**Noname manuscript No.**
(will be inserted by the editor)

# SQL-PL4OCL : An automatic code generator from OCL to SQL Procedural Language

Marina Egea · Carolina Dania

the date of receipt and acceptance should be inserted later

**Abstract** In this paper we introduce a SQL-PL code generator for OCL expressions that, in contrast to other proposals, is able to map OCL iterate and iterator expressions thanks to our use of stored procedures. We explain how the mapping presented here introduces key differences to the previous version of our mapping in order to i) simplify its definition, ii) improve the evaluation time of the resulting code, and iii) consider OCL three-valued evaluation semantics. Moreover, we have implemented our mapping to target several relational database management systems (RDBMS), i.e., MySQL, MariaDB, PostgreSQL, and SQL server, which allows us to widen its usability and to benchmark the evaluation time of the SQL-PL code produced.

## 1 Introduction

Model building is at the heart of system design. This is true in many engineering disciplines and is increasingly the case in software engineering. Model-driven engineering (MDE) [16] is a software development methodology that focuses on creating models of different system views from which system artifacts such as code and configuration data are automatically generated.This vision has already produced results that are available for industrial practice, but these results are only partial and specific to certain domains and languages.

The best possible scenario occurs when a source modeling language can be perfectly linked to a target language of election. Namely, a well defined mapping bridges the gap between the source and the target language. Otherwise, manual encoding of the system design is cumbersome and error prone. Moreover, keeping the resulting code and the design views synchronized is very difficult since any changes in each of them will require manual changes to the other part.

In this setting, we provide the definition of a mapping comes to bridge the gap between chosen source and target languages with the aim of saving the effort and exposition to errors that a manual translation conveys. More concretely, we introduce a SQL-PL[1] code generator for OCL expressions. Namely, our source language is the Object Constraint Language (OCL) [23] that is used to express constraints and queries using a textual notation on UML models. Our target language is the procedural language (PL) extension to the Structured Query Language (SQL). SQL is a special-purpose programming language designed for managing data in relational database management systems (RDBMS). The purpose of PL for SQL is to combine database language and procedural programming language.

A variety of applications arises for a mapping from OCL to SQL expressions. Among others, there are three prominent types. These are i) evaluation of OCL expressions (analysis queries and metrics) on large model's instances, (in line with the discussions in [6, 19]) ii) identification of constraints during data modeling that have to be checked as integrity constraints on actual data (in line with the discussion in [25]); iii) automatic code generation from models (in line with the discussion in [2]).

In the past, we explored other strategies to address i) and contribute to iii). To address i) we built EOS, an

M. Egea is with Minsait (by Indra), Madrid, Spain. E-mail: msegea@minsait.com
C. Dania is with IMDEA Software Institute, Madrid, Spain. E-mail: carolina.dania@imdea.org

[1] Please, notice that in this paper SQL-PL stands simply for SQL Procedural Language. It is not bound to any proprietary PL dialect.

efficient Java component for OCL evaluation [6]. Moreover, we contributed our previous OCL to MySQL mapping to advance iii). It was used as a key component of a toolkit [2] that automatically generated ready-to-deploy web applications for secure data management from design models. The security policies that the toolkit handled were written in SecureUML [3] over a data model. SecureUML extends role-based access control policies with dynamic authorization constraints that have to be evaluated at runtime. Our component was used to map and evaluate these OCL constraints specified in the SecureUML policies.

OCL is an OMG [23] and ISO standard [14] specification language. As part of UML, it was originally intended for modeling properties that could not be easily or naturally captured using graphical notation (e.g., class invariants in a UML class diagram).

SQL is also an ISO standard [31]. However, SQL full standard is divided into several parts dealing with different aspects of the language or its processing. Also, different RDBMS implement certain syntactic variations to the standard SQL notation. Thus, we had to adapt the implementation of our mapping to each of them. As implementation targets we selected MariaDB [17], PostgreSQL [27], and MS SQL Server [18]. Also, we kept MySQL [20] which was our first target. MariaDB and PostgreSQL were selected because they are open source and widely used by developers. MS SQL server was selected to be able to compare evaluation time from open source to commercial RDBMS. Yet, it is in our roadmap to implement our mapping into other commercial engines like Oracle 12c or the Adaptive Server Enterprise/Anywhere RDBMS by Sybase, among others. Our code generator is defined recursively over the structure of OCL expressions and it is implemented in the SQL-PL*4*OCL tool that is publicly available at [10]. In the following sections, we discuss the structure of the code produced by our new mapping, provide examples, and benchmark query evaluation time in MariaDB, MySQL, PostgreSQL, and MS SQL Server.

The seminal work of the mapping presented here can be found in [13,9]. The key idea that enables the mapping from OCL iterator expressions to iterative stored procedures remains the same, but the work detailed in this paper introduces a novel mapping from OCL expressions to SQL-PL stored procedures.

The most remarkable differences are stated in Remark 1.

*Remark 1* Key differences to our previous mapping

i. Each OCL expression, either non-iterator or (nested) iterator expression is mapped into just one stored procedure.
ii. The evaluation of the source OCL expression once mapped is retrieved by executing exactly one *call-statement*. This call-statement provokes the execution of the procedure and, in particular, the execution of an SQL query written in the last part of the outermost block of the procedure that retrieves the evaluation of the OCL expression.
iii. We only use temporary tables for *intermediate* and *final* values' storage. Final values' tables hold the resulting value of a query execution.
iv. We have adapted our mapping to deal with the three-valued semantics of OCL.

Decisions (i) and (ii) have facilitated the recursive definition of the code generator and simplifies its definition. Decision (iii) has significantly decreased the time required for the evaluation of the code generated. Feature (iv) enables to deal properly with the three-valued evaluation semantics of OCL. In addition, our original work and implementation was intended only for the procedural extension of MySQL, while our new definition eased the implementation of the mapping into other relational database management systems. In turn, we can now evaluate the resulting code using different RDBMS, which permits us to widen our discussion regarding efficiency in terms of *evaluation-time* of the code produced by SQL-PL*4*OCL tool.

Organization

In Section 2 we explain the basics about the source and target languages of our mapping, namely, OCL and SQL-PL. In Section 3 we explain how OCL contextual models are mapped to databases' schemas and records. In Section 4 we summarize the main ideas behind our mapping definition and explain the expected new structure of the PL blocks of code. Section 5 provides details about the definition that map OCL to SQL-PL expressions. In Section 6 we explain the architecture of the SQL-PL*4*OCL tool, how syntactic variations among the DBMS are tackled, and benchmark the times obtained by evaluating examples into the different engines. Finally, Sections 7 and 8 discuss related work, future work and conclusions.

## 2 Background

Data Models

We use a strict subset of UML class diagrams for modeling the data. This restricted modeling language is used as the contextual model for OCL. It essentially provides a simplified subset of UML class models where

*classes* can be related by *associations* and may have *attributes*. Also, classes may be related by generalization relationships. *Attributes* may have either primitive or class types and *association-ends* have class types.[2] As expected, the type of an attribute is the type of the attribute values, and the type associated to an association-end is the type of the objects which may be linked at this end of the association.

Object Constraint Language (OCL): Constraints and Queries

The Object Constraint Language (OCL) [14] is a pure specification language, also considered as a textual modeling language. In fact, OCL expressions are always written in the context of a model, and they are evaluated on scenarios of this model. This evaluation returns a value but does not change anything of the model: OCL is a side-effect free language. OCL can be used as a constraint language and as a query language, i.e., OCL can be used to analyse models and to validate them over selected scenarios or concrete system states as well as to launch arbitrary queries upon models.

We summarize next the main elements of the OCL language which are used in this paper. OCL is a strongly typed language. Expressions either have a primitive type (namely, Boolean, Integer, Real, and String), a class type, or a collection type (built up on a element type that may be either a primitive type or a class type). OCL distinguishes three different collection types: Set, Sequence, OrderedSet and Bag. Set means a mathematical set. It does not contain duplicate elements. A Bag is like a Set, which may contain duplicates (it corresponds to the mahematical structure multiset); that is, the same element may be in a bag twice or more times. A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them. OCL provides the standard operators on primitive types and on collections. For example, the operator includes checks whether a given object is part of a collection, and the operator isEmpty checks whether a collection is empty. Furthermore, OCL provides a dot-operator to navigate to the properties of the objects, i.e., objects' attributes and association-ends, and to access some operations. For example, let $u$ be an object of the class Car. Then, the expression $u$.model refers to the value of the attribute model for the Car $u$, and the expression $u$.owners refers to the objects linked to the Car $u$ through the association-end owners. In addition, OCL provides the operator allInstances to retrieve all instances of a class. For example, the expression Car.allInstances() refers to all the objects of the class Car. Finally, OCL provides operators to iterate on collections as forAll, exists, select, reject, one, and collect. E.g., Car.allInstances()−>select(u|u.model='BMW') refers to the collection of objects of the class Car whose attribute model has the value 'BMW'.

[2] We only consider binary associations and we do not consider attributes of entity or collection types.

## 2.1 Structured Query Language (SQL): Queries and Stored Procedures

The Structured Query Language (SQL) 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. Accordingly, SQL commands can be divided into two: the Data Definition Language (DDL) that contains the commands used to create and destroy databases and database objects; and the Data Manipulation Language (DML) that can be used to insert, delete, retrieve and modify the data stored in databases. Although SQL is to a great extent a declarative language, it also includes procedural elements.

Currently, SQL corresponds to an ISO standard [31]. However, issues of SQL code portability between major RDBMS products still exist due to lack of full compliance with, or different interpretations of, the standard. Among the reasons mentioned are the large size and incomplete specification of the standard, as well as vendor lock-in. For the work presented in this paper, we actually use as a target language a procedural extension of SQL which was originally developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. It was later adopted by other RDBMS. Namely, PL/pgSQL in PostgreSQL, stored procedures in MySQL and MariaDB, or TransactSQL (T-SQL) in SQL Server.

In particular, the procedural extensions to SQL support stored procedures which are routines (like a subprogram in a regular computing language) that are stored in the database. The procedural extension to SQL allows sending an entire block of statements to the database at one time within a stored procedure. A stored procedure has a name, may have a parameter list, and a SQL statement, which can contain many other SQL statements. The procedural languages are designed to extend the SQL's abilities while being able to integrate well with SQL. Yet, stored procedures cannot be called within SQL queries.

```
create procedure program_name()
begin
  begin
    begin
      ...
    end;
  end;
  ...
end;
```

Fig. 1: Nested blocks structure in Stored Procedures

```
create procedure program_name()
begin
  begin
    ...
  end;
  ...
  begin
    ...
  end;
  ...
end;
```

Fig. 2: Sequencial blocks structure in Stored Procedures

Stored procedures provide a special syntax for local variables, error handling, loop control, if-conditions and cursors, and flow control which allow the definition of iterative structures. Within stored programs, `begin-end` blocks are used to enclose multiple SQL statements, namely, to write compound statements. A block consists of various types of declarations (e.g., variables, cursors, handlers) and program code (e.g., assignments, conditional statements, loops). The order in which these can occur in a routine body is the following 1) variable and condition declarations; 2) cursor declarations; 3) handler declarations; 4) program code.

Moreover, `begin-end` blocks have two other features that are particularly useful in our case: (i) `begin-end` blocks can be nested; (ii) variables declared in outer `begin-end` blocks are visible in the inner blocks at any level of depth. Both of these features are crucial in our mapping to easily and recursively map OCL expressions that contain nested operators expressions. Figure 1 gives an idea of the structure that nested blocks adopt within stored procedures. Another case is OCL sequential operators; in such case, these are mapped into sequential blocks. Figure 2 gives an idea of the structure that sequential blocks adopt within stored procedures. Futhermore, we can have a combination of sequential and nested operators, in that case, the stored procedure will have a combination of sequencial and nested blocks. Finally, to invoke a stored procedure, we use the `call` statement; i.e. the routines showed in the Figure 1 or Figure 2, are invoked by the following statement:

```
call program_name
```

## 3 Mapping Data Models to databases

In this section, we will explain how a restricted subset of UML class diagrams (i.e., data models) and object diagrams are mapped to SQL-PL tables by our code generator. We will introduce first how we map OCL types to SQL-PL types. Second, we will detail the definition of our code generator.

### 3.1 A brief description of the relation between OCL and SQL type systems

OCL is a contextual language which takes syntactic constructs from its contextual model. But, independently of the contextual model, the OCL type system contains the primitive types Boolean, Integer, Real and String. Our code generator maps these types to the following SQL types: `Boolean`, `Int`, `Real`, and `Varchar`( 250), respectively. When the contextual model for the OCL expressions is a structural model, like our data model, the OCL type system also contains one class type for each class specified in the class diagram. In this section, we will also explain how our code generator maps UML class types to SQL tables. Collection types are also present in OCL, for instance, Set, Bag, OrderedSet, and Sequence that may take as a parameter a primitive type, or a class type, e.g., Set(Integer). These types do not have a direct mapping to SQL since SQL type system does not have collection types. However, the result of an OCL query may be a collection of elements, and the execution of the code generated in SQL to translate this OCL query will also return a collection of elements. Collection of collections are also possible in OCL. These are collection types taking as parameter another collection type, for example, Bag(Set(Car)). We decided not to map collection of collections to SQL since the complexity added to our code generator would be major and, on the other hand, they are difficult to use by designers or developers unless they have an advanced knowledge of the OCL language. In [13] we mentioned an strategy that is still valid for the mapping presented here. Namely, to cover collections of collections we have to modify our queries $codegen_q(exp)$ in order to obtain more structured result-sets. More concretely, to cope with expressions denoting types, each

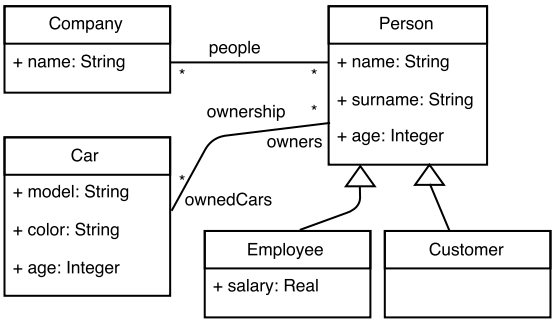



Fig. 3: Example: The Car Company model

element in the result-set of a query produced by our code generator shall not only hold a value, but also its type. Then, to cope with expressions defining collection of collections, the result-set returned by executing the query produced by our code generator shall take the form of a left-join, in which all the elements of the same subcollection are joint together. Like collection types, OCL tuple types cannot be mapped to SQL types, however, we could implement the evaluation semantics of OCL tuples by expanding the strategy that we apply for sequence types. Namely, we could perform the evaluation of each of the n-tuples separatedly and ensure the allocation of each tuple evaluation result in a different table's column. Due to the complexity it would add to our code generator, we leave this discussion out of the scope of this paper. Last but not least, the OCL special types, i.e., Invalid, Void, and Any do not have a counterpart in SQL either. Yet, the null value which is the unique value of the Void type, is mapped to the **`null`** value of SQL. We do not consider the invalid value in our mapping.

### 3.2 Guiding example: The Car-Company model

Let us now introduce a `Car-Company` model that we will use as our guiding example. The `Car-Company` model shown in Figure 3 is a data model that contains five classes: the class Car, the class Company, the class Person, and two subclasses of the latter: Employees and Customer, which are used, respectively, to distinguish among employees and customers of the company. The class Company has an association, people, to the class Person to indicate that objects of type Company are related to objects of type Person. The classes Car and Person are related by an association to reflect that cars sold by the company may be owned by people, either customers or employees, who may also buy a car. The association is called ownership, and its association ends are, respectively, ownedCars and owners. The class Company has the attribute name of type String. The class Car has the attributes model, and color of type String, and the attribute price, of type Real. The class Person has the attributes name, surname, of type String, and age, of type Int. The class Customer inherits the attributes specified in the class Person. In addition to the attributes inherited from the class Person, the class Employee has the attribute salary of type Real.

### 3.3 Mapping Data and Object models to SQL-PL tables and records

Our code generator maps the underlying data and object models (i.e., the 'context' and the evaluation scenario of the OCL queries) to SQL-PL tables and records (resp.) following the next (rather) standard rules.

Let $M$ be a class diagram and let $O$ be an instance of $M$. Then,

Class. Each class $A$ in $M$ is mapped to a table $nm(A)$[3], which contains, by default, a column `pk` of type `Int` as its primary key. Then, each object $o$ in $O$ of class type $A$ is represented by a row in table $nm(A)$ and is identified by a unique value placed automatically in the column `pk` ($> 0$ and not null). This value is also automatically incremented ($+1$) each time a new row is inserted.

Class attribute. Given a class $A$, each attribute $W$ of $A$ is mapped to a column $nm(W)$ of table $nm(A)$, being the type of $nm(W)$ the corresponding type of $W$, according to the rules for mapping types that we introduced at the beginning of this section. Then, the value of $W$ for an object $o$, instance of class $A$, is mapped to the value held by the column $nm(W)$ for the record that is identified by the `pk` value assigned to $o$ in table $nm(A)$[4].

Association. Given two classes $A$ and $B$, each many-to-many association $P$ between $A$ and $B$, with association-ends $rl\_A$ (at the class $A$) and $rl\_B$ (at the class $B$), is mapped to a junction table $nm(P)$, which contains two columns $nm(rl\_A)$ and $nm(rl\_B)$, both of type `Int`. Then, a $P$-link between an object $o$ of class $A$ and an object $o'$ of class $B$ is represented by a row in table $nm(P)$, where $nm(rl\_A)$ holds the key denoting $o$ and $nm(rl\_B)$ holds the key denoting $o'$ as foreign keys' references.

For one-to-many associations, we add a foreign key column on the table corresponding to the class in the *many-side* of the relationship. This column holds

[3] $nm()$ generates unique names for classes, attributes, and associations.

[4] Fig. 4 shows the resulting table for a simple Car-Company model.

the key value referencing the object linked in the *one-side* of the association.

Inheritance. Each class $C$, subclass of a class $A$, is mapped to a table $nm(C)$ together with its direct (i.e., not inherited) attributes and associations following the definitions described above. But, in addition, a foreign key column, `fk`, is added to $nm(C)$ referencing the primary key column of the table $nm(A)$ that maps class $A$.

Although it is not completely obvious, this definition is controlling how tables which correspond to classes related by inheritance are populated. We avoid discussing it further here since it would add a complexity that is not of direct value to the presentation of our code generator. Yet, we provide examples next that will help to understand the rationale behind our definition. The interested reader can find the details in [9].

*Remark 2* The above mapping rules assume that source data models satisfy the following (rather) natural constraints:

- Each class has a unique name.
- Each attribute within a class has a unique name.
- A class cannot inherit properties, i.e., association ends or attributes, that have the same name along inheritance relationships.
- Each association is a binary relation that is uniquely characterized by its association-ends. Moreover, the association-ends in a self-association have different names.

*Mapping the Car-Company model to a database structure.*

From now on we will choose MariaDB (fully compatible with MySQL) syntax to illustrate the code generated by our mapping, both for the definitions and the examples.

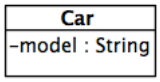
The command that is automatically generated to map the class Person to a SQL table is:

```
create table Person (
 pk int not null primary key auto_increment,
 name varchar(250),
 surname varchar(250),
 age int);
```

Similarly, the classes Car and Company are mapped to tables.

The command that is automatically generated to map the class Employee to a SQL table is:

```
create table Employee (
 pk int not null primary key auto_increment,
 salary int, fkPerson int,
 foreign key (fkPerson)
   references Person(pk));
```




(a)

| pk | model |
|---|---|
| | |

(b)

Fig. 4: (a) Simple Car company model. (b) Car company table.

Similarly, the class Customer is mapped to a table.

The command that is automatically generated to map the association ownership to a SQL table is:

```
create table ownership (
 owners int,
 ownedCars int,
 foreign key (owners) references Person(pk),
 foreign key (ownedCar) references Car(pk));
```

Similarly, the association people is mapped to a table.

Please, notice that in the structure of the tables that we create for the subclasses Employee (and Customer), the subclasses hold an additional column `fkPerson` as a foreign key to the primary key of the table `Person` that corresponds to their parent class.

## 4 SQL-PL4OCL : structure of the generated code in a Nutshell

In this section we briefly introduce the novel structure of the code produced by our SQL-PL generator for OCL expressions. This section is intended to help the understanding of our mapping definition in the following section. For any input OCL expression, our code generator always produces a stored procedure that can be invoked using a call statement, as we explain next.

Given an OCL expression *exp*, our code generator *patternproc*(*exp*) generates the following pattern.

```
1 create procedure nm(exp)()
2 begin
3   codegen_b(exp)
4   codegen_q(exp);
5 end;//
6 call nm(exp)//
```

The generated code contains the declaration of the stored procedure (lines 2-5), headed by its creation command and name (line 1). The main block is enclosed by the delimiters `begin-end`. The code contained by the main block is generated by the auxiliary functions

$codegen_b(exp)$ and $codegen_q(exp)$ (lines 3-4). These functions generate code that mirrors the structure of the OCL expressions. The role of the function $codegen_b(exp)$ is to generate code when the mapping of the expression, *exp*, needs of an auxiliary block definition. The role of the function $codegen_q(exp)$ is always to generate a query that retrieves the values corresponding to the evaluation of *exp*. Finally, the function $patternproc(exp)$ also generates a `call`-statement to execute the stored procedure (line 6).[5]

*Simple expressions.* There are cases in which the function $codegen_b(exp)$ does not generate any code. It happens when *exp* is a kind of expression that does not need any auxiliary block definition within the stored procedure to be mapped. Examples of this kind of expressions are operators over classes, operators between sets or bags, math operators, etc..

*Example 1* The code generated by $patternproc(exp)$ for the expression *exp*=Car.allInstances() is:

```
create procedure carallinstances()
begin
  codegen_q(exp);
end;//
call carallinstances//
```

Where $codegen_q(exp)$ generates the following specific code:

```
select Car.pk as val from Car
```

Note that when the stored procedure is executed, the result is a table containing a column called `val`, which holds all the values of the column pk (primary key) from the records of table Car. □

*Example 2* Consider now the expression *exp*=Car.allInstances().model. The code generated by $patternproc(exp)$ is:

```
create procedure modelallinstances()
begin
 codegen_q(exp);
end;//
call modelallinstances//
```

Where $codegen_q(exp)$ generates the following specific code:

```
select Car.model as val
from (select pk as val from Car) as t0
left join Car on Car.pk = t0.val
```

[5] Please, note that our delimiter in SQL-PL is set to '//'.

Note that when the stored procedure is executed, the result is a table containing a column called `val`, which holds all values of the column model from the records of the Car table. □

*Example 3* Consider the following OCL expression *exp*, $exp = exp_1 -> \text{notEmpty}()$, where $exp_1$ is an expression which does not contain any operator subexpression that requires a block definition, then $patternproc(exp)$ generates the following code:

```
create procedure exp1notEmpty()
begin
  select count(*) > 0 as val
  from (codegen_q(exp_1)) as t1;
end;//
call exp1notEmpty//
```

□

In what follows, we will see how our code generator can recursively deal with the recursive structure of OCL expressions.

*Complex expressions.* There are other cases for which the function $codegen_b(exp)$ does generate code because mapping a given expression, *exp*, needs of an auxiliary block definition. This auxiliary block is required either for the expression to be properly mapped or because we have noticed that it brings efficiency to the execution. For example, in some cases we noticed that executing a given sequence of operations within a block required less time than executing a given SQL query, and we tailored our mapping accordingly. We consider occurrences of complex expressions to operators over sequences, iterators, etc. Next, we sketch the idea of our mapping in these cases and provide examples.

Sequence Operators.

Let *exp* be a sequence expression. Let the shape of this expression be $op(exp_1,\ldots,exp_n)$ and consider that the subexpressions $exp_1,\ldots,exp_n$ need to be mapped into blocks too. Then, $codegen_b(exp)$ generates the SQL-PL blocks:

```
begin
codegen_b(exp_1)
...
codegen_b(exp_n)
drop table if exists nm(codegen_b(exp));
create temporary table nm(codegen_b(exp))
 (pos int not null auto_increment,
 val basictype(exp), primary key(pos));
```

```
insert into nm(codegen_b(exp))(val)
  (codegen_q(exp_1));
...
insert into nm(codegen_b(exp))(val)
  (codegen_q(exp_n));
end;
```

while, $codegen_q(exp)$ generates:

```
select * from nm(codegen_b(exp));
```

Note that $basictype(tp)$ is the SQL type associated to the UML type $tp$.

*Example 4* Consider now the expression $exp$='hi'.characters().union('ho'.characters()). Then, the code generated by $patternproc(exp)$ is:

```
create procedure unionLits()
begin
 codegen_b(exp)
 codegen_q(exp);
end;//
call unionLits//
```

Where $codegen_b(exp)$ generates the following specific code:

```
begin
-- sub-block 'hi'.sequence()
begin
drop table if exists wchars;
create temporary table wchars
  (pos int not null auto_increment,
  val varchar(250), primary key(pos));
insert into wchars(val)
  (select 'h' as val);
insert into wchars(val)
  (select 'i' as val);
end;
-- sub-block 'ho'.sequence()
begin
drop table if exists w1chars;
create temporary table w1chars
  (pos int not null auto_increment,
  val varchar(250), primary key(pos));
insert into w1chars(val)
  (select 'h' as val);
insert into w1chars(val)
  (select 'o' as val);
end;
-- code for operator union
drop table if exists unionLits;
create temporary table
  unionLits(val varchar(250));
insert into unionLits(val)
  (select wchars.val as val
  from wchars as t1
  order by wchars.pos asc);
insert into unionLits(val)
  (select w1chars.val as val
  from w1chars as t2
  order by w1chars.pos asc);
end;
```

While $codegen_q(exp)$ generates the following specific code:

```
select * from unionLits
```

Note that when a stored procedure is executed to evaluate a expression of Sequence type, the result is stored in a table containing two columns called pos and val, which holds all values (in the column val) ordered by the position given in the column pos.

Iterator expressions.

These expressions are of the form $src{->}iterOp(v|body)$ whose top-operator is an iterator operator.[6] For each iterator expression $exp$, our code generator produces a stored procedure composed of an iterative *block* and a *query* following the structure introduced at the begining of the section.

When the stored procedured is called, it

Step 1. creates a temporary table;
Step 2. executes, for each element in the $src$-collection that is instantiating the iterator variable $v$ the $body$ of the iterator expression;
Step 3. processes and stores in the table, created in Step 1, the result of the query $codegen_q(body)$, according to the semantics of the iterator operator.

The function $codegen_q(exp)$ generates a query that retrieves the values corresponding to the evaluation of $exp$ from the table that has been created and filled in during the execution of the iterative block of the stored procedure. Finally, as we shown before, the function $patternproc(exp)$ also generates a call-statement to actually execute the procedure $patternproc(exp)$.

*Example 5* Iterator expressions. Consider the expression $exp$ = Car.allInstances()->select(u|u.model='BMW'). The code generated by $patternproc(exp)$ is:

[6] For the sake of simplicity, we will consider here that the top-operator of $src$ is a *simple* expression. The case when the iterator expressions are nested deserve, however, a particular attention.

```
create procedure selectproc()
begin
 codegen_b(exp)
 codegen_q(exp);
end;//
call selectproc//
```

Where $codegen_b(exp)$, generates the following especific code:

```
begin
 declare done int default 0;                          2
 declare var int;
 declare crs cursor for                               4
   (select pk as val from Car);
 declare continue handler for                         6
   sqlstate '02000' set done = 1;
 drop table if exists selectproc;                     8
 create temporary table selectproc(val int);
 open crs;                                            10
 repeat
  fetch crs into var;                                 12
  if not done then
    if exists                                         14
      (select True from
        (select model = 'BMW' as val                  16
        from Car where pk = var) as t1) as t2
    then                                              18
     insert into selectproc(val) values (var);
  end if;                                             20
  until done end repeat;
 close crs;                                           22
end;
```

The definition of the block (line 1-23) contains the following declarations: first some variables are declared (lines 2-7); following Step 1, a new temporary table is created (note that it is deleted if it exists) (lines 8-9); following Step 2, for each element of the source (lines 11-12), the value of the result of the execution of the body is calculated; however, following Step 3, this value is only inserted into the new table (lines 18-19) if the condition of the body is satisfied (lines 13-21), according to the semantics of the iterator operation.

Finally, $codegen_q(exp)$ generates the following specific code:

```
select val from selectproc
```

Note that, as it happened for Example 1, the result of the execution of the stored procedure is a table containing a column called `val`, which holds all records of the table Car whose model is 'BMW'. □

To conclude, let us say that the potential complexity of the OCL expression is mirrored within the stored procedure by using the function $codegen_b(exp)$.

Within such procedure, the general idea that drives the mapping of OCL complex expressions is that OCL sequential operators are mapped to sequential blocks, and OCL nested operators are mapped to nested blocks. In addition, there will always be an outermost `begin-end` enclosing block that contains the query to retrieve the evaluation result when the procedure is invoked.

*Remark 3* Scope.

We do not cover yet completely the whole OCL language. However, we cover most of the operators listed in the OCL standard library [23, Chapter 11]. More concretely, we cover operators on primitive types String, Boolean, Integer and Real; operators on Set, Bag and Sequence types; and all iterator operators except orderBy and closure. Last but not least, we do cover nested iterator expressions, i.e., iterator expressions whose body also contains iterator expressions, for example, Person.allInstances()−>exists(p | p.ownedCars−>exists(c | p.ownedCars−>includes(c))). We will deal in detail with this type of expression in the following section. Yet, we do not support tuples or nested collections. Finally, we neither support static collections of AnyType, and we have to refer the null value explicity, i.e. null::String.

## 5 The SQL-PL4OCL code generator

In this section, we take advantage of the explanation about the structure of the code generated in previous section. It will allow the reader to understand more easily the definition of our mapping. Below, we provide the mapping definition for those operations from the OCL standard library [22, Chapter 11] that we have considered more illustrative. The exhaustive definition of the mapping for all the operations of the OCL standard library is provided in [10]. We start each definition with the name of the operator, followed by a brief description of its semantics, and the definition of its mapping.

### 5.1 Mapping simple OCL expressions.

In this section we show how we define our mapping for simple expressions. Recall from the previous section that these are expressions for which the top operator is mapped directly to a SQL query without the need of declaring auxiliary SQL-PL blocks. Fall within this category model specific operators, boolean, numeric, and collection operators for sets and bags.

Model specific operators.

There are operations in OCL that the language 'borrows' from the contextual model. These operations vary when the contextual model changes and they refer to association ends, classes' attributes and classes' identifiers.
In the following, we consider $exp_1$ to be an OCL expression of type class, or (not ordered) set or bag.

allInstances(). It returns all the instances of the class that it receives as argument. Let $exp$ be an expression of the form $C$.allInstances(), where $C$ is a class of the contextual model. Then, $codegen_q(exp)$ returns the followingSQL query:

```
select nm(C).pk as val from nm(C)
```

Attribute Expression. It retrieves an attribute's values of the instances returned by the source expression.

Let $exp$ be an expression of the form $exp_1.attr$ where $attr$ is an attribute of a class $A$. Then, $codegen_q(exp)$ returns the followingSQL query:

```
select nm(A).nm(attr) as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
left join nm(A)
 on al(codegen_q(exp_1)).val = nm(A).pk
```

Note that $al()$ generates a unique alias names for tables.

Association−End Expression. It retrieves the instances linked to the objects returned by the source expression through the association end.

Let $exp$ be an expression of the form $exp_1.rl\_A$ (resp. $exp_1.rl\_B$), where $rl\_A$ (resp. $rl\_B$) is the $A$-end (resp. $B$-end) of an association $P$ between two classes $A$ and $B$. Then, $codegen_q(exp)$ returns the followingSQL query:

```
select nm(P).nm(rl_A) as val from
(codegen_q(exp_1)) as al(codegen_q(exp_1))
left join nm(P)
 on al(codegen_q(exp_1)).val = nm(P).nm(rl_B)
where nm(P).nm(rl_A) is not null
```

In all cases previously described, the top expression $exp$ does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its subexpression:

```
codegen_b(exp_1)
```

*Example 6* Model specific operators. The following examples do only generate SQL queries. None of them need blocks for their definition, i.e., $codegen_b(exp)$ is empty in all cases.

*Q1.* Query the ages of all employees.

```
Employee.allInstances().age
```

```
select Person.age as val
from (
 select fkEmployee as val
 from (select pk as val from Employee) as t0
 left join Employee
 on t0.val = Employee.pk) as t1
left join Person on t1.val = Person.pk
```

Notice that since Employee is a subclass of Person that inherits from it the attribute age, we recover with the SQL query the column age of the table `Person`, but only for the rows contained by the table `Employee`. This is enforced by the left join used to align the foreign keys contained by the table `Employee` with the keys contained by the table `Person`.

*Q2.* Query the cars owned by all persons.

```
Person.allInstances().ownedCars
```

```
select ownership.ownedCars as val
from (select pk as val from Person) as t0
left join ownership
 on t0.val = ownership.owners
where ownership.ownedCars is not null
```

□

Boolean value returning operators.

In all cases described below, the top expression $exp$ does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its sub-expression:

```
codegen_b(exp_1)
```

isEmpty(). It returns 'true' if the source collection is empty, and 'false' otherwise. Let $exp$ be an expression of the form $exp_1$->isEmpty(). Then, $codegen_q(exp)$ is the following SQL query:

```
select count(*) = 0 as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
```

The operator isEmpty() does not require any block definition, thus $codegen_b(exp)$ is composed by the blocks of its subexpression (if any):

$codegen_b(exp_1)$

For the operator notEmpty(), '>' replaces '=' in the above SQL query.

includes. It returns 'true' if the source collection $exp_1$ contains the element $exp$.

Let $exp$ be an expression of the form $exp_1$->includes($exp_2$). Then, $codegen_q(exp)$ returns the following SQL query:

```
select codegen_q(exp_2) in codegen_q(exp_1) as val
```

The operator includes does not require any block definition, thus $codegen_b(exp)$ is composed by the blocks of its subexpressions (if any):

$codegen_b(exp_1)$
$codegen_b(exp_2)$

For the operator excludes, '**not in**' replaces '**in**' in the above SQL query.

includesAll. It returns 'true' if the collection $exp_1$ contains all the elements in the collection $exp_2$, and 'false' otherwise. Let $exp$ be an expression of the form $exp_1$->includesAll($exp_2$). Then, $codegen_q(exp)$ returns the following SQL query:

```
select count(al(codegen_q(exp_2)).val) = 0 as val
from (codegen_q(exp_2)) as al(codegen_q(exp_2))
where al(codegen_q(exp_2)).val
not in (codegen_q(exp_1))
```

The operator excludesAll returns 'true' if the collection $exp_1$ does not contain all the elements in the collection $exp_2$, and 'false' otherwise. For the operator excludesAll, '**not in**' replaces '**in**' in the above SQL-PL statement.

In all cases previously described, the expression $exp$ does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its subexpressions:

$codegen_b(exp_1)$
$codegen_b(exp_2)$

*Example 7* Boolean value returning operators. The following examples only need to generate SQL queries. None of them require a block definition. $codegen_b(exp)$, in all cases, is empty.

*Q3.* Query whether there are 'BMW' cars in the company.

Car.allInstances().model->includes('BMW')

```
select (select 'BMW' as val) in
 (select Car.model as val
 from (select Car.pk as val from Car) as t0
 left join Car
  on t0.val = Car.pk) as val
```

□

Numeric value returning operators.

Again, for all cases described below, the top expression $exp$ does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its sub-expression:

$codegen_b(exp_1)$

size. It returns the size of the source collection. Let $exp$ be an expression of the form $exp_1$->size(). Then, $codegen_q(exp)$ is the following SQL query:

```
select count(*) as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
```

sum. It returns the sum of the elements in the source collection that must be of numeric type. Let $exp$ be an expression of the form $exp_1$->sum().
Then, $codegen_q(exp)$ is the following SQL query:

```
select sum(val) as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
```

*Example 8* Numeric value returning operators. The following examples do only generate SQL queries. None of them need blocks for their definition, i.e., $codegen_b(exp)$ is empty in all cases.

*Q4.* Count the number of customers.

Customer.allInstances()->size()

```
select count(*) as val
from (select Customer.pk as val
      from Customer) as t0
```

□

Collection operators for Set and Bag types.

asSet. The set containing all the elements from the source collection, with duplicates removed (if any). Let $exp$ be an expression of the form $exp_1$->asSet(). Then, $codegen_q(exp)$ is the following SQL query:

```
select distinct al(codegen_q(exp1)).val as val
from (codegen_q(exp1)) as al(codegen_q(exp1))
```

union. It returns the set union (resp. multiset union) of both sets (resp. bags) passed as arguments to the operation. Let *exp* be an expression of the form $exp_1$−>union($exp_2$), where both $exp_1$ and $exp_2$ are sets. Then, $codegen_q(exp)$ returns the followingSQL query:

```
select al(codegen_q(exp1)).val
from (codegen_q(exp2) union codegen_q(exp1))
 as al(codegen_q(exp1))
```

When $exp_1$ or $exp_2$ are bags, then '`union all`' will replace '`union`' in the above SQL query. The operator including that returns the bag containing all elements of the source collection $exp_1$ plus the element $exp_2$ passed as argument is mapped exactly as the operator union is.

excluding. It returns the bag that results from removing the element $exp_2$ from the source collection $exp_1$. Let *exp* be an expression of the form $exp_1$−>excluding($exp_2$). Then, $codegen_q(exp)$ returns the following SQL query:

```
select al(codegen_q(exp1)).val
from (codegen_q(exp1)) as al(codegen_q(exp1))
where al(codegen_q(exp1)).val
 not in codegen_q(exp2)
```

*Example 9* Collection Operators. The following examples do only generate SQL queries. None of them need blocks for their definition, i.e., $codegen_b(exp)$ is empty in all cases.

*Q5.* Query the surnames of all customers but those whose surname is 'Smith'.

```
Customer.allInstances().surname->excluding('Smith')
```

```
select t2.val
from
 (select Person.surname as val
 from
  (select fkCustomer as val
  from (select pk as val from Customer) as t0
  left join Customer
  on t0.val = Customer.pk) as t1
 left join Person
 on t1.val = Person.pk) as t2
where t2.val not in (select 'Smith' as val)
```

5.2 Mapping complex OCL expressions.

In this section we introduce the mapping definition for those top operators whose definition needs to generate both SQL queries and blocks. Namely, sequence and iterator operators.

Sequence Operators.

In OCL there is an operation for building a sequence from a set or a bag of elements. This operation is asSequence(). Remember that, when we talk about a sequence in OCL we talk about a collection of elements that are assigned a position in a list. Sequences allow for duplicated elements.

asSequence(). Let *exp* be an expression of the form $exp_1$.asSequence(). Then, $codegen_b(exp)$ generates the SQL-PL blocks:

```
begin
 drop table if exists nm(codegen_b(exp));
 create temporary table nm(codegen_b(exp));
 insert into nm(codegen_b(exp))(val)
 select al(codegen_q(exp1)).val as val as
  from (codegen_q(exp1)) as al(codegen_q(exp1));
end;
```

while, $codegen_q(exp)$ generates:

```
select pos, val from nm(codegen_b(exp))
```

*Example 10* Sequence Operators.

*Q6.* Query the length of a sequence that contains all instances of Person.

```
Person.allInstances()->asSequence()->size()
```

```
begin
drop table if exists personAsSequence;
create temporary table personAsSequence
 (pos int not null auto_increment,
  val int, primary key(pos));
insert into personAsSequence(val)
 select t0.val as val as
 from (select pk as val from Person) as t0;
end;
select count(*) as val
from (select * from personAsSequence) as t1;
```

□

Mapping OCL iterator expressions.

Since the semantics of each OCL iterator operator can be defined through a mapping from the iterator to the iterate construct, we could have decided to translate the iterate expressions resulting from those mappings in order to generate code for the iterator operations like reject, select, forAll, exists, collect, one, sortedBy, isUnique and any by applying the iterate pattern. In fact, this was the decision made for the definition of the OCL2SQL code generator in [28], however they did not succeed in finding a pattern to map the iterate expressions and therefore the iterator expressions were not mapped either. Instead, we decided to generate code specifically for each iterator operator according to its semantics. In this way, we can generate code that is less complex and more tailored to the semantics of each iterator operator. Also this decision allows us, as we explain below, to end a block at an intermediate iteration step once the evaluation result of the translated iterator is clear. For instance, when the execution of the code generated to map the body of a forAll expression returns false at one iteration step, the procedure is terminated returning false.

The basic idea is therefore that, for each iterator expression *exp*, our code generator produces a SQL-PL block that, when it is called creates a table, denoted by $nm(codegen_b(exp))$, from which we obtain using a simple **select**-statement the values corresponding to the evaluation of *exp*. By now, we assume that the types of the *src*-subexpressions are either sets or bags of primitive or class types.

Let *exp* be an iterator expression of the form *src* $->iter\_op(var|body)$. Then, $codegen_q(exp)$ returns the followingSQL query:

```
select * from nm(codegen_b(exp));
```

While, $codegen_b(exp)$ generates the following scheme of SQL-PL blocks:

```
codegen_b(src)
begin                                                    2
declare done int default 0;
declare var [cursor-specific type];                      4
declare crs cursor for (codegen_q(src));
declare continue handler                                 6
 for sqlstate '02000' set done = 1;
drop table if exists nm(codegen_b(exp));                 8
create temporary table nm(codegen_b(exp))
 (val [value-specific type]);                            10
[Initialization-specific code
(only for forAll, one, exists and sortedBy)]
open crs;                                                12
repeat
 fetch crs into var;                                     14
 codegen_b(body)
 if not done then                                        16
  [Iterator-specific processing code]
 end if;                                                 18
until done end repeat;
close crs;                                               20
[End-specific code (only for isUnique)]
end;                                                     22
```

Basically, $codegen_b(exp)$ generates a block [lines 2–22] which creates the table $nm(codegen_b(exp))$ [line 9] and execute, for each element in the *src*-collection [lines 5,12-14], the *body* [line 15] of the iterator expression *exp*. More concretely, until all elements in the *src*-collection have been considered, $codegen_b(exp)$ repeats the following process: (*i*) it instantiates the iterator variable *var* in the *body*-subexpression, each time with a different element of the *src*-collection, which it fetches from $codegen_q(src)$ using a cursor [lines 12–14]; and (*ii*) using the so called "iterator-specific processing code", it processes in $nm(codegen_b(exp))$ the result of the query $codegen_q(body)$, according to the semantics of the iterator *iter_op* [line 17]. In addition, in the case of the four iterators: forAll, one, exists and sortedBy, the table $nm(codegen_b(exp))$ is initialized, using the so called "initialization-specific code" [line 11], and in the case of the iterator isUnique, an "end-specific code" is required. Moreover, for the iterators forAll and exists, the process described above will also be finished when, for any element in the *src*-collection, the result of the query $codegen_q(body)$ contains the value corresponding, in the case of the iterator forAll, to False or, in the case of the iterator exists, to True.

In the remaining of this subsection, we specify, for each case of iterator expression, the corresponding "value-specific type", "initialization-specific code" , "iterator-specific processing code" and "end-specific code" produced by our code generator when instantiating the general schema. Again, for all cases, the "cursor-specific type" is the SQL-PL type which represents, according to our mapping (see section 3.1), the type of the elements in the *src*.

forAll-*iterator.* Let *exp* be an expression of the form *src*`->forAll(`*var*|*body*`)`. This operation returns 'true' if *body* is 'true' for all elements in the source collection *src*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* boolean.
- *Initialization code:*

```
insert into nm(codegen_b(exp)) (val)
  values (True);
```

– *Iteration-processing code:*

```
update nm(codegen_b(exp)) set val = False
where (codegen_q(body)) = False;
if exists
  (select True from nm(codegen_b(exp))
   where val = False)
then set done = 1;
end if;
```

exists-*iterator.* Let *exp* be an expression of the form *src*−>exists(*var*|*body*). This operation returns 'true' if *body* is 'true' for at least one element in the source collection *src*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

– *value-specific type:* boolean.
– *Initialization code:*

```
insert into nm(codegen_b(exp)) (val)
 values (False);
```

– *Iteration-processing code:*

```
update nm(codegen_b(exp))
 set val = True
where (codegen_q(body)) = True;
if exists
  (select True from nm(codegen_b(exp))
   where val = True)
then set done = 1;
end if;
```

one-*iterator.* Let *exp* be an expression of the form *src* −>one(*var*|*body*). This operation returns 'true' if *body* is 'true' for exactly one element in the source collection *src*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

– *value-specific type:* boolean.
– *Initialization code:*

```
insert into nm(codegen_b(exp))(val)
 values (False);
set @counter = 0;
```

– *Iteration-processing code:*

```
if exists
(select nm(codegen_b(body)).val
 from (codegen_q(body)) as nm(codegen_b(body))
 where nm(codegen_b(body)).val = True)
then
 set @counter = @counter+1;
 update nm(codegen_b(exp)) set val = True;
end if;
if @counter = 2 then
 update nm(codegen_b(exp)) set val = False;
 set done = 1;
end if;
```

sortedBy-*iterator.* According to [23], it results in the OrderedSet containing all elements of the source collection ordered in descending order according to the values returned by the evaluation of the body expression. The order considered is given by the operation $<$ that should be defined on the type of the body expression. We consider instead the order given by the operation $\leq$ in order to be able to include in the resulting ordered set those elements for which the evaluation of the body returns exactly the same value.

Let *exp* be an expression of the form *src*−>sortedBy (*var*|*body*). This operation returns the collection of elements in the *src* expression ordered by the criterion specified by *body*.
The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

– *value-specific type:* the SQL type which represents, according to our mapping, the type of the *body*.
– *Initialization code:*

```
create temporary table nm_seq(codegen_b(exp))
  (pos int not null auto_increment,
   val [value-specific type]);
```

– *Iteration-processing code:*

```
insert into nm(codegen_b(exp))(val)
  codegen_q(body);
insert into nm_seq(codegen_b(exp))(val)
  (select val from nm(codegen_b(exp))
   order by val desc);
```

collect-*iterator.* Let *exp* be an expression of the form *src*−>collect(*var*|*body*). This expression returns the collection of objects that result from evaluating *body* for each element in the source collection *src*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

– *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the *body*.
– *Iteration-processing code:*

```
insert into nm(codegen_b(exp))(val)
  codegen_q(body);
```

select-*iterator*. Let *exp* be an expression of the form *src*−>select(*var*|*body*). This expression returns a subcollection of the source collection *src* containing all elements for which *body* evaluates to 'true'. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the elements in the *src*.
- *Iteration-processing code:*

```
if exists
  (select al(codegen_q(body)).val
   from (codegen_q(body)) as al(codegen_q(body))
   where al(codegen_q(body)).val = True)
then
  insert into nm(codegen_b(exp))(val)
    values (var);
end if;
```

reject-*iterator*. Let *exp* be an expression of the form *source*−>reject(*var*|*body*). This expression returns a subcollection of the source collection *src* containing all elements for which *body* evaluates to *false*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the elements in the *src*.
- *Iteration-processing code:*

```
if exists
  (select True
   from (codegen_q(body)) as al(codegen_q(body))
   where val = False)
then
  insert into nm(codegen_b(exp))(val)
    values (var);
end if;
```

isUnique-*iterator*. Let *exp* be an expression of the form *source*−>isUnique(*var* | *body*). This expression returns True if all elements of the collection of objects that result from evaluating *body* for each element in the source collection *src*, are different. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* boolean
- *Initialization code:*

```
create temporary table nm_acc(codegen_b(exp))
(val value-specific type);
```

  where *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the elements in the *body*.

- *Iteration-processing code:*

```
insert into nm_acc(codegen_b(exp))(val)
  codegen_q(body);
```

- *End code:*

```
insert into nm(codegen_b(exp))(val)
(select
al_1(codegen_q(exp)).val = al(codegen_q(exp)).val
from
  (select count(*) as val
  from
    (select distinct val
    from nm_acc(codegen_q(exp)))
      as al(codegen_q(body)))
        as al_1(codegen_q(body)),
  (select count(*) as val
  from nm_acc(codegen_b(exp)))
    as al(codegen_q(body));
```

*Example 11* Nested and sequential iterator expressions.

*Q7.* Check whether there is a car owner whose surname is Perez.

```
Car.allInstances()
  −>select(c| c.owners−>exists(p|p.surname='Perez'))
```

```
begin
declare done int default 0;
declare body Boolean default false;
declare var0 int;
declare crs cursor
  for select pk as val from Car;
declare continue handler
  for sqlstate '02000' set done = 1;
drop table if exists select0;
create temporary table select0(val int);
open crs;
repeat
  fetch crs into var0;
  begin
  declare done int default 0;
  declare result boolean default false;
  declare tResult int default 0;
  declare var01 int;
  declare crs cursor for
   (select ownership.owners as val
   from (select var0 as val) as t0
   left join ownership
   on t0.val = ownership.ownedCars
   where ownership.owners is not null);
   declare continue handler
  for sqlstate '02000' set done = 1;
```

```
  drop table if exists exists01;
  create temporary table exists01(val int);
  open crs;
  repeat
   fetch crs into var01;
   if not done then
    select val into tResult
    from
     (select (select Person.name as val
      from (select var01 as val) as t1
      left join Person
        on t1.val = Person.pk) =
      (select 'Perez' as val) as val) as t;
    if tResult then
     set done = 1;
     set result = 1;
    end if;
   end if;
   until done end repeat;
   insert into exists01(val) values (result);
   close crs;
  end;
  if not done then
 select val into body
  from (select * from exists01) as t;
 if body then
    insert into select0(val) values (var0);
end if; end if;
until done end repeat;
close crs;
end;
select * from select0;
```

*Q8.* Check whether exists a person, who owner a car, with surname Perez.

```
Car.allInstances()
 ->collect(p|p.owners)
 ->exists(q|q.surname='Perez')
```

```
begin
  begin
  declare done int default 0;
  declare var1 int;
  declare crs cursor for
    select pk as val from Car;
  declare continue handler
    for sqlstate '02000' set done = 1;
  drop table if exists collect0;
  create temporary table
    collect0(val boolean);
  open crs;
  repeat
    fetch crs into var1;
    if not done then
      insert into collect0(val)
        (select ownership.owners as val
        from (select var1 as val) as tbl1
        left join ownership
        on tbl1.val = ownership.ownedCars
        where ownership.owners is not null
          or tbl1.val is null);
    end if;
    until done end repeat;
    close crs;
  end;
  begin
    declare done int default 0 ;
    declare result boolean default false;
    declare tempResult boolean default false;
    declare var2 int;
    declare crs cursor for
      select val from collect0;
    declare continue handler
      for sqlstate '02000' set done = 1;
  drop table if exists exists0;
  create temporary table
    exists0(val bool);
  open crs;
  repeat
    fetch crs into var2;
    if not done then
      select val into tempResult
      from
        (select tbl5.val = tbl6.val as val
        from
          (select Person.surname as val
          from Person,
          (select var2 as val) as tbl4
          where pk = tbl4.val) as tbl5,
          (select 'Perez' as val)
          as tbl6) as tbl8;
      if tempResult then
        set done = 1;
        set result = True;
      end if;
    end if;
    until done end repeat;
    insert into exists0(val)
      (select result as val);
    close crs;
end;
select val from exists0;
end;
```

□

To conclude this section, we would like to remark, some general invariants in our mappings:

- nested operators, which require blocks definitions, are mapped into nested blocks, while sequential operators are mapped into sequential blocks.
- the results of expressions with simple types and sets are mapped into tables with a column called `val`; while expressions with sequence types are mapped into tables with two columns, one for the values (i.e. `val`) and the another for the positions (i.e. `pos`).
- when we talk about iterators, the statement:

```
declare crs cursor for (codegen_q(src));
```

defined when the *src*-collection is a sequence has the following format:

```
declare crs cursor for
  (select al(codegen_q(src)).val
  from (codegen_q(src))) as al(codegen_q(src))
  order by al(codegen_q(src)).pos;
```

## 6 The SQL-PL4OCL tool

The SQL-PL4OCL tool rewrites the tool introduced in [13] to target not just MySQL (or MariaDB) but also PostgreSQL and SQL Server DBMS. The new implementation does not comply to the mapping we introduced in [9,13] but to the one defined in section 5. Please, recall Remark 1 (Section 1) for a summary of the differences.

Essentially, SQL-PL4OCL is a code generator tool that using as input a data model (as specified in Section 3), a list of OCL queries, and a vendor identifier, it generates a set of statements ready to create the database with the tables that correspond to the data model (following the mapping introduced in Section 3), and a list of stored procedures (one per OCL query, following the definition specified in section 5). Figure 7 shows two screenshots of the tool interface. Of course, the resulting code is produced adapted to the syntax of each target RDBMS.

Figure 5 shows the main components of the tool architecture. These are:

- DM validator: This component checks whether the input data model fulfills the restrictions about well-formedness that we explain in Section 3 (Remark 2), so as to serve as a valid context for OCL queries.
- OCL validator: This component parses each OCL query of input in the context of the data model. Only if a query parses correctly (and our mapping covers it), it is used as input to produce code.
- DB engine selector: This component receives as input the vendor identifier so as the code generated is syntactically adapted to the selected RDBMS.
- DB model generator: This component generates the *engine-specific* statements to create the database and corresponding tables.
- SQL-PL generator: This component generates the *engine-specific* statements to create the SQL-PL stored procedures corresponding to the input OCL queries.

The complexity of supporting multiple RDBMS is brought by their implementation differences. Perhaps the most noticeable difference is the language they parse. Even though all engines use some flavor of SQL, these all differ in how variables, stored procedures, and built-in functions are declared in their procedural extensions. Also, PostgreSQL supports different procedural languages (we targeted at PL/pgSQL), MS SQL Server uses Transact SQL and MySQL uses yet another dialect (fully compatible with MariaDB's).

As implementation strategy, we avoided the burden of dealing with the subtleties of each SQL dialect within the mapping algorithm by defining a plugin-based architecture. In this architecture, each plugin component is responsible for performing the apropriate translation for the RDBMS it targets. In [32], the reader can find a comparison that gives idea of the variations among the differents SQL dialects. We encourage the interested reader to use our tool, which is available at [10], to investigate them.

### 6.1 A benchmark to explore the efficiency of the code generated

Figure 6 shows a benchmark to test the performance (in terms of the evaluation time) of a sample of OCL mapped queries into the different DBMS. In this sample, we included both simple expressions (Q1-Q7), and complex expressions (Q8-Q14), including iterator and sequence operators. All the expressions in the benchmark were evaluated on an artificial scenario that we created. The scenario is an instance of the Car-Company data model depicted in Figure 3. This instance contains $10^6$ instances of class Car, $10^5$ instances of class Person (all of them are Employees), and $10^2$ instances of class Company, where each company is associated to $10^2$ instances of Person, and each person owns 10 different cars. All car instances have a color different from black.

We used bold font to highlight the lowest evaluation time of each query in Figure 6. By just taking a look, it turns apparent that MariaDB, an open source database, achieves the fastest evaluation times for the majority

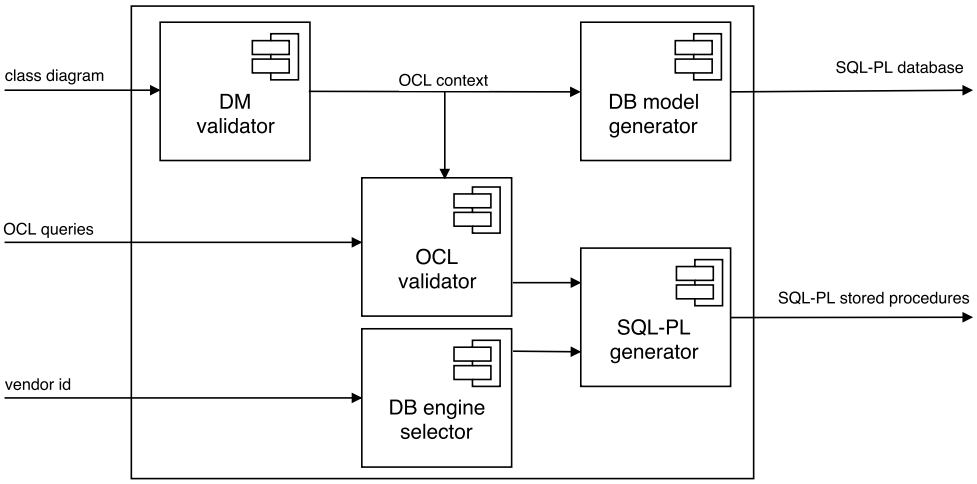



Fig. 5: SQL-PL*4*OCL tool component architecture

of the queries and, most importantly, for almost the totality of complex expressions.[7]

In our view, a dedicated experimentation would be needed in order to outline a function that may relate an OCL expression evaluation over an scenario with the time that the evaluation of the translated query takes over a database. Yet, we have identified three parameters which seem to correlate directly to the increase in the evaluation time of an expression translated by our mapping. More concretely,

i. The OCL expression contains access to attributes or association-ends. Their translation into left joins (of size n × m) makes them expensive in time. Also, the materialization of a left join performed between different tables (i.e., for translating an association, as in Q3 and Q7) is more expensive than one performed by a table with itself (i.e., for translating access to an attribute, as in Q2 and Q6). The time gets worse when the source table is larger, i.e., with a high $n$. For example, compare evaluation times for queries Q3 and Q4 where the size of the source collection is $10^6$ and $10^5$ (resp.), or queries Q2 and Q12 for which the size of the left join (owners.ownedCars) is $10^6 \times 10$ and $1 \times 10$ (resp.).
ii. The size of the outermost source collection in an OCL iterator expression (if there is no stop criterion applied). For example, to evaluate Q9 the cursor has to fetch values from a table of size $10^6$, however, to evaluate Q10 the cursor only fetches one value and the procedure stops. Notice also the different evaluation time between Q2 and Q11 (which are similar expressions in semantics) since the last is shaped as an iterator expression.
iii. The number of insertions to a table when this is required by the mapping to translate a query. In particular, insertions to a table are always required for evaluating sequence expressions. As an example we compare queries Q8 and Q9. The size of the source expression for both queries is the same ($10^6$). However, the evaluation of Q8 requires the insertion of intermediate values into a table while Q9 evaluation does not. Similarly happens with Q2 and Q13. The different evaluation time between Q8 and Q14 seems to be due to the generation of the autoincremented *position* value for the latter.

[7] We ran the benchmark in a laptop with an Intel Core m7, 1.3 GHz, 8 GB RAM, and 500 GB Flash Storage. The RDBMS versions used were MySQL 5.7, MariaDB 10.1, SQL Server 2016 Express, and PostgreSQL 9.6.1.

## 7 Related Work

The work [29] is concerned about the translation of OCL to SQL and viceversa. This translation supports only OCL class invariants and, partially, the operators forAll, select, and exists. Because of this clear limitation many of the problems discussed in previous sections are not considered. Another limited translation is presented in [30]. Its main result is the implementation of a solution that generates SQL code from OCL simple expressions as a part of Enterprise Architect. However, this solution cannot deal with OCL iterator expressions or sequences.

To the best of our knowledge the idea of mapping OCL iterators to stored procedures was first proposed in [28], however the idea was not fully developed:

| | Queries | MySQL | MariaDB | PostgreSQL | MSSQL |
|---|---|---|---|---|---|
| Q1 | `p1->size()` | 0.19s | 0.13s | **0.10s** | 0.12s |
| Q2 | `p1.model->size()` | 0.25s | **0.20s** | 0.33s | 0.28s |
| Q3 | `p1.owners->size()` | 0.36s | 0.35s | 0.27s | **0.26s** |
| Q4 | `Employee.allInstances().company->size()` | **0.04s** | **0.04s** | **0.04s** | 0.05s |
| Q5 | `p1.owners.name->size()` | 0.55s | **0.40s** | **0.40s** | 0.42s |
| Q6 | `p1.owners->oclAsType(Employee).salary->size()` | 1.05s | **0.55s** | 1.06s | 1.03s |
| Q7 | `p1.owners->oclAsType(Employee).ownedCars->size()` | 2.07s | **1.56s** | 1.99s | 2.08s |
| Q8 | `p1->select(c\|c.color<>"black")->size()` | 50.02s | **43.08s** | 57.04s | 53.47s |
| Q9 | `p1->forAll(c\|c.color<>"black")` | 9.14s | **8.00s** | 8.18s | 8.89s |
| Q10 | `p1->exists(c\|c.color<>'black")` | 0.05s | **0.04s** | 0.07s | 0.05s |
| Q11 | `p1->collect(x\|x.color)->size()` | 49.56s | **40.02s** | 40.10s | 43.46s |
| Q12 | `p1->collect(x\|x.owners.ownedCars)->size()` | 59.58s | **51.23s** | 51.25s | 54.82s |
| Q13 | `p1.model->asSequence()->size()` | **1.67s** | 1.98s | 2.35s | 1.90s |
| Q14 | `p1->asSequence()->select(c\|c.color<>"black")->size()` | 59.52s | **54.33s** | 63.35s | 58.33s |

where: $p1$ = `Car.allInstances()`

Fig. 6: Evaluation times.

*'Das Ergebnis des hier vorgestellten Abbildungsmusters kann für einen Teilausdruck nicht direkt in das Abbildungsergebnis eines anderen Teilausdrucks eingesetzt werden. Die Kombinationstechnik wird nicht formal beschrieben.' [28, pag.59]* [...]
*'Es ist in dieser Arbeit nicht gelungen, eine übersichtliche und vollständig formale Darstellung für die prozeduralen Abbildungsmuster zu finden.' [28, pag.112]* [8]

Neither have we found any other development of it afterwards. Since we are concerned with query evaluation, it is cruzial for the mapping to preserve evaluation semantics, in particular for navigation expressions. For instance, the expression p.ownedCars.owners where p is an object, returns a bag where some elements may be repeated. However, the translation proposed in [28,11] removes duplicates because it relies on the SQL `in` operator. Thus, to preserve the evaluation semantics of navigations expressions we decided to employ SQL *left joins* instead of the `in` operator.

There are other much less relevant differences between both mappings that we do not treat here due to space limitations. In [12] they present the architecture of the Dresden OCL2SQL tool where they introduce `views` to hold those elements which do not fulfill a constraint mapped using the patterns in [28,11]. They also propose some pattern refinements to ease the implementation and tailor the results for different DBMS. In [15] they propose a novel architecture for a query code generation framework where different transformation patterns, e.g., OCL2SQL or OCL2XQuery could be integrated. The patterns to perform the mapping OCL2SQL are those already reviewed. In [26] they model geographical information systems with UML and OCL but they also propose an extension to the OCL type system to represent some basic geometric elements. Their final aim is to implement the modeled systems and to evaluate their constraints in the relational database which contains the actual spatial data so they intend to study whether the tool OCL2SQL would fit their needs. Most of the OCL constraints handled in this work contain iterators so, in principle, we could cover their generation. However, we need to study further how well we could deal with their extension to the OCL type system. In [1] they explore a model transformation approach from UML to CWM [21] and from OCL to a patterns metamodel. This is a feasible approach but as far as we know, it did not have further development. In any case, we do not use model transformations as the mapping technique.

The work in [5] introduces a different strategy for query translation to ours. Instead of a compile time translation, they propose a runtime query translation from model level languages like EOL, to persistent query languages like SQL. Each EOL query is splitted up into subexpresions that are handled by the apropriate implementation classes. We expect to obtain an interesting comparison when this runtime implementation strategy is applied to translate OCL to SQL.

In [4] authors explore how participation constraints defined on binary associations, e.g. 'xor' constraint, can be expressed at two different levels, in OCL as a constraint language, and as SQL triggers. No mapping from OCL to SQL expressions is proposed.

In [7] the author proposes OCL transformations rules to SQL standard for some simple OCL expressions.

[8] *'The result of the mapping model presented here may not apply a part of the expression directly into the result of another subexpression.The combination technique is not formally described.'*
[...]
*'This work did not succeed to find a concise and complete formal representation for procedural mapping patterns.'*

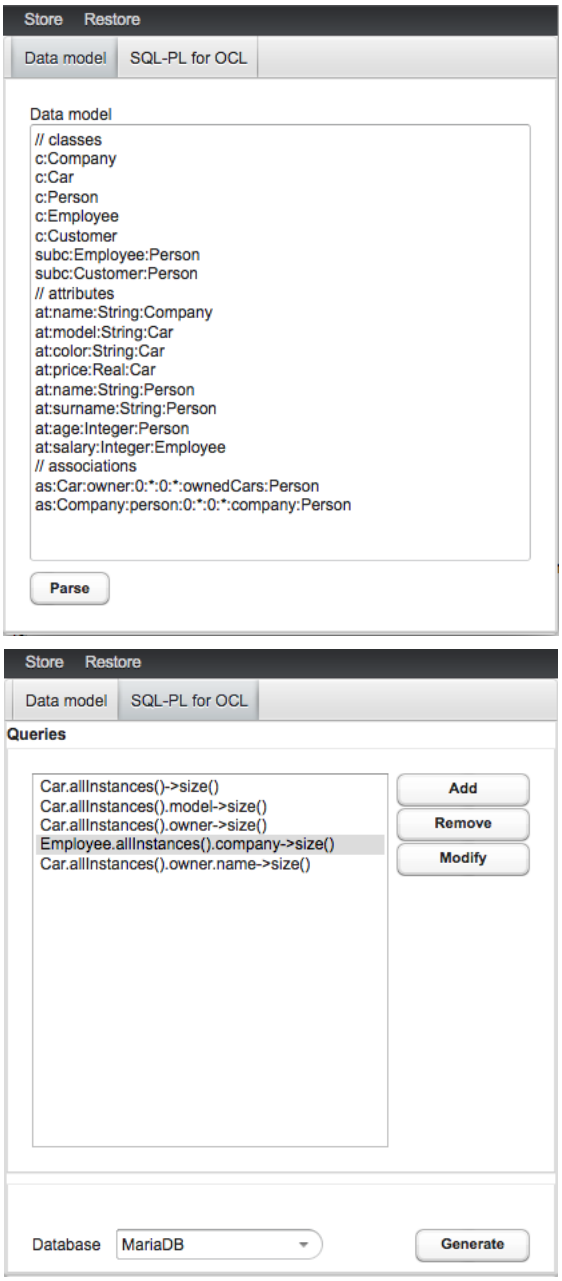



Fig. 7: SQL-PL*4*OCL tool: screenshots

However, complex expressions [9] are not covered, neither the recursive nature of the OCL language. We could not test their tool since it does not seem to be publically available.

In [25] the authors propose an approach to reduce the problem of the satisfiability of an OCL constraint to check the emptiness of some SQL query in an RDBMS. This evaluation can be performed incrementally if some update is applied to the data stored. In their paper, the mapping from OCL constraints does not directly target SQL. In contrast, it translates OCL to a logic called Event-Dependency Constraints (EDC). From EDC they generate SQL statements with a pattern-based approach. The coverage of OCL supported by their mapping is not detailed. However, they report about an experiment using four examples for which they reach lower evaluation times in SQL than the times returned by evaluating the code produced with MySQL4OCL. In our view, a detailed comparison in terms of efficiency and in terms of OCL language coverage is needed.

The work in [8] is motivated by the concern of expressing database integrity constraints as business rules in a more abstract language. In the process of business rules identification, it describes the mapping between SQL SELECT statements, certain type of PL blocks and the equivalent OCL expressions. Although very interesting, this mapping that is based in the structure of SQL expressions, is focused in covering the mapping for SQL projections, joins, conditions, functions, group by and having clauses. To the best of our knowledge this is the only work dealing with the translation from SQL to OCL up to date.

## 8 Conclusions and Future Work

In this work we have detailed a novel mapping from OCL expressions to SQL-PL stored procedures. The seminal work of our mapping was introduced in [13,9]. However, the definition provided here is improved with respect to the previous one, being the most remarkable differences the following: 1) each OCL expression (no matter its complexity) is mapped to just one stored procedure that is executed by just one call-statement; 2) we employ temporary tables in the stored procedures which help improve evaluation time of resulting code; 3) we consider the three-valued evaluation semantics of OCL. Moreover, while our original work met only the procedural extension of MySQL, our new definition has eased the implementation task and we managed to target several relational database management systems, both open source and proprietary. This fact allowed us to compare the evaluation time of the resulting code into the different RDBMS. Finally, we implemented and made available our SQL-PL*4*OCL tool at [10].

Since OCL is a language created to be used at design time of the software engineering lifecycle, we would like, as a matter of primary objective for future work, to integrate our code generator with CASE (Computer-Aided Software Engineering) tools which support design of systems. As part of this work we will extend our mapping to cover Aggregation and Composition

[9] Notice that here we employ the terminology 'simple expressions' and 'complex expressions' following our definition in Section 4.

relationships which are frequently used by software architects and developers to indicate a part-whole relationship. Both types of relationships are binary associations. Since the semantics of Aggregation (i.e., shared aggregation) varies by application area and modeler [24, page 110], we will study its mapping case by case. However, since the semantics of Composition (i.e., composite aggregation) states that the composite object has the responsibility for the existence and storage of the composed objects [24, page 110], Composition will be mapped as a one-to-many relationship (as we explained in section 3.3). Moreover, when the instance at the 'one' side of the Composition is removed, all instances linked to it through this relation will also be removed.

Regarding evaluation times, we would like to implement a lazy evaluation strategy for our SQL-PL4OCL tool to optimize OCL expressions' evaluation times, as we identified in [6]. Nevertheless, without using a lazy strategy we have improved resulting evaluation times with respect to previous versions of our mapping [13].

Of course, as a priority in our roadmap is removing current limitations of our mapping.

Also, we noticed that very few works deal with the translation of SQL to OCL. The lessons learned by defining the presented mapping appear to us as a good starting point to address the backwards traceability from SQL to OCL.

Other interesting future lines of work are, on the one hand, adapting our mapping to mobile embedded databases, i.e., SQLite. On the other hand is to study the feasibility of mapping OCL to NoSQL databases. Yet, we are aware of the difficulty of the mapping definition and the implementation efforts from one NoSQL database to another, since they lack of standarization.

Model specific operations

| | |
|---|---|
| allInstances | projection over column pk |
| AssociationEnds | left join |
| Attribute | left join |

Constants Operations

| | |
|---|---|
| Boolean Literal | - |
| Null Literal | - |
| Integer Literal | - |
| Real Literal | - |
| String Literal | cast |
| Set/Bag Literal | union all |
| OrderedSet/Sequence Literal | temp table + creation of the index |

Any Operations

| | |
|---|---|
| =, <> | Depending on the specific type, apply the map for collections or primitive types. Other cases are not supported |
| oclIsUndefined | = null |
| oclAsType | left join |
| oclIsTypeOf | left join |
| oclIsKindOf | left joins |
| oclType | temp table + creation of all values |

Iterators

| | |
|---|---|
| closure, exists, forAll, isUnique, any, one, collect, select, reject, sortedBy | Cursor over source, create temp table and save result on it. Data saved into the table depend on the definition of each iterator. |

Table 1: Guideline of the mapping of OCL Model Specific, Literal, Any and Iterators to SQL-PL

## A Guideline to implementation

This annex is intended to provide a high-level overview, abstracting away the details, of how OCL operators are translated to SQL and its procedural extension. This overview is presented in Tables 2, 3, and 1.

## References


1. A. Armonas and L. Nemuraité. Pattern Based Generation of Full-Fledged Relational Schemas From UML/OCL Models. *Information Technology and Control*, 35(1), 2006.
2. D. Basin, M. Clavel, M. Egea, M. A. G. de Dios, C. Dania, G. Ortiz, and J. Valdazo. Model-driven development of security-aware guis for data-centric applications. In *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, volume 6858 of *LNCS*, pages 101–124. Springer, 2011.
3. D. A. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information & Software Technology*, 51(5):815–831, 2009.
4. D. Berrabah and F. Boufarès. Constraints checking in UML class diagrams: SQL vs OCL. In R. Wagner, N. Revell, and G. Pernul, editors, *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4653 of *LNCS*, pages 593–602. Springer, 2007.
5. X. D. Carlos, G. Sagardui, and S. Trujillo. MQT, an approach for run-time query translation: From EOL to SQL. In A. D. Brucker, C. Dania, G. Georg, and M. Gogolla, editors, *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, volume 1285 of *CEUR Workshop Proceedings*, pages 13–22. CEUR-WS.org, 2014.
6. M. Clavel, M. Egea, and M. A. G. de Dios. ECEASST building an efficient component for OCL evaluation. *ECEASST*, 15, 2008.
7. S. C. Cortázar. Transformación de las restricciones OCL de un esquema UML a consultas de SQL. trabajo de fin de grado. Technical report, Universidad Carlos III de Madrid, 2012. `http://e-archivo.uc3m.es/bitstream/handle/10016/16799/TFG_Sergio_Casillas_Cortazar.pdf?sequence=1&isAllowed=y`.

| Real Operations | |
|---|---|
| +, -, *, /, -(unary) | +, -, *, / -unary |
| abs, floor, round | abs, floor, round |
| max, min | max, min |
| <, >, <=, >=, = | <, >, <=, >=, = |
| toString | cast |
| **Integer Operations** | |
| -:Integer | |
| +, -, *, / | +, -, *, / |
| div | case analysis + / |
| abs, mod | abs, mod |
| max, min | max, min |
| size | count |
| count | count |
| toString | cast |
| **String Operations** | |
| +, concat | concat |
| size | len |
| substring | instr |
| toInteger | add 0 |
| toReal | add 0.0 |
| toUpperCase. toLowerCase | upper, lower |
| indexOf | instr |
| equalsIgnoreCase | = |
| at | substr |
| characters | temp table + creation index |
| toBoolean | = 'True' |
| <, >, <=, >= | <, >, <=, >= |
| **Boolean Operations** | |
| or, and, not | or, and, not |
| xor, implies | or + not |
| toString | cast |
| =, <> | and + not + or |
| includes, excludes | in, not in |
| includesAll, excludesAll | count + in, count, not in |
| isEmpty | count =0 |
| notEmpty | count ≠0 |
| asSet, asBag | - |
| asOrderedSet, asSequence | temp table + creation index |

Table 2: Guideline of the mapping of OCL Real, Integer and Strings operators to SQL-PL


8. V. Cosentino. *A model-based approach for extracting business rules out of legacy information systems.* PhD thesis, École des mines de Nantes, France, 2013.
9. C. Dania. MySQL4OCL: Un compilador de OCL a MySQL, 2011. Master thesis. Universidad Complutense de Madrid.
10. C. Dania and M. Egea. SQLPL4OCL tool, 2016. `http://software.imdea.org/~dania/tools/sqlpl4ocl`.
11. 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.
12. 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.
13. M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *ECEASST*, 36, 2010.
14. I. O. for Standardization. Object Management Group Object Constraint Language (OCL), 2012. `https://www.iso.org/obp/ui/\#iso:std:iso-iec:19507:ed-1:v1:en`.
15. F. Heidenreich, C. Wende, and B. Demuth. A Framework for Generating Query Language Code from OCL Invariants. *ECEASST*, 9, 2008.
16. A. Kleppe, W. Bast, J. B. Warmer, and A. Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.
17. MariaDB, 2016. `https://mariadb.org/`.
18. Microsoft. SQL Server, 2016. `https://www.microsoft.com/es-es/server-cloud/products/sql-server/overview.aspx`.
19. M. Monperrus, J. Jézéquel, B. Baudry, J. Champeau, and B. Hoeltzener. Model-driven generative development of measurement software. *Software and System Modeling*, 10(4):537–552, 2011.
20. MySQL 5.7 Reference Manual. `http://dev.mysql.com/doc/refman/5.7/`.
21. Object Management Group. *Common Warehouse Metamodel specification*, March 2003. OMG document available at `http://www.omg.org/technology/documents/formal/cwm.htm`.
22. Object Management Group. *Object Constraint Language specification*, May 2006. OMG document available at `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.
23. Object Management Group. Object constraint language specification version 2.4. Technical report, OMG, 2014. `http://www.omg.org/spec/OCL/2.4`.
24. Object Management Group. Unified modeling language version 2.5. Technical report, OMG, 2015. `http://www.omg.org/spec/UML/2.5/PDF/`.
25. 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, *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, volume 1285 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org, 2014.
26. F. Pinet, M. Kang, and F. Vigier. Spatial Constraint Modelling with a GIS Extension of UML and OCL: Application to Agricultural Information Systems. In U. K. Wiil, editor, *Metainformatics*, volume 3511 of *LNCS*, pages 160–178. Springer, 2004.
27. PL/pgSQL - SQL procedural language, 2016. `https://www.postgresql.org/docs/9.2/static/plpgsql.html`.
28. A. Schmidt. Untersuchungen zur Abbildung von OCL-ausdrücken auf SQL. Master's thesis, Institut für Softwaretechnik II - Technische Universität Dresden, Germany, 1998.
29. N. Siripornpanit and S. Lekcharoen. An adaptive algorithms translating and back-translating of object constraint language into structure query language. In *International Conference on Information and Multimedia Technology, 2009. ICIMT'09.*, pages 149–151. IEEE, 2009.
30. P. Sobotka. Transformation from OCL into SQL, 2012. Master thesis. Charles University in Prague. `https://is.cuni.cz/webapps/zzp/download/120076745`.
31. ISO/IEC 9075-(1–10) Information technology – Database languages – SQL. Technical report, International Organization for Standardization, 2011. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=63555`.
32. SQL Dialects Reference, 2016. `https://en.wikibooks.org/wiki/SQL_Dialects_Reference/Print_version`.

## About the Authors

Marina Egea is the Head of the Tiger Team, Cybersecurity Operations at Minsait (by Indra). The Tiger Team areas of action are Ethical Hacking/Pentesting, Rapid Response to Incidents, Software Security Assurance, Digital Forensics, Malware Analysis and R&D&i projects. She got her PhD in Computer Science at Universidad Complutense de Madrid in 2008. Then, she moved for a posdoctoral position in the Information Security Group at ETH Zurich (Switzerland) and, later, to IMDEA Software (Spain). After being involved in the proposal and technical coordination of more than six European projects (FP7 and H2020 framework programmes), she joined Indra company at the beginning of 2015. Currently, she leads the technical and business development of two European Innovation projects funded by EIT Digital and the European Defence Agency (resp.). She is an Associate Professor for 'Security Engineering' at Universidad Carlos III de Madrid and she also gives lectures on 'Secure Software Development' at Master Indra in Cybersecurity. She takes advantage of every opportunity of R&D&i that may help to improve industrial practice.

Carolina Dania is a Ph.D student at IMDEA Software (Madrid, Spain). She received her bachelor's degree from Universidad Nacional de Cordoba (Argentina), and her masters degree from Universidad Complutense de Madrid (Spain). Her research interests include software engineering, formal methods and security. In particular, she is working in tools and techniques for modeling, building and validating secure and reliable software systems.

Collection Operations

| | |
|---|---|
| union | union |
| union (between bags) | union all |
| = (between sets) | sort the elements and check one by one that elements are equals |
| intersection (between sets) | check the elements one by one and save in temp table the minimum size of the common ones |
| - | not in |
| including | union |
| excluding | not in |
| symmetricDifference | union all + not in |
| count | count |
| asBag | project over column val |
| asSequence, asOrderedSet | temp table + creation of index |
| asSet | project over column val using distinct |

Bag Operations

| | |
|---|---|
| = | sort and check elements one by one |
| union | union all |
| intersection (between sets or bags) | check elements one by one and add only one value if it exists in both collections |
| intersection (between sets) | check elements one by one and make an union of the minimum of the values |
| including | union |
| excluding | not in |
| count | count |
| asBag | - |
| asSet | distinct |
| asOrderedSet, asSequence | temp table + creation of the index |

Set Operations ≈ Bag operations

Sequence Operations

| | |
|---|---|
| count | count |
| = | idem than bag, but ordering by pos |
| union, append, prepend, insertAt | idem than bags, but creating a new index |
| subOrderedSet | projection with restriccions over pos. Recreation of pos |
| at | projection over val with restriction |
| indexOf | projection over val, pos is equals to the given |
| first | projection over val, pos = min position |
| last | projection over val, pos = max position |
| including, excluding | adding/removing element and change of the index for the rest |
| reverse | sorted by pos desc and changing index |
| sum | sum |
| asBag | projection over column val |
| asSequence, asOrderedSet | - |
| asSet | projection over column val (using distinct) |

Table 3: Guideline of the mapping of OCL Collection operators to SQL-PL