Hello World with Epsilon

Louis M. Rose\textsuperscript{1}, Antonio García-Domínguez\textsuperscript{2}, James R. Williams\textsuperscript{1}, Dimitrios S. Kolovos\textsuperscript{1}, Richard F. Paige\textsuperscript{1}, and Fiona A.C. Polack\textsuperscript{1}

\textsuperscript{1} Department of Computer Science, University of York, UK. [louis,jw,dkolovos,paige,fiona]@cs.york.ac.uk
\textsuperscript{2} University of Cádiz, Department of Computer Languages and Systems, C/Chile 1, 11002, Cádiz, Spain antonio.garciadominguez@uca.es

Abstract. Epsilon is an extensible platform of integrated and task-specific languages for model management. With solutions to the 2011 TTC Hello World case, this paper demonstrates some of the key features of the Epsilon Object Language (an extension and reworking of OCL), which is at the core of Epsilon. In addition, the paper introduces several of the task-specific languages provided by Epsilon including the Epsilon Generation Language (for model-to-text transformation), the Epsilon Validation Language (for model validation) and Epsilon Flock (for model migration).

1 Introduction

This paper presents a solution to the 2011 TTC Hello World case that uses Epsilon, an extensible platform of integrated and task-specific languages for model transformation, validation, merging, comparison, refactoring and migration \cite{1}. To provide an introduction to Epsilon for new users, solutions to each of the problems presented in the case are described and some of the key features of Epsilon and several of its task-specific languages are demonstrated. First, Epsilon is briefly described, and then solutions to each of the problems in the case are presented.

1.1 Epsilon

Epsilon is built atop Eclipse, and interoperates seamlessly with several modelling technologies, including EMF, MDR, CZT and plain XML. Further information on Epsilon can be found on the project website\textsuperscript{3} and in the Epsilon book\textsuperscript{4}.

The Epsilon Object Language (EOL) \cite{2} is at the core of the Epsilon and provides functionality similar to that of OCL. However, EOL provides an extended feature set, which includes the ability to update models, access to multiple models, conditional and loop statements, statement sequencing, and provision of standard output and error streams.

\begin{thebibliography}{9}
\bibitem{1} Department of Computer Science, University of York, UK. [louis,jw,dkolovos,paige,fiona]@cs.york.ac.uk
\bibitem{2} University of Cádiz, Department of Computer Languages and Systems, C/Chile 1, 11002, Cádiz, Spain antonio.garciadominguez@uca.es
\end{thebibliography}
2 Task 1: Greeting with EOL and EGL

EOL has been used to instantiate the Hello World metamodels [4, Section 2.1]. In
EOL, model elements are created using the new keyword and a model element’s
attributes are accessed with dot notation. The EOL program in Listing 1.1 instan-
tiates the Greeting metamodel type (line 1) and sets the text feature of the Greeting (line 2). References are accessed in the same manner as attributes: using dot notation (see lines 4 and 7 of Listing 1.2), for example.

EOL is dynamically typed. Consequently, EOL programs do not specify to
which metamodel particular types belong (e.g. Greeting). Instead, the user can
specify the models (and hence metamodels) on which the EOL program operates
just before executing the program. When using the Eclipse-based development
tools, for example, the user specifies the name, type, location and metamodel of
each model when creating an Eclipse launch configuration.

```
1 var g = new Greeting;
2 g.text = "Hello World";
```

**Listing 1.1.** Instantiating the simple Hello World metamodel with EOL.

```
1 var g = new Greeting;
2 3 g.greetingMessage = new GreetingMessage;
4 g.greetingMessage.text = "Hello";
5 6 g.person = new Person;
7 g.person.name = "TTC Participants";
```

**Listing 1.2.** Instantiating the extended Hello World metamodel EOL.

For model-to-text transformation, Epsilon provides a dedicated task-specific
language, the Epsilon Generation Language (EGL) [6]. EGL is a template-based
language that provides dynamic and static sections. The former contain EOL
code which is used to extract data from input models; while the latter contain
plain text which remains constant for varying input models. Dynamic sections
can emit text using the `out.print` operation. Alternatively, the `[%=text%]`
syntactic shortcut can be used (instead of `[% out.print(text); %]`).

Listing 1.3 uses three dynamic sections (contained in `[% %]` tags) and two
static sections (containing a space and an exclamation mark, respectively, on line
2). The dynamic section on line 1 uses the `all` (available on every metamodel
type and used to retrieve all of the instances of the that metamodel type) and
`first` (used to retrieve the first element of a collection) properties to find the
sole instance of Greeting in the input model. The two dynamic sections on line
2 emit the value of the `greetingMessage.text` and `person.name` features
of the Greeting, respectively. Executing the template in Listing 1.3 emits a
string of the form `<text> <name>!` (such as `Hello Franz!`).

```
1 [% var g = Greeting.all.first; %]
2 [%=g.greetingMessage.text%] [%=g.person.name%!]
```

**Listing 1.3.** A model-to-text transformation with EGL.
3 Task 2: Counting with EOL

The all property (discussed above) returns a collection containing all of the instances of the specified metamodel type. In EOL, the number of elements in a collection is determined using the size property. Hence, line 1 of Listing 1.4 prints the number of Nodes in the input model. Like OCL, EOL provides a number of higher-order operations for collection types. The select operation, for example, returns a filtered copy a collection containing only those elements that satisfy the specified predicate. Therefore, line 2 of Listing 1.4 prints the number of Edges whose src and trg features are equal (i.e. looping edges). Line 3 performs a similar query for counting isolated nodes using a user-defined operation.

User-defined operations allow existing (primitive or metamodel) types to be enriched with additional functionality. For example, lines 5-7 of Listing 1.4 define an operation, isIsolated, for the Node type. The isIsolated operation returns a Boolean value. The body of the isIsolated operation uses another of EOL’s higher-order operations for collections, exists, which implements existential quantification.

```
1 Node.all.size.println();
2 Edge.all.select(e|e.src == e.trg).size.println();
3 Node.all.select(n|n.isIsolated()).size.println();
4 operation Node isIsolated() : Boolean {
5     return not Edge.all.exists(e|e.src == self or e.trg == self);
6 }
```

Listing 1.4. Counting nodes, looping edges and isolated nodes with EOL.

Finding patterns of more than one or two model elements in EOL is complicated to specify in terms of the higher-order operations on collections, and hence imperative programming constructs are typically used instead. For example, finding cycles of three Nodes involves performing a depth-first search over the successors of each Node in the model (Listing 1.5). The successors() operation is not shown, for brevity, but traverses the outgoing Edges of a Node to identify successor Nodes.

```
1 var results : Sequence;
2 3 for (node1 in Node.all) {
4     for (node2 in node1.successors()) {
5         if (node2 == node1) continue;
6         for (node3 in node2.successors()) {
7             if (node3 == node2 or node3 == node1) continue;
8                 if (node3.successors().contains(node1)) {
9                     results.add(Sequence { node1, node2, node3 });
10                 }
11             }
12         }
13     }
14 results.println();
```

Listing 1.5. Counting circles of three nodes in EOL.

Section A.1 describes the optional task of checking for dangling edges.
4 Task 3: Reversing with EOL and Epsilon Flock

Epsilon programs are granted read-only, write-only or read-write access to a particular model by the user. As such, EOL can be used for querying models (read-only), constructing models (write-only) or modifying models (read-write). For reversing all of the Edges in a model, EOL has been used to specify an in-place update transformation (i.e. read-write access) on the input model (Listing 1.6). The for construct is used to iterate over the Nodes in the input model. EOL does not support parallel assignment, so a temporary (temp) is used.

```plaintext
for (edge in Edge.all) {
    var temp = edge.src;
    edge.src = edge.trg;
    edge.trg = temp;
}
```

Listing 1.6. Reversing edges with EOL.

Alternative, an Epsilon Flock [5] migration strategy can be used to reverse edges. Compared to the EOL solution (Listing 1.6), the iteration is performed declaratively (migrate Edge on line 1 of Listing 1.7) rather than imperatively (with a for loop), and no temporary variable is required because Flock provides the original and migrated model elements, which are bound to distinct model elements. Epsilon Flock is discussed further in Section 5.

```plaintext
migrate Edge {
    migrated.src = original.trg.equivalent();
    migrated.trg = original.src.equivalent();
}
```

Listing 1.7. Reversing edges with Flock.

5 Task 4: Migrating with Epsilon Flock

Epsilon provides a dedicated task-specific language for performing model migration, Epsilon Flock [5]. Metamodel evolution typically involves changes to a small proportion of a metamodel [7], and Flock exploits this by automatically copying model elements that have not been affected by metamodel evolution. Migration rules are specified only for those model elements that have been affected by metamodel evolution. For example, the case description describes a metamodel evolution in which the Graph type merges its nodes and edges reference to form a new gcs reference, and the name property of the Node class is renamed to text.

Flock migration rules are specified on a particular source metamodel type. Each rule is executed once for each instance of that type in the source model. In the body of a rule, the original and migrated variables are bound to an element of the source model and its equivalent element in the target model, respectively. Listing 1.8 demonstrates two migration rules: the first (lines 1-4) copies the contents of the nodes and edges references into the gcs reference for instances of Graph; and the second (lines 6-8) copies the value of the name feature into the text for instances of Node.
Listing 1.8. Migrating to the evolved graph metamodel with Flock.

Section A.2 describes the optional task of migrating to the even more evolved graph metamodel.

6 Task 5: Deleting with EOL

As discussed in Section 4, EOL programs can be used to perform in-place update transformations. Deleting a model element is possible with the delete keyword. Deleting the node with name n1 (Listing 1.9) has been achieved using the selectOne higher-order operation (a shorthand for select followed by first) to locate the relevant node, and using the delete keyword.

Listing 1.9. Deleting a node with EOL.

Section A.3 describes the optional task of deleting a node and its edges.

References

A Optional Tasks

Solutions to the optional tasks of the case are now described.

A.1 Task 2.5: Checking for dangling edges with EVL

Counting the dangling edges in a model can be formulated in the same manner as counting isolated nodes (Section 3). However, the Epsilon Validation Language (EVL) [3] can be used to check for – and reconcile – dangling edges by specifying a validation constraint and a corresponding fix (an in-place transformations that reconcile validation problems), as shown in Listing 1.10. An EVL constraint (line 2) is specified in the context (line 1) of a particular metamodel type (Edge in Listing 1.10). When the check (line 3) part of a constraint is not satisfied (returns false), the user is presented with the message part (line 4) of the constraint and can optionally invoke one of the fix parts (lines 5-11). The constraint in Listing 1.10 provides one fix, which deletes the dangling edge from the model. EVL executes the DanglingEdges constraint once for every instance of Edge in the input model.

1  context Edge {
2   constraint DanglingEdges {
3     check: not self.isDangling()
4     message: "The edge " + self + " is dangling."
5     fix {
6       title: "Remove this edge"
7       do {
8         delete self;
9       }
10     }
11   }
12  }
13
14  operation Edge isDangling() : Boolean {
15    return self.src.isUndefined() or self.trg.isUndefined();
16  }
17

Listing 1.10. Checking for dangling edges with EVL.

A.2 Task 4.2: Migrating to the even more evolved graph metamodel with Epsilon Flock

The case describes a second metamodel evolution in which edges are specified with reference values rather than model elements. The evolved metamodel no longer contains an Edge class, and instead the Node class references itself via the linksTo reference. Listing 1.11 demonstrates the way in which this migration can be specified for Epsilon Flock.

Flock provides an Eclipse extension point for distributing migration strategies to metamodel users. The current version of Flock (0.9) does not provide built-in support for chaining multiple migration strategies together, but this could be achieved by using Java to iterate over each migration strategy file, invoking Flock for each strategy.
migrate Graph {
  migrated.nodes = original.gcs.equivalent();
}
migrate Node {
  migrated.linksTo = original.successors().equivalent();
}
operation Original!Node successors() : Collection(Node) {
  return self.outgoing().collect(e|e.trg);
}
operation Original!Node outgoing() : Collection(Edge) {
  return Edge.all.select(e|e.src == self);
}

Listing 1.11. Migrating to the even more evolved graph metamodel with Flock.

A.3 Task 5.2: Removing a node and its incident edges with EOL

The delete keyword removes from the model a model element and all model elements contained in the deleted model element. To delete a Node and its incident Edges, three delete keywords have been used (Listing 1.12) because, in the graph metamodel provided by the case, a Node does not contain its Edges.

var n1 : Node = Node.all.selectOne(n|n.name == "n1");
delete n1.incoming();
delete n1.outgoing();
delete n1;
operation Node incoming() : Collection(Edge) {
  return Edge.all.select(e|e.trg == self);
}
operation Node outgoing() : Collection(Edge) {
  return Edge.all.select(e|e.src == self);
}

Listing 1.12. Deleting a node and its incident edges with EOL.

Alternatively, a Flock migration strategy can be used to specify the nodes and edges that should not be copied to the migrated model (Listing 1.13). Flock provides the delete construct for specifying model elements that should not be copied. Deletions are guarded using the when keyword.

delete Node when: original.name == "n1"
delete Edge when: original.src.name == "n1" or original.trg.name == "n1"

Listing 1.13. Deleting a node and its incident edges with EOL.

A.4 Task 6: Inserting transitive edges with EOL

Inserting transitive edges with EOL is a two-step process (Listing 1.14). First, the graph is inspected to calculate the set of new edges to be created (using the calculateTransitiveEdges operation on lines 7-21). The src and
trg Nodes are stored in the set for each transitive edge to be created. Secondly, an instance of Edge is created for each member of the set (using the calculateTransitiveEdges operation on lines 23-31). The two steps are separate to ensure that transitive edges are created from only those edges that existed at the start of the transformation, and not for edges added partway through the transformation.

Notice that the successors and outgoing operations on (lines 33-36 and 38-41 respectively) might be called more than once for each Node in the model. Consequently the @cached annotation is used to indicate that EOL should cache the results of executing these operations. Caching provides a mechanism for increasing execution time when the result of an operation remains constant throughout the execution of an Epsilon program.

```java
1 var g = Graph.all.first;
2 var edgeSpecs = g.calculateTransitiveEdges();
3 g.addEdgesFromSpecification(edgeSpecs);
4
5 operation Graph calculateTransitiveEdges() : Set {
6     var edgeSpecs = new Set;
7     for (node in Node.all) {
8         for (successor in node.successors()) {
9             for (grandsuccessor in successor.successors()) {
10                 if (not Edge.all.exists(e|e.src == node and e.trg == grandsuccessor)) {
11                     edgeSpecs.add(Sequence { node, grandsuccessor });
12                 }
13             }
14         }
15     }
16     return edgeSpecs;
17 }
18
19 operation Graph addEdgesFromSpecification(edgeSpecs : Set) {
20     for (edgeSpec in edgeSpecs) {
21         var edge : Edge = new Edge;
22         edge.src = edgeSpec.first;
23         edge.trg = edgeSpec.second;
24         self.edges.add(edge);
25         (edge.src.name + "->" + edge.trg.name).println();
26     }
27 }
28
29 @cached operation Node successors() : Collection(Node) {
30     return self.outgoing().collect(e|e.trg);
31 }
32
33 @cached operation Node outgoing() : Collection(Edge) {
34     return Edge.all.select(e|e.src == self);
35 }
36
Listing 1.14. Inserting transitive edges with EOL.
```