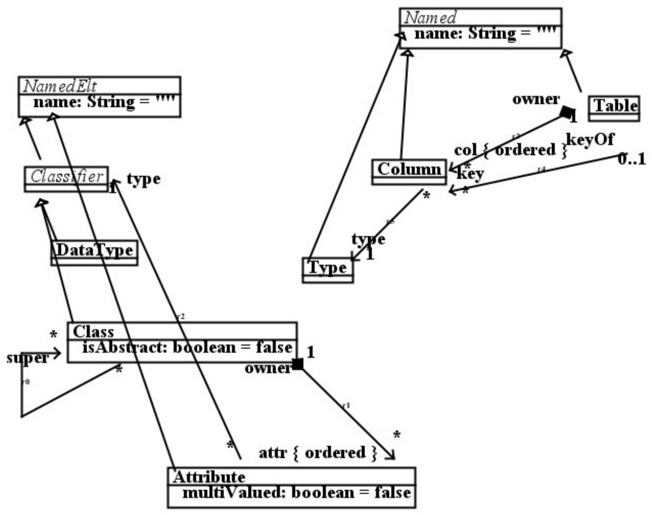


Figure 1: Class and Relational metamodels

```
abstract class NamedElt {
attribute name : String;
}

abstract class Classifier extends NamedElt {
}

class DataType extends Classifier {

}

class Class extends Classifier {
reference super[*] : Class;
reference attr[*] ordered container : Attribute oppositeOf owner;
attribute isAbstract : boolean;
}

class Attribute extends NamedElt {
attribute multiValued : boolean;
reference type : Classifier;
reference owner : Class oppositeOf attr;
}
}
```

The target metamodel is:

```
package Relational {

abstract class Named {
attribute name : String;
}

class Table extends Named {
reference col[*] ordered container : Column oppositeOf owner;
reference key[*] : Column oppositeOf keyOf;
}

class Column extends Named {
reference owner : Table oppositeOf col;
reference keyOf[0-1] : Table oppositeOf key;
reference type : Type;
}

class Type extends Named {

}
}
```

Once both metamodels are loaded, select the option *Synthesise transformation* from the *Synthesis* menu. This provides several options for matching strategies (Table 1). A matching comprises a relation *cm* between the classes of the two metamodels, and a relation *fm* between the features.

Measure	*Definition*
Data structure similarity (DSS)	Classes possess similar data in their owned, inherited or composed features [3]
Graph structural similarity (GSS)	Class neighbourhoods in the 2 metamodels have similar graph structure metrics [9]
Graph edit similarity (GES)	Class reachability graphs in the 2 metamodels have low graph edit distance [2]
Name syntactic similarity (NSS)	Classes have names with low string edit distances [8]
Name semantic similarity (NMS)	Class names are synonymous terms or in the same/linked term families according to a thesaurus [4]
Semantic context similarity (SCS)	Classes play similar semantic roles in the 2 metamodels [10].

Table 1: Syntactic and semantic similarity measures for classes

For DSS either a general matching can be used, or matchings can be restricted to be *inheritance preserving*: a subclass D of source class C is only permitted to map to a class $C1$ which C maps to, or to a subclass/descendant of such a $C1$.

For small examples such as the class/relational mapping, the NMS or DSS options are suitable. NMS uses a thesaurus (in output/thesaurus.txt) to match classes, and it is more appropriate if there are some linguistic similarities between the metamodels (such as *NamedElt* and *Named*, or *Class* and *Table*). If the metamodels have quite different terminologies then DSS is more suitable.

The tool will prompt you for the maximum navigation path to be considered on the source and target side. This means the maximum length of feature chains such as *super.isAbstract* or *key.type* (both of length 2). For cases where there is a close structural similarity between the metamodels, such as the Ant/Maven case, the choice of length 1 for source and target is usually adequate.

The results of the matching are shown in the console (Figure 2) and written to *output/forward.tl* for the forward mapping, and *output/reverse.tl* for the reverse mapping.

For example, the initial forward matchings derived by NMS with maximum source and target navigation length 1 look as follows:

$$
\begin{array}{l}
\mathit{NamedElt} \longmapsto \mathit{Named} \\
\quad \mathit{name} \longmapsto \mathit{name} \\
\mathit{Class} \longmapsto \mathit{Table} \\
\quad \mathit{name} \longmapsto \mathit{name} \\
\quad \mathit{attr} \longmapsto \mathit{col} \\
\mathit{Attribute} \longmapsto \mathit{Column} \\
\quad \mathit{name} \longmapsto \mathit{name} \\
\quad \mathit{owner} \longmapsto \mathit{owner} \\
\quad \mathit{type} \longmapsto \mathit{type} \\
\mathit{Classifier} \longmapsto \mathit{Type} \\
\quad \mathit{name} \longmapsto \mathit{name} \\
\mathit{DataType} \longmapsto \mathit{Type} \\
\quad \mathit{name} \longmapsto \mathit{name}
\end{array}
$$

However, this matching is incomplete on both target and source sides (*isAbstract*, *multiValued* and *super* are unused source features, *key* and *keyOf* are unused target features). In addition, there is a potential inconsistency in that *Class* is mapped to *Table*, but *Table* is not a specialisation of (or equal to) the image *Type* of *Classifier*, even though *Class* is a specialisation of *Classifier*.

An interactive process following the matching derivation is used to identify such flaws and to suggest possible resolutions.

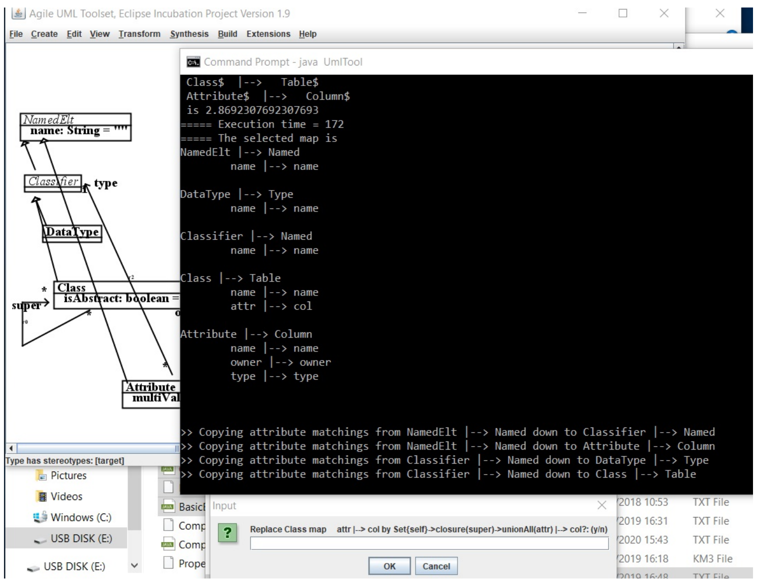
Table 2 summarises the different checks which we use.

For the case of feature mapping incompleteness in Class2Relational, because the unused source feature *super* is a self-association on *Class*, the system proposes to replace $\mathit{attr} \longmapsto \mathit{col}$ by the mapping

$$
\begin{array}{l}
\mathit{Set}\{\mathit{self}\}{\rightarrow}\mathit{closure}(\\
\qquad \mathit{super}){\rightarrow}\mathit{unionAll}(\mathit{attr}) \ \longmapsto \ \mathit{col}
\end{array}
$$

Issue	*Correction*
Class mapping $Sub \longmapsto T$ for Sub subclass of E, has T not subclass/or equal to F, where $E \longmapsto F$	Retarget Sub mapping, or add target splitting map $Sub \longmapsto F$
Two directions of bidirectional association r not mapped to mutually reverse target features	Modify one feature mapping to ensure consistency
Source, target features have different multiplicities	Propose modified mappings
Unused target subclasses $F1$ of F, where $E \longmapsto F$	Introduce condition $F1C$ and mapping $\{F1C\}E \longmapsto F1$
Unused source or target feature f	Suggest class or feature mapping that uses f
Feature mapping $f \longmapsto r.g$ with $r : R$ of abstract type/element type	Propose concrete subclass $RSub$ of R for instantiation of r.

Table 2: Consistency and completeness checks


Agile UML Toolset, Eclipse Incubation Project Version 1.9
File Create Edit View Transform Synthesis Build Extensions Help
NamedElt
name: String = ""
Classifier
type
DataType
Class
isAbstract: boolean =
super
Attribute
multiVal
Type has stereotypes: [target]
Pictures
Videos
Windows (C:)
USB DISK (E:)
USB DISK (E:)
Command Prompt - java UmlTool
Input
OK
Cancel

```
 Class$  |-->   Table$
 Attribute$  |-->   Column$
 is 2.8692307692307693
===== Execution time = 172
===== The selected map is
NamedElt |--> Named
        name |--> name

DataType |--> Type
        name |--> name

Classifier |--> Named
        name |--> name

Class |--> Table
        name |--> name
        attr |--> col

Attribute |--> Column
        name |--> name
        owner |--> owner
        type |--> type


>> Copying attribute matchings from NamedElt |--> Named down to Classifier |--> Named
>> Copying attribute matchings from NamedElt |--> Named down to Attribute |--> Column
>> Copying attribute matchings from Classifier |--> Named down to DataType |--> Type
>> Copying attribute matchings from Classifier |--> Named down to Class |--> Table
```

Replace Class map   attr |--> col by Set{self}->closure(super)->unionAll(attr) |--> col?: (y/n)

Figure 2: Initial metamodel matching

of all defined attributes of a class to the columns of a table, ie., all attributes of the class itself and of all its ancestors are mapped to columns of the table corresponding to the class (Figure 2).

Because of the inheritance conflict in the targets of the class mappings, the additional class mapping

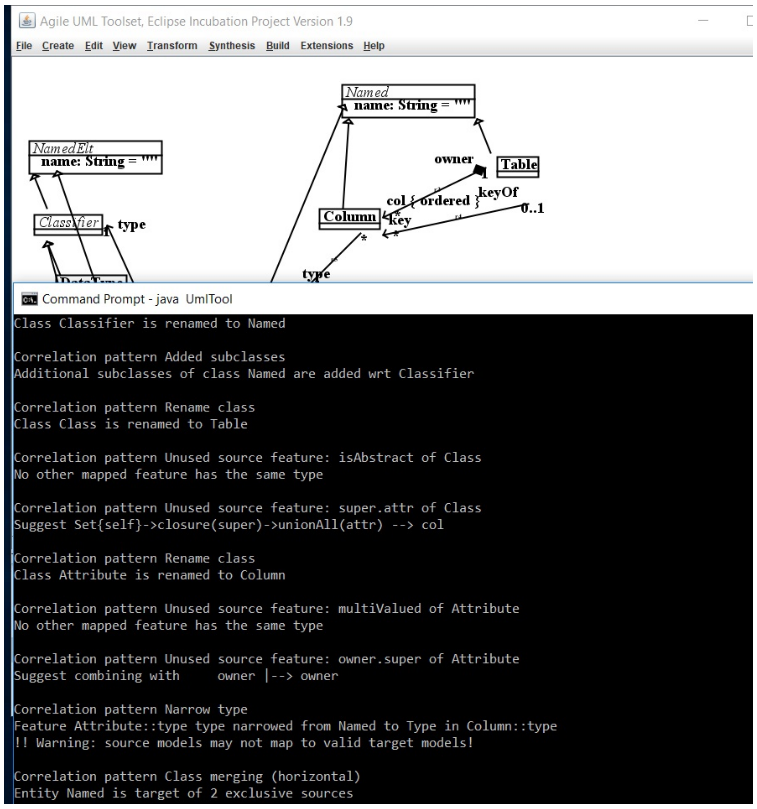
$$\begin{array}{l} \mathit{Class} \longmapsto \mathit{Type} \\ \quad \mathit{name} \longmapsto \mathit{name} \end{array}$$

is also proposed: this is a 'vertical class splitting' of *Class*: each *Class* instance in a source model is represented by both a *Type* instance and a *Table* instance in the resulting *Relational* model[1].

In the final stage of metamodel matching, the details of the matching and any correspondence patterns identified are listed in the console (Figure 3). Warnings are given in cases (such as multiplicity or type narrowing of target features relative to the source) where semantic problems may arise in mapping source models to target models.

The matchings can also be checked against specific source and target models, to identify detailed corrections in feature mappings. This is the option "Check model wrt TL" on the File menu. The models should be stored in a file *output/out.txt*. Load the metamodels and the $\mathcal{TL}$ transformation (this loads forward.tl), then run the check model option. This checks for numeric functional

[1]The target classes must have no common MM_2 ancestor which is a type/element type of some $g \in \mathrm{ran}(fm)$


Agile UML Toolset, Eclipse Incubation Project Version 1.9
File Create Edit View Transform Synthesis Build Extensions Help
Named
name: String = ""
NamedElt
name: String = ""
Classifier
type
owner
Table
col { ordered }
keyOf
0..1
Column
key
type
Command Prompt - java UmlTool

```
Class Classifier is renamed to Named

Correlation pattern Added subclasses
Additional subclasses of class Named are added wrt Classifier

Correlation pattern Rename class
Class Class is renamed to Table

Correlation pattern Unused source feature: isAbstract of Class
No other mapped feature has the same type

Correlation pattern Unused source feature: super.attr of Class
Suggest Set{self}->closure(super)->unionAll(attr) --> col

Correlation pattern Rename class
Class Attribute is renamed to Column

Correlation pattern Unused source feature: multiValued of Attribute
No other mapped feature has the same type

Correlation pattern Unused source feature: owner.super of Attribute
Suggest combining with     owner |--> owner

Correlation pattern Narrow type
Feature Attribute::type type narrowed from Named to Type in Column::type
!! Warning: source models may not map to valid target models!

Correlation pattern Class merging (horizontal)
Entity Named is target of 2 exclusive sources
```

Figure 3: Metamodel matching with correspondence patterns

relationships between numeric features (linear, quadratic and exponential relations), string transformations such as case changes and prefixing/suffixing, and collection transformations such as front/tail/reverse. Eg., the $Class \longmapsto Table$ matching could have example model data:

```
c1 : Class
c1.name = "Person"
c2 : Class
c2.name = "Family"
t1 : Table
t1.name = "PersonTable"
t2 : Table
t2.name = "FamilyTable"
```

This violates the feature matching $name \longmapsto name$, but agrees with a matching

$$name + \text{``}Table\text{''} \longmapsto name$$

and this is proposed.

3 Generating transformation specifications

Together with the metamodel matchings, the tool produces files *forward.txt* and *reverse.txt*, which contain transformation specifications for the two directions of the matching, in QVT-R, QVT-O, UML-RSDS, ATL and ETL.

While $\mathcal{TL}$ class matchings translate in general to rules in the MT languages, sometimes multiple class matchings must be combined in a single rule (in ATL), or one class matching is split into several rules (in QVT-R). In ATL and ETL, composite target features in mappings $f \longmapsto g.h$ must be implemented using additional lazy/called rules. In QVT-R, QVT-O and ETL rule inheritance is used to remove redundant mappings (in cases where the same feature mappings occur for a class and its superclass).

For example, the synthesised QVT-R of the above case is:

```
transformation tau(source: MM1, target: MM2)
{
  abstract top relation NamedElt2Named
  { checkonly domain source namedelt$x : NamedElt {};
    enforce domain target named$x : Named {};
  }

  abstract top relation Classifier2Type overrides NamedElt2Named
  { checkonly domain source classifier$x : Classifier {};
    enforce domain target type$x : Type {};
  }

  top relation DataType2Type overrides Classifier2Type
```

```
 { checkonly domain source datatype$x : DataType {};
   enforce domain target type$x : Type {};
 }

 top relation Class2Table overrides Classifier2Type
 { checkonly domain source class$x : Class {};
   enforce domain target table$x : Table {};
 }

 top relation Attribute2Column overrides NamedElt2Named
 { checkonly domain source attribute$x : Attribute {};
   enforce domain target column$x : Column {};
 }

 top relation Class2Type overrides Classifier2Type
 { checkonly domain source classx : Class {};
   enforce domain target typex : Type {};
 }

 top relation MapDataType2Type
 {  checkonly domain source
   datatype$x : DataType { name = datatype$x_name$value };
   enforce domain target
   type$x : Type { name = datatype$x_name$value, typeFlag = "DataType" };
   when {
   DataType2Type(datatype$x,type$x) }
 }

 top relation MapClass2Table
 {   domain source var$0 : Attribute {};
  checkonly domain source
   class$x : Class { name = class$x_name$value }
   { Set{class$x}->closure(super)->unionAll(attr)->includes(var$0)  };
   enforce domain target
   table$x : Table { col = table$x_col$x : Column {  },
name = class$x_name$value };
   when {
   Class2Table(class$x,table$x) and
          Attribute2Column(var$0,table$x_col$x) }
 }

 top relation MapAttribute2Column
 {
checkonly domain source
   attribute$x : Attribute { name = attribute$x_name$value,
     owner = attribute$x_owner$x : Class {  },
      type = attribute$x_type$x : Classifier {  } };
    enforce domain target
    column$x : Column { name = attribute$x_name$value,
      owner = column$x_owner$x : Table {  },
      type = column$x_type$x : Type {  } };
    when {
    Attribute2Column(attribute$x,column$x) and
              Class2Table(attribute$x_owner$x,column$x_owner$x) and
          Classifier2Type(attribute$x_type$x,column$x_type$x) }
  }

  top relation MapClass2Type
  {  checkonly domain source
    classx : Class { name = classx_name$value };
    enforce domain target
    typex : Type { name = classx_name$value };
    when {
    Class2Type(classx,typex) }
  }

}
```

An alternative mapping, aimed at generating bidirectional (bx) transformations is provided by the "Map TL to bx" option on the *File* menu. This generates QVT-R and UML-RSDS.

3.1 Generation of ETL

ETL is a widely-used hybrid transformation language within a general MDE and MT framework, Epsilon [5]. However it lacks a formal semantics, and published ETL specifications seem to have a high frequency of flaws or code 'smells' [1, 6], in particular due to the use of implicit rule invocation (the *equivalent*() operator).

Examining ETL cases, we find that the powerful hybrid nature of the language sometimes leads specifiers to (i) overuse imperative coding; (ii) use complex ad-hoc navigations in the target model to link target elements; (iii) overuse implicit invocation.

An automated synthesis procedure for ETL could help to reduce such problems by (i) using declarative constructs where possible; (ii) using standard strategies for navigating the target model; (iii) always using the version of *equivalent* parameterised by an explicitly-named rule to be invoked.

For each $\mathcal{TL}$ rule $\{Cond\}\ E \longmapsto F$, with E a concrete class, we generate an ETL concrete rule of the schematic form

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
```

```
  guard: ( ex.Cond )
  ...
}
```

A rule for abstract E is instead defined as an abstract ETL rule:

```
@abstract
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
  ...
}
```

When these ETL rules are used to convert the data of an E-typed MM_1 feature r to data of an F-typed MM_2 feature rr, the conversion of data is referred to as $r.equivalent('E2F')$. Because of the construction process of the $\mathcal{TL}$ specification τ by metamodel matching, distinct rules of τ always either have distinct source classes or (for class-splitting cases) distinct target classes. Hence there can only be one ETL rule with the given name.

ETL supports rule inheritance, this can be used in cases where both a general rule $\{Cond\}\ E \longmapsto F$ and a specialised rule $\{Cond1\}\ ESub \longmapsto F1$ are present in the $\mathcal{TL}$ specification τ, with $ESub$ a subclass of E and $F1$ equal to F or a descendant of F. The specialised rule is implemented as:

```
rule ESub2F1
  transform esubx : MM1!ESub
  to f1x : MM2!F1
  extends E2F {
  guard: ( esubx.Cond1 )
  ...
}
```

Feature mappings which are common to the generalised and specialised rules do not need to be explicitly implemented in $ESub2F1$. If $Cond1$ is the same as $Cond$, the guard of $ESub2F1$ can be omitted.

A feature mapping $f \longmapsto g$ of class mapping $\{Cond\}\ E \longmapsto F$ is represented by an ETL assignment $fx.g = ex.f;$ in cases where f and g are of value types and are not composed. Source compositions $r.f$ are evaluated as $ex.r.f$. If f and g are of class types, then $f \longmapsto g$ is implemented by

```
  fx.g = ex.f.equivalent('ERef2FRef');
```

where f and g are not composed, and $ERef$ is the class type of f, and $FRef$ the class type of g[2]. Likewise for composed source features $r.f$ of class type:

```
  fx.g = ex.r.f.equivalent('ERef2FRef');
```

[2]If there is no class mapping $ERef \longmapsto FRef$ then the most specific mapping with source class equal to or an ancestor of $ERef$ and target equal to or a descendant of $FRef$ is quoted.

If f or $r.f$ is of 0..1 multiplicity, the assignments are guarded:

```
  if (ex.f.isDefined())
  { fx.g = ex.f.equivalent('ERef2FRef'); }
```

and

```
  if (ex.r.f.isDefined())
  { fx.g = ex.r.f.equivalent('ERef2FRef'); }
```

In the case of a feature mapping $f \longmapsto r.g$, where r is of 1-multiplicity or 0..1 multiplicity, of concrete class type R, a new variable rx of type $MM2!R$ is defined:

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
    if (ex.f.isDefined())
    { var rx = new MM2!R;
      fx.r = rx;
      rx.g = ex.f.equivalent('ERef2G');
      ...
    }
}
```

The conditional test is included if f is of 0..1 multiplicity. Assignments to rx features implement the feature mapping $f \longmapsto g$, and other mappings $k \longmapsto l$, for each $k \longmapsto r.l$ which is a feature mapping of the $E \longmapsto F$ mapping.

If there is a direct mapping $f \longmapsto r$ and also composed mappings $g \longmapsto r.h$, assignments are needed for the composed mappings:

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
    fx.r = ex.f.equivalent('ERef2R');
    fx.r.h = ex.g.equivalent('G2H');
}
```

where r is of 1 multiplicity and f is of type *ERef*.

For other multiplicity r, a *for* loop implementation of $g \longmapsto r.h$ is used:

```
  for (rx in fx.r)
  { rx.h = ex.g.equivalent('G2H'); }
```

If there is no direct mapping $f \longmapsto r$, r is of $*$ multiplicity, and the only feature mappings $f \longmapsto r.g$ of $E \longmapsto F$ are such that f's upper multiplicity is 1[3],

[3]ie., f is of 1 or 0..1 multiplicity

then r can be defined by an additional variable $var\ rx = new\ MM2!R$; $rx.g = ex.f.equivalent('ERef2G')$; which has additional assignments $rx.l = ex.k.equivalent('K2L')$; for each $k \longmapsto r.l$, k of type K, l of type L. The rx creation is conditional on $ex.f.isDefined()$ if f is 0..1 multiplicity. Likewise, additional assignments are conditional if k is of 0..1 multiplicity.

If there are cases of mappings $f \longmapsto r.g$ where f and r are of *-multiplicity, then instead sets of R elements are created using lazy rules or operations. In this case, if f has a class type $ERef$, and $r.g$ has class type G, the mapping $f \longmapsto r.g$ in cases where r and f have * multiplicity can be implemented by introducing a new lazy rule:

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
  fx.r = ex.f.equivalent('MapERef2Rg');
}

@lazy
rule MapERef2Rg
  transform erefx : MM1!ERef
  to rx : MM2!R {
    rx.g = erefx.equivalent('ERef2G');
}
```

The effect of this approach is to produce a set of R objects, one for each element *erefx* of $ex.f$. R must be a concrete class for this to be valid.

Further updates to the r with additional $E \longmapsto F$ mappings $k \longmapsto r.l$ with $l.upper = 0$ or $l.upper \geq k.upper$ must be handled in further *for* loops of the $E2F$ rule:

```
for (rx in fx.r)
{ rx.l = ex.k.equivalent('K2L'); }
```

Further updates to the r with additional $E \longmapsto F$ mappings $k \longmapsto r.l$ with $l.upper \neq 0$ and $k.upper = 0$ require the creation of additional R objects via a new lazy rule $MapE22R1$:

```
  fx.r.addAll(ex.k.equivalent('MapE22Rl'));
```

If instead f has a value type T, the mapping $f \longmapsto r.g$ can be implemented by introducing a new called operation:

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
  fx.r.addAll(f.MapT2Rg());
}

operation T MapT2Rg()
{ var rx = new MM2!R;
  rx.g = self;
  return rx;
}
```

The generated code has a systematic structure, and all calls of rules via *equivalent* have been made explicit. Duplication of code due to common feature mappings of inheritance related $\mathcal{TL}$ rules is avoided by using ETL rule inheritance. However, excessive rule and transformation size may occur due to the size of the metamodels.

3.2 Generation of QVT-O transformations

QVT-O is structured around imperative *mapping* rules, which are the counterpart of QVT-R *relations*.

Mapping rules

```
mapping E::E2F() : F {}
```

implement the first phase of a $\mathcal{TL}$ class mapping $E \longmapsto F$, and rules

```
mapping E::MapE2F() : F
{ init
  { result :=
      self.resolveoneIn(E::E2F(),F); }
  result.q1 := t1;
  ...;
  result.qn := tn;
}
```

perform the second phase object linking and assignment of features according to feature mappings $p_i \longmapsto q_i$, where the *ti* compute the representation of *self.pi* in the target model. The *disjuncts* mechanism of QVT-O is used to express rule inheritance.

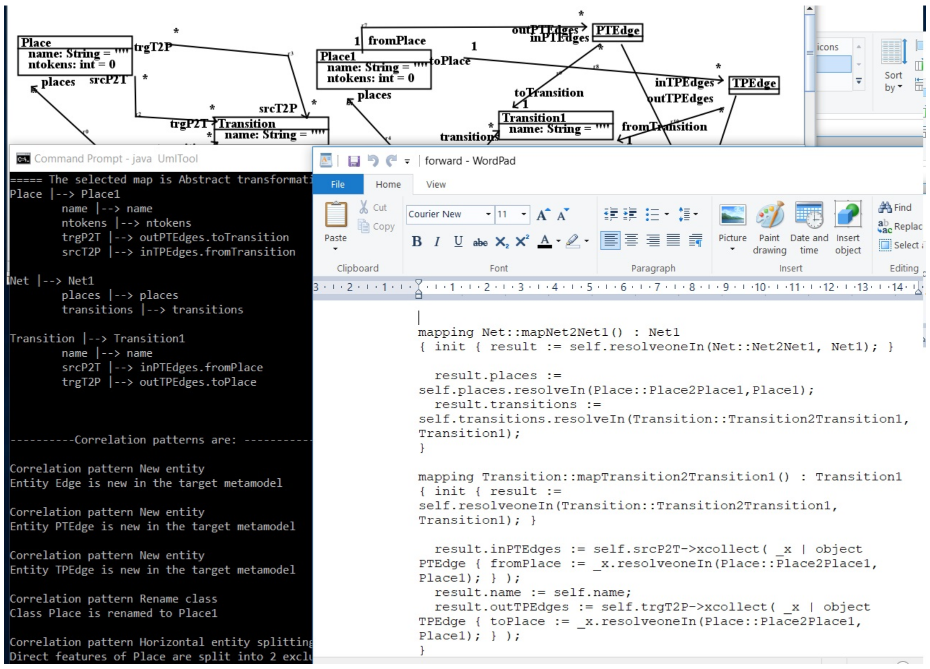
In QVT-O expression-to-feature mappings *expr* $\longmapsto$ *r* are implemented (schematically) by assignments of the form:

```
result.r := (self.expr)->xcollect( _x |
           _x.resolveoneIn(E1::E12F1,F1) );
```

An example of QVT-O synthesis is shown in Figure 4.

The overall scheduling of rules in a QVT-O transformation is carried out by the *main*() operation:

```
main()
{ in.objects[E]->map E2F();
```



```
===== The selected map is Abstract transformati
Place |--> Place1
        name |--> name
        ntokens |--> ntokens
        trgP2T |--> outPTEdges.toTransition
        srcT2P |--> inTPEdges.fromTransition

Net |--> Net1
        places |--> places
        transitions |--> transitions

Transition |--> Transition1
        name |--> name
        srcP2T |--> inPTEdges.fromPlace
        trgT2P |--> outTPEdges.toPlace



----------Correlation patterns are: ----------

Correlation pattern New entity
Entity Edge is new in the target metamodel

Correlation pattern New entity
Entity PTEdge is new in the target metamodel

Correlation pattern New entity
Entity TPEdge is new in the target metamodel

Correlation pattern Rename class
Class Place is renamed to Place1

Correlation pattern Horizontal entity splitting
Direct features of Place are split into 2 exclu
```

```
mapping Net::mapNet2Net1() : Net1
{ init { result := self.resolveoneIn(Net::Net2Net1, Net1); }

  result.places :=
self.places.resolveIn(Place::Place2Place1,Place1);
  result.transitions :=
self.transitions.resolveIn(Transition::Transition2Transition1,
Transition1);
}

mapping Transition::mapTransition2Transition1() : Transition1
{ init { result :=
self.resolveoneIn(Transition::Transition2Transition1,
Transition1); }

  result.inPTEdges := self.srcP2T->xcollect( _x | object
PTEdge { fromPlace := _x.resolveoneIn(Place::Place2Place1,
Place1); } );
  result.name := self.name;
  result.outTPEdges := self.trgT2P->xcollect( _x | object
TPEdge { toPlace := _x.resolveoneIn(Place::Place2Place1,
Place1); } );
}
```

Figure 4: Automated synthesis of weighted/unweighted Petri Nets case [14]

```
  ...
  in.objects[A]->map mapE2F();
  ...
}
```

All first phase mappings are performed before any of the second phase mappings. Unlike UML-RSDS, new key attributes are not needed, as QVT-O uses implicit tracing.

References

[1] N. Bonet, K. Garces, R. Casallas, M. Correal, R. Wei, *Influence of programming style in transformation bad smells: Mining of ETL repositories*, Education Symposium, 2017.

[2] H. Bunke, K. Riesen, *Recent advances in graph-based pattern recognition*, Pattern Recognition 44, pp. 1057–1067, 2011.

[3] S. Fang, K. Lano, *Extracting Correspondences from Metamodels Using Metamodel Matching*, Doctorial Symposium, STAF 2019.

[4] D. Kless, S. Milton, *Comparison of thesauri and ontologies from a semiotic perspective*, AOW 2010.

[5] D. Kolovos, L. Rose, A. Garcia-Dominguez, R. Paige, *The Epsilon Book*, 2018.

[6] K. Lano, S. Kolahdouz-Rahimi, M. Sharbaf, H. Alfraihi, *Technical debt in Model Transformation specifications*, ICMT 2018.

[7] K. Lano, S. Fang, *Automated synthesis of ATL transformations from metamodel correspondences*, Modelsward 2020.

[8] I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Cybernetics and control theory, 10(8), 1966, pp. 707–710.

[9] O. Macindoe, W. Richards, *Graph comparison using fine structure analysis*, IEEE 2nd Int. Conf. Social Computing, 2010.

[10] A. Maedche, S. Staab, *Comparing ontologies – similarity measures and a comparison study*, EKAW 2002.

[11] S. Melnik, H. Garcia-Molina, E. Rahm, *Similarity flooding: a versatile graph matching algorithm and its application to schema matching*, 18th International Conf. Data Engineering, IEEE, 2002, pp. 117–128.

[12] MetamodelRefactoring.org, *Metamodel refactorings catalog*, www.metamodelrefactoring.org, 2020.

[13] OMG, *MOF2 Query/View/Transformation v1.3*, 2016.

[14] B. Westfechtel, *Case-based exploration of bidirectional transformations in QVT Relations*, in *SoSyM 17*, 2018, pp. 989–1029.