Solving the TTC 2013 Flowgraphs Case with FunnyQT

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

March 22, 2013

Abstract

This paper describes the FunnyQT solution to the TTC 2013 Flowgraphs 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 polymorphic function 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 and the polymorphic function API have been used for the JaMoPP to structure graph transformation. The control and data flow analysis is performed algorithmically using FunnyQT's plain querying and model manipulation APIs. The final task requesting a simple DSL for validating the result models has been tackled by combining FunnyQT's querying API and Clojure metaprogramming.

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

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

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].

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 and its polymorphic function API have been used for converting the JaMoPP models into simpler structure graphs (Task 1, Section 2). Both the control flow analylis (Task 2, Section 3) and the data flow analysis (Task 3, Section 4) have been tackled algorithmically using FunnyQT's plain querying and model manipulation APIs. The task of validating the control and data flow links using a simple DSL (Task 4, Section 5) has been solved by using FunnyQT's querying API and Clojure metaprogramming.

2 Task 1: JaMoPP to StructureGraph

According to the case description [Hor13], the goal of this task is to transform a fine-granular Java syntax graph conforming to the JaMoPP metamodel [HJSW09] into a much simpler structure graph model that only contains statements and expressions that are neither structured nor subdivided any further. However, the original Java code of these statements and expressions should be reflected in the new elements' `txt` attribute. This model-to-text transformation is described in Section 2.1. Thereafter, the model-to-model transformation creating a structure graph from a JaMoPP model is described in Section 2.2.

2.1 JaMoPP to Text

This model-to-text transformation is implemented using FunnyQT's polymorphic function API. A polymorphic function is a function that is declared once, and then arbitrary many implementations for concrete metamodel types can be added. When a polymorphic function is called, the actual implementation is determined similarly to the typical dispatch in object-oriented programming languages. If there's no implementation provided for the element's type or one of its supertypes, an exception is thrown.

The function `stmt2str` implements the model-to-text transformation required for solving task 1. It is declared as follows.

```
1 (declare-polyfn stmt2str [elem])
```

`declare-polyfn` declares a new polymorphic function. Its name is `stmt2str`, and it receives exactly one parameter `elem`. It's task is to create a string representation matching the concrete Java syntax for the provided JaMoPP model element.

After the polymorphic function has been declared, implementations for concrete metamodel types can be added using `defpolyfn`. For example, this is the implementation for JaMoPP elements of type `ClassMethod`:

```
1 (defpolyfn stmt2str 'members.ClassMethod
2   [method]
3   (str (eget method :name) "()"))
```

I.e., to convert a JaMoPP method element to a string, the method's name is concatenated with a pair of parentheses.

For structured elements, the implementations simply call the polymorphic function for their children concatenating the results. For example, this is the polymorphic function for converting elements of type `AssignmentExpression` to a string.

```
(defpolyfn stmt2str 'expressions.AssignmentExpression
  [ae]
  (str (stmt2str (eget ae :child)) " "
       (stmt2str (eget ae :assignmentOperator)) " "
       (stmt2str (eget ae :value))))
```

The `child` of the assignment expression is some variable, the assignment operator is one of `=`, `+=`, `-=`, `*=`, or `/=`, and `value` is an arbitrary expression. These three components are converted to strings and then concatenated.

Basically, there could be one polymorphic function per JaMoPP type occuring in one of the input models. However, to save some lines of code, some implementations are defined for some superclass and dispatch on concrete subclasses themselves. One of these implementations is that for operators which will be invoked for the `assignmentOperator` in the previous listing:

```
(defpolyfn stmt2str 'operators.Operator
  [op]
  (type-case op
    'operators.Multiplication "*"
    'operators.Subtraction    "-"
    'operators.Addition       "+"
    'operators.Division       "/"
    'operators.LessThan       "<"
    'operators.GreaterThan    ">"
    'operators.Assignment     "="
    'operators.MinusMinus     "--"
    'operators.PlusPlus       "++"
    'operators.AssignmentPlus "+="
    'operators.Equal          "=="))
```

`type-case` gets some model element and several clauses. Each clause consists of a metamodel type and an expression. The expression paired with the first matching metamodel type is evaluated to form the result of the `type-case`. If no clause matches, an exception is thrown. As can be seen, there are several Java operators missing (e.g., `AssignmentMinus` aka `-=`), because they don't occur in the provided input models.

All in all, the polymorphic `stmt2str` function consists of 22 implementations for various JaMoPP metamodel types accounting to a total of about 110 lines of code.

2.2 JaMoPP to Structure Graph

The model-to-model transformation part of task 1 is implemented using FunnyQT's ATL- or ETL-like model transformation API. It realizes the JaMoPP to structure graph transformation requested by task 1, and it also creates the vars and params important for the data flow analysis as requested by task 3 (Section 4).

Before going into the details of the transformation, one helper function `used-vars` needs to be introduced.

```
(defn used-vars [s]
  (reachables s [p-seq [p-* <>--]
                 [p-restr 'references.IdentifierReference]
                 :target]))
```

It takes a JaMoPP statement `s` and returns the set of variables used inside this statement. `reachables` gets an element and regular path expression specifying the structure of paths that may be traversed. Here, the path expression specifies that from `s` zero or many (`p-*`) containment references from container to child (`<>--`) may be traversed. The intermediate results are

then filtered to elements of type `IdentifierReference`, and from those the `target` reference is traversed leading to a variable.

The actual transformation starts by defining its name and input and output models.

```
5  (deftransformation java2flowgraph [[in :emf] [out :emf]]
```

There could be arbitrary many input and output models, and they could be of different kinds, e.g., a transformation could receive a JGraLab TGraph and some EMF model, and create an output EMF model. Here, it gets only the JaMoPP EMF input model which is bound the the variable `in`, and one single structure graph output model bound to `out`.

In the body of such a transformation, arbitrary rules may be declared. The first rule is the `method2method` rule shown in the next listing. The `^:top` metadata preceeding the rule name specifies that the rule is a top-level rule. Such rules are applied to all matching elements by the transformation itself, whereas non-top-level rules have to be called explicitly from a top-level rule (directly or indirectly).

```
6    (^:top method2method [m]
7        :from 'members.ClassMethod
8        :when-let [mstmts (seq (eget m :statements))]
9        :to [fgm 'flowgraph.Method, ex 'flowgraph.Exit]
10       (eset! fgm :txt (stmt2str m))
11       (eset! ex :txt "Exit")
12       (eset! fgm :stmts (map stmt2item mstmts))
13       (eset! fgm :exit ex)
14       (eset! fgm :def (map param2param (eget m :parameters))))
```

It receives a model element `m` which must be of type `ClassMethod` in order for the rule to be applicable. The `:when-let` clause states that this rule is only applicable to methods that contain statements[3]. If it does, these statements are bound to `mstmts`. The reason for this restriction is that the input JaMoPP models contain many methods of classes in the Java standard library, but those don't have statements because they were not parsed from source code. The only method parsed from code is the single test method contained in the provided Java files.

The `:to` clause declares the objects to be created. Here, for a given JaMoPP method, a corresponding flowgraph method and its exit element are created.

The remainder of the rule is its body. Here, the `txt` attribute of the new method and its exit are set, the former using the polymorphic `stmt2str` function discussed in Section 2.1.

The method's `stmts` reference is set by applying another rule, `stmt2item`, to the statements of the JaMoPP method. The higher-order function `map` takes a function and a collection and applies the function to every element of the collection returning a sequence of function results.

Similarly, the method's parameters are transformed by mapping them to the `param2param` rule for setting the method's `def` reference.

When a rule is called and is applicable, the elements specified in the `:to` clause are created, the body is evaluated, a mapping from input element to output elements is created, and a vector containing the newly created elements in the order of their declaration in `:to` is returned. As a special case, if a rule creates only one target element, the mapping is from input element to that output element.

When the rule is applied another time for the same input element, it simply returns the result of the first application without creating anything new.

A special kind of rules are generalizing rules such as the one shown in the next listing.

```
15   (stmt2item [stmt]
16       :generalizes [local-var-stmt2simple-stmt condition2if block2block
17                     return2return while-loop2loop break2break continue2continue
18                     label2label stmt2simple-stmt])
```

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

This is quite similar to mapping disjunction in QVT Operational Mappings. When this rule is called, the rules specified in the `:generalizes` vector are tried one after the other, and the first applicable one is applied and its result is returned. The rules are ordered from most specific to most lax, and the last rule, `stmt2simple-stmt` is a catch-all rule applicable to any JaMoPP statement.

Since it is also the most complex rule, it is depicted in the next listing.

```
19  (stmt2simple-stmt [s]
20      :from 'statements.Statement
21      :to [fgss 'flowgraph.SimpleStmt]
22      (eset! fgss :txt (stmt2str s))
23      (doseq [aex  (reachables s [p-seq [p-* <>--]
24                                  [p-restr 'expressions.AssignmentExpression]])]
25        (eadd! fgss :def (var-creating-rule (the (used-vars (adj aex :child)))))
26        (eaddall! fgss :use (map var-creating-rule (used-vars (adj aex :value)))))
27      (doseq [umex (reachables s [p-seq [p-* <>--]
28                                  [p-restr 'expressions.UnaryModificationExpression]])]
29        (let [var (var-creating-rule (the (used-vars (adj umex :child))))]
30          (eadd! fgss :def var)
31          (eadd! fgss :use var))))
```

It receives a JaMoPP statement `s`, creates a new `SimpleStmt` `fgss` in the target model, and sets its `txt` attribute using the polymorphic `stmt2str` function.

The remainder of the rule deals with setting the new simple statement's `def` and `use` references. If the statement contains assignment expressions such as `a = b + c`, the flowgraph var corresponding to the assigned variable of each such expression is added to the statement's `def` reference. `var-creating-rule` is another rule generalizing the two rules transforming JaMoPP method parameters and local variables to flowgraph `Param` and `Var` objects, respectively. The `def` reference is set to the flowgraph vars corresponding to the variables used in the `value` expression of the assignment expression. Because the regular path expression in line 23 uses the zero-or-many iteration `p-*`, it might also be that `s` doesn't contain an assignment statement but it is an assignment statement itself.

Likewise, if the statement `s` contains or is a `UnaryModificationExpression` such as `i++`, the `def` and `use` references are set to the flowgraph var element corresponding to the JaMoPP variable being modified.

The complete `java2flowgraph` model-to-model transformation consists of 15 rules with 89 lines of code in total.

3 Task 2: Control Flow Analysis

The sole purpose of this task is to create `cfNext` links between `FlowInstr` elements in the flowgraph model created by the model-to-model transformation realizing task 1. Every such flow instruction should be connected to every other flow instruction that may be the next one in the program's control flow. This challenge has been tackled algorithmically using FunnyQT's plain quering and model manipulation APIs.

The idea of the algorithm is quite easy. It uses a sequence of statements as intermediate representation to work on realizing a pre-order depth-first iteration with look-ahead through the method's statements. In the general case, every flow instruction in that sequence has to be connected with the immediately following flow instruction in the sequence. For various kinds of statements, special rules are needed. For example, when encountering a block in the sequence (which is no flow instruction), the the block is replaced with its contents.

Since the next statement in the sequence might not be a flow instruction but some structured statement like a block, an if-statement, or a loop, there's the helper function `cf-peek`.

```
1  (defn cf-peek [el]
2    (if (has-type? el 'flowgraph.FlowInstr)
3      el
4      (recur (first (econtents el)))))
```

It receives some element `el`. If that element is a flow instruction, it is simply returned. Else, the function calls itself recursively with the first element of its contents, thus delivering the first flow instruction of a possibly nested composite statement. For example, when it is called for a label containing a block whose first statement is a loop, it will return the the expression which is the loop's condition.

The function synthesizing the control flow links is named `cf-synth` using the algorithm sketched above will be explained in the next listings. It receives the sequence of statements `v`, the method's `Exit` node `exit`, the current loop's `loop-expr`, the statement following the current loop `loop-succ`, and a map `label-succ-map` that assigns to each label reachable in the current scope the statement following the labeled statement. The `exit` parameter is used for handling return statements, and the last three parameters are used for handling break and continue statements. Initially, the function is called with `v` only containing the method, and `exit` bound to that method's exit. All other parameters are `nil`.

```
5  (defn cf-synth [v exit loop-expr loop-succ label-succ-map]
6    (when (seq v)
7      (let [[el & [n & _ :as tail]] v]
8        (type-case el
```

If the sequence `v` is not empty, its first element is bound to `el`, and its rest is bound to `tail`. Furthermore, the first element of the rest (i.e., the second element of the sequence) is bound to `n`. This technique of binding components of collections by mimicing the collection's structure is known as *destructuring* in the Lisp-world.

After binding these elements, a `type-case` dispatches on `el`'s metamodel type. If the element is a method, a control flow link to that method's first flow instruction is created, and the function recurses with the method's statements. `recur` is a Clojure special form enabling a tail-recursion that doesn't consume space on the call stack. If `recur` doesn't occur in tail-position, the Clojure compiler will throw an exception.

```
9          'flowgraph.Method
10                   (let [stmts (econtents el)]
11                     (eadd! el :cfNext (cf-peek (first stmts)))
12                     (recur stmts exit nil nil nil))
```

If the current element is a simple statement and there's a next element in the sequence, a control flow link is added to the first flow instruction of that next statement. Then, the function recurses with the tail of the sequence keeping the remaining parameters as-is.

```
13         'flowgraph.SimpleStmt
14                   (do (when n (eadd! el :cfNext (cf-peek n)))
15                       (recur tail exit loop-expr loop-succ label-succ-map))
```

If the current element is a block, the function simply recurses with that block's statements prepended to the tail of the sequence keeping the remaining parameters as-is. That is, a block is replaced with its contents.

```
16         'flowgraph.Block
17                   (recur (concat (econtents el) tail)
18                          exit loop-expr loop-succ label-succ-map)
```

If the current element is an expression of some if- or loop-statement and there's a next statement in the sequence, a control flow link is added to the first flow instruction of that next statement. Again, the function recurses with the tail of the sequence keeping the remaining parameters as-is.

```
19          'flowgraph.Expr
20                      (do (when n (eadd! el :cfNext (cf-peek n)))
21                          (recur tail exit loop-expr loop-succ label-succ-map))
```

If the current element is a label, the function recurses with that label's statement prepended to the tail of the sequence. A mapping from this label to the statement following it is added to the `label-succ-map`. This statement's first flow instruction is where the control flow continues when breaking to this label.

```
22          'flowgraph.Label
23                      (recur (cons (eget el :stmt) tail) exit loop-expr loop-succ
24                             (assoc label-succ-map el n))
```

If the current element is a return statement, a control flow link is added to the exit node of the method. Thereafter, the function recurses again with unchanged parameters.

```
25          'flowgraph.Return
26                      (do (eadd! el :cfNext exit)
27                          (recur tail exit loop-expr loop-succ label-succ-map))
```

If the current element is a break statement, two cases have to be destinguished. If the break is labled, a control flow link is added to the first flow instruction of the statement following the label. The statement following the label is looked up in the `label-succ-map`.

If the break is not labled, a control flow link is added to the first flow instruction in the statement following the surrounding loop which is bound to `loop-succ`.

In any case, the function recurses with the tail of the sequence keeping all other parameters as-is.

```
28          'flowgraph.Break
29                      (do (if-let [l (eget el :label)]
30                            (eadd! el :cfNext (cf-peek (label-succ-map l)))
31                            (eadd! el :cfNext (cf-peek loop-succ)))
32                          (recur tail exit loop-expr loop-succ label-succ-map))
```

If the current element is a continue statement, the same two cases have to be distinguished. In case of a labeled continue, a control flow link to the first flow instruction of that label is added. Because Java allows only to continue to labeled loops, this flow instruction is the expression of that loop's condition.

If the continue is not labeled, a control flow link to the nearest surrounding loop's expression, `loop-expr`, is added.

```
33          'flowgraph.Continue
34                      (do (if-let [l (eget el :label)]
35                            (eadd! el :cfNext (cf-peek l))
36                            (eadd! el :cfNext loop-expr))
37                          (recur tail exit loop-expr loop-succ label-succ-map))
```

If the current element is a loop, this loop's condition is bound to `expr`, and its body is bound to `body`. The function recurses with the expression, the body, and again the expression prepended to the tail of the sequence. The effect is that there will be a control flow link from the expression to the first flow instruction in the body (the true-case of the loop condition), control flow links from the final non-return/break/continue flow instructions in the body to the expression (the repeation), and a control flow link from the expression to the first flow instruction following the loop (the false-case of the loop condition). In the recursive call, the current loop expression `loop-expr` is bound to `expr`, and the statement following the loop `loop-succ` is bound to `n`, thus establishing a new scope for contained and non-labeled break and continue statements.

```
38          'flowgraph.Loop
39                      (let [[expr body] (econtents el)]
40                        (recur (cons expr (cons body (cons expr tail)))
41                               exit expr n label-succ-map))
```

If the current element is an if-statement, its condition is bound to `expr`, its then-statement is bound to `then`, and its else-statement (which might be `nil`) is bound to `else`.

First the function recurses with a vector containing only the expression, the then-statement, and the first flow instruction in the statement following the if-statement. The effect is that a control flow link will be added from the expression to the first flow instruction in the then-statement, and the final non-return/break/continue flow instructions in the then-statement will be connected to the first flow instruction following the if-statement. Note that this call is done by calling the function itself rather than `recur`, because the call is not in tail-position. As a result, deeply nested if-statements could produce stack-overflows in theory. However, the nesting would need to exceed at least 3000 levels with standard JVM settings in order.

Depending on wether the if-statement has an else-part, the function then recurses either with the expression and else-statement prepended to the tail, or with only the expression prepended to the tail. In the first case, the effect is that a control flow link will be created from the expression to the first flow instruction in the else-statement, and the final non-return/break/continue flow instructions in the else-statement will be connected to the first flow instruction following the if-statement. In the other case, a control flow link will be created from the expression to the first flow instruction following the if-statement.

```
42          'flowgraph.If
43                      (let [[expr then else] (econtents el)]
44                        (cf-synth [expr then (cf-peek n)]
45                                  exit loop-expr loop-succ label-succ-map)
46                        (if else
47                          (recur (cons expr (cons else tail))
48                                 exit loop-expr loop-succ label-succ-map)
49                          (recur (cons expr tail)
50                                 exit loop-expr loop-succ label-succ-map)))
```

Finally, if the current element is the method's exit node, a sanity check is performed. At this point, the sequence `v` should be exhausted, so there must be no next statement `n`.

```
51          'flowgraph.Exit (assert (nil? n))))))
```

The entry point to the control flow transformation is the function `synthesize-cf-edges` shown in the next listing. It receives a flowgraph model and calls the `cf-synth` function for any method contained in the model providing the method's exit node via its `exit` parameter.

```
52  (defn synthesize-cf-edges [model]
53    (doseq [m (eallobjects model 'flowgraph.Method)
```

```
54            :let [exit (the (eallobjects model 'flowgraph.Exit))]]
55      (cf-synth [m] exit nil nil nil)))
```

The control flow transformation consists of 55 lines of code and has been completely printed and discussed in this section.

4 Task 3: Data Flow Analysis

The purpose of this task is to create `dfNext` links between `FlowInst` elements where the target element is a control flow successor of the source element, the target element uses (reads) a variable that was defined (written) by the source element, and the variable hasn't been rewritten in between. This definition has been implemented exactly as stated here, because although it's not the most efficient algorithm for the task, it is very clear and concise.

The function `find-nearest-definers` receives a flow instruction `fi` and a variable `uv` used by it, and it returns a vector of the nearest control flow predecessors that define that variable.

```
1  (defn find-nearest-definers [fi uv]
2    (loop [preds (mapcat #(adjs % :cfPrev) (if (coll? fi) fi [fi]))
3           r []
4           known #{}]
5      (if (seq preds)
6        (let [definers (filter #(member? uv (eget % :def)) preds)
7              others   (remove #(member? uv (eget % :def)) preds)]
8          (recur (remove #(member? % known) (mapcat #(adjs % :cfPrev) others))
9                 (into r definers)
10                (into known preds)))
11       r)))
```

In Clojure, `loop` and `recur` implement a local tail-recursion, that is, inside a `loop` a `recur` form recurses not to the surrounding function but to the surrounding `loop`. Initially, `preds` is bound to the immediate control flow predecessors of `fi`, the result variable `r` is bound to the empty vector, and `known` is bound to the empty set.

If there are no predecessors, the result `r` is returned (the else-branch of the if). If there are control flow predecessors, those are sorted into `definers`, i.e., flow instructions that define `uv`. The other predecessors are sorted into `others`, i.e., flow instructions that don't write `uv`.

Then it is recursed to the surrounding `loop`. `preds` is rebound to those control flow predecessors of `others` that aren't already known in order not to recurse infinitely in case of control flow cycles, the result vector `r` is rebound to the current `r` value plus the new `definers`, and `known` is rebound to the union of the current `known` value and the current `preds`.

The function `synthesize-df-edges` shown in the next listing is the entry point to the data flow analysis receiving the flowgraph model. It performs a nested iteration[4] over all flow instructions, the variables used by them, and the definers of these used variables. A data flow link is added from every nearest definer to the flow instruction.

```
12 (defn synthesize-df-edges [model]
13   (doseq [fi (eallobjects model 'flowgraph.FlowInstr)
14           used-var (eget fi :use)
15           nearest-definer (find-nearest-definers fi used-var)]
16     (eadd! nearest-definer :dfNext fi))
17   (doseq [v (vec (eallobjects model 'Var))]
18     (edelete! v)))
```

[4]Clojure's doseq is similar to Java's enhanced for-each loop, but it also allows for nested iterations.

Lastly, all `Var` elements in the model are deleted. This isn't requested by the task, but because the unit tests of the transformation also create visualizations of the result models, a smaller model size with much fewer links results in a nicer layout.

Like with the control flow transformation, the data flow transformation consisting of 18 lines of code has been completely printed and discussed.

5 Task 4: Control and Data Flow Validation

The goal of task 4 is to enable offloading testing effort for the transformations solving tasks 1 to 3 to programmers knowing Java but not the transformation language used. Therefore, some simple DSL should be provided that lets them write down the expected control and data flow links. Some tool should be able to read that DSL and validate it against some result model. This tool should print all missing `cfNext`/`dfNext` links, i.e., links defined in the specification but not occuring in the result model, and it should print all false links, i.e., links occuring in the model but not in the specification.

Before digging into the FunnyQT solution's implementation of this task, the next listing shows the specification of the `Test0.java.xmi` model.

```
(make-test test-fg-transform-test0 "models/Test0.java.xmi"
           #{["testMethod()"    "int a = 1;"]     ;; expected cfNext links
             ["int a = 1;"      "int b = 2;"]
             ["int b = 2;"      "int c = a + b;"]
             ["int c = a + b;" "a = c;"]
             ["a = c;"          "b = a;"]
             ["b = a;"          "c = a / b;"]
             ["c = a / b;"      "b = a - b;"]
             ["b = a - b;"      "return b * c;"]
             ["return b * c;"   "Exit"]}
           #{["int a = 1;"      "int c = a + b;"] ;; expected dfNext links
             ["int b = 2;"      "int c = a + b;"]
             ["int c = a + b;" "a = c;"]
             ["a = c;"          "b = a;"]
             ["a = c;"          "c = a / b;"]
             ["a = c;"          "b = a - b;"]
             ["b = a;"          "c = a / b;"]
             ["b = a;"          "b = a - b;"]
             ["c = a / b;"      "return b * c;"]
             ["b = a - b;"      "return b * c;"]})
```

`make-test` is the entry point to the validation DSL. It creates a new test case with the given name (e.g., `test-fg-transform-test0`) which tests some concrete model whose XMI file is provided. Then, a set of expected `cfNext` and expected `dfNext` links follow. Every link is represented as a vector containing the expected source and target flow instructions, every flow instruction being represented as string of its concrete Java syntax. These strings identify the flow instructions unambiguously, because in the test cases, no statement occurs more than once.

Alternatively, instead of providing the links as sets of tuples, only the expected number of `cfNext`/`dfNext` links may be provided. For the larger models, writing down the expected control and data flow links is simply not feasible.

```
(make-test test-fg-transform-test9 "models/Test9.java.xmi" 14452 27202)
```

Of course, here the validation might succeed although the model is wrong anyway, however such a slight validation is still much better than no validation at all.

`make-test` is a *macro*. A macro is a function that will be called by the Clojure compiler at compile-time. It receives the unevaluated arguments given to it, that is, its parameters are

bound to code. Clojure, like all Lisps, is *homoiconic*, meaning that code is represented using usual Clojure data structures, e.g., lists, vectors, symbols, literals, etc. Thus, the macro is able to transform the code provided to it using standard Clojure functions to some new bunch of code that takes its place.

Without going into details, simple macros usually use a syntax-quoted form as the template for the code to generate, and inside such a template the parameters of the macro can be inserted by unquoting them with a tilde. Furthemore, new variables can be introduced by suffixing them with a hash. Those variables get an automatically generated name that is guaranteed to be unique and thus cannot clash with symbols contained in the code provided as macro arguments.

```
1  (defmacro make-test [n file expected-cfs expected-dfs]
2    `(deftest ~n
3       (let [fg-trg# (run-transformations ~file)
4             exp-cfs# ~expected-cfs
5             exp-dfs# ~expected-dfs
6             cfs# (set (map (fn [[s# t#]] [(eget s# :txt) (eget t# :txt)])
7                            (ecrosspairs fg-trg# :cfPrev :cfNext)))
8             dfs# (set (map (fn [[s# t#]] [(eget s# :txt) (eget t# :txt)])
9                            (ecrosspairs fg-trg# nil :dfNext)))]
10        (cond
11         (set? exp-cfs#) (let [cf-d1# (clojure.set/difference exp-cfs# cfs#)
12                               cf-d2# (clojure.set/difference cfs# exp-cfs#)]
13                           (is (empty? cf-d1#) "Missing cf-edges")
14                           (is (empty? cf-d2#) "Too many cf-edges"))
15         (number? exp-cfs#) (do
16                              (println "Only checking number of cfNext links.")
17                              (is (= exp-cfs# (count cfs#))))
18         :else (println "No expected cfNext links given."))
19        (cond
20         (set? exp-dfs#) (let [df-d1# (clojure.set/difference exp-dfs# dfs#)
21                               df-d2# (clojure.set/difference dfs# exp-dfs#)]
22                           (is (empty? df-d1#) "Missing df-edges")
23                           (is (empty? df-d2#) "Too many df-edges"))
24         (number? exp-dfs#) (do
25                              (println "Only checking number of dfNext links.")
26                              (is (= exp-dfs# (count dfs#))))
27         :else (println "No expected dfNext links given."))))))
```

In line 2, `make-test` generates a new unit test with the name given as parameter `n`. In line 3, the transformations are run transformation[5] and the result model is bound to a variable. Likewise, the expected links are bound to variables. Lines 6 and 8 bind the links actually contained in the model to variables. The FunnyQT function `ecrosspairs` returns the sequence of crosslinks restricted to those matching given source and target reference names. Each link is represented as a vector of source and target element. Using `map`, these links are converted to the form of the links in the sets of expected links, i.e., every link is represented by a vector of the source and target element's `txt` attribute.

In lines 11 to 18 and lines 20 to 27, the differences between expected and actual links and vice versa are calculated. Both should be empty, else there are either missing or too many links. `is` is similar to JUnit's `Assert.assertTrue()` method. In case the expected links were provided as a number, only actual number of links is validated against it.

[5] run-transformations is a function that takes a model file and applies the 3 transformations one after the other returning the final program dependence graph. It also times the execution, saves the result model, and generates visualizations for smaller models.

6 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 Petri-Nets to Statecharts project:
   ```
   $ cd ~/Desktop/FunnyQT_Solutions/ttc-2013-flowgraphs/
   ```
3. Run the test cases:
   ```
   $ lein test
   ```

This will run the complete transformation (JaMoPP to structure graph with vars, control flow analysis, data flow analysis, validation) on all provided test JaMoPP models and print the execution times. The result models and visualizations of the main test cases' results are saved to the `results` directory.

7 Evaluation

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.

[HJSW09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. JaMoPP: The Java Model Parser and Printer. Technical Report TUD-FI09-10, Technische Universität Dresden, Fakultät Informatik, 2009. `ftp://ftp.inf.tu-dresden.de/pub/berichte/tud09-10.pdf`.

[Hor13] Tassilo Horn. The TTC 2013 Flowgraphs Case. `https://github.com/tsdh/ttc-2013-flowgraphs-case/blob/master/desc/ttc-2013-flowgraphs-case.pdf?raw=true`, 2013.

[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.