Solving the TTC Movie Database Case with FunnyQT

Tassilo Horn
horn@uni-koblenz.de

April 15, 2014

Abstract

FunnyQT is a model querying and model transformation library for the functional Lisp-dialect Clojure providing a rich and efficient querying and transformation API. This paper describes the FunnyQT solution to the TTC 2014 Movie Database transformation case. All core tasks and all extension tasks have been solved.

1 Introduction

This paper describes a solution of the TTC 2014 Movie Database Case [HKT14]. All core and extension tasks have been solved, and the solution also scales well with large models. The solution project is available on Github[1], and it is set up for easy reproduction on the SHARE[2] image `Ubuntu12LTS_TTC14_64bit_FunnyQT4.vdi`.

The solution is implemented using FunnyQT [Hor13] which is a model querying and transformation library for the functional Lisp dialect Clojure[3]. Queries and transformations are plain Clojure programs using the features provided by the FunnyQT API. This API is structured into several task-specific sub-APIs/namespaces, e.g., there is a namespace *funnyqt.in-place* containing constructs for writing in-place transformations, a namespace *funnyqt.model2model* containing constructs for model-to-model transformations, a namespace *funnyqt.bidi* containing constructs for bidirectional transformations, and so forth.

As a Lisp dialect, Clojure provides strong metaprogramming capabilities that are exploited by FunnyQT in order to define several *embedded domain-specific languages* (DSL, [Fow10]) for different tasks. For example, the pattern matching, in-place transformation, model-to-model transformation, and bidirectional model transformation constructs are provided in terms of a small, task-oriented DSL each.

FunnyQT currently supports querying and transforming EMF [SBPM08] and JGraLab[4] TGraph models, and support for other modeling frameworks can be added without touching FunnyQT's internals. For both EMF and JGraLab, there is one FunnyQT core namespace (i.e., *funnyqt.emf* and *funnyqt.tg*) providing functions for accessing and manipulation models of that kind using the framework's terminology and giving access to any feature provided by that framework. These core namespaces are complemented by a namespace *funnyqt.generic* which provides functions for functionality available on all frameworks in a generic, framework-agnostic manner.

[1]`https://github.com/tsdh/ttc14-movie-couples`
[2]`http://is.ieis.tue.nl/staff/pvgorp/share/`
[3]`http://clojure.org`
[4]`http://jgralab.uni-koblenz.de`

2 Solution Description

In this section, the complete transformation and query specifications for all core and extension tasks are going to be explained. In the listings given in the following, all function calls are shown in a namespace-qualified form to make it explicit in which Clojure or FunnyQT namespace those functions are defined. Clojure allows to define short aliases for used namespaces in order to allow qualification while still being concise. Table 1 gives an overview of all namespaces used by the solution, and the aliases used for accessing them.

Alias	Namespace	Description
emf	funnyqt.emf	Core EMF API
gen	funnyqt.generic	Generic model access functions
io	clojure.java.io	File IO functions
ip	funnyqt.in-place	In-place transformation API
poly	funnyqt.polyfns	Polymorphic function API
set	clojure.set	Set functions (e.g., union, intersection,...)
str	clojure.string	String utility functions
u	funnyqt.utils	Utility functions (e.g., error handling)

Table 1: Used Clojure and FunnyQT namespaces with their aliases

All function calls that are not qualified with an alias are calls to functions in the *clojure.core* namespace which is available in any other namespace by default.

2.1 Task 1: Generating Test Data

The first task is generating test data. The case description [HKT14] illustrates the task with Henshin rules. Since the rules actually don't match anything but simply create new elements in the model, we have implemented them as plain functions receiving the model and an integer parameter `i` from which the movie ratings and actor names are derived.

The function `create-positive!`[5] creates five movies, three actors, and two actresses. The persons' movies references are set as requested by the case description, i.e., every persons acts in the second, third, and fourth movie, and the first two persons additionally act in the first movie while the last two persons additionally act in the fifth movie.

Forward-looking to task 2, every invocation of the `create-positive!` function creates $\binom{5}{2} = 10$ couples with three common movies.

```
(defn ^:private create-positive! [model i]
  (let [m1 (emf/ecreate! model 'Movie :rating (+ 0.0 (* 10 i)))
        m2 (emf/ecreate! model 'Movie :rating (+ 1.0 (* 10 i)))
        m3 (emf/ecreate! model 'Movie :rating (+ 2.0 (* 10 i)))
        m4 (emf/ecreate! model 'Movie :rating (+ 3.0 (* 10 i)))
        m5 (emf/ecreate! model 'Movie :rating (+ 4.0 (* 10 i)))]
    (emf/ecreate! model 'Actor   :name (str "a" (* 10 i))       :movies [m1 m2 m3 m4])
    (emf/ecreate! model 'Actor   :name (str "a" (+ 1 (* 10 i))) :movies [m1 m2 m3 m4])
    (emf/ecreate! model 'Actor   :name (str "a" (+ 2 (* 10 i))) :movies [m2 m3 m4])
    (emf/ecreate! model 'Actress :name (str "a" (+ 3 (* 10 i))) :movies [m2 m3 m4 m5])
    (emf/ecreate! model 'Actress :name (str "a" (+ 4 (* 10 i))) :movies [m2 m3 m4 m5])))
```

The `create-negative!` function is defined similarly. It also creates five movies and five persons assigning ratings for the former and names for the latter. However, every pair of two of those

[5]The `^:private` is a metadata annotation declaring that this function is private, i.e., it is not accessible from another namespace.

persons have acted together in at most two movies, so there is no couple to be found in task 2 here.

```
(defn ^:private create-negative! [model i]
  (let [m1 (emf/ecreate! model 'Movie :rating (+ 5.0 (* 10 i)))
        m2 (emf/ecreate! model 'Movie :rating (+ 6.0 (* 10 i)))
        m3 (emf/ecreate! model 'Movie :rating (+ 7.0 (* 10 i)))
        m4 (emf/ecreate! model 'Movie :rating (+ 8.0 (* 10 i)))
        m5 (emf/ecreate! model 'Movie :rating (+ 9.0 (* 10 i)))]
    (emf/ecreate! model 'Actor   :name (str "a" (+ 5 (* 10 i))) :movies [m1 m2])
    (emf/ecreate! model 'Actor   :name (str "a" (+ 6 (* 10 i))) :movies [m1 m2 m3])
    (emf/ecreate! model 'Actress :name (str "a" (+ 7 (* 10 i))) :movies [m2 m3 m4])
    (emf/ecreate! model 'Actress :name (str "a" (+ 8 (* 10 i))) :movies [m3 m4 m5])
    (emf/ecreate! model 'Actress :name (str "a" (+ 9 (* 10 i))) :movies [m4 m5])))
```

The last function `create-example` is the entry point to the generation of synthetic test models. It receives an integer parameter `n`, creates a new model (a resource in EMF parlance), and then invokes the `create-positive!` and `create-negative!` rules `n` times with the new model and values for `i` from 0 to `n-1`. Finally, the new model populated with the movies and actors is returned.

```
(defn create-example [n]
  (let [model (emf/new-resource)]
    (dotimes [i n]
      (create-positive! model i)
      (create-negative! model i))
    model))
```

The `create-example` function is invoked by the benchmark runner with the `n`-values 1000, 2000, 3000, 4000, 5000, 10000, 50000, 100000, and 200000.

2.2 Task 2 & 3 and Extension Task 2 & 3: Finding Couples/Cliques & Compute Average Rankings

As the case description already mentions, the *Finding Cliques* task is a generalization of the *Finding Couples* task, so the FunnyQT solution solves both one go. For finding cliques of arbitrary sizes $n \geq 3$, a higher-order transformation should be defined that creates a transformation rule for that n. The FunnyQT solution also allows for $n = 2$ and deals with the fact that in this case, `Couple` elements should be created rather than `Clique` elements.

Also, the *computation of the average rankings* of a couple's or clique's common movies is done while creating the `Couple` or `Clique` element instead of doing it separately in a further step.

The higher-order transformation generating a FunnyQT in-place transformation rule for a given $n \geq 2$ is a Clojure *macro*. A macro is a function which is executed at compile-time by the Clojure compiler. It receives code passed to it as arguments, processes it, and returns new code that takes the place of it's call. This new code is called the macro's *expansion*. Because like all Lisps, Clojure is *homoiconic*, i.e., Clojure code is represented using Clojure datastructures[6] (literals, symbols, lists, vectors), a macro is essentially a transformation on the abstract syntax tree of the Clojure code that's passed to the macro.

Before discussing the rule generation macro, a few helper functions are going to be discussed. Those will be used as constraints in the patterns of the generated rules.

```
(defn movie-count [p]
  (.size ^java.util.Collection (emf/eget-raw p :movies)))

(defn person-count [m]
  (.size ^java.util.Collection (emf/eget-raw m :persons)))

(defn movie-set [p]
  (into #{} (emf/eget-raw p :movies)))
```

[6]In other words, Lisp and Clojure do not separate concrete and abstract syntax.

The function `movie-count` gets some `Person` element `p` and returns the number of movies that this person has acted in. `emf/eget-raw` is a function that returns the value of some element's property without any conversion[7], e.g., here an EMF `EList` object is returned and its `size()` method (defined in the EMF Java API) is invoked. The syntax `^java.util.Collection` is a type hint which allows the Clojure compiler to generate byte-code for a direct rather than a reflective Java method call (`EList` implements the `Collection` interface which declares the method `size()`).

Similarly, `person-count` returns the number of persons that acted in a given movie `m`.

Lastly, `movie-set` gets a person element `p` and returns the movies that person acted in as a set. `#{}` is the empty Clojure set literal.

The `avg-rating` function shown in the next listing gets a collection of movie elements and returns their average rating. The higher-order function `map` takes a function and a collection and applies the function to each element of the collection returning the sequence of results, which is the sequence of the given movies ratings here. `reduce` then aggregates the sequence using `+`, i.e., it computes the sum of all ratings. Finally, this sum is divided by the number of given movies.

```
(defn avg-rating [movies]
  (/ (reduce + (map #(emf/eget % :rating) movies))
     (count movies)))
```

The function `n-common-movies?` printed in the next listing gets an integer `n`, a person element `p`, and additional person elements `more`[8]. If all given persons act together in at least `n` movies, the set of common movies is returned. Otherwise, `nil` is returned. Since in Clojure the values `nil` and `false` are falsy while every other value is truthy, this function can act as a predicate and still return more information, i.e., the common movies, in the positive case.

```
(defn n-common-movies? [n p & more]
  (loop [common (movie-set p), more more]
    (when (>= (count common) n)
      (if (seq more)
        (recur (set/intersection common (movie-set (first more)))
               (rest more))
        common))))
```

The next listings shows the `define-groups-rule` macro which is the higher-order transformation solving the task. It receives an parameter `n` and, as its name suggests, expands into a rule for finding couples (if `n` equals 2) or cliques of size `n`.

```
(defmacro define-group-rule [n]
  (let [psyms (map #(symbol (str "p" %)) (range n))]
    `(ip/defrule ~(symbol (str "make-groups-of-" n "!"))
       {:forall true :no-result-vec true}
       [~'model ~'c]
       [~'m<Movie>
        :when (>= (person-count ~'m) ~n)
        ~@(mapcat (fn [i]
                    (let [ps (nth psyms i)]
                      `[~'m -<persons>-> ~ps
                        :when (>= (movie-count ~ps) ~'c)
                        ~@(when-not (zero? i)
                             `[:when (neg? (compare (emf/eget-raw ~(nth psyms (dec i)) :name)
                                                    (emf/eget-raw ~ps :name)))])
                        ~@(when-not (or (zero? i) (= i (dec n)))
                             `[:when (n-common-movies? ~'c ~@(take (inc i) psyms))])]))
                  (range n))
        :when-let [~'cms (n-common-movies? ~'c ~@psyms)]
        :as [~'cms ~@psyms]
        :distinct]
```

[7]The usual FunnyQT EMF property getter `emf/eget` coerces Java collections to immutable, persistent Clojure collections. Since we are only interested in the size, this coercion would be superfluous overhead.

[8]The Clojure varargs syntax `& els` is similar to Java's `Type... els` syntax.

```
39       (emf/ecreate! ~'model ~@(if (= n 2)
40                                 `['Couple :p1 ~(first psyms) :p2 ~(second psyms)]
41                                 `['Clique :persons [~@psyms]])
42                     :commonMovies ~'cms :avgRating (avg-rating ~'cms)))))
```

We're not going to discuss the macro in details, however the central idea of the Clojure (or Lisp) macrosystem is that one defines the basic structure of the macro's expansion using a *quasi-quoted* (backticked) form as a kind of template. In this quasi-quoted form, values computed at compile-time can be inserted using the *unquote* (`~`) and *unquote-splicing* (`~@`) operators to fill the template's variable parts. What the macro above generates is a `(ip/defrule ...)` FunnyQT in-place transformation rule definition where the rule's name is `make-groups-of-<n>!` (line 22), where two special rule options are set (line 23), which has two parameters `model` and `c` (line 24), a complex pattern involving constraints using the `person-count`, `movie-count`, and `n-common-movies?` functions defined above (lines 25-39), and which finally contains an action to be applied to found matches that creates either a `Couple` or a `Clique` and sets the common movies and average rating (lines 40-43).

The last part of the implementation of the tasks 2 and 3 and the extension tasks 2 and 3 is to actually invoke the macro to create the transformation rules for couples and cliques of 3, 4, and 5 persons.

```
43 (define-group-rule 2) ;; make-groups-of-2!: The Couples rule
44 (define-group-rule 3) ;; make-groups-of-3!: The Cliques of Three rule
45 (define-group-rule 4) ;; make-groups-of-4!: The Cliques of Four rule
46 (define-group-rule 5) ;; make-groups-of-5!: The Cliques of Five rule
```

Instead of discussing the rule generation macro in details, it makes more sense to have an in-depth look at one of its expansion like the one for `n` being 3 shown below. A FunnyQT in-place transformation rule is defined whose name is `make-groups-of-3!`, and it gets as arguments the `model` on which it should be applied, and an integer `c` which determines how many common movies a clique of three persons needs to have. The case description fixes `c` to 3, but with this parameter, we allow for a bit more generality.

Lines 4 to 12 define the rule's pattern. The structural part defines that it matches a `Movie` element `m` which references three `Person` elements `p0`, `p1`, and `p2` using its `persons` reference.

Additionally, the pattern defines several constraints using the `:when` keyword. The movie `m` needs to have at least three acting persons (line 4), and all persons need to act in at least `c` movies (lines 5, 6, and 9). To avoid duplicate matches where only the order of the three person elements differs, the constraints in line 7 and 10 enforce a lexicographically ascending order of the names of the persons `p0`, `p1`, and `p2`.

```
 1 (ip/defrule make-groups-of-3!
 2   {:forall true, :no-result-vec true}
 3   [model c]
 4   [m<Movie>           :when (>= (person-count m) 3)
 5    m -<persons>-> p0 :when (>= (movie-count p0) c)
 6    m -<persons>-> p1 :when (>= (movie-count p1) c)
 7                      :when (neg? (compare (emf/eget-raw p0 :name) (emf/eget-raw p1 :name)))
 8                      :when (n-common-movies? c p0 p1)
 9    m -<persons>-> p2 :when (>= (movie-count p2) c)
10                      :when (neg? (compare (emf/eget-raw p1 :name) (emf/eget-raw p2 :name)))
11    :when-let [cms (n-common-movies? c p0 p1 p2)]
12    :as [cms p0 p1 p2] :distinct]
13   (emf/ecreate! model 'Clique :persons [p0 p1 p2] :commonMovies cms
14                 :avgRating (avg-rating cms)))
```

Furthermore, line 8 ensures that `p0` and `p1` have at least `c` common movies. The same for the complete clique of three persons is also asserted in line 11, where the common movies are also bound to the variable `cms`.

The constraints in lines 4, 5, 6, 8, and 9 are not really needed. Omitting them would result is the very same set of matches. However, such a FunnyQT pattern is syntactic sugar for a search starting at `Movie` elements `m` and iterating all combinations of the elements targeted by their `persons` references, so it makes sense to add constraints as early as possible in order to cut the search space. Clearly, if a movie has less than 3 actors, it can't be part of a clique of three. Likewise, persons that acted in less than `c` movies can't be part of a clique that requires `c` common movies. And similarly, if `p0` and `p1` do not have the required number of `c` common movies, then `p0`, `p1`, and `p2` cannot have them as well.

The last line of the pattern, line 12, defines that each match should be represented as a vector containing the set of common movies `cms` and the three persons. The keyword `:distinct` specifies that only distinct matches should be found. The reason is that if some clique of three acts in x common movies, there are exactly x matches that differ only in the movie `m`. By omitting the movie from the match representation and specifying that we are only interested in distinct matches, those duplicates are suppressed.

The last two lines define the action that should be applied on matches. A new `Clique` element is created that gets assigned the found persons with their common movies, and the average rating of the common movies.

What has been skipped from explanation until now are the rule's options specified in line 2. The usual rule application semantics are that the rule finds exactly one match (the one it finds first) and then applies its action on it. To transform all matches in the model, one would apply the rule iteratively until no matches can be found anymore. This behavior is especially important when a rule application generates new matches or invalidates existing matches. However, in this case, neither of both is done. After a `Clique` has been created, the movie and the three persons are still a valid match, so calling the rule repeatedly would create multiple `Clique` elements for the same persons. One could use negative application conditions in order to forbid matches where there is a `Clique` already, but the `:forall` option handles the case in a more concise and efficient way. It specifies that when applying the rule *all* matches are searched at once, and then the action is applied to each of them. On multi-core systems, FunnyQT automatically parallelizes this search using Java 7's *ForkJoin* library.

By default, such a `:forall` rule returns a sequence of rule application results (e.g., here a sequence of `Clique` elements). The `:no-result-vec` option specifies that we don't need this sequence which will make the rule only return the number of matches it has processed in order to save some memory.

2.3 Extension Task 1 & Extension Task 4: Compute Top-15 Couples & Cliques

The case description demands for the Extension Tasks 1 and 4 the computation of the top-15 couples and cliques according to the criteria
(a) *average rating of common movies*, and
(b) *number of common movies.*

Whenever there's a tie between two groups according to the current criterium of interest, the case description allows for an arbitrary but stable order. The FunnyQT solution does a bit more: if there's a tie between two groups for the current criterium, the respective other criterium is used to cut it. If that doesn't suffice, i.e., both groups have the same average rating and number of common movies, the names of the group's members are compared as a fallback. Since the person names are unique in the models, there is no chance that no distinction can be made.

The implementation is quite simple in that the sequence of all `Couple` elements (or all `Clique` elements of a given size) are sorted using some comparator. Similar to Java, a comparator in

Clojure is a function that receives two objects and returns a negative integer if the first object should be sorted before the second, a positive integer if the first object should be sorted after the second item, and zero if both objects are equal with respect to sorting order.

The comparator for the average rating is shown in the following listing.

```
1 (defn rating-comparator [a b]
2   (compare (emf/eget b :avgRating) (emf/eget a :avgRating)))
```

As every comparator, it gets two objects `a` and `b` (here, two couples or cliques) and compares them. `compare` is the standard Clojure comparator which works for objects of any class implementing the `java.lang.Comparable` interface. Since we compare the average rating of `b` with the average rating of `a`, a descending order is defined.

The comparator for the number of common movies is shown below.

```
3 (defn common-movies-comparator [a b]
4   (compare (.size ^java.util.Collection (emf/eget-raw b :commonMovies))
5            (.size ^java.util.Collection (emf/eget-raw a :commonMovies))))
```

Like with the `person-count` and `movie-count` functions above, we use type-hints to call the Java method `Collection.size()` directly on the `EList` holding the common movies of the two groups `a` and `b`.

For the next comparator that compares the two groups' actors according to their names, we need some helper functions first. Polyfns are FunnyQT's way to define functions that dispatch polymorphically according to the metamodel type of its first argument. First, a polyfn is declared using `declare-polyfn`, and then arbitrary many implementations for different metamodel types can be added using `defpolyfn`.

```
6 (poly/declare-polyfn actors [group])
7
8 (poly/defpolyfn actors movies.Couple [group]
9   [(emf/eget group :p1) (emf/eget group :p2)])
10
11 (poly/defpolyfn actors movies.Clique [group]
12   (emf/eget-raw group :persons))
```

In line 6, the polyfn `actors` with one parameter `group` is declared. Its intent is to return a collection of all actors and actresses that are part of the given group.

In lines 7 and 8, one implementation for elements of metamodel type `Couple` is added. It returns a vector of two elements: the person in the couple's `p1` reference, and the person in the couple's `p2` reference.

Lines 9 and 10 define another implementation for elements of metamodel type `Clique`. Here, the contents of the group's `persons` reference is returned which already is the collection of all the clique's members.

Using this polyfn, the comparator for ordering groups according to the names of their members can be defined like shown in the next listing.

```
13 (defn names-comparator [a b]
14   (compare (str/join ";" (map #(emf/eget % :name) (actors a)))
15            (str/join ";" (map #(emf/eget % :name) (actors b)))))
```

It simply compares the strings that result from interleaving each group's actor names with a semicolon as a separator. Since here we compare group `a` with group `b`, a lexicographically ascending order is achieved.

Until now, there are only three individual comparators, but sorting is always done with one single comparator. So the following listing defines a higher-order comparator, e.g., a function that receives arbitrary many comparators and returns a new comparator which compares using the given ones.

```
(defn comparator-combinator [& comparators]
  (fn [a b]
    (loop [cs comparators]
      (if (seq cs)
        (let [r ((first cs) a b)]
          (if (zero? r)
            (recur (rest cs))
            r))
        (u/errorf "%s and %s are incomparable!" a b)))))
```

The function `comparator-combinator` returns an anonymous function with two arguments `a` and `b`. This function recurses[9] over the given `comparators`. It applies the first one to `a` and `b`, and if that results in a non-zero value returns this value. But if the value is zero, i.e., no distinction with respect to sorting order can be made with that comparator, the function recurs and `cs` is rebound to the remaining comparators. In case all comparators return zero for two given groups, and error is signalled.

So finally, here are the two top-15 groups functions.

```
(defn groups-by-avg-rating [groups]
  (sort (comparator-combinator rating-comparator common-movies-comparator names-comparator)
        groups))

(defn groups-by-common-movies [groups]
  (sort (comparator-combinator common-movies-comparator rating-comparator names-comparator)
        groups))
```

`groups-by-avg-rating` gets a collection of groups `groups` and then sorts them by the combined comparator first taking the average rating into account, then the number of common movies, and eventually the names of the groups' actors if neither of the two former comparators could decide on the two groups order.

`group-by-common-movies` is defined similar except that the `common-movies-comparator` is applied first instead of the `rating-comparator`.

Those where the actually important parts for solving the top-15 tasks. The solution contains 39 more lines of code that apply the sorting functions to the groups in a model, then take the first 15 groups, format the results nicely, and spit them to files.

3 Evaluation

With respect to correctness, the FunnyQT solution computes the exact same numbers of couples and cliques of various sizes as printed in Table 1 and Table 2 of the case description. Also, the top-15 lists are identical for all models.

Table 2 shows the execution times for the synthetic models, and Table 3 shows the execution times for the IMDb models. These times include the pattern matching time, the time needed for creating the `Couple` and `Clique` elements, and the time needed for setting their attributes and references including the computation of the average ratings. The times needed for generating the synthetic models and for loading the IMDb models are excluded as are the query times for computing the top-15 lists.

The benchmarks were run on a GNU/Linux virtual machine with eight 2.8GHz cores and 32GB of RAM, 30GB of which were dedicated to the JVM process.

As can be seen in Table 2, the FunnyQT solution scales completely linearly for the synthetic models which is expected due to their construction.

[9] Clojure's `(loop [<bindings>] ... (recur <newvals>))` is a local tail-recursion. `loop` establishes bindings just like `let`, and `recur` jumps back to the `loop` providing new values for the variables.

Model (N)	**Couples**	**3-Cliques**	**4-Cliques**	**5-Cliques**
1000	0.274602528	0.349061152	0.483500704	0.422209712
2000	0.547466160	0.722435888	0.724224848	0.628254896
3000	0.867141568	1.069310832	1.028428128	0.949939408
4000	1.119036992	1.437665888	1.375804384	1.216168544
5000	1.406045888	1.798415536	1.723140656	1.502462976
10000	2.830416128	3.607990864	3.482801824	3.098102832
50000	14.324653712	17.972272736	17.346692096	15.327367392
100000	28.300907200	36.352164048	35.396692384	31.440208448
200000	57.159804192	72.209621472	69.233912672	60.430165808

Table 2: Execution times in seconds for the synthetic models

Model	**Couples**	**3-Cliques**
imdb-0005000-49930.movies.bin	1.677278992	6.957570992
imdb-0010000-98168.movies.bin	1.702668160	11.333623024
imdb-0030000-207420.movies.bin	2.610028032	12.507362768
imdb-0045000-299504.movies.bin	3.947932624	15.714349728
imdb-0065000-404920.movies.bin	6.170203664	21.243492448
imdb-0085000-499995.movies.bin	9.012942560	26.388751712
imdb-0130000-709551.movies.bin	18.135307008	54.709762992
imdb-0200000-1004463.movies.bin	35.409998560	117.833811360
imdb-0340000-1505143.movies.bin	88.052992592	366.832061200
imdb-0495000-2000900.movies.bin	159.973018224	757.280006768
imdb-0660000-2501893.movies.bin	278.318685728	1457.689379247
imdb-all-3257145.movies.bin	619.160156640	4295.030516512

Table 3: Execution times in seconds for the IMDb models

In contrast, the transformation of the real IMDb models requires more effort in general because they are much more cross-linked. Whereas in the synthetic models, every person acts in at most five movies, and every movie has at most five acting persons, in the complete IMDb model, there are persons acting in up to 1800 movies, and movies with more than 1200 actors.

The main bottleneck of the FunnyQT transformation is the required memory. The generated rules for finding groups of a given size first compute all matches (each match being represented as a vector containing the persons being members of the group plus their set of common movies) and then generate one new couple or clique element for each match. This means that the original model, all matches, and also all new elements reside in memory at the same time.

One could sacrifice a bit of performance for better memory-efficiency by not including the set of common movies already in the matches, but instead re-compute it in the rule's action creating the couple or clique elements.

4 Conclusion

In this paper, the FunnyQT solution to the TTC 2014 Movie Database Case has been discussed. It correctly solves all core and extension tasks, and its performance is quite good due to the fact that FunnyQT is able to perform pattern matching in parallel on multi-core systems.

Also, the solution is concise. All in all, it consists of 152 lines of code (including boilerplace code like namespace definitions) in three source files, one for the generation of the synthetic test models (30 LOC), one for the couple and cliques rules (52 LOC), and one for the queries (70 LOC).

References

[Fow10] Martin Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010.

[HKT14] Tassilo Horn, Christian Krause, and Matthias Tichy. The ttc 2014 movie database case. In *Transformation Tool Contest 2014*, 2014.

[Hor13] Tassilo Horn. Model querying with funnyqt - (extended abstract). In Keith Duddy and Gerti Kappel, editors, *ICMT*, volume 7909 of *Lecture Notes in Computer Science*, pages 56–57. Springer, 2013.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, 2 edition, 2008.