Solving the Flowgraphs Case with Eclectic

Jesús Sánchez Cuadrado
Universidad Autónoma de Madrid (Spain)
jesus.sanchez.cuadrado@uam.es

This paper presents a solution for the Flow Graphs case of the Transformation Tool Contest 2013, using the Eclectic model transformation tool. The solution makes use of several languages of Eclectic, showing how it is possible to combine them to address a non-trivial transformation problem in a concise and modulary way.

1 Introduction

The TTC 2013 Flow Graphs case [3] proposes the analysis of Java programs, conforming to the JaMoPP meta-model [2], by transforming them into a language-independent meta-model which represents the structure of the program and includes information about control and data flows. This solution makes use of the Eclectic transformation tool [1] to solve the four proposed tasks. The fourth task has been addressed using METADEPTH [4] to create a small DSL. [1]

Eclectic is a transformation tool designed as a family of model transformation languages, that is, a set of transformation languages each one specifically designed to address a specific transformation concern, as well as some composition mechanisms for their combination. The objective of this solution is thus to show how it is possible to address a non-trivial transformation task, such as this case, using several languages and how this approach has the potential of improving modularity and readability.

Eclectic currently provides the following languages: *i)* a mapping language for establishing one-to-one and one-to-many correspondences, *ii)* a target-oriented language with object notation and explicit rule calls, *iii)* a traversal language based on in the idea of attributed grammars, *iv)* a pattern matching language which used object-notation, and *v)* a lower-level scripting language, which also plays the role of scheduling language. Languages *i, ii and iv* do not allow complex expressions, but these need to be encoded in navigation libraries, written in the scripting language.

In principle, the combination of these languages permits covering many model transformation scenarios, in a more intentional way than using a general purpose transformation language. Addressing case studies could allow this intuition to be evaluated in practice. The solution of this case has used the mapping language, the attribution language, the pattern matching language, the scripting language and a navigation module. The target-oriented language is not needed because it is typically useful for synthesis tasks, but the case only involves mappings and analysis tasks.

In Eclectic every language is compiled to an intermediate representation, called IDC. It provides primitive instructions for model manipulation. Then, IDC is compiled to the Java Virtual Machine (JVM) bytecode format. In this way, all Eclectic languages share the same execution infrastructure. The composition mechanisms are implemented at the IDC level. There is also a runtime library, which provides datatypes (e.g., immutable lists), a model manager (i.e., EMF and METADEPTH are supported), etc.

[1] This solution is available as a SHARE image: `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC2013_Eclectic_FlowGraphs.vdi`

Pieter Van Gorp, Louis M. Rose, Christian Krause (Eds.):
Sixth Transformation Tool Contest (TTC 2013)
EPTCS 135, 2013, pp. 43–56, doi:10.4204/EPTCS.135.6

```
1  eclectic task1 (in) -> (out)
2
3  mappings task1_map(in) -> (out)
4    uses task1_attribution
5    uses task1_patterns
6
7    from src : in!ClassMethod
8      to tgt : out!Method, exit : out!Exit
9      linking tgt.exit = exit
10       tgt.stmts <- src.statements
11       tgt.txt = task1_attribution!text[src]
12   end
13
14   from src : in!WhileLoop
15     to tgt : out!Loop
16       tgt.expr <- src.condition
17       tgt.body <- src.statement
18       tgt.txt = task1_attribution!text[src]
19   end
20
21   from src : task_patterns!LoopExpression
22     to tgt : out!Expr
23       tgt.txt = task1_attribution!text[src]
24   end
25 end
26
27 attribution task1_attribution(in) -> (out)
28   syn text : _!String
29
30   rule in!WhileLoop
31     text[self] <- "while"
32   end
33
34   rule in!AssignmentExpression
35     left = text[self.child]
36     right = text[self.value]
37
38     text[self] <- left.concat(' = ').concat(right)
39   end
40 end
41
42 patterns task1_patterns(in)
43   def LoopExpression -> (e)
44     l : in!WhileLoop {
45       condition = e : in!Expression { }
46     }
47   end
48   // ... Likewise for ConditionalExpression ...
49 end
```

Figure 1: Excerpt of the mapping from JaMoPP to FlowGraph

2 Solution

2.1 Task 1

The first task is a model-to-model transformation, which comprises three different concerns that should be implemented in three different modules: i) A simple mapping between JaMoPP and FlowGraph elements must be performed. The mapping is mostly one-to-one, therefore the Eclectic mapping language would suffice. ii) A bottom-up text serialization of the JaMoPP abstract syntax tree. This could be implemented with a series of helper methods or using the attribution language, which allows us to propagate text from the leaves of a statement to the root, creating the serialization during the process. iii) An Expression element must not be translated, unless it is the condition of a *loop* or an *if*. To tackle this, the pattern language would be in charge of recognizing the cases and it is combined with the mapping language.

In this way, the proposed solution makes use of three modules (task1_map, task1_attribution, and task1_patterns). The mapping module has a dependency on the attribution module, to retrieve the textual representation of each source element, and on the pattern matching module, which feeds it with non-trivial matches. The listing in Figure 1 shows an excerpt of the transformation. It declares an Eclectic transformation called task1, which encloses the three modules.

The mapping transformation is more or less straightforward. Its semantics is basically similar to ATL. Rules are executed at top level (i.e., non-lazy execution), and the ← operation (a binding) resolves a target element from a source element. Interestingly, only simple expressions are allowed in the right part of a binding. The most subtle detail is how to "communicate" with the other modules.

To interoperate with the attribution transformation the syntax transformation!attribute[expr] is used (see lines 11, 18 and 23), which means: *retrieve the element associated to expr through the attribute*. As a concrete example, the text for the WhileLoop (retrieved in line 18) is actually produced by the assignment of the text attribute in line 31.

To interoperate with the pattern language, the mapping language treats a pattern as a regular type. It can be seen as an extended layer put on top of the original meta-model. In this way, the rule in lines 7–10 will be executed for each ocurrence of the LoopExpression pattern, defined in lines 43–46. This pattern is matched if there is a WhileLoop containing an Expression in this condition, and in such case the expression (variable e) is "returned".

The attribution transformation is also very simple, but the mechanics of attributes has to be taken into account. The language supports synthesized and inherited attributes (i.e., attributes propagated bottom-up and top-down, respectively).[2]

An attribute is assigned with the syntax attribute[$expr_1$] ← $expr_2$, and it has the effect of creating a trace link between the value obtained with $expr_1$ and $expr_2$. Conversely, retrieving the attribute associated to an element is done with the syntax attribute[expr]. For instance, in lines 35 and 36 the value of the text attribute is retrieved for the left and right parts of the assignment expression, and then these two values are used to give the text value to the assignment expression, that is, the *self* of the rule (line 38).

With respect to the integration at run-time of the different modules, all modules are executed concurrently, exchanging data among them as the execution proceeds. When all modules have finished its execution, the transformation is finished.

2.2 Task 2

This task is intended to complete the program structure computed in the previous task with the links defining the control flow graph of the program.

It is an in-place transformation, as the source model has to be augmented with the flow information. However, the main challenge is the computation of the implicit flow relationships. This task is particularly well suited for attribute-based traversal, because control flow attributes have to be propagated along the program structure (bottom-up and top-down). The presented solution makes use of two attributes. i) *successors* which is an inherited attribute specifying the list of "flow" siblings of each statement. In addition, it relieves statements from knowning its position within its container statement. ii) *cf_next*, which is a synthesized attribute representing the flow instruction that corresponds to an element. This is useful to make the transformation more homogenous since every element will have a corresponding flow instruction (e.g., a Block)[3].

In this section only the rules for blocks and simple statements are shown (see Figure 2), just to give an impression of the style of the solution. The complete explanation is given in Appendix A.2.

The rule for Block[4] first retrieves the block's successors (line 7) and propagates them to the following sibling (line 8). Then, it initializes the attribute successors for its enclosed statements (lines 11-13), adding its first successor, so that the enclosed statements have an "exit point" (i.e., this has the advantage that there is no need to check if an element is the last one of a block). Finally, the control flow instruction of a block is the control flow instruction of the first enclosed statement (line 16), that is, the flow reaches the block and goes on through the first statement. Please note that for a series of nested blocks this approach will seamlessly work. The cf_next attribute is thus used in the transformation with the purpose of attaching a control flow instruction (a FlowInstr element) to every element of the program tree, so that all elements can be homogenously treated as flow instructions even when some of them are not FlowInstr elements, as it happens in the rule for Block

[2] In practice, Eclectic treats both types of attributes equally, but it is useful to differentiate to improve readability.

[3] A better name would be *cf_instr*, since it does not represent the next control flow instruction (as cfNext does in the meta-model). The text, however, sticks to the name originally given in the solution uploaded to SHARE.

[4] The type ends with "!" meaning that only instances of this type, but no subtypes, should be matched.

The rule for SimpleStmt first propagates the successors to the immediate sibling (this operation has to be done in every rule). Then, it establishes that the flow instruction for the statement is itself (line 23). Finally, the cfNext link is the control flow instruction of its first successor.

```
1  attribution task2_attribution(flow) -> ()
2    inh successors : _!List
3    syn cf_next : flow!FlowInstr
4
5    rule flow!Block!
6      // Propagate the successors to immediate sibling
7      successors = successors[self]
8      successors[successors.first] <- successors.tail
9
10     // Initialize successors for the enclosed statements
11     successor = successors.first
12     successors[self.stmts.first] <-
13         self.stmts.tail.add(successor)
14
15     // Compute the control flow
16     cf_next[self] <- cf_next[self.stmts.first]
17   end
18
19   rule flow!SimpleStmt
20     successors = successors[self]
21     successors[successors.first] <- successors.tail
22
23     cf_next[self] <- self
24
25     next_flow = cf_next[successors.first]
26     self.cfNext = next_flow
27   end
```

Figure 2: Computing the flow graph: blocks and simple statements

2.3 Task 3.1

This task complements Task 1 by adding variable declarations to the FlowGraph models, and computing the information about definitions and uses of the variables.

Thus, this transformation module (an attribution transformation) depends on the mapping transformation, so that its rules retrieve objects created by the latter. To this end, the syntax transformation!tlink.tfeature[expr] is used, which means: "retrieve a trace link called *tlink* from *transformation*, corresponding to the source element obtained with *expr*". A more detailed explanation about this feature and the transformation itself is given in Appendix A.1.

2.4 Task 3.2

This task has been implemented using the straightforward algorithm commented in the case description, using the scripting language. It was not possible to use attribute grammars because Eclectic does not support circular dependencies yet. Basically, for each variable use in a flow instruction, each path to reach the instruction is looked up (using the cfPrev link). Then, for each path, every flow predecessor is computed in a helper method (*all_previous*). This works because *all_previous* returns the list of predecessors in order, so that if a variable is defined twice, the closest predecessor is the first in the list. The complete transformation is given in Appendix A.3.

2.5 Task 4

This task requires building a small DSL to allow validation specifications to be written. To this end the template language of METADEPTH [4] has been used. It allows concrete syntaxes to be created "on the fly" (with intermediate code generation, but it is handled internally). METADEPTH is a powerful multi-level modeling framework, but its use here is very simple, so it is not fully introduced.

The meta-model for the abstract syntax of the DSL is shown to the left of Figure 3. The model ValidationDSL acts as root element, which encloses RequiredLink elements. This meta-class simply specifies that an instruction identified in left must have the instruction identified in right as a successor. The ControlFlowLink and DataFlowLink meta-classes specialize RequiredLink for the control and data flow.

```
1  Model ValidationDSL@1 {
2    abstract Node RequiredLink {
3        left : String;
4        right : String;
5    }
6
7    Node ControlFlowLink : RequiredLink
8    { }
9
10   Node DataFlowLink : RequiredLink
11   { }
12 }
```

```
1  load "validation_dsl"
2  Syntax ValidationDSLSyntax for ValidationDSL [".validate"] {
3      model template ValidationDSL@1 for "ValidationDSL"
4          "validate" ^Id
5          (_:ControlFlowLinkTemplate)* (_:DataFlowLinkTemplate)* ;
6
7      node template ControlFlowLinkTemplate@1 for ControlFlowLink
8          "cfNext" ":" #left "-->" #right ;
9
10     node template DataFlowLinkTemplate@1 for DataFlowLink
11         "dfNext" ":" #left "-->" #right ;
12 }
```

Figure 3: Meta-model of the DSL (left). Template specification (right)

The right of Figure 3 shows the specification of the concrete syntax. It is a template language, based on associating a type with a specification of its serialization, which is later interpreted to generate a parser. For instance, _:ControlFlowLinkTemplate invokes a template (line 7) and #left (line 11) indicates the serialization of the left property.

The algorithm to check this specification against the generated models basically consists of two nested loops, for traversing the specification and the check model (see Appendix A.4).

3 Evaluation

All tasks have been solved, and the results for the smaller input models has been checked manually. The only issue detected, in Task 3.2, has been missing data flow links for unary expressions.

With respect to comprehensibility and conciseness, the table summarizes the use of the different languages of Eclectic and the amount of code written (LOC, including whitespace). As has been shown in the previous section, it was natural to combine different languages in order to favour modularity, and ultimately readability through expressive and concise specifications.

Task	Style	LOC
1	Mapping	87
	Attribute propagation	160
	Simple pattern matching	12
2	Attribute propagation	140
3.1	Attribute propagation	123
3.2	Scripting	40
4	Scripting	102
	MetaDepth (meta-model)	10
	MetaDepth (c. syntax)	10
Total		694

Finally, performance was not as good as expected. In particular, the control flow transformation did not scale well when large models were tried (notably tests 8 and 9). Therefore, a line of future work is to profile and optimize the transformation engine.

Acknowledgements. Work partially funded by the Spanish Ministry of Economy and Competitivity (TIN2011-24139), and the R&D programme of Madrid Region (S2009/TIC-1650).

References

[1] Jesús Sánchez Cuadrado (2012): *Towards a Family of Model Transformation Languages*. *LNCS* 7307, Springer, pp. 176–191, doi:10.1007/978-3-642-30476-7_12.

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

[3] Tassilo Horn (2013): *The TTC 2013 Flowgraphs Case*. In: *Sixth Transformation Tool Contest (TTC 2013)*, *EPTCS* this volume.

[4] Juan de Lara & Esther Guerra (2010): *Deep Meta-Modelling with* METADEPTH. *LNCS* 6141, Springer, pp. 1–20, doi:10.1007/978-3-642-13953-6_1.

A Complete code

A.1 Mapping to JaMoPP

The following listing shows the code that solves Task 1 and Task 3.1. It is split into four modules.

- A mapping module (task1_map, lines 3–87).
- An attribute computation module (task1_attribution, lines 89–249).
- An pattern matching module (task_patterns, lines 258–263)
- An attribute computation module (task3_1_varuses, lines 269–391)

As an implementation note, the expression language of Eclectic is currently very simple, for instance, it does not have binary expressions or if statements. The reasons is that it has not been decided yet which style to use: a conventional one or a Smalltalk-like (i.e., based on keyword methods). In any case, by using method calls and closures it is possible to express complex structures in practice (although not in a very readable manner, see for example lines 34–41 in Figure 6).

```
1  eclectic task1 (in) -> (out)
2
3  mappings task1_map(in) -> (out)
4      uses task1_attribution as task1_attribution
5      uses task_patterns as task_patterns
6
7      from src : in!ClassMethod
8        to tgt : out!Method, exit : out!Exit
9        linking tgt.exit = exit
10         tgt.stmts <- src.statements
11
12         tgt.txt = task1_attribution!text[src]
13         exit.txt = "Exit"
14     end
15
16     // ---------------
17     // Statements
18     // ---------------
19
20     from src : in!LocalVariableStatement
21       to tgt : out!SimpleStmt
22         tgt.txt = task1_attribution!text[src]
23     end
24
25     from src : in!ExpressionStatement
26       to tgt : out!SimpleStmt
27         tgt.txt = task1_attribution!text[src]
28     end
29
30     from src : in!WhileLoop
31       to tgt : out!Loop
32         tgt.expr <- src.condition
33         tgt.body <- src.statement
34
35         tgt.txt = task1_attribution!text[src]
36     end
37
38     from src : in!Condition
39       to tgt : out!If
40         tgt.txt = task1_attribution!text[src]
41
42         tgt.expr <- src.condition
43         tgt.then <- src.statement
44         tgt.^else <- src.elseStatement
45     end
46
47     from src : in!Return
48       to tgt : out!Return
49         tgt.txt = task1_attribution!text[src]
50     end
51
52     from src : in!Break
53       to tgt : out!Break
54         tgt.txt = task1_attribution!text[src]
55     end
56
57     from src : in!Continue
58       to tgt : out!Continue
59         tgt.txt = task1_attribution!text[src]
60     end
61
62     from src : in!JumpLabel
63       to tgt : out!Label
64         tgt.stmt <- src.statement
65         tgt.txt = task1_attribution!text[src]
66     end
67
68     from src : in!Block
69       to tgt : out!Block
70         tgt.stmts <- src.statements
71         tgt.txt = task1_attribution!text[src]
72     end
73
74     // ---------------
75     // Expressions
76     // ---------------
77     from src : task_patterns!ConditionalExpression
78       to tgt : out!Expr
79         tgt.txt = task1_attribution!text[src]
80     end
81
82     from src : task_patterns!LoopExpression
83       to tgt : out!Expr
84         tgt.txt = task1_attribution!text[src]
```

```
85      end
86
87  end
88
89  attribution task1_attribution(in) -> (out)
90  // optimizations : enabled
91    syn text : _!String
92
93    rule in!Method
94      text[self] <- self.name.concat('()')
95    end
96
97    rule in!LocalVariableStatement
98      init_text = text[self.variable.initialValue]
99      type_ref = text[self.variable.typeReference]
100
101     text[self] <- type_ref.concat(' ').concat(
102                     self.variable.name.concat(' = ').
103                     concat(init_text)).concat(';')
104   end
105
106   rule in!ExpressionStatement
107     init_text = text[self.expression]
108
109     text[self] <- init_text.concat(";")
110   end
111
112   rule in!AssignmentExpression
113     left = text[self.child]
114     right = text[self.value]
115     operator = text[self.assignmentOperator]
116
117     text[self] <- left.concat(' = ').concat(right)
118   end
119
120   rule in!SuffixUnaryModificationExpression
121     expr_text = text[self.child]
122     operator = text[self.operator]
123     text[self] <- expr_text.concat(operator)
124   end
125
126   rule in!MultiplicativeExpression
127     first = text[self.children.first]
128     rest = self.children.tail.zip(self.multiplicativeOperators)
129
130     text[self] <- rest.inject(first) { |tmp, v|
131       tmp.concat(text[v.second]).concat(text[v.first])
132     }
133   end
134
135   rule in!AdditiveExpression
136     first = text[self.children.first]
137     rest = self.children.tail.zip(self.additiveOperators)
138
139     text[self] <- rest.inject(first) { |tmp, v|
140       tmp.concat(text[v.second]).concat(text[v.first])
141     }
142   end
143
144   rule in!RelationExpression
145     first = text[self.children.first]
146     rest = self.children.tail.zip(self.relationOperators)
147
148     text[self] <- rest.inject(first) { |tmp, v|
149       tmp.concat(text[v.second]).concat(text[v.first])
150     }
151   end
152
153   rule in!EqualityExpression
154     first = text[self.children.first]
155     rest = self.children.tail.zip(self.equalityOperators)
156
157     text[self] <- rest.inject(first) { |tmp, v|
158       tmp.concat(text[v.second]).concat(text[v.first])
159     }
160   end
161
162   rule in!IdentifierReference
163     text[self] <- self.target.name
164   end
165
166   rule in!DecimalIntegerLiteral
167     text[self] <- self.decimalValue.to_s
168   end
169
170   rule in!WhileLoop
171     text[self] <- "while"
172   end
173
174   rule in!Condition
175     text[self] <- "if"
176   end
177
178   rule in!Block
179     text[self] <- "{...}"
180   end
181
182   rule in!Continue
183     text[self] <- "continue"
184   end
185
186   rule in!Break
187     text[self] <- "break"
188   end
189
190   rule in!Return
191     rvalue = self.returnValue.is_nil.if_else({
192       ';'
193     }, {
194       v = text[self.returnValue]
195       ' '.concat(v.concat(';'))
196     })
197     text[self] <- "return".concat(rvalue)
198   end
199
200   rule in!JumpLabel
201     text[self] <- self.name.concat(":")
202   end
203
204   // Types
205   rule in!Int
206     text[self] <- 'int'
207   end
208
209   // Operators
210   rule in!Assignment
211     text[self] <- ' = '
212   end
213
214   rule in!Multiplication
```

```
    text[self] <- ' * '
  end

  rule in!Addition
    text[self] <- ' + '
  end

  rule in!Division
    text[self] <- ' / '
  end

  rule in!Subtraction
    text[self] <- ' - '
  end

  rule in!Equal
    text[self] <- ' == '
  end
  rule in!GreaterThan
    text[self] <- ' > '
  end

  rule in!LessThan
    text[self] <- ' < '
  end

  rule in!PlusPlus
    text[self] <- '++'
  end

  rule in!MinusMinus
    text[self] <- '--'
  end

end

patterns task_patterns(in)
  def LoopExpression -> (e)
    l : in!WhileLoop {
      condition = e : in!Expression { }
    }
  end

  def ConditionalExpression -> (e)
    l : in!Conditional {
      condition = e : in!Expression { }
    }
  end
end

// ----------------------
// Task 3.1
// ----------------------

attribution task3_1_varuses(in) -> (out)
   uses task1_map as task1_map
    uses task_patterns as task_patterns

  inh vardef : out!Var
  syn writes : _!List
  syn reads : _!List

  // Create variables
  rule in!Method
    translation = task1_map!default.t[self]
  vars = self.parameters.map { |p|
    pvar = out!Param.new
    pvar.txt = p.name

    vardef[p] <- pvar

    pvar
  }
  translation.vars = vars
  translation.^def = vars
end

rule in!LocalVariableStatement
  avar = out!Var.new
  avar.txt = self.variable.name

  vardef[self.variable] <- avar

  translation = task1_map!default.t[self.up_to(in!Method)]
  translation.vars = avar
end

// Compute reads/writes for statements
rule in!ExpressionStatement
  reads = reads[self.expression]
  writes = writes[self.expression]

  translation = task1_map!default.t[self]
  translation.use = reads
  translation.^def = writes
end

rule in!LocalVariableStatement
  left = vardef[self.variable]
  reads = reads[self.variable.initialValue]
  writes = writes[self.variable.initialValue]

  translation = task1_map!default.t[self]
  translation.use = reads
  translation.^def = writes.add(left)
end

rule in!Return
  self.returnValue.is_nil.if_false {
    reads = reads[self.returnValue]
    translation = task1_map!default.t[self]
    translation.use = reads
  }
end

rule in!UnaryModificationExpression
  avar = vardef[self.child.target]
  writes[self] <- avar.as_list
  reads[self] <- avar.as_list
end

// Compute reads/writes for expressions
rule in!AssignmentExpression
  writes[self] <- vardef[self.child.target]
  reads[self] <- reads[self.value]
end

// covers ShiftExpression, AdditiveExpression,
      MultiplicativeExpression
rule in!RelationExpression
```

```
344     writes[self] <- self.children.map { |c| r = writes[c] }.
            flatten
345     reads[self] <- self.children.map { |c| r = reads[c] }.
            flatten
346   end
347
348   rule in!AdditiveExpression
349     writes[self] <- self.children.map { |c| r = writes[c] }.
            flatten
350     reads[self] <- self.children.map { |c| r = reads[c] }.
            flatten
351   end
352
353   rule in!MultiplicativeExpression
354     writes[self] <- self.children.map { |c| r = writes[c] }.
            flatten
355     reads[self] <- self.children.map { |c| r = reads[c] }.
            flatten
356   end
357
358   rule in!EqualityExpression
359     writes[self] <- self.children.map { |c| r = writes[c] }.
            flatten
360     reads[self] <- self.children.map { |c| r = reads[c] }.
            flatten
361   end
362
363   rule in!DecimalIntegerLiteral
364     writes[self] <- _!List.new
365     reads[self] <- _!List.new
366   end
367
368   rule in!IdentifierReference
369     writes[self] <- _!List.new
370     reads[self] <- vardef[self.target]
371   end
372
373   // Expressions
374   rule task_patterns!LoopExpression
375     translation = task1_map!default.t[self]
376     reads = reads[self]
377     writes = writes[self]
378
379     translation.use = reads
380     translation.^def = writes
381   end
382
383   rule task_patterns!ConditionalExpression
384     translation = task1_map!default.t[self]
385     reads = reads[self]
386     writes = writes[self]
387
388     translation.use = reads
389     translation.^def = writes
390   end
391 end
```

A.2 Computing the control flow

This transformation is perhaps the most complex one of the case, so to simplify the explanation, the complete transformation has been split into several listings. First, listing in Figure 4 shows the header of the transformation, including the attribute declarations (already explained in Section 2.2), and the rules for Method and Block.

The rule for Method, initializes the successors attribute for the first statement (line 7). It adds the exit element to the list of sucessors as a fallback, so that the sucessor of the last statement is the exit element (i.e., this has the advantage that there is no need to check if an element is the last one of a block). Besides, the control flow instruction of exit is itself. Lines 12–13 obtain the flow instruction for the first statement, and set the cfNext link.

The rule for Block is similar to Method[5], but first it retrieves the block's successors (line 19) and propagates them to the following sibling (line 20). Then, it initializes the successors attribute for its statements (lines 23-25), adding its first successor, so that the enclosed statements have an "exit point". Finally, the control flow instruction of a block, is the control flow instruction of the first enclosed statement (line 28). Please note that for a series of nested blocks this approach will seamlessly work.

Once the two basic enclosing structures have been presented, the easiest elements are simple statements (SimpleStmt) and returns (Return), which are addressed in the listing shown in Figure 5.

The rule for SimpleStmt first propagates the successors to the immediate sibling (this operation has to be done in every rule, so it will not be explained in the following). Then, it establishes that the flow instruction for the statement is itself (line 5). Finally, the cfNext link is the control flow instruction of its first successor.

In contrast, the rule for Return needs to look up the Method in which the instruction is enclosed, in

[5]The type ends with "!" meaning that only instances of this type, but no subtypes, should be matched.

```
1  attribution task2_attribution(flow) -> ()
2    inh successors : _!List
3    syn cf_next : flow!FlowInstr
4
5    rule flow!Method
6      // Initialize sucessors for enclosed stmts
7      successors[self.stmts.first] <-
8          self.stmts.tail.add(self.exit)
9
10     cf_next[self.exit] <- self.exit
11
12     // Set flow link with the first flow instruction
13     next_flow = cf_next[self.stmts.first]
14     self.cfNext = next_flow
15   end
16
17   rule flow!Block!
18     // Propagate the successors to immediate sibling
19     successors = successors[self]
20     successors[successors.first] <- successors.tail
21
22     // Initialize sucessors for the enclosed statements
23     successor = successors.first
24     successors[self.stmts.first] <-
25         self.stmts.tail.add(successor)
26
27     // Compute the control flow
28     cf_next[self] <- cf_next[self.stmts.first]
29   end
```

Figure 4: Computing the flow graph: methods and blocks

```
1  rule flow!SimpleStmt
2    successors = successors[self]
3    successors[successors.first] <- successors.tail
4
5    cf_next[self] <- self
6
7    next_flow = cf_next[successors.first]
8    self.cfNext = next_flow
9  end

10 rule flow!Return
11   successors = successors[self]
12   successors[successors.first] <- successors.tail
13
14   cf_next[self] <- self
15
16   method = self.up_to(flow!Method)
17   self.cfNext = method.exit
18 end
```

Figure 5: Computing the flow graph: simple statements and return

order to set the cfNext link to the method's exit element (lines 16–17). The up_to facility returns the first ancestor with the given type.

The approach for *loops* and *conditionals* follows a similar schema, but taking into account that the actual flow instruction is their condition, as well as the particularities of each instruction. The solution is shown in the listing of Figure 6.

In the case of Loop, the successors attribute for its body has to be the condition expression, that is, the control flow successor of the loop's last statement will be the loop's condition (lines 5–6). The control flow instruction of the loop is its condition, and the control flow of the condition is itself (this is needed because other instructions will refer to the control flow instruction of the condition as it has been designated the successor of the loop). Finally, the cfNext link is set to the next successor as usual, but also to the first enclosed flow instruction (lines 11–15).

The solution for conditionals (meta-class If, lines 22–42) is conceptually easier. The successors of the then part are the if's successors (line 26), the flow instruction is its condition (line 29) and the successor of the condition is the instruction within the then (lines 31–32). Finally, it requires checking whether there is an *else* part (line 34)[6]. If not, the next control flow instruction is just the following successor (lines 35–36). Otherwise, the successor attribute has to be initialized for the else part, and the next control flow instruction is the one within the then part (lines 38–40).

Finally, rules to deal with Break and Continue statements (including Labels) are introduced. In both cases, the key issue is to determine the jump location, which will be different depending on whether there is a label or not. The listing in Figure 7 shows the solution.

In the case of a Break, the jump location is the enclosing loop or the label (lines 8-12). Then, the next

[6]This syntax for conditionals is only a syntatic limitation, as the current expression language is kept to a minimum.

```
1  rule flow!Loop
2    successors = successors[self]
3    successors[successors.first] <- successors.tail
4
5    condition = self.expr
6    successors[self.body] <- condition.as_list
7
8    cf_next[self] <- condition
9    cf_next[condition] <- condition
10
11   next_flow = cf_next[successors.first]
12   condition.cfNext = next_flow
13
14   first_within = cf_next[self.body]
15   condition.cfNext = first_within
16 end
17
18
19
20
21
22 rule flow!If
23   successors = successors[self]
24   successors[successors.first] <- successors.tail
25
26   successors[self.then] <- successors
27
28   condition = self.expr
29   cf_next[self] <- condition
30
31   first_then = cf_next[self.then]
32   condition.cfNext = first_then
33
34   self.else.is_nil.if_else({
35     next_flow = cf_next[successors.first]
36     condition.cfNext = next_flow
37   }, {
38     successors[self.else] <- successors
39     first_within = cf_next[self.else]
40     condition.cfNext = first_within
41   })
42 end
```

Figure 6: Computing the flow graph: loops and conditionals

flow instruction is simply the successor of the jump location (lines 14–16).

In the case of a Continue, the jump location is assumed to be the condition expression of a loop, either the enclosing loop or a loop with a label assigned (lines 25–30). Thus, the next flow instruction is just this expression (line 32).

Finally, for a Label the control flow instruction is the control flow instruction of the statement that it is labelling (line 41).

```
1
2  rule flow!Break
3    successors = successors[self]
4    successors[successors.first] <- successors.tail
5
6    cf_next[self] <- self
7
8    jump_location = self.label.is_nil.if_else({
9      self.up_to(flow!Loop)
10   }, {
11     self.label
12   })
13
14   break_successors = successors[jump_location]
15   next_flow = cf_next[break_successors.first]
16   self.cfNext = next_flow
17 end
18
19 rule flow!Continue
20   successors = successors[self]
21   successors[successors.first] <- successors.tail
22
23   cf_next[self] <- self
24
25   expr = self.label.is_nil.if_else({
26     loop = self.up_to(flow!Loop)
27     loop.expr
28   }, {
29     self.label.stmt.expr
30   })
31
32   self.cfNext = expr
33 end
34
35 rule flow!Label
36   successors = successors[self]
37   successors[successors.first] <- successors.tail
38   successors[self.stmt] <- successors
39
40   cf_next[self] <- cf_next[self.stmt]
41 end
```

Figure 7: Computing the flow graph: break and continue

```
1  navigation task3_2_navigation(flow)
2
3      def flow!FlowInstr.all_previous
4        visited_map = _!Map.new.^put(self, true)
5        self.all_previous_aux(visited_map)
6      end
7
8      def flow!FlowInstr.all_previous_aux(visited)
9        not_visited = self.cfPrev.
10           reject { |p| visited.include(p) }
11
12       previous = not_visited.map { |p|
13         p.all_previous_aux(visited.^put(p, true))
14       }.flatten
15
16       self.as_list.concat(previous.concat(not_visited))
17     end
18
19 end
20
21
22 attribution task3_2_attribution(flow) -> ()
23   uses task3_2_navigation
24
25   rule flow!FlowInstr
26     self.use.each { |v|
27       // Look in each of the paths
28       self.cfPrev.each { |i|
29         def_instruction = i.all_previous.select { |prev|
30           prev.^def.include(v)
31         }.first
32
33         def_instruction.dfNext = self
34       }
35
36       self.^def.include(v).if_true {
37         self.dfNext = self
38       }
39     }
40   end
41 end
```

Figure 8: Computing the data flow

A.3 Computing the data flow

The listing in Figure 8 shows the implementation of this task. There is a navigation module task3_2_navigation which adds the method all_previous to FlowInstr elements, so that it can be used by task3_2_attribution to set the data flow links.

It is worth mentioning that a solution based on attribute propagation, following the algorithm proposed in the Dragon Book was tried, but it requires circular attributes, which are currently not supported in Eclectic. Nevertheless, this solution shows that navigation modules are also possible, as well as scripting-based transformations.

A.4 Checking control and data flow models

The comparison of the control of the data flow models against the validation specification expressed with the DSL created in Section 2.5 has been implemented with the Eclectic low-level scripting language. Interestingly, the Eclectic high-level languages are compiled to a representation similar to this one, so this explanation may serve to give the reader an intuition of how Eclectic works under the hood.

The program shown in Listing 9 takes two input models, the specification written with the DSL and the flow graph model. It outputs a report model (actually, the current implementation just prints the reports, but it will be straightforward to create elements of the report model).

The scripting transformation allows temporary data structures to be defined, which serve as intermediate data for the transformation. In this way, lines 2–7 defines a model called inline, with the FlowLink class. This class will hold a control flow or data flow relationship in the form of a string representing the source element and another string representing the target element.

Afterwards, queues are defined. In the scripting language (and in IDC, the intermediate representation used by Eclectic) communication happens through queues. A model queue (lines 9–11) declares the interest of a transformation in a certain type. A local queue (lines 13–17) is used internally by communicating values between two places of the transformation. The flow_cfLinks and flow_dfLinks will contain links appearing in the flow model, and the dsl_cfLinks and dsl_dfLinks will contain links appearing in the

DSL specification.

The transformation code can be logically organised into segments. In this way, the find_flow_links segment (line 19) contains code to find flow links. The forall instruction is able to receive elements of a queue (e.g., line 20). The emit instruction sends an object to a queue, in particular it is used to send FlowLink elements when a link is found (e.g., line 25). This is the basic communication mechanism between patterns and rules (although in this language the distinction is implicit).

Then, segment validate (lines 52–98) receives the notifications of the found flow links (through the four local queues) and check false links and missing links. As Eclectic has full support for closures, it is possible to declare a closure as if it were a local variable, acting as kind of local function. This is done, for example, in lines 53–60 to create a facility to check false links.

```
scripting task4_script(dsl, flow) -> (report)
  model inline
    class FlowLink
      ref source : _!String
      ref target : _!String
    end
  end

  model queue mFlowInstr : flow!FlowInstr
  model queue mControlFlowLink : dsl!ControlFlowLink
  model queue mDataFlowLink : dsl!DataFlowLink

  local queue flow_cfLinks : inline!FlowLink
  local queue flow_dfLinks : inline!FlowLink

  local queue dsl_cfLinks : inline!FlowLink
  local queue dsl_dfLinks : inline!FlowLink

  segment find_flow_links
    forall flow_instr from mFlowInstr
      flow_instr.cfNext.each { |target|
        lnk = inline!FlowLink.new
        lnk.source = flow_instr.txt
        lnk.target = target.txt
        emit lnk to flow_cfLinks
      }

      flow_instr.dfNext.each { |target|
        lnk = inline!FlowLink.new
        lnk.source = flow_instr.txt
        lnk.target = target.txt
        emit lnk to flow_dfLinks
      }
    end

    forall control_flow from mControlFlowLink
      lnk = inline!FlowLink.new
      lnk.source = control_flow.left
      lnk.target = control_flow.right
      emit lnk to dsl_cfLinks
    end

    forall data_flow from mDataFlowLink
      lnk = inline!FlowLink.new
      lnk.source = data_flow.left
      lnk.target = data_flow.right
      emit lnk to dsl_dfLinks
    end

  end

  segment validate
    check_false_link = { |type, lnk, dsl_links|
      dsl_links.find { |cf|
        cf.left.eq(lnk.source).and(
          cf.right.eq(lnk.target))
      }.if_nil {
        lnk.source.concat(' ==> ').concat(lnk.target).
            println(type.concat(" false link: "))
      }
    }

    dsl_expected_cfs = dsl!ControlFlowLink.all_instances
    dsl_expected_dfs = dsl!DataFlowLink.all_instances

    // For any cfNext or dfNext link in the model,
    // check if it is also defined in the spec.
    forall cfLink from flow_cfLinks
      check_false_link.call('Control', cfLink, dsl_expected_cfs)
    end

    forall dfLink from flow_dfLinks
      check_false_link.call('Data', dfLink, dsl_expected_dfs)
    end

    // Check that every link in the specification
    // occurs in the flow graph
    flow_instrs = flow!FlowInstr.all_instances

    check_missing_link = { |type, lnk, featureName|
      flow_instrs.find { |fi|
        next_txt = fi.get(featureName).map { |n| n.txt }

        fi.txt.eq(lnk.source).
        and(next_txt.include(lnk.target))
      }.if_nil {
        lnk.source.concat(' ==> ').concat(lnk.target).
            println(type.concat(" missing link: "))
      }
    }

    forall dsl_cfLink from dsl_cfLinks
      check_missing_link.call('Control', dsl_cfLink, 'cfNext')
    end

    forall dsl_dfLink from dsl_dfLinks
      check_missing_link.call('Data', dsl_dfLink, 'dfNext')
    end
  end
end
```

Figure 9: Validating the flow graph using the scripting language