Mapping OCL into SQL: Challenges and Opportunities Ahead *

Manuel Clavel[1](✉) and Hoàng Nguyễn Phước Bảo[2]

[1] Vietnamese-German University, Bình Dương, Vietnam
manuel.clavel@vgu.edu.vn
[2] Vietnamese-German University, Bình Dương, Vietnam
ngpbhoang1406@gmail.com

Abstract. In this paper, we discuss some of the challenges and opportunities that arise when mapping OCL into SQL. For the challenges, we review the past proposals, evaluating their key design decisions and limitations. For the opportunities, we highlight the key role that OCL-to-SQL code-generators can play in model-driven software development. By no means we pretend to exhaust the subject: other challenges and opportunities can be identified and brought up. Our present goal is to stir up again the discussion on this subject among the OCL community.

Keywords: OCL · SQL · Code generation

1 Introduction

Model-driven engineering (MDE) aims to develop software systems using *models* as the driving force. Models are artifacts that specify the different aspects of the intended software system. Appropriate *code-generators* should then bridge the gap between models and executable code.

The Unified Modeling Language [12] is the facto standard modeling language for MDE. Originally, UML was conceived as a graphical language: models were defined using diagrammatic notation. However, it soon became clear that UML diagrams were not expressive enough to specify certain aspects of software systems. To address this limitation, the Object Constraint Language (OCL) [11] was added to the UML standard. OCL is a textual language, with a semi-formal semantics. It can be used to specify in a precise, unambiguous way complex *constraints* and *queries* over models. For example, to define *integrity constraints* and *authorization constraints* in the context of secure database-centric applications model-driven development [4].

Several mappings from OCL to SQL have been proposed in the past [5, 7, 8, 10, 13]. The limitations of these mappings, when used as OCL-to-SQL code-generators, reveal some of the non-trivial challenges ahead. We can organize these challenges into two groups. The first group contains the challenges related

to *language coverage*, i.e., how much of the OCL language a mapping can cover. The second group contains the challenges related to *execution time efficiency*, i.e., how long it takes to execute a query generated by a mapping.

Interestingly, the limitations of the past mappings from OCL to SQL also show that the opportunities ahead are not insignificant. In a nutshell, correctly implementing in SQL complex queries is not an easy task, and, arguably, a more difficult task than specifying them in OCL. Furthermore, implementing complex queries in SQL, which execute efficiently on a large database, is a task often reserved for SQL connoisseurs. Hence, the opportunities ahead for an OCL-to-SQL code generator are plentiful.

Organization The rest of the paper is organized as follows. In Section 2 we introduce the main example that we use throughout the paper. Next, in Section 3 we review the current mappings from OCL to SQL, evaluating their key design decisions and limitations. Then, in Section 4 we discuss some of the challenges that arise when mapping OCL into SQL, especially from the point of view of for the *execution time efficiency* of the queries generated by the mapping. Finally, in Section 5 we conclude with some remarks and propose future work.

2 Our Example

To evaluate the limitations of the current mappings from OCL to SQL, and to illustrate the challenges and opportunities ahead, we use throughout this paper the following example. Consider the diagram `CarOwnership` shown in Figure 1. It models a simple domain, where there are only *cars* and *persons*. The persons can own cars (they are their *owners*), and, logically, the cars can be owned by persons (they are their *ownedCars*). No restriction is imposed regarding *ownership*: a person can own many different cars (or none), and a car can be owned by many different persons (or by none). Finally, each car can have a *color*, and each person can have a *name*.

The `CarOwnership` model can be implemented in SQL as a database `CarDB` containing:

- A table `Car` to store the cars, with a column `color` to store the color of each car, and a column `Car_id` to store the (primary) *key* of each car.
- A table `Person` to store the persons, with a column `name` to store the name of each person, and a column `Person_id` to store the (primary) *key* of each person.
- A table `Ownership` to store the many-to-many relationship of ownership between cars and persons, with columns `ownedCars` and `owners` storing the (foreign) *keys* of the corresponding cars and persons.

In the sections that follow, we will use this implementation, shown in Figure 2. Notice that we have not added other indexes to the tables `Car`, `Person`, and `Ownership` excepts those created by declaring the columns `Car_id` and `Person_id` as primary keys, and the columns `ownedCars` and `owners` as foreign

keys. When executing queries on the database `CarDB` we will consider different *scenarios*. In particular, `CarDB`(n) will denote an instance of the database `CarDB` containing 10^n cars and $10^{(n-1)}$ persons, where each car is owned by one person and each person owns 10 different cars, and each car has a color different from `'no-color'`, and each person has a name different from `'no-name'`. Finally, when executing queries on the database `CarDB`, we use a server machine, with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz with 16 GB RAM, using MySQL 5.7.25.[3] The execution times reported in the following sections correspond to the arithmetic mean of 50 executions. The figures reported are given in seconds, unless otherwise stated.

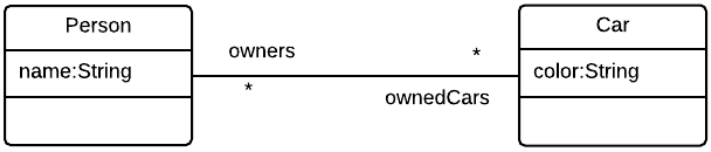

Fig. 1. The `CarOwnership` model

3 Mappings from OCL to SQL

In this section we review the current mappings from OCL to SQL, evaluating their key design decisions and their limitations.[4] Unfortunately, in the majority of cases, we have not been able to access the tools implementing the mappings, and we must base our analysis in the published reports.

3.1 OCL2SQL

To the best of our knowledge, OCL2SQL [5, 6, 10] was the first attempt of mapping OCL into SQL. Based on a set of transformation *templates*, OCL2SQL

[3] Although we have used MySQL for running our examples, we believe that our overall results should apply likewise to the other SQL engines.

[4] Logically, the first key design decision is how to map the OCL contextual models to SQL schemata (the so-called OR mapping). To the best of our knowledge, the mappings reviewed here use essentially the same underlying OR-mapping (classes are mapped to tables, attributes to columns, and many-to-many associations to tables with appropriate foreign-keys). An interesting discussion, already introduced in [6], is how to make the OCL-to-SQL mappings *independent* of the underlying OR-mappings. This discussion is, however, outside the scope of this paper.

```
CREATE TABLE Car (
  Car_id int(11) NOT NULL AUTO_INCREMENT,
  color varchar(255) DEFAULT NULL,
  PRIMARY KEY (Car_id)
) ENGINE=InnoDB

CREATE TABLE Person (
  Person_id int(11) NOT NULL AUTO_INCREMENT,
  name varchar(255) DEFAULT NULL,
  PRIMARY KEY (Person_id)
) ENGINE=InnoDB

CREATE TABLE Ownership (
  ownedCars int(11) DEFAULT NULL,
  owners int(11) DEFAULT NULL,
  KEY fk_ownership_ownedCars (ownedCars),
  KEY fk_ownership_owners (owners),
  CONSTRAINT ownership_ibfk_1
     FOREIGN KEY (ownedCars) REFERENCES Car (Car_id),
  CONSTRAINT ownership_ibfk_2
     FOREIGN KEY (owners) REFERENCES Person (Person_id)
) ENGINE=InnoDB
```

Fig. 2. `CarDB`: An SQL implementation of the `CarOwnership` model

automatically generates SQL queries from OCL expressions. An interesting application of OCL2SQL is described in [14]. OCL2SQL only covers (a subset of) OCL boolean expressions. Moreover, the high execution time for the queries generated by OCL2SQL makes it impractical, as an OCL-to-SQL code-generator, for large scenarios. For example, [3] reported that the query generated by OCL2SQL for the expression:

Writer.allInstances−>forAll(a|a.books−>forAll(b|b.page >300))

takes more than 45 minutes to execute on a scenario consisting of 10^2 writers and 10^5 books, each writer being the author of 10^3 books and each book having exactly 150 pages.[5]

3.2 MySQL4OCL

MySQL4OCL [8] is defined recursively over the structure of OCL expressions. For each OCL expression, MySQL4OCL generates a *stored procedure*[6] that, when

[5] The experiment was carried out on a machine with Intel Pentium M 2.00GHz 600MHz, and 1GB of RAM.

[6] Stored procedures are routines (like a subprogram in a regular computing language) that are stored in the database. Stored procedures provide a special syntax for local variables, error handling, loop control, if-conditions, and cursors, which allow the definition of iterative structures.

called, creates a *temporary table* containing the values corresponding to the evaluation of the given expression. More concretely, for the case of iterator expressions, the stored procedure generated by MySQL4OCL repeats, using a *loop*, the following process: i) it *fetches* from the iterator's source collection a new element, using a *cursor*; ii) it calls the stored procedure corresponding to the iterator's body with the newly fetched element as a *parameter*; iii) it processes the resulting temporary table according to the semantics of the iterator's operator.

Although cursors and loops (inside stored procedures) allow MySQL4OCL to cover a large subclass of the OCL language (including nested iterators), they also bring about a fundamental limitation to the use of MySQL4OCL as an OCL-to-SQL code-generator: they often impede the highly-optimized execution strategies implemented by SQL engines. Still, the interested reader can find in [8] a preliminary discussion about the efficiency of the code produced by MySQL4OCL, as well as a comparison with previous known results on evaluating OCL expressions on medium-large scenarios. Unfortunately, the writer-book example mentioned before for the case of OCL2SQL is not included in this comparison.

3.3 Incremental OCL constraints checking

An interesting method for evaluating OCL constraints consists of checking if the SQL query characterizing the tuples that violate the given constraint returns the empty set. This method was first introduced in [6] and then exploited in [13] when incrementally checking OCL constraints. As in the case of OCL2SQL, this method is limited to OCL boolean expressions. With regards to execution time efficiency, the figures provided in [13] are not easily comparable with *normal* execution times, since the generated SQL queries are computed in an *incremental* way. More specifically, "whenever a change in the data occurs, only the constraints that may be violated because of such change are checked and only the relevant values given by the change are taken into account."

3.4 SQL-PL4OCL

SQL-PL4OCL [7] closely follows the design of MySQL4OCL and, consequently, bears the same fundamental limitation regarding execution time efficiency, as we illustrate with an example below. Still, concerning its predecessor, SQL-PL4OCL simplifies the definition of the mapping, improves the execution time of the generated queries (by reducing the number of temporary tables), and implements some of the features that were left in [8] as future work: namely, handling the *null* value and supporting (not parametrized) sequences.

Example 1. To illustrate the *costly* consequences, in terms of execution time efficiency, of using cursors and loops to implement OCL iterator expressions, consider the following OCL query:

```
Car.allInstances()->select(c|c.owners->exists(p|p.name ='no-name'))->size().
```

The stored procedure generated by SQL-PL4OCL for this query is given in [7] (Example 11). Now, if we call this stored procedure on the scenarios `CarDB`(3), `CarDB`(4), `CarDB`(5), `CarDB`(6), and `CarDB`(7), we obtain the following execution times:[7]

	`CarDB`(3)	`CarDB`(4)	`CarDB`(5)	`CarDB`(6)	`CarDB`(7)
SQL-PL4OCL	0.76	6.17	1min 3.02	10min 24.00	> 90min

We will go back to this example at the end of Section 4. Here we simply note that the above query, when implemented in SQL in the *expected* way (without cursors and loops), takes less than 1 second to execute on the scenario `CarDB`(7), while the stored procedure generated by SQL-PL4OCL (using cursors and loops) did not finish its execution after 90 minutes.

3.5 OCL2PSQL

As part of our on-going research, we are implementing a new mapping from OCL to SQL called OCL2PSQL. The interested reader can experiment with our current prototype at:

`http://researcher-paper.ap-southeast-1.elasticbeanstalk.com/`

As in the case MySQL4OCL/SQL-PL4OCL, our mapping is defined recursively over the structure of OCL expressions. However, OCL2PSQL diverts completely from MySQL4OCL/SQL-PL4OCL in that it does not rely on the use of cursors and loops for implementing iterator expressions, neither does it creates temporary tables for storing intermediate results. Instead, i) for intermediate results, it uses standard *subqueries* and ii) for iterator expressions, it adds to the subquery corresponding to the iterators' body an extra column corresponding to the iterator's variable. Intuitively, this column stores the element in the iterator's source that is "responsible" for the result that is stored in the corresponding row.

Although OCL2PSQL seems to avoid the limitation of MySQL4OCL/SQL-PL4OCL in terms of execution time efficiency, its implementation is still undergoing and it cannot yet successfully address some of the other challenges discussed below.

4 Challenges

After reviewing the key design decisions and limitations of the existing mappings from OCL to SQL, we highlight now some of the challenges ahead. We organize these challenges into two groups. The first group contains the challenges related to *language coverage*, i.e., how much part of the OCL language a mapping covers. The second group contains the challenges related to *execution time efficiency*, i.e., how much time it takes to execute a query generated by a mapping.

[7] The definition of the scenarios `CarDB`(n) is given in Section 2.

4.1 Language coverage

OCL [11] is a language for specifying constraints and queries using a textual notation. Every OCL expression is written in the context of a model, called the *contextual* model. OCL is strongly typed. Expressions either have a *primitive* type, a *class* type, a *tuple* type, a *collection* type, or a *special* type. OCL provides a *dot*-operator to access the values of attributes and association-ends. OCL also provides standard operators on primitive data, tuples, and collections, and special operators to *iterate* over collections, such as forAll, exists, select, and collect. Collections can be *sets*, *bags*, *ordered sets* and *sequences*, and can be parametrized by any type, including other collection types. To represent *undefinedness*, OCL provides two constants, namely, null and invalid, of a special type. Intuitively, null represents an unknown or undefined value, whereas invalid represents an error or exception.

The challenges regarding language coverage are many, including mapping complex iterator expressions such as the transitive closure expressions. Here we only highlight two challenges, which, in our opinion, are among the most "pressing" ones, in the sense that they deal with features of the OCL language that are commonly used.

Since SQL does not *natively* support (parametrized) structured collections, mappings from OCL to SQL would need to explicitly *encode* this "structure" in the generated queries, as proposed in [8]. This is the approach followed in [7] to support (not parametrized) *sequences*. Currently, none of the existing mappings can support parametrized collections.

Although SQL supports the *null*-value, it is not obvious how (or if) it can be used to implement the null constant in OCL, and it is even less obvious how the constant invalid can be implemented in SQL. Currently, among the published mappings, only [7] covers OCL expressions dealing with OCL *null*-undefinedness. None of the existing mappings can support OCL *invalid*.

4.2 Execution time efficiency

SQL engines have highly optimized strategies for executing queries over large databases. We discuss below some of the optimization "tips" that OCL-to-SQL code-generators should be aware of, to generate queries that can execute efficiently on large databases.[8]

To illustrate our discussion, we include below examples of SQL queries executed in `CarDB`(6) and `CarDB`(7). As explained before, MySQL4OCL/SQL-PL4OCL cannot efficiently handle (nested) iterators over large collections. Thus, in our comparisons, we can only include the execution times corresponding to queries generated by OCL2PSQL.

[8] Nevertheless, we should be aware that "development is ongoing, so no optimization tip is reliable for the long term." (MySQL 8.0 Reference Manual, (13.2.11.11 Optimizing Subqueries).

Indexes Indexes are used by SQL engines to improving the speed of operations in a table. Indexes can be created using one or more columns. The following example illustrates well the difference between using indexes or not, in terms of execution time efficiency.

Example Suppose that we want to know the number of cars whose color is 'no-color'. In SQL we can use the following query:

```
SELECT COUNT(*)
   FROM (SELECT * FROM Car WHERE color = 'no-color') AS TEMP;
```

In OCL we can specify the same query using the following expression:

```
Car.allInstances()->select(c|c.color ='no-color')->size()
```

Currently, OCL2PSQL translates this expression as follows:

```
SELECT COUNT(TEMP_select_body.ref_c) AS res, TRUE AS val
FROM
  (SELECT TEMP_LEFT.res = TEMP_RIGHT.res AS res,
    TEMP_LEFT.ref_c AS ref_c, TEMP_LEFT.val_c AS val_c
   FROM
     (SELECT color AS res, Car_id AS ref_c,
      TRUE AS val_c FROM Car) AS TEMP_LEFT
   JOIN
     (SELECT 'no-color' AS res, TRUE AS val) AS TEMP_RIGHT
  ) AS TEMP_select_body
WHERE TEMP_select_body.res = TRUE;
```

If we execute the above queries on `CarDB`(6) and `CarDB`(7), with and without indexing the column `color` (columns $\texttt{CarDB}(n)^{[\texttt{color}]}$ and `CarDB`(n), respectively) in the table `Car`, we obtain the following results.[9]

	`CarDB`(6)	$\texttt{CarDB}(6)^{[\texttt{color}]}$	`CarDB`(7)	$\texttt{CarDB}(7)^{[\texttt{color}]}$
SQL	0.22	0.00	2.48	0.00
OCL2PSQL	0.24	0.24	3.00	2.60

As expected, the execution performance for the SQL-query dramatically improves when indexing the column `color`. However, the query generated by OCL-2PSQL does not *prompt* the SQL engine to use the index `color`, and therefore its performance does not improve. Advanced/smart OCL-to-SQL code-generators should automatically i) add indexes (for one or more columns) to the tables, and ii) generate queries that *benefit* from the added indexes. The challenge here is to add to the tables *only* the indexes that will increase the execution performance for the given SQL-queries, since the insert and update-statements will naturally take more time on tables having indexes.

[9] As reported in [7] (Query Q8, Figure 6: "Evaluation times"), the query generated by SQL-PL4OCL for a similar OCL expression takes 50.02 seconds to execute on a scenario like `CarDB`(6), on an Intel Core m7, 1.3 GHz, 8 GB RAM, using MySQL 5.7.

`EXISTS` function When using the `EXISTS` function, SQL engines are optimized to stop processing when a row is returned. The following example illustrates well this optimization.

Example Suppose that we want to know if there is at least one car whose color is different from 'no-color'. In SQL we can use the following query, which uses the `COUNT` function:

```
SELECT COUNT(*) > 0
FROM (SELECT * FROM Car WHERE color <> 'no-color') AS TEMP;
```

Alternatively, we can use the following query, which uses the `EXISTS` function:

```
SELECT EXISTS (SELECT * FROM Car WHERE color <> 'no-color');
```

On the other hand, we can specify in OCL the original query using the following expression:

```
Car.allInstances()->exists(c|c.color <>'no-color')
```

Currently, OCL2PSQL translates this expression as follows:

```
SELECT COUNT(*) > 0 AS res, TRUE AS val
FROM
  (SELECT TEMP_LEFT.res <> TEMP_RIGHT.res AS res,
   TEMP_LEFT.ref_c AS ref_c, TEMP_LEFT.val_c AS val_c
   FROM
     (SELECT color AS res, Car_id AS ref_c,
      TRUE as val_c FROM Car) AS TEMP_LEFT
     JOIN
     (SELECT 'no-color' AS res, TRUE as val) AS TEMP_RIGHT
  ) AS TEMP_exists_body
WHERE TEMP_exists_body.res = TRUE;
```

If we execute the above queries on `CarDB`(6) and `CarDB`(7), without indexing the column `color` in the table `Car`, we obtain the following results.[10]

[10] Interestingly, [7] (Query Q10, Figure 6: "Evaluation times") reports that the query generated by SQL-PL4OCL for the same OCL expression only takes 0.05 seconds to execute on a scenario like `CarDB`(6), on an Intel Core m7, 1.3 GHz, 8 GB RAM, using MySQL 5.7. However, this low execution time can be misleading. In a sense, it corresponds to the "best-case scenario". MySQL4OCL/SQL-PL4OCL treats the forAll and exists iterators differently from other iterators. First of all, (the stored procedure generated by) MySQL4OCL/SQL-PL4OCL does not create a temporary table of the size of the source collection to store the *intermediate* results, but a temporary table with a single row. Secondly, (the stored procedure generated by) MySQL4OCL/SQL-PL4OCL stops the execution of the internal *loop* as soon as it *finds* an element in the source collection that makes the execution of the iterator's body returns `FALSE` or `TRUE`, depending on whether the iterator is a forAll or an exists. Since all the cars in `CarDB`(6) have a color different from 'no-color', (the stored procedure generated by) MySQL4OCL/SQL-PL4OCL stops the internal loop after just one iteration.

	CarDB(6)	CarDB(7)
SQL(using COUNT)	0.22	2.72
SQL(using EXISTS)	0.00	0.00
OCL2PSQL	0.24	3.10

As expected, the execution performance for the SQL-query dramatically improves when using the EXISTS function instead of the COUNT function. OCL-2PSQL currently uses the COUNT function to implement the exists iterator. Thus, its performance does not benefit from the *short-circuiting* provided by the EXISTS function. Advanced/smart OCL-to-SQL code generators should automatically translate OCL expressions using operators capable of *short-circuiting* the execution, whenever possible. The challenges here are (i) to find both the short-circuiting functions in SQL and the operators in OCL that are semantically "compatible" with each other, and (ii) to properly handle the case when the former effectively short-circuits the execution of the query.

Joins versus correlated subqueries Mappings from OCL to SQL may try to use *correlated subqueries* to implement OCL iterators. A correlated subquery is a subquery that contains a reference to a table that also appears in the outer query. However, only for certain cases, SQL engines are optimized for executing correlated subqueries. The following example illustrates well this problem.

Example Suppose that we want to know the number of cars that have at least one owner whose name is 'no-name'. In SQL we can use the following query, which uses a correlated subquery:

```
SELECT COUNT(*) FROM Car AS TEMP
WHERE EXISTS
  (SELECT 1 FROM Ownership
   JOIN Person
   ON Person.Person_id = owners
   WHERE Person.name = 'no-name'
      AND TEMP.Car_id = ownedCars);
```

Alternatively, we can also use the following query, which uses joins instead of correlated subqueries:

```
SELECT COUNT(*)
FROM (SELECT COUNT(*) > 0 FROM Car
      JOIN Ownership
      on Car_id = ownedCars
      JOIN Person
      ON Person.Person_id = owners
      WHERE Person.name = 'no-name'
      GROUP BY Car_id) AS TEMP;
```

On the other hand, we can specify in OCL the original query using the following expression:

```
Car.allInstances()->select(c|c.owners->exists(p|p.name ='no-name'))->size()
```

Currently, OCL2PSQL translates this expression as follows:

```
SELECT COUNT(TEMP_select_body.ref_c) AS res, TRUE as val
FROM
   (SELECT COUNT(*) > 0 AS res,
    TEMP_exists_body_bool_exp.ref_c AS ref_c
    FROM
       (SELECT TEMP_LEFT.res = TEMP_RIGHT.res AS res,
        TEMP_LEFT.ref_c AS ref_c, TEMP_LEFT.val_c AS val_c,
        TEMP_LEFT.ref_p AS ref_p, TEMP_LEFT.val_p AS val_p
        FROM
           (SELECT name AS res,
            TEMP_p.ref_c AS ref_c, TEMP_p.val_c AS val_c,
            TEMP_p.ref_p AS ref_p, Person_id IS NOT NULL as val_p
            FROM
               (SELECT owners AS res,
                TEMP_c.ref_c AS ref_c, TEMP_c.val_c AS val_c,
                ownedCars AS ref_p, ownedCars IS NOT NULL AS val_p
                FROM
                 (SELECT Car_id AS res, Car_id AS ref_c,
                  TRUE AS val_c FROM Car) AS TEMP_c
                LEFT JOIN Ownership
                ON ownedCars = TEMP_c.ref_c
                AND TEMP_c.val_c = TRUE) AS TEMP_p
             LEFT JOIN Person
             ON Person.Person_id = TEMP_p.ref_p
             AND TEMP_p.val_p = TRUE) AS TEMP_LEFT
          JOIN (SELECT 'no-name' AS res,
          TRUE AS val) AS TEMP_RIGHT)
     AS TEMP_exists_body_bool_exp
     WHERE TEMP_exists_body_bool_exp.res = TRUE
     AND TEMP_exists_body_bool_exp.val_p = TRUE
     GROUP BY ref_c) AS TEMP_select_body
WHERE TEMP_select_body.res = TRUE;
```

If we execute the above queries on `CarDB`(6) and `CarDB`(7), without indexing the column `name` in the table `Person`, we obtain the following results.[11]

[11] As reported in Section 3, for the scenario `CarDB`(6), it takes over 10 minutes to execute the stored procedure generated by SQL-PL4OCL for this query; for the scenario `CarDB`(7), the execution did not finish after 90 minutes.

	CarDB(6)	CarDB(7)
SQL (using correlation)	14.38	2min 59.52
SQL (using joins)	0.04	0.28
OCL2PSQL	0.04	0.30

As expected, the execution performance for the SQL-query improves dramatically when using *joins* instead of *correlated subqueries*. OCL2PSQL uses *joins* to implement iterators. The performance of the query generated by OCL2PSQL is on par with the performance of the SQL-query implemented using *joins*. The challenge here is to properly handle the case when the elements to be joined do not *match* with each other.

5 Concluding Remarks and Future Work

The Object Constraint Language (OCL) plays a key role in adding precision to UML models, and therefore it is called to be an important actor in model-driven engineering (MDE). However, to fulfill this role, smart/advanced *code-generators* must bridge the gap between UML/OCL models and executable code. This is certainly the case for database-centric applications.

In this paper we have reviewed the existing mappings from OCL to SQL, evaluating their key design decisions and their limitations.[12] We recognize that, up to now, the main efforts have been placed in *covering* as much as possible of the OCL language. Without underestimating that challenge, we want to emphasize the need to look at the other key challenge, namely, to generate, from OCL expressions, SQL queries that can perform *on par with* SQL queries implemented by professionals. In this regard, we have provided examples that show that none of the existing OCL-to-SQL mapping is really up to this task, although OCL2PSQL shows significant progress in that direction. We have also identified some of the optimization "tips" supported by SQL engines that smart/advanced OCL-to-SQL code-generators should be aware of, in order to generate queries that can execute efficiently on large databases. A related challenge, which we have not discussed here, is the readability of the SQL queries generated by the OCL-to-SQL mappings. Ideally, the generated queries should be easy to understand and to modify, if needed. Arguably, this has not been the case up to now, and therefore we consider it as an additional challenge for smart/advanced OCL-to-SQL code-generators.

We also recognize that implementing in SQL *complex* queries is not an easy task; if fact, we can argue that it is a more difficult task than specifying them in OCL. Suppose, for example, that we are interested in querying our database

[12] There have been also different proposals [1–3,9] in the past for what we may call OCL *evaluators*. These are tools that *load* first the scenario on which an OCL expression is to be evaluated and then *evaluate* this expression using an OCL interpreter. As reported in [3], the main problem with OCL evaluators is the time required for *loading* large scenarios.

`CarDB` (without assuming that every car has at least one owner) about: i) if it *exists a car whose owners all have the name 'no-name'*, and ii) *how many cars have at least one owner with no name declared yet*. We can specify i) in OCL as follows:

```
Car.allInstances()->exists(c|c.owners->forAll(p|p.name='no-name'))
```

Similarly, we can specify ii) in OCL as follows:

```
Car.allInstances()->select(c|c.owners->exists(p|p.name.oclIsUndefined()))->size()
```

We invite the reader to implement i) and ii) in SQL, and draw his/her own conclusions.

In our opinion, this state of affairs offers exciting opportunities for smart/advanced OCL-to-SQL code-generators. To prove our point, we want to propose the following case study for the OCL (and database) community:

- Take a group of students who have just taken a Database course, and ask them to write in SQL some queries (given in English). The students will be provided with the underlying data model (a class diagram or an ER diagram).
- Take another group of students who have taken the same (or similar) Database course, plus a short-course on OCL. Ask them to write in OCL the same queries as before. The students will be provided with the same underlying data model as the other group.
- Then, evaluate and compare the results, taking into consideration:
  - The correctness (from the semantic point of view) of the SQL queries (how many students wrote them correctly in SQL) versus the correctness of the OCL queries (how many students wrote them correctly in OCL).
  - The efficiency of the SQL queries (how long it takes to execute them on large scenarios) versus the efficiency of the SQL queries generated by the OCL-to-SQL code-generator of choice (how long it takes to execute them on the same large scenarios).

Finally, we leave as another task for the OCL community to carry out a more systematic, in-depth comparison of the different OCL-to-SQL code-generators, including not only language-coverage and execution time efficiency, but also underlying OR mappings, SQL constructs used (queries, views, stored procedures, SQL dialects), and other implementations issues (RDBMS, software architecture, and so on).

References

1. T. Baar and S. Markovic. The RoclET tool. `http://www.roclet.org/index.php`, 2007.
2. D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Carcu. An OCL environment (OCLE) 2.0.4. `http://lci.cs.ubbcluj.ro/ocle/`, 2005. Laboratorul de Cercetare in Informatica, University of BABES-BOLYAI.

3. M. Clavel, M. Egea, and M. A. G. de Dios. Building an efficient component for OCL evaluation. *ECEASST*, 15, 2008.
4. M. A. G. de Dios, C. Dania, D. A. Basin, and M. Clavel. Model-driven development of a secure ehealth application. In M. Heisel, W. Joosen, J. Lopez, and F. Martinelli, editors, *Engineering Secure Future Internet Services and Systems - Current Research*, volume 8431 of *LNCS*, pages 97–118. Springer, 2014.
5. B. Demuth and H. Hußmann. Using UML/OCL constraints for relational database design. In R. B. France and B. Rumpe, editors, *UML*, volume 1723 of *LNCS*, pages 598–613. Springer, 1999.
6. B. Demuth, H. Hußmann, and S. Loecher. OCL as a specification language for business rules in database applications. In M. Gogolla and C. Kobryn, editors, *UML*, volume 2185 of *LNCS*, pages 104–117. Springer, 2001.
7. M. Egea and C. Dania. SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. *Software & Systems Modeling*, 2017.
8. M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *ECEASST*, 36, 2010.
9. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.
10. F. Heidenreich, C. Wende, and B. Demuth. A framework for generating query language code from OCL invariants. *ECEASST*, 9, 2008.
11. Object Management Group. Object constraint language specification version 2.4. Technical report, OMG, February 2014. `https://www.omg.org/spec/OCL/About-OCL/`.
12. Object Management Group. Unified Modeling Language. Technical report, OMG, December 2017. `https://www.omg.org/spec/UML/About-UML/`.
13. X. Oriol and E. Teniente. Incremental checking of OCL constraints through SQL queries. In A. D. Brucker, C. Dania, G. Georg, and M. Gogolla, editors, *OCL@MoDELS*, volume 1285 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org, 2014.
14. H. Sneed, B. Demuth, and B. Freitag. A process for assessing data quality. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*, 03 2013.