Electronic Communications of the EASST
Volume 36 (2010)

Proceedings of the
Workshop on OCL and Textual Modelling
(OCL 2010)

MySQL*4*OCL: A Stored Procedure-Based MySQL Code Generator for OCL

Marina Egea, Carolina Dania and Manuel Clavel

16 pages

Guest Editors: Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

MySQL*4*OCL: A Stored Procedure-Based MySQL Code Generator for OCL

Marina Egea[1], **Carolina Dania**[12] **and Manuel Clavel**[12*]

IMDEA Software Institute, Madrid, Spain.
[marina.egea,carolina.dania,manuel.clavel]@imdea.org[1]

Depto. Sistemas Informáticos y Computación[2],
Universidad Complutense de Madrid, Spain.

Abstract: In this paper we introduce a MySQL code generator for a significant subset of OCL expressions which is based on the use of stored procedures for mapping OCL iterators. Our code generator is defined recursively over the structure of OCL expressions. We discuss the class of OCL expressions covered by our definition (which includes, possibly nested, iterator expressions) as well as some extensions needed to cover the full OCL language. We also discuss the efficiency of the MySQL code produced by our code generator, and compare it with previous known results on evaluating OCL expressions on medium-large scenarios. We have implemented our code generator in the MySQL*4*OCL tool.

Keywords: OCL, code generator, stored-procedures, MySQL

1 Introduction

In this paper we introduce a MySQL code generator for a significant subset of OCL expressions which is based on the use of stored procedures for mapping OCL iterators. We have implemented this code generator in the MySQL*4*OCL tool, which is available at [EDC10].

Motivation. Our motivation here is two-fold. On the one hand, our code generator addresses the problem, already discussed in [CED08] of evaluating OCL expressions on really large scenarios: instead of having to "load" them in memory (a time-consuming, or even impractical task for OCL evaluators), we can (i) store these scenarios in a database, (ii) apply our code generator to the expressions to be evaluated, and (iii) execute the resulting query statements on the database. On the other hand, our code generator provides a key component for any models-to-code development process, where the source models include OCL expressions and the target code evaluates these expressions on relational databases. This is precisely the case of our models-to-code toolchain for developing smart, security-aware GUIs [DDS^{+}10].

OCL to MySQL in a Nutshell. In general, given an expression *expr*, our code generator produces a MySQL query, denoted by *codegen*(*expr*), whose execution returns a result-set containing the values corresponding to the evaluation of *expr*. The case of *iterator expressions*, i.e., of

* Research partially supported by EU Program PCOFUND-2008-229599, Spanish MEC Projects TIN2009-14599-C03-01 and TIN2009-14599-C03-02, and by Comunidad de Madrid Program S2009/TIC-1465.

expressions whose top-operator is an iterator operator, deserves, however, a particular attention. Basically, for each iterator expression *expr*, our code generator produces a MySQL *stored procedure*[1] that, when called, it creates a table containing the values corresponding to the evaluation of *expr*.

Our code generator is defined recursively over the structure of OCL expressions. That is, let *expr* be an expression with top-operator *op* and immediate subexpressions $expr_1,\ldots,expr_n$. Thus, in general, $codegen(expr)$ has as subqueries $codegen(expr_1),\ldots,codegen(expr_n)$ (possibly with aliases, that we denote by $\overline{codegen(expr_1)},\ldots,\overline{codegen(expr_n)}$), which are produced by recursively calling our code generator on the corresponding subexpressions of *expr*.

Scope of our Mapping. As presented here, our code generator does not cover the full OCL language, a non-realistic task anyway given the limited space available. Among the features of OCL which are not currently covered are the followings: operations on sequences and ordered-sets; operations on collection of collections; operations on types; and user-defined operations. In our concluding remarks we briefly discussed how to extend our mapping to overcome its current limitations.

However, the subset of OCL that we do cover here is significant in two ways. First, it allows us to apply our code generator to (and experiment with) a large class of OCL queries: we have included some examples in Appendix A and many more are available at [EDC10]. Second, it allows us to introduce the key idea underlying our mapping, namely, the use of stored procedures to deal with (possibly nested) iterator expressions. Interestingly, this idea was already hinted in [Sch98]. However, as it is recognized there, "this work did not succeed in finding a concise and complete formal representation for procedural mapping patterns", and it was not further developed afterwards.

Organization. The rest of this paper is organized as follows. In Section 2 we recall the basic mapping from UML class and object diagrams to MySQL. Then, in Section 3, Section 4, and Section 5, we give the definition of the mapping underlying our code generator. Afterwards, in Section 6, we discuss the efficiency of the MySQL code produced by our code generator and, in Section 7, we report on related work. We conclude the paper with our future work.

2 Mapping UML Class and Object Diagrams to MySQL

Our code generator assumes that the underlying UML class and object diagrams (i.e., the "contexts" of the OCL queries) are represented using MySQL tables according to the following (rather) standard rules. Examples can be found in Appendix A. For the sake of simplicity, we do not consider here the full class of UML class diagrams. In particular, we do not consider the following modeling elements: generalizations, enumerations, multivalued attributes, n-ary associations (with $n > 2$), and ordered or qualified associations.

[1] Stored procedures are routines (like a subprogram in a regular computing language) that are stored in the database. A stored procedure has a name, may have a parameter list, and an SQL statement, which can contain many other SQL statements. Stored procedures provide a special syntax for local variables, error handling, loop control, if-conditions and cursors, which allow the definition of iterative structures.

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

- Each class A in M is mapped to a table $\lceil A \rceil$, which contains, by default, a column `pk` of type `Int` as its primary key. Then, each object in O of class A is represented by a row in $\lceil A \rceil$ and it will be denoted by its key (i.e., the value in `pk`).
- Given a class A, each attribute W of A is mapped to a column $W^\sharp$ in $\lceil A \rceil$, the type of $W^\sharp$ being the type of W.[2] Then, the value of W in an object o in O is represented by the value which the column $W^\sharp$ holds for the row that represents o in $\lceil A \rceil$.
- Given two classes A and B, each 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 table $\lceil P \rceil$, which contains two columns $rl\_A^\sharp$ and $rl\_B^\sharp$, 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 $\lceil P \rceil$, where $rl\_A^\sharp$ holds the key denoting o and $rl\_B^\sharp$ holds the key denoting o'.

The above mapping rules assumes that the input UML class diagrams satisfy the following (rather) natural constraints:

- Each class has a unique name.
- Each attribute within a class has a unique name.
- Each association is uniquely characterized by its association-ends. Also, the association-ends in a self-association have different names.

3 Mapping Non-Iterator Expressions to MySQL Queries

In this section we define the query $codegen(expr)$ produced by our code generator for the case of expressions $expr$ whose top-operator is a non-iterator operator. Examples can be found in Appendix A.

We proceed by cases, each case being characterized by the expression's top-operator. Due to space constraints, we only consider a subset of the OCL non-iterator operators. Also, we assume that the types of these operators are either primitive types, sets or bags of primitive or class types, or class types. For the sake of presentation, we assume that, in all cases, the immediate subexpressions of $expr$ are non-iterator expressions. The remaining cases are dealt with as explained in Section 5.

Primitive operators

Variables. Let *var* be a variable. Then, $codegen(var)$ is:

```
select var as value
```

[2] More specifically, the UML/OCL primitive types `Boolean`, `Integer`, and `String`, are mapped, respectively, to the MySQL types `Int`, `Int`, and `char(65)`. The UML/OCL class types are mapped to the MySQL type `Int` (which is the type of the primary keys of the tables representing classes). We have decided to map `String` to `char(65)` for efficiency reasons: obviously, a more general solution is to map `String` to `text`.

Literals. Let *lit* be a primitive literal. Then, $codegen(lit)$ is:

`select` *lit* `as value`

Operators. Let *expr* be an expression or the form $expr_1$ *prim_op* $expr_2$, where *prim_op* is $+$, $-$, `x`, `and`, `or`, `implies`, or $=$. Then, $codegen(expr)$ is:

`select` $(codegen(expr_1))\ prim\_op^\flat\ (codegen(expr_2))$

where $prim\_op^\flat$ denotes the operation in MySQL that corresponds to *prim_op*.[3] Similarly, for expressions of the form *prim_op* $expr_1$, where *prim_op* is $-$ or `not`.

Type literals. Let *class* be a class identifier. Then, $codegen(class)$ is:

`select` *pk* `as value from` $\lceil class \rceil$

Boolean operators

`isEmpty/notEmpty`. Let *expr* be an expression of the form $expr_1$`->isEmpty()`. Then, $codegen(expr)$ is:

`select count(*)=0 as value from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$

For the operator `notEmpty`, “`count(*)>0`” replaces “`count(*)=0`” above.

`includes/excludes`. Let *expr* be an expression of the form $expr_1$`->includes(`$expr_2$`)`. Then, $codegen(expr)$ is:

`select` $codegen(expr_2)$ `in` $codegen(expr_1)$ `as value`

For the operator `excludes`, “`not in`” replaces “`in`” above.

Numeric operators

`size`. Let *expr* be an expression of the form $expr_1$`->size()`. Then, $codegen(expr)$ is:

`select count(*) as value from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$

`sum`. Let *expr* be an expression of the form $expr_1$`->sum()`. Then, $codegen(expr)$ is:

`select sum(*) as value from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$

[3] In general, since there are primitive operators in OCL that do not have a direct counterpart in MySQL (e.g., `implies`), some expressions may need to be rewritten into equivalent ones before calling the code generator.

Model specific operators

`allInstances`. Let *expr* be an expression of the form $expr_1$`.allInstances()`. Then, $codegen(expr)$ is:

`select * from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$

Association-ends. Let *expr* be an expression of the form $expr_1 \,.\, rl\_A$ (resp. $expr_1 \,.\, rl\_B$), where rl_A (resp. rl_B) is the A-end (resp. B-end) of an association P between two classes A and B. Then, $codegen(expr)$ is:

`select` $\lceil P \rceil$`.`$rl\_A^{\sharp}$ `as value`
  `from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$
  `left join` $\lceil P \rceil$ `on` $\overline{codegen(expr_1)}$`.value =` $\lceil P \rceil$`.`rl_B
  `where` $\lceil P \rceil$`.`$rl\_A^{\sharp}$ `is not null`

Attributes. Let *expr* be an expression of the form $expr_1 \,.\, attr$ where *attr* is an attribute of a class A. Then, $codegen(expr)$ is:

`select` $\lceil A \rceil$`.`$attr^{\sharp}$ `as value`
  `from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$
  `left join` $\lceil A \rceil$ `on` $\overline{codegen(expr_1)}$ `.value =` $\lceil A \rceil$`.pk`

Collection operators

`asSet`. Let *expr* be an expression of the form $expr_1$`->asSet()`. Then, $codegen(expr)$ is the following MySQL statement:

`select distinct * from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$

`asBag`. Let *expr* be an expression of the form $expr_1$`->asBag()`. Then, $codegen(expr)$ is:

$codegen(expr_1)$

`union`. Let *expr* be an expression of the form $expr_1$`->union(`$expr_2$`)`, where both $expr_1$ and $expr_2$ are sets. Then, $codegen(expr)$ is:

$codegen(expr_2)$ `union` $codegen(expr_1)$

When $expr_1$ or $expr_2$ are bags, then "`union all`" will replace "`union`" above.

`including`. Let *expr* be an expression of the form $expr_1$`->including(`$expr_2$`)`, where $expr_1$ is a set. Then, $codegen(expr)$ is:

$codegen(expr_2)$ `union` $codegen(expr_1)$

When $expr_1$ is a bag, then "`union all`" will replace "`union`" above.

`excluding`. Let *expr* be an expression of the form $expr_1$`->excluding(`$expr_2$`)`. Then, $codegen(expr)$ is:

`select * from` $codegen(expr_1)$ `as` $\overline{codegen(expr_1)}$
 `where value not in` $codegen(expr_2)$

4 Mapping Iterator Expressions to MySQL Procedures

In this section we define the query $codegen(expr)$ produced by our code generator for the case of expressions *expr* whose top-operator is an iterator. Examples can be found in Appendix A.

Here, the basic idea is that, for each iterator expression *expr*, our code generator produces a MySQL stored procedure, denoted by $codegen_{proc}(expr)$, that, when called, it creates a table, denoted by $\lceil codegen_{proc}(expr)\rceil$, containing the values corresponding to the evaluation of *expr*. Due to space constraints, we only consider a subset of the iterator operators in OCL: namely, `forAll`, `exists`, `collect`, `select`, and `reject`. Also, we assume that the types of the *source*-subexpressions are either sets or bags of primitive or class types, and that, in the case of `collect`-expressions, the types of their *body*-subexpressions are either primitive or class types, or set or bags of primitive or class types. For the sake of presentation, we assume that, in all cases, the *source*- and *body*-subexpressions of *expr* are not themselves iterator expressions. The others cases (which include the cases of nested iterators) are dealt with as explained in Section 5.

Let *expr* be an iterator expression of the form *source*`->`*iter_op*`(`*var* | *body*`)`. Then, $codegen(expr)$ is:

`call` $codegen_{proc}(expr)$`;`
`select * from` $\lceil codegen_{proc}(codegen(expr))\rceil$`;`

where $codegen_{proc}(expr)$ is a MySQL stored procedure. The definition of this procedure, also generated by our code generator, follows the scheme shown in Figure 1. Basically, the function $codegen_{proc}(expr)$ creates the table $\lceil codegen_{proc}(expr)\rceil$ and execute, for each element in the *source*-collection, the *body* of the iterator expression *expr*. More concretely, until all elements in the *source*-collection have been considered, $codegen_{proc}(expr)$ repeats the following process: i) it instantiates the iterator variable *var* in the *body*-subexpression, each time with a different element of the *source*-collection, which it fetches from $codegen(source)$ using a cursor; and ii) using the so called "iterator-specific processing code", it processes in $\lceil codegen_{proc}(expr)\rceil$ the result of the query $codegen(body)$, according to the semantics of the iterator *iter_op*. Additionally, in the case of the iterators `forAll` and `exists`, the table $\lceil codegen_{proc}(expr)\rceil$ is initialized, using the so called "initialization-specific code". Moreover, for the iterators `forAll` and `exists`, the process described above will also be finished when, for any element in the *source*-collection, the result of the query $codegen(body)$ contains the value corresponding, in the case of the iterator `forAll`, to false (i.e., `0`) or, in the case of the iterator `exists`, to true (i.e., `1`).

In the remaining of this section, we specify, for each case of iterator expression, the corresponding "value-specific type", "initialization-specific code" and "iterator-specific processing code" produced by our code generator when instantiating the general schema. For all cases, the "cursor-specific type" is the MySQL type which represents, according to our mapping (see footnote 2), the type of the elements in the *source*.

```
create procedure codegen_proc(expr) ()
begin
declare done int default 0;
declare var [cursor-specific type];
declare crs cursor for codegen(source);
declare continue handler for sqlstate '02000' set done = 1;
drop table if exists ⌈codegen_proc(expr)⌉;
create table ⌈codegen_proc(expr)⌉ (value [value-specific type]);
[Initialization-specific code (only for forAll and exists)]
open crs;
  repeat
  fetch crs into var;
  if not done then
  [Iterator-specific processing code]
  end if;
  until done end repeat;
close crs;
end;
```

Figure 1: General schema for mapping iterator expressions as stored procedures.

`forAll`-iterator. Let *expr* be an expression of the form *source*`->forAll(`*var* | *body*`)`. Then, the "holes" in the scheme shown in Figure 1 will be filled as follows:

- *value-specific type:* `int`.
- *Initialization code:*

  ```
  insert into ⌈codegen_proc(expr)⌉ (value) values (1);
  ```

- *Iterator-processing code:*

  ```
  update ⌈codegen_proc(expr)⌉ set value = 0 where (codegen(body)) = 0;
  if exists (select 1 from ⌈codegen_proc(expr)⌉ where value = 0)
  then set done = 1;
  end if;
  ```

`exists`-iterator. Let *expr* be an expression of the form *source*`->exists(`*var* | *body*`)`. Then, the "holes" in the scheme shown in Figure 1 will be filled as follows:

- *value-specific type:* `int`.
- *Initialization code:*

```
insert into $\lceil codegen_{proc}(expr)\rceil$ (value) values (0);
```

- *Iterator-processing code:*

  ```
  update $\lceil codegen_{proc}(expr)\rceil$ set value = 1 where ($codegen(body)$) = 1;
  if exists (select 1 from $\lceil codegen_{proc}(expr)\rceil$ where value = 1)
  then set done = 1;
  end if;
  ```

`collect`-iterator. Let *expr* be an expression of the form *source*`->collect(`*var* | *body*`)`. Then, the "holes" in the scheme shown in Figure 1 will be filled as follows:

- *value-specific type:* the MySQL type which represents, according to our mapping, the type of the *body*.

- *Iterator-processing code:*

  ```
  insert into $\lceil codegen_{proc}(expr)\rceil$ (value) $codegen(body)$;
  ```

`select`-iterator. Let *expr* be an expression of the form *source*`->select(`*var* | *body*`)`. Then, the "holes" in the scheme shown in Figure 1 will be filled as follows:

- *value-specific type:* the MySQL type which represents, according to our mapping, the type of the elements in the *source*.

- *Iterator-processing code:*

  ```
  if exists
    (select 1 from ($codegen(body)$) as $\overline{codegen(body)}$
      where value = 1)
  then insert into $\lceil codegen_{proc}(expr)\rceil$ (value) values ($var$);
  end if;
  ```

`reject`-iterator. Let *expr* be an expression of the form *source*`->reject(`*var* | *body*`)`. Then, the "holes" in the scheme shown in Figure 1 will be filled as follows:

- *value-specific type:* the MySQL type which represents, according to our mapping, the type of the elements in the *source*.

- *Iterator-processing code:*

  ```
  if exists
    (select 1 from ($codegen(body)$) as $\overline{codegen(body)}$
      where value = 0)
  then insert into $\lceil codegen_{proc}(expr)\rceil$ (value) values ($var$);
  end if;
  ```

5 Dealing with Iterator Subexpressions

The key idea underlying our mapping from OCL to MySQL is the use of stored procedures to deal with iterator expressions. However, since stored procedures cannot be called within queries, the recursive definition of our code generator needs to treat the case of immediate subexpressions which are iterator expressions in a special way.

More concretely, let *expr* be an OCL expression with top-operator *op* and immediate subexpressions $expr_1,\ldots,expr_n$. Now, let $expr_i$, $1 \leq i \leq n$, be an iterator expression. Then, except for the case of iterator expressions whose *body*-subexpressions are themselves iterator expressions (i.e., the case of nested iterator expressions), which we will discuss later, the query $codegen(expr)$ produced by our code generator will be preceded by

`call` $codegen_{proc}(expr_i)$`;`

and, moreover, any subquery $codegen(expr_i)$ occurring in the definition of $codegen(expr)$, as given in Section 3 and Section 4, will be replaced by the following subquery

`select * from` $\lceil codegen_{proc}(expr_i)\rceil$ `as` $\overline{codegen_{proc}(expr_1)}$`;`

Example 1 *For example, let expr be the OCL expression* $expr_1$`->notEmpty()`*, where* $expr_1$ *is the non-iterator expression* `Car.allInstances()`*. Our code generator will produce the following query:*

`select count(*)>0 as value from` $(codegen(expr_1))$ `as` $\overline{codegen(expr_1)}$`;`

On the other hand, let expr be the OCL expression $expr_1$`->notEmpty()`*, where* $expr_1$ *is the iterator expression* `Car.allInstances()->collect(c|c.model)`*. Our code generator will produce the following query:*

`call` $codegen_{proc}(expr_1)$`;`
`select count(*)>0 as value from`
`(select * from` $\lceil codegen_{proc}(expr_1)\rceil$`)` `as` $\overline{codegen_{proc}(expr_1)}$`;`

5.1 Nested iterators

Let *expr* be an iterator expression of the form *source*->*iter_op*`(`*var*`|`*body*`)`. Then, if the subexpression *source* is itself an iterator expression, but not the subexpression *body*), we simply proceed as explained above. However, when the subexpression *body* is itself and iterator expression, the general scheme for mapping iterator expressions shown in Figure 1 is slightly modified. More concretely, right before the "iterator specific processing code", our code generator will insert

`call` $codegen_{proc}(body)$`(`*var*`);`

where now the procedure generated for mapping the *body*-iterator subexpression takes one parameter: namely, the iterator variable *var* introduced by the enclosing iterator expression.[4] Moreover, our code generator will replace any subquery $codegen(body)$ occurring in the "iterator-specific processing code", as given in Section 4, by the following subquery:

[4] More generally, in the case of nested iterator expressions, the procedure generated for mapping an inner iterator *body*-subexpression will take as parameters the iterator variables introduced by all its enclosing iterator expressions, which we assume to have different names.

Scenario I: 10^3 persons × 10 non-“black” cars, p = `Car.allInstances().owner.ownedCars`	EOS	MySQL 4OCL
p`->size()`	30ms	180ms
p`->collect(x|x.color)->size()`	80ms	8.70s
p`->collect(x|x.color <> 'black')->size()`	90ms	8.73s
p`->collect(x|x.owner.ownedCars)->size()`	240ms	15.25s
p`->collect(x|x.owner.ownedCars->includes(x))->size()`	221ms	17.24s
p`->forAll(x|x.owner.ownedCars->includes(x))`	251ms	16.20s
p`->select(x|x.owner.ownedCars->includes(x))->size()`	260ms	19.84s
p`->collect(x|x.owner.ownedCars.color)->size()`	290ms	37.78s
p`->collect(x|x.owner.ownedCars.color->size())->sum()`	270ms	36.79s
p`->forAll(x|x.owner.ownedCars.color->excludes('black'))`	280ms	33.42s

Table 1: Efficiency evaluation. Preliminary results.

`select * from` $\lceil codegen_{proc}(body)\,(var)\rceil$

6 A Preliminary Discussion on Efficiency

In [CED08] we discussed: i) the need for an efficient implementation of OCL; ii) the aspects to be taken into consideration to improve the efficiency of OCL evaluators on medium-large scenarios; iii) the limits of the current OCL implementations for dealing with really large scenarios. To motivate i), we included a benchmark showing the performance of some OCL tools on medium-large scenarios.

Although the aim of our code generator is not to address i), that is, the efficient implementation of OCL evaluation, but rather to overcome iii), that is, the limits of the current OCL implementations for dealing with really large scenarios, we found interesting to compare the execution of the code produced by our MySQL4OCL with EOS,[5] using for this purpose essentially the same benchmark proposed in [CED08]. All the expressions in the benchmark were evaluated on the same scenario, namely, an instance of the “Car-ownership” class diagram which contains 10^3 persons, each person owning 10 different cars, and each car with a color different from “black”. The results are shown in Table 1.[6] Notice that, for the sake of the experiment, we artificially increased the size of the collections to be iterated upon: more concretely, in Table 1, p stands for the expression `Car.allInstances().owner.ownedCars`, which, on the given scenario, evaluates to a collection with 10^5 cars.

As expected, for small-medium size scenarios, it is faster to evaluate OCL expressions using EOS than to execute the code generated by MySQL4OCL. Interestingly, the cost of executing this MySQL code seems to depend, as in the case of EOS, on two measurements: first, the

[5] In the benchmark proposed in [CED08], EOS outperformed the other OCL evaluators. We do not have more recent comparison figures.

[6] In the case of MySQL4OCL, the benchmark was run on a laptop computer with two processors at 2.40GHz, 2GB of RAM and default settings for mySQL 5.1 Community Server. In the case of EOS, the benchmark was run on a laptop computer, with a single processor at 2GHz, 1GB of RAM, and setting JVM parameters -Xms and -Xmx to 1024m.

maximum number of times that objects' properties will be accessed and, second, the maximum size of the collections that will be built. In fact, for the expressions in this benchmark, the extra-cost of executing the code generated by MySQL*4*OCL is essentially linear with respect to the cost of evaluating the expressions in EOS.

The advantage of using our code generator comes when evaluating OCL expressions on large scenarios. As reported in [CED08], none of the available OCL evaluators, including EOS, were able to finish loading a scenario with 10^6 cars in less than 20 minutes.[7] In contrast, loading this scenario on a MySQL server may take less than a minute. However, it remains to be addressed the question of whether executing the code produced by MySQL*4*OCL on large scenarios is sufficiently efficient. For this purpose, we have run again on MySQL*4*OCL the benchmark proposed in [CED08], but this time on a considerably larger scenario: namely, one that contains 10^5 persons, each person owning 10 different cars, and each car with a color different from "black". As expected, the execution times scale-up linearly with respect to those shown for MySQL*4*OCL in Table 1: basically, they are multiplied by 10^2, in line with the fact that the number of persons and cars are also multiplied by 10^2. Although these results are encouraging, more experiments and comparisons are still needed in order to extract definite conclusions about the efficiency on large scenarios of the code generated by MySQL*4*OCL.

7 Related Work

The work most directly related with this paper can be found in [DH99, Sch98] and provides the foundations of the OCL2SQL tool [DHL01, HWD08]. As already discussed in [CED08], the solution offered in [DH99, Sch98] is not satisfactory: it only considers a rather restricted subset of the OCL language; it only applies to boolean expressions and not to arbitrary queries; and the "complexity" of the produced code makes impractical its use for evaluating expressions on medium-large scenarios.[8] There are many differences between our mapping and the one underlying OCL2SQL (along with some commonalities, of course). Here we only discuss the two most relevant differences. First, we map navigation expressions using "left join", while OCL2SQL uses "in". With "left join", we avoid erroneously removing duplicated elements when dealing with bags in arbitrary OCL queries. Second, and more relevant for our present purposes, we give a well-defined mapping of OCL iterator operators using stored procedures. Interestingly, this idea was already hinted in [Sch98]. However, as it is recognized there, "this work did not succeed in finding a concise and complete formal representation for procedural mapping patterns", and the idea was not further developed afterwards.

The mapping from OCL to MQL (the SAP metamodel repository query language) proposed in [Bur06] is also related with our work. Unfortunately, iterator expressions or boolean operators on collections (e.g., `isEmpty()` or `includes()`) are not covered by the proposed mapping, due to the limited expressiveness of the target language. On a more abstract level,

[7] In fact, we do not know how long would it take to actually finish this task for the different OCL evaluators, since we decided to stopped, in all cases, the loading process after 20 minutes. In any event, it is clear that, for most applications, loading large scenarios in memory will be rather impractical.

[8] As reported in [CED08], the cost of executing the code generated by OCL2SQL for a simple `forall`-expression was the following: 25s for a collection with 10^4 elements, and 45m for a collection with 10^5 elements.

several methodologies have been recently proposed in [MHS09, AN06, AN09] to generate code (possibly for specific data storage language) from OCL expressions. It remains to be investigated how our code generator fits in these (yet not fully developed) proposals. Also, there have been several interesting proposals for mapping OCL expressions into Java code [DBL05, Wil09]. However, we envision for the code generated by these mappings the same general limitation reported in [CED08] for OCL evaluators: namely, that for really large scenarios they need first to solve the loading problem within a practical time-frame.

Finally, it is worthwhile mentioning the work done in [MAB08, Coo09, GMRS09] on translating other (constraint/query) languages to SQL, since they aim to bridge application and databases, which is also part of our motivation. A detailed discussion on the expressiveness and/or practical interest of OCL with respect to the (class of) source languages considered in [MAB08, Coo09, GMRS09] is, however, out of the scope of this paper.

8 Conclusions and Future work

In this paper, we have introduced a MySQL code generator for a significant subset of OCL expressions (including, possibly nested, iterator expressions) which is based on the use of stored procedures for mapping OCL iterators. Our code generator has been already implemented in the MySQL4OCL tool, which is available at [EDC10]. Since the features and language constructs that are employed in our mapping from OCL to MySQL are supported (leaving aside syntactic differences) by other relational databases such as Oracle and PostgreSQL, we expect that our code generator could be adapted for other database management systems. Also, we have discussed the efficiency of the code produced by our code generator, and we have compared it with previous known results on evaluating OCL expressions on medium-large scenarios. Additional experiments and comparisons are still needed in order to extract more definite conclusions, and we plan to carry them on as part of our future work. In this context, we plan to investigate which transformations between equivalent OCL expressions are useful in order to optimize the code produced by our code generator.

Among the features of OCL that our code generator, as presented here, does not cover, we briefly discuss how we shall extend our mapping to cover three of them: namely, (i) the possibility of defining collection of collections (e.g., using `collectNested`); (ii) the possibility of denoting or checking types (e.g., using `oclType`, `oclIsTypeOf`, and `oclIsKindOf`); and (iii) the possibility of defining (maybe using recursion) operations. Now, to cover (i) and (ii), we have to modify our queries $codegen(expr)$ in order to obtain more "structured" result-sets. More concretely, to cope with expressions denoting or checking types, each 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. Next, to cope with (iii) we will resort to the use of stored procedures. Finally, we have also left uncovered the dealing with the special value "undefined". In principle, we shall treat undefinedness adding conditions in the code produced by our code generator for the different OCL operators.

Acknowledgements: We thank Michael Schläpfer for his interesting discussions and help during the initial stage of this proposal.

Bibliography

[AN06] A. Armonas, L. Nemuraitė. Pattern Based Generation of Full-Fledge Relational Schemas From UML/OCL Models. *Information Technology and Control* 35(1), 2006.

[AN09] A. Armonas, L. Nemuraitė. Using Attributes and Merging Algorithms for Transforming OCL expressions into Code. *Information Technology and Control* 38(4), 2009.

[Bur06] E. Burger. Query Infrastructure and OCL within the SAP Project "Modeling Infrastructure"- Studienarbeit. Technical report, Institut für Theoretische Informatik - Technische Universität Karlsruhe, Germany, 2006.

[CED08] M. Clavel, M. Egea, M. G. de Dios. Building and Efficient Component for OCL Evaluation. In *Proc. of 8th OCL Workshop at the UML/MoDELS Conference: OCL Concepts and Tools: From Implementation to Evaluation and Comparison*. ECEASST 15. Tolouse, France, September 2008.

[Coo09] E. Cooper. The Script-Writer's Dream: How to Write Great SQL in Your Own Language and Be Sure It Will Succeed. In Gardner and Geerts (eds.), *Proc. of 12th International Symposium Database Programming Languages - DBPL 2009*. LNCS 5708, pp. 36–51. Springer, 2009.

[DBL05] W. Dzidek, L. Briand, Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In *Proc. of the 4th OCL workshop at MoDELS'05 Conference: Tool Support for OCL and Related Formalisms - Needs and Trends*. LNCS 3844, pp. 10–19. Springer-Verlag Berlin Heidelberg, Montego Bay, Jamaica, October 2005.

[DDS$^+$10] M. A. G. de Dios, C. Dania, M. Schläpfer, D. A. Basin, M. Clavel, M. Egea. SSG: A Model-Based Development Environment for Smart, Security-Aware GUIs. In Kramer et al. (eds.), *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. Pp. 311–312. ACM, 2010. http://www.bm1software.com.

[DH99] B. Demuth, H. Hußmann. Using UML/OCL Constraints for Relational Database Design. In France and Rumpe (eds.), *Proc. of UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*. LNCS 1723, pp. 598–613. Springer, 1999.

[DHL01] B. Demuth, H. Hußmann, S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In *Proc. of UML 2001: The Unified Modeling*

Language. Modeling languages, Concepts and Tools. LNCS 2185, pp. 104–117. Springer, Toronto, Canada, 2001.

[EDC10] M. Egea, C. Dania, M. Clavel. The MySQL-OCL Code Generator. August 2010. http://www.bm1software.com/mysql-ocl.

[GMRS09] T. Grust, M. Mayr, J. Rittinger, T. Schreiber. FERRY: Database-supported Program Execution. In Cetintemel et al. (eds.), *Proc. of the 35th SIGMOD international conference on Management of data*. SIGMOD '09, pp. 1063–1066. ACM, New York, NY, USA, 2009.

[HWD08] F. Heidenreich, C. Wende, B. Demuth. A Framework for Generating Query Language Code from OCL Invariants. In *Proc. of 7th OCL Workshop at the UML/MoDELS Conference: Ocl4All: Modelling Systems with OCL*. ECEASST 9. Nashville, Tennessee, October 2008.

[MAB08] S. Melnik, A. Adya, P. A. Bernstein. Compiling Mappings to Bridge Applications and Databases. *ACM Transactions Database Systems* 33:1–50, December 2008.

[MHS09] R. Moiseev, S. Hayashi, M. Saeki. Generating Assertion Code from OCL: A Transformational Approach Based on Similarities of Implementation Languages. In A.Schürr and Selic (eds.), *Proc. of Model Driven Engineering Languages and Systems, 12th International Conference, MoDELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*. LNCS 5795, pp. 650–664. Springer, 2009.

[Sch98] A. Schmidt. Untersuchungen zur Abbildung von OCL-Ausdrücken auf SQL. Master's thesis, Institut für Softwaretechnik II - Technische Universität Dresden, Germany, 1998.

[Wil09] C. Wilke. Java Code Generation for Dresden OCL2 for Eclipse- Grosser Beleg. Technical report, Fakultät Informatik - Institut für Software un Multimediatechnik - Technische Universität Dresden - Lehrstuhl Softwaretechnolgie, Germany, 2009.

A Examples

Consider the "Car-ownership" UML class diagram in Figure 2: it contains two classes, "Person" and "Car", which are related by the association "Ownership", which links persons ("owner") with their cars ("ownedCars"). To illustrate the mapping rules for UML class and object diagrams given in Section 2, we show below the MySQL statements that create the tables corresponding to "Car-ownership".

```
create table Person (pk int auto_increment primary key,
   name char(65), age int, phoneno int);
create table Car (pk int auto_incremente primary key,
   model char(65), color char(65));
create table ownership (owner int not null, ownedCars int not null,
   foreign key (owner) references Person (pk),
   foreign key (ownedCars) references Car (pk));
```

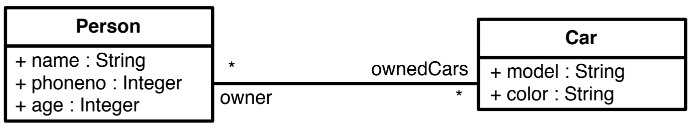

Figure 2: The "Car-ownership" class diagram

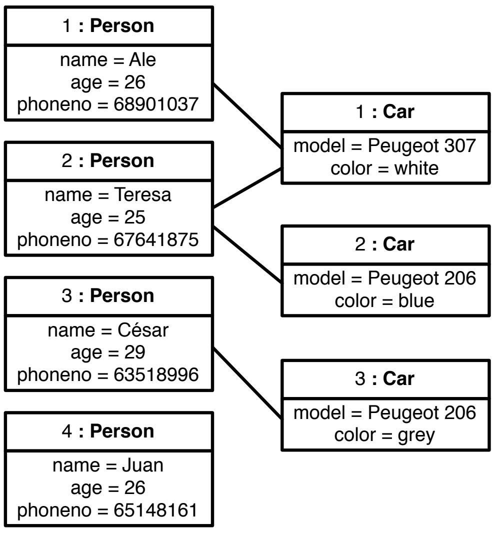

Figure 3: A "Car-ownership" object diagram

Person

pk	name	age	phoneno
1	Ale	26	68901037
2	Teresa	25	67641875
3	César	29	63518996
4	Juan	26	65148161

Car

pk	model	color
1	Peugeot 307	white
2	Peugeot 206	blue
3	Peugeot 206	grey

ownership

owner	ownedCars
1	1
2	1
2	2
3	3

Figure 4: Mapping a "Car-ownership" object diagram.

Also, consider the instance of "Car-ownership" in Figure 3. We show in Figure 4 the representation of this object diagram in the tables created by our mapping for "Car-ownership".

Next, to illustrate the recursive definition of our code generator given in Section 3 and Section 4, we introduce the following examples:

Example 2 *Let expr be the OCL expression* `Car.allInstances().model`*. Then, codegen*$(expr)$ *is the following MySQL statement:*

```
select Car.model as value
from (select pk value from Car) as temp
left join Car on temp.value = Car.pk;
```

Example 3 *Let expr be the OCL expression* `Car.allInstances().owner`*. Then, codegen*$(expr)$ *is the following MySQL statement:*

```
select ownership.owner as value
```

```
from (select pk value from Car) as temp
left join ownership on temp.value = ownership.ownedCars
where ownership.owner is not null;
```

Example 4 *Let expr be the OCL expression* `Car.allInstances()->forAll(c| c.owner.ownedCars->includes(c))`. *Then, codegen*(*expr*) *is the following MySQL statement:*

```
create procedure forAll0()
begin
declare done Int default 0;
declare result boolean default true;
declare var Int;
declare crs cursor for (select pkCar from Car);
declare continue handler for sqltate '02000' set done = 1;
drop table if exists forAll0;
create table forAll0(value int);
insert into forAll0 (value) values (1);
open crs;
  repeat
  fetch crs into var;
  if not done then
    update forAll0 set value = 0 where(
      select var in (
      (select ownership.ownedCars as value from
      (select ownership.owner as value from
      (select var as value) as t3
      left join ownership on t3.value=ownership.ownedCars
      where ownership.owner is not null) as t2
      left join ownership on t2.value=ownership.owner
      where ownership.ownedCars is not null) as t1) = 0;
    if exists (select 1 from forAll0 where value = 0)
    then set done = 1;
    end if;
  end if;
until done end repeat;
close crs;
end;
call forAll0();
select * from forAll0;
```