Solving the TTC Model Execution Case with FunnyQT

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

This paper describes the FunnyQT solution to the TTC 2015 Model Execution transformation case. The solution solves the third variant of the case, i.e., it considers and implements the execution semantics of the complete UML Activity Diagram language. The solution won the *most correct solution award*.

FunnyQT is a model querying and model transformation library for the functional Lisp-dialect Clojure providing a comprehensive and efficient querying and transformation API, many parts of which are provided as task-oriented embedded DSLs.

1 Introduction

This paper describes the FunnyQT[1] [1, 2] solution of the TTC 2015 Model Execution Case [3]. It implements the third variant of the case description, i.e., it implements the execution semantics of the complete UML Activity Diagram language. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3]. The solution has won the *most correct solution award*.

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. Queries and transformations are Clojure programs using the features provided by the FunnyQT API.

Clojure provides strong metaprogramming capabilities that are used by FunnyQT in order to define several *embedded domain-specific languages* (DSL) for different querying and transformation tasks.

FunnyQT is designed with extensibility in mind. By default, it supports EMF models and JGraLab TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into several namespaces, each namespace providing constructs supporting concrete querying and transformation use-cases, e.g., model management, functional querying, polymorphic functions, relational querying, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and some more. For solving the model execution case, only the model management, the functional querying, and the polymorphic functions APIs have been used.

2 Solution Description

The explanations in the case description about the operational semantics on UML Activity Diagrams suggest an algorithmic solution to the transformation case. The FunnyQT solution tries to be almost a literal translation of the case description to Clojure code.

[1] `http://funnyqt.org`
[2] `https://github.com/tsdh/ttc15-model-execution-funnyqt`
[3] The SHARE image name is `ArchLinux64_TTC15-FunnyQT_2`
[4] `http://clojure.org`

Submitted to:
TTC 2015

FunnyQT is able to generate metamodel-specific model management APIs. This feature has been used for solving this case. The generated API consists of element creation functions, lazy element sequence functions, attribute access functions, and reference access functions. For example, `(a/create-ControlToken! ad)` creates a new control token and adds it to the activity diagram model `ad`, `(a/isa-Token? x)` returns true if and only if `x` is a token, `(a/all-Inputs ad)` returns the lazy sequence of input elements in `ad`, `(a/running? n)` and `(a/set-running! n true)` query and set the node `n`'s running attribute, and `(a/->locals a)`, `(a/->set-locals! a ls)`, `(a/->add-locals! a l)`, and `(a/->remove-locals! a l)` query, set, add to, and remove from the locals reference of the activity `a`.

In the following, the solution is presented in a top-down manner similar to how the case description defines the operational semantics of activity diagrams. The following listing shows the function `execute-activity-diagram` which contains the transformation's main loop.

```
1  (defn execute-activity-diagram [ad]
2    (let [activity (the (a/all-Activities ad))
3          trace (a/create-Trace! nil)]
4      (a/->set-trace! activity trace)
5      (init-variables activity (first (a/all-Inputs ad)))
6      (mapc #(a/set-running! % true) (a/->nodes activity))
7      (loop [en (first (filter a/isa-InitialNode? (a/->nodes activity)))]
8        (when en
9          (exec-node en)
10         (a/->add-executedNodes! trace en)
11         (recur (first (enabled-nodes activity)))))
12     trace))
```

The function queries the single activity in the diagram, creates a new trace, and assigns that to the activity. The activity's variables are initialized and its nodes are set running.

Then, a `loop`-`recur` iteration[5] performs the actual execution of the activity. Initially, the variable `en` is bound to the activity's initial node, which gets executed and added to the trace. Thereafter, the loop is restarted with the next enabled node. Eventually, there won't be an enabled node left, and then the function returns the trace.

The first step in the execution of an activity is the initialization of its local and input variables. The corresponding function `init-variables` is shown in the next listing. For locals, their current value is set to their initial value if there is one defined. For input variables, their current value is set to the value of the input's corresponding input value element.

```
13 (defn init-variables [activity input]
14   (doseq [lv (a/->locals activity)]
15     (when-let [init-value (a/->initialValue lv)]
16       (a/->set-currentValue! lv init-value)))
17   (doseq [iv (and input (a/->inputValues input))]
18     (when-let [val (a/->value iv)]
19       (a/->set-currentValue! (a/->variable iv) val))))
```

After initializing the variables, the main function sets the activity's nodes running, and the main loop starts with the activity's initial node.

For different kinds of activity nodes, different execution semantics have to be encoded. This is exactly the use-case of FunnyQT's polymorphic functions (polyfn). A polymorphic function is declared once, and then different implementations for instances of different metamodel types can be defined. When

[5] `loop` is not a loop in the sense of Java's `for` or `while` but a local tail-recursion. The `loop` declares variables with their initial bindings, and in the `loop`'s body `recur` forms may recurse back to the beginning of the `loop` providing new bindings for the `loop`'s variables.

the polyfn is called, a polymorphic dispatch based on the polyfn's first argument's metamodel type is performed to pick out the right implementation[6].

The next listing shows the declaration of the polyfn `exec-node` and its implementation for initial nodes. The declaration only defines the name of the polyfn and the number of its arguments (just one, here). The implementation for initial nodes simply offers one new control token to the initial node's outgoing control flow edge.[7]

```
20 (declare-polyfn exec-node [node])

21 (defn offer-one-ctrl-token [node]
22   (let [ctrl-t (a/create-ControlToken! nil)
23         out-cf (the (a/->outgoing node))
24         offer  (a/create-Offer! nil {:offeredTokens [ctrl-t]})]
25     (a/->add-heldTokens! node ctrl-t)
26     (a/->add-offers! out-cf offer)))

27 (defpolyfn exec-node InitialNode [i]
28   (offer-one-ctrl-token i))
```

The following listing shows the `exec-node` implementations for join, merge, and decision nodes. Join and Merge nodes simply consume their input offers and pass the tokens they have been offered on all outgoing control flows. Decision nodes act similar but offer their input tokens only on the outgoing control flow whose guard variable's current value is true[8].

```
29 (defn pass-tokens
30   ([n] (pass-tokens n nil))
31   ([n out-cf]
32    (let [in-toks (consume-offers n)]
33      (a/->set-heldTokens! n in-toks)
34      (doseq [out-cf (if out-cf [out-cf] (a/->outgoing n))]
35        (a/->add-offers!
36         out-cf (a/create-Offer!
37                 nil {:offeredTokens in-toks}))))))

38 (defpolyfn exec-node JoinNode [jn]
39   (pass-tokens jn))

40 (defpolyfn exec-node MergeNode [mn]
41   (pass-tokens mn))

42 (defpolyfn exec-node DecisionNode [dn]
43   (pass-tokens dn (the #(-> % a/->guard a/->currentValue a/value)
44                        (a/->outgoing dn))))
```

So how does a node consume offers? This is defined by the `consume-offers` function shown in the following listing. First, the offers and the tokens offered by them are calculated. Then, the offered tokens are divided into control and forked tokens. For control tokens, their holder is unset. For forked tokens, the corresponding base token's holder is unset. The forked tokens' remainingOffersCount is decremented. If it has become zero then, the forked token is removed from its holder. Lastly, the offers are deleted, and the incoming tokens are returned.

```
45 (defn consume-offers [node]
46   (let [offers    (mapcat a/->offers (a/->incoming node))
```

[6] Polyfns support multiple inheritance. In case of an ambiguity during dispatch, e.g., two or more inherited implementations are applicable, an error is signaled.

[7] The FunnyQT function `the` is similar to Clojure's `first` except that it signals an error if the given collection contains zero or more than one element. Thus, it makes the assumption that there must be only one outgoing control flow explicit.

[8] `(the predicate collection)` returns the single element of the collection for which the predicate returns true. If there is no or more elements satisfying the predicate, an error is signaled.

```
47        tokens    (mapcat a/->offeredTokens offers)
48        ctrl-toks (filter a/isa-ControlToken? tokens)
49        fork-toks (filter a/isa-ForkedToken? tokens)]
50    (doseq [ct ctrl-toks]
51      (a/->set-holder! ct nil))
52    (doseq [ft fork-toks]
53      (when-let [bt (a/->baseToken ft)]
54        (a/->set-holder! bt nil))
55      (a/set-remainingOffersCount! ft (dec (a/remainingOffersCount ft)))
56      (when (zero? (a/remainingOffersCount ft))
57        (a/->set-holder! ft nil)))
58    (mapc edelete! offers)
59    tokens))
```

The remaining kinds of activity nodes are fork nodes, activity final nodes and opaque actions. Their `exec-node` implementations are printed in the next listing.

A fork node consumes its offers and creates one forked token per incoming token. The incoming tokens are set as the forked tokens' base tokens, and the remaining offers count is set to the number of outgoing control flows. All created forked tokens are offered on each outgoing control flow.

```
60 (defpolyfn exec-node ForkNode [fn]
61   (let [in-toks  (consume-offers fn)
62         out-cfs  (a/->outgoing fn)
63         out-toks (mapv #(a/create-ForkedToken!
64                          nil {:baseToken %, :holder fn,
65                               :remainingOffersCount (count out-cfs)})
66                        in-toks)]
67     (a/->set-heldTokens! fn in-toks)
68     (doseq [out-cf out-cfs]
69       (a/->add-offers! out-cf (a/create-Offer!
70                                nil {:offeredTokens out-toks})))))

71 (defpolyfn exec-node ActivityFinalNode [afn]
72   (consume-offers afn)
73   (mapc #(a/set-running! % false)
74         (-> afn a/->activity a/->nodes)))

75 (defpolyfn exec-node OpaqueAction [oa]
76   (consume-offers oa)
77   (mapc eval-exp (a/->expressions oa))
78   (offer-one-ctrl-token oa))
```

An activity final node simply consumes all offers and then sets the running attribute of all nodes contained by the executed activity to false. An opaque action also consumes all offers, then evaluates all its expressions in sequence using the `eval-exp` function, and finally offers one single control token on the outgoing control flow.

How an expression is evaluated depends on (1) its type and (2) on the value of its operator attribute. The expression's type is only important in order to separate unary from binary expressions, and the operator defines the semantics. Therefore, the `eval-exp` function shown in the next listing has a special case for boolean unary expressions which negates the expression's current value using `not`. For all binary expressions, the map `op2fn` mapping from operator enum constants to Clojure functions having the semantics of that operator is used. The function determined by looking up the expression's operator is applied to both operands to compute the new value.

```
79 (def op2fn {(a/enum-IntegerCalculationOperator-ADD)            +
80             (a/enum-IntegerCalculationOperator-SUBRACT)        -
81             (a/enum-IntegerComparisonOperator-SMALLER)         <
82             (a/enum-IntegerComparisonOperator-SMALLER_EQUALS) <=
83             (a/enum-IntegerComparisonOperator-EQUALS)          =
84             (a/enum-IntegerComparisonOperator-GREATER_EQUALS) >=
85             (a/enum-IntegerComparisonOperator-GREATER)         >
```

```
            (a/enum-BooleanBinaryOperator-AND)                #(and %1 %2)
            (a/enum-BooleanBinaryOperator-OR)                 #(or  %1 %2)})

(defn eval-exp [exp]
  (a/set-value! (-> exp a/->assignee a/->currentValue)
                (if (a/isa-BooleanUnaryExpression? exp)
                  (not (-> exp a/->operand a/->currentValue a/value))
                  ((op2fn (a/operator exp))
                   (-> exp a/->operand1 a/->currentValue a/value)
                   (-> exp a/->operand2 a/->currentValue a/value)))))
```

After executing all enabled nodes, the transformation's main function `execute-activity-diagram` recomputes the enabled nodes and resumes the execution. The enabled nodes are computed by the `enabled-nodes` function shown in the following listing. The enabled nodes are those nodes of a given activity which are set running, are no initial nodes[9], and receive an offer on each incoming control flow, or, in the case of a merge node, on one incoming control flow.

```
(defn enabled-nodes [activity]
  (filter (fn [n]
            (and (a/running? n)
                 (not (a/isa-InitialNode? n))
                 ((if (a/isa-MergeNode? n) exists? forall?)
                  #(seq (a/->offers %)) (a/->incoming n))))
          (a/->nodes activity)))
```

These 101 NCLOC of algorithmic FunnyQT/Clojure code implement the complete operational semantics of UML Activity Diagrams (with the exception of data flows which has not been demanded by the case description).

3 Evaluation

In this section, the FunnyQT solution is evaluated according to the criteria stated in the case description.

The solution comes with a test suite, and during the official evaluation achieved a full *correctness* score winning the *most correct solution award*.

With 101 lines of non-commented source code, the FunnyQT solution is quite *concise*. Of course, *understandability* is a very subjective measure measure. The solution should be evident for any Clojure programmer but even without prior Clojure knowledge, the solution shouldn't be hard to follow due to the usage of the metamodel-specific API. Another strong point is that all steps in the execution of an activity are encoded in one function each whose definition is almost a literal translation of the English description to FunnyQT/Clojure code.

The following table shows the *performance* in terms of execution times of the FunnyQT solution for all provided test models. These times were measured on a normal 4-core laptop with 2.6 GHz and 2 GB of RAM dedicated to the JVM.

Model	Time	Model	Time	Model	Time
test1	1.3 ms	*test5*	0.5 ms	*performance-variant-2*	1246.5 ms
test2	0.6 ms	*test6 (false)*	3.7 ms	*performance-variant-3-1*	1159.6 ms
test3	4.1 ms	*test6 (true)*	5.4 ms	*performance-variant-3-2*	72.7 ms
test4	3.2 ms	*performance-variant-1*	1104.0 ms		

When compared with the reference Java solution, the FunnyQT solution is slightly faster for all normal and performance test models, and about 8 times faster for the *performance-variant-3-1* model.

[9] Initial nodes have to be excluded because if they are set running, all of their (zero) incoming control flows have offers.

4 Conclusion

In this paper, the FunnyQT solution to the TTC 2015 Model Execution case has been discussed. The solution implements the full operational semantics of UML Activity Diagrams with the exception of object flows which haven't been considered in this case.

The solution achieved the full score with respect to correctness and won the most correct solution award. With 101 NCLOC, the specification is very concise. It is also well-understandable given that it is an almost literal translation of the case description to functions and one polymorphic function. These functions use a metamodel-specific API which FunnyQT has generated from the activity diagram metamodel. Also the performance is quite good. In general, its execution times are very similar to those of the Java reference solution, and for two of the performance test models, the FunnyQT solution is in fact several times faster.

Overall, FunnyQT seems to be very adequate for defining model interpreters. Especially its polymorphic function facility has been explicitly designed for these kinds of tasks.

References

[1] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT*, *Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[2] Tassilo Horn (2015): *Graph Pattern Matching as an Embedded Clojure DSL*. In: *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015*. To appear.

[3] Tanja Mayerhofer & Manuel Wimmer (2015): *The TTC 2015 Model Execution Case*. In: *Transformation Tool Contest 2015*.