Solving the Petri-Nets to Statecharts 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 Petri-Nets to Statcharts 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.

For solving this case, the 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.

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 meta-models. 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, FunnyQT's model transformation API has been used for the initialization transformation discussed in Section 2, while the reduction transformation explained in Section 3 has been tackled algorithmically using the plain querying and model manipulation APIs.

2 The Initialization Transformation

The complete initialization transformation is shown in Listing 2. It uses FunnyQT's model transformation API.

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 arbitrary many rules.

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 or output models, and the models could belong to different modeling frameworks as well.

The transformation consists of two transformation rules: `place2basic-and-or`, and `transition2hyperedge`. Every rule has exactly one parameter denoting the source element. The `:from` clause defines the metamodel type the rule is applicable for. The `:to` clause specifies which target elements have to be created. Additionally, arbitrary constraints could be defined for a rule using a `:when` clause. Thereafter, arbitrary code may follow for initializing the newly created elements and calling other rules.

When a rule gets called and is applicable with respect to its declared `:from` type and `: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.

```
1  (deftransformation initialize-statechart [[pn :emf] [sc :emf]]
2    (^:top place2basic-and-or [p]
3           :from 'Place
4           :to [o 'OR, b 'Basic]
5           (eset! b :name (eget p :name))
6           (eset! b :rcontains o)
7           (eset! b :rnext (map transition2hyperedge
8                                (eget p :pret)))
9           (eset! b :next  (map transition2hyperedge
10                               (eget p :postt))))
11   (transition2hyperedge [t]
12          :from 'Transition
13          :to [he 'HyperEdge]
14          (eset! he :name (eget t :name))))
```

Listing 1: The initialization transformation

The `place2basic-and-or` rule creates an `OR o` and a `Basic b` for any `Place p` it is called with, it sets the name of the `Basic` to the name of the `Place`, and assigns the new `OR` as container of `b`. Lastly, it maps the `transition2hyperedge` rule over the pre/post-transitions of `p` setting the results to `b`'s `rnext`/`next` references.

The `^:top` metadata at the rule specifies that this rule is called automatically for any `Place` in the source Petri-net model `pn`. In contrast, the `transition2hyperedge` rule is only called explicitly from `place2basic-and-or`.

The result of such a transformation is always a map of traceability information. The keys of the map are keywords denoting the rules, the values are maps from source elements to rule results. So in this concrete case, the traceability map returned by the transformation has the form:

```
{:place2basic-and-or   {<place1> [<or1> <basic1>], ...},
 :transition2hyperedge {<transition1> <hyperedge1>, ...}}
```

The function `init-statechart` depicted in Listing 2 is a convenience wrapper for applying the transformation.

```
15 (defn init-statechart [pn]
16   (let [sc (new-model)
17         trace (initialize-statechart pn sc)]
18     [sc
19      (apply hash-map (mapcat (fn [[p [o b]]] [p o])
20                              (:place2basic-and-or trace)))
21      (apply hash-map (mapcat (fn [[p [o b]]] [p b])
22                              (:place2basic-and-or trace)))
23      (:transition2hyperedge trace)]))
```

Listing 2: An initialization transformation wrapper function

It receives a Petri-net model `pn`, creates a new empty model `sc` for the statechart, calls the `initialize-statechart` transformation, and then mangles the traceability map in order to return a vector of four components: the initialized statechart model, a map from places to corresponding `OR` elements, a map from places to corresponding `Basic` elements, and a map from transitions to corresponding hyperedges. The first map is required by the reduction transformation, whereas the other two maps are only used by the unit-tests for the initialization transformation (see Section 4).

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 [vGR13],
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.

3.1 The Reduction Function

The reduction transformation (a plain function) is shown in Listing 3.1.

It receives a Petri-net model `pn` and calls the statechart initialization function from Listing 2. As discussed above, this function returns a vector containing the new statechart and three traceability maps. By imitating the result structure in the `let`, the new statechart is assigned to `sc`, and the map from places to OR objects is assigned to `place2or`. The other two maps are not needed for the reduction, so they are assigned to a variable _, which is conventionally used as "don't care". This technique of binding parts of the contents collections directly by imitating their structure is known as *destructuring* in the Lisp-world.

All Clojure datastructures are immutable. However, the transformation rules will need to modify the `place2or` traceability map. Therefore, it is wrapped in and atom. An atom is a mutable reference that can be swapped to a new value atomically.

```
24 (defn create-statechart [pn]
25   (let [[sc place2or _ _] (init/init-statechart pn)
26         place2or (atom place2or)]
27     (iteratively (fn []
28                    (let [r     (and-rule pn sc prep  place2or)
29                          r (or (and-rule pn sc postp place2or) r)
30                          r (or (or-rule  pn sc place2or)       r)]
31                      r)))
32     (create-top sc)
33     (assign-hyperedges sc)
34     sc))
```

Listing 3: The reduction function implementing the transformation

Starting with line 27, the rules are applied. The higher-order function `iteratively` takes a function and applies it as long as it returns logically true[3]

The anonymous function it is called with applies the `and-rule` once for pre-places and once for post-places, and finally it calls the `or-rule`. The results of the rules are combined using disjunction in such a way that all rules get applied in that sequence and the final result `r` is logical true if at least one rule could be applied.

Lastly, the `create-top` rule creating a `Statechart` element and its top-AND-State, and the `assign-hyperedges` rule are applied. The final result is the new statechart `sc`.

3.2 Reduction Helper Functions

Before discussing the rules, the helper functions depicted in Listing 3.2 are explained.

```
35 (defn refs-as-set [ref elem]
36   (set (eget-raw elem ref)))
37
38 (def postt (partial refs-as-set :postt))
39 (def pret  (partial refs-as-set :pret))
40 (def postp (partial refs-as-set :postp))
41 (def prep  (partial refs-as-set :prep))
```

Listing 4: Helper functions for the reduction rules

The function `refs-as-set` gets a (multi-valued) reference name as a keyword and some model element, and it returns the value of this reference coerced to a set[4].

Then, there are four partial applications of this function where its first parameter is already preset, thus leaving functions that just receive an element. That is, `postt` and `pret` are functions for getting the set of post- and pre-transitions of a given `Place`, and `postp` and `prep` are functions for getting the set of post- and pre-places of a given `Transition`.

3.3 The AND Rule

The AND rule is depicted in Listing 3.3. In contrast to the Figure 2 in the case description [vGR13], it doesn't delete all places q_1 to q_n to create a new place p, but instead it reuses q_1 as

[3] In Clojure, everything is logically true except for false and nil.

[4] eget-raw is similar to eget, except that the former just returns the EMF collection, i.e., a mutable EList, wheras the latter also coerces to an immutable Clojure collections. Because we are coercing to an immutable Clojure set anyway, this is unneeded effort here.

p and deletes only q_2 to q_n. This is consistent with Louis Rose's EOL solution, and it even feels more natural at least for algorithmic solutions.

The rule function receives the Petri-net model `pn`, the statechart model `sc`, either the function `prep` or `postp` as `prep-or-postp`, and the traceability map atom `place2or`.

```
42 (defn and-rule [pn sc prep-or-postp place2or]
43   (loop [ts (eallobjects pn 'Transition), applied false]
44     (if (seq ts)
45       (let [t (first ts), preps-or-postps (prep-or-postp t)]
46         (if (> (count preps-or-postps) 1)
47           (let [p (first preps-or-postps), prets (pret p), postts (postt p)]
48             (if (forall? #(and (= prets  (pret %))
49                                (= postts (postt %)))
50                           (rest preps-or-postps))
51               (let [new-or  (ecreate! sc 'OR), new-and (ecreate! sc 'AND)]
52                 (eset! new-and :contains (mapv @place2or preps-or-postps))
53                 (eadd! new-or  :contains new-and)
54                 (swap! place2or assoc p new-or)
55                 (doseq [op (rest preps-or-postps)]
56                   (edelete! op))
57                 (recur (rest ts) true))
58               (recur (rest ts) applied)))
59           (recur (rest ts) applied)))
60       applied)))
```

Listing 5: The AND rule

It uses `loop` and `recur` which implement a local tail-recursion, i.e., a recursion that doesn't consume space on the call-stack. `loop` defines the initial bindings, and `recur` restarts the loop with new bindings. So initially, `ts` is bound to the sequence of all transitions in the model, and `applied`, which is used to indicate to the caller if at least one match has been found, is `false`.

If there are no transitions left[5] (the else-branch of the `if` in line 44), the function returns with the current value of `applied`. If there are still transitions, the first one is bound to `t` and its set of pre- or post-places is bound to `preps-or-posts` (line 45).

If there aren't more than one pre- or post-places of `t` (the else branch of the `if` in line 46), the `loop` gets restarted with the rest of transitions and the current value of `applied`. If there are more than one pre- or post-places, the first one is bound to `p`, and `prets` and `postts` are its pre- and post-transitions (line 47).

If the other pre- or post-places don't have the same sets of pre- and post-transitions (the else-branch of the `if` in line 48), the `loop` is restarted with the remaining transitions and the current value of `applied`. However, if the pre- and post-transitition sets are all equal, the rule matches. In that case, lines 51 to 56 create a new `OR` and a new `AND` where the `OR` contains the `AND`, and the `AND` contains all the `OR` states corresponding to the pre- or post-places of the current transition `t`[6]. The first place `p` is preserved and its traceability mapping is updated to point to the new `OR` (line 54). The other places are deleted in lines 55 and 56. Finally, the `loop` is restarted with the remaining transitions and an `applied` value of `true`.

3.4 The OR Rule

The OR rule is depicted in Listing 3.4. 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`. It's mechanics for searching for matches in terms of `loop` and `recur` are almost identical to the `and-rule`, so the rule is described a bit more concisely here. One minor

[5] (seq coll) is the canonical non-emptiness check in Clojure.

[6] @some-atom atomically dereferences an atom resulting in its current value.

difference is that the variable `ts` is initially bound to a vector of all transitions. `eallobjects` returns a lazy sequence, that is, a sequence where elements are computed (*realized*) when they are consumed. Since this rule deletes transitions, the fail-fast EMF model iterator underlying the lazy sequence will break. The explicit conversion to a vector forces that all transitions are computed beforehand.

```
61 (defn or-rule [pn sc place2or]
62   (loop [ts (vec (eallobjects pn 'Transition)), applied false]
63     (if (seq ts)
64       (let [t (first ts), preps (prep t), postps (postp t)]
65         (if (= 1 (count preps) (count postps))
66           (let [q (first preps), r (first postps)]
67             (if (and (not (member? r (adjs q :pret :postp)))
68                      (not (member? r (adjs q :postt :prep))))
69               (let [merger (@place2or q), mergee (@place2or r)]
70                 (edelete! t)
71                 (eaddall! q :pret  (eget-raw r :pret))
72                 (eaddall! q :postt (eget-raw r :postt))
73                 (edelete! r)
74                 (eaddall! merger :contains (eget-raw mergee :contains))
75                 (edelete! mergee)
76                 (recur (rest ts) true))
77               (recur (rest ts) applied)))
78           (recur (rest ts) applied)))
79       applied)))
```

Listing 6: The OR rule

Lines 63 to 68 specify the application condition of the rule. If the pre- and post-place sets of the current transition `t` both contain only a single place `q` and `r`, respectively, and if `r` is neither reachable from `q` by traversing the `pret` reference followed by the `postp` reference, nor reachable from `q` by traversing the `postt` followed by the `prep` reference, then `t` is a matching transition. In that case, the transition `t` is deleted. `r`'s `pret` and `postt` references are merged into `q`, and `r` is deleted. Similarly, the OR states corresponding to `q` (`merger`) and `r` (`mergee`) are merged, i.e., the contents of `mergee` are transferred to `merger`, and `mergee` is deleted. Finally, the loop is restarted with the remaining transitions and an `applied` value of `true`.

3.5 Extension: The HyperEdge Assignment Rule

The hyperedge assignment rule is shown in Listing 3.5. It assigns each `HyperEdge` in the target statechart model to the `Compound` state that contains all its `rnext` and `next` states.

The rule receives the statechart model `sc`.

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

Listing 7: The hyperedge assignment rule

It iterates over all `HypedEdge` elements in the model, and for each hyperedge `e`, it sets its `rcontains` reference. The container element is determined as follows.

For every `Basic` state in `e`'s `next` and `rnext` references, the ordered set of containers is computed using a *regular path expression*. The `reachables` function gets the start element of the search, and a vector describing the path expression. It returns the ordered set of elements reachable by a path matching the regular path expression. Here, the `--<>` defines that elements

should be traversed towards their container, and by wrapping it in `[p-+ ...]` this iteration may take place one or many times (transitive closure). Thus, the ordered result set contains all containers in the order from nearest to farthest in the containment hierarchy. The intersection of all those ordered sets is an ordered set of all `Compound` states containing all predecessor and successor `Basic` states of `e`, and its first element is the deepest one in the overall containment hierarchy.

If the reduction rules didn't terminate with exactly one `Place` left and one single top-level `OR` state, as it happens with some of the test models, the intersection above is empty. In that case, the `rcontains` reference is set to `nil`, i.e., it is left unset.

3.6 The Statechart Creation Rule

The final rule of the transformation is the `create-top` rule shown in Listing 3.6. It receives the target statechart model `sc` as argument.

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

Listing 8: The statechart creation rule

If there is exactly one `OR` state that's not contained in some other `Compound` state, the reduction process has been applied successfully. In that case, a new `Statechart` element `statechart` and a new `AND` state `top` are created. `top` is set as `topState` of the `statechart`, and the contents of `top` are set to the single top-level `OR` state.

4 Extension: Validation

As an extension to the case (in addition to the hyperedge assignment rule discussed in Section 3.5), a small testing project for validating the result statechart models of this transformation case has been implemented. It is published at github[7] where also its usage documentation can be found.

This project is capable of testing the correctness of the result statecharts of provided eleven Petri-nets, and it also checks the result statecharts generated from the 15 performance testing Petri-nets.

The following constraints are checked for the primary test cases:

1. For the test cases where a complete reduction is feasible, there has to be exactly one `Statechart` element containing exactly one `AND` state containing the top-most `OR` state created by the reduction rules.
2. For every element type in the statechart metamodel, the expected number of instances is checked against the actual number of instances.
3. The expected containment hierarchy is checked against the actual containment hierarchy. If at least one hyperedge is contained in some `Compound` state, the correct containments of hyperedges is also tested. Else, a warning is issued referring to the hyperedge containment extension.
4. The expected contents of the `rnext` and `next` references of each `HyperEdge` are checked against their actual contents.

For the large results of the performance test cases, only the checks 1 and 2 are performed.

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

5 Running the Transformation on SHARE

The FunnyQT solution of 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 Petri-Nets to Statecharts project:
   ```
   $ cd ~/Desktop/FunnyQT_Solutions/ttc-2013-pn2sc/
   ```
3. Run the test cases:
   ```
   $ lein test
   ```

This will run the complete transformation (initialization + reduction) on all provided test Petri-net models (the 11 main test cases and the performance test cases) and print the execution times. The result models are also validated using the testing project discussed in Section 4. The result models and visualizations of the main test cases' results are saved to the `results` directory.

Furthermore, the stand-alone initialization transformation is applied to every provided model as well. Again, the times needed to apply the transformation are printed.

6 Evaluation

In this section, the solution is evaluated according to the evaluation criteria listed in the case description [vGR13].

Transformation correctness. The validation project discussed in Section 4 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.

Transformation performance. Table 1 shows the evaluation times of the initialization tranformation, the reduction transformation, and the complete transformation involving initialization and reduction. The times needed for loading and saving the models from/to XMI files, the times needed for validation, and the times needed for creating visualizations are excluded.

The measurements have been done directly on the SHARE demo. As stated in Section 5, `lein test` will first evaluate the complete transformation on all models, and then it'll evaluate the initialization transformation again on all models. The evaluation times printed there are contained in the tables third and first column, respectively. The second column is the difference between the value in the third column and the value in the first column.

Test Case	Init. only	Red. only	Init. & Red.
1	1 ms	81 ms	82 ms
2	1 ms	37 ms	38 ms
3	<1 ms	23 ms	24 ms
4	<1 ms	9 ms	10 ms
5	<1 ms	2 ms	3 ms
6	<1 ms	2 ms	3 ms
7	<1 ms	15 ms	16 ms
8	<1 ms	20 ms	21 ms
9	<1 ms	7 ms	8 ms
10	<1 ms	7 ms	8 ms
11	<1 ms	9 ms	10 ms
sp200	12 ms	140 ms	152 ms
sp300	18 ms	92 ms	110 ms
sp400	24 ms	97 ms	121 ms
sp500	28 ms	107 ms	125 ms
sp1000	69 ms	223 ms	292 ms
sp2000	109 ms	267 ms	376 ms
sp3000	153 ms	448 ms	601 ms
sp4000	226 ms	517 ms	743 ms
sp5000	274 ms	743 ms	1017 ms
sp10000	553 ms	1509 ms	2062 ms
sp20000	960 ms	3652 ms	4612 ms
sp40000	1887 ms	9457 ms	11344 ms
sp80000	3849 ms	26442 ms	30291 ms
sp100000	4755 ms	35639 ms	40394 ms
sp200000	9407 ms	105257 ms	114664 ms

Table 1: Evaluation times on SHARE

That the first test cases evaluate a bit slower than subsequent ones is probably caused by the JVM just-in-time compiling critical code paths. Fluctuations in the evaluation times of models of comparable size are also partly caused by JVM-internals such as garbage collection. Nevertheless, the order of magnitude is stable across multiple runs of the transformations.

Anyhow, the evaluation times are quite good, both absolute as well as relative compared to the input model sizes. The initialization transformation scales linearly with the size of the input Petri-net models. The reduction transformation also scales in that order of magnitude. When taking the time needed for the `sp1000` model as a baseline, then the `sp10000` model is transformed in $0.7n$ milliseconds for n being the time expected when assuming a linear correlation between model size and execution time. The `sp40000` model takes n milliseconds, the `sp100000` takes $1.6n$ milliseconds, and the `sp200000` model takes $2.4n$ milliseconds.

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 need just the 91 lines of code that was depicted completely in the previous sections. The initialization transformation including its wrapper function mangling the traceability mappings is just 23 lines of

code. The complete reduction transformation with its four rules, the few helper functions, and the function iteratively applying the rules as long as possible amounts to 68 lines of code.

Bonus criteria. The bonus tasks dealing with *verification* and *simulation support* haven't been tackled.

The initialization transformation could be extended quite easily to deal with the *change propagation* scenarios, simply because FunnyQT transformations are written in the full-fledged programming language Clojure having access to any JVM library's features. Thus, adapters performing the required changes on the target statechart could be registered to handle notifications about new, deleted, or updated elements in the source Petri-net using the standart EMF notification framework. Since the initialization transformation created a complete traceability mapping, referring to previously created elements is easy.

There is no support special support for *reversing the transformation*, or for defining the transformation bidirectionally in the first place.

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

[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.

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

[vGR13] Pieter van Gorp and Louis M. Rose. The petri-nets to statecharts transformation case. `http://planet-sl.org/ttc2013/images/userdirs/122/ttc2013/pn2sc.pdf`, 2013.