Solving the TTC 2014 Movie Database Case with UML-RSDS

K. Lano, S. Yassipour-Tehrani
Dept of Informatics, King's College London

This paper describes a solution to the Movie Database case using UML-RSDS. The solution specification is declarative and logically clear, whilst the implementation (in Java) is of practical efficiency.

1 Solution definition as a UML-RSDS specification

UML-RSDS [1] is a hybrid MT language which uses UML notations to specify transformations: source and target metamodels of a transformation are defined as UML class diagrams, transformations are expressed as use cases, whose effect is specified by a sequence of postconditions written in OCL. This provides an expressiveness similar to other hybrid languages such as GrGen or ETL. The UML-RSDS tools automatically synthesise executable implementations of transformations from the UML specifications.

For the case study specification, we define separate use cases for each task of the case study. Each use case defines a sub-transformation of the problem.

Task 1: Create synthetic datasets We implement this task by a use case *task*1 which has parameter n : *Integer* and a single postcondition

```
Integer.subrange(0,n-1)->forAll( x | Movie.createPositive(x) & Movie.createNegative(x) )
```

where *createPositive* is a static operation of *Movie* which creates the 5 movies, 3 actors and 2 actresses of each positive case, and *createNegative* is a static operation of *Movie* which creates the 5 movies, 2 actors and 3 actresses of each negative case.

createPositive is:

```
createPositive(n : Integer)
pre: n >= 0
post:
  Movie->exists( m1 | m1.rating = 10*n &
    Movie->exists( m2 | m2.rating = 10*n + 1 &
     Movie->exists( m3 | m3.rating = 10*n + 2 &
       Movie->exists( m4 | m4.rating = 10*n + 3 &
         Movie->exists( m5 | m5.rating = 10*n + 4 &
           Movie.createPositiveActors(n,m1,m2,m3,m4,m5) &
           Movie.createPositiveActresses(n,m1,m2,m3,m4,m5) ) ) ) ) )
```

where *createPositiveActors* creates the actors *a*, *b* and *c* and links them to the movies as required, and likewise for *createPositiveActresses*. The definition of *createNegative* is similar.

Task 2: Find couples We implement this task by a use case *task*2 which has a single postcondition:

```
p : Person & q : p.movies.persons & p.name < q.name &
comm = p.movies /\ q.movies & comm.size > 2    =>
    Couple->exists( c |  p : c.p1 & q : c.p2 & c.commonMovies = comm )
```

To appear in EPTCS.

This constraint is implicitly $\forall$-quantified over persons p and q. It creates a couple c for each distinct pair p and q of persons whose set of common movies *comm* has size at least 3. $/\backslash$ denotes intersection, also written as $\cap$. Only one couple is created for each pair because of the restriction that $p1$ always holds the person with the lexicographically smallest name.

The quantifier $q : Person$ can be restricted to $q : p.movies.persons$ because the conditions $comm = p.movies \cap q.movies$ & $comm.size > 2$ imply that $q \in p.movies.persons$ (a case of the Restricting Input Ranges transformation design pattern [2]).

The implementation is a linear iteration through *Person* and its execution time should therefore be of order $Person.size * C$ where C is the maximum size of $movies.persons$. However, efficient computation of set intersections is needed for situations where the sets of common movies become large.

Task 3: Calculate average scores for couples This is implemented by a use case *task*3 with a single postcondition operating on context *Couple*:

```
avgRating = ( commonMovies->collect(rating)->sum() ) / commonMovies.size
```

This iterates over objects *self* of *Couple*, and sets the average rating of each couple equal to the average of the rating of each of their common movies (if two or more movies have the same rating, these ratings are all counted separately in the sum).

Extension task 1: List best 15 couples The set of existing couples can be sorted in different orders using the *sortedBy* operator. For example:

$$Couple \rightarrow sortedBy(-avgRating)$$

is the sequence of couples in order of decreasing *avgRating*.

However this would be very inefficient in this situation, where only the best 15 elements with respect to a given measure are needed, out of possibly millions of elements.

In UML-RSDS it is possible to extend the system library with new functions, which are provided with an implementation by the developer. Here we need a version of *sortedBy* which takes a bound on the number of elements to return: $SortLib.sortByN(s, s \rightarrow collect(e), n)$ returns the best n elements of s according to e, sorted in ascending e-value order. Semantically it is the same as $s \rightarrow sortedBy(e).subrange(1, n)$.

We define an *external* module *SortLib* with *sortByN* as a static operation, and provide (hand-written) Java code for this operation, making use of the existing UML-RSDS merge sort algorithm. The use case then has the postcondition:

```
bestcouples = SortLib.sortByN(Couple.allInstances,
                              Couple->collect(-avgRating), 15)   =>
                                    bestcouples->forAll( c | c->display() )
```

A $toString() : String$ operation is added to *Couple* which returns a display string consisting of the average score, number of movies and persons of each couple. This string is printed to the console by $c \rightarrow display()$.

An example of the output is:

```
Couple avgRating 9992.5, 4 movies (a9993; a9994)
Couple avgRating 9992.0, 3 movies (a9990; a9992)
Couple avgRating 9992.0, 3 movies (a9990; a9993)
Couple avgRating 9992.0, 3 movies (a9990; a9994)
Couple avgRating 9992.0, 3 movies (a9991; a9992)
Couple avgRating 9992.0, 3 movies (a9991; a9993)
```

```
Couple avgRating 9992.0, 3 movies (a9991; a9994)
Couple avgRating 9992.0, 3 movies (a9992; a9993)
Couple avgRating 9992.0, 3 movies (a9992; a9994)
Couple avgRating 9991.5, 4 movies (a9990; a9991)
Couple avgRating 9982.5, 4 movies (a9983; a9984)
Couple avgRating 9982.0, 3 movies (a9980; a9982)
Couple avgRating 9982.0, 3 movies (a9980; a9983)
Couple avgRating 9982.0, 3 movies (a9980; a9984)
Couple avgRating 9982.0, 3 movies (a9981; a9982)
```

for the test case with N = 1000.

Similarly, couples can be displayed in decreasing order of the number of common movies:

```
bestcouples2 = SortLib.sortByN(Couple.allInstances,
                               Couple->collect(-commonMovies.size), 15)   =>
                                       bestcouples2->forAll( c | c->display() )
```

Extension task 2: Generate cliques This use case assumes that task 2 has been completed. A use case *couples*2*cliques* creates a 2-clique for each couple:

```
Clique->exists( c |  c.persons = p1 \/ p2 & c.commonMovies = commonMovies )
```

This constraint has context *Couple* and is applied to each instance *self* of *Couple*.

A use case *nextcliques* generates cliques of size $n+1$ from those of size n:

```
persons@pre.size = n & p : commonMovies@pre.persons &
p.name > persons@pre.name->max() &
comm = p.movies /\ commonMovies@pre & comm.size > 2    =>
          Clique->exists( c |  c.persons = cl.persons@pre->including( p ) &
                               c.commonMovies = comm )
```

This iterates over *Clique*@pre, so that only pre-existing cliques are considered as input to the rule, not cliques generated by the rule. The *nextcliques* implementation is therefore a linear iteration over *Clique*, rather than a fixed-point iteration.

Extension task 3: Calculate average score for cliques This task is implemented by a use case *exttask*3 with a single postcondition operating on context *Clique*:

```
  avgRating = ( commonMovies->collect(rating)->sum() ) / commonMovies.size
```

Extension Task 4: List best 15 cliques As with extension task 1, this task can be achieved using a specialised sorting operator that returns the best 15 cliques according to a valuation expression. Only cliques of a given size n are of interest:

```
ncliques = Clique->select( persons.size = n ) &
bestcliques = SortLib.sortByN(ncliques, ncliques->collect(-avgRating), 15)   =>
                                          bestcliques->forAll( c | c->display() )
```

Similarly for the display of cliques by the number of common movies:

```
ncliques2 = Clique->select( persons.size = n ) &
bestcliques2 = SortLib.sortByN(ncliques2, ncliques2->collect(-commonMovies.size), 15) =>
                             bestcliques2->forAll( c | c->display() )
```

2 Results

To run the use cases for couples from the command line, type

```
java Controller couples N
```

where N is the synthetic data set required (1000, 2000, etc).

Table 1 shows the execution times of the tasks on SHARE for the synthesised data sets, using an unoptimised Java 4 implementation (in which sets are represented as Vectors).

N	*task2*
1000	110ms
2000	162ms
3000	262ms
5000	602ms
10000	670ms

Table 1: Execution times for synthetic data sets (Java 4)

Using the *filter* architectural pattern we could pre-filter the data to reduce input model size by removing all movies with fewer than 2 (fewer than M for M-cliques) cast members, and all people with fewer than 3 movies [2]. This reduces the execution time for *task*2 and extension task 2.

To run the use cases for cliques from the command line, type

```
java Controller cliques N
```

where N is the synthetic data set required (1000, 2000, etc).

Table 2 shows the execution time for extension task 2 for clique sizes from 3 to 5.

N	*exttask2 (3)*	*exttask2 (4)*	*exttask2 (5)*
1000	115ms	114ms	75ms
2000	202ms	261ms	128ms
3000	271ms	274ms	192ms
5000	438ms	423ms	301ms
10000	824ms	996ms	606ms

Table 2: Execution times for clique generation for synthetic data sets, Java 4

The transformation has also been applied to the three IMDb models imdb-0005000-49930, imdb-0010000-98168, imdb-0030000-207420. To apply the transformation to these, invoke it as:

```
java Controller mcouples in1.txt
```

and likewise for in2.txt, in3.txt. Table 3 shows the results.

Data set	*task2*
in1.txt	1864ms
in2.txt	5816ms
in3.txt	More than 120s

Table 3: Execution times for IMDb data sets (Java 4)

To run the use cases for cliques for the IMDb files from the command line, type

```
java Controller mcliques in1.txt
```

This runs task2, couples2cliques, nextcliques (for parameter 2 to generate the cliques of size 3), extension task 3 and extension task 4 (for cliques of size 3). Table 4 shows the results for clique generation.

Model	*exttask2 (3)*
in1.txt	6973ms
in2.txt	11860ms

Table 4: Execution times for 3-clique generation for IMDb data sets, Java 4

The implemented transformation may be obtained at:

`http://www.dcs.kcl.ac.uk/staff/kcl/movies.zip`

It has also been uploaded to the umlrsds TTC14 workspace on SHARE, in the *Public/rsync* directory (remoteUbuntu12LTS_TTC14_umlrsds_new). The execution times in the SHARE environment are slightly lower than those given above. A version using a pre-filter can also be executed, using *FController* instead of *Controller* in the above commands. The filter can substantially reduce the size of the input models by discarding people and movies which cannot contribute to the sets of couples or cliques. This makes the computation of couples for the dataset in3.txt feasible (execution time 4296ms) although the filter takes 45 seconds to execute. Similarly for clique calculation for in3.txt.

References

[1] K. Lano, *The UML-RSDS manual*, www.dcs.kcl.ac.uk/staff/kcl/umlrsds.pdf, 2014.

[2] K. Lano, S. Kolahdouz-Rahimi, *Model transformation design patterns*, IEEE Transactions in Software Engineering, vol. 40, 2014.

[3] T. Horn, C. Krause, M. Tichy, *The TTC 2014 Movie Database Case*, TTC 2014.