Solving the Class Diagram Restructuring Transformation Case with FunnyQT

Dipl.-Inform. Tassilo Horn
horn@uni-koblenz.de
Institute for Software Technology, University Koblenz-Landau, Germany

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 2013 Class Diagram Restructuring Transformation Case. This solution and the GROOVE solution share the *best overall solution award* for this case.

1 Introduction

FunnyQT is a new model querying and transformation approach which is implemented as an API for the functional, JVM-based Lisp-dialect Clojure. It provides several sub-APIs for implementing different kinds of queries and transformations. For example, there is a model-to-model transformation API, and there is an in-place transformation API for writing programmed graph transformations. FunnyQT currently supports EMF and JGraLab models, and it can be extended to other modeling frameworks, too.

For solving the tasks of this transformation case[1], only FunnyQT's plain querying and model manipulation APIs have been used, and the task is tackled algorithmically.

2 The Core Task

The core task's solution consists of some helper functions, a function for finding sets of pullable properties and sorting them heuristically in order to achieve effective results, the rules depicted in the case description [LKR13], and a function applying the rules in order to realize the transformation. This section starts with the helpers, then explains the function finding pullable properties, and then discusses the restructuring rules. The complete source code is printed in Appendix A.

Helper Functions. The helper functions discussed in this section are quite simple and factor out functionality that is used at several places in the rules. There are several very simple helpers that are not explained in detail. `add-prop!` adds a new `Property` with some given name and `Type` to some given `Entity`. `delete-prop!` deletes the property identified by a given name from a given entity. The `pull-up` function gets a list of `[prop-name type]` tuples, a set of source entities, and a target entity. It then adds new properties to the target entity and deletes them from the source entities. Lastly, there's `make-generalization!` which creates a new `Generalization` between a sub- and its super-entity, and there's `make-entity!` that creates a new `Entity`.

[1] This FunnyQT solution is available at `https://github.com/tsdh/ttc-2013-cd-restruct` and on SHARE (image TTC13::Ubuntu12LTS_TTC13::FunnyQT.vdi)

Submitted to:
Transformation Tool Contest 2013 (TTC'13)

The function `prop-type-set` gets an `Entity e` and returns its set of `[prop-name type]` tuples, i.e., there's one such tuple for any owned attribute of `e`.

```
(defn prop-type-set [e]
  (set (map (fn [p] [(eget p :name) (eget p :type)])
            (eget e :ownedAttribute))))
```

The `filter-by-properties` function gets a collection of `[prop-name type]` tuples via its `pnts` parameter, and a collection of entities via its `entities` parameter. It returns the subset of `entities` for which every entity defines all of the given properties with identical types.

```
(defn filter-by-properties [pnts entities]
  (set (filter (fn [e] (forall? #(member? % (prop-type-set e)) pnts))
               entities)))
```

Restructuring Heuristics. This solution doesn't pull up one attribute at a time, but instead it pulls up the *maximal set of properties that are shared by a maximum of entities*. I.e., the heuristics can be specified as follows. Let P_1 and P_2 be sets of properties shared by the sets of entities E_1 and E_2, respectively.

1. If $|E_1| > |E_2|$, then the solution pulls up the properties of P_1 instead of the properties of P_2. (*Maximality wrt. the number of entities declaring these properties*)
2. If $|E_1| = |E_2|$, then the solution pulls up the properties of P_i where $i = \begin{cases} 1 & \text{if } |P_1| \geq |P_2| \\ 2 & \text{otherwise} \end{cases}$. (*Maximality wrt. the number of pullable properties.*)

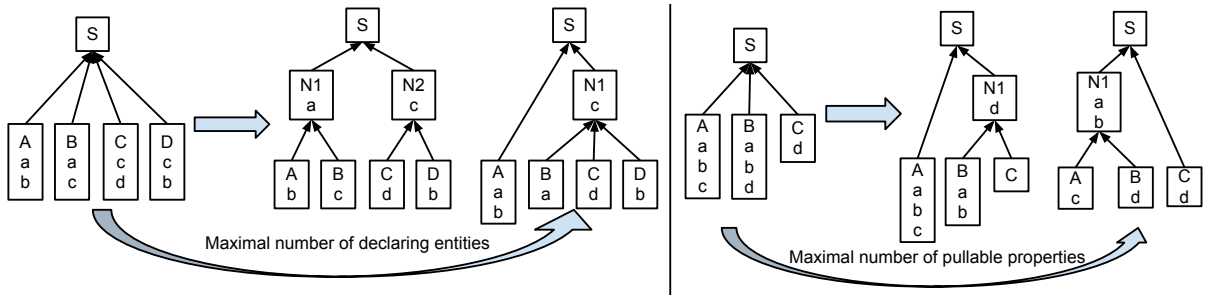

Figure 1: Examples for the heuristics

Figure 1 illustrates these heuristics with two examples. In the left example, the property `c` is shared by three classes, whereas `a` and `b` are shared by only 2 classes each. If the transformation pulls up `a` first into a new class, `c` can be pulled up only from `C` and `D` into another new class. The number of property declaration decreases from 8 to 6, `b` and `c` remain duplicated once, and 2 new classes have been created. If the transformation uses the first heuristic, it pulls up `c` first because that's common to more classes than `a` and `b`. This also results in 6 remaining property declarations, `a` and `b` remain duplicated once, but only one new class has been created.

In the right example, the set of properties `{a, b}` and `{d}` are shared by two classes both. If the transformation decides to pull up `d`, the number of property declarations decreases from 7 to 6, `a` and `b` remain duplicated once, and one new class has been created. If the transformation uses the second heuristic, it pulls up `a` and `b`, and the number of property declarations decreases from 7 to 5, only `d` remains duplicated once, and again one new class has been created.

The `common-props` function finds sets of pullable properties and sorts them according to the heuristics. It is by far the most complex function of the transformation. The function receives a set of entities via its `classes` parameter and returns the properties common to a maximal subset of these entities.

```
1  (defn common-props [classes]
2    (let [pes (set (map (fn [pnt] [pnt (filter-by-properties [pnt] classes)])
3                        (set (mapcat prop-type-set classes))))
4          freq-map (apply hash-map
5                          (mapcat (fn [[_ ents]] [ents (count (filter #(= ents (second %))
6                                                                      pes))])
7                                  pes))
8          collapse (fn collapse [aes]
9                     (when-let [[pnt entities] (first aes)]
10                      (let [[s r] (split-with (fn [[_ ents]] (= entities ents)) aes)]
11                        (cons [(map first s) entities]
12                              (lazy-seq (collapse r))))))]
13     (collapse (into (sorted-set-by
14                      (fn [[_ aes :as a] [_ bes :as b]]
15                        (let [x (- (count bes) (count aes))]
16                          (if (zero? x)
17                            (let [x (- (freq-map bes) (freq-map aes))]
18                              (if (zero? x) (compare a b) x))
19                            x))))
20                     pes))))
```

In line 2, `pes` is bound to a set of tuples `[pnt entity-set]`, where `pnt` is a `[prop-name type]` tuple and `entity-set` is the set of all entities declaring such a property. That is, `pes` has the following form[2].

```
#{[[pn1 t1] #{e1 e2 e3}]     [[pn4 t2] #{e2 e3 e4}]
  [[pn2 t2] #{e2 e3 e4 e5}]  [[pn3 t2] #{e1 e2 e3}]}
```

In line 4, `freq-map` is bound to a hash-map that maps to each set of entities occuring in the items of `pes` the number of occurences in there. This map is used to implement the second heuristic.

In line 8, a local function `collapse` is defined. Before explaining that, first lines 13 to 22 are to be explained. What's done there is that the entries of the set `pes` are put into a sorted set. The sorting order is determined by the comparator function defined in lines 14 to 21. It receives two items of the `pes` set, binds their `entity-sets` to `aes` and `bes`, respectively, and then performs these checks:

1. If `bes` contains more entities than `aes`, `b` should be sorted before `a`. This implements heuristic 1.
2. Else, if the entity set `bes` occurs more often in the items of `pes`, `b` should be sorted before `a`. This implements heuristic 2.
3. Else, the sorting order is not important and determined by Clojure's standard `compare` function that produces a stable ordering upon all objects implementing `Comparable`.

As a result, the sorted set has the following structure, i.e., items with larger entity sets are sorted before items with smaller entity sets. The item with `pn2` is sorted before the others because it is shared by 4 entities.

```
#{[[pn2 t2] #{e2 e3 e4 e5}]  [[pn1 t1] #{e1 e2 e3}]
  [[pn3 t2] #{e1 e2 e3}]     [[pn4 t2] #{e2 e3 e4}]}
```

[2] #{...} is a Clojure set literal.

In case of equally large entity sets, the number of occurences of the entity sets determines the sorting order, e.g., the items with `pn1` and `pn2` are sorted before the item with `pn4`, because their entity sets occur twice whereas the entity set of `pn4` occurs only once.

Finally, this set is mangled by the local `collapse` function defined in lines 8 to 12. It simply collapses (merges) adjacent items with equal entity sets, thus the result of the function has the following form.

```
([([pn2 t2]) #{e2 e3 e4 e5}],    [([pn1 t1] [pn3 t2]) #{e1 e2 e3}],    [([pn4 t2]) #{e2 e3 e4}]}
```

Because the items of `pn1` and `pn3` have the same entity set, they are merged into one item.

Restructuring Rules. The solution defines the function `pull-up-helper` shown in Listing 1 which can implement all three restructuring rules by parameterizing it appropriately. The function receives the root `model` object `mo`, a superclass `super`, and a set of entities `classes` in which to find common properties,. In case of rule 1 and rule 2, `super` is the superclass of all `classes`, and in case of rule 3, the `super` parameter is `nil` and `classes` is the set of top-level classes.

```
1  (defn pull-up-helper [mo super classes]
2    (when (seq classes)
3      (when-let [[pnts entities] (first (common-props classes))]
4        (if (and super (= classes entities))
5          (pull-up mo pnts entities super)  ;; rule 1
6          (when (> (count entities) 1)
7            (let [nc (make-entity! mo)]     ;; rule 2 if super, else rule 3
8              (pull-up mo pnts entities nc)
9              (doseq [s entities]
10               (doseq [oldgen (eget s :generalization)
11                       :when (= super (adj oldgen :general))]
12                 (edelete! oldgen))
13               (make-generalization! mo s nc))
14             (when super (make-generalization! mo nc super))
15             true))))))
```

Listing 1: A restructuring function able to implement all three rules

When the set of classes is not empty (line 2), and if there are common properties (line 3), the largest list of common properties among the lists of properties declared by a maximal number of entities is bound to `pnts`, and the entities declaring these properties are bound to `entities`. In case `entities` equals the set of all `classes` (line 4), the situation is that of rule 1, and all properties in `pnts` are pulled up to `super` (line 5). In the other case, the maximal set of common properties is shared by a maximal but strict subset of `classes`. Here, it has to be ensured that there are more than one entity declaring these properties (line 6), because else the inheritance depth would increase without removing declarations. Then, the situation is that of rule 2 if `super` is non-nil, and the situation is that of rule 3 if `super` is nil. In any case, all shared properties are pulled into a new entity `nc`, and the generalizations are adapted.

The overall transformation function simply calls the `pull-up-helper` function shown above with appropriate parametrization as long as it can find a match.

Multiple Inheritance Extension. The solution discussed so far works well also if the initial model already contains multiple inheritance. However, they won't create new entities that specialize more than one other entity. To exploit multiple inheritance in order to restructure the model resulting from the

core rules so that there are no duplicate properties, one additional rule is used. It computes the set of duplicated properties of all classes, and then acts according these heuristics.

1. If one of the entities declaring the duplicated property is a top-level class created by the core task, then the other entities become its subclasses. Only top-level entities created by the core task are reused, because reusing one that already existed in the original class model makes the result's type hierarchy incompatible with original one, i.e., before B was no subclass of A, but afterwards it is.
2. Else, a new entity is created as superclass of the entities, and the property is pulled up.

3 Evaluation

The evaluation results requested by the case description [LKR13] are summarized in Table 1.

With 110 LOC (core + extension task), the solution is quite *concise* given that its heuristics are more advanced than what was demanded. Due to these heuristics, its *effectiveness* is 100% for all provided and several additional models (e.g., the ones in Figure 1). The case description defines the *complexitiy* as the sum of operator occurences, type references, and feature references. The FunnyQT solution consists of 161 function calls (or calls to special forms or macros), 4 type references, and 25 feature references. The *development effort* has been about 8 hours for the solution plus 2 hours for writing unit tests for it.

Size (LOC)	90 (core only), 105 (core + extension)
Complexity	190 = 161 funcalls + 4 type refs + 25 feature refs
Effectiveness	100%
Development effort	approx. 8 hours (solution) + 2 hours (tests)
Execution time	6 secs for the largest model (`testcase2_10000.xmi`)
History of use	approx. 1 year
No. of case studies	published: the 3 TTC13 cases, unpublished: approx. 20
Maximum capability	approx. 2 million elements on SHARE

Table 1: Evaluation measures

The detailed *execution times* on SHARE for the larger models are depicted in Table2. The largest provided model `testcase2_10000` consisting of 100000 elements[3] can be processed in about six seconds which is more than a thousand times faster than the reference UML-RSDS solution.

Model	Core	Core and Extension
testcase2_1000	418 ms	434 ms
testcase2_5000	2455 ms	2585 ms
testcase2_10000	5656 ms	6041 ms
testcase3	248 ms	268 ms
testcase2_200000	1006848 ms	1045648 ms

Table 2: Detailed execution times on SHARE

To determine the *maximum capability* of the solution, models up to two millions of elements have been created. Given the limited amount of 800 MB RAM available to the JVM process on SHARE, the model with 2 million elements (`testcase2_200000`) is about the maximum capability for the solution.

References

[LKR13] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Case study: Class diagram restructuring, 2013.

[3] The models testcase2_n actually consist of $10 \times n$ elements.

A The Complete Source Code of the Solution

```
(defn add-prop! [mo e pn t]
  (let [p (ecreate! 'Property)]
    (eadd! mo :propertys p)
    (eset! p :name pn)
    (eset! p :type t)
    (eadd! e :ownedAttribute p)))

(defn delete-prop! [e pn]
  (let [p (first (filter #(= pn (eget % :name))
                         (eget e :ownedAttribute)))]
    (edelete! p)
    (eremove! e :ownedAttribute p)))

(defn pull-up [mo pnts froms to]
  (doseq [[pn t] pnts]
    (add-prop! mo to pn t)
    (doseq [s froms]
      (delete-prop! s pn)))
  true)

(defn make-generalization! [mo sub super]
  (let [gen (ecreate! 'Generalization)]
    (eadd! mo :generalizations gen)
    (eset! gen :general super)
    (eset! gen :specific sub)))

(defn make-entity! [mo]
  (let [e (ecreate! 'Entity)]
    (eadd! mo :entitys e)
    (eset! e :name (str (gensym "NewClass")))))

(defn prop-type-set [e]
  (set (map (fn [p] [(eget p :name) (eget p :type)])
            (eget e :ownedAttribute))))

(defn filter-by-properties [pnts entities]
  (set (filter (fn [e]
                 (forall? #(member? % (prop-type-set e)) pnts))
               entities)))

(defn common-props [classes]
  (let [pes (set (map (fn [pnt]
                        [pnt (filter-by-properties [pnt] classes)])
                      (set (mapcat prop-type-set classes))))
        freq-map (apply hash-map
                        (mapcat (fn [[_ ents]]
                                  [ents (count (filter #(= ents (second %))
                                                       pes))])
                                pes))
        collapse (fn collapse [aes]
                   (when-let [[pnt entities] (first aes)]
                     (let [[s r] (split-with (fn [[_ ents]]
                                               (= entities ents)) aes)]
```

```
                        (cons [(map first s) entities]
                              (lazy-seq (collapse r)))))))]
    (collapse (into (sorted-set-by
                     (fn [[_ aes :as a] [_ bes :as b]]
                       (let [x (- (count bes) (count aes))]
                         (if (zero? x)
                           (let [x (- (freq-map bes) (freq-map aes))]
                             (if (zero? x)
                               (compare a b)
                               x))
                           x))))
                    pes))))

(defn pull-up-helper [mo super classes]
  (when (seq classes)
    (when-let [[pnts entities] (first (common-props classes))]
      (if (and super (= classes entities))
        (pull-up mo pnts entities super)  ;; rule 1
        (when (> (count entities) 1)
          (let [nc (make-entity! mo)]     ;; if super rule 2, else rule 3
            (pull-up mo pnts entities nc)
            (doseq [s entities]
              (doseq [oldgen (eget s :generalization)
                      :when (= super (adj oldgen :general))]
                (edelete! oldgen))
              (make-generalization! mo s nc))
            (when super (make-generalization! mo nc super))
            true))))))

(defn exploit-multiple-inheritance [mo]
  (doseq [[pnts entities] (common-props (eget mo :entitys))
          :while (> (count entities) 1)]
    (let [[nc reuse]
          (if-let [top (first (filter
                               #(and (empty? (eget % :generalization))
                                     (re-matches #"NewClass.*" (eget % :name)))
                               entities))]
            [top true]
            [(make-entity! mo) false])]
      (doseq [[pn t] pnts]
        (when-not reuse
          (add-prop! mo nc pn t))
        (doseq [e (remove #(= nc %) entities)]
          (delete-prop! e pn)
          (make-generalization! mo e nc))))))

(defn pull-up-1-2 [mo]
  (loop [classes (eget mo :entitys), applied false]
    (if (seq classes)
      (let [super (first classes)
            result (pull-up-helper
                    mo super (set (adjs super :specialization :specific)))]
        (recur (rest classes) (or result applied)))
      applied)))

(defn pull-up-3 [mo]
```

```
110   (pull-up-helper mo nil (set (remove #(seq (eget % :generalization))
111                                       (eget mo :entitys)))))
112
113 (defn pull-up-attributes [model multi-inheritance]
114   (let [mo (the (eallobjects model 'model))]
115     (iteratively #(let [r (pull-up-1-2 mo)]
116                     (or (pull-up-3 mo) r)))
117     (when multi-inheritance (exploit-multiple-inheritance mo))
118     model))
```