Appendix E: Evaluation

In this appendix we consider the cases of [12] and other bx examples, and use transformation patterns and the interpretation of QVT-R in UML-RSDS to provide systematic specifications of these.

In each case we used Version 1.9 of the Agile UML tools for UML-RSDS, available at https://projects.eclipse.org/projects/modeling.agileuml. We used the current version of the Medini QVT tools [3]. All tests were carried out on a Windows 10 i5-6300 dual core laptop with 2.4GHz clock frequency, 8GB RAM and 3MB cache. We generated example models using Java, replicating a basic model structure for each case multiple times in order to produce models of different scales. We computed execution times as an average over three independent runs for each model. All execution times are expressed in *ms*.

The code of all examples, together with their semantic interpretations in UML-RSDS and example execution scenarios, can be found at [9].

E.1 Batch-mode bidirectional transformations

Seven cases of batch-mode bidirectional transformations are considered in [12] to investigate the capabilities of QVT-R for bx specification:

1. Migration of person databases
2. Unweighted and weighted Petri nets
3. Unordered and ordered sets
4. Migration of bags
5. Expression trees and dags
6. Gantt diagrams and CPM networks
7. Ecore models and relational schemas.

We will respecify cases 1 to 6 using our approach, and evaluate the quality and efficiency of these versions compared to the versions of [12].

E.1.1 Migration of person databases

This is a simple case of the Lens pattern, and can be specified as follows:

```
top relation Database2Database1
{ enforce domain src d : Database { name = n };
  enforce domain trg d1 : Database1 { name = n };
}

top relation Person2Person1
{ enforce domain src p : Person
  { id = i, birthday = b, placeOfBirth = pob,
    database = d : Database {} };
 enforce domain trg p1 : Person1
  { id = i, birthday = b, placeOfBirth = pob,
    database = d1 : Database1 {} };
  when { Database2Database1(d,d1) }
  where
  { p1.name = p.firstName + ' ' + p.lastName and
    p.firstName = p1.name->before(' ') and
```

```
    p.lastName = p1.name->after(' ')
  }
}
```

This version is very similar to the original case of [12], but omits function definitions, and uses the UML-RSDS $\rightarrow$*before* and $\rightarrow$*after* operators on strings. Only the first where clause assignment is effective for update in the *trg* direction, and only the second and third in the *src* direction. Thus target data is write-only in both directions. Restricting to src models where all first names contain no spaces, the reverse transformation is a right inverse to the forward transformation.

The conditions (a) to (e) of Section 5 can be checked to hold in both *src* and *trg* directions. (a), (c) and (e) are direct, (b) holds since although the two relations both write to features of $Database1$ (in the *trg* direction), these features are distinct and unrelated (*name* in the first relation and *persons* in the second). Likewise for the *src* direction. Condition (d) holds because the mandatory feature *database* is set when $Person$ or $Person1$ instances are created.

This transformation can be executed also in incremental mode in both directions using our semantics. Execution traces of example scenarios can be found in the *qvt2umlrsds* dataset.

Table 1 summarises the performance measurements for Java implementations of the case study, generated from the UML-RSDS semantic representations for the forward and reverse directions. The time measure is an average of three repetitions for each model. We only include execution times below $1000s$.

The forward and reverse directions of this transformation are almost identical in terms of execution times, as are incremental-mode forward and reverse executions (changing the *firstName*/*name* of all $Person/Person1$ elements). The batch modes have approximately quadratic time complexity, and the incremental modes are approximately linear. The quadratic time complexity is expected because two object variables p, d appear in the LHS of the constraints interpreting the $Person2Person1$ relation in the forward direction, and likewise two variables $p1$, $d1$ in the reverse direction. Each variable is iterated over separately. The original batch mode transformation version of [12] executed using Medini QVT is less efficient, by a factor of around 10^3 for the largest case, this seems primarily due to the use of interpreted execution in Medini QVT.

$Person/Person1$ elements	500	1000	5000	10000	50000	100000
Batch Forward	0	10.7	114.3	421.3	10527.7	42522.7
Batch Reverse	5	10.3	109.3	443.3	10160	42296
Incremental Forward	0	5.3	0	10.3	20.7	36.7
Incremental Reverse	0	5	16	20.3	33	52.3
Medini Batch forward	605.7	2171.3	52,724.7	376,404.7	–	–
Medini Batch reverse	632.7	2173	57,027.7	254,766.7	–	–

Table 1: Execution times (ms) for person migration

E.1.2 Unweighted and weighted Petri nets

The forward transformation is a case of introducing intermediate classes (*Edge* subclasses *TPEdge* and *PTEdge*) between the original *Transition* and *Place* classes. More specifically, it comprises two applications of the class diagram refinement "Replace a many-many association by an intermediate class" [6]. As such, the transformation has a standard form, and it was automatically generated

from the metamodels using the metamodel matching techniques of [1]. The original references $trgT2P : Transition \longrightarrow_* Place$ and $trgP2T : Place \longrightarrow_* Transition$ are refined by compositions $outTPEdges.toPlace : Transition1 \longrightarrow_* Place1$ and $outPTEdges.toTransition : Place1 \longrightarrow_* Transition1$ via $TPEdge$ and $PTEdge$ respectively. This standard restructuring has as its right inverse the removal of the indirection, ie., flattening of the composition.

The specification of the QVT-R transformation is then a mechanical process, using Map objects before links to organise the relations. The matching of each *-multiplicity reference is defined in a separate rule, to avoid the complications discussed in Section 2.4:

```
top relation Place2Place1
{ enforce domain source place$x : Place
  { name = name$value, ntokens = ntokens$value };
  enforce domain target place1$x : Place1
  { name = name$value, ntokens = ntokens$value };
}

top relation Net2Net1
{ enforce domain source net$x : Net {};
  enforce domain target net1$x : Net1 {};
}

top relation Transition2Transition1
{ enforce domain source transition$x : Transition
  { name = name$value };
  enforce domain target transition1$x : Transition1
  { name = name$value };
}

top relation MapPlace2Place1
{ enforce domain source place$x : Place
  { trgP2T = place$x_trgP2T$x : Transition {  } };
  enforce domain target place1$x : Place1
  { outPTEdges = place1$x_outPTEdges$x : PTEdge
    { toTransition = place1$x_outPTEdges_toTransition$x : Transition1 {  } } };
  when
  { Place2Place1(place$x,place1$x) and
    Transition2Transition1(place$x_trgP2T$x,place1$x_outPTEdges_toTransition$x) }
}

top relation MapNet2Net1places
{ enforce domain source net$x : Net
    { places = net$x_places$x : Place {  } };
  enforce domain target net1$x : Net1
    { places = net1$x_places$x : Place1 {  } };
  when
  { Net2Net1(net$x,net1$x) and
    Place2Place1(net$x_places$x,net1$x_places$x) }
}

top relation MapNet2Net1transitions
{ enforce domain source net$x : Net
    { transitions = net$x_transitions$x : Transition {  } };
  enforce domain target net1$x : Net1
    { transitions = net1$x_transitions$x : Transition1 {  } };
```

```
  when
  { Net2Net1(net$x,net1$x) and
    Transition2Transition1(net$x_transitions$x,net1$x_transitions$x) }
}

top relation MapTransition2Transition1
{ enforce domain source transition$x : Transition
  { trgT2P = transition$x_trgT2P$x : Place {  } };
  enforce domain target transition1$x : Transition1
  { outTPEdges = transition1$x_outTPEdges$x : TPEdge
    { toPlace = transition1$x_outTPEdges_toPlace$x : Place1 {  } } };
  when
  { Transition2Transition1(transition$x,transition1$x) and
    Place2Place1(transition$x_trgT2P$x,transition1$x_outTPEdges_toPlace$x) }
}
```

The first phase of the transformation (the first three rules) performs a 1-1 mapping between *Place* and *Place*1, *Net* and *Net*1 and *Transition* and *Transition*1. The 1-1 mapping arises because of the default mandatory creation target resolution policy (if check-before-enforce was adopted, different source elements would be merged into single target elements by these relations). Alternatively, the 1-1 mapping could be enforced by making *name* a key of these elements. In the second phase (the remaining rules), links from *Transition* to *Place* correspond to *TPEdge* instances and associated links, and links from *Place* to *Transition* correspond to *PTEdge* instances and associated links.

Correctness conditions (a) and (e) hold by construction. (b) holds since *MapPlace*2*Place*1 and *MapTransition*2*Transition*1 update disjoint sets of features of *Place* and *Transition*. Similarly *MapNet*2*Net*1*places* and *MapNet*2*Net*1*transitions* update different features of *Net*/*Net*1. (c) holds since distinct instances of *PTEdge* are created for each distinct $x : place\$x.trgP2T$ in *MapPlace*2*Place*1 – assuming default *exists* semantics for the : *PTEdge* instantiation, and likewise for *TPEdge* in *MapTransition*2*Transition*1. (d) holds since *toTransition* is set for *PTEdge* instances when they are created, and likewise *toPlace* for *TPEdges*. The aggregation owner *net* of all places and transitions is also set by the *MapNet* rules. In the reverse direction the transformation is an example of flattening, with the compositions *outTPEdges.toPlace* and *outPTEdges.toTransition* replaced by $trgT2P$ and $trgP2T$ respectively. Change-propagation of addition, creation and deletion changes is supported for incremental execution in both directions.

Scenarios of the forward and reverse transformation are given in our dataset. Execution times are given in Table 2. The batch forward and reverse transformations have approximately quadratic time complexity. A comparison with Medini execution of the original version of [12] is also included, where execution was feasible: larger cases encountered "out of memory" errors, which our implementation does not.

E.1.3 Unordered and ordered sets

In the direction from ordered to unordered sets, the mapping of this transformation is from a recursively composed reference $elements \rightarrow closure(next)$ of *MyOrderedSet* to the *elements* reference of *MySet* (Figure 1). However there is no systematic way to invert such a mapping. Instead, the Object Indexing pattern can be used to introduce a String-valued key/identity attribute *elementId* into each of *Element* and *Element*1, which defines a 1-1 correspondence of instances of these classes based on equality of the key values:

```
key Element { elementId };
key Element1 { elementId };

top relation MySet2MyOrderedSet
{ enforce domain src m : MySet { name = n };
  enforce domain trg m1 : MyOrderedSet { name = n }
```

Place/*Place*1/ *Transition*/*Transition*1 elements	500	1000	5000	10000	50000	100000
Batch Forward	26.3	47	749	2957.7	67,790.3	–
Batch Reverse	21.3	48.3	740.7	2873.3	68,404.7	–
Incremental Forward	5	20.7	230.7	1180.3	22,370	–
Incremental Reverse	6.3	18.7	189	3117.7	25,604	–
Medini batch Forward	1381	4050	170,897.7	–	–	–
Medini batch Reverse	1311.7	3776.3	134,778	–	–	–

Table 2: Execution times (ms) for Petri Net mappings

```
}

top relation Element2Element1
{ enforce domain src e : Element
  { elementId = id, value = v, set = m : MySet {} };
  enforce domain trg e1 : Element1
  { elementId = id, value = v, orderedSet = m1 : MyOrderedSet {} };
  when
  { MySet2MyOrderedSet(m,m1) }
}
```

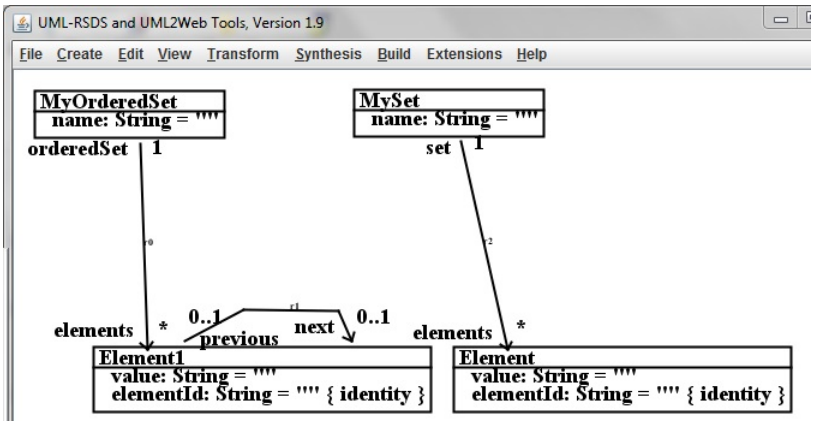

Figure 1: Unordered and ordered sets metamodels

The Map objects before links pattern is then used to define *next*/*previous* links based on lexicographic ordering of the (distinct) values of the *elementId*s:

```
top relation LinkElement1Element1
{ enforce domain src ea : Element { };
  enforce domain src eb : Element { };
  enforce domain trg ea1 : Element1 { next = eb1 : Element1 { } };
  when
```

```
  { ea.set = eb.set and
    Element2Element1(ea,ea1) and
    Element2Element1(eb,eb1) and
    ea.elementId < eb.elementId and
    ea.set.elements->forAll( ec |
      ec.elementId > ea.elementId implies ec.elementId >= eb.elementId ) }
}
```

This linking relation has no effect when executed in the *src* direction, apart from creating the relation trace. In this direction the relation should be omitted because it violates correctness condition (a), since it reads target data *elements*, *set*, *elementId* in the *when* clause.

Otherwise, properties (a) and (e) are clearly satisfied. (b) holds since distinct features *name* and *elements* of *MySet*/*MyOrderedSet* are updated by the first two relations. (c) holds for *LinkElement*1*Element*1 because the *next*/*previous* features cannot be set to contradictory values for the same elements by different relation executions: the lexicographic ordering is a total ordering. (d) holds because *set*/*orderedSet* are assigned to elements in *Element*2*Element*1.

By examining the logical semantics of the transformation, we can deduce that it can be executed in incremental mode in the forward direction.

For example:

- Creation of a new e : *Element* with $e.set = s$ and $e1.elementId < e.elementId$ and $e.elementId < e2.elementId$ where $e1$ and $e2$ are pre-existing elements in s with corresponding *Element*1 instances $e2' \in e1'.next$: $Con_{Element2Element1}$ creates a new e' : *Element*1 corresponding to e, and $Con_{LinkElement1Element1}$ executes on pairs $e1', e'$ and $e', e2'$ to set $e' \in e1'.next$ and $e2' \in e'.next$, hence also removing $e2'$ from $e1'.next$ (because *next* has 0..1 multiplicity).

- Deletion of such an intermediate e: the trace elements of *Element*2*Element*1$*trace* and *LinkElement*1*Element*1$*trace* linked to e are deleted. $Con_{LinkElement1Element1}$ becomes enabled on $(e1, e2)$ and establishes $e2' \in e1'.next$, hence removing e' from $e1'.next$ and from $e2'.previous$, thus also removing $e2'$ from $e'.next$. $Cleanup_{Element1}$ then deletes e'.

Examples of execution of these scenarios can be found at [9].

Unlike the previous two examples, this transformation is not bijective, because many different non-isomorphic target models (ordered sets) could correspond to the same source model (sets). Table 3 summarises the performance measurements for Java implementations of the case study, generated from the UML-RSDS semantic representations for the forward and reverse directions. The forward transformation is much more time consuming than the reverse, since it needs to create the ordering of elements, which is discarded by the reverse direction. The reverse batch mode has approximately quadratic time complexity, as does the forward incremental mode (for *value* changes of all *Element* instances). The times for Medini QVT execution of the version of [12] are also included. The reverse direction has quadratic time complexity, but is a factor of 563 slower than our version for the largest case. Because of the high time complexity of the forward direction of [12] it was not possible to execute any test case larger than 200 elements in the forward direction.

E.1.4 Migration of bags

Because the source and target models in this case are related in a semantically complex manner, we use Auxiliary models to split the transformation into a sequential composition of two subtransformations *Bag*1*Bag*2, *Bag*2*Bag*3 (Figure 2). The separate transformations each involve 1-1 mappings (*Element*1 to *Element*2 in *Bag*1*Bag*2 and *ElementCollection* to *Element*3 in *Bag*2*Bag*3).

The first subtransformation relates individual $e1$: *Element*1 elements to the *ElementCollection* representing the group of all elements of the bag which have the same *value* as $e1$. Firstly, using Object Indexing, we introduce unique identifier attributes *bagId*, *elementId* for bags and elements, to enforce 1-1 relations of *Bag*1 to *Bag*2 and *Element*1 to *Element*2. *ElementCollection*

Element/*Element*1 elements	500	1000	5000	10000	50000	100000
Batch Forward	239.7	921	60293	406,152	–	–
Batch Reverse	0	5	114.7	421.7	10146	42751
Incremental Forward	10.3	15.3	436	1721	–	–
Incremental Reverse	0	0	0	5	26	55.3
Medini batch Forward	–	–	–	–	–	–
Medini batch Reverse	541.3	1819.3	52,728.7	237,516.3	–	–

Table 3: Execution times (ms) for sets and ordered sets

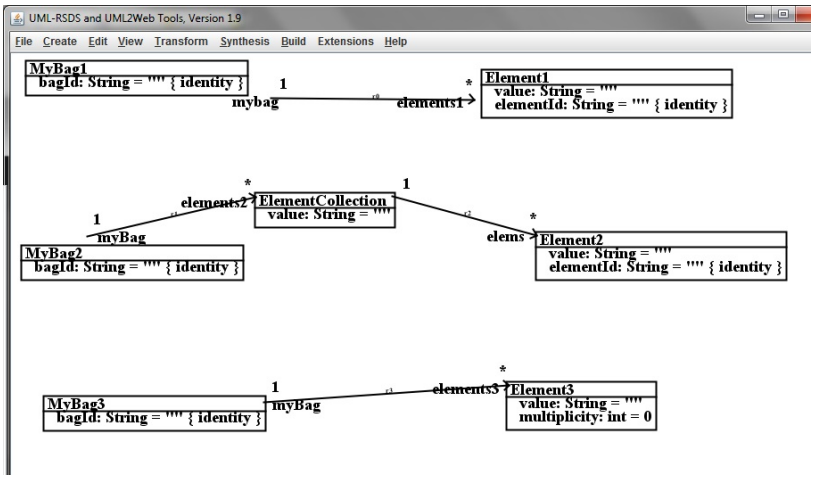

Figure 2: Migration of bags metamodels

is introduced as an intermediate class between bags and elements. It has a key formed as a combination of the linked bag and its own *value*:

```
key Bag1 { bagId };
key Element1 { elementId };
key Bag2 { bagId };
key Element2 { elementId };
key ElementCollection { myBag, value };

subtransformation Bag1Bag2(bag1 : Bag1, bag2 : Bag2)
{ top relation MyBag12MyBag2
  { enforce domain bag1 b1 : MyBag1 { bagId = id };
    enforce domain bag2 b2 : MyBag2 { bagId = id };
  }

  top relation Element12Element2
  { enforce domain bag1 e1 : Element1 { elementId = id, value = v };
    enforce domain bag2 e2 : Element2 { elementId = id, value = v };
  }

  top relation IntroduceElementCollection
  { enforce domain bag1 b1 : MyBag1
    { elements1 = e1 : Element1 { value = v } };
    enforce domain bag2 b2 : MyBag2
    { elements2 = ec : ElementCollection
      { value = v, elems = e2 : Element2 { } } };
    when
    { MyBag12MyBag2(b1,b2) and
      Element12Element2(e1,e2)
    }
  }
};
```

Because of the key definitions only one *ElementCollection* is introduced for each different *value* within one bag. In UML-RSDS this is expressed by

```
ElementCollection->existsLC( ec | ec : b2.elements2 & ec.value = v & ... )
```

This subtransformation satisfies correctness conditions (a), (b), (c) – because separate additions to the collection-valued references *elements*1, *elements*2 and *elems* are non-interfering. It also satisfies (d), (e), and is reversible, as for the general Introduce intermediate class pattern.

In the second subtransformation, *Bag*2*Bag*3, element collections are mapped to individual *Element*3 elements with multiplicity equal to the number of *elems* in the collection. As with element collections, *Element*3 instances are identified by their bag and value:

```
subtransformation Bag2Bag3(bag 2 : Bag2, bag3 : Bag3)
{ key Element3 { myBag, value };

  top relation MyBag22MyBag3
  { enforce domain bag2 b2 : MyBag2 { bagId = id };
    enforce domain bag3 b3 : MyBag3 { bagId = id };
  }

  top relation ElementCollection2Element3
  { enforce domain bag2 ec : ElementCollection
    { myBag = b2 : MyBag2 {}, value = v };
```

```
    enforce domain bag3 e3 : Element3 { myBag = b3 : MyBag3 {}, value = v };
    when { MyBag22MyBag3(b2,b3) }
  }
```

The critical step in the transformation is the mapping from an *Element*3 to an *ElementCollection* with a set of *Element*2 instances. The most direct way to achieve this is to create copies of an *Element*2 for each index i from 1 up to $e3.multiplicity$, for the given $e3 : Element3$:

```
  top relation MapElementCollection2Element3
  { enforce domain bag2 ec : ElementCollection
    { value = v };
    enforce domain bag3 e3 : Element3 { value = v };
    when
    { ElementCollection2Element3(ec,e3) }
    where
    { e3.multiplicity = ec.elems->size() and
      Integer.subrange(1,e3.multiplicity)->forAll( i |
         Element2->exists( e2 |
               e2.elementId = e3.myBag.bagId + "~" + v + "~" + i and
               e2.value = v and e2 : ec.elems ) )
    }
  }
}
```

The first conjunct of the *where* clause is only effective for update in the *bag*3 direction, and the second is only effective for update in the *bag*2 direction, because it updates data *Element*2, *Element*2 :: *elementId*, etc of *bag*2.

Note that the second clause violates semantic condition (a) in the *bag*2 direction, because it uses an explicit creation of *Element*2 instances in the *where* clause, and these created *Element*2 instances will not exist in any relation trace, and hence will be deleted by the *Cleanup* phase of the transformation. To correct this, a non-top relation should be used to create and record the *Element*2 instances:

```
relation Element32Element2
{ primitive domain i : int;
  enforce domain bag3 e3 : Element3
  { value = v, myBag = b3 : MyBag3 { bagId = bId } };
  enforce domain bag2 ec : ElementCollection
  { elems = e2 : Element2
    { elementId = bId + "~" + v + "~" + i, value = v } }
}
```

This is called as:

```
    Integer.subrange(1,e3.multiplicity)->forAll( i |
                                                 Element32Element2(i,e3,ec))
```

in the second where clause of *MapElementCollection*2*Element*3. Thus our language extension to permit subtransformations enables an improved solution to this case compared to the original version of [12].

Table 4 summarises the performance measurements for Java implementations of the case study, generated from the UML-RSDS semantic representations for the forward and reverse directions. The times for the two subtransformations are shown separately, as $bag1bag2 + bag2bag3$ in the forward direction and as $bag3bag2 + bag2bag1$ in the reverse. The step from $Bag3/Element3$ to $Bag2/Element2$ is the most time-consuming up to 50,000 elements. This step and the $Bag2$ to $Bag1$ step are both approximately quadratic, but the $Bag1$ to $Bag2$ step has a higher time

complexity, because it is a structure constructing transformation. The reverse direction of the version of [12] is not executable in Medini. The forward direction can only be executed for the smallest case.

Element/*Element*1 elements	500	1000	5000	10000	50000	100000
Batch Forward	10.3 + 0	28 + 0	550.3 + 0	4404 + 5	36,512.3 + 8.7	–
Batch Reverse	5.3 + 5.3	26 + 15.7	346.3 + 203	1216.3 + 754.7	36,962.7 + 17,900.7	145,972 + 71,318
Medini forward Batch	467,662.7	–	–	–	–	–

Table 4: Execution times (ms) for bag migration

E.1.5 Expression trees and dags

This is an example of an update-in-place bx with recurrent rewrites of models in both forward and reverse directions. For this case UML-RSDS is more suitable than QVT-R, since the case involves explicit deletion of elements (in the tree to dag direction).

Figure 3 shows our adaption of the metamodels of this case. We assume that basic expressions only occur as part of a larger expression, ie., *be.outgoing* is non-empty for each *be* : *BasicExpression*.

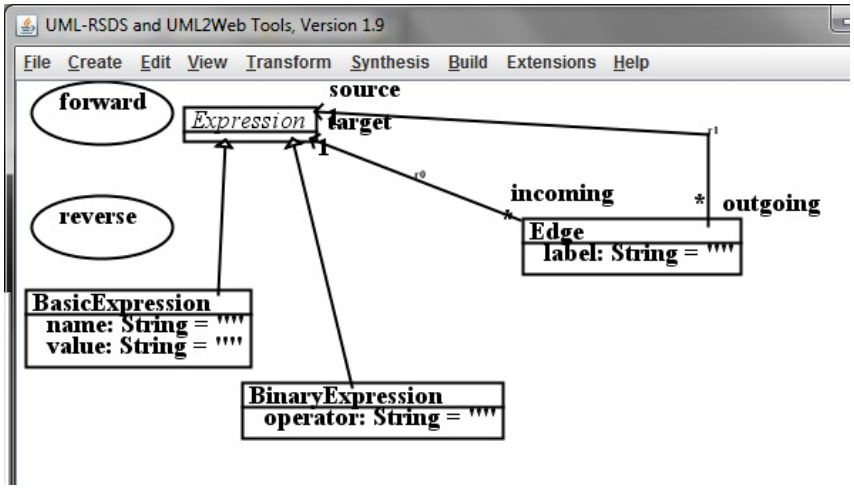

Figure 3: Trees and dags metamodels

In the forward direction there are two rewrite rules. The first operates on pairs of basic expressions *self*, *be* with *self* $\neq$ *be* with duplicate data, the two expressions are then merged into *self*, which gains all the outgoing edges of *be*, and *be* is deleted:

```
BasicExpression::
  be : BasicExpression & be /= self &
  be.name = name & be.value = value  =>
      outgoing->includesAll(be.outgoing) & be->isDeleted()
```

Basic expressions form the leaves of the expression tree. The above rule executes until all such duplicated leaf expressions are removed.

This rule could be imitated in QVT-R using update-in-place semantics:

```
top relation MergeBasicExpr
{ checkonly domain source be : BasicExpression { };
  checkonly domain source be1 : BasicExpression { };
  enforce domain target be : BasicExpression { };
  when
  { be /= be1 and be.name@pre = be1.name@pre and
    be.value@pre = be1.value@pre and
    be.outgoing@pre.size > 0
  }
  where
  { be1.outgoing@pre->forAll( e | be.outgoing->includes(e) ) }
}

top relation CopyBasicExpr
{ checkonly domain source be : BasicExpression { };
  enforce domain target be : BasicExpression { };
  when
  { be.outgoing@pre.size > 0 }
}
```

The effect of the first rule is to move all the outgoing edges from *be*1 to *be*. The second rule ensures that basic expressions with non-empty *outgoing* are preserved (ie., that other basic expressions are implicitly deleted). This is an example of deletion by selective copying. Successive iterations of the transformation result in a single *BasicExpression* for each *name*, *value* combination, with all the outgoing edges of any of the original basic expressions with that combination.

Similarly, two binary expressions with the same operator and arguments are merged by the following rule:

```
BinaryExpression::
  bx : BinaryExpression & bx /= self &
  bx.operator = operator & bx.incoming.source = incoming.source  =>
              outgoing->includesAll(bx.outgoing) & bx->isDeleted()
```

This rule repeats until all such pairs of binary expressions are removed.

The corresponding QVT-R is:

```
top relation MergeBinaryExpr
{ checkonly domain source bx : BinaryExpression { };
  checkonly domain source bx1 : BinaryExpression { };
  enforce domain target bx : BinaryExpression { };
  when
  { bx /= bx1 and bx.operator@pre = bx1.operator@pre and
    bx.incoming.source@pre = bx1.incoming.source@pre and
    bx.incoming@pre.size > 0
  }
  where
  { bx1.incoming@pre->forAll( f | bx.incoming->includes(f) ) and
    bx1.outgoing@pre->forAll( e | bx.outgoing->includes(e) ) }
}

top relation CopyBinaryExpr
{ checkonly domain source bx : BinaryExpression { };
  enforce domain target bx : BinaryExpression { };
  when
  { bx.incoming@pre.size > 0 }
}
```

The reverse transformation applies the model rewrites in the reverse direction, and in reverse order:

```
BinaryExpression::
  outgoing.size > 1 & nx : outgoing  =>
    BinaryExpression->exists( be | be.operator = operator & be.outgoing = Set{nx} &
      incoming->forAll( e |
        Edge->exists( ed | ed.label = e.label & ed.target = be &
                           ed.source = e.source ) ) ) & outgoing->excludes(nx)

BasicExpression::
outgoing.size > 1 & nx : outgoing  =>
    BasicExpression->exists( be | be.value = value & be.name = name &
                                  be.outgoing->includes(nx) ) & outgoing->excludes(nx)
```

The default semantics for *exists* is used here, to create distinct edges between distinct pairs of expressions, instead of least-change semantics. Again, it is possible to define an update-in-place QVT-R version based on the UML-RSDS rules. For basic expressions we have:

```
  top relation SplitBasicExpr
  { checkonly domain source be : BasicExpression
    { value = val, name = nme, outgoing = nx : Edge {} } { be.outgoing->size() > 1 };
    enforce domain target be1 : BasicExpression { value = val, name = nme };
    enforce domain target be : BasicExpression { };
    where
    { be1.incoming->includes(nx) & be.outgoing->excludes(nx) }
  }

  top relation CopyBasicExpr
  { checkonly domain source be : BasicExpression { };
    enforce domain target be : BasicExpression { };
    when
    { be.outgoing@pre.size <= 1 }
  }
```

Table 5 gives the execution times of the forward and reverse transformations in batch mode. We give both the UML-RSDS and QVT-R via UML-RSDS results, and the results of the original QVT-R version executed in Medini. The forward update-in-place QVT-R solution is less efficient than the pure UML-RSDS version, because of the additional trace management and complexity involved in QVT-R implicit deletion (the UML-RSDS version immediately discards spurious additional copies of expressions, but the QVT-R version retains these until the *Cleanup* phase). The *Con* phase of the QVT-R version takes between 10 to 20 times longer to execute than the *Cleanup* phase in this case. Only a single iteration of the transformation is required. Similarly in the reverse direction there is greater overhead of trace testing and management in the QVT-R via UML-RSDS version. Both directions seem to have approximately quadratic time complexity in all versions.

E.1.6 Gantt diagrams and CPM networks

This case involves a combination of two patterns: (1) Entity merging/splitting (horizontal) – both Gantt activities and dependencies map to CPM activities; (2) Introduce intermediate class (CPM *Event*). In the case of end-to-start dependencies of activities, the association *Activity* :: *incomingDependencies* is refined to the composition *Activity*1 :: *sourceEvent*.*incomingActivities*, and *Activity* :: *outgoingDependencies* is refined to the composition *Activity*1 :: *targetEvent*.*outgoingActivities*. Similarly for other forms of dependency. This transformation was automatically generated (for the end-to-start case) using the techniques of [1, 4], except for the assignment to *Activity*1 :: *name* for mapped dependencies, and the use of the least-change operator `<:=`.

Expression elements	500	1000	5000	10000	50000	100000
UML-RSDS Forward	11.3	16	135.7	521.3	13568.7	51893.7
QVT-R/UML-RSDS Forward	62.3	213.7	4539	18685	467,492	–
UML-RSDS Reverse	15.7	47	667	2498.7	59,430	253,103
QVT-R/UML-RSDS Reverse	31	65.7	1235.7	4773.3	118,792	–
Medini batch Forward	634	1426	25,898.3	151,066.3	–	–
Medini batch Reverse	893.7	1784.7	23,220	164,488.7	–	–

Table 5: Execution times (ms) for trees and dags

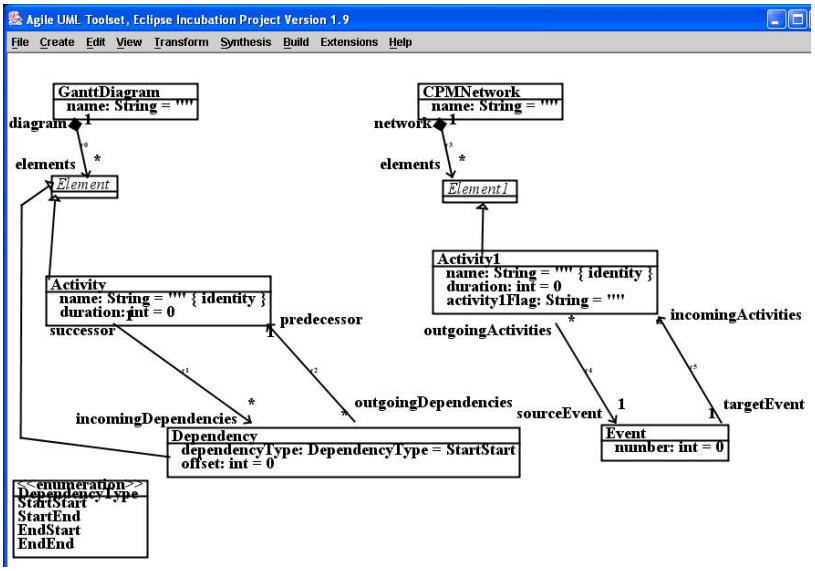

Figure 4: Gantt to CPM metamodels

Figure 4 shows the metamodels of the case.
The QVT-R specification is:

```
top relation GanttDiagram2CPMNetwork
{ enforce domain source ganttdiagram$x : GanttDiagram
  { name = ganttdiagram$x_name$value };
  enforce domain target cpmnetwork$x : CPMNetwork
  { name = ganttdiagram$x_name$value };
}

abstract top relation Element2Element1
{ enforce domain source element$x : Element
  { diagram = d : GanttDiagram {} };
  enforce domain target element1$x : Element1
  { network = n : CPMNetwork {} };
  when { GanttDiagram2CPMNetwork(d,n) }
}

top relation Activity2Activity1 overrides Element2Element1
{ enforce domain source activity$x : Activity
  { name = activity$x_name$value,
    duration = activity$x_duration$value };
  enforce domain target activity1$x : Activity1
  { activity1Flag = "Activity",
    duration = activity$x_duration$value,
    name = activity$x_name$value
  };
}

top relation Dependency2Activity1 overrides Element2Element1
{ enforce domain source dependency$x : Dependency
  { offset = dependency$x_offset$value };
  enforce domain target activity1$x : Activity1
  { activity1Flag = "Dependency",
    duration = dependency$x_offset$value };
  where
  { activity1$x.name =
      dependency$x.predecessor.name + "-->" + dependency$x.successor.name
  }
}

top relation MapActivity2Activity1incoming
{ enforce domain source activity$x : Activity
  { incomingDependencies = activity$x_incomingDependencies$x : Dependency {  } };
  enforce domain target activity1$x : Activity1
  { sourceEvent = activity1$x_sourceEvent$x <:= Event
    { incomingActivities = activity1$x_sourceEvent_incomingActivities$x :
                             Activity1 {  } } };
  when
  { Activity2Activity1(activity$x,activity1$x) and
    Dependency2Activity1(activity$x_incomingDependencies$x,
                    activity1$x_sourceEvent_incomingActivities$x) }
}
```

```
top relation MapActivity2Activity1outgoing
{ enforce domain source activity$x : Activity
  { outgoingDependencies = activity$x_outgoingDependencies$x : Dependency {  } };
  enforce domain target activity1$x : Activity1
  { targetEvent = activity1$x_targetEvent$x <:= Event
    { outgoingActivities = activity1$x_targetEvent_outgoingActivities$x :
                           Activity1 {  } } };
  when
  { Activity2Activity1(activity$x,activity1$x) and
    Dependency2Activity1(activity$x_outgoingDependencies$x,
                activity1$x_targetEvent_outgoingActivities$x) }
}
```

The variable *activity*1*Flag* is introduced by the Entity merging horizontal pattern, and records the origin of the *Activity*1 element as either a Gantt *Activity* or *Dependency*. This enables the transformation to be correctly reversed.

Correctness conditions (a) and (e) hold by construction. Condition (b) holds because *Activity*2*Activity*1 and *Dependency*2*Activity*1 update disjoint sets of *Activity*1 elements. Likewise for *MapActivity*2*Activity*1*incoming*/*outgoing*. (c) holds because the *outgoingActivities* and *incomingActivities* of events are sets and hence additions of an *Activity*1 instance to these features by different rule applications are non-interfering. Least-change *existsLC* semantics is used for the *Event* instantiations, to ensure that single source and target *Event* instances are maintained for each *Activity* in the source model. For (d), for each *Element*/*Element*1, the mandatory link to the *diagram*/*network* of the element is established by the transformation. However, source events are not necessarily created for activities if they have no incoming dependencies, likewise target events may be omitted for activities without outgoing dependencies. To correct this, the events would need to be created by *Activity*2*Activity*1 and looked up by the *MapActivity* rules.

Table 6 shows execution times for the batch and incremental modes of this case study. The batch transformations have approximately quadratic time complexity. We attempted to execute the original transformation of [12] in Medini, however the forward transformation could not be executed for more than 100 elements (with execution time 59,254ms) without producing a stack overflow error, and the reverse could not execute a case of 50 elements.

Activity/*Activity*1 elements	500	1000	5000	10000	50000	100000
Batch Forward	0	20.7	237	796	17437	68,892.3
Batch Reverse	10.7	21	222	734.7	16778.3	69,693.7
Incremental Forward	5.3	5.3	19.7	17.7	95.3	803
Incremental Reverse	5.3	10.3	10.3	24	115.7	2485

Table 6: Execution times for Gantt to CPM

E.1.7 Ecore models and relational schemas

This transformation combines horizontal entity merging and vertical entity splitting. For example, in the forward direction an *EClass* maps to a *Table* and a linked *PrimaryKey* (vertical entity splitting), and in the reverse direction a *Column* may map to either an *EAttribute* or an *EReference*. Auxiliary metamodel is also used to add an *Annotation* class in the schema metamodel to record specific Ecore model information which is discarded by the main forward translation [12]. Intro-

duce intermediate class is used in the forward direction to represent Ecore inheritance via SQL foreign keys.

E.2 Incremental bidirectional transformations

We consider three incremental bx cases which illustrate the application of MT design patterns to QVT-R:

- Trees to graphs [5]
- UML to Python
- Hsm2nhsm [10]

E.2.1 Trees to graphs

This case is an example of introducing/removing an intermediate class (*Edge*) into/from a self-association. Figure 5 shows the metamodels of this transformation. The transformation was automatically generated from these metamodels using the techniques of [1].

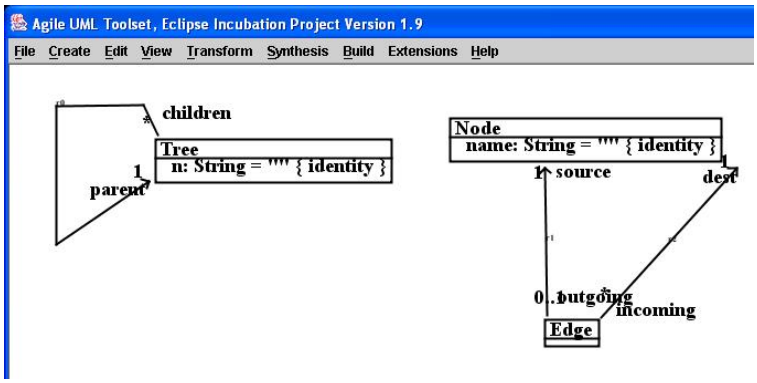

Figure 5: Tree and graph metamodels

The conceptual mapping here is from *Tree* features *parent* and *children* to the *Node* composed features *outgoing.dest* and *incoming.source*.

The Map objects before links pattern is used to map source elements in one relation, with element links mapped in a second relation:

```
top relation Tree2Node
{ enforce domain source treex : Tree
  { n = treex_n$value };
  enforce domain target nodex : Node
  { name = treex_n$value };
}
```

For each tree, the link to its parent tree is mapped to the graph node outgoing edge, this edge is also added to the incoming edges for the node representing the parent:

```
top relation MapTree2Node
{ enforce domain source treex : Tree
  { parent = treex_parent$x : Tree {  } };
  enforce domain target nodex : Node
  { outgoing = nodex_outgoing$x : Edge
    { dest = nodex_outgoing_dest$x : Node {  } } };
  when
```

```
  { Tree2Node(treex,nodex) and
    Tree2Node(treex_parent$x,nodex_outgoing_dest$x) }
}
```

Correctness properties (a), (b) and (e) are clearly true. (c) holds since *incoming* and *children* are set-valued with unbounded multiplicity and hence addition updates to them in one relation application cannot invalidate positive membership properties established for them in another relation. (d) holds since *source* and *dest* are set for each edge on its creation, and *parent* is set when a tree is defined from a node.

The transformation is bijective, with tree instances corresponding 1-1 to nodes, and tree to tree parent links corresponding 1-1 to edges. The reverse transformation only produces valid tree models if executed on a graph model where each node has $outgoing \rightarrow size() \leq 1$. We permit 'trees' to have cycles in the *parent* relation. The root tree is its own parent. The transformation is incremental in both directions for creation/deletion and addition changes. Example scenarios are included in the *qvt2umlrsds* dataset.

Table 7 gives efficiency results for the forward and reverse transformations. The time complexity appears to be approximately linear in both directions.

Tree/Node elements	500	1000	5000	10000	50000	100000
Batch Forward	5	10.3	31.3	68	219	502
Batch Reverse	5.3	11.7	22.3	59.3	212.7	427.3
Incremental Forward	0	0	5.3	15.3	52.3	94
Incremental Reverse	0	5	5	15.3	52.3	101

Table 7: Execution times (ms) for tree2graph

E.2.2 UML to Python

An example case of the flattening pattern is the flattening of class inheritance hierarchies when mapping from UML to Python (Figure 7): each class e in UML is represented in Python by a class c that owns all direct and inherited attributes of e, and the inheritance relation is discarded.

This is an example of Recursive *-accumulation (Figure 6) of the UML attribute sets into the Python attribute sets. This form of flattening maps an association closure in the source model to a single association in the target model, in this case the set

$$Set\{self\} \rightarrow closure(general) \rightarrow unionAll(ownedAttribute)$$

evaluated on *Entity* is mapped to *PythonClass* :: *attributes*.

This transformation can be reversed by reconstructing UML inheritance based on the idea that class $e1$ is a subclass of $e2$ iff the collection of attributes of the flattened class $c1$ is a superset of those of the flattened class $c2$. The *Auxiliary metamodel* pattern [7] is used to introduce a new attribute for each UML class, so that no two different classes have the same owned or accumulated attributes.

UML properties are copied 1-1 to Python attributes by a top relation *Property2Attribute*, and UML classes are copied 1-1 to Python classes by a top relation *Entity2PythonClass*:

```
top relation Entity2PythonClass
{ enforce domain design e : Entity { name = n };
  enforce domain py c : PythonClass { name = n };
}
```

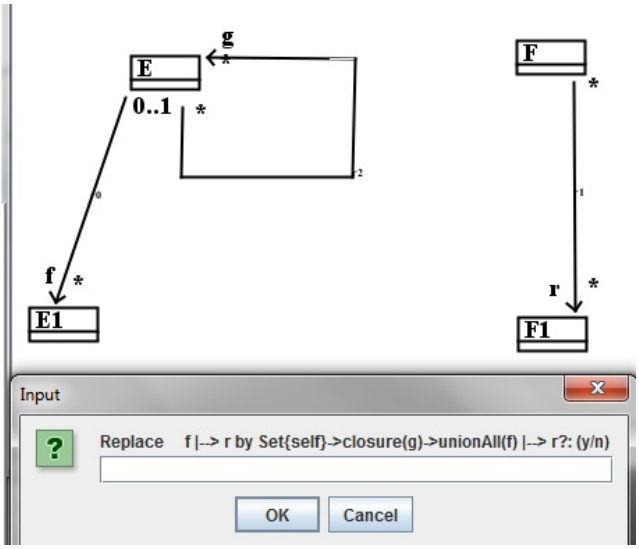

Figure 6: Recursive *-accumulation

name is a unique key for *Entity* and *PythonClass*, and *elementId* for *Property* and *Attribute*.

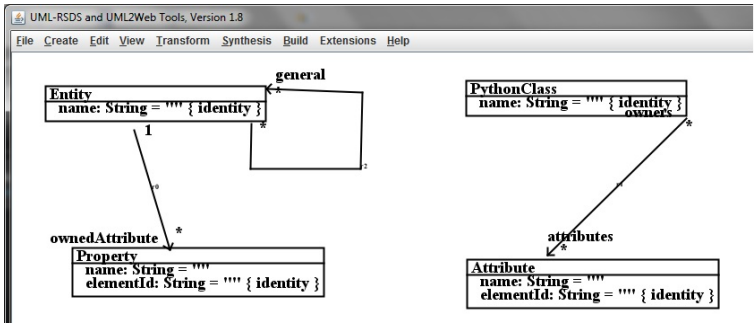

Figure 7: Extracts of UML design and Python metamodels

The Map objects before links pattern then applies to connect Python attributes to classes in the *py* direction, and UML properties to classes in the *design* direction:

```
top relation LinkAttributes2PythonClasses
{ enforce domain design e : Entity { ownedAttribute = prop : Property {} };
  enforce domain py c : PythonClass { attributes = a : Attribute {} }
    { a.owners@pre->selectMinimals(attributes@pre->size())->includes(c) };
  when { Property2Attribute(prop,a) and Entity2PythonClass(e,c) }
}
```

In the *py* direction only features of c and a can be modified to make the relation true. The owned attributes of e are copied to c. Any other c' that already contains these attributes must be less abstract than c, ie., it corresponds to a subclass of e, and hence the additional condition of the c domain is automatically true. The condition is not effective as an update in the *py* direction and is not included in the Con_τ rule succedent in this direction – otherwise it would break correctness condition (a). The @pre suffix on target data in the condition is required because otherwise the condition would violate property (a): target data being read in the *py* direction.

In the *design* direction only features of *prop* and e are writable. Attributes a are only copied from c to e if they do not exist in a more abstract Python class, ie., they are owned by e instead

of being inherited. In a similar manner, UML inheritance can be reconstructed on the basis of subsetting of attribute sets in the Python model:

```
top relation LinkGeneralisations
{ enforce domain design e1 : Entity { };
  enforce domain design e2 : Entity { } { e1.general->includes(e2) };
  enforce domain py c1 : PythonClass { };
  enforce domain py c2 : PythonClass { }
    { c1.attributes->includesAll(c2.attributes@pre) and
      PythonClass->forAll( cx |
          c1.attributes@pre->includesAll(cx.attributes@pre) and
          cx.attributes@pre->includesAll(c2.attributes@pre) implies
            (cx = c1 or cx = c2)) };
  when { LinkAttributes2PythonClasses(e1,c1) and
         LinkAttributes2PythonClasses(e2,c2) and
         e1 <> e2 and c1 <> c2 }
}
```

The condition of $c2$ states that $c2$ represents a direct superclass of $c1$ – there is no strictly intermediate class cx in the ordering induced by subsetting of class attribute sets. The use of *attributes*@pre in the effective update (first predicate) of the *where* clause is necessary to avoid breaking correctness condition (a). The second predicate is not effective for update (cx cannot be updated within the expression) and is used only as a test, in the *design* direction.

Correctness condition (b) holds in the forward direction since the three relations either write to separate features of *PythonClass*, or add elements to the set-valued *owners* and *attributes* features. In the reverse direction the same applies for *ownedAttribute* and *general* features of *Entity*. (d) is satisfied since there are no mandatory references. Example scenarios can be found in the *qvt2umlrsds* dataset.

Table 8 gives efficiency results for the forward and reverse transformations. The relatively high execution times are due to the complex conditions involved in the transformation relations, particularly in the reverse direction, which reconstructs inheritance information. However the time complexity appears to be approximately quadratic in both directions.

Property/Attribute elements	500	1000	5000	10000	50000	100000
Batch Forward	93.7	336.3	7800.3	29843.7	879,615	–
Batch Reverse	213.7	767.3	18617.3	74922.3	–	–
Incremental Forward	0	5	16	15.3	83.3	–
Incremental Reverse	5	0	16	26	–	–

Table 8: Execution times (ms) for UML2Python

Given the restriction that $p1 \neq p2 \Rightarrow p1.attributes \neq p2.attributes$, the transformation is bijective. We have used a similar strategy to rewrite the classic UML to RDBMS example as a bx, without the quality flaws of excessive coupling and mutually-recursive relations present in the original version [11]. The revised version uses only *when* dependencies, with no *where* clauses.

Other types of flattening are an extension of this situation, where the discarded nodes need to be reconstructed. For example, if only leaf Python classes had been retained, other classes could be reconstructed based on finite sets ps of leaf classes such that $ps \rightarrow intersectAll(attributes)$ is non-empty. In the *design* direction a UML class would be constructed for each such set. Inheritance would be based on the subsetting of these sets.

E.2.3 Hsm2nhsm

Another example of flattening is the hierarchical to non-hierarchical state machine transformation of [10]. Figure 8 shows the source and target metamodels of our version of this case. The forward transformation was derived from the metamodels using the approach of [1], together with the use of the recursive *-accumulation flattening pattern.

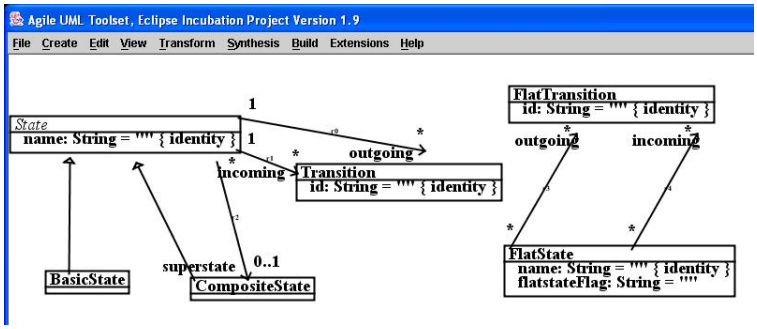

Figure 8: Hierarchical to flat state machine metamodels

States and transitions are copied from the source to the target model, with all state inclusion structure being discarded. There is horizontal entity merging of *CompositeState* and *BasicState* into *FlatState*, and a new flag attribute is used to distinguish these cases:

```
abstract top relation State2FlatState
{ enforce domain source state$x : State
  { name = v };
  enforce domain target flatstate$x : FlatState
  { name = v };
}

top relation CompositeState2FlatState overrides State2FlatState
{ enforce domain source compositestate$x : CompositeState { };
  enforce domain target flatstate$x : FlatState
  { flatstateFlag = "CompositeState" };
}

top relation Transition2FlatTransition
{ enforce domain source transition$x : Transition
  { id = v };
  enforce domain target flattransition$x : FlatTransition
  { id = v };
}

top relation BasicState2FlatState overrides State2FlatState
{ enforce domain source basicstate$x : BasicState
  { };
  enforce domain target flatstate$x : FlatState
  { flatstateFlag = "BasicState" };
}
```

The outgoing and incoming transitions of a flattened state are all its direct outgoing/incoming transitions, together with those from all its recursively containing states:

```
top relation MapCompositeState2FlatStateincoming
```

```
{ checkonly domain source v1 : Transition {};
  enforce domain source compositestate$x : CompositeState { }
  { Set{compositestate$x}->closure(superstate)->unionAll(incoming)->includes(v1) };
  enforce domain target flatstate$x : FlatState
  { incoming = flatstate$x_incoming$x : FlatTransition {  } };
  when
  { CompositeState2FlatState(compositestate$x,flatstate$x) and
    Transition2FlatTransition(v1,flatstate$x_incoming$x) }
}

top relation MapCompositeState2FlatStateoutgoing
{ checkonly domain source v0 : Transition {};
  enforce domain source compositestate$x : CompositeState { }
  { Set{compositestate$x}->closure(superstate)->unionAll(outgoing)->includes(v0) };
  enforce domain target flatstate$x : FlatState
  { outgoing = flatstate$x_outgoing$x : FlatTransition {  } };
  when
  { CompositeState2FlatState(compositestate$x,flatstate$x) and
    Transition2FlatTransition(v0,flatstate$x_outgoing$x) }
}

top relation MapBasicState2FlatStateincoming
{ checkonly domain source v1 : Transition {};
  enforce domain source basicstate$x : BasicState
  { Set{basicstate$x}->closure(superstate)->unionAll(incoming)->includes(v1) };
  enforce domain target flatstate$x : FlatState
  { incoming = flatstate$x_incoming$x : FlatTransition {  } };
  when
  { BasicState2FlatState(basicstate$x,flatstate$x) and
    Transition2FlatTransition(v1,flatstate$x_incoming$x) }
}

top relation MapBasicState2FlatStateoutgoing
{ checkonly domain source v1 : Transition {};
  enforce domain source basicstate$x : BasicState
  { Set{basicstate$x}->closure(superstate)->unionAll(outgoing)->includes(v1) };
  enforce domain target flatstate$x : FlatState
  { outgoing = flatstate$x_outgoing$x : FlatTransition {  } };
  when
  { BasicState2FlatState(basicstate$x,flatstate$x) and
    Transition2FlatTransition(v1,flatstate$x_outgoing$x) }
}
```

The transformation follows the standard template for recursive *-accumulation. Conditions (a) and (e) hold by construction. Condition (b) holds since basic and composite states are mapped to disjoint sets of flat states (since *name* is a key for states). (c) holds since the features *incoming* and *outgoing* are set-valued and hence successive additions to these are non-interfering. (d) holds since there are no mandatory references. Separate inverse rules for the $Map*$ relations are needed for the reverse transformation, to reconstruct *superstate* and the *incoming* and *outgoing* references of the hierarchical model from the flattened model. These are defined as a reverse of recursive *-accumulation (Figure 6). The reverse direction is specified by two linking relations, which

reconstruct g and f based on r, according to the mappings:

$$F \rightarrow select(a \mid r \rightarrow includesAll(a.r) \ and \ r \neq a.r) \rightarrow selectMaximals(r.size) \longmapsto g$$
$$r - F \rightarrow select(a \mid r \rightarrow includesAll(a.r) \ and \ r \neq a.r) \rightarrow unionAll(r) \longmapsto f$$

These are valid if $a1 \neq a2 \Rightarrow a1.r \neq a2.r$, so that g has no cycles, where F is the owner of r and is the image of the class on which g is a self-association. The idea of the reverse mapping is that $gx : F$ represents a g-ancestor of $fx : F$ iff $fx.r \rightarrow includesAll(gx.r)$ *and* $gx \neq fx$. The gx representing immediate g-parents of fx are those g-ancestors with maximal $gx.r$ size. In the state machine example, both *incoming* and *outgoing* of flat states need to be considered in reconstructing *superstate*: the idea is that flat state sx represents a containing state of flat state fx iff

$$fx \neq sx \ and$$
$$fx.outgoing \rightarrow includesAll(sx.outgoing) \ and$$
$$fx.incoming \rightarrow includesAll(sx.incoming)$$

Table 9 shows the performance data for this case. The time complexity is approximately quadratic in both directions. As with the UML to Python case, the reverse transformation is significantly more costly than the forward direction, because these reverse transformations involve reconstruction of model structure, compared to the flattening forward transformations, which discard model structure.

State/Transition elements	500	1000	5000	10000	50000	100000
Batch Forward	216.3	843.7	17307.7	73,618.3	–	–
Batch Reverse	489.3	1907	43,533.7	199,853.3	–	–
Incremental Forward	0	12.7	17.3	47	–	–
Incremental Reverse	191.7	733.7	17,657.7	77,116	–	–

Table 9: Execution times (ms) for HSM2FSM

E.3 Comparison

The above cases can be evaluated in terms of the quality metrics of [8] and in terms of the bx properties they support. Table 10 compares previous versions of the cases in QVT-R or ETL (for the tree to graph case) with the QVT-R and UML-RSDS versions defined in this paper, with regard to the number of quality flaws per LOC. For the tree to dag case the data of the update in place QVT-R version is used in the forward direction, and data of the UML-RSDS version in the reverse direction. We give the performance gain ratios of our versions relative to the original versions (for the cases of [12]). In each case our solutions have the same or fewer flaws than previous solutions, and are more efficient for batch mode execution.

Table 11 summarises the bx properties of our solutions.

We have therefore improved on the properties of the solutions of [12] in two cases: the mapping of bags has been specified by a bidirectional transformation instead of by two separate forward and reverse transformations, and for trees and dags we specified forward and reverse transformations which are closely related and mutually inverse. We also provided incremental solutions for 4

Case	*Original version Flaws/LOC*	*Revised version Flaws/LOC*	*Performance gain*
Tree to graph [5]	1/15	0/17	–
UML to Python	–	0/30	–
Hsm2nhsm	2/48	1/70	–
Person migration	0/63	0/19	893 (forward) 575 (reverse)
Weighted/unweighted Petri nets	2/115	1/60	228 (forward) 182 (reverse)
Unordered/ordered sets	1/121	0/29	563 (reverse)
Migration of bags	1/157	0/66	45,404 (forward)
Expression trees/dags	8/439	0/80	8.1 (forward) 34.5 (reverse)
Gantt2CPM	10/378	1/54	–

Table 10: Quality flaw and performance measures for cases

of the 6 cases of [12], and provided a deterministic solution for the sets/ordered sets case. We improved the Hsm2nhsm transformation of [10] by eliminating the circular calling dependencies of the previous solution. The application of patterns and ideas from UML-RSDS have helped to simplify and systematise the transformation specifications.

References

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

[2] J. Foster, M. Greenwald, J. Moore, B. Pierce, A. Schmitt, *Combinators for bi-directional tree transformations*, ACM Trans. Prog. Lang. Sys., 29(3), 2007.

[3] IKV technologies, Medini QVT, projects.ikv.de/qvt/downloads, accessed Dec. 2019.

[4] K. Lano, S. Fang, S. Kolahdouz-Rahimi, *Automated Synthesis of ATL Transformations from Metamodel Correspondences*, Modelsward 2020.

[5] D. Kolovos, R. Paige, F. Polack, *The Epsilon Transformation Language*, ICMT 2008.

[6] K. Lano, *Catalogue of model transformations*, http://nms.kcl.ac.uk/kevin.lano/tcat.pdf, 2005.

[7] K. Lano et al., *A survey of model transformation design patterns in practice*, JSS, 2018.

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

[9] K. Lano, *QVT2UMLRSDS dataset*, doi:10.5281/zenodo.3951061, 2020.

[10] N. Macedo, A. Cunha, *Least-change bidirectional model transformation with QVT-R and ATL*, SoSyM (2016) 15: 783–810.

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

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

Case	*Bidirectionality*	*Batch/Incremental*	*Deterministic*	*Patterns used*
Tree to graph	Bidirectional	Incremental	Yes	Introduce/remove intermediate class; Map objects before links
UML to Python	Bidirectional	Incremental	Yes	Recursive *-accumulation; Auxiliary metamodel; Map objects before links
Hsm2nhsm	Partly separate forward/reverse	Incremental	Yes	Recursive *-accumulation; Entity merging/ splitting (horizontal); Map objects before links
Person migration	Bidirectional	Incremental	Yes	Lens
Weighted/ unweighted Petri nets	Bidirectional	Incremental	Yes	Introduce/remove intermediate classes; Map objects before links
Unordered/ ordered sets	Bidirectional	Incremental	Yes	Map objects before links; Object indexing
Migration of bags	Bidirectional	Batch	Yes	Auxiliary models; Object indexing; Lens; Introduce/ remove intermediate class; Map objects before links
Expression trees/ dags	Mutually inverse forward/reverse	Batch	No	Deletion by selective copy
Gantt2CPM	Bidirectional	Incremental	Yes	Entity merging splitting (horizontal); Introduce/remove intermediate class; Map objects before links

Table 11: Bx properties for cases