An NMF Solution to the TTC 2021 OCL to SQL Case

Georg Hinkel

Am Rathaus 4b, 65207 Wiesbaden, Germany
georg.hinkel@gmail.com

Abstract

This paper presents a solution to the OCL to SQL translation case at the Transformation Tool Contest (TTC) 2021 using dynamic C# code, but without any dedicated model transformation language, using NMF for the model representation. The transformation tools of NMF are not used because the case does not fall under NMFs definition of a model transformation problem.

1 Introduction

The Object Constraint Language (OCL) is an important language to denote expressions based on models. If the models are stored in a database, it is desirable to translate these queries to SQL statements such that they can be processed directly by the database.

Using models for OCL and SQL, the OCL to SQL case at the Transformation Tool Contest (TTC) asks tool authors to transform models of OCL queries into models of SQL statements. For this purpose, metamodels are provided to understand OCL and SQL as models.

The .NET Modeling Framework [1] is a framework for model-driven engineering on the .NET platform and paper presents a solution of the OCL to SQL case using NMF.

In the remainder of the paper, I first briefly introduce the Dynamic Language Runtime that is heavily used for the solution in this paper in Section 2. Section 3 presents the solution and Section 4 shows the results achieved with the solution.

2 Dynamic C#

The solution makes use of the dynamic language runtime (DLR) that is part of the .NET Framework but perhaps not so widely known. The idea of the DLR is to allow elements of dynamic programming languages in the scope of the .NET runtime. These features are also available in C#, in particular the ability for late binding. That is, by converting variables to `dynamics`, the compiler sees that method calls are only resolved at runtime, based on the usual C# overload selection principles which the compiler attaches to make them available at runtime. However, especially when passing dynamic objects only as parameters, the compiler is able to calculate the set of methods that are candidates for a certain call already, which makes the actual call very efficient. Further, editors such as Visual Studio even show errors, if no suitable candidates could be found, the reference count counts all candidates and the "‘Go To Definition"’ feature lists all candidates.

3 Solution

To discuss the solution, I first give an overview in Section 3.1 before Sections 3.2, 3.3 and 3.4 go into details for the actual translation process, pruning and printing the SQL statement models to strings.

Copyright held by the author(s).

In: A. Garcia-Dominguez, G. Hinkel, and A. Boronat (eds.): Proceedings of the 14th Transformation Tool Contest, on-line, 25-06-2021

3.1 Overview

NMF does have a model transformation language (NTL, [2], [3]) but I decided not to use it for this case. Why? According to the philosophy of NTL, the biggest challenge of a model transformation is to establish an isomorphism between source and target models that provides a tracing functionality and that is used to ensure that certain input model elements are only transformed once and not once for every reference. This is because maintaining such a trace is difficult in general-purpose programming languages because it requires a lot of bookkeeping – one essentially requires a dedicated hashtable for each type and as soon as inheritance is in place, things start to become messy.

However, both the OCL and the SQL metamodels are essentially expression models that have a tree structure with very few cross-references, even none in the case of SQL. Because NMF takes containments very serious and model elements must always have exactly one parent, trying to add an existing model element to a containment reference of another model element removes it from its old container. Therefore, not only that a trace is not needed, it is even counter-productive.

Since the availability of a trace is not an argument in favor of NTL, the question is whether NTL still adds value against a pure general-purpose code solution and I believe the answer is plainly no. Especially using features like the DLR, the late binding can be implemented directly in C# with concepts known by a lot more developers and therefore easier to understand and better supported by tools.

Therefore, I decided to create a solution to the case using plain C# code making use of DLR features.

3.2 Translator

The general idea of the solution is to translate the OCL expressions in a (mutable) context to SQL expressions. This context includes a notion of open variables and their types as well as the body of the enclosing SQL statement and a counter of temporary tables created for a statement in order that they do not get confused. While simple expressions can be mapped to simple SQL expressions, other OCL expressions require to modify the context in which they are called.

Listing 1 shows how this applies to boolean expressions where the literal is simply converted to an EqualsToExpression, either that $1 = 1$ for `true` or $1 = 0$ for `false`.

```
private IExpression GetExpression(SelectContext context, BooleanLiteralExp booleanLiteral) {
  return new EqualsToExpression {
    LeftExp = new LongValue { Value = 1 },
    RightExp = new LongValue {
      Value = booleanLiteral.BooleanValue.GetValueOrDefault() ? 1 : 0
    }
  };
}
```

Listing 1: Translating simple boolean expressions

Calls to `GetExpression` can be nested as denoted in Listing 2 that depicts how to translate And call expressions.

```
return new AndExpression {
  LeftExp = GetExpression(context, (dynamic)callExpression.Source),
  RightExp = GetExpression(context, (dynamic)callExpression.Argument[0])
};
```

Listing 2: Nesting translation calls to translate an And call expression

More interesting is the handling of the AllInstances method as depicted in Listing 3. Because it does not directly have an impact on the result, we return a null reference, but this time change the context and set it to the table with the name of the referred type.

```
private IExpression GetAllInstances(SelectContext context, IEntity referredType) {
  var table = new Table { Name = referredType.Name };
  if (context.Body.FromItem == null) {
    context.Body.FromItem = table;
  } else {
    context.Body.Joins.Add(new Join { RightItem = table });
  }
  return null;
}
```

Listing 3: Handling the AllInstances method

To handle iterators, we need to determine how to bind the variable. For this, the considered subset of the OCL language knows to collections that can be iterated: A collection returned by the AllInstances method or an association of a different variable. In both cases, we add an open variable to the select context while calculating the expression for the iterator body and remove it afterwards.

With the iterators in place, we can implement the PropertyCallExp expressions as depicted in Listing 4.

```
private IExpression GetExpression(SelectContext context, PropertyCallExp propertyCall) {
  switch (propertyCall.Source) {
    case VariableExp variableRef:
      var table = context.Variables[variableRef.ReferredVariable.Name];
      return new Column {
        Table = new Table {
          Name = table,
               Alias = new Alias {
            Name = variableRef.ReferredVariable.Name
          }
        },
        Name = propertyCall.ReferredProperty.Name
      };
    default:
      throw new NotSupportedException();
  }
}
```

Listing 4: Transformation of a PropertyCallExp

The (syntactically allowed) case that a property of a property is queried would require adding more joins, which is ignored in the current solution, particularly given that this was not required for the reference inputs.

In case of an AssociationCallExp, we register the join as last join in the context and add the join to the current select context as depicted in Listing 5.

```
private IExpression GetExpression(SelectContext context, AssociationClassCallExp association) {
  switch (association.Source) {
    case VariableExp variableRef:
      var variable = variableRef.ReferredVariable.Name;
      var associationEnd = association.ReferredAssociationEnds;
      var alias = variable + "_" + associationEnd.Association;
      context.Body.Joins.Add(new Join {
        Left = false,
        RightItem = new Table {
          Name = associationEnd.Association,
          Alias = new Alias { Name = alias }
        },
        OnExp = new EqualsToExpression {
          LeftExp = new Column {
            Table = new Table {
              Name = context.Variables[variable],
              Alias = new Alias { Name = variable }
            },
            Name = associationEnd.Name,
          },
          RightExp = new Column {
            Table = new Table {
              Name = associationEnd.Association,
              Alias = new Alias { Name = alias }
            },
            Name = context.Variables[variable] + "_id",
          }
        }
      });
      context.LastJoin = Tuple.Create(variable, associationEnd);
      return null;
    default:
      throw new NotSupportedException();
  }
}
```

Listing 5: Transformation of an AssociationCallExp

Perhaps the most interesting expression is the method to return the sizes. This is because the the aggregate drastically changes the execution of the query and we need to return rows for actually empty combinations. To do this, we create a temporary sub-select model with the current context query inside, group that query by all context variables and return a column of the temporary table. However, because this eliminates the open variables that might be needed elsewhere, we group the result by all open variables and add these variables to the result. To make them available in the sub-select, which is the new context select statement, we add joins for each open variable from their original table.

To see this, consider an extension of stage 8 where we reuse the open variable `c` as depicted in Listing 6. We refer to this query later on as stage 9.

```
Car.allInstances()->exists(c|c.owners->exists(p|p.name = 'Peter') and c.color='black')
```

Listing 6: Slight extension of the stage 8 query that reuses the open variable `c`

Note, the `exists` method is treated as a filter condition and an additional size aggregate. We need to keep the variable `c` in order to be able to check whether the color is black.

3.3 Pruning

The resulting SQL statement may join tables that are not actually needed, e.g. when joined tables are not actually needed. This gets apparent in challenge 8, where the open variable `c` is only used to calculate the size, but given that we are not interested in any of its properties, we do not actually need to join the Car table once again after the initial context is gone.

```
if (selectBody.SelItems.Select(s => s.Exp).OfType<CountAllFunction>().Any()) {
  return;
}
var expressionsToCheck = selectBody.SelItems.Select(s => s.Exp).ToList();
if (selectBody.WhereExp != null) {
  expressionsToCheck.Add(selectBody.WhereExp);
}
var usedAliases = (from selectExp in expressionsToCheck
                   from column in selectExp.Descendants().OfType<Column>()
                   select column.Table.Alias.Name).Distinct();
for (int i = selectBody.Joins.Count - 1; i >= 0; i--) {
  var join = selectBody.Joins[i];
  if (join.RightItem is Table table && !usedAliases.Contains(table.Alias.Name)) {
    selectBody.Joins.RemoveAt(i);
  }
}
if (selectBody.FromItem is SubSelect subSelect) {
  Prune(subSelect.SelectBody);
}
```

Listing 7: Pruning the joins of the resulting SQL statement

The implementation of the pruning is depicted in Listing 7. Aggregate (sub-)queries are not pruned because removing joins changes the number of result elements and thus the result get incorrect. Otherwise, we select all table aliases that appear either in the selection or in the where clause and remove all joins that join tables that are not actually needed. Lastly, we recurse in case the source is a sub-query.

3.4 Printer

The solution to print the SQL statement models to strings works similar by using the DLR to dispatch the different object types and then print them to strings.

```
public static string Print(IPlainSelect selectBody) {
  var resultBuilder = new StringBuilder();
  resultBuilder.Append($"SELECT␣{string.Join(", ",␣selectBody.SelItems.Select(Print))}");
  if (selectBody.FromItem != null) {
    resultBuilder.Append($"␣FROM␣{PrintFrom((dynamic)selectBody.FromItem)}");
  }
  foreach (var join in selectBody.Joins) {
    resultBuilder.Append($"␣{(join.Left.GetValueOrDefault()␣?␣"LEFT"␣:␣"INNER")}␣JOIN␣{PrintFrom((dynamic)join.RightItem)}␣ON␣{
        PrintExpression((dynamic)join.OnExp)}");
  }
  if (selectBody.WhereExp != null) {
    resultBuilder.Append($"␣WHERE␣{PrintExpression((dynamic)selectBody.WhereExp)}");
  }
  if (selectBody.GroupBy != null) {
    resultBuilder.Append($"␣GROUP␣BY␣{string.Join(", ",␣selectBody.GroupBy.GroupByExps.Select(exp␣=>␣PrintExpression((dynamic)exp))
        )}");
  }
  return resultBuilder.ToString();
}
```

Listing 8: Printing the resulting SQL statement using the DLR

As an example, the method to print the actual SQL statement is depicted in Listing 8. The query printer makes intensive use of the string interpolation available in C#.

4 Evaluation

The solution has been integrated into the benchmark framework. The resulting queries are depicted in Listing 9.

```
***** Stage#0 ***
+++ challenge#0: SQL: SELECT 2 res
+++ challenge#1: SQL: SELECT 'Peter' res
+++ challenge#2: SQL: SELECT 1 = 1 res
***** Stage#1 ***
+++ challenge#0: SQL: SELECT 2 = 3 res
+++ challenge#1: SQL: SELECT 'Peter' = 'Peter' res
+++ challenge#2: SQL: SELECT 1 = 1 and 1 = 1 res
***** Stage#2 ***
+++ challenge#0: SQL: SELECT Car_id res FROM Car
***** Stage#3 ***
+++ challenge#0: SQL: SELECT tmp1.res res FROM (SELECT COUNT(*) res FROM Car) AS tmp1
+++ challenge#1: SQL: SELECT tmp1.res = 1 res FROM (SELECT COUNT(*) res FROM Car) AS tmp1
***** Stage#4 ***
+++ challenge#0: SQL: SELECT 5 res FROM Car AS c
+++ challenge#1: SQL: SELECT c.Car_id res FROM Car AS c
+++ challenge#2: SQL: SELECT 1 = 0 res FROM Car AS c
***** Stage#5 ***
+++ challenge#0: SQL: SELECT c.color res FROM Car AS c
+++ challenge#1: SQL: SELECT c.color = 'black' res FROM Car AS c
***** Stage#6 ***
+++ challenge#0: SQL: SELECT tmp1.res res FROM (SELECT c.Car_id, COUNT(c_Ownership.ownedCars) res FROM Car AS c LEFT JOIN Ownership
      AS c_Ownership ON c.Car_id = c_Ownership.ownedCars GROUP BY c.Car_id) AS tmp1
+++ challenge#1: SQL: SELECT tmp1.res = 0 res FROM (SELECT c.Car_id, COUNT(c_Ownership.ownedCars) res FROM Car AS c LEFT JOIN
     Ownership AS c_Ownership ON c.Car_id = c_Ownership.ownedCars GROUP BY c.Car_id) AS tmp1
***** Stage#7 ***
+++ challenge#0: SQL: SELECT tmp1.res > 0 res FROM (SELECT COUNT(*) res FROM Car AS c WHERE 1 = 1) AS tmp1
+++ challenge#1: SELECT tmp1.res > 0 res FROM (SELECT COUNT(*) res FROM Car AS c WHERE 1 = 0) AS tmp1
+++ challenge#2: SQL: SELECT tmp1.res > 0 res FROM (SELECT COUNT(*) res FROM Car AS c WHERE c.color = 'black') AS tmp1
+++ challenge#3: SQL: SELECT tmp2.res > 0 res FROM (SELECT COUNT(*) res FROM (SELECT c.Car_id, COUNT(c_Ownership.ownedCars) res
     FROM Car AS c LEFT JOIN Ownership AS c_Ownership ON c.Car_id = c_Ownership.ownedCars GROUP BY c.Car_id) AS tmp1 WHERE tmp1.
     res = 1) AS tmp2
***** Stage#8 ***
+++ challenge#0: SQL: SELECT tmp2.res > 0 res FROM (SELECT COUNT(*) res FROM (SELECT c.Car_id, COUNT(p.Person_id) res FROM Car AS c
      LEFT JOIN Ownership AS c_Ownership ON c.Car_id = c_Ownership.ownedCars LEFT JOIN Person AS p ON c_Ownership.ownedCars = p.
     Person_id WHERE p.name = 'Peter' GROUP BY c.Car_id) AS tmp1 WHERE tmp1.res > 0) AS tmp2
```

Listing 9: Resulting SQL Statements

Notably, to reduce the influence of just-in-time compilation, we actually run the solution 100 times and divide the result by 100[1]. The resulting transformation times then are in the range of up to 1.4ms for the stage 8 query and in the sub-millisecond area for most of the other queries. The time for the test lies around 20ms but that certainly gets more interesting once the solution is tested with larger databases.

References

[1] G. Hinkel, "NMF: A multi-platform Modeling Framework," in *Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-29, 2018. Proceedings*, accepted, to appear, Springer International Publishing, 2018.

[2] G. Hinkel, "An approach to maintainable model transformations using an internal DSL," Master's thesis, Karlsruhe Institute of Technology, 2013.

[3] G. Hinkel, T. Goldschmidt, E. Burger, and R. Reussner, "Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations," *Software & Systems Modeling*, pp. 1–27, 2017.

[1]Actually, we do not because the smallest time unit in .NET happens to be 100ns, so we merge the division by 100 with the multiplication by 100.