Solving the TTC 2013 Flowgraphs Case with FunnyQT

Dipl.-Inform. 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 Flowgraphs Transformation Case. It solves all four tasks, and it has won the *best efficiency award* for this case.

1 Introduction

FunnyQT 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[1], FunnyQT's model transformation API and its polymorphic function API have been used for task 1. Both task 2 and task 3 have been tackled algorithmically using FunnyQT's plain querying and model manipulation APIs. Task 4 has been solved by using FunnyQT's querying API and Clojure metaprogramming.

2 Solution Description

Task 1: JaMoPP to StructureGraph. According to the case description [2], the goal of this task is to transform a fine-granular Java syntax graph conforming to the JaMoPP metamodel [1] 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 the next paragraph. Thereafter, the model-to-model transformation creating a structure graph from a JaMoPP model is described.

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

Submitted to:
Transformation Tool Contest 2013 (TTC'13)

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

`declare-polyfn` declares a new polymorphic function. Its name is `stmt2str`, and it receives exactly one parameter `elem`. Its 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 `AssignmentExpression`:

```
(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 using `stmt2str` again which are then concatenated.

All in all, the polymorphic `stmt2str` function consists of 22 implementations for various JaMoPP metamodel types accounting to a total of 107 lines of code. The complete model-to-text transformation is printed in Appendix A.

JaMoPP to Structure Graph. The JaMoPP-to-StructureGraph transformation is implemented using FunnyQT's model-to-model transformation API. This transformation also creates `Var` and `Param` objects as requested by task 3.1.

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

```
(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`, which is also an EMF model.

In the body of such a transformation, arbitrary many rules may be defined. The first one 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).

```
  (^:top method2method [m]
         :from 'members.ClassMethod
         :to [fgm 'flowgraph.Method, ex 'flowgraph.Exit]
         (eset! fgm :txt (stmt2str m)) ;; Invoke the model-to-text transformation
         (eset! ex :txt "Exit")
         (eset! fgm :exit ex)
         (eset! fgm :stmts (map stmt2item (eget m :statements)))  ;; transform the statements
         (eset! fgm :def (map param2param (eget m :parameters)))) ;; transform the parameters
```

It receives a JaMoPP model element `m`. The `:from` clause dictates that `m` must be of type `ClassMethod` in order for the rule to be applicable. The `:to` clause declares the objects to be created. Here, for a given JaMoPP method, a corresponding flowgraph method and its exit object are created. The remainder of the rule is its body containing arbitrary code to set attributes and references. Here, the `txt` attribute of the new method and its exit are set, the former using the polymorphic `stmt2str` function discussed above. The method's `stmts` reference is set by applying another rule, `stmt2item`, to the statements of the JaMoPP method. Likewise, the method's parameters are transformed by mapping them to the `param2param` rule for setting the method's `def` reference.

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

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

This concept 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. Furthermore, a generalizing rule also combines the traceability mappings of all subrules.

The complete `java2flowgraph` model-to-model transformation consists of 15 rules with 93 lines of code in total. It is printed in Appendix B.

Task 2: Control Flow Analysis. The 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 algorithm uses a sequence of statements as intermediate representation to work on realizing a pre-order depth-first traversal 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 a helper function `cf-peek`. It receives some element and returns either this element if it is a flow instruction, or otherwise the first flow instruction inside this element.

The function `cf-synth` synthesizing the control flow links using the algorithm sketched above is explained in the next listings. It receives the sequence of statements `v`, the method's `Exit` node `exit`, the current loop's test expression (`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`.

```
(defn cf-synth [v exit loop-expr loop-succ label-succ-map]
  (when (seq v)
    (let [[el & [n & _ :as tail]] v]
      (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`.

After binding these elements, a `type-case` dispatches on `el`'s metamodel type. For example, 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.

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

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 its following statement is added to the `label-succ-map`. This statement's first flow instruction is where the control flow continues when breaking to this label.

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

If the current element is a break statement, two cases have to be distinguished. If the break is labeled, a control flow link is added to the first flow instruction of the statement following the label which can be 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.

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

There are similar cases for handling objects of the other metamodel types. The complete control flow transformation consists of 57 lines of code and is printed in Appendix C.

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.

```
(defn find-nearest-definers [fi uv]
  (loop [preds (mapcat #(adjs % :cfPrev) (if (coll? fi) fi [fi])),
         r [], known #{}]
    (if (seq preds)
      (let [definers (filter #(member? uv (eget % :def)) preds)
            others   (remove #(member? uv (eget % :def)) preds)]
        (recur (remove #(member? % known) (mapcat #(adjs % :cfPrev) others))
               (into r definers) (into known preds)))
      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` and `others`, i.e., flow instructions that write to `uv`, and flow instructions that don't write to `uv`, respectively.

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 main function of this task simply uses this function to find the nearest definers of all flow instructions and their used variables and creates `dfNext` links.

The complete data flow transformation consists of 19 lines of code and is printed in Appendix D.

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 only Java by equipping them with some easy to use DSL. The next listing shows an example validation specification as provided by the FunnyQT solution.

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

The FunnyQT solution uses Clojure's metaprogramming facilities to create an *internal validation DSL*. `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. Here, `make-test` creates a unit test that loads the given XMI model and compares it against the expected control and data flow links.

The complete macro implementation and two complete validation specifications are printed in Appendix E.

3 Evaluation

In this section, the FunnyQT solution to the Flowgraphs case is evaluated according to the critera listed in the case description [2].

All four tasks have been solved, and the results of every task are *complete and correct*. The FunnyQT solution consists of 313 lines of code excluding comments and empty lines, making it the shortest of all provided solutions. It is also the solution with the best *performance* and has won the *best efficiency award* for this case. However, because FunnyQT is a Clojure API with a functional alignment, its *understandability* depends largely on a reader's prior knowledge about Clojure and functional programming.

References

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

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

A The complete JaMoPP-to-Text Transformation

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

(defn reduce-str [els]
  (reduce #(str %1 (stmt2str %2)) "" els))

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

(defpolyfn stmt2str 'types.PrimitiveType [^org.eclipse.emf.ecore.EObject pt]
  (clojure.string/lower-case (.getName (.eClass pt))))

(defpolyfn stmt2str 'statements.LocalVariableStatement [lv]
  (let [v (eget lv :variable)]
    (str (stmt2str (eget v :typeReference)) " " (stmt2str v)
         (when-let [iv (eget v :initialValue)]
           (str " = " (stmt2str iv)))
         ";")))

(defpolyfn stmt2str 'references.IdentifierReference [ir]
  (stmt2str (eget ir :target)))

(defpolyfn stmt2str 'variables.Variable [v]
  (eget v :name))

(defpolyfn stmt2str 'expressions.MultiplicativeExpression [me]
  (let [[c1 c2] (eget me :children)]
    (str (stmt2str c1) " " (reduce-str (eget me :multiplicativeOperators))
         " " (stmt2str c2))))

(defpolyfn stmt2str 'expressions.EqualityExpression [ee]
  (let [[c1 c2] (eget ee :children)]
    (str (stmt2str c1) " " (reduce-str (eget ee :equalityOperators))
         " " (stmt2str c2))))

(defpolyfn stmt2str 'expressions.AdditiveExpression [ae]
  (let [[c1 c2] (eget ae :children)]
    (str (stmt2str c1) " " (reduce-str (eget ae :additiveOperators))
         " " (stmt2str c2))))

(defpolyfn stmt2str 'expressions.UnaryExpression [ue]
  (str (reduce-str (eget ue :operators))
       (stmt2str (eget ue :child))))

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

(defpolyfn stmt2str 'expressions.RelationExpression [re]
  (let [[c1 c2] (eget re :children)]
    (str (stmt2str c1) " "
         (reduce-str (eget re :relationOperators))
         " " (stmt2str c2))))
```

```
(defpolyfn stmt2str 'expressions.SuffixUnaryModificationExpression [se]
  (str (stmt2str (eget se :child))
       (stmt2str (eget se :operator))))

(defpolyfn stmt2str 'statements.Block [b]
  "\\{...\\}")

(defpolyfn stmt2str 'statements.Condition [c]
  "if")

(defpolyfn stmt2str 'statements.WhileLoop [c]
  "while")

(defpolyfn stmt2str 'statements.JumpLabel [l]
  (str (eget l :name) ":"))

(defpolyfn stmt2str 'statements.Break [b]
  (str "break"
       (when-let [l (eget b :target)]
         (str " " (eget l :name)))
       ";"))

(defpolyfn stmt2str 'statements.Continue [c]
  (str "continue"
       (when-let [l (eget c :target)]
         (str " " (eget l :name)))
       ";"))

(defpolyfn stmt2str 'statements.Return [r]
  (str "return" (when-let [rv (eget r :returnValue)]
                  (str " " (stmt2str rv)))
       ";"))

(defpolyfn stmt2str 'statements.ExpressionStatement [stmt]
  (str (stmt2str (eget stmt :expression)) ";"))

(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          "=="))

(defpolyfn stmt2str 'literals.Literal [l]
  (type-case l
    'literals.DecimalIntegerLiteral (eget l :decimalValue)))
```

B The complete JaMoPP-to-StructureGraph Transformation

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

(deftransformation java2flowgraph [[in :emf] [out :emf]]
  (^:top method2method [m]
     :from 'members.ClassMethod
     :to [fgm 'flowgraph.Method, ex 'flowgraph.Exit]
     (eset! fgm :txt (stmt2str m))
     (eset! ex :txt "Exit")
     (eset! fgm :stmts (map stmt2item (seq (eget m :statements))))
     (eset! fgm :exit ex)
     (eset! fgm :def (map param2param (eget m :parameters))))
  (stmt2item [stmt]
     :generalizes [local-var-stmt2simple-stmt condition2if block2block
                   return2return while-loop2loop break2break continue2continue
                   label2label stmt2simple-stmt])
  (var-creating-rule [v]
     :generalizes [param2param local-var2var])
  (param2param [p]
     :from 'parameters.Parameter
     :to [fgp 'flowgraph.Param]
     (eset! fgp :txt (stmt2str p)))
  (local-var2var [lv]
     :from 'variables.LocalVariable
     :to [fgv 'flowgraph.Var]
     (eset! fgv :txt (stmt2str lv)))
  (local-var-stmt2simple-stmt [lv]
     :from 'statements.LocalVariableStatement
     :to [fgss 'flowgraph.SimpleStmt]
     (let [v (local-var2var (adj lv :variable))]
       (eset! fgss :txt (stmt2str lv))
       (eadd! fgss :def v)
       (eset! fgss :use (map var-creating-rule
                             (used-vars (adj lv :variable :initialValue))))))
  (stmt2simple-stmt [s]
     :from 'statements.Statement
     :to [fgss 'flowgraph.SimpleStmt]
     (eset! fgss :txt (stmt2str s))
     (doseq [aex  (reachables s [p-seq [p-* <>--]
                                 [p-restr 'expressions.AssignmentExpression]])]
       (eadd! fgss :def (var-creating-rule (the (used-vars (adj aex :child)))))
       (eaddall! fgss :use (map var-creating-rule (used-vars (adj aex :value)))))
     (doseq [umex (reachables s [p-seq [p-* <>--]
                                 [p-restr 'expressions.UnaryModificationExpression]])]
       (let [var (var-creating-rule (the (used-vars (adj umex :child))))]
         (eadd! fgss :def var)
         (eadd! fgss :use var))))
  (label2label [l]
     :from 'statements.JumpLabel
     :to [fgl 'flowgraph.Label]
     (eset! fgl :txt (stmt2str l))
     (eset! fgl :stmt (stmt2item (eget l :statement))))
```

```
  (expression2expr [ex]
      :from 'expressions.Expression
      :to [fgex 'flowgraph.Expr]
      (eset! fgex :txt (stmt2str ex))
      (eset! fgex :use (map var-creating-rule (used-vars ex))))
  (condition2if [c]
      :from 'statements.Condition
      :to [fgif 'flowgraph.If]
      (eset! fgif :txt (stmt2str c))
      (eset! fgif :expr (expression2expr (eget c :condition)))
      (eset! fgif :then (stmt2item (eget c :statement)))
      (when-let [else (eget c :elseStatement)]
        (eset! fgif :else (stmt2item else))))
  (block2block [b]
      :from 'statements.Block
      :to [fgb 'flowgraph.Block]
      (eset! fgb :txt (stmt2str b))
      (eset! fgb :stmts (map stmt2item (eget b :statements))))
  (return2return [r]
      :from 'statements.Return
      :to [fgr 'flowgraph.Return]
      (eset! fgr :txt (stmt2str r))
      (eset! fgr :use (map var-creating-rule (used-vars r))))
  (break2break [b]
      :from 'statements.Break
      :to [fgb 'flowgraph.Break]
      (eset! fgb :txt (stmt2str b))
      (eset! fgb :label (label2label (eget b :target))))
  (continue2continue [c]
      :from 'statements.Continue
      :to [fgc 'flowgraph.Continue]
      (eset! fgc :txt (stmt2str c))
      (eset! fgc :label (label2label (eget c :target))))
  (while-loop2loop [wl]
      :from 'statements.WhileLoop
      :to   [fgl 'flowgraph.Loop]
      (eset! fgl :txt (stmt2str wl))
      (eset! fgl :expr (expression2expr (eget wl :condition)))
      (eset! fgl :body (stmt2item (eget wl :statement)))))
```

C The complete Control Flow Transformation

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

(defn cf-synth [v exit loop-expr loop-succ label-succ-map]
  (when (seq v)
    (let [[el & [n & _ :as tail]] v]
      (type-case el
        'flowgraph.Method
                   (let [stmts (econtents el)]
                     (eadd! el :cfNext (cf-peek (first stmts)))
                     (recur stmts exit nil nil nil))
```

```
        'flowgraph.SimpleStmt
                   (do (when n (eadd! el :cfNext (cf-peek n)))
                       (recur tail exit loop-expr loop-succ label-succ-map))
        'flowgraph.Block
                   (recur (concat (econtents el) tail)
                          exit loop-expr loop-succ label-succ-map)
        'flowgraph.Expr
                   (do (when n (eadd! el :cfNext (cf-peek n)))
                       (recur tail exit loop-expr loop-succ label-succ-map))
        'flowgraph.Label
                   (recur (cons (eget el :stmt) tail) exit loop-expr loop-succ
                          (assoc label-succ-map el n))
        'flowgraph.Return
                   (do (eadd! el :cfNext exit)
                       (recur tail exit loop-expr loop-succ label-succ-map))
        'flowgraph.Break
                   (do (if-let [l (eget el :label)]
                         (eadd! el :cfNext (cf-peek (label-succ-map l)))
                         (eadd! el :cfNext (cf-peek loop-succ)))
                       (recur tail exit loop-expr loop-succ label-succ-map))
        'flowgraph.Continue
                   (do (if-let [l (eget el :label)]
                         (eadd! el :cfNext (cf-peek l))
                         (eadd! el :cfNext loop-expr))
                       (recur tail exit loop-expr loop-succ label-succ-map))
        'flowgraph.Loop
                   (let [[expr body] (econtents el)]
                     (recur (cons expr (cons body (cons expr tail)))
                            exit expr n label-succ-map))
        'flowgraph.If
                   (let [[expr then else] (econtents el)]
                     (cf-synth [expr then (cf-peek n)]
                               exit loop-expr loop-succ label-succ-map)
                     (if else
                       (recur (cons expr (cons else tail))
                              exit loop-expr loop-succ label-succ-map)
                       (recur (cons expr tail)
                              exit loop-expr loop-succ label-succ-map)))
        'flowgraph.Exit (assert (nil? n))))))

(defn synthesize-cf-edges [model]
  (doseq [m (eallobjects model 'flowgraph.Method)
          :let [exit (the (eallobjects model 'flowgraph.Exit))]]
    (cf-synth [m] exit nil nil nil)))
```

D The complete Data Flow Transformation

```
(defn find-nearest-definers [fi uv]
  (loop [preds (mapcat #(adjs % :cfPrev) (if (coll? fi) fi [fi]))
         r []
         known #{}]
    (if (seq preds)
      (let [definers (filter #(member? uv (eget % :def)) preds)
            others   (remove #(member? uv (eget % :def)) preds)]
        (recur (remove #(member? % known) (mapcat #(adjs % :cfPrev) others))
```

```clojure
                (into r definers)
                (into known preds)))
        r)))

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

E The complete Validation DSL Implementation

```clojure
(defn run-flowgraph-transformations [file]
  (System/gc)
  (println "Running Transformation on" file)
  (print "Load Time: ")
  (let [jamopp-model (time (load-model file))
        outfile (str/replace (str/replace file "models/" "results/")
                             ".java.xmi" ".xmi")
        outvizfile (str/replace outfile ".xmi" ".pdf")
        fg-trg (new-model)]
    (println "Execution Times:")
    (print "  - JaMoPP to StructureGraph (with Vars): ")
    (time (java2flowgraph jamopp-model fg-trg))
    (print "  - Control Flow Analysis:                ")
    (time (synthesize-cf-edges fg-trg))
    (print "  - Data Flow Analysis:                   ")
    (time (synthesize-df-edges fg-trg))
    (save-model fg-trg outfile)
    (when (< (count (eallobjects fg-trg)) 80)
      (print-model fg-trg outvizfile))
    fg-trg))

(defmacro make-test [n file expected-cfs expected-dfs]
  `(deftest ~n
     (println "=========================================================================")
     (let [fg-trg# (run-flowgraph-transformations ~file)
           exp-cfs# ~expected-cfs
           exp-dfs# ~expected-dfs
           cfs# (set (map (fn [[s# t#]] [(eget s# :txt) (eget t# :txt)])
                          (ecrosspairs fg-trg# :cfPrev :cfNext)))
           dfs# (set (map (fn [[s# t#]] [(eget s# :txt) (eget t# :txt)])
                          (ecrosspairs fg-trg# nil :dfNext)))]
       (cond
        (set? exp-cfs#) (let [cf-d1# (clojure.set/difference exp-cfs# cfs#)
                              cf-d2# (clojure.set/difference cfs# exp-cfs#)]
                          (is (empty? cf-d1#) "Missing cf-edges")
                          (is (empty? cf-d2#) "Too many cf-edges"))
        (number? exp-cfs#) (do
                             (println "Only checking number of cfNext links.")
                             (is (= exp-cfs# (count cfs#))))
        :else (println "No expected cfNext links given."))
       (cond
```

```
        (set? exp-dfs#) (let [df-d1# (clojure.set/difference exp-dfs# dfs#)
                              df-d2# (clojure.set/difference dfs# exp-dfs#)]
                          (is (empty? df-d1#) "Missing df-edges")
                          (is (empty? df-d2#) "Too many df-edges"))
        (number? exp-dfs#) (do
                             (println "Only checking number of dfNext links.")
                             (is (= exp-dfs# (count dfs#))))
        :else (println "No expected dfNext links given.")))))
```

E.1 Two Example Validation Specifications

```
(make-test test-fg-transform-test4 "models/Test4.java.xmi"
           #{["testMethod()" "int i = 100;"]
             ["int i = 100;" "i > 0"]
             ["i > 0"        "Exit"]
             ["i > 0"        "i > 50"]
             ["i > 50"       "i--;"]
             ["i > 50"       "i = i - 10;"]
             ["i = i - 10;"  "i == 50"]
             ["i == 50"      "break;"]
             ["i == 50"      "i > 50"]
             ["break;"       "i--;"]
             ["i--;"         "i > 0"]}
           #{["int i = 100;" "i > 0"]
             ["int i = 100;" "i > 50"]
             ["int i = 100;" "i = i - 10;"]
             ["int i = 100;" "i--;"]
             ["i = i - 10;"  "i == 50"]
             ["i = i - 10;"  "i > 50"]
             ["i = i - 10;"  "i = i - 10;"]
             ["i = i - 10;"  "i--;"]
             ["i--;"         "i > 0"]
             ["i--;"         "i > 50"]
             ["i--;"         "i = i - 10;"]
             ["i--;"         "i--;"]})

;; For the large models, only the correct number of cfNext/dfNext links is asserted.
(make-test test-fg-transform-test9 "models/Test9.java.xmi" 14452 27202)
```