Solving the Class Diagram Restructuring Transformation Case with FunnyQT

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

March 20, 2013

Abstract

This paper describes the FunnyQT solution to the TTC 2013 Class Diagram Restructuring Transformation Case.

FunnyQT is a model querying and model transformation library for the functional Lisp-dialect Clojure. It supports the modeling frameworks JGraLab and EMF natively, and it is designed to be extensible towards supporting other frameworks as well.

FunnyQT provides a rich and efficient querying API, a model manipulation API, and on top of those, there are several sub-APIs for implementing several kinds of transformations such as ATL-like model transformations or programmed graph transformations.

This solution tackles the case algorithmically using FunnyQT's plain querying and model manipulation APIs.

1 Introduction

FunnyQT is a new model querying and transformation approach. Instead of inventing yet another language with its own concrete syntax and semantics, it is implemented as an API for the functional, JVM-based Lisp-dialect Clojure[1]. It's JVM-basing provides wrapper-free access to all existing Clojure and Java libraries, and to other tools in the rich Java ecosystem such as profilers.

FunnyQT natively supports the de-facto standard modeling framework EMF [SBPM08] and the TGraph modeling framework JGraLab[2], and it is designed to be extensible towards other frameworks as well.

FunnyQT's API is split up in several sub-APIs. On the lowest level there is a core API for any supported modeling framework providing functions for loading and storing models, accessing, creating, and deleting model elements, and accessing and setting attribute values. These core APIs mainly provide a concise and expressive interface to the native Java APIs of the frameworks. On top of that, there's a generic API providing the subset of core functionality that is common to both supported frameworks such as navigation via role names, access to and manipulation of element properties, or functionality concerned with typing imposed by metamodels. Furthermore, there is a generic quering API providing important querying concepts such as quantified expressions, regular path expressions, or pattern matching.

Based on those querying and model manipulation APIs, there are several sub-APIs for implementing different kinds of transformations. For example, there is a model transformation API similar to ATL [JK05] or ETL [KRP13], or there is an in-place transformation API for writing programmed graph transformations similar to GrGen.NET [BGJ13].

[1] `http://clojure.org/`
[2] `http://jgralab.uni-koblenz.de`

Especially the pattern matching API and the transformation APIs make use of Clojure's Lisp-inherited metaprogramming facilities [Gra93, Hoy08] in that they provide macros creating internal DSLs [Fow10] providing concise, boilerplate-free syntaxes to users. Patterns and transformations written in these internal DSLs get transformed to usual Clojure code using the FunnyQT querying and model manipulation APIs by the Clojure compiler.

For solving the tasks of this transformation case, only FunnyQT's plain querying and model manipulation APIs have been used.

2 The Core Task

The core task's solution consists of several helper functions, a function for finding sets of pullable properties and sorting them heuristically in order to achieve effective results, the three restructuring rules depicted in the case description [LKR13], and a last function composing the rules in order to realized the transformation. This solution description starts with the helpers, Section 2.2 dives into the details of the function finding pullable properties, Section 2.3 explains the restructuring rules, and Section 2.4 describes the transformation function.

2.1 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.

The function `add-prop!` shown in Listing 1 receives the single root `model` object `mo`, an `Entity e`, a property name `pn`, and a `Type t`. It creates a new `Property p` with the given name and type, and assigns it to both the given entity and model object.

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

Listing 1: A function for adding a property to an entity

The `delete-prop!` function depicted in Listing 2 takes an `Entity e`, and a property name `pn`. It then selects the `Property p` in the `ownedAttribute` list of `e` which has this name and deletes it.

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

Listing 2: A function for deleting a property from an entity

The function `pull-up` depicted in Listing 3 combines `add-prop!` and `delete-prop!`. It receives the root `model` object `mo`, a set of `[propname, type]` tuples `pnts`, a set of entities `froms` that have these properties, and a single entity `to`. For every `[propname, type]` tuple in `pnts`, it creates a new property in `to`, and deletes the corresponding properties in all `froms`.

Clojure's `doseq` is similar to the *enhanced for loop* in Java, that is, the elements of an iterable object are bound to a variable one after the other, and the body is executed for side-effects only. One interesting difference visible in line 13 is that `doseq` allows for *destructuring*. In Lisp, destructuring means binding the contents of some structured object directly by imitating

```clojure
(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)
```

Listing 3: A function for pulling up properties

the object's structure. Since `pnts` is a set of `[propname, type]` tuples, in every iteration the property name is bound directly to `pn`, and the type is directly bound to `t`.[3]

The next helper, `make-generalization!` shown in Listing 4, is very simple again. It creates a new `Generalization` relationship between an entity `sub` and its determined superclass `super`.

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

Listing 4: A function for creating a generalization relationship

The `make-entity!` function is Listing 5 creates a new `Entity` and assigns it to the root `model` object `mo`. Its name is set to the string "NewClass" followed by some number. `gensym` is a tool used mostly for metaprogramming. It returns a guaranteed unique symbol whose name starts with the given prefix and gets a number appended. Here, this symbol is just converted to a string.

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

Listing 5: A function for creating an entity

The function `prop-type-set` shown in Listing 6 gets an `Entity e` and returns its set of `[propname, type]` tuples, i.e., there's one such tuple for any owned attribute of `e`.

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

Listing 6: A function retrieving the set of [propname, type] tuples of an entity

The standard Clojure higher-order function `map` gets a function and a collection, and it returns a sequence containing the results of applying that function to every element of the collection. `set` then coerces this sequence to a set.

The `filter-by-properties` function in Listing 7 gets a collection of `[propname, type]` tuples via its `pnts` parameter, and a collection of entities via its `classes` parameter. It returns the subset of `classes`, where every entity defines all of the given properties with identical types.

The syntax `#(member? % (prop-type-set e))` is the shortest form of defining an anonymous function in Clojure. The parameters are defined as `%i` for `i` being a number larger than 0,

[3]Destructuring is supported by all binding forms including let, doseq, loop, the sequence comprehension for, and function parameter vectors.

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

Listing 7: A function for filtering entities to those declaring a given set of properties

and `%` is equivalent to `%1`. The largest `i` defines the function's arity. So here a local function of arity 1 is defined that checks if its single argument, a `[propname, type]` tuple, is member of the `prop-type-set` of entity `e`.

2.2 Restructuring Heuristics

The rules of this solution don't pull up one attribute at a time, but instead they pull up the *maximal set of properties that are shared by a maximum of entities*. That is, the heuristics used 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 illustrates these heuristics with two examples. In the upper 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 lower 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.

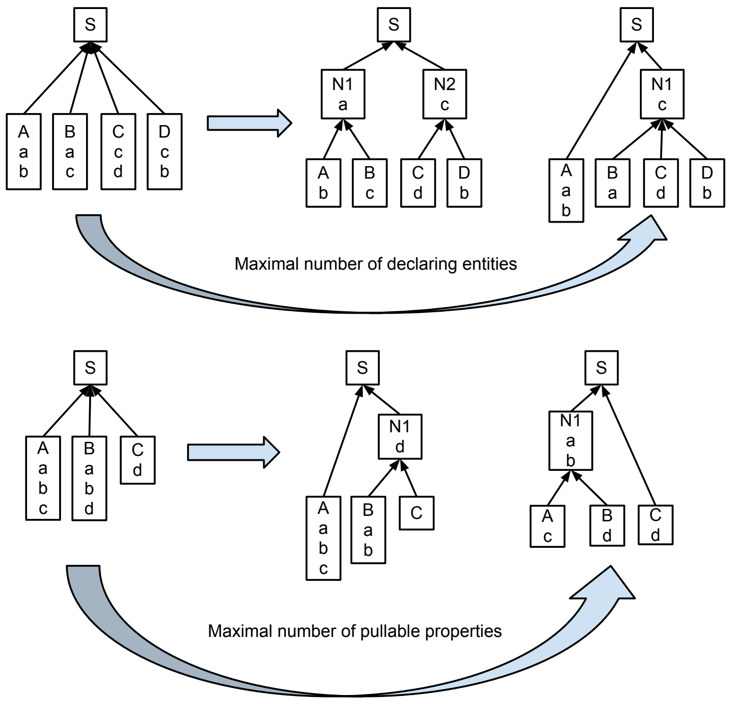
The `common-props` function for finding sets of pullable properties and sorting them according to the heuristics is shown in Listing 8. It is by far the most complex function of the transformation. The function receives a set of entities via its `classes` parameter. The properties common to a maximal subset of these entities are to be found.

In line 35, the variable `pes` is bound to a set of the following form[4].

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

I.e., the set's items are tuples where the first component is a `[propname, type]` tuple, and the second component is the set of entities declaring this property. There is exactly one item for every unique pair of property name and type.

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


Maximal number of declaring entities
Maximal number of pullable properties
Figure 1: Examples for the heuristics

In line 38, `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 will be used to implement the second heuristic.

In line 43, a local function `collapse` is defined. Before explaining that, first lines 49 to 58 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 50 to 57. It's

```
34 (defn common-props [classes]
35   (let [pes (set (map (fn [pnt]
36                         [pnt (filter-by-properties [pnt] classes)])
37                       (set (mapcat prop-type-set classes))))
38         freq-map (apply hash-map
39                         (mapcat (fn [[_ ents]]
40                                   [ents (count (filter #(= ents (second %))
41                                                        pes))])
42                                 pes))
43         collapse (fn collapse [aes]
44                    (when-let [[pnt entities] (first aes)]
45                      (let [[s r] (split-with (fn [[_ ents]]
46                                                (= entities ents)) aes)]
47                        (cons [(map first s) entities]
48                              (lazy-seq (collapse r))))))]
49     (collapse (into (sorted-set-by
50                      (fn [[_ aes :as a] [_ bes :as b]]
51                        (let [x (- (count bes) (count aes))]
52                          (if (zero? x)
53                            (let [x (- (freq-map bes) (freq-map aes))]
54                              (if (zero? x)
55                                (compare a b)
56                                x))
57                            x))))
58                     pes))))
```

Listing 8: A function for retrieving the maximal set of properties shared by a maximum of classes

meaning is equivalent to comparator objects in Java. It gets two objects and returns a negative number if the first object should be sorted before the second, it returns a positive number if the second object should be sorted before the first one, and it returns zero if both objects are equal.

Here, the comparator function receives two items of the `pes` set. It uses destructuring to bind the first one to `a` and its entity set to `aes`. The first component of each item, the `[propname, type]` tuple, is ignored. Similarly, the second item is bound to `b` and its set of entities is bound to `bes`. The comparator function 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.

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

Items with larger entity sets are sorted before items with smaller entity sets, e.g., the item with `pn2` is sorted before the others because it is shared by 4 entities. 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. When the provided sorted set `aes` contains a first entry (line 44), the set is split in two parts `s` and `r`, where `s` contains all the items that have the very same entities set as the first item. The result of the function is a list[5] where the head is a tuple `[pnts entities]`, where `pnts` is a list of `[propname, type]` tuples, and the rest is the result of applying `collapse` to the split-off part `r`. Using the example again, the result structure is as follows.

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

That is, neighboring items with equal entity sets are collapsed into one.

2.3 Restructuring Rules

In this section, the three restructuring rules are explained. With the `common-props` function in place, one observations can be made. Rule 1 and rule 2 are nearly identical. When looking for shared properties in a set of subclasses, if there are properties shared by all of them, this is of course the maximal set, too. The only difference is that for rule 1, no new entity has to be created, while for rule 2, where only a subset of subclasses shares properties, a new entity has to be created.

Also rule 2 and rule 3 are very similar. The difference is only if common properties are searched below subclasses of a given class, or below top-level classes.

[5]In fact, the result is no list but a lazy sequence because the recursive call is wrapped with `lazy-seq`. A lazy seq is similar to a list, but the elements in its rest are not computed (*realized*) before someone tries to access them. That is, the rest of a lazy sequence is actually a closure that knows how to realize the sequence one step further.

Therefore, the solution defines the function `pull-up-helper` shown in Listing 9 which can implement all 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.

```
59 (defn pull-up-helper [mo super classes]
60   (when (seq classes)
61     (when-let [[pnts entities] (first (common-props classes))]
62       (if (and super (= classes entities))
63         (pull-up mo pnts entities super)  ;; rule 1
64         (when (> (count entities) 1)
65           (let [nc (make-entity! mo)]      ;; rule 2 if super, else rule 3
66             (pull-up mo pnts entities nc)
67             (doseq [s entities]
68               (doseq [oldgen (eget s :generalization)
69                       :when (= super (adj oldgen :general))]
70                 (edelete! oldgen))
71               (make-generalization! mo s nc))
72             (when super (make-generalization! mo nc super))
73             true))))))
```

Listing 9: A restructuring function able to implement all three rules

When the set of classes is not empty[6] (line 60), and if there are common properties, 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` (line 61).

In case `entities` equals the set of all `classes` (line 62), the situation is that of rule 1, and all properties in `pnts` are pulled up to `super` (line 63).

In the other case, the maximal set of common properties is shared by a maximal but real subset of `classes`. In that case, it has to be ensured that there are more than one entity declaring these properties (line 64), because else the inheritance depth would increase without removing declarations.

Then, the situation is that of rule 2 if `super` is non-nil, or the situation is that of rule 3 if `super` is nil. In any case, a new entity `nc` has to be created (line 65), and all properties `pnts` have to be pulled from `entities` to `nc` (line 66). The original generalizations to `super` are deleted (lines 67 to 70), and new generalizations from the entities to `nc` are created (line 71).

If `super` is non-nil, also a new generalization from `nc` to the original superclass `super` is created (line 72). Finally, `true` is returned to indicate to the caller that a restructuring has performed.

Using this helper, the individual rules are very easy. Listing 10 shows the function implementing rule 1 and 2. It uses `loop/recur`, a local tail-recursion, to call the `pull-up-helper` once for any class and its subclasses. The `applied` variable determines the return value of the function. It is `true` if the helper could perform a restructuring at least once.

```
(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)))
```

Listing 10: A function applying rule 1 and 2 to all classes

[6](seq coll) is the canonical non-emptyness check in Clojure.

Listing 11 depicts the function `pull-up-3` implementing rule 3.

```
(defn pull-up-3 [mo]
  (pull-up-helper mo nil (set (remove #(seq (eget % :generalization))
                                      (eget mo :entitys)))))
```

Listing 11: A function applying rule 3 to all top-level classes

It simply calls the helper providing `nil` as superclass parameter, and the set of top-level classes.

2.4 The Transformation Function

The function `pull-up-attributes` shown in Listing 12 implements the overall transformation by calling the rules defined in the previous section. It receives the class diagram EMF model via its `model` parameter, and the `multi-inheritance` parameter specifies if multiple inheritance should be exploited in order to minimize the number of property declarations. This extension task is discussed in Section 3.

```
85 (defn pull-up-attributes [model multi-inheritance]
86   (let [mo (the (eallobjects model 'model))]
87     (iteratively #(let [r (pull-up-1-2 mo)]
88                     (or (pull-up-3 mo) r)))
89     (when multi-inheritance (exploit-multiple-inheritance mo))
90     model))
```

Listing 12: The transformation function

In line 86, the single root `model` object is retrieved and bound to the variable `mo`. The function `iteratively` in line 87 takes a function and applies it as long as it returns `true`. The anonymous function given to it first applies the `pull-up-1-2` rule, then the `pull-up-3` rule. It returns `true` if at least one of the rules could be applied.

In case `multi-inheritance` is `true`, another rule which is explained in Section 3 is applied. Finally, the `model` is returned.

3 The Multiple Inheritance Extension Task

The rules discussed in Section 2.3 work well also if the initial model already contains multiple inheritance. However, they won't create new classes that specialize more than one single superclass.

To exploit the presence of multiple inheritance in order to restructure the model resulting from the core rules so that every property is declared exactly once, the additional rule shown in Listing 13 is used. It receives the root `model` object as its only parameter.

The function iterates over the result list of common properties of all entities in the model that are shared by at least 2 entities. In every iteration, `pnts` is bound to a list of `[propname type]` tuples, and `entities` is the set of entities declaring these properties.

Naively, one could create a new entity in every iteration, pull the common properties into it, and make all `entities` subclasses of the new one. But this might create more new entities than necessary. Instead, it is preferrable to reuse some existing top-level class that already declares these properties. In that case, only new generalizations have to be created from all `entities` (except this one) to this top-level class, and the shared properties have to be deleted from the subclasses.

The function takes into account one further issue. When looking for such a top-level class, only the classes created by the rules of the core task are considered. The reason is that reusing an entity that already existed in the original class model will make the transformation result's

```
(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))))))
```

Listing 13: A function for exploiting multiple inheritance

type hierarchy incompatible with that of the original model, that is, in the original model `B` was no subclass of `A`, but in the result model it is. Such a change is likely to break programs implemented using these classes since its polymorphic method calls and instance tests might deliver unexpected results.

4 Running the Transformation on SHARE

The FunnyQT solution to this case (and the other cases) are installed on the SHARE image `Ubuntu12LTS_TTC13::FunnyQT.vdi`. Running the solution is simple.

1. Open a terminal.
2. Change to the Class Diagram Restructuring project:
   ```
   $ cd ~/Desktop/FunnyQT_Solutions/ttc-2013-cd-restruct/
   ```
3. To run the transformation for all provided and some additional but small models, use:
   ```
   $ lein test
   ```
   This will run the complete transformation on all provided and some additional test models and print the execution times. For every model, the core task transformation and also the multiple inheritance enabled transformation is performed.
   The result models are also validated using unit tests defined in the file `test/ttc_2013_cd_restruct/core_test.clj`. The result models and visualizations are saved to the `results` directory.
4. To determine the maximum capability of the solution, some more variants of test case 2 have been generated. Here the model sizes are 500000, 1000000, and 2000000 elements. To apply the transformation to these models, use:
   ```
   $ lein test :stress
   ```
   The maximum capability (on SHARE) is the largest model with 2 million elements. To apply the transformation to only this model, use:
   ```
   $ lein test :maximum
   ```
   Running the tests with `:maximum` will take about 40 minutes, and running the tests with `:stess` will take about an hour.

5 Evaluation

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

With respect to *size*, the FunnyQT solution to the core task consists of exactly 90 lines of code excluding comments, empty lines, and namespace declarations. The extension rule for exploiting multiple inheritance accounts for another 15 LOCs. This seems quite good, although the UML-RSDS reference solution is only about 30 LOCs due to its very declarative nature.

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.

For all provided test models, the *effectiveness* of the core task solution is 100%. The unit tests also apply the core transformation to several additional models (such as the ones depicted in Figure 1) in which the implemented heuristics are of importance. For these models, the effectiveness is 100%, too. When the transformation including the multiple inheritance rule is applied to the models, the result is that every duplication is removed, i.e., every property is declared exactly once.

The *development effort* is quite hard to estimate retrospectively, because it has been an on-off task. The initial working version for the core task has been implemented from scratch in about two hours. After thinking about it in my spare time and restructuring some more sample models using pen and paper, it has been refactored and enhanced to the solution presented in this paper. All in all, a development effort of approximately 8 hours seems realistic. About two additional hours have been invested for defining the unit tests.

Measure	**Value**
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
Maximum capability	unknown, but at least 1 million elements should work

Table 1: Evaluation measures

The detailed *execution times* rounded to whole milliseconds on SHARE for all provided models are depicted in Table2. The largest provided model consisting of 100000 elements[7] can be processed in about six seconds which is more than a thousand times faster than the reference UML-RSDS solution. It can be seen that the additional multiple inheritance rule has only little influence on the execution times.

Provided model	**Core task only**	**Core and Extension task**
testcase1	1 ms	1 ms
testcase2	1 ms	1 ms
testcase2_1000	418 ms	434 ms
testcase2_5000	2455 ms	2585 ms
testcase2_10000	5656 ms	6041 ms
testcase3	248 ms	268 ms
mitest1	1 ms	1 ms
mitest2	1 ms	1 ms
mitest3	1 ms	1 ms
Additional model		
testcase2_50000	75238 ms	76835 ms
testcase2_100000	262917 ms	264483 ms
testcase2_200000	1006848 ms	1045648 ms

Table 2: Detailed execution times on SHARE

To determine the *maximum capability* of the solution, the models `testcase2_50000`, `testcase2_100000`, and `testcase2_200000` consisting of half a million, one million, and

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

two million elements, respectively, have been generated. Transforming `testcase2_50000` runs slightly more than one minute, `testcase2_100000` runs in about five minutes, and `testcase2_200000` runs in about 17 minutes. Given the limited amount of 800 MB memory available to the JVM process on SHARE, the model with 2 million elements is approximately the maximum capability for the FunnyQT solution.

References

[BGJ13] Jakob Blomer, Rubino Geiß, and Edgar Jakumeit. *The GrGen.NET User Manual*. Institute for Programme Structures and Data Organisation, Department of Informatics, University Karlsruhe, January 2013.

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

[Gra93] Paul Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Inc., 1993.

[Hoy08] Doug Hoyte. *Let Over Lambda*. Lulu.com, April 2008. `http://letoverlambda.com`.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[KRP13] Dimitrios Kolovos, Louis Rose, and Richard Paige. The Epsilon Book. `http://www.eclipse.org/epsilon/doc/book/`, January 2013.

[LKR13] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Case study: Class diagram restructuring. `http://planet-sl.org/ttc2013/images/userdirs/618/ttc2013_classdiagram_case.pdf`, 2013.

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