The SDMLib solution to the MovieDB case for TTC2014

Christoph Eickhoff, Tobias George, Stefan Lindel, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

`cei|tge|slin|zuendorf@cs.uni-kassel.de`

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.

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. 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

SDMLib is able to load an Ecore file and to translate the EMF class model into an SDMLib class model and to generate an efficient Java implementation. 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.

Figure 1 shows the SDMLib model transformation used to find 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 1 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 1 ensures that at least three movies have been added to our new clique. If this is the case, action `1:` of Figure 1 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, 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.)

Submitted to:
TTC 2014

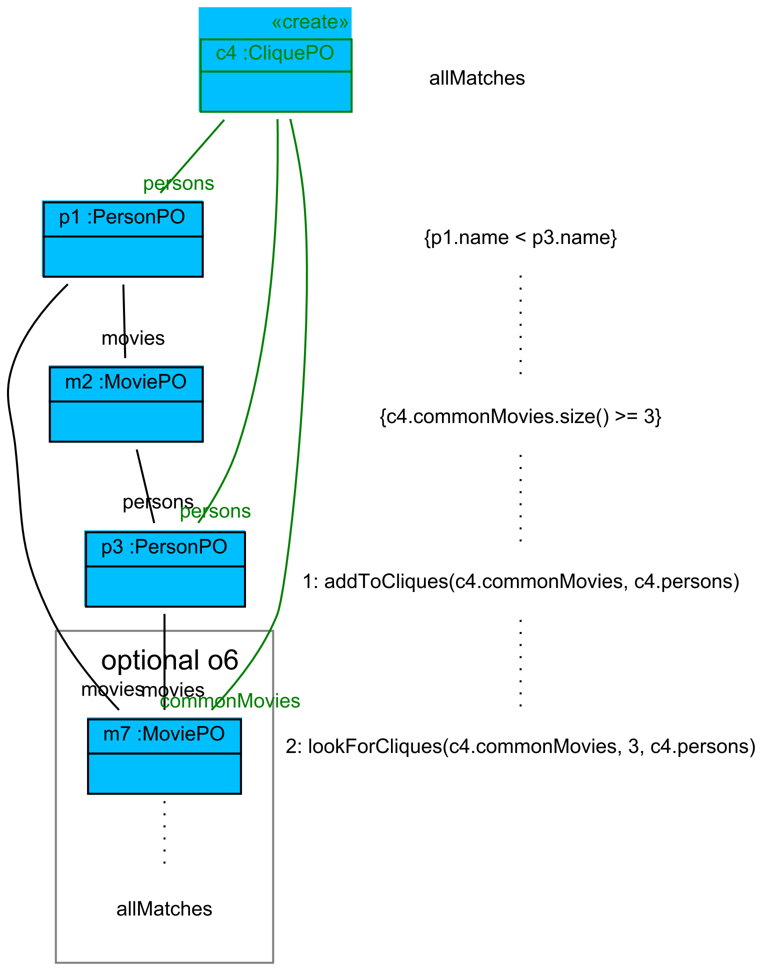

Figure 1: Look for Couples Model Transformation

The `lookForCliques` model transformation shown in Figure 2 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 2 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.)

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 line 3 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.

```
private void lookForCliques(MovieSet commonMovies, int wantedSize,
        PersonSet persons) {
 if (wantedSize <= maxCliqueSize)       {
    PersonSet newClique = (PersonSet) persons.clone();
    newClique.add(dummyPerson);
    for (Person p : commonMovies.getPersons()) {
       if (persons.get(persons.size()-1).getName().compareTo(p.getName())<0){
          MovieSet intersection = commonMovies.intersection(p.getMovies());
          if (intersection.size() >= 3) {
             newClique.set(wantedSize-1, p);
             addToCliques(intersection, newClique);
             // look for larger cliques
13                  lookForCliques(commonMovies, wantedSize + 1, newClique);
14  }  }  }  }
```

Listing 1: Set Base Model Transformation `lookForCliques`

Line 6 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 `PersonSets` within a (flat) `result` set using a `result.union(newSet)` operation. In our method `lookForCliques` 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 8 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 7 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 9 ensures that the `intersection` of movies has at least 3 entries. Thus, when we reach line 10 we have found a new clique and line 11 adds this new clique to the rankings and line 13 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. 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 `Cliques` 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. As this benchmark uses many small sets of objects, using ArrayLists 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. 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. With the parallelization 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.

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

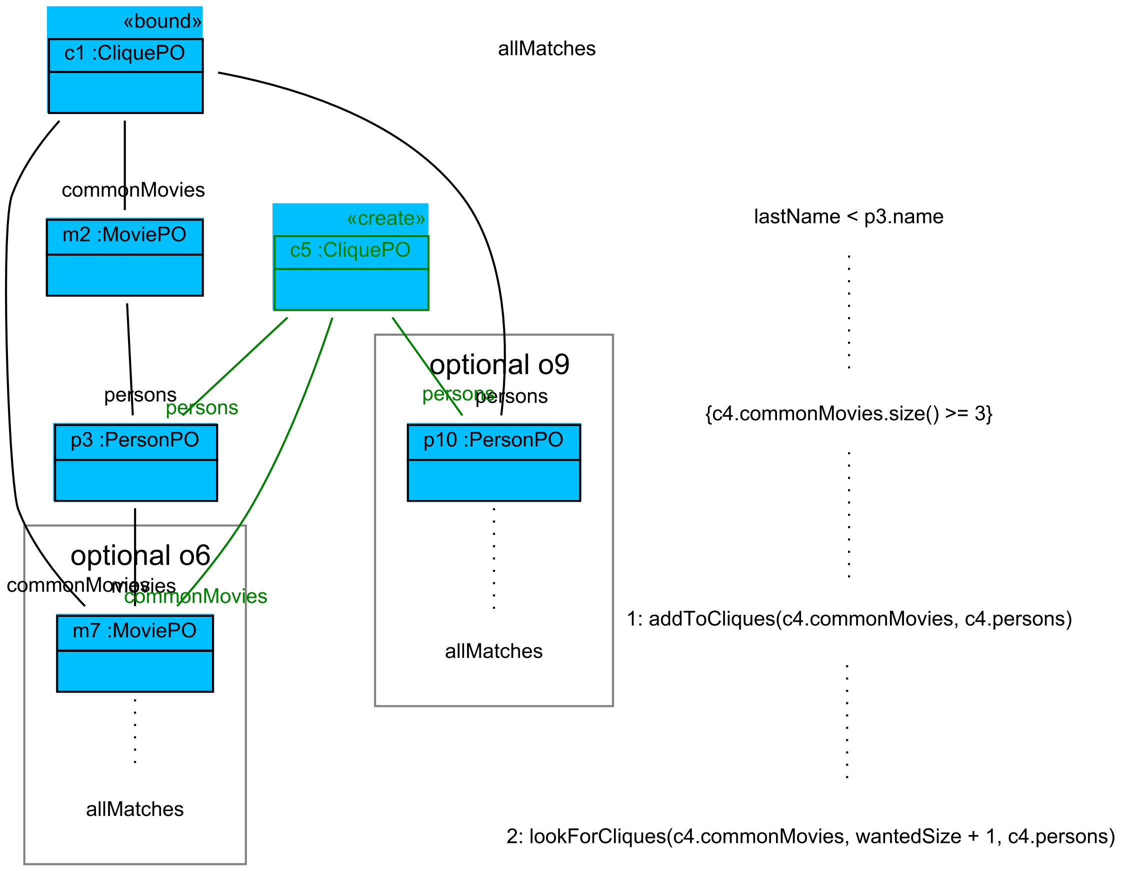

Figure 2: Look for Cliques Model Transformation

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 learned from a conversation with the organizer that the call for solutions requires to create all cliques explicitly. Actually, explicit clique creation needs about 0.5 seconds for two threads and thus probably about 1 second on a single thread. Finally, we 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		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. Actually, we have already spotted some other inefficient heap operations within our 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. 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 1 and figure 2 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. 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 `LinkedHashSets` for the implementation of to-many associations. This especially was a distinction from EMF based models that use `ELists` 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.

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.