The SDMLib solution to the MovieDB case for TTC2014

Christoph Eickhoff[1], Tobias George[1], Stefan Lindel[1], Albert Zündorf[1]

Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
cei|tge|slin|zuendorf@cs.uni-kassel.de

Abstract. This paper describes the SDMLib solution to the MovieDB case for the TTC2014 [4]. We explain a model transformation based solution and a plain Java solution based on a set-based model layer generated by SDMLib. In addition we discuss several refactorings we have used to improve the runtime performance of our solutions. Especially, we show an explicitly parallel solution.

1 Introduction

SDMLib [3] is a light-weight model transformation approach based on graph grammar theory. SDMLib provides a Java API that allows to build a class model and to generate an SDMLib specific Java implementation for it. The generated model classes provide bidirectional association implementations, a reflection layer, and XML and JSON serialization mechanisms. In addition, SDMLib generates a set based layer for the model, where each method provided for a single model object is also provided for a set of such model objects. This is frequently used for model navigation e.g in actor1.getMovies().getPersons(). Here we ask an actor for the set of movies the actor has done and on this set we ask for the set of persons that participated in (at least one of) these movies. Note, SDMLib computes a flat set of persons not a set of sets of persons. Finally, SDMLib generates a pattern matching layer for the model that provides classes to build model specific object patterns and model transformations.

To solve the MovieDB case, we mainly use the set based layer. This enables a very efficient implementation of the clique detection task. However, for completeness, we also provide a solution using SDMLib model transformations.

2 The solution

Recently, we have enabled SDMLib to enhance EMF [2] Java classes with a set based and a model transformation layer. Thus, we might have developed our solution on top of an EMF model implementation. However, we suspect the EMF model implementations to be inefficient in certain cases and thus we decided to use SDMLib's model implementation. Fortunately, SDMLib is able to load an Ecore file and to translate the EMF class model into an SDMLib class model, cf. Figure 1. We have extended the original class model with class `Ranking` used to store the 15 best cliques with respect to average ranking and number of movies.

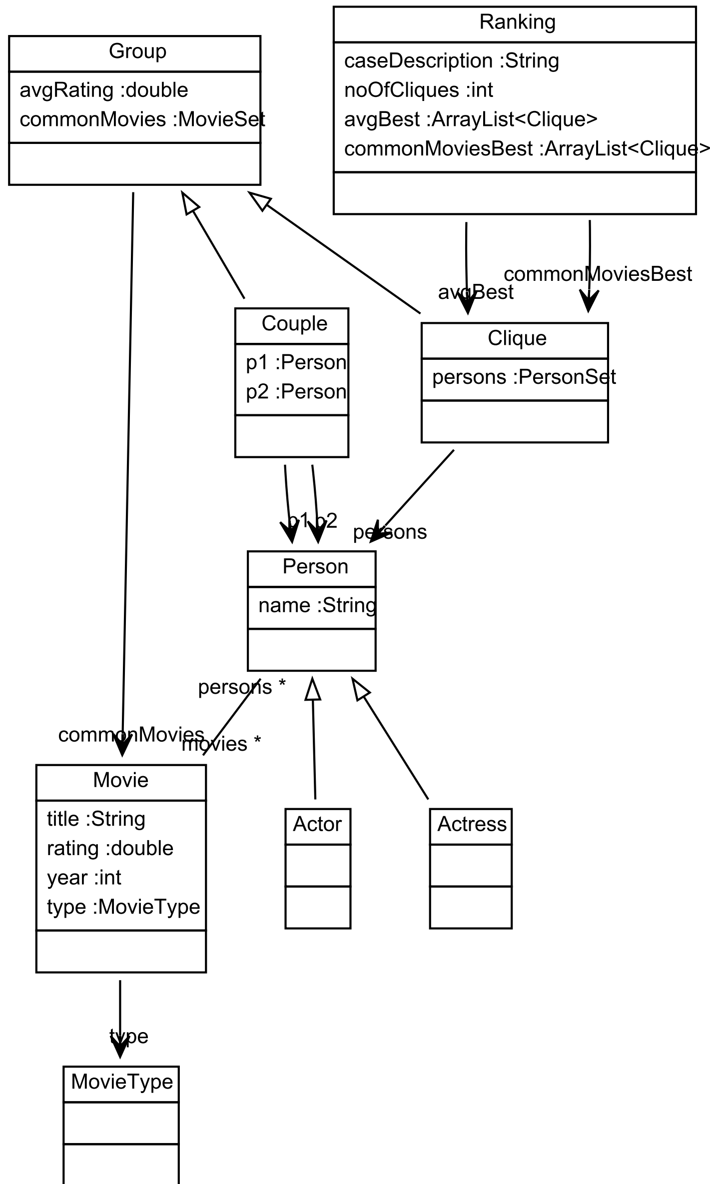

Fig. 1. Class Model imported from Ecore

Figure 2 shows the SDMLib model transformation used to find couples / cliques of two. The search starts with pattern object `p1` that matches to any `Person` in our database. Via `Movie m2` we look for any `Person p3` that has collaborated with `p1`. The first constraint on the right of Figure 2 requires that the name of `p3` is alphabetically later than the name of `p1`. This avoids mirrored couples. Next, the subpattern `o6` searches

for all movies `m7` done by both persons. Each such movie is added to a new `Clique` object `c4`. The second constraint of Figure 2 ensures that at least three movies have been added to our new clique. If this is the case, action `1:` of figure 2 calls method `addToCliques` that stores the clique and maintains ranking tables. Finally, the last action `2:` calls another model transformation `lookForCliques` that looks for larger cliques. (Note, for technical reasons the graphical representation of our model transformation does not show all details of the execution order. Such details are revealed by the Java code that build up the model transformation. This Java code is omitted for lack of space.)

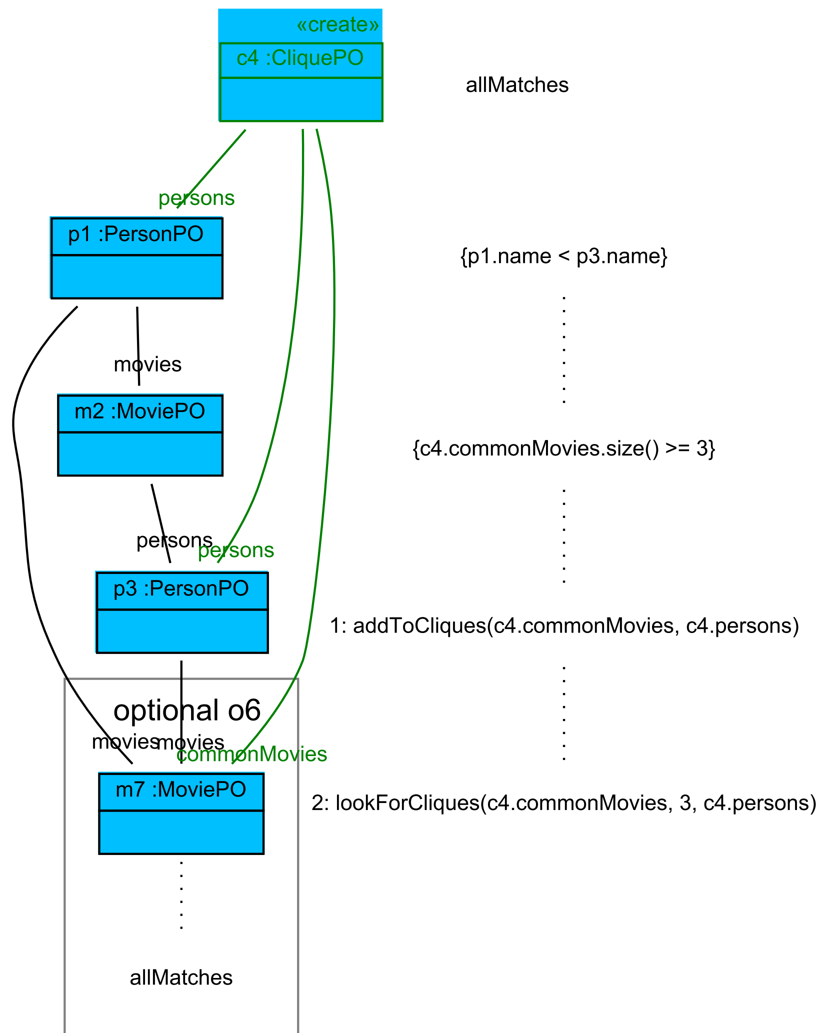

Fig. 2. Look for Couples Model Transformation

The `lookForCliques` model transformation shown in Figure 3 takes a `Clique` c1 and searches through the common movies `m2` for a new `Person` `p3`. An additional constraint ensures that the name of the last person (which is computed separately) in the clique is alphabetically lower than the name of the new person `p3`. Then the subpattern `o6` searches for all movies `m7` that belong to the clique `c1` and to the new person `p3`. The second constraint of Figure 3 ensures that at least three common movies are found. For each match, a new `Clique` object `c4` is created and each common movie `m7` is attached to it. Finally, subpattern `o9` attaches all persons `p10` to the new clique and the new person `p3` is attached, too. Through additional constraints each new clique is added to the rankings (method call `addToCliques`) and we call method `lookForCliques` recursively to find larger cliques. (An additional condition (not shown) terminates this recursion e.g. as soon as cliques of size 5 are reached.)

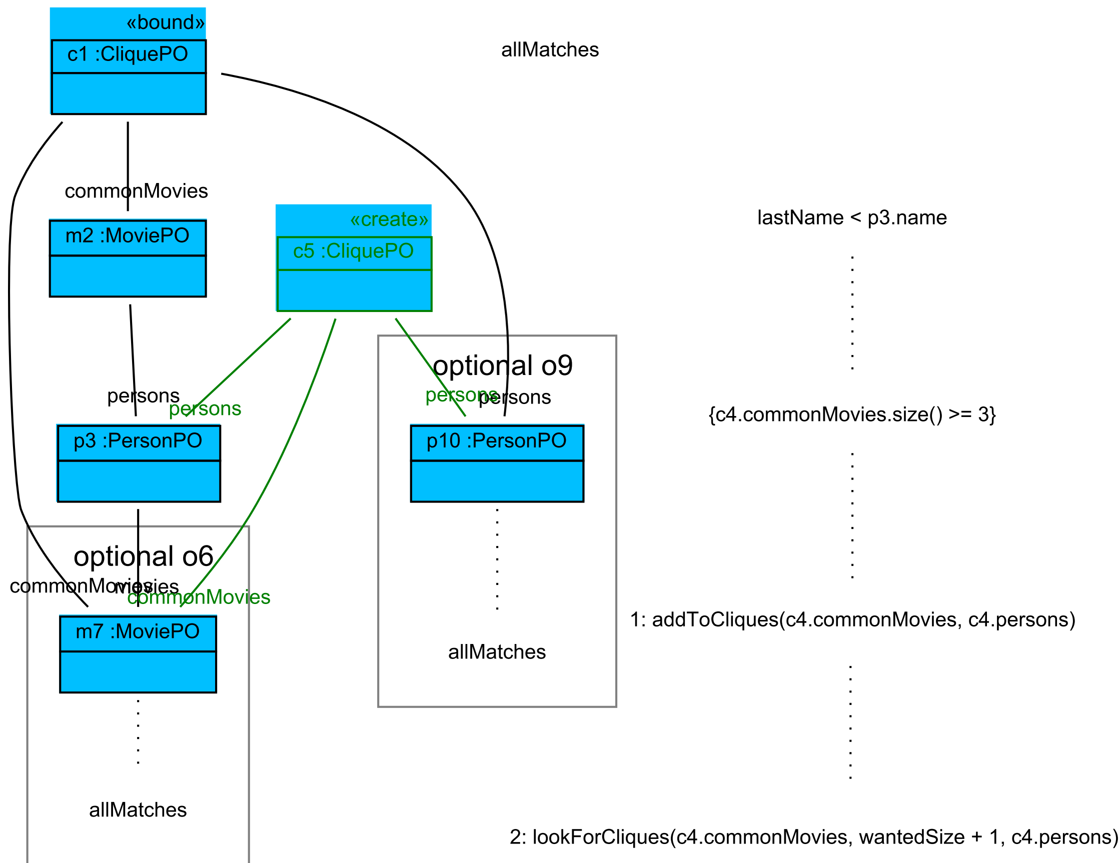

Fig. 3. Look for Cliques Model Transformation

To be honest, the initial versions of our clique finding methods have been built using the set based model layer generated by SDMLib. In Listing 1.1 line 4 we first check whether the wanted clique size is already

reached. Method `lookForCliques` gets a set of common movies and a set of persons from the previous clique as parameter. The idea was to avoid to build clique objects explicitly to save memory. Reading the case description in more detail, we noticed that creating cliques explicitly is mandatory. We do it now in method `addToCliques`. There we use an extra flag to toggle clique creation in order to measure the time overhead this actually causes.

Line 6 of listing 1.1 clones the set of persons passed as parameter and line 7 adds a dummy person to it. Each time we find a new person that forms a clique with the passed persons, we will replace the extra person, cf. line 17. Note, class `PersonSet` is essentially an `java.util.ArrayList`. Originally, SDMLib uses `java.util.LinkedHashSet` as implementation for model sets. However, the benchmarking has shown that this causes a large runtime and memory overhead. Thus for the MovieDB case, we enabled `java.util.ArrayList` based model sets. On insertion, the `ArrayList` based model sets still check whether the set already contains the new element. This is done with linear search and becomes inefficient for large model sets. However in the MovieDB case the sets of movies and the sets of persons we handle contain only some 20 up to to some 100 elements and the search overhead can be neglected. On the other hand, an `ArrayList` consists only of a single object, the array object. While a `LinkedHashSet` employs extra internal `Node` objects for each element you add to the set. In this benchmark the allocation of these internal `Node` objects causes a huge runtime and memory overhead. In the inner loop of listing 1.1 on each iteration we form a new person set consisting of the person set `newClique` created in line 6 and a new person `p`. Ususally, we would call something like `newClique.remove(oldPerson)` and `newClique.add(p)` to replace the last person in `newClique`. By switching to `ArrayLists` and using the `dummyPerson` we achieve the same effect more efficiently by calling `newClique.set(persons.size()-1)`, cf. line 17.

```
1  private void lookForCliques(MovieSet commonMovies, int wantedSize,
2           PersonSet persons)
3  {
4     if (wantedSize <= maxCliqueSize)
5     {
6        PersonSet newClique = (PersonSet) persons.clone();
7        newClique.add(dummyPerson);
8
9        for (Person p : commonMovies.getPersons())
10       {
11          if (persons.get(persons.size()-1).getName().compareTo(p.getName()) < 0)
12          {
13             MovieSet intersection = commonMovies.intersection(p.getMovies());
14
15             if (intersection.size() >= 3)
16             {
17                newClique.set(wantedSize-1, p);
18
19                addToCliques(intersection, newClique);
20
21                // look for larger cliques
22                lookForCliques(commonMovies, wantedSize + 1, newClique);
23             }
24          }
```

```
25          }
26      }
27  }
```

Listing 1.1. Set Base Model Transformation `lookForCliques`

Line 9 loops through the set of all persons that participate in one of the common movies passed as parameter. Note the call to `commonMovies.getPersons()`. Parameter `commonMovies` is of type `MovieSet`. This class is generated by SDMLib as an addition to the model class `Movie`. Class `MovieSet` provides all methods provided by class `Movie` and extends these methods to work on sets of objects. Thus method `MovieSet::getPersons()` calls methods `Movie::getPersons()` on each element of `commonMovies`. Method `Movie::getPersons()` has return type `PersonSet`, i.e. the set of persons working on a given movie. Method `MovieSet::getPersons()` collects these `PersonSet`s within a (flat) `result` set using a `result.union(newSet)` operation. In our method `lookForCliques` this this set based `getPersons` operation saves us an explicit outer loop through the `commonMovies` set and we do not need an extra data structure to keep track of already handled persons.

Similarly, line 13 uses the set based method `intersection` to compute the set of common movies from the parameter `commonMovies` and the movies of the current person `p`.

The if statement in line 11 ensures that we consider only persons with a name later than the name of the last person in `newClique`. This avoids multiple cliques of the same persons that differ only in the ordering. The if statement in line 15 ensures that the `intersection` of movies has at least 3 entries. Thus, when we reach line 17 we have found a new clique and line 19 adds this new clique to the rankings and line 22 tries to extend the new clique recursively.

3 Performance

The first version of our solution used the SDMLib generated model implementation, the set based model layer, and plain Java code as outlined in listing 1.1. In that version we did not create all found cliques explicitly but we only collected the 15 best cliques for each ranking. Without further optimizations the 20 000 synthetic MovieDB case needed about 50 seconds on a 2.67 GHz Intel i7 dual core (M60) 64 bit CPU (with hyper threading) and 8 GB main memory running windows 7. We call this our reference laptop from now on. Actually, first measurements with different case sizes for the synthetic MovieDB produced strange results where e.g the 10 000 case was much slower then the 20 000 case. We figured out that the Java virtual machine hot compile has a strong influence on our measurements. Hot compile causes up to 10 times speed-ups. Thus we added a warm up phase to our benchmark where we run a large synthetic case just to trigger the hot compile.

Then we replaced the `java.util.LinkedHashSet` implementation used for `Clique`s to store sets of common movies and sets of persons by an `java.util.ArrayList` based implementation. Our `ArrayList` based implementation still ensured set semantics, i.e. before adding e.g. a new `Person` object, it checks whether this object is already contained. This change also affected a dummy `Clique` object used to store the set of all movies and all persons of the current movie database and as start for the clique detection. If you anticipate the size, an `ArrayList` is allocated in a single heap operation and uses very little memory overhead. Contrarily, a `LinkedHashSet` internally uses one array to hold the hash table and additional `Entry` objects for each model object added to it. These `Entry` objects server as placeholders within the hash tables buckets. Thus for n model elements a `LinkedHashSet` allocates n `Entry` objects plus one array for the hash table itself. Especially the allocation of the `Entry` objects proofed as very time consuming. Avoiding this for the `Clique`s resulted in a speed-up of factor 5.

Next, the call for solutions states that the benchmark shall be done on workstation with an 8 core CPU. Thus we redesigned our solution to run in multiple threads, cf. listing 1.2. Line 2 creates a pool of threads to run multiple tasks in parallel, one for each CPU core. Next, line 4 splits the `caseSize` into multiple chunks, one for each thread. The loop of line 8 creates `CliqueTask` objects for each chunk in line 11. Each `CliqueTask` gets the list of all `persons`, the start index `i` of its chunk and the `chunkSize` as parameter (plus some boolean flags discussed later). `CliqueTask` implements `java.util.concurrent.Callable` and thus line 13 can submit the `CliqueTask` objects to our thread pool for excecution. The submit operation returns a `Future` object that is used in line 26 to retrieve the `Ranking` for the considered chunk. Note, the operation `future.get` is blocked until the corresponding thread / `CliqueTask` has returned a result. For each clique size (from 2 to 5) a `Ranking` contains a list of the 15 best ranked cliques plus a list for the 15 cliques with the largest number of common movies plus the total number of cliques of that size for that chunk. Thus the loop from line 28 to line 36 collects the ranking data from each chunk for each clique size. Later on, the resulting overall `cliqueTabList` elements are sorted again and the overall 15 best cliques are reported.

On our dual core reference laptop this created a speed-up of roughly factor 2. We have also tested it on a 12 core workstation where we achieved a speed-up of factor 10. Thus, for the MovieDB, the explicit parallelization actually gives you a speed-up corresponding to the number of CPU cores you have, i.e. the case is easy to parallelize. Thinking about it, our couple and clique finding always starts with a first person and we than we (recursively) add more persons to this nucleus. Obviously, this can be done for each person in parallel without interference. We just do it in larger chunks to minimize the overhead caused by combining the computation results. Note, for the real data cases, different chunks of persons may produce very different numbers of cliques. Some persons have done only one movie and do not create cliques. Some persons have a large number of movies and produce a large number of cliques. Thus, different chunks need a very different amount of time to complete. To address this, for the real data cases we use 10 chunks per available processor. Our thread pool then schedules the `CliqueTask`s to the available threads as soon as a task has completed and thus the work load of all threads is balanced.

```
1   ...
2   ExecutorService executor = Executors.newFixedThreadPool(processors);
3
4   int chunkSize = caseSize / processors;
5   ...
6   System.out.print("\n[");
7
8   for (int i = 0; i < caseSize; i += chunkSize)
9   {
10     System.out.print("␣");
11     CliqueTask cliqueTask = new CliqueTask(persons, i, chunkSize, ...);
12
13     Future<ArrayList<Ranking>> future = executor.submit(cliqueTask);
14
15     futures.add(future);
16  }
17
18  System.out.print("]\n␣");
19
20  ArrayList<Ranking> cliqueTabList = ...;
```

```
21
22 for (Future<ArrayList<Ranking>> future : futures)
23 {
24    try
25    {
26       ArrayList<Ranking> results = future.get();
27
28       for (int i = 0; i < maxCliqueSize − 1; i++)
29       {
30          Ranking ranking = cliqueTabList.get(i);
31          Ranking partialResult = results.get(i);
32          ranking.getAvgBest().addAll(partialResult.getAvgBest());
33          ranking.getCommonMoviesBest().addAll(partialResult.getCommonMoviesBest());
34          ranking.setNoOfCliques(
35             ranking.getNoOfCliques() + partialResult.getNoOfCliques());
36       }
37    } catch (InterruptedException | ExecutionException e) { e.printStackTrace();}
38 }
39 executor.shutdown();
40 ...
```

Listing 1.2. Parallel execution of clique detection tasks.

With this approach we achieved an execution time of 12,263 seconds for the N=200 000 synthetic case using only one core and 5,695 seconds using both cores of our reference laptop, cf. row one of table 1. Next, we enhanced our model implementation by replacing the `LinkedHashSet` based implementation for `persons` in class `Movie` by an `ArrayList` based implementation of class `PersonSet`. This saved about 3.3 seconds on single core, cf. row two of table table 1.

In the synthetic case movies are generated with ascending rankings. Thus looping through the persons in order of their creation results in cliques with an ascending order of average ranking. Thus, when we maintain the list of the 15 best ranked cliques, we constantly replace old entries with higher ranked new entries. To avoid this, we just visit the persons in reverse order. This saves again 2.4 seconds on our reference laptop. Well, to some extend this is cheating as this trick will not show an improvement on the real data.

Next we changed the implementation of class `MovieSet` used in class `Person` to store a person's movies to use `ArrayList`. At the same time we learned from a conversation with the organizer that the call for solutions requires to create all cliques explicitly. Until now, we just used parameters of type MovieSet and PersonSet in our recursive `lookForCliques` method. To meet the case requirement, we just extended method `addToCliques` to create an explicit clique object and to add the `commonMovies` and the `persons` to it. In row 4 of table 1 column `manual` now shows single thread execution time with explicit clique creation, column `parallel` shows two thread execution time with explicit clique creation, and column `no create` shows two thread execution time *without* clique creation. Actually, explicit clique creation needs about 0.5 seconds for two threads and thus probably about 1 second on a single thread. Still this is outperformed by the change to the the `MovieSet` implementation and we now need about 5 seconds to detect all couples and all cliques in a single thread for the N=200 000 synthetic case.

solution feature	trafo (sec)	manual (sec)	parallel (sec)	no create (sec)
Introduced ArrayList for cliques		12,263	5,695	-
Changed PersonSet to ArrayList<Person>		8,897	4,641	-
Looping through persons in reverse order		6,461	3,043	-
Changed MovieSet to Array List, creating Cliques explicit		4,740	2,379	1,919
Added trafo, improved it by factor 5	213,250	5,723	2,795	2,330
Caching trafos	74,596	4,697	2,247	1,858

Table 1. Evaluation results

At this point in time, we added the model transformation based solution to the clique detection mechanism as discussed in section 2. Initially, the trafo solution already took some 200 seconds for the N=20 000 case. We identified that the SDMLib model transformation mechanism did a lot of copying of candidate sets during search. By removing many of these copies and by using `ArrayList` where possible we achieved a speed-up of about factor 6 resulting in the times reported in row 5 of table 1. Thus, the improved model transformation used 213 seconds for the N=200 000 synthetic case. Unhappy with this execution time, we identified that the `lookForCliques` transformation is called recursively some million times and that we construct the object structure that represents the model transformation each time anew. Thus, we added a cache for the object structure that represents the model transformation and just reinitialized it to start the pattern matching from a new clique each time. This reduced the execution time to some 75 seconds, cf. last row of table 1.

Overall, the transformation based solution is still 15 times slower than the set based solution. Some fraction of this overhead is surely caused by the interpreter that runs through the object structure representing a model transformation and executes it. However, at last years transformation tool contest this interpretation overhead seemed not that much a problem. Actually, we have already spotted some other inefficient heap operations within the interpreter. We work on more improvements on that.

4 Conclusions

Our first approach to attack the MovieDB case was a manually written Java method exploiting the model implementation generated by SDMLib and especially exploiting the generated set-based model layer as shown in listing 1.1. Coming up with this solution was quite straight forward and we think it is reasonably concise and it seems to be reasonably efficient.

For comparison, we also developed a model transformation based approach. While the graphical representation of the model transformations in figure 2 and figure 3 is reasonably understandable (at least if you have developed them yourself :), the Java code that creates the object structure that represents the model transformations is about double the size of the set-based solution. In addition, the Java code is not as comprehensible as the set-based code. And finally, the model transformation based solution is slower by a factor of 15.

During the development of SDMLib and during teaching it, we recognized that in many cases the set-based model layer suffices to program a certain model operation. Actually, it was pretty tricky to come up with a real world example where the set-based layer did not suffice and a model transformation was more handy. While such cases exist and thus model transformations are useful, the MovieDB case does is more easily addressed by a set-based solution.

Note, the set-based model layer generated by SDMLib compares to simple OCL expressions [1]. Thus, a comparable solution might have been created using EMF and OCL. Next, before this benchmark the model

layer generated by SDMLib relied on `LinkedHashSet`s for the implementation of to-many associations. We thought that `LinkedHashSet` is the most efficient data structure for to-many associations as each add operation requires to check whether the element is already contained. This avoids that an element is contained in the set of neighbors multiple times. This especially was a distinction from EMF based models that use `EList`s to implement to-many associations which finally compares to an `ArrayList`. In this benchmark we followed the advice of EMF and used an `ArrayList` based solution, too. Actually, this is more efficient as long as the sets are reasonable small (some hundred to some 1000 elements). When we used an `ArrayList` based `PersonSet` (guaranteeing the uniqueness of contained elements) for the root clique of the MovieDB case that contains all movies and all persons, the `ArrayList` performance caved in. Actually, the check for containment is not necessary while creating the synthetic cases or reading the real case files. Thus, the choice of the right data structure heavily depends on the situation and it may even change during execution time (initially a lot of add operations, then only reads). For SDMLib we will soon provide an option to enable the user to choose the data structure that fits the user's purposes most.

Please find some clique ranking reports in the appendix A.

References

1. O. M. G. (OMG). Object constraint language (ocl). version 2.3.1, 2012.
2. Eclipse Modeling Framework. http://sdmlib.org/, 2014.
3. Story Driven Modeling Library. https://www.eclipse.org/modeling/emf/, 2014.
4. Movie Database Case for the TTC 2014. https://github.com/ckrause/ttc2014-imdb, 2014.