Using the QVT-R analyser and code generator

K. Lano

January 6, 2020

1 Introduction

The Agile UML toolset provides a QVT-R parser and analyser, which converts QVT-R specifications into the UML-RSDS subset of UML. The Agile UML tools can then be used to generate executable implementations of the specifications in Java or other 3GLs.

This facility enables bidirectional (bx) transformation specification, and alternative transformation semantics to be adopted.

The latest version of the tools can be obtained from: https://nms.kcl.ac.uk/kevin.lano/uml2web/.

2 Analysis of QVT-R

Transformations in QVT-R consist of a set of named rules, or *relations*, for example:

```
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
  { checkonly domain source fm : FamilyMember
    { membername = n };
    enforce domain target m : Male
    { name = fm.memberOf.familyname + ", " + n };
    when
    { fm.memberOf.father->includes(fm) or
      fm.memberOf.sons->includes(fm)
    }
  }

}
```

Transformations are written to operate over source and target metamodels, in this case simplified metamodels of the Families to Persons case (Figure 1).

The metamodels should be loaded into the Agile UML tools first, eg., a text metamodel file *output/mm.txt* is loaded via the File menu option Recent. Then

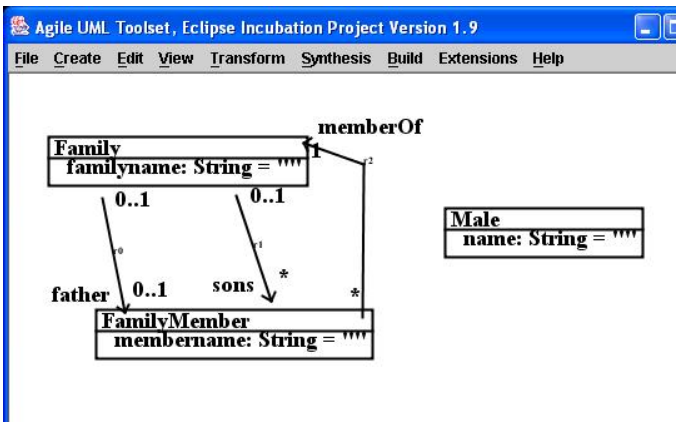

Figure 1: Simple Families to Persons metamodels

the QVT-R file *output/mm.qvt* is loaded with the File option Load transformation → Load QVT-R.

Syntax and type-checking is performed, with warning messages in the case of invalid syntax and undefined identifiers. Metrics calculation of the measures of [2] is also performed, and results are written to *output/qvtmeasures*.

The following consistency checks are made:

- If rule S is tested in the *when* clause of rule R, then S precedes R in the specification;
- If non-top rule S is called in R's *where* clause (directly or via another rule), and top relation P occurs in S's *when* clause, then P precedes R.
- If S is called in R's *where* clause, and in P's *when* clause, then R precedes P.

A UML-RSDS specification expressing the semantics of the QVT-R specification is also produced. This consists of three subtransformations, Pres, Con, Cleanup. For example:

```
**** UML-RSDS of QVT-R is:
Use Case, name: tau$Pres

FamilyMember2Male$trace@pre::
  not(fm : FamilyMember & fm.memberOf.father->includes(fm)) &
  not(fm : FamilyMember & fm.memberOf.sons->includes(fm)) =>
     self->isDeleted()


FamilyMember2Male$trace::
n = fm.membername & fm.memberOf.father->includes(fm) =>
```

```
     m.name = fm.memberOf.familyname + ", " + n


FamilyMember2Male$trace::
n = fm.membername & fm.memberOf.sons->includes(fm) =>
     m.name = fm.memberOf.familyname + ", " + n


Use Case, name: tau$Con

::
fm : FamilyMember & n = fm.membername & fm.memberOf.father->includes(fm) &
not(fm.traces$FamilyMember2Male$fm@pre->exists( tr$1 | true )) =>
     Male->exists( m | m.name = fm.memberOf.familyname + ", " + n &
     FamilyMember2Male$trace->exists( tr$0 | tr$0.fm = fm & tr$0.m = m ) )


::
fm : FamilyMember & n = fm.membername & fm.memberOf.sons->includes(fm) &
not(fm.traces$FamilyMember2Male$fm@pre->exists( tr$1 | true )) =>
     Male->exists( m | m.name = fm.memberOf.familyname + ", " + n &
     FamilyMember2Male$trace->exists( tr$0 | tr$0.fm = fm & tr$0.m = m ) )


Use Case, name: tau$Cleanup

Male::
traces$FamilyMember2Male$m@pre->isEmpty() =>
     self->isDeleted()
```

Inspection of this specification can identify if the QVT-R transformation has the intended semantics. The first *Pres* constraint removes traces whose *fm* element no longer satisfies the application condition of the rule. The second and third *Pres* constraints update *m.name* if *fm.memberOf.familyname* or *fm.membername* have changed, and the trace linking *m* and *fm* is still valid. The *Con* constraints create *Male* instances *m* for applicable *FamilyMember* instances *fm*, and create a trace linking these instances. *Cleanup* removes *Male* instances which are not linked to any *FamilyMember*.

If we rewrite the specification to put the *when* condition into the first domain condition:

```
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
  { checkonly domain source fm : FamilyMember
    { membername = n }
    { fm.memberOf.father->includes(fm) or
```

```
        fm.memberOf.sons->includes(fm)
    };
    enforce domain target m : Male
    { name = fm.memberOf.familyname + ", " + n };
  }
}
```

Then the generated UML-RSDS is almost identical to the previous version, and it can be seen that the semantics of the two versions are the same.

Likewise, if the effect of the relation is moved to the *where* clause:

```
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
  { checkonly domain source fm : FamilyMember { }
    { fm.memberOf.father->includes(fm) or
        fm.memberOf.sons->includes(fm)
    };
    enforce domain target m : Male { };
    where
    { m.name = fm.memberOf.familyname + ", " + fm.membername };
  }
}
```

The produced semantics for this version can also be seen to be unchanged.

Finally, we can introduce assignments for the persons to families direction:

```
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
  { checkonly domain source fm : FamilyMember { }
    { fm.memberOf.father->includes(fm) or
        fm.memberOf.sons->includes(fm)
    };
    enforce domain target m : Male { };
    where
    { m.name = fm.memberOf.familyname + ", " + fm.membername and
      fm.memberOf.familyname = m.name->before(", ") and
      fm.membername = m.name->after(", ") };
  }
}
```

In the forward direction these additional assignments are ignored, because they are not updates to target features of target object variables (m). Thus the semantics remains unchanged in this direction. In the reverse direction (swapping the *checkonly* and *enforce* labels on the domains), we get the semantics:

```
**** UML-RSDS of QVT-R is:
```

```
Use Case, name: tau$Pres

Postcondition 0 is:
  FamilyMember2Male$trace@pre::
not(m : Male) =>
     self->isDeleted()


Postcondition 1 is:
  FamilyMember2Male$trace::
true =>
   ( fm.memberOf.father->includes(fm) or fm.memberOf.sons->includes(fm) ) &
   fm.memberOf.familyname = m.name->before(", ") &
   fm.membername = m.name->after(", ")



Use Case, name: tau$Con

Postcondition 0 is:
  ::
m : Male & not(m.traces$FamilyMember2Male$m@pre->exists( tr$1 | true )) =>
   FamilyMember->exists( fm |
      ( fm.memberOf.father->includes(fm) or fm.memberOf.sons->includes(fm) ) &
      FamilyMember2Male$trace->exists( tr$0 | tr$0.fm = fm & tr$0.m = m &
      fm.memberOf.familyname = m.name->before(", ") &
      fm.membername = m.name->after(", ") ) )



Use Case, name: tau$Cleanup

Postcondition 0 is:
  FamilyMember::
traces$FamilyMember2Male$fm@pre->isEmpty() =>
     self->isDeleted()
```

This version will propagate name changes from the persons model to the families model – provided that new families are not required. Eg., if we change a *Male* person's name from "Smith, Jack" to "Smith, Peter", the respective *FamilyMember* instance will have its membername changed to "Peter".

However, for fully bidirectional execution (bx), the transformation must instead be written using only domain conditions:

```
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
```

```
  { checkonly domain source f : Family {}
      { f.familyname = m.name@pre->before(", ")} ;
    checkonly domain source fm : FamilyMember { }
    { fm.membername = m.name@pre->after(", ") and
      (f.father->includes(fm) xor
       f.sons->includes(fm))
    };
    enforce domain target m : Male { }
      { m.name = f.familyname@pre + ", " + fm.membername@pre
      };
  }
}
```

In the reverse direction this creates a new family for each new male person, even if a family with the same surname already exists. A similar rule is used to map between family members in the *mother* or *daughters* roles and *Female* persons. This version successfully passes test cases 1 to 8 of the benchmarx tests [1].

3 Executing QVT-R specifications

As with any UML-RSDS specification, the transformation produced from a QVT-R specification can be implemented in Java or other 3GL by the following steps:

- Type-check the specification, using the option *Synthesis* → *Typecheck*
- Select the *Synthesis* menu option *Generate design* to produce a design-level description. Warnings will be issued if there are data-update conflicts (two rules or two applications of the same rule writing to the same data). Select *y* to any question about the choice of bounded-loop implementation for a rule.
- On the *Build* menu, select *Java*4 for a basic Java implementation which should be executable under any JVM.
- The Java code is written to the *output* directory in files Controller.java, GUI.java, SystemTypes.java, etc. Compile these using *javac*.
- Execute *GUI.java* as

  ```
  java GUI
  ```

  Input files *in.txt* define models in text format, and are read in by the *Load model* option (Figure 2).

An example input model could be:

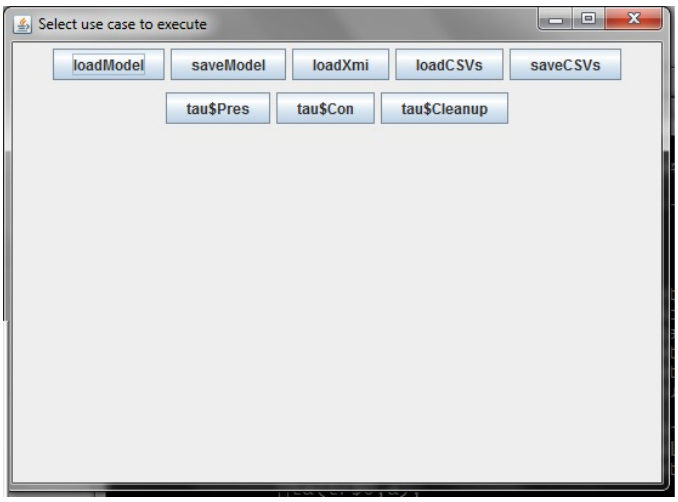

Figure 2: UI screen of Java implementation

```
f : Family
f.familyname = "Smith"
m : FamilyMember
m.membername = "John"
m : f.father
m.memberOf = f
s : FamilyMember
s.membername = "Jack"
s : f.sons
s.memberOf = f
```

Execution of *Con* and *Cleanup* produces the model:

```
familyx_0 : Family
familyx_0.familyname = "Smith"
familymemberx_0 : FamilyMember
familymemberx_0.membername = "John"
familymemberx_1 : FamilyMember
familymemberx_1.membername = "Jack"
malex_0 : Male
malex_0.name = "Smith, John"
malex_1 : Male
malex_1.name = "Smith, Jack"
familymember2male$tracex_0 : FamilyMember2Male$trace
familymember2male$tracex_1 : FamilyMember2Male$trace
familymemberx_0 : familyx_0.father
familymemberx_1 : familyx_0.sons
familymemberx_0.memberOf = familyx_0
```

```
familymember2male$tracex_0 : familymemberx_0.traces$FamilyMember2Male$fm
familymemberx_1.memberOf = familyx_0
familymember2male$tracex_1 : familymemberx_1.traces$FamilyMember2Male$fm
familymember2male$tracex_0 : malex_0.traces$FamilyMember2Male$m
familymember2male$tracex_1 : malex_1.traces$FamilyMember2Male$m
familymember2male$tracex_0.fm = familymemberx_0
familymember2male$tracex_0.m = malex_0
familymember2male$tracex_1.fm = familymemberx_1
familymember2male$tracex_1.m = malex_1
```

This is in agreement with our expectations: separate male elements are produced, and a trace element for each execution of a *Con* rule.

With the default semantics (mandatory creation), distinct target elements are created even if the resulting names are the same:

```
f : Family
f.familyname = "Smith"
m : FamilyMember
m.membername = "John"
m : f.father
m.memberOf = f
s : FamilyMember
s.membername = "John"
s : f.sons
s.memberOf = f
```

Execution of the transformation produces the model:

```
familyx_0 : Family
familyx_0.familyname = "Smith"
familymemberx_0 : FamilyMember
familymemberx_0.membername = "John"
familymemberx_1 : FamilyMember
familymemberx_1.membername = "John"
malex_0 : Male
malex_0.name = "Smith, John"
malex_1 : Male
malex_1.name = "Smith, John"
familymember2male$tracex_0 : FamilyMember2Male$trace
familymember2male$tracex_1 : FamilyMember2Male$trace
familymemberx_0 : familyx_0.father
familymemberx_1 : familyx_0.sons
familymemberx_0.memberOf = familyx_0
familymember2male$tracex_0 : familymemberx_0.traces$FamilyMember2Male$fm
familymemberx_1.memberOf = familyx_0
familymember2male$tracex_1 : familymemberx_1.traces$FamilyMember2Male$fm
familymember2male$tracex_0 : malex_0.traces$FamilyMember2Male$m
```

```
familymember2male$tracex_1 : malex_1.traces$FamilyMember2Male$m
familymember2male$tracex_0.fm = familymemberx_0
familymember2male$tracex_0.m = malex_0
familymember2male$tracex_1.fm = familymemberx_1
familymember2male$tracex_1.m = malex_1
```

Transformations can also be executed in incremental mode. If we change the above model by setting

```
familymemberx_1.membername = "Jack"
```

and re-execute the transformation (executing *Pres* is sufficient), we obtain the same model but with the updated setting

```
malex_1.name = "Smith, Jack"
```

Object moves are also supported. In this case, moving *familymemberx*_0 from *familyx*_0.*father* to *familyx*_0.*sons* has no effect on the target model.

4 Extensions of QVT-R/Medini QVT

4.1 Least-change semantics

The default semantics of our tools is the same as for Medini QVT (mandatory creation unless keys are defined for target elements; persistent traces). In cases where keys are not used, an alternative target resolution approach, close to the "check-before-enforce" of the standard, is available as "least change" instantiation. This is written as t <:= T $\{P\}$ instead of $t : T$ $\{P\}$ for target templates t. The effect is to select t where possible to satisfy P, instead of creating a new T instance and making P true for it. Using this operator we could rewrite the families to persons example as:

```
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
  { checkonly domain source fm : FamilyMember
    { membername = n };
    enforce domain target m <:= Male
    { name = fm.memberOf.familyname + ", " + n };
    when
    { fm.memberOf.father->includes(fm) or
      fm.memberOf.sons->includes(fm)
    }
  }
}
```

The semantics of this is similar to the previous version, only the definition of *Con* is changed:

```
Use Case, name: tau$Con

::
fm : FamilyMember & n = fm.membername & fm.memberOf.father->includes(fm) &
not(fm.traces$FamilyMember2Male$fm@pre->exists( tr$1 | true )) =>
     Male->existsLC( m | m.name = fm.memberOf.familyname + ", " + n &
     FamilyMember2Male$trace->exists( tr$0 | tr$0.fm = fm & tr$0.m = m ) )


::
fm : FamilyMember & n = fm.membername & fm.memberOf.sons->includes(fm) &
not(fm.traces$FamilyMember2Male$fm@pre->exists( tr$1 | true )) =>
     Male->existsLC( m | m.name = fm.memberOf.familyname + ", " + n &
     FamilyMember2Male$trace->exists( tr$0 | tr$0.fm = fm & tr$0.m = m ) )
```

Executing this version with the second input model above produces the output:

```
familyx_0 : Family
familyx_0.familyname = "Smith"
familymemberx_0 : FamilyMember
familymemberx_0.membername = "John"
familymemberx_1 : FamilyMember
familymemberx_1.membername = "John"
malex_0 : Male
malex_0.name = "Smith, John"
familymember2male$tracex_0 : FamilyMember2Male$trace
familymember2male$tracex_1 : FamilyMember2Male$trace
familymemberx_0 : familyx_0.father
familymemberx_1 : familyx_0.sons
familymemberx_0.memberOf = familyx_0
familymember2male$tracex_0 : familymemberx_0.traces$FamilyMember2Male$fm
familymemberx_1.memberOf = familyx_0
familymember2male$tracex_1 : familymemberx_1.traces$FamilyMember2Male$fm
familymember2male$tracex_0 : malex_0.traces$FamilyMember2Male$m
familymember2male$tracex_1 : malex_0.traces$FamilyMember2Male$m
familymember2male$tracex_0.fm = familymemberx_0
familymember2male$tracex_0.m = malex_0
familymember2male$tracex_1.fm = familymemberx_1
familymember2male$tracex_1.m = malex_0
```

Only one target element is produced with this semantics, as with "check-before-enforce" semantics. For this case, this is actually the incorrect semantics.

However, in the reverse direction we can write:

```
transformation tau(source: MM1, target: MM2)
{
```

```
  top relation FamilyMember2Male
  { enforce domain source f <:= Family {}
      { f.familyname = m.name@pre->before(", ") };
    enforce domain source fm : FamilyMember { }
    { fm.membername = m.name@pre->after(", ") and
      (f.father->includes(fm) xor
       f.sons->includes(fm))
    };
    checkonly domain target m : Male { }
      { m.name = f.familyname@pre + ", " + fm.membername@pre
      };
  }
}
```

This will place all male persons with surname F in one family with family name F. For example, for source model

```
m1 : Male
m1.name = "Smith, Jack"
m2 : Male
m2.name = "Smith, Peter"
```

with an empty target families model, the result is a single family:

```
familyx_0 : Family
familyx_0.familyname = "Smith"
familymemberx_0 : FamilyMember
familymemberx_0.membername = "Jack"
familymemberx_1 : FamilyMember
familymemberx_1.membername = "Peter"
malex_0 : Male
malex_0.name = "Smith, Jack"
malex_1 : Male
malex_1.name = "Smith, Peter"
familymemberx_0 : familyx_0.father
familymemberx_1 : familyx_0.sons
```

One element is added to the *father* of a family if *father* is empty, otherwise elements are added to *sons*.

4.2 Update-in-place transformations

Transformations can be written to operate on a single model, in which case all domains of the transformation relations have this model. An element is updated in-place by using it in both a source and target domain:

```
top relation R
{ checkonly domain m e : E { ... source domain ... };
```

```
  enforce domain m e : E { ... target domain defining update of e ... };
  when { ... }
  where { ... }
}
```

References

[1] A. Anjorin et al., *Benchmarx reloaded: a practical benchmark framework for bidirectional transformations*, BX 2017.

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

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