An NMF solution to the TTC 2021 OCL to SQL case

Georg Hinkel[1]

[1]Am Rathaus 4b, 65207 Wiesbaden, Germany

Abstract

Recent advancements in modern general-purpose programming languages challenge the often stated assumption that dedicated model transformation languages are required to express model transformations, especially when not only few of the typical properties of model transformations are required. In particular, the OCL to SQL case at the Transformation Tool Contest (TTC) 2021 asks for solutions to a transformation from OCL, a typical tree model. This paper presents a solution to this case using dynamic C# code, but without any dedicated model transformation language, only 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 and we discuss the reasons for that.

Keywords

Model Queries, OCL, SQL

1. Introduction

In a frequently cited paper [1], Sendall and Kozaczynski state that dedicated model transformation languages should be used for most model transformation activities because general-purpose languages are less suited for this kind of tasks. Particularly in recent years, this assumption has become very popular [2]. However, modern general-purpose programming languages have significantly evolved since the paper from Sendall and Kozaczynski was published and it is fair to doubt whether that statement is still correct, in particular since surprisingly few empirical evidence is given to support the assumption [2].

One of the arguments in favor of model transformation languages is the simple and usually type-safe access to a trace model. However, the trace is only important when model elements are referenced more than once and there is an important category of models where model elements are typically only referenced only once, namely trees, for instance expression trees.

A particularly important expression tree model is the Object Constraint Language (OCL). OCL is an important language to denote expressions based on models in order to formulate constraints, but it is also used to specify queries. 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 [3] is a framework for model-driven engineering on the .NET platform and paper presents a solution of the OCL to SQL case using NMF. NMF even has multiple model transformation languages, but these do not fit to the problem at hand. Instead, this paper presents a solution using plain C# with massive usage of the Dynamic Language Runtime.

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. Finally, Section 4 discusses 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 `dynamic`s, 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, integrated editors such as Visual Studio even show errors, if no suitable candidates could be found, the reference count counts all possible methods that the call could be resolved to and the "'Go To Definition"' feature lists all of them.

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).

georg.hinkel@gmail.com (G. Hinkel)

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.

3.1. Overview

NMF does have a model transformation language (NTL, [4, 5])[1] 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 seriously 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`.

```
1  private IExpression GetExpression(SelectContext context,
       BooleanLiteralExp booleanLiteral) {
2   return new EqualsToExpression {
3    LeftExp = new LongValue { Value = 1 },
4    RightExp = new LongValue {
5     Value = booleanLiteral.BooleanValue.GetValueOrDefault()
            ? 1 : 0
6    }
7   };
8  }
```

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.

```
1  return new AndExpression {
2   LeftExp = GetExpression(context, (dynamic)callExpression.
        Source),
3   RightExp = GetExpression(context, (dynamic)callExpression.
        Argument[0])
4  };
```

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.

```
1  private IExpression GetAllInstances(SelectContext context,
       IEntity referredType) {
2   var table = new Table { Name = referredType.Name };
3   if (context.Body.FromItem == null) {
4    context.Body.FromItem = table;
5   } else {
6    context.Body.Joins.Add(new Join { RightItem = table });
7   }
8   return null;
9  }
```

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.

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.

[1] In fact, NMF even has two model transformation languages where NTL is the rather imperative approach. NMF Synchronizations [6] is more declarative and targets incremental and/or bidirectional model transformations.

```
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

```
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

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.

Perhaps the most interesting expression is the method to return the sizes. This is because 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.

```
1  public static string Print(IPlainSelect selectBody) {
2   var resultBuilder = new StringBuilder();
3   resultBuilder.Append($"SELECT {string.Join(", ",
          selectBody.SelItems.Select(Print))}");
4   if (selectBody.FromItem != null) {
5    resultBuilder.Append($" FROM {PrintFrom((dynamic)
           selectBody.FromItem)}");
6   }
7   foreach (var join in selectBody.Joins) {
8    resultBuilder.Append($" {(join.Left.GetValueOrDefault() ?
           "LEFT" : "INNER")} JOIN {PrintFrom((dynamic)join.
          RightItem)} ON {PrintExpression((dynamic)join.
          OnExp)}");
9   }
10  if (selectBody.WhereExp != null) {
11   resultBuilder.Append($" WHERE {PrintExpression((dynamic)
          selectBody.WhereExp)}");
12  }
13  if (selectBody.GroupBy != null) {
14   resultBuilder.Append($" GROUP BY {string.Join(", ",
          selectBody.GroupBy.GroupByExps.Select(exp =>
          PrintExpression((dynamic)exp)))}");
15  }
16  return resultBuilder.ToString();
17 }
```

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 and discussion

The solution has been integrated into the benchmark framework. In order to get an insight on the generated SQL queries, the resulting queries are depicted in Listing 9.

Notably, to reduce the influence of just-in-time compilation, I actually run the solution 100 times on a Intel Core i7-8550U CPU clocked at 1.99 Ghz in a system with 8GB RAM running Windows 10 and divide the result by 100[2]. 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 and thus is negligible. The time for the test lies around 20ms but that certainly gets more interesting once the solution is tested with larger databases. First, the transformation scheme used inside this paper differs from the original OCL2PSQL transformation scheme [7] and a correctness proof for the transformation scheme presented used in this paper is correct is out of scope for this paper. The reason that I did not use the OCL2PSQL mapping is that I am generally not satisfied with the verbosity of the SQL statements generated by it, whereas the SQL statements generated by the transformation scheme presented here are much easier to comprehend in my opinion. The validation of the mapping scheme presented here will be subject of future work. However, this different mapping scheme also makes a comparison with alternative implementations that stick more closely to the OCL2PSQL mapping more difficult.

In my opinion, the solution shows well how to use the Dynamic Language Runtime available in C# to perform dispatch on parameters, a frequent selling argument of model transformation languages apart from access to trace, incrementality and bidirectionality. The latter two properties require a very declarative way of specifying model transformations such as exemplified e.g. by NMF Synchronizations [6], but the transformation at hand is written in a very imperative style. Anyways, incremental change propagation is not relevant for the case at hand, since changes the main purpose of the transformation is to execute the resulting SQL statement and analyze the result data. Bidirectionality would be very interesting to reverse-engineer SQL statements in order to make them more understandable, but it is unclear to what extend this is possible at all. Given that the trace is not important in this case, there is just no reason not to use the Dynamic Language Runtime, especially taking into account the very good performance results.

A further advantage of a solution in plain C# is that it can be easily integrated into model transformations written in internal DSLs using C# as a host language, in particular NTL, since usually, not the entire model forms a tree structure and hence, access to the trace is required.

This integration, however, would be much more difficult when incrementality was important despite incrementalization systems like NMF Expressions [8] that operate on C# code (or models thereof). It will be subject of future work how transformations like the mapping from OCL to SQL can be supported when incremental change propagation is required.

[2] Actually, I 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.

References

[1] S. Sendall, W. Kozaczynski, Model transformation the heart and soul of model-driven software development, Technical Report, 2003.

[2] S. Götz, M. Tichy, R. Groner, Claimed advantages and disadvantages of (dedicated) model transformation

```
***** 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

languages: a systematic literature review, Software and Systems Modeling 20 (2021) 469–503.

[3] G. Hinkel, NMF: A multi-platform Modeling Framework, in: A. Rensink, J. S. Cuadrado (Eds.), Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-29, 2018. Proceedings, Springer International Publishing, Cham, 2018, pp. 184–194.

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

[5] G. Hinkel, T. Goldschmidt, E. Burger, R. Reussner, Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations, Software & Systems Modeling (2017) 1–27. URL: http://rdcu.be/oTED. doi:10.1007/s10270-017-0578-9.

[6] G. Hinkel, E. Burger, Change Propagation and Bidirectionality in Internal Transformation DSLs, Software & Systems Modeling (2017). URL: http://rdcu.be/u9PT. doi:10.1007/s10270-017-0617-6.

[7] H. N. P. Bao, M. Clavel, Ocl2psql: An ocl-to-sql code-generator for model-driven engineering, in: International Conference on Future Data and Security Engineering, Springer, 2019, pp. 185–203.

[8] G. Hinkel, R. Heinrich, R. Reussner, An extensible approach to implicit incremental model analyses, Software & Systems Modeling 18 (2019) 3151–3187.