OCL2PSQL: An OCL-to-SQL Code-Generator for Model-Driven Engineering

Hoang Nguyen Phuoc Bao(✉) and Manuel Clavel

Faculty of Engineering, Vietnamese-German University, Thu Dau Mot City, Vietnam
ngpbhoang1406@gmail.com, manuel.clavel@vgu.edu.vn

Abstract. The Object Constraint Language (OCL) is a textual, declarative language typically used as part of the UML standard for specifying constraints and queries on models. Several attempts have been proposed in the past for translating OCL expressions into SQL queries in the context of Model Driven Engineering (MDE). To cope with OCL expressions that include iterators, previous attempts resorted to imperative features (loops, cursors) of SQL, with the consequent loss of efficiency. In this paper, we define (and implement) a novel mapping from OCL to SQL that covers (possibly nested) iterators, without resorting to imperative, non-declarative features of SQL. We show with a preliminary benchmark that our mapping generates SQL queries that can be efficiently executed on mid- and large-size SQL databases.

Keywords: OCL · SQL · Model-driven engineering · Code-generation

1 Introduction

In the context of software development, model-driven engineering (MDE) aspires to develop software systems by using *models* as the driving-force. Models are artifacts defining the different aspects and views of the intended software system. Ideally, the *gap* between the models and the real software systems would be covered by appropriate *code-generators*.

The Unified Modeling Language [12] is the facto standard modeling language for MDE. Originally, it was conceived as a graphical language: models were defined using diagrammatic notation. However, it promptly became clear that UML diagrams were not expressive enough to define certain aspects of the intended software systems, and the Object Constraint Language (OCL) [11] was added to the UML standard. OCL is a textual language, with a 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 application model-driven development [5].

T. K. Dang et al. (Eds.): FDSE 2019, LNCS 11814, pp. 185–203, 2019.
https://doi.org/10.1007/978-3-030-35653-8_13

In the past a number of mappings from OCL to other languages have been proposed, each with its own goals and limitations. In particular, [7,8,10,13] introduce different mappings from OCL to SQL. The limitations of these mappings, when used as OCL-to-SQL code-generators in a software development process driven by UML/OCL models, can be organized into two groups. The first group contains the limitations related to *language coverage*, i.e., how much part of the OCL language a mapping can cover. The second group contains the limitations related to *execution-time efficiency*, i.e., how much time it takes to execute a query generated by a mapping.

In this paper we provide a novel solution to the question of whether OCL queries can be *transformed* into executable software. Our solution covers a significant subset of the OCL language, including (possibly) nested iterators as well as undefined values. It also reduces significantly the limitation of previous mappings regarding execution-time efficiency.

Organization. The rest of the paper is organized as follows. In Sect. 2 we introduce background material about SQL and OCL. Next, in Sect. 3 we define the sub-language of OCL that our OCL-to-SQL mapping currently covers. Then, in Sect. 4 we introduce our mapping, which is defined recursively over the sub-language presented in Sect. 3. Next, in Sect. 5 we discuss some preliminary benchmarks. Finally, in Sect. 6 we review previously proposed OCL-to-SQL, and we conclude, in Sect. 7, with some closing remarks and future work.

2 Background

SQL [14] is a special-purpose programming language designed for managing data in relational database management systems (RDBMS). Originally based upon relational algebra and tuple relational calculus, its scope includes data insert, query, update and delete, schema creation and modification, and data access control. Although SQL is to a great extent a declarative language, it also includes procedural elements. In particular, the procedural extensions to SQL support stored procedures which are routines (like a subprogram in a regular computing language, possibly with loops) that are stored in the database.

Notation. Let tb be a table. Let r and col be, respectively, a row and a column of tb. In what follows, we denote by $col(r)$ the value stored in the column col in the row r. Let qry be an SQL-query. Let db be an SQL-database. We denote by $\text{Exec}(qry, db)$ the result of executing qry on db.

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, or a collection type. OCL provides standard operators on primitive data, tuples, and collections. For example, the operator `includes` checks whether an element is inside a collection. OCL also provides a dot-operator to access the values of class instances' attributes and association-ends in the given data model instance. For example, suppose that the contextual

model includes a class c with an attribute at and an association-end as. Then, if o is an object of class c in the given data model instance, the expression $o.at$ refers to the value of the attribute at for the object o in this data model instance, and $o.as$ refers to the objects linked to the object o through the association-end as. OCL provides operators to iterate over collections, such as `forAll`, `exists`, `select`, `reject`, and `collect`. Collections can be *sets*, *bags*, *ordered sets* and *sequences*, and can be parametrized by any type, including other collection types. Finally, 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. OCL is a pure specification language: when an expression is evaluated, it simply returns a value without changing anything in the model.

Notation. Let e be an OCL expression. In what follows, we denote by $\mathrm{FVars}(e)$ the set of variables that occur *free* in e, i.e., that are not *bound* by any iterator. Let e be an OCL expression, and let v be a variable introduced in e by an iterator expression $s\ {->}iter\ (v \mid b\ )$. In what follows, $\mathrm{src}_e(v)$ denotes the *source* s of v in e. Let e be an OCL expression and let e' be a subexpression of e. Then, we denote by $\mathrm{SVars}_e(e')$ the set of variables which (the value of) e' depends on, and is defined as follows:

$$\mathrm{SVars}_e(e') = \bigcup_{v \in \mathrm{FVars}(e')} \{v\} \cup \mathrm{SVars}_e(\mathrm{src}_e(v)).$$

Let e be an OCL expression, such that $\mathrm{FVars}(e) = \emptyset$. Let $\mathcal{O}$ be an OCL Scenario. In what follows, we denote by $\mathrm{Eval}(e, \mathcal{O})$ the result of *evaluating* e in $\mathcal{O}$, according to the semantics of the language.

In what follows, without loss of generality, we assume that the names of the iterator variables within an OCL expression are unique.

3 The OCL2PSQL Language

The OCL2PSQL-language is currently defined by the following inductive rules:[1]

- true and false are OCL2PSQL-expression of type Bool.
- i is an OCL2PSQL-expression, for i an integer number. The type of i is Integer.
- w is an OCL2PSQL-expression, for w a string. The type of w is String.
- c .allInstances() is an OCL2PSQL-expression, for c a class-type. The type of c .allInstances() is $\mathrm{Set}(c)$.

[1] Notice that we do not include in our language operations on collections of collections, nor we support the invalid value. 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]. Similarly, since SQL does not *natively* support the *invalid* value, mappings from OCL to SQL would have to handle this value in "ad hoc" ways. Currently, none of the other mappings is able to support (parametrized) structured collections or the invalid value.

- v .at is an OCL2PSQL-expression, for v a variable of a class-type t, and at an attribute of the class-type t. The type of v .at is the type of at.
- v .ase is an OCL2PSQL-expression, for v a variable of a class-type t, and ase an association-end of the class-type t. The type of v .ase is Set(c'), where c' is the class at the other end of ase.
- $l = r$ is an OCL2PSQL-expression, for l and r OCL2PSQL-expressions of the same predefined type or class-type. The type of $l = r$ is Bool.
- s ->size() is an OCL2PSQL-expression, for s an OCL2PSQL-expression of type Col(t), where Col is either Set or Bag and t is a predefined type or a class-type. The type of s ->size() is Integer.
- s ->forAll(v | b) is an OCL2PSQL-expression, for s an OCL2PSQL-expression of type Col(t), where Col is either Set or Bag, with t a class type or a predefined type, and where v is a variable of the type t, and b is an OCL2PSQL-expression of type Bool. The type of s ->forAll(v | b) is Bool.
- s ->exists(v | b) is an OCL2PSQL-expression, for s an OCL2PSQL-expression of type Col(t), where Col is either Set or Bag, with t a class type or a predefined type, and where v is a variable of the type t, and b is an OCL2PSQL-expression of type Bool. The type of s ->exists(v | b) is Bool.
- s ->select(v | b) is an OCL2PSQL-expression, for s an OCL2PSQL-expression of type Col(t), where Col is either Set or Bag, with t a class type or a predefined type, and where v is a variable of the type t, and b is an OCL2PSQL-expression of type Bool. The type of s ->select(v | b) is Set(t).
- s ->collect(v | b) is an OCL2PSQL-expression, for s an OCL2PSQL-expression of type Col(t), where Col is either Set or Bag, with t a class type or a predefined type, and where v is a variable of the type t, and b is an OCL2PSQL-expression of type u, where u either a class-type or a predefined type or a type Col(u'), where u' either a class-type or a predefined type. The type of s ->collect(v | b) is Bag(u).
- s ->flatten() is an OCL2PSQL-expression, for s an OCL2PSQL-expression of type Bag(Col(t)), where Col is either Set or Bag, with t a predefined type or a class-type. The type of s ->flatten() is Bag(t).
- l .oclIsUndefined() is an OCL2PSQL-expression, for l an OCL2PSQL-expression of type t, with t a predefined type or a class-type. The type of l .oclIsUndefined() is Bool.

4 The OCL2PSQL Mapping

To illustrate the definition of our mapping, we will use the following example throughout this section. Consider the diagram `CarOwnership` shown in Fig. 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*.

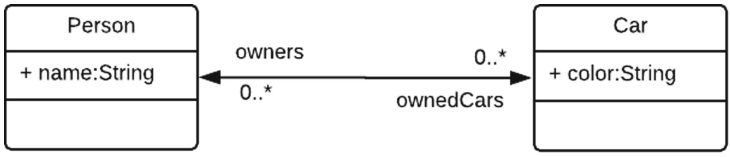

Fig. 1. The `CarOwnership` model

4.1 Data Models

Firstly, we formally define the valid *context* for OCL2PSQL expressions. An OCL2PSQL-*data model* is a tuple $\langle C, AT, AS \rangle$ where:

- C is a set of *classes* c.
- AT is a set of *attribute* declarations. An attribute declaration $\langle at, c, t \rangle$ denotes that at is an attribute, of type t, of the class c.
- AS is a set of *association* declarations. An association declaration $\langle as, ase_l, c_l, ase_r, c_r \rangle$ denotes that as is an association between two classes, c_l and c_r, where ase_l is the association-end whose *source* is c_l and *target* is c_r, and, vice versa, ase_r is the association-end whose *source* is c_r and *target* is c_l.

Next, we define a mapping map() from OCL2PSQL-data models to SQL-schemata.

Let $\mathcal{D} = \langle C, AT, AS \rangle$, be a data model. Then map($\mathcal{D}$) is defined as follows:

- For every $c \in C$,

```
CREATE TABLE c (
   c_id int NOT NULL
    AUTO_INCREMENT PRIMARY KEY);
```

- For every attribute $\langle at, c, t \rangle \in AT$

```
ALTER TABLE c ADD COLUMN at SqlType(t);
```

where:
  - if $t =$ `Integer`, then SqlType(t) = `int`;
  - if $t =$ `String`, then SqlType(t) = `varchar`;
  - if $t \in C$, then SqlType(t) = `int`.

Moreover, if $t \in C$, then

```
ALTER TABLE c ADD FOREIGN KEY fk_c_iat(at)
  REFERENCES t(t_id);
```

- For every association $\langle as, ase_l, c_l, ase_r, c_r \rangle \in AS$,

```
CREATE TABLE as (
  ase_l int,
  ase_r int,
  FOREIGN KEY fk_c_l_ase_l(ase_l)
    REFERENCES c_l(c_l_id),
  FOREIGN KEY fk_c_r_ase_r(ase_r)
    REFERENCES c_r(c_r_id));
```

Example 1. In Fig. 2 we show the SQL-schema generated by our mapping for the OCL2PSQL-data model `CarOwnership`. In what follows, we denote by `CarDB` the database create with the aforementioned schema.

```
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. Example: map(`CarOwnership`)

4.2 Data Model Instances

Firstly, we formally define the instances of OCL2PSQL-data models, i.e., the valid *scenarios* for evaluating OCL2PSQL-expressions. Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. A $\mathcal{D}$-*instance* is a tuple $\langle OC, OAT, OAS \rangle$ where:

- OC is a set of objects declarations. An object declaration (oc, c) denotes that oc is an object of the class c.
- OAT is a set of attribute value declarations. An attribute value declaration $\langle \langle at, c', t \rangle, (oc, c), vl \rangle$ denotes that the value of the attribute at in the object oc is vl.
- OAS is a set of association link declarations. Each association link declaration $\langle \langle as, ase_l, c_l, ase_r, c_r \rangle, (oc_l, c_l), (oc_r, c_r) \rangle$ denotes that the objects oc_l and oc_r are linked through the association as, in such a way that oc_r is among the objects linked to oc_l through the association-end ase_l, and, vice versa, oc_l is among the objects linked to oc_r through the association-end ase_r.

Next, we define a mapping map() from instances of OCL2PSQL-data models to SQL-databases.

Let $\mathcal{D} = \langle C, AT, AS \rangle$ be an OCL2PSQL-data model. Let $\mathcal{OD} = \langle OC, OAT, OAS \rangle$ be a $\mathcal{D}$-*instance*. Then map($\mathcal{O}$) is defined as follows:

– For every $oc \in OC$,

  `INSERT INTO` c `VALUES ();`

  For our reference, let $\mathrm{id}(oc)$ denote the integer-value automatically generated for the column c`_id`.

– For every $\langle\langle at, c', t\rangle, (oc, c), vl\rangle \in OAT$

  `UPDATE` c `SET` at `=` vl `WHERE` c`_id =` $\mathrm{id}(oc)$`;`

– For every $\langle as, ase_l, c_l, ase_r, c_r\rangle \in OAS$,

  `INSERT INTO` as `(`ase_l`,` ase_r`) VALUES (`oc_l`,` oc_r`);`

4.3 Expressions

Finally, in this section we recursively define our mapping map() from OCL2PSQL-expressions to SQL-queries. The correctness of our mapping could be formalized as follows: Let e be an OCL2PSQL-expression, such that $\mathrm{FVars}(e) = \emptyset$, and let $\mathcal{O}$ be an OCL2PSQL-*scenario*. Then,

$$\mathrm{Exec}(\mathrm{map}_e(e), \mathrm{map}(\mathcal{O})) \equiv_{\mathrm{OCL2PSQL}} \mathrm{Eval}(e, \mathcal{O}).$$

The different cases in our recursive definition below follow the same key idea underlying our mapping: namely, let e be an OCL2PSQL-expression, let e' be a subexpression of e, and let $\mathcal{O}$ be an OCL2PSQL-scenario. Then, $\mathrm{Exec}(\mathrm{map}_e(e'), \mathrm{map}(\mathcal{O}))$ returns a table, with a column `res`, a column `val`, and, for each $v \in \mathrm{SVars}_e(e')$, a column `ref_`$v$. Informally, for each row in this table: (i) the columns `ref_`v contain a valid "instantiation" for the iterator variables of which the evaluation of e' depends on (if any); (ii) the column `val` contains 0 when evaluating the expression e', with the "instantiation" represented by the columns `ref_`v, evaluates to the *empty set*; otherwise, the column `val` contains 1; (iii) when the column `val` contains 1, the column `res` contains the result of evaluating the expression e' with the "instantiation" represented by the columns `ref_`v; when the column `val` contains 0, the value contained in the column `res` is not meaningful. More concretely,

Remark 1. Let e be an OCL2PSQL-expression. Let e' be a subexpression of e, such that $\mathrm{FVars}(e') = \emptyset$. Let $\mathcal{O}$ be an OCL2PSQL-scenario. Then, $\mathrm{Exec}(\mathrm{map}_e(e'), \mathrm{map}(\mathcal{O}))$ returns a table, with a column `res` and column `val`, such that, for each row r, $\mathtt{val}(r) = 1$. Intuitively, each row in the table returned by $\mathrm{Exec}(\mathrm{map}_e(e'), \mathcal{O})$ represents an element in $\mathrm{Eval}(e', \mathcal{O})$, and vice versa.

Remark 2. Let e be an OCL2PSQL-expression. Let e' be a subexpression of e, such that $\mathrm{FVars}(e')$ is not empty. Let $\mathcal{O}$ be an OCL2PSQL-scenario. Then, $\mathrm{Exec}(\mathrm{map}_e(e'), \mathcal{O})$ returns a table, with a column `res`, a column `val`, and, for each $v \in \mathrm{SVars}_e(e')$, a column `ref_`$v$, and such that, for each row r, $\mathtt{val}(r)$ is either 1 or 0. Intuitively, for each row r in the table returned by $\mathrm{Exec}(\mathrm{map}_e(e'), \mathcal{O})$

- if $\texttt{val}(r)$ is 1, then $\texttt{res}(r)$ is an element in $\text{Eval}(\theta(e'), \mathcal{O})$, and vice versa, where θ is the substitution $\{v \mapsto \texttt{ref\_}v(r) \mid v \in \text{SVars}(e')\}$.
- if $\texttt{val}(r)$ is 0, then $\text{Eval}(\theta(e'), \mathcal{O}) = \emptyset$, where, as before, θ is the substitution $\{v \mapsto \texttt{ref\_}v(r) \mid v \in \text{SVars}(e')\}$.

Variables

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = v$, where v is a variable. Then,

$\text{map}_e(v) =$
```
SELECT
  TEMP_dmn.res as res,
  TEMP_dmn.res as ref_v,
  TEMP_dmn.val as val,
  TEMP_dmn.ref_v' as ref_v', for each v' ∈ SVars_e(src(v))
FROM (map_e(src(v))) as TEMP_dmn.
```

Attributes

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = v$ $.att$, where v is a variable of class-type c and att is an attribute of the class c. Then,

$\text{map}_e(v\ .att) =$
```
SELECT
  c.att as res,
  TEMP_obj.val as val,
  TEMP_obj.ref_v' as ref_v', for each v' ∈ SVars_e(v)
FROM (map_e(v)) as TEMP_obj
LEFT JOIN c
ON TEMP_obj.ref_v = c.c_id AND TEMP_obj.val = 1.
```

Association-ends

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = v$ $.ase$, where v is a variable of class-type c, and ase is an association-end of the class c. Let $\text{Assoc}(ase)$ be the association to which ase belongs, and let $\text{Oppos}(ase)$ be the association-end at the opposite end of ase in $\text{Assoc}(ase)$. Then,

$\text{map}_e(v\ .ase) =$
```
SELECT
  Assoc(ase).ase as res,
  CASE Assoc(ase).Oppos(ase) IS NULL
    WHEN 1 THEN 0
    ELSE 1 END as val,
```

```
  TEMP_obj.ref_v' as ref_v', for each v' ∈ SVars_e(v)
FROM (map_e(v)) as TEMP_obj
LEFT JOIN Assoc(ase)
ON TEMP_obj.ref_v = Assoc(ase) .Oppos(ase) AND TEMP_obj.val = 1.
```

AllInstances

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = c$ .allInstances(), where c is a class type. Then,

```
map_e(c .allInstances())=
SELECT c_id as res, 1 as val FROM c.
```

Equality

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = (l = r)$. We need to consider the following cases:

- $\text{FVars}(l) = \text{FVars}(r) = \emptyset$. Then,

  ```
  map_e(l =r) =
  SELECT
    TEMP_left.res = TEMP_right.res as res,
    1 as val
  FROM (map_e(l)) AS TEMP_left, (map_e(r)) AS TEMP_right
  ```

- $\text{FVars}(l) \neq \emptyset$, $\text{SVars}(r) \subseteq \text{SVars}(l)$. Then,

  ```
  map_e(l =r) =
  SELECT
    TEMP_left.res = TEMP_right.res as res,
    CASE TEMP_left.val = 0 OR TEMP_right.val = 0
      WHEN 1 THEN 0
      ELSE 1 END as val,
    TEMP_left.ref_v as ref_v, for each v ∈ SVars_e(l)
  FROM (map_e(l)) AS TEMP_left
  [LEFT] JOIN (map_e(r)) AS TEMP_right
  [ON] TEMP_left.ref_v = TEMP_right.ref_v, for each v ∈ SVars_e(l) ∩ SVars_e(r).
  ```

- $\text{FVars}(r) \neq \emptyset$, $\text{SVars}(l) \subseteq \text{SVars}(r)$. As before, but swapping the order of the elements in the left-join.
- $\text{FVars}(l) \neq \emptyset$, $\text{FVars}(r) \neq \emptyset$, $\text{SVars}(l) \not\subseteq \text{SVars}(r)$, and $\text{SVars}(r) \not\subseteq \text{SVars}(l)$. Then,

  ```
  map_e(l =r) =
  SELECT
    TEMP_left.res = TEMP_right.res as res,
    CASE TEMP_left.val = 0 OR TEMP_right.val = 0
      WHEN 1 THEN 0
  ```

```
    ELSE 1 END as val,
  TEMP_left.ref_v, for each v ∈ SVars_e(l),
  TEMP_right.ref_v, for each v ∈ SVars_e(r)
FROM (map_e(l)) AS TEMP_left, (map_e(r)) AS TEMP_right
```

Size

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = s$ −>size(). We need to consider the following cases:

- $\text{FVars}(s) = \emptyset$. Then,

  $\text{map}_e(s$ −>size()) =

```
SELECT
  COUNT(*) as res,
  1 as val
FROM (map_e(s)) AS TEMP_src.
```

- $\text{FVars}(l) \neq \emptyset$, Then,

  $\text{map}_e(s$ −>size()) =

```
SELECT
  CASE TEMP_src.val = 0
    WHEN 1 THEN 0
    ELSE COUNT(*) END as res,
  TEMP_src.ref_v as ref_v, for each v ∈ SVars_e(s)
  1 as val
FROM (map_e(s)) AS TEMP_src
GROUP BY TEMP_src.ref_v, for each v ∈ SVars_e(s), TEMP_src.val.
```

Collect

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = s$ −>collect($v \mid b$). We need to consider the following cases:

- $v \in \text{FVars}(b)$ and $\text{FVars}(e') = \emptyset$.

```
SELECT TEMP_body.res as res,
  TEMP_body.val as val,
FROM (map_e(b)) as TEMP_body
```

- $v \in \text{FVars}(b)$ and $\text{FVars}(e') \neq \emptyset$.

```
SELECT TEMP_body.res as res,
  TEMP_body.val as val,
  TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(s),
  TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(b) \ SVars(s) \ {v}
FROM (map_e(b)) as TEMP_body
```

- $v \notin \text{FVars}(b)$. Similarly, but the source and the body would need to be *joined* using a `JOIN`-clause.

Exists

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = s$ ->exists($v \mid b$). We need to consider the following cases:

- $v \in \text{FVars}(b)$ and $\text{FVars}(e') = \emptyset$. Then

```
SELECT
  COUNT(*) > 0 as res,
  1 as val
FROM (map_e(b)) as TEMP_body
WHERE TEMP_body.res = 1
```

- $v \in \text{FVars}(b)$ and $\text{FVars}(e') \neq \emptyset$. Then

```
SELECT
  CASE TEMP_src.ref_v IS NULL
    WHEN 1 THEN 0
    ELSE TEMP.res END as res,
  1 as val,
  TEMP_src.ref_v' as ref_v', for each v' ∈ SVars(s),
  TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(b) \ SVars(s) \ {v}
FROM (map_e(s)) as TEMP_src
LEFT JOIN (
  SELECT COUNT(*) > 0 as res,
    TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(b) \ {v}
  FROM (map_e(b)) as TEMP_body
  WHERE TEMP_body.res = 1
  GROUP BY TEMP_body.ref_v', for each v' ∈ SVars(b) \ {v}
) as TEMP_body
ON TEMP_src.ref_v' = TEMP_body.ref_v', for each v' ∈ SVars(s)
```

- $v \notin \text{FVars}(b)$. Similarly, but the source and the body would need to be *joined* using a `JOIN`-clause.

ForAll

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = s$ ->forAll($v \mid b$). We need to consider the following cases:

- $v \in \text{FVars}(b)$ and $\text{FVars}(e') = \emptyset$. Then

```
SELECT
  COUNT(*) = 0 as res,
  1 as val
FROM (map_e(b)) as TEMP_body
WHERE TEMP_body.res = 0
```

– $v \in \text{FVars}(b)$ and $\text{FVars}(e') \neq \emptyset$.

```
SELECT
  CASE TEMP_src.ref_v IS NULL
    WHEN 1 THEN 1
    ELSE TEMP.res END as res,
  1 as val,
  TEMP_src.ref_v' as ref_v', for each v' ∈ SVars(s),
  TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(b) \ SVars(s) \ {v}
FROM (map_e(s)) as TEMP_src
LEFT JOIN (
  SELECT COUNT(*) = 0 as res,
    TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(b) \ {v}
  FROM (map_e(b)) as TEMP_body
  WHERE IFNULL(TEMP_body.res, 0) = 0 AND TEMP_body.val = 1
  GROUP BY TEMP_body.ref_v', for each v' ∈ SVars(b) \ {v}
) as TEMP_body
ON TEMP_src.ref_v' = TEMP_body.ref_v', for each v' ∈ SVars(s)
```

– $v \notin \text{FVars}(b)$. Similarly, but the source and the body would need to be *joined* using a `JOIN`-clause.

Select

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = s$ ->select($v \mid b$). We need to consider the following cases:

– $v \in \text{FVars}(b)$ and $\text{FVars}(e') = \emptyset$.

```
SELECT
  TEMP_body.ref_v as res,
  TEMP_body.val as val
FROM (map_e(b)) as TEMP_body
  WHERE TEMP_body.res = 1
```

– $v \in \text{FVars}(b)$ and $\text{FVars}(e') \neq \emptyset$.

```
SELECT
  CASE TEMP_src.val = 0 OR TEMP_body.ref_v IS NULL
    WHEN 1 THEN NULL
    ELSE TEMP_src.res END as res,
  CASE TEMP_src.val = 0 OR TEMP_body.ref_v IS NULL
    WHEN 1 THEN 0
    ELSE 1 END as val,
  TEMP_src.ref_v' as ref_v', for each v' ∈ SVars(s),
  TEMP_body.ref_v' as ref_v', for each v' ∈ SVars(b) \ SVars(s) \ {v}
FROM (map_e(s)) as TEMP_src
LEFT JOIN (
```

```
    SELECT * FROM (map_e(b)) as TEMP
    WHERE TEMP.res = 1) AS TEMP_body
  ON TEMP_src.res = TEMP_body.ref_v,
    TEMP_src.ref_v' = TEMP_body.ref_v', for each v' ∈ SVars(s).
```

– $v \notin \text{FVars}(b)$. Similarly, but the source and the body would need to be *joined* using a `JOIN`-clause.

OclIsUndefined

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = s.$ oclIsUndefined(). Then,

```
map_e(s. oclIsUndefined()) =
SELECT
  CASE TEMP_src.val = 0
    WHEN 1 THEN NULL
    ELSE TEMP_src.res IS NULL as res END as res,
  TEMP_src.val as val,
  TEMP_ref.v' as ref_v', for each v' ∈ SVars_e(s)
FROM map(s) as TEMP_src.
```

Flatten

Let e be an OCL2PSQL-expression. Let e' be a subexpression of e. Let $e' = e''$ −>flatten(). Currently, e'' must be a collect-expression, $e'' = s$ −>collect($v \mid b$), where b is of type Col(t), where Col is either Set or Bag, and where t is a predefined type or a class-type.

We need to consider the following cases:

– $\text{FVars}(e') = \emptyset$.

```
SELECT
  TEMP_src.res as res,
  1 as val
FROM (map_e(e'')) as TEMP_src
  WHERE TEMP_src.val = 1
```

– $\text{FVars}(e') \neq \emptyset$. Let assume that $v \in \text{FVars}(b)$.

```
SELECT
  CASE TEMP_flat.val IS NULL
    WHEN 1 THEN NULL
    ELSE TEMP_flat.res END as res,
  CASE TEMP_flat.val IS NULL
    WHEN 1 THEN 0
```

```
      ELSE TEMP_flat.val END as val,
    TEMP_flat.ref_v' as ref_v', for each v' ∈ SVars(s)
  FROM (map_e(s)) as TEMP_src
  LEFT JOIN (
    SELECT * FROM (map_e(e'')) as TEMP
    WHERE TEMP.val = 1) as TEMP_flat
  ON TEMP_src.ref_v' = TEMP_flat.ref_v', for each v' ∈ SVars(s)
```

5 Preliminary Benchmarks

As part of the work presented here, we have implemented in Java the OCL2PSQL mapping. The interested reader can experiment with the latest version of our tool at:[2]

http://cs.vgu.edu.vn/se/tools/ocl2psql/.

In this section we provide some experiments to evaluate our mapping with respect to execution-time efficiency of the generated queries.[3] To this end, we will consider different *scenarios* of the database `CarDB`, introduced in Example 1. 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'`. We use a server machine, with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz with 16 GB RAM, using MySQL 5.7.25.[4] The execution-times reported here correspond to the arithmetic mean of 50 executions. The figures reported are given in seconds, unless otherwise stated.

First, 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()
```

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. The figures shown in the row "OCL2PSQL" correspond to the execution-time of the SQL-query generated, for the above OCL expression, by the OCL2PSQL mapping.

[2] At the time of writing, our tool temporarily uses an OCL parser that requires adding contextual information to the OCL expressions to be input. Please, check the latest information about this issue in our tool's web site.

[3] SQL engines have highly optimized strategies for executing queries over large databases. Our OCL2PSQL mapping follows some of the well-known optimization "tips" for SQL engines. Nevertheless, we should be aware that "development [for SQL engines] is ongoing, so no optimization tip is reliable for the long term." (MySQL 8.0 Reference Manual, (13.2.11.11 Optimizing Subqueries).

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

	CarDB(6)	CarDB(7)
SQL	0.22	2.28
OCL2PSQL	0.26	3.30

Next, 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,

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

We can specify in OCL the original query using the following expression:

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

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.

	CarDB(6)	CarDB(7)
SQL	0.22	2.72
OCL2PSQL	0.28	3.28

Finally, suppose that we want to know the number of cars that has at least one owner whose name is 'no-name'. In SQL 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;
```

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()
```

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.

	CarDB(6)	CarDB(7)
SQL	0.04	0.28
OCL2PSQL	0.36	4.24

6 Related Work

To the best of our knowledge, OCL2SQL [6,10] was the first attempt of mapping OCL into SQL. Based on a set of transformation *templates*, OCL2SQL automatically generates SQL queries from OCL expressions. 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, [4] 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 min 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]

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

An interesting method for efficiently checking OCL constraints by means of SQL queries is proposed in [13]. According to this method, an OCL constraint is satisfied if its corresponding SQL query returns the empty set. As in the case of OCL2SQL, this method is limited to (a subset of) 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."

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, with respect to its predecessor, SQL-PL4OCL simplifies the definition of the mapping, improves the execution-time of the generated queries (by reducing the number of temporary

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

tables), and implements some of the features that were left in [8] as future work: namely, handling the *null* value and supporting (unparameterized) sequences.

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:

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

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 min. As reported in Sect. 5, the SQL-query generated by OCL2PSQL takes less than 10 seconds to execute on the scenario `CarDB`(7). 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 create 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 iterator's 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.

Finally, there have been also different proposals [1,3,4,9] 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 [4], the (insurmountable) problem with OCL evaluators is the time required for *loading* a large scenario: none of the existing tools were able to finish loading a scenario with 10^6 objects after 20 min.

7 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 a main 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 secure database-centric applications [2].

In this paper, we have defined (and implemented) a novel mapping, called OCL2PSQL, from a significant subset of OCL to SQL. Our mapping generates queries that can perform *on par with* queries manually implemented in SQL for non-trivial examples, overcoming the main limitations of previously proposed mappings.

Looking ahead, we recognize that manually implementing in SQL *complex* queries is not an easy task; in 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 `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.

Our challenge now is two-fold. On the one hand, we want to extend our mapping to cover the part of the OCL language that is not covered yet. In particular, (i) collection of collections (e.g., OCL expressions that represent set of sets, or sequences, or ordered sets); (ii) operations on collections of collections (e.g., union, intersection); (iii) type operations (e.g., `oclIsTypeOf` or `oclAsType`, which are particularly relevant when dealing with UML models that include *generalizations*); and (iv) invalid values. On the other hand, we want to formally prove the *correctness* of our mapping, based on the semantics of SQL and OCL, using an interactive theorem prover (proof assistant) like Isabelle/HOL or Coq.

Finally, we plan to use our mapping to generate appropriate executable code from UML/OCL models containing OCL invariants, and pre and post conditions. Notice that these are ultimately OCL queries of type Boolean, and therefore are covered by our mapping.

Acknowledgments. This work has been supported by the Vietnamese-German University (VGU-PSSG grant 14/01 - 11/06/2019).

References

1. Baar, T., Markovic, S.: The RoclET tool (2007). http://www.roclet.org/index.php
2. Basin, D.A., Clavel, M., Egea, M., de Dios, M.A.G., Dania, C.: A model-driven methodology for developing secure data-management applications. IEEE Trans. Softw. Eng. **40**(4), 324–337 (2014)
3. Chiorean, D., Bortes, M., Corutiu, D., Botiza, C., Carcu, A.: An OCL environment (OCLE) 2.0.4 (2005). Laboratorul de Cercetare in Informatica, University of BABES-BOLYAI. http://lci.cs.ubbcluj.ro/ocle/

4. Clavel, M., Egea, M., de Dios, M.A.G.: Building an efficient component for OCL evaluation. ECEASST **15** (2008)
5. de Dios, M.A.G., Dania, C., Basin, D., Clavel, M.: Model-driven development of a secure eHealth application. In: Heisel, M., Joosen, W., Lopez, J., Martinelli, F. (eds.) Engineering Secure Future Internet Services and Systems. LNCS, vol. 8431, pp. 97–118. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07452-8_4
6. Demuth, B., Hussmann, H.: Using UML/OCL constraints for relational database design. In: France, R., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 598–613. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-46852-8_42
7. Egea, M., Dania, C.: SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. Softw. Syst. Model. **18**, 769–791 (2017)
8. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: a stored procedure-based MySQL code generator for OCL. ECEASST **36** (2010)
9. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**, 27–34 (2007)
10. Heidenreich, F., Wende, C., Demuth, B.: 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. Oriol, X., Teniente, E.: Incremental checking of OCL constraints through SQL queries. In: Brucker, A.D., Dania, C., Georg, G., Gogolla, M., (eds.) OCL@MoDELS, volume 1285 of CEUR Workshop Proceedings, pp. 23–32. CEUR-WS.org (2014)
14. ISO/IEC 9075-(1–10) Information technology - Database languages - SQL. Technical report, International Organization for Standardization (2011). http://www.iso.org/iso