Solving the TTC 2014 Movie Database Case with UML-RSDS

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

April 28, 2014

Abstract

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 Case Specification

Figure 1 shows the metamodel and use cases of the case specified in UML-RSDS. For flexibility, we separate each task of the case study in a separate use case. Each use case defines a subtransformation of the problem.

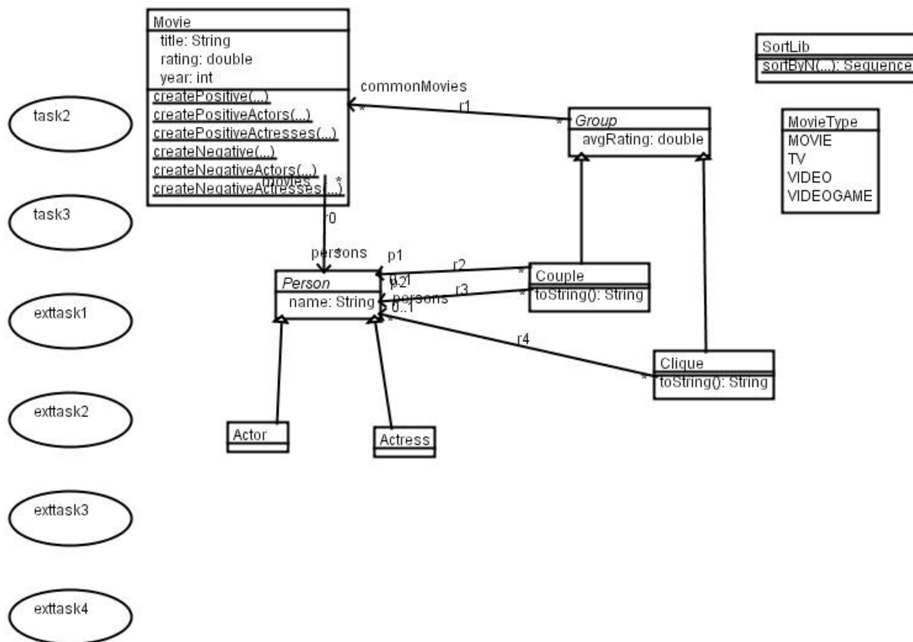

Figure 1: Movie specification in UML-RSDS

Task 1 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 We implement this task by a use case *task*2 which has a single postcondition:

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

This constraint creates a couple c for each distinct pair p and q of persons whose set of common movies *comm* has size at least 3. Only one couple is created for each pair because of the restriction that $p1$ always holds the person with the lexicographically smallest name. The implementation is a double linear iteration through *Person* and its execution time should therefore be of order $Person.size * Person.size$. However, efficient computation of set intersections is needed for situations where the sets of common movies become large.

Task 3 This is implemented by a use case *task*3 with a single postcondition operating on context *Couple*:

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

It 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). Its worst-case execution time is of order $Couple.size * Movie.size$.

Extension task 1 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 (handwritten) Java code for this operation, making use of the existing UML-RSDS merge sort algorithm. Two versions of *SortLib* are needed, one for Java 4 and another for Java 6. 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.

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 This use case assumes that task 2 has been completed. To generate all cliques (of arbitrary size) we first generate those of size 2, based on the couples:

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

This constraint iterates over *Couple* and its time complexity is of order *Couple.size*.

Cliques of size n+1 are then generated from those of size n:

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

This iterates over the set of existing cliques and persons. The logical interpretation is that this constraint holds true at completion of the sub-transformation, ie., that *Clique* is closed wrt the construction step of building a clique of size n+1 from one of size n:

$$\forall\, cl : Clique;\ p : Person;\ comm \cdot p.name > cl.persons.name{\rightarrow}max()\ and\ comm = p.movies \cap cl.commonMovies\ and\ comm.size > 2\ \Rightarrow\ \exists\, c : Clique \cdot c.persons = cl.persons{\rightarrow}including(p)\ and\ c.commonMovies = comm$$

Consideration of the examples suggests that *persons* should be ordered.

A fixed-point implementation of this 2nd constraint will be automatically selected by the UML-RSDS code generator, since *Clique* is both read and written by the constraint.

This implementation terminates, because each application of the rule extracts a new previously undiscovered clique from the data and hence reduces by 1 the number of such undiscovered cliques, but the bound on this number is exponential: $2^{Person.size}$ in the worst case where every person is in every movie.

To improve the efficiency, a size limit on the new clique can be imposed, by only considering cliques smaller than a bound supplied as parameter to the use case. Alternatively, we can split the process into incremental steps. A use case *couples2cliques* creates a 2-clique for each couple:

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

This constraint iterates over *Couple*.

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

```
  persons@pre.size = n & p : Person & 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.

The *nextcliques* implementation is a linear iteration over $Clique * Person$, rather than a fixed-point iteration.

Extension task 3 This 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 As with extension task 1, this 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 display of cliques by 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). This runs task1, task2, task3, extension task 1, and displays the time taken for task2. The generated data is in the file out.txt.

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

N	*task2*	*task3*	*exttask1*
1000	50s	16ms	31ms
2000	200s	31ms	63ms
3000	437s	46ms	78ms
4000	815s	50ms	62ms
5000	1,226s	59ms	103ms

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

The hardware was a single processor Intel i3 Windows 7 laptop, with this process allocated 25% of CPU.

We also implemented the transformation using the Java 6 generator of UML-RSDS. The Java 6 implementation uses HashSet for set-valued associations such as *commonMovies*, so that intersection and other set operations should be more efficient. Surprisingly this implementation was less efficient on the synthetic data (Table 2).

N	*task2*
1000	110s
2000	429s
3000	917s

Table 2: Execution times for synthetic data sets (Java 6)

There does not appear to be any way to further optimise the search for couples using Java. 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 [1], (Figure 2).

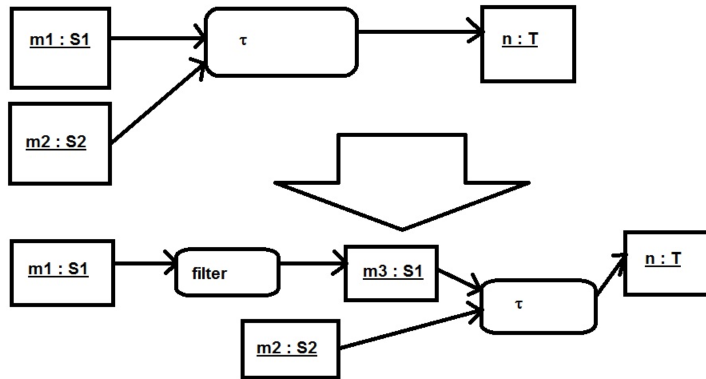

Figure 2: Introducing filtering

We will investigate the efficiency of C# and C++ implementations prior to the TTC workshop.

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). This runs task1, task2, couples2cliques, nextcliques (for parameters 2, 3, 4 to generate the cliques of sizes 3, 4, 5 successively), extension task 3 and extension task 4 (for cliques of sizes 3, 4, 5 separately) and displays the time taken for the exttask2 steps. The generated data is in the file out.txt.

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

N	*exttask2 (3)*	*exttask2 (4)*	*exttask2 (5)*
1000	74s	73s	37s
2000	282s	270s	135s
3000	627s	621s	307s
4000	1098s	1093s	547s

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

computed using an incremental approach: first cliques of size 2 are created from couples, then cliques of size 3 from those of size 2, etc. The execution times are those for the individual steps.

The transformation has also been applied to the three IMDb models imdb-0005000-49930, imdb-0010000-98168, imdb-0030000-207420. We have converted the CSV versions of these models into UML-RSDS model format, in the files in1.txt, in2.txt, in3.txt respectively. To apply the transformation to these, invoke it as:

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

and likewise for in2.txt, in3.txt. This runs task2, task3, extension task 1, and displays the time taken for task2. The generated data is in the file out.txt.

Data set	*task2*
in1.txt	890s
in2.txt	4783s
in3.txt	More than 5000s

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

The converter program is in CSVParser.java, it takes file IMDd.txt as input and produces file in.txt.

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) and displays the time taken for the nextcliques step. The generated data is in the file out.txt.

Model	*exttask2 (3)*
in1.txt	527s
in2.txt	More than 2000s

Table 5: Execution times for 3-clique generation for IMDb data sets, Java 4

3 Optimisation using filter pattern

The idea of this transformation pattern is to filter input models to exclude elements which cannot be matches for the main transformation, therefore reducing model size and the search space of the main transformation.

In this case study, no movie with fewer than 2 cast members can contribute to a couple or clique, nor can any person who is in less than 3 movies. Therefore the filter can be written as:

```
persons.size < 2   =>   self->isDeleted()
```

on Movie, and

```
movies.size < 3   =>   self->isDeleted()
```

on Person. These constraints are mutually dependent (deleting movies may lead to further possible deletions of people, and vice-versa) so are iterated in a fixpoint loop. The execution time is bounded by $Movie.size + Person.size$.

Applied between task1 and task2, this filter removes all the negative case test data from the synthetic models, reducing the model size by 50%. Applied to the IMDb data substantial reductions also occur: the 44931 persons in the first data set are reduced to 2809 by the filter, and the 5000 movies to 1366.

Table 6 shows the results using the filter pattern for the synthetic data sets.

There is therefore an overall reduction in processing time using this approach.

Table 7 shows the optimised results for the IMDb data sets.

Of course, if a direct search for 3-cliques were used, pre-filtering could also exclude movies with fewer than 3 cast members.

The options described above can also be executed with the filter option, by invoking

N	*filter*	*task2*
1000	16s	13s
2000	70s	53s
3000	153s	122s
4000	266s	215s
5000	438s	325s

Table 6: Optimised execution times for synthetic data sets (Java 4)

Data set	*filter*	*task2*
in1.txt	23s	6s
in2.txt	128s	28s
in3.txt	1759s	262s

Table 7: Optimised execution times for IMDb data sets (Java 4)

```
java FController couples N
java FController cliques N
java FController mcouples in1.txt
java FController mcliques in1.txt
```

from the command line.

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 *Desktop/rsync* directory (remoteUbuntu12LTS_TTC14_umlrsds2). The execution times in the SHARE environment are slightly lower than those given above.

References

[1] K. Lano, S. Yassipour-Tehrani, *Model transformation architectural patterns*, Dept of Informatics, King's College London, 2013.

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

Model	*exttask2 (3)*
in1.txt	42s
in2.txt	147s

Table 8: Optimised execution times for 3-clique generation for IMDb data sets, Java 4