Solving the Petri-Nets to Statecharts Transformation Case with FunnyQT

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 Petri-Nets to Statcharts Transformation Case. This solution has won the *best overall solution award* and the *best efficiency award* for this case.

1 Introduction

FunnyQT[1] 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[2], FunnyQT's model transformation API has been used for the initialization transformation, while the reduction transformation has been tackled algorithmically using the plain querying and model manipulation APIs. This solution has won the *best overall solution award* and the *best efficiency award* for this case.

2 The Initialization Transformation

The initialization transformation using FunnyQT's model transformation API is shown in Listing 2. This API provides an internal DSL [1] for model-to-model transformations similar to ATL [2] or ETL [3].

A transformation is declared with the `deftransformation` macro. It receives the name of the transformation, i.e., `initialize-statechart`, a vector of input and output models, and arbitrarily many rules. Here, the argument vector declares that the transformation receives one single input model `pn` which is an EMF model, and it receives exactly one output model `sc` which is also an EMF model. It could also receive many input and output models, and those could belong to different modeling frameworks as well.

The transformation consists of two rules: `place2basic-and-or`, and `transition2hyperedge`. The former receives an input `Place` and creates an `OR` and a `Basic` in the output model. It also sets the new basic's name to the name of the place and assigns it as content of the new `OR`. Finally, it sets the basic's `rnext` and `next` references to the value of applying `transition2hyperedge` to any pre-transition or post-transition of the input place[3]. `place2basic-and-or` is a top-level rule meaning it's applied automatically

[1] The FunnyQT homepage: `https://github.com/jgralab/funnyqt`

[2] This FunnyQT solution is available at `https://github.com/tsdh/ttc-2013-pn2sc` and on SHARE (image `TTC13::Ubuntu12LTS_TTC13::FunnyQT.vdi`)

[3] `map` takes a function f and a collection c. It applies f to the items of c returning the sequence of results.

Pieter Van Gorp, Louis M. Rose, Christian Krause (Eds.):
Sixth Transformation Tool Contest (TTC 2013)
EPTCS 135, 2013, pp. 88–94, doi:10.4204/EPTCS.135.11

to all matching elements while the non-top-level rule `transition2hyperedge` has to be called explicitly from other rules.

```
(deftransformation initialize-statechart [[pn :emf] [sc :emf]]
  (^:top place2basic-and-or [p]
         :from 'Place
         :to [o 'OR, b 'Basic]
         (eset! b :name (eget p :name))
         (eset! b :rcontains o)
         (eset! b :rnext (map transition2hyperedge (eget p :pret)))
         (eset! b :next  (map transition2hyperedge (eget p :postt))))
  (transition2hyperedge [t]
         :from 'Transition
         :to [he 'HyperEdge]
         (eset! he :name (eget t :name))))
```

Listing 1: The initialization transformation

When a rule gets called and is applicable with respect to its declared `:from` type (and optional `:when` constraint), it creates the elements declared in `:to` in the target model, and evaluates its body. In case there is just one new element declared in `:to`, it returns just that. If there are many new elements, it returns them as a vector in their declaration order. Furthermore, a traceability mapping is created from the source element to the rule's return value. If a rule gets called multiple times for a single element, the second and all following calls just return the result of the first invocation.

3 The Reduction Transformation

The reduction transformation is implemented algorithmically based on FunnyQT's querying and model manipulation APIs. It consists of four rules (functions):

1. The AND-rule as discussed in the case description [4],
2. the OR-rule as discussed in the case description,
3. an additional, extension rule assigning hyperedges to the nearest `Compound` state containing all their predecessor and successor `Basic` states,
4. and a rule creating a `Statechart` with an `AND` top-state if the reduction could be completed successfully.

In this section, only the AND- and OR-rule are discussed. The main reduction function simply applies them as long as possible, then invokes the hyperedge assignment rule followed by the statechart creating rule. The complete reduction transformation is printed in Appendix B.

Reduction Helper Functions. Before discussing the rules, some helper functions need to be introduced. Those are `pret` and `postt` returning the sets of pre-/post-transitions for a given place. Likewise, `prep` and `postp` return the sets of pre-/post-places for a given transition.

The AND Rule. The AND rule is depicted in Listing 3. In contrast to the Figure 2 in the case description [4], it doesn't delete all places q_1 to q_n to create a new place p, but instead it reuses q_1 as p and deletes only q_2 to q_n, which is consistent with Louis Rose's EOL solution.

The rule function receives the source Petri-net model `pn`, the target statechart model `sc`, either the function `prep` or `postp` as `prep-or-postp`, and the traceability map `place2or` gathered from the initialization transformation mapping input places to output `OR` states wrapped in a Clojure atom[4].

```
1  (defn and-rule [pn sc prep-or-postp place2or]
2    (loop [ts (eallobjects pn 'Transition), applied false]
3      (if (seq ts)
4        (let [t (first ts), preps-or-postps (prep-or-postp t)]
5          (if (> (count preps-or-postps) 1)
6            (let [p (first preps-or-postps), prets (pret p), postts (postt p)]
7              (if (forall? #(and (= prets  (pret %))
8                                 (= postts (postt %)))
9                           (rest preps-or-postps))
10               (let [new-or  (ecreate! sc 'OR), new-and (ecreate! sc 'AND)]
11                 (eset! new-and :contains (mapv @place2or preps-or-postps))
12                 (eadd! new-or  :contains new-and)
13                 (swap! place2or assoc p new-or)
14                 (doseq [op (rest preps-or-postps)]
15                   (edelete! op))
16                 (recur (rest ts) true))
17               (recur (rest ts) applied)))
18           (recur (rest ts) applied)))
19       applied)))
```

Listing 2: The AND rule

The rule iterates over all transitions[5] in the petri-net model `pn` using a local tail-recursion (`loop` and `recur`). Lines 5 to 9 check the preconditions of the rule: If the transition `t` has more than one pre-/post-place, all of them must have the same set of pre- and post-transitions. If that's the case, a new `AND` and a new `OR` state is created. The new `AND` contains all existing `OR` states being the pre- or post-places of the transition `t`, and the new `OR` contains the new `AND`. Furthermore, the traceability map atom is updated in line 13 so that the first pre-/post-place `p` now maps to the new `OR`. Lastly, all other pre-/post-places are deleted[6].

The OR Rule. The OR rule is depicted in Listing 3. In contrast to the case description, it doesn't delete the places (or corresponding OR states) q and r to create a new place (or corresponding OR state) p, but instead it reuses q as p and only deletes r.

The `or-rule` gets the Petri-net model `pn`, the statechart model `sc`, and the traceability map atom `place2or`.

Like the AND-rule, it iterates all transitions in the petri-net model. Lines 5 to 9 check the preconditions of the rule: If the transition `t` has exactly one pre-place `q` and one post-place `r`, and if `q` and `r` are identical or `q` and `r` are not connected by other transitions, then the rule matches. In that case, the `OR` corresponding to `r` is merged with the `OR` corresponding to `q`, and the transition `t` is deleted.

[4] All Clojure data structures are immutable. An atom is a mutable reference to some immutable data structure that can be swapped atomically. This is important here in order to update the traceability mapping when the AND-rule matches.

[5] `seq` takes a collection and returns a sequential view on it or `nil` if the collection is empty. Therefore, it is the canonical non-emptiness check in Clojure.

[6] `doseq` is equivalent to Java's extended `for` loop.

```
(defn or-rule [pn sc place2or]
  (loop [ts (vec (eallobjects pn 'Transition)), applied false]
    (if (seq ts)
      (let [t (first ts), preps (prep t), postps (postp t)]
        (if (= 1 (count preps) (count postps))
          (let [q (first preps), r (first postps)]
            (if (or (identical? q r)
                    (and (not (member? r (adjs q :pret :postp)))
                         (not (member? r (adjs q :postt :prep)))))
              (let [merger (@place2or q), mergee (@place2or r)]
                (when-not (identical? q r)
                  (eaddall! q :pret  (eget-raw r :pret))
                  (eaddall! q :postt (eget-raw r :postt))
                  (edelete! r)
                  (eaddall! merger :contains (eget-raw mergee :contains))
                  (edelete! mergee))
                (edelete! t)
                (recur (rest ts) true))
              (recur (rest ts) applied)))
          (recur (rest ts) applied)))
      applied)))
```

Listing 3: The OR rule

Extensions. Two extensions were implemented for this task. Firstly, there is an additional rule `assign-hyperedges` that assigns each hyperedge to the nearest compound state which contains all basic states connected by the hyperedge. Secondly, a validation tool[7] has been implemented that uses FunnyQT to check result statechart models against their expected outcome in terms of a very detailed unit test suite.

4 Evaluation

In this section, the solution is evaluated according to the evaluation criteria listed in the case description [4].

Transformation correctness. The validation project that has been implemented as an extension to this case allows for testing the result statechart models. For the main test cases, every important aspect of the result models including the containment hierarchy and the predecessors and successors of hyperedges are checked, and for the performance test cases, only the number of instances of every metamodel class is checked. All tests pass for the result models of this solution. Similarly, all tests pass for the result models created by the reference GrGen.NET solution.

The validation project has also been tested with intentionally slightly wrong models, e.g., some `next` link is missing at some hyperedge, there's some additional element, or an element is contained by the wrong `Compound` state. In all those cases, an assertion of the validation project failed. So there's a high confidence that if the result models pass the tests, the transformation producing them is correct.

[7] `https://github.com/tsdh/ttc-2013-pn2sc-validation`

Transformation performance. This FunnyQT solution is by far the most efficient of all submitted solutions, especially for large models, so it has won the *best efficiency award* for this case. For the performance test models `sp5000` and `sp10000` the complete transformation (initialization and reduction) takes about one and two seconds, which is about as fast as the second fastest solution. But with the `sp40000`/`sp200000` model, the FunnyQT solution is already three/twelve times faster than the second fastest solution, taking 11 and 114 seconds, respectively.

Transformation understandability. Although the solution requires some understanding of Clojure, it shouldn't be hard to get a grasp on it. The initialization transformation uses a FunnyQT facility allowing to specify typical model transformations with a syntax and semantics similar to ATL or ETL, so people knowing these languages should feel right at home.

The reduction transformation is a bit more complex, but the application conditions of the rules and the actions that are performed are taken quite literally from the case description with the exception that some elements are preserved and merged instead of replaced.

One important aspect with respect to understandability is also the fact that the transformations are very concise. In total, the initialization and the reduction transformation are only 96 lines of code.

Bonus criteria. The bonus tasks dealing with *verification*, *simulation support*, *change propagation*, and *reversing the transformation* haven't been tackled.

Proper *debugging support* is also not yet ready for prime-time in the Clojure world. There are some attempts at debuggers allowing to set breakpoints and examine the lexical extent around the breakpoint, but those are not too usable right now. Another difficulty with functional languages involving some kind of laziness is that errors might be signaled at a location very different to where the bug is actually manifested in the source code. Nevertheless, FunnyQT has rather good model visualization tools that have been used while programming the reduction rules in order to visualize the matching elements when a rule has been applicable.

References

[1] Martin Fowler (2010): *Domain-Specific Languages*. Addison-Wesley Professional.

[2] Frédéric Jouault & Ivan Kurtev (2005): *Transforming Models with ATL*. In Jean-Michel Bruel, editor: *MoDELS Satellite Events*, *Lecture Notes in Computer Science* 3844, Springer, pp. 128–138, doi:10.1007/11663430_14.

[3] Dimitrios Kolovos, Louis Rose & Richard Paige (2013): *The Epsilon Book*. `http://www.eclipse.org/epsilon/doc/book/`.

[4] Pieter Van Gorp & Louis Rose (2013): *The Petri-Nets to Statecharts Transformation Case*. In Pieter Van Gorp, Louis Rose & Christian Krause, editors: *Sixth Transformation Tool Contest (TTC 2013)*, *EPTCS*, this volume.

A The complete Initialization Transformation

```
(deftransformation initialize-statechart [[pn :emf] [sc :emf]]
  (^:top place2basic-and-or [p]
         :from 'Place
         :to [o 'OR, b 'Basic]
         (eset! b :name (eget p :name))
         (eset! b :rcontains o)
         (eset! b :rnext (map transition2hyperedge (eget p :pret)))
         (eset! b :next  (map transition2hyperedge (eget p :postt))))
  (transition2hyperedge [t]
         :from 'Transition
         :to [he 'HyperEdge]
         (eset! he :name (eget t :name))))

(defn init-statechart [pn]
  (let [sc (new-model)
        trace (initialize-statechart pn sc)]
    [sc
     (apply hash-map (mapcat (fn [[p [o b]]] [p o])
                             (:place2basic-and-or trace)))
     (apply hash-map (mapcat (fn [[p [o b]]] [p b])
                             (:place2basic-and-or trace)))
     (:transition2hyperedge trace)]))
```

B The complete Reduction Transformation

```
(defn refs-as-set [ref elem]
  (set (eget-raw elem ref)))

(def postt (partial refs-as-set :postt))
(def pret  (partial refs-as-set :pret))
(def postp (partial refs-as-set :postp))
(def prep  (partial refs-as-set :prep))

(defn and-rule [pn sc prep-or-postp place2or]
  (loop [ts (eallobjects pn 'Transition), applied false]
    (if (seq ts)
      (let [t (first ts), preps-or-postps (prep-or-postp t)]
        (if (> (count preps-or-postps) 1)
          (let [p (first preps-or-postps), prets (pret p), postts (postt p)]
            (if (forall? #(and (= prets  (pret %))
                               (= postts (postt %)))
                         (rest preps-or-postps))
              (let [new-or  (ecreate! sc 'OR), new-and (ecreate! sc 'AND)]
                (eset! new-and :contains (mapv @place2or preps-or-postps))
                (eadd! new-or  :contains new-and)
                (swap! place2or assoc p new-or)
                (doseq [op (rest preps-or-postps)]
                  (edelete! op))
                (recur (rest ts) true))
              (recur (rest ts) applied)))
          (recur (rest ts) applied)))
      applied)))

```

```
(defn or-rule [pn sc place2or]
  (loop [ts (vec (eallobjects pn 'Transition)), applied false]
    (if (seq ts)
      (let [t (first ts), preps (prep t), postps (postp t)]
        (if (= 1 (count preps) (count postps))
          (let [q (first preps), r (first postps)]
            (if (or (identical? q r)
                    (and (not (member? r (adjs q :pret :postp)))
                         (not (member? r (adjs q :postt :prep)))))
              (let [merger (@place2or q), mergee (@place2or r)]
                (when-not (identical? q r)
                  (eaddall! q :pret  (eget-raw r :pret))
                  (eaddall! q :postt (eget-raw r :postt))
                  (edelete! r)
                  (eaddall! merger :contains (eget-raw mergee :contains))
                  (edelete! mergee))
                (edelete! t)
                (recur (rest ts) true))
              (recur (rest ts) applied)))
          (recur (rest ts) applied)))
      applied)))

(defn assign-hyperedges [sc]
  (doseq [e (eallobjects sc 'HyperEdge)]
    (eset! e :rcontains
           (first (apply clojure.set/intersection
                         (map #(reachables % [p-+ --<>])
                              (concat (eget e :next) (eget e :rnext))))))))

(defn create-top [sc]
  (let [top-ors (filter #(not (eget % :rcontains)) (eallobjects sc 'OR))]
    (when (= 1 (count top-ors))
      (let [statechart (ecreate! sc 'Statechart), top (ecreate! sc 'AND)]
        (eset! statechart :topState top)
        (eset! top :contains top-ors)))))

(defn create-statechart [pn]
  (let [[sc place2or _ _] (init/init-statechart pn)
        place2or (atom place2or)]
    (iteratively (fn []
                   (let [r     (and-rule pn sc prep  place2or)
                         r (or (and-rule pn sc postp place2or) r)
                         r (or (or-rule  pn sc place2or)       r)]
                     r)))
    (create-top sc)
    (assign-hyperedges sc)
    sc))
```