The Epsilon Solution to the OCL2PSQL Case

Antonio Garcia-Dominguez[1]

[1]Aston University, Birmingham, UK

Abstract

There have been several attempts to map Object Constraint Language queries to SQL: one of these is the OCL2PSQL mapping proposed by Nguyen and Clavel. In this paper, I describe an implementation of OCL2PSQL using two languages from the Eclipse Epsilon project: 744 lines of ETL code for the model-to-model transformation itself, and 97 lines of EGL code for a model-to-text transformation that produces more readable SQL than the reference version. The solution passes all correctness tests set out in the original framework: the transformation has a median time of 0.93s across all scenarios.

Keywords

OCL, SQL, model transformation, abstract syntax graphs

1. Introduction

The OMG Object Constraint Language is a well known, standardized language for specifying constraints and queries in models: while typically associated with UML, it has been widely used for MOF-based modelling languages, and particularly those based on the Ecore implementation of EMOF. Given the increasingly large size of the models used by Model-Driven Engineering practitioners, one common solution is to persist them in databases. This has motivated attempts to map the OCL queries (written in terms of the abstract syntax of the modeling language) to SQL queries that run directly on the underlying relational database used to store the models.

Some of these attempts have used imperative features such as loops and cursors to deal with iterators, possibly reducing their compatibility across vendors (due to the limited standardization of these features). OCL2PSQL ("OCL to pure SQL") is an approach that provides a mapping of nested iterators while staying entirely within the broadly standardized and declarative parts of SQL [1]. The OCL2PSQL TTC case has selected a core subset of this mapping (with some erratum added since the original release of the mapping), and has invited tool authors to demonstrate the usability, conciseness and ease of understanding of their model transformation languages through alternative implementations of this subset.

This paper presents an outline of a solution based on the Eclipse Epsilon family of model management languages. Since the original release in 2006 [2], Eclipse Epsilon has grown to include languages for model-to-model transformation (the Epsilon Transformation Language), model-to-text transformation (the Epsilon Generation Language), model validation, pattern matching, model migration, unit testing and other tasks. The solution passes all correctness checks, though some minor refinements of the proposed mappings were required.

The rest of the paper is structured as follows: Section 2 explains the overall structure of the solution. Section 3 describes the key features of the implemented model-to-model transformation from OCL to SQL. Section 4 describes the alternative model-to-text transformation that has been developed from the SQL models to textual SQL queries. Finally, Section 5 presents the preliminary performance results obtained by the solution author.

2. Overall structure

The Epsilon solution to the OCL2PSQL case is a Java application, using Apache Maven for its dependency management. Epsilon is usable as a standalone Java library, with stable versions available through Maven Central and snapshot versions available through the OSS Sonatype repository. The solution should work in Epsilon 2.3.0, but uses the latest 2.4.0 interim versions to avoid a warning message when using the Epsilon Generation Language.

The solution (now merged into the TTC'21 OCL2PSQL solutions repository[1]) reuses the basic scaffolding of the reference solution, including the generated code for the OCL and SQL metamodels, and the classes responsible for interpreting the environment variables, communicating with the MySQL database, and performing the correctness tests. The solution adds the following Java classes:

- SampleLauncher, which transforms all OCL queries without using the environment variables of the benchmark framework. This is mostly for internal development.

[1]https://github.com/TransformationToolContest/ttc2021-ocl2psql/tree/master/solutions/Epsilon

TTC'21: Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 25 June 2021, ~~Bergen, Norway~~ (online).

a.garcia-dominguez@aston.ac.uk (A. Garcia-Dominguez)
0000-0002-4744-9150 (A. Garcia-Dominguez)

- OCL2SQL, which encapsulates a model-to-model transformation written in the Epsilon Transformation Language into an easy-to-use Java class. More information on the ETL transformation is available in Section 3.
- SQL2Text, which replaces the model-to-text transformation in the reference solution with one based on the Epsilon Generation Language. More details on the EGL transformation are given in Section 4.

Finally, the solution changed the code of the Solution class in the reference solution to use the OCL2SQL and SQL2Text classes from above.

3. Model-to-model transformation with ETL

The Epsilon project includes several languages for performing model transformations:

- The Epsilon Object Language (EOL) is an OCL-inspired imperative language which is well suited for small in-place (endogenous) model transformations, though it can be used for purely imperative exogenous transformations as well.
- The Epsilon Transformation Language (ETL) builds on top of EOL by adding the concept of a *rule*, which transforms a certain type of source model element (possibly with some *guards* limiting its applicability) to a target model element. The rule scheduling can be kept entirely declarative, or can be controlled to some extent through the use of "greedy" or "lazy" rules. ETL is well-suited for exogenous transformations, where a new model is produced from the source model.
- Flock can be thought of as ETL with an automated "copy-unless-otherwise-stated" default strategy. It is well suited for *model migration* tasks, where a model has to undergo small changes from one version of a metamodel to the next.
- The Epsilon Wizard Language (EWL) is a variant of EOL which allows users to define "wizards" that users can manually trigger on specific model elements, performing small in-place transformations (perhaps with some simple user interaction).

Out of these languages, ETL was chosen since the original formulation of OCL2PSQL as a collection of recursive mappings by source type was a close match to the rules used by ETL. For the most part, each of those mappings was translated into an ETL rule, and calls to $\text{map}_e(e)$ were translated into e.equivalent() calls in ETL. The e.equivalent() built-in operation retrieves the SQL subgraph produced from the OCL e subtree, allowing the different SQL subtrees to be linked together. The OCL2PSQL case did not require any manual rule scheduling or the use of greedy/lazy rules: the automated rule scheduling done by ETL based on source element types and guards was sufficient.

Listing 1: Main ETL script

```
import 's0_literals.etl';
import 's1_equals_and.etl';
import 's2_allInstances.etl';
import 's3_size.etl';
import 's4_collect_variable.etl';
import 's5_attributes.etl';
import 's6_associationEnds.etl';
import 's7_exists.etl';
import 's8_existsWithFree.etl';

post {
  var firstRootSelect = SQL!PlainSelect.all.selectOne(
      ps|ps.eContainer.isUndefined());

  var stmt = new SQL!SelectStatement;
  stmt.selectBody = firstRootSelect;
}
```

ETL allows for breaking up the transformation across several modules. This made it relatively easy to iteratively implement the various stages in OCL2PSQL and test out how it behaved for the various challenges. Listing 1 shows the main script of the ETL transformation: it is composed of a number of **import** statements that bring in the rules needed for each stage, and a **post** rule which places the one PlainSelect without a container into a SelectStatement element as mandated by the SQL metamodel.

Some of the OCL2PSQL mappings had to produce significantly large SQL subtrees: to keep the code concise, a library of EOL operations (`utilities.eol`) was created and reused from the ETL rules. This library largely contained a set of functions for simple creation of SQL model elements, an implementation of the OCL2PSQL functions for listing the *free* variables in an OCL expression e ($\text{FVars}(e)$) and for listing the *source* variables that the value of a subexpression e' of the OCL expression e depends upon ($\text{SVars}_e(e')$), and several other miscellaneous functions. As a simple example, Listing 2 shows the code needed to transform OCL integer literals to the target SQL metamodel.

The ETL rules are for the most part a direct one-to-one translation from the descriptions at the end of the OCL2PSQL case, except for two changes.

The first change was considering one special case listed

Listing 2: Excerpt of ETL for stage 0 (integer literals)

```
import 'utilities.eol';

/*
 * All these boil down to:
 * mape(l) = SELECT l as res, 1 as val
 */

rule IntLiteral
transform e:OCL!IntegerLiteralExp
to ps:SQL!PlainSelect {
  ps.selItems.add(longSelectItem('res', e.integerValue.asLong()));
  ps.selItems.add(longSelectItem('val', 1l));
}
```

in the original OCL2PSQL paper [1] but not in the case paper. The mapping of collect and exists in the original OCL2PSQL paper covered the case when $v \notin FVars(b)$, but this mapping had been omitted from the OCL2PSQL case description. It turned out that this special case was needed for some of the queries, e.g. challenge 0 in stage 4 (Car.allInstances()–>collect(c|5)).

The second change was due to an unexpected interaction between the recursive approach used to define OCL2PSQL, the definition of *SubSelect.selectBody* as a containment reference in the SQL metamodel, and how the ETL e.equivalent() operation works. ETL will only apply a certain rule once to each matching source model element, and from them on e.equivalent() will always return the same counterpart in the target model (e.g. the exact same object). This resulted in some SQL queries "losing" the body of their SubSelect objects to other subtrees of the SQL model, as they also needed the mapping for that part of the OCL expression.

For instance, consider the final challenge:

```
Car.allInstances()–>exists(c|
  c.owners–>exists(p|
    p.name = 'Peter'))
```

In this query, the SQL mapping of c.owners–>exists(...) and the SQL mapping of p.name both require using the mapping of c.owners as a subquery. ETL successfully maps c.owners to SQL, but EMF will not allow a single model element to be contained from more than one place. Since the mapping of c.owners–>exists(...) will complete last, it will effectively "steal" the subquery representing c.owners from the mapping of p.name.

The fix for this issue turned out to be simple, as shown in Listing 3. All uses of the e.equivalent() operation were wrapped into a new EOL operation: the operation tested if this "stealing" was about to take place (i.e. if the PlainSelect was already contained in another SubSelect), and if so it performed a deep cloning of the SQL subtree.

Listing 3: Fix for "subtree stealing" in ETL

```
operation copyIfContained(value) {
  var emfTool = new Native(
    "org.eclipse.epsilon.emc.emf.tools.EmfTool");
  if (value.eContainer.isDefined()) {
    return emfTool.ecoreUtil.copy(value);
  }
  return value;
}
```

A better fix (which unfortunately would have required a rewriting of the input files for this case) would be to change the SQL metamodel so that *SubSelect.selectBody* is no longer a containment reference, and the same PlainSelect can be reused from multiple SubSelect model elements.

Even further, this suggests that the OCL2PSQL mapping really produces a SQL expression graph (where some subexpressions are reused) rather than a SQL abstract syntax tree. Instead of running the same subquery from several places, it may be advisable to redefine OCL2PSQL so it produces a sequence of SQL queries rather than a single large SQL query: it would run these reused subqueries first, and then provide their results to the higher-level queries. Otherwise, there may be a risk that the SQL query could grow exponentially if sufficiently large subqueries have to be duplicated across several locations.

Overall, the transformation required writing 14 rules across 531 lines of ETL code, with a support library of EOL operations that was 213 lines long. These line counts included whitespace and comments: if these are excluded, the line counts are reduced to 305 lines of ETL code and

Listing 4: Excerpt of EGL to generate SQL query text

```
1  [%= SelectStatement.all.first.generate() %][%
2
3  @template
4  operation SelectStatement generate() { %]
5  [%=self.selectBody.generate()%];
6  [% }
7
8  @template
9  operation PlainSelect generate() { %]
10 SELECT
11 [% for (si in self.selItems) { %]
12 [%=si.generate() + (hasMore ? "," : "")%]
13 [% }
14     if (self.fromItem.isDefined()) { %]
15 FROM [%=self.fromItem.generate() %]
16 [% }
17     for (join in self.joins) {%]
18 [%=join.generate()%]
19 [% }
20     if (self.whereExp.isDefined()) { %]
21 WHERE [%=self.whereExp.generate() %]
22 [% }
23    if (self.groupBy.isDefined()) { %]
24 [%= self.groupBy.generate() %]
25 [%
26    }
```

154 lines of EOL code[2].

4. Model-to-text transformation to SQL with EGL

The reference solution included a model-to-text transformation that produced the SQL query to be run in MySQL from the SQL model. During the development of this solution, it was found that the generated SQL was difficult to read in the presence of multiple levels of subqueries, as it was entirely on one line.

In order to improve the readability of the SQL queries and help with the debugging, an alternative implementation was written in 96 lines of EGL. The EGL template traverses the SQL model recursively from the root SelectStatement, breaking up SELECT statements, CASE expressions, joins, and subqueries across multiple lines.

Listing 4 shows an excerpt of the EGL script: the first line is the entry point of the entire script, kicking off the recursive descent of the SQL model from the SelectStatement. The EGL script makes heavy use of *template* operations, which allow EGL templates in their body and return strings which can be used within expressions (e.g. for concatenating separators, as in line 12).

The script also uses some of the built-in Epsilon variables: *hasMore* is a built-in Epsilon variable available in loops which is true if and only if there are more values after the current one.

One useful feature in EGL is its ability to integrate *formatters* that postprocess the generated text. In particular, the EGL LanguageFormatter was customised for SQL (as shown in Listing 5) to automatically indent the lines of the SQL script to improve readability, while keeping the EGL script as simple as possible. This class only requires the regular expressions that should increase and decrease the indentation level after a match. Using this script, queries are generated in the more readable form shown in Listing 6.

[2]These counts were obtained using the `count-etl-lines.sh` script included in the solution folder.

Listing 5: SQL-specific LanguageFormatter used to indent the SQL query text

```
private static class SQLFormatter                                  1
  extends LanguageFormatter                                        2
{                                                                  3
  private static final String increasePattern = "\\(\\s*$";        4
  private static final String decreasePattern = "^\\)";            5
                                                                   6
  public SQLFormatter() {                                          7
    super(Pattern.compile(increasePattern,                         8
                          Pattern.MULTILINE),                      9
          Pattern.compile(decreasePattern,                         10
                          Pattern.MULTILINE));                     11
  }                                                                12
}                                                                  13
```

Listing 6: SQL query for challenge 0 in stage 1, as generated by EGL

```
SELECT                                                             1
TEMP_left.res = TEMP_right.res AS res,                             2
1 AS val                                                           3
FROM (                                                             4
        SELECT                                                     5
        2 AS res,                                                  6
        1 AS val                                                   7
) AS TEMP_left                                                     8
JOIN (                                                             9
        SELECT                                                     10
        3 AS res,                                                  11
        1 AS val                                                   12
) AS TEMP_right;                                                   13
```

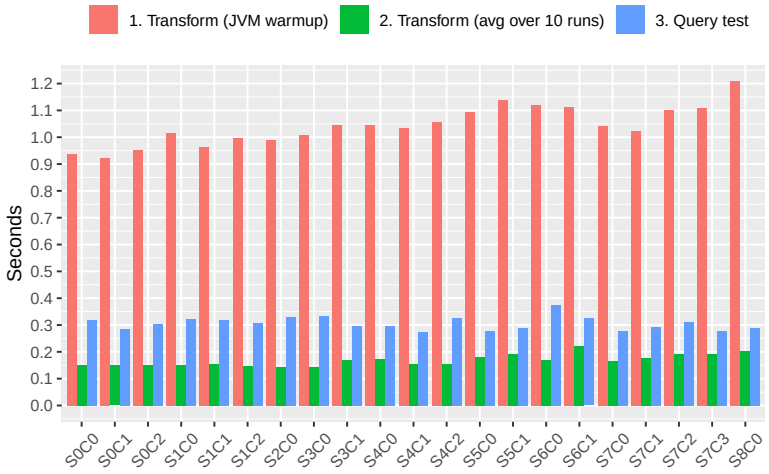

Figure 1: Execution times in seconds per stage and challenge, for the first warmup execution of the transformation, the average of 10 additional runs of the transformation, and the test of the generated SQL statement.

5. Results

After implementing the ETL model-to-model transformation and the EGL model-to-text transformation, Java code to encapsulate these transformations and integrate them with the TTC benchmark framework was added. The transformations passed all correctness cases for all scenarios across all stages and challenges.

In terms of execution time, the transformations were run on a Lenovo X1 laptop with an i7-6600U CPU running at 2.60GHz with 16GiB of physical RAM, running Ubuntu Linux 20.04.2 LTS with Linux 5.4.0-74-generic and the Oracle JDK 11.0.8. The default Java memory allocation settings were used (no `-Xmx` or other JVM options were given). The Docker image provided by the OCL2PSQL case authors was used to run MySQL, using Docker Engine 20.10.7.

The transformation and test times are shown in Figure 1: the transformation times include both the ETL model-to-model transformation and the EGL model-to-text transformation, in order to mimic the two transformations done by the reference implementation (the proper OCL2PSQL transformation, and a model-to-model transformation between a 3rd-party JSqlParser metamodel and the EMF-based metamodel). It was noted during the open peer review stage of the contest that the first execution of the transformation was considerably slower than later executions, due to Java class-loading and just-in-time recompiler warmup times. In order to obtain more representative results from a typical user (who would have a long-running Java process running the transformation repeatedly for different OCL queries), the transformation was then run 10 more times within the same JVM, and the average execution time was recorded. This was followed by a single execution of the generated SQL query, to test if the expected results were produced.

Whereas the execution times for the "warmup" runs are between 0.9s and 1.2s, the average execution times are much smaller and comparable to other solutions of the contest, ranging between 0.1s and 0.25s. Test execution times ranged between 0.3s and 0.4s.

References

[1] H. Nguyen Phuoc Bao, M. Clavel, OCL2PSQL: An OCL-to-SQL code-generator for model-driven engineering, in: T. K. Dang, J. Küng, M. Takizawa, S. H. Bui (Eds.), Future Data and Security Engineering, Springer International Publishing, Cham, 2019, pp. 185–203. doi:10.1007/978-3-030-35653-8_13.

[2] D. S. Kolovos, R. F. Paige, F. Polack, The Epsilon Object Language (EOL), in: Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings, 2006, pp. 128–142. doi:10.1007/11787044_11.