Saying Hello World with Epsilon – A Solution to the 2011 Instructive Case

Louis M. Rose[1]   Antonio García-Domínguez[2]   James R. Williams[1]

Dimitrios S. Kolovos[1]   Richard F. Paige[1]

Fiona A.C. Polack[1]

[1] Department of Computer Science, University of York, UK.

`[louis,jw,dkolovos,paige,fiona]@cs.york.ac.uk`

[2] University of Cádiz, Department of Computer Languages and Systems
C/Chile 1, 11002, Cádiz, Spain

`antonio.garciadominguez@uca.es`

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 [1]. To provide an introduction to Epsilon for new users, each part of the solution is discussed and several of the Epsilon languages are demonstrated. Epsilon is briefly described below, 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[1] and in the Epsilon book[2]. The Epsilon Object Language (EOL) [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.

The solutions described in this paper use EOL, as well as the Epsilon Generation Language (EGL) [6] for model-to-text transformation, the Epsilon Validation Language (EVL) [3] for model validation, and Epsilon Flock [5] for a specialised form of in-place model-to-model transformation (model migration). Further languages in the Epsilon platform which were not used for the Hello World case include the Epsilon Comparison Language (ECL) for comparing and matching models, the Epsilon Merging Language (EML) for combining models, and the Epsilon Wizard Language (EWL) for refactoring models.

---

[1] `http://www.eclipse.org/gmt/epsilon`
[2] `http://www.eclipse.org/gmt/epsilon/doc/book`

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 instantiates 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 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";
```

<div align="center">Listing 1: Instantiating the simple Hello World metamodel with EOL.</div>

```
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";
```

<div align="center">Listing 2: Instantiating the extended Hello World metamodel EOL.</div>

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 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 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%]!
```

<div align="center">Listing 3: A model-to-text transformation with EGL.</div>

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 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 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 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
5  operation Node isIsolated() : Boolean {
6    return not Edge.all.exists(e|e.src == self or e.trg == self);
7  }
```

Listing 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 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 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 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.

Alternative, an Epsilon Flock [5] migration strategy can be used to reverse edges. Compared to the EOL solution (Listing 6), the iteration is performed declaratively (`migrate Edge` on line 1 of Listing 7) rather than imperatively (with a `for` loop), and no temporary variable is required because Flock

```
1  for (edge in Edge.all) {
2    var temp = edge.src;
3    edge.src = edge.trg;
4    edge.trg = temp;
5  }
```

<div align="center">Listing 6: Reversing edges with EOL.</div>

provides the `original` and migrated `model` elements, which are bound to distinct model elements. Epsilon Flock is discussed further in Section 5.

```
1  migrate Edge {
2    migrated.src = original.trg.equivalent();
3    migrated.trg = original.src.equivalent();
4  }
```

<div align="center">Listing 7: Reversing edges with Flock.</div>

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 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`.

```
1  migrate Graph {
2    migrated.gcs.addAll(original.nodes.equivalent());
3    migrated.gcs.addAll(original.edges.equivalent());
4  }
5
6  migrate Node {
7    migrated.text = original.name;
8  }
```

<div align="center">Listing 8: Migrating to the evolved graph metamodel with Flock.</div>

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 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.

```
1  delete Node.all.selectOne(n|n.name == "n1");
```

<center>Listing 9: Deleting a node with EOL.</center>

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

7 Opponent Statements

Three opponents were assigned to the Epsilon solution, and their statements[3] are now summarised. Every opponent remarked that most of the solutions to the Hello World are very concise and readable when formulated with Epsilon. However, solutions that required matching complex patterns (such as finding cycles of three nodes in a graph, Section 3) were less concise and readable due to the use of imperative constructs for specifying patterns. We are investigating the possibility of adding pattern matching constructs to EOL, via a more declarative style of syntax.

Two of the statements remarked that using a family of task-specific languages enhanced readability and conciseness of solutions, and the understandability of Epsilon as a whole. One of the statements suggested that learning the similarities and differences between the family of languages might be a challenge for new users of Epsilon. To smooth and reduce the learning curve for users, Epsilon provides online documentation[4] including examples of Epsilon programs, tutorial articles, and even a free book[5]. Finally, as one of the statements suggests, we are currently working on extending the content assistance provided by the Epsilon development tools to support, for example, auto-completion for metamodel types.

References

[1] D.S. Kolovos (2009): *An Extensible Platform for Specification of Integrated Languages for Model Management*. Ph.D. thesis, University of York, United Kingdom.

[2] D.S. Kolovos, R.F. Paige & F.A.C. Polack (2006): *The Epsilon Object Language (EOL)*. In: *Proc. ECMDA-FA*, *LNCS* 4066, Springer, pp. 128–142, doi:10.1007/11787044_11.

[3] D.S. Kolovos, R.F. Paige & F.A.C. Polack (2009): *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. In: *Rigorous Methods for Software Construction and Analysis*, *LNCS* 5115, Springer, pp. 204–218, doi:10.1007/978-3-642-11447-2_13.

[4] S. Mazanek (2011): *Hello World! An Instructive Case for TTC*. In P. Van Gorp, S. Mazanek & L. Rose, editors: *Proc. TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011*, EPTCS.

[5] L.M. Rose, D.S. Kolovos, R.F. Paige & F.A.C Polack (2010): *Model Migration with Epsilon Flock*. In: *Proc. ICMT*, *LNCS* 6142, Springer, pp. 184–198, doi:10.1007/978-3-642-13688-7_13.

[6] L.M. Rose, R.F. Paige, D.S. Kolovos & F.A.C. Polack (2008): *The Epsilon Generation Language*. In: *Proc. ECMDA-FA*, *LNCS* 5095, Springer, pp. 1–16, doi:10.1007/978-3-540-69100-6_1.

[7] J. Sprinkle (2003): *Metamodel Driven Model Migration*. Ph.D. thesis, Vanderbilt University, TN, USA.

---

[3]`http://planet-research20.org/ttc2011/index.php?option=com_community&view=groups&task=viewdiscussions&groupid=13&Itemid=150&view=groups&task=viewdiscussions&groupid=13&Itemid=150` (registration required)

[4]`http://www.eclipse.org/gmt/epsilon/doc/`

[5]`http://www.eclipse.org/gmt/epsilon/doc/book`

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 10. An EVL `constraint` (line 2) is specified in the `context` (line 1) of a particular metamodel type (`Edge` in Listing 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 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
8        do {
9          delete self;
10       }
11     }
12   }
13 }
14
15 operation Edge isDangling() : Boolean {
16   return self.src.isUndefined() or self.trg.isUndefined();
17 }
```

Listing 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 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.

```
1  migrate Graph {
2    migrated.nodes = original.gcs.equivalent();
3  }
4
5  migrate Node {
6    migrated.linksTo = original.successors().equivalent();
7  }
8
```

```
9   operation Original!Node successors() : Collection(Node) {
10    return self.outgoing().collect(e|e.trg);
11  }
12
13  operation Original!Node outgoing() : Collection(Edge) {
14    return Edge.all.select(e|e.src == self);
15  }
```

Listing 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 12) because, in the graph metamodel provided by the case, a `Node` does not contain its
`Edges`.

```
1   var n1 : Node = Node.all.selectOne(n|n.name == "n1");
2
3   delete n1.incoming();
4   delete n1.outgoing();
5   delete n1;
6
7   operation Node incoming() : Collection(Edge) {
8     return Edge.all.select(e|e.trg == self);
9   }
10
11  operation Node outgoing() : Collection(Edge) {
12    return Edge.all.select(e|e.src == self);
13  }
```

Listing 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 13). Flock provides the `delete` construct for specifying model
elements that should not be copied. Deletions are guarded using the `when` keyword.

```
1   delete Node when: original.name == "n1"
2
3   delete Edge when: original.src.name == "n1" or
4                     original.trg.name == "n1"
```

Listing 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 14). First, the graph is inspected to cal-
culate 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 indicated 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.

```
1  var g       = Graph.all.first;
2  var edgeSpecs = g.calculateTransitiveEdges();
3
4  g.addEdgesFromSpecification(edgeSpecs);
5
6
7  operation Graph calculateTransitiveEdges() : Set {
8    var edgeSpecs = new Set;
9
10   for (node in Node.all) {
11     for (successor in node.successors()) {
12       for (grandsuccessor in successor.successors()) {
13         if (not Edge.all.exists(e|e.src == node and e.trg == grandsuccessor)) {
14           edgeSpecs.add(Sequence { node, grandsuccessor });
15         }
16       }
17     }
18   }
19
20   return edgeSpecs;
21 }
22
23 operation Graph addEdgesFromSpecification(edgeSpecs : Set) {
24   for (edgeSpec in edgeSpecs) {
25     var edge : Edge = new Edge;
26     edge.src = edgeSpec.first;
27     edge.trg = edgeSpec.second;
28     self.edges.add(edge);
29     (edge.src.name + "->" + edge.trg.name).println();
30   }
31 }
32
33 @cached
34 operation Node successors() : Collection(Node) {
35   return self.outgoing().collect(e|e.trg);
36 }
37
38 @cached
39 operation Node outgoing() : Collection(Edge) {
40   return Edge.all.select(e|e.src == self);
41 }
```

Listing 14: Inserting transitive edges with EOL.