An Epsilon Solution to the Flowgraphs Case

R. Wei Adolfo Sánchez-Barbudo Herrera Babajide Ogunyomi Louis M. Rose
Dimitrios S. Kolovos
Department of Computer Science, University of York, UK.
[rw542,asbh500,bjo500,louis.rose, dimitris.kolovos]@york.ac.uk

We summarise our solution to the Flowgraphs case of the 2013 Transformation Tool Contest, implemented in Epsilon – an extensible platform of integrated and task-specific languages for model management. Our solution uses the Epsilon Object Language (an extension and reworking of OCL), the Epsilon Transformation Language (for model-to-model transformations), the Epsilon Comparison Language (for model comparison), and EUnit (for testing model management operations).

1 Solution Description

Our solution[1] to the Flowgraphs case of the 2013 Transformation Tool has been implemented in Epsilon. For task 4, we have used Xtext[2] in addition to Epsilon. We wrote no code in a general-purpose programming language (e.g., Java). We now describe our solution, assuming that the reader is already familiar with the Flowgraphs[3] case.

1.1 Task 1: Structure Graph

We approached task 1 by constructing a model transformation in the Epsilon Transformation Language (ETL) comprising two parts. Firstly, we use the declarative transformation rules of ETL to specify how elements of the JaMoPP (Java) metamodel are transformed to elements of the Flowgraph metamodel. Secondly, we embed a model-to-text transformation that populates the `txt` property of all of the constructed Flowgraph model elements in the model-to-model transformation. The model-to-text transformation is implemented as a visitor transformation via a set of operations that implement a pretty printer for each element of the JaMoPP metamodel.

It is interesting to note that while Epsilon includes a dedicated model-to-text transformation language (EGL), we implemented the model-to-text transformation directly with ETL. We made this design decision as EGL provides many additional features that we did not need, such as support for generating multiple files, preserving hand-written changes to generated text, and formatting of generated text.

1.2 Task 2: Control Flow Graph

We approached task 2 by constructing an endogenous model transformation in the Epsilon Object Language (EOL). Our solution is implemented as a visitor transformation via a set of operations that implement control flow analysis for each element of the Flowgraphs metamodel. We also defined a set of operations to assist in traversing the Flowgraphs model and, in particular, to avoid visiting constructs

[1] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_Epsilon_Flowgraphs.vdi`

[2] `http://www.eclipse.org/Xtext`

[3] Tassilo Horn, The TTC 2013 Flowgraphs Case, in proceedings of Sixth Transformation Tool Contest 2013, EPTCS 2013

Submitted to:
TTC 2013

```
1 operation Loop process(next: Item) {
2   self.~exit = next;
3   stack.push(self);
4
5   self.expr.process(self.body.traverse());
6   self.body.process(self.expr);
7
8   self.expr.cfNext.add(next.traverse());
9 }
```

Listing 1: Performing control flow analysis for a Loop element with EOL.

that do not control the flow of the program. For example the `Block` and `Conditional` constructs are merely containers for other constructs that control flow, and so we do not visit them when performing control flow analysis.

The visitor operations, all named `process`, add control flow information to the flowgraphs model. Listing 1 demonstrates our approach for the `Loop` construct of the Flowgraphs metamodel. Each of our `process` operations performs some initialisation (lines 2-3), processes any nested constructs (lines 5-6) and updates any control flow information for this element (line 8). Note that each process operation receives as an argument the `next` construct that will be visited after this one.

Initialisation of the process operation for `Loop` involves storing the `next` construct in an extended property[4] called `exit` (line 2). The process operations for jump constructs make use of the `exit` property. Additionally we push the current `Loop` onto a stack (line 3) that we can use when analysing the control flow of jump statements that contain no explicit target. Similar initialisation steps are performed for other constructs that can be the (effective) target of a jump instruction (`Label`) and for other constructs that push a new frame stack when executed (`Label`, `If`, `Method`), respectively.

When processing a construct, we also process its nested elements. In the case of a `Loop`, we process its condition (nested `Expr`) passing the (first statement of the) body of the loop as the `next` construct (line 5). Similarly, we process the body of the `Loop`, passing the condition as the `next` construct (line 6).

Eventually, calls to the process operations will result in an update to the `cfNext` reference of a construct. In the case of a `Loop`, the control flow of the condition is updated to include `next` (line 8), i.e., the construct that immediately follows the `Loop` (to indicate the control flow of the program when the condition is not satisfied and the loop terminates).

1.3 Task 3: Data Flow Graph

We approached task 3 by extending our ETL JaMoPP to flowgraphs transformation from task 1, and by constructing an endogenous model transformation in EOL. The former involved extending the transformation to also emit variable and parameter program constructs. We have approached this task by extending our ETL transformation from task 1 with two additional rules, and performing a series of post-processing steps that visit each JaMoPP construct and initiate an analysis that traverses nested constructs to determine which variables are read and/or written by that construct.

The latter involved updating the flow instructions in the flowgraph to indicate the way in which data flows through the program. We have approached task 3.2 by constructing an endogenous model transformation in EOL that implements the inefficient algorithm $\in O(n^2)$. In particular, our implementation

[4] EOL's extended properties allow arbitrary metamodel extension at runtime.

```
1  for (root in FlowGraph!FlowInstr.all) {
2    for (variable in root.use) {
3      root.searchPredecessors(root, variable);
4    }
5  }
6
7  operation FlowInstr searchPredecessors(root: FlowGraph!FlowInstr, variable: FlowGraph!Var
       ) {
8    self.cfPrev.forAll(predecessor|predecessor.search(root,variable));
9  }
10
11 operation Continue search(root: FlowGraph!FlowInstr, variable: FlowGraph!Var) {
12   if (self.~flag.isUndefined()) {
13     self.~flag = true;
14     self.searchPredecessors(root, variable);
15     self.~flag = null;
16   }
17 }
18
19 operation FlowInstr search(root: FlowGraph!FlowInstr, variable: FlowGraph!Var) {
20   if (self.def.includes(variable)) {
21     self.dfNext.add(root);
22
23   } else if (self <> root and self.cfPrev.isDefined()) {
24     self.searchPredecessors(root, variable);
25   }
26 }
```

Listing 2: Performing data flow analysis with EOL.

(Listing 2) is a recursive algorithm that is executed for every pair $\langle root, variable \rangle$ where *root* is a flow instruction and *variable* is a variable that is read by *root*. The algorithm recurses over each predecessor of *root* (lines 7-9) until a flow instruction that writes to *variable* is located (lines 20-22) or until there are no further flow instructions to visit (lines 23-25). The recursive case is slightly altered for the `Continue` statement by using a flag (lines 11-17). The flag ensures that predecessors of a continue statement are not repeatedly visited and ensures that the search is able to traverse other control flows.

1.4 Task 4: Validation

Task 4 involved providing mechanisms for allowing non-MDE experts to validate the output of the transformations constructed for task 2 and task 3. We have approached this task by using Xtext to implement the domain-specific language introduced in the case description, by implementing a model-to-model transformation between the flowgraphs model and our DSL, and by using Epsilon's testing framework, EUnit, to compare models constructed in the DSL with models produced by our transformation. We have used a domain-specific comparison implemented in ECL to ensure that the ordering of the links is not significant in the comparison.

1.4.1 Implementing the DSL with Xtext

As Epsilon does not provide built-in support for constructing textual domain-specific languages, we have used Xtext[5] for this purpose. Our Xtext grammar is straightforward (4 grammar rules), and is consistent with the specification outlined in the case description.

[5] `http://www.eclipse.org/Xtext`

```
1  pre {
2    var fgvModel = new FlowGraphValidation!FGVModel;
3  }
4
5  @greedy
6  rule ControlFlowLinks
7    transform s : FlowGraph!FlowInstr
8    to t : Sequence(FlowGraphValidation!CFLink) {
9
10   for (cfNext in s.cfNext) {
11     var newLink = new FlowGraphValidation!CFLink;
12     newLink.prev = s.txt;
13     newLink.next = cfNext.txt;
14     fgvModel.cfLinks.add(newLink);
15   }
16 }
```

Listing 3: Creating cfLinks with ETL.

1.4.2 Transforming to flowgraph validation models

To facilitate comparison between models written in the DSL with models produced by the transformation chain from previous tasks, we have implemented a model-to-model transformation that consumes models produced by the transformation chain and emits an equivalent model conforming to the validation DSL (listing 3). The transformation maps each flow instruction, s, to a sequence of `CFLinks` (lines 5-16) and `DFLinks` (not shown). For every pair $\langle s, cfNext \rangle$ where $cfNext$ is a control flow successor of s, an instance of `CFLink` is created (line 11) between s (line 12) and $cfNext$ (line 13). Similar logic is executed for `DFLinks`.

Note that we use a `@greedy` annotation (line 5) to force the transformation rule to be executed for all sub-types of `FlowInstr`. (Without the annotation, the rule would be executed only for those model elements that directly instantiate `FlowInstr`). It is also interesting to note that the transformation would be more concise if ETL supported more sophisticated pattern matching.

1.4.3 Comparing flowgraph validation models

To facilitate comparision between flowgraph validation models, we have used the Epsilon testing framework, EUnit. In particular, we have implemented domain-specific comparison logic with ECL (listing 4). An ECL program is executed on two models (normally called $left$ and $right$), a `matchTrace` which contains a set of triples $\langle l, r, matching \rangle$ where $l \in left$, $r \in right$ and $matching \in \{true, false\}$. The value of $matching$ is determined by executing the rules defined in the ECL program on each pair $\langle l, r \rangle$. In this case, we state that two links are matched if and only if their `prev` and `next` features are the same (lines 5 and 12).

We needed to implement domain-specific comparison logic because the domain-independent comparison logic in EMF incorrectly attempts to match model elements whose `prev` attributes are the same but whose `next` attributes are not. This is problematic when links in the two models are equivalent but appear in a different order.

We have implemented an EUnit test case that processes the match trace (set of triples $\langle l, r, matching \rangle$) produced by the ECL comparison and presents appropriate error messages to the user. In particular, we notify the user of any missing links (i.e., links that are defined in the expected model but do not appear in the obtained model) and any extra links (i.e., links that are defined in the obtained model but do not

```
rule CFLink
  match l : Left!CFLink
  with r : Right!CFLink {

  compare : l.prev = r.prev and l.next = r.next
}

rule DFLink
  match l : Left!DFLink
  with r : Right!DFLink {

  compare : l.prev = r.prev and l.next = r.next
}
```

Listing 4: Comparing validation models with ECL.

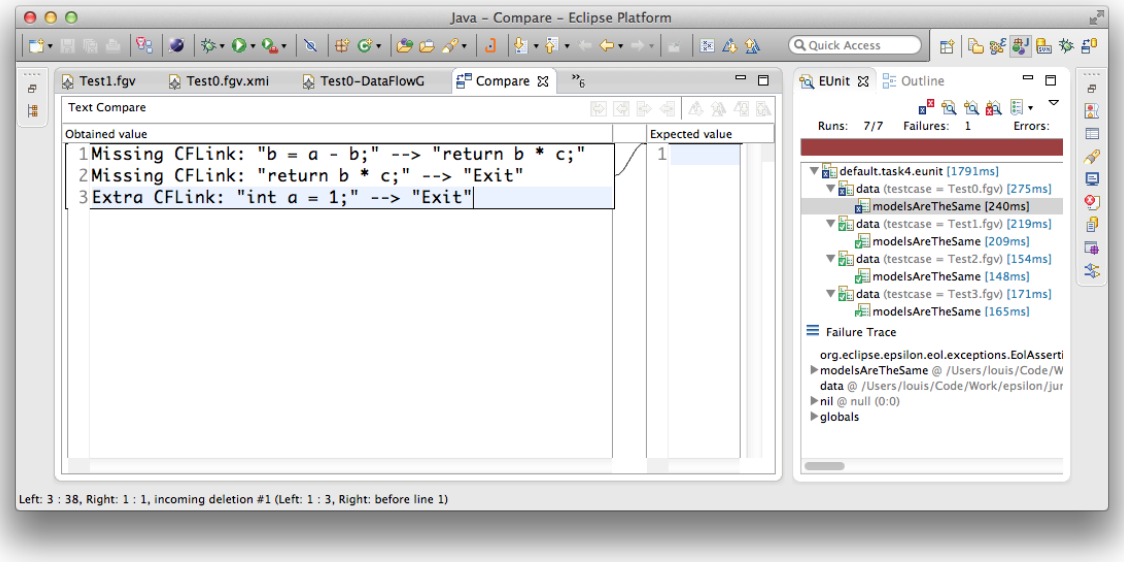

Figure 1: Showing the differences between two validation models with EUnit.

appear in the expected model). The results are shown in EUnit's comparison view (figure 1).

2 Evaluation

Reviewers noted that our solution was concise and understandable. Furthermore, our solution was awarded the overall prize for the best solution to the Flowgraphs case at the TTC workshop. Notwithstanding these positive results, performance and in particular scalability are areas in which our solution lags behind others, and we plan to address this in future versions of Epsilon.

Acknowledgements

Our work was funded in part by the EPSRC via the LSCITS (Large-Scale Complex IT Systems) initiative.