The TTC 2020 OCL2PSQL Case

Hoang Nguyen Phuoc Bao
ngpbhoang1406@gmail.com
Vietnamese-German University
Binh Duong, Vietnam

Antonio García-Domínguez
a.garcia-dominguez@aston.ac.uk
Aston University
Birmingham, United Kingdom

Manuel Clavel
manuel.clavel@vgu.edu.vn
Vietnamese-German University
Binh Duong, Vietnam

ABSTRACT

The Object Constraint Language (OCL) is a textual, declarative language used as part of the UML standard for specifying constraints and queries on models. As such, generating code from OCL expressions is part of an end-to-end model-driven development process. Certainly, this is the case for database-centric application development, where integrity constraints and queries can be naturally specified using OCL. Not surprisingly, there have been already several attempts to map OCL into SQL. In this case study, we invite participants to implement, using their own model-transformation methods, one of these mappings, called OCL2PSQL. We propose this case study as a showcase for different methods to prove their readiness to coping with moderately complex model-transformations, by showing the usability, conciseness, and ease of understanding of their solutions when implementing OCL2PSQL.

KEYWORDS

Model-transformation, OCL, SQL, TTC

ACM Reference Format:
Hoang Nguyen Phuoc Bao, Antonio García-Domínguez, and Manuel Clavel. 2020. The TTC 2020 OCL2PSQL Case. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Object Constraint Language (OCL) [7] is a textual language typically used, as part of the UML standard [8], for specifying constraints and queries on models. It is a pure specification language: when an expression is evaluated, it simply returns a value without changing anything in the underlying model. OCL is a strongly-typed language: expressions either have a primitive type, a class type, a tuple type, or a collection type. The language provides standard operators on primitive data, tuples, and collections. It also provides a dot-operator to access the properties of the objects, and several *iterators* to iterate over collections.

The Structured Query Language (SQL) [9] is a special-purpose programming language designed for managing data in relational database management systems (RDBMS). Its scope includes data insert, query, update and delete, schema creation and modification, and data access control. Although SQL is to a great extent a declarative language, it also contains *stored procedures.* These are routines stored in the database that may execute *loops* using the so-called *cursors.*

In the context of model-driven engineering (MDE), there exists several proposals for translating OCL into SQL [1, 3–5], which mostly differ in the way OCL iterators are translated. In particular, [3, 4] resort to imperative features of SQL, namely, loops and cursors, for translating OCL iterators, while [1] introduces a mapping, called OCL2PSQL, which only uses standard *subselects* and *joins* for translating OCL iterators. [1]

As an example of the transformations produced by OCL2PSQL, suppose that we want to know if, in a given scenario, *there is at least one car whose color is not 'black'.* We can formalize this query in OCL as follows:

Car.allInstances()−>exists(c|c.color <>'black')

where Car is a class and color is an attribute of Car. OCL2PSQL translates this expression into an SQL select-statement as follows:

```
SELECT COUNT(*) > 0 as res, 1 as val
FROM (
  SELECT TEMP_left.res <> TEMP_right.res as res,
  CASE TEMP_left.val = 0 OR TEMP_right.val = 0
  WHEN 1 THEN 0 ELSE 1 END as val,
  TEMP_left.ref_c as ref_c
  FROM (
    SELECT color as res,
    TEMP_obj.val as val,
    TEMP_obj.ref_c as ref_c
    FROM (
      SELECT TEMP_dmn.res as res,
      TEMP_dmn.res as ref_c,
      TEMP_dmn.val as val
      FROM (
        SELECT Car_id as res, 1 as val FROM Car
      ) as TEMP_dmn
    ) as TEMP_obj
    LEFT JOIN Car
   ON TEMP_obj.ref_c = Car.Car_id AND TEMP_obj.val = 1
  ) AS TEMP_left
  JOIN (
    SELECT 'black' as res, 1 as val
  ) AS TEMP_right
) as TEMP_body
WHERE TEMP_body.res = 1;
```

The full recursive definition of OCL2PSQL can be found in [1]. For the sake of illustration, we have included in Appendix A the

[1]The letter "P" in OCL2PSQL stands for *pure* SQL. The idea is that OCL2PSQL only uses the declarative features of SQL for mapping OCL expressions.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

definition of the recursive cases involved in the above transformation. The *correctness* of the mapping is formulated as follows. Let e be an OCL expression (with no free variables) and let O be a scenario of its context model. Then, the evaluation of the expression e in the scenario O should return the same result that the execution of the query OCL2PSQL(e), i.e., the SQL query generated by OCL2PSQL from e, in the database OCL2PSQL(O), i.e., the database corresponding to O according to OCL2PSQL.[2]

The TTC 2020 OCL2PSQL Case welcomes participants to implement the OCL2PSQL mapping using their own model-transformation methods. This case study can serve as a showcase for different methods to prove their readiness to coping with moderately complex model-transformations, by showing the usability, conciseness, and understandability of their solutions when implementing OCL2PSQL. Participants are also welcome to extend or modify the OCL2PSQL mapping, or even to propose their own mapping from OCL to SQL, in which cases they should also provide convincing arguments that their solution is correct. Finally, participants are most welcome to propose their own attributes of interests: for example, flexibility for multiple RDBMSs, or support for formal verification.

All resources for this case are available on Github [3]. Please follow the description in the footnote and create a pull request with your own solution after you have submitted your description to EasyChair.

The rest of the document is structured as follows: Section 2 describes the input and output of the OCL2PSQL transformation. Then, Section 3 provides the main task that should be tackled in a solution (participants are free to propose their own tasks of interest). Finally, Section 4 proposes the case evaluation scheme for the contest.

2 TRANSFORMATION DESCRIPTION

OCL2PSQL is a recently proposed mapping from OCL to SQL [1]. It addresses some of the challenges and limitations of previous OCL-to-SQL mappings, particularly with respect to the execution-time efficiency of the generated SQL queries. [2]

Next, we give a detail description of the input and output metamodels for the TTC 2020 OCL2PSQL Case. [4] The input metamodels represent the part of OCL language that is currently covered by OCL2PSQL. The output metamodel represent the part of the SQL language that is currently used by OCL2PSQL to translate OCL. Obviously, for solutions that extend or modify the OCL2PSQL mapping, as well as for solutions that propose an entirely different mapping from OCL to SQL, the input and output metamodels presented here may need to be extended or modify accordingly.

2.1 Input Metamodel

OCL is a contextual language: its expressions are written in the context provided by a *data model*. Consequently, the input metamodel for OCL2PSQL can be seen as consisting of two, inter-related metamodels: namely, the metamodel for data models and the metamodel for OCL expressions.

2.1.1 Input metamodel for data models. For OCL2PSQL, a data model contains classes and associations. A class may have attributes and associations-ends. The multiplicity of an association-end is either 'one' or 'many'.

The data model metamodel for OCL2PSQL is shown in Figure 1. `EDataModel` is the root element and contains a set of `EEntitys`. Every `EEntity` represents a class in the data model: it has a unique name and is related to a set of `EAttributes` and a set of `EAssociationEnds`. Each `EAttribute` represents an attribute of a class: it has a name and a type. Each `EAssociationEnd` represents an association-end: it has a name and an `EMultiplicity` value. Each `EAssociationEnd` is also linked to its opposite `EAssociationEnd`, and with its `target` `EEntity`.

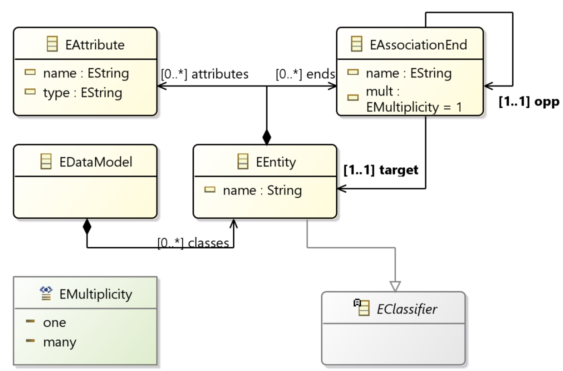

Figure 1: OCL2PSQL metamodel for data models.

2.1.2 Input metamodel for OCL expressions. The definition of the OCL mapping, as presented in [1], only covers a subset of the OCL language. The OCL2PSQL metamodel for OCL is shown in Figure 2. `EOclExpression` is the root element. It is an abstract class. An `EOclExpression` can be either an `ECallExp`, an `ELiteralExp`, an `EVariableExp`, or an `ETypeExp`. Next, we describe each of these classes.

An `ELiteralExp` represents a literal value. It is an abstract class. An `ELiteralExp` can be either an `EIntegerLiteralExp`, an `EStringLiteralExp`, or an `EBooleanLiteralExp`. Each of these classes contains an attribute to represent, respectively, an integer, a string, or a Boolean literal value.

An `ETypeExp` represents a type expression. It contains an attribute `referredType` of type EClassifier. In this contest, we consider one realization of `EClassfier`, which is `EEntity` class from the OCL2PSQL metamodel for data models.

An `EVariableExp` represents a variable expression.

An `ECallExp` represents an expression that consists of calling a *feature* over a `source`. The later is represented by an `EOclExpression`. `ECallExp` is an abstract class. An `ECallExp` can be

[2]The OCL2PSQL mapping rests on an underlying mapping between data models and SQL database schema. The full definition of this mapping is also provided in [1].
[3]https://github.com/bluezio/ttc2020-ocl2sql (temporary, to be moved to main TTC Github organisation if accepted)
[4]Although we use "E" as a prefix in the names of the classes in the input and output metamodels, none of these classes actually belong to the Ecore metamodel, except for `EString`, `EBooleanObject`, `EIntegerObject`, and `EBoolean`.

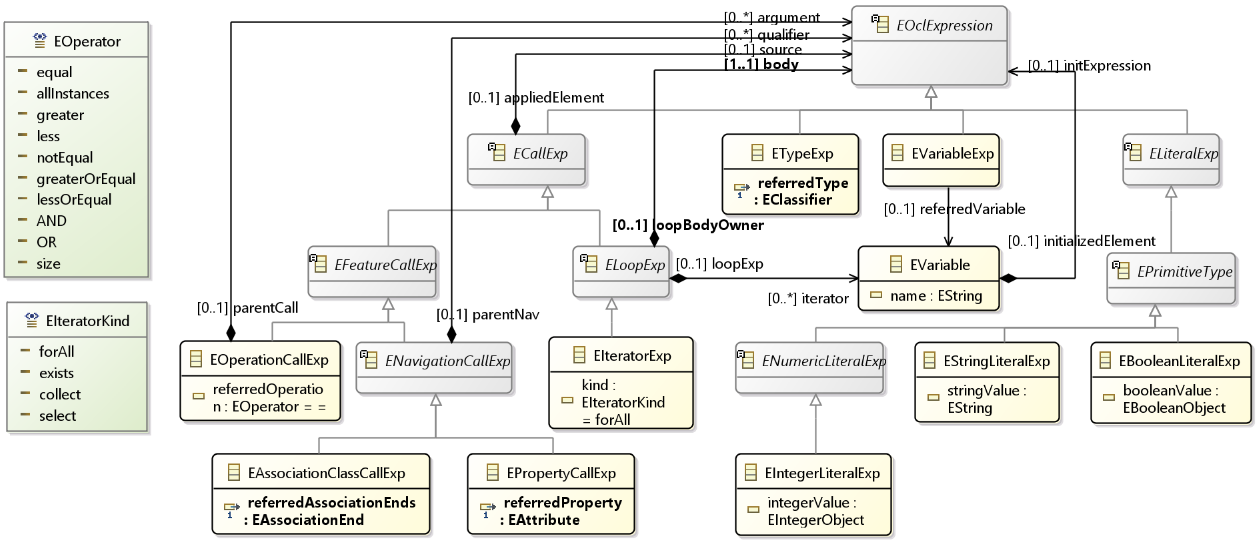

Figure 2: OCL2PSQL metamodel for OCL expression.

either an `EOperationCallExp`, an `EPropertyCallExp`, an `EAssociationClassCallExp`, or an `EIteratorExp`. Next, we describe each of these classes.

An `EOperationCallExp` represents an expression that calls an operation over its `source`, possibly with arguments. The OCL2PSQL metamodel for OCL only considers the following operations: (i) literal comparisons: =,<>,>,≥,<,≤; (ii) logical operations: AND, OR, NOT; and (iii) operations on collections: allInstances() and size().

An `EPropertyCallExp` represents an expression that calls an attribute over an object. The former is represented by an `EAttribute` and the later is represented by an `EEntity`; both belong to the OCL2PSQL metamodel for data models. OCL2PSQL only supports `EPropertyCallExp` expressions whose `source` is an `EVariableExp` expression.

An `EAssociationClassCallExp` represents an expression that calls an association-end over an object. The former is represented by an `EAssociationEnd` and the later is represented by an `EEntity`; both belong to the OCL2PSQL metamodel for data models. OCL2PSQL only supports `EAssociationClassCallExp` expressions whose `source` is an `EVariableExp` expression.

An `EIteratorExp` represents an expression that calls an iterator over a collection. The body of the iterator is represented by an `EOclExpression` expression. The iterator-variable is represented by a `Variable`. OCL2PSQL supports the following kinds of iterators: `forAll`, `exists`, `collect`, and `select`.

2.2 Output Metamodel

For OCL2PSQL, a SQL query is a basic SQL `SELECT`-statement, which may contain subselects, `WHERE`-clauses, `GROUP BY`-clauses, and `JOIN`s.

The OCL2PSQL metamodel for SQL queries consists of two, inter-related metamodels: namely, the metamodel for SQL select-statements, shown in Figure 3, and the metamodel for SQL expressions, shown in Figure 4.

2.2.1 *Output metamodel for SQL select-statements.*

`ESelect` is the root element. Every `ESelect` has a `selectBody` of type `ESelectBody`, which represents the *body* of the select-statement. `ESelectBody` is an interface. OCL2PSQL only supports one type of `ESelectBody`, namely, `EPlainSelect`.

An `EPlainSelect` may contain the following elements: a `fromItem` element of type `EFromItem`; a list of `selectedItems` elements, each of type `ESelectItem`; a `distinct` element of type `EDistinct`; a `where` element of type `EExpression`; a `groupBy` element of type `EGroupByElement`; a list of `joins` elements, each of type `EJoin`; and a `having` element of type `EExpression`. Next, we describe each of these classes:

An `ESelectItem` represents a *column* that the select-statement retrieves. It is an interface. An `ESelectItem` element can be either an `EExpression` (see below) or an `EAllColumns`. The later represents *.

An `EDistinct` element represents the modifier `DISTINCT`. It contains the list `onSelectItems` of the columns to which the modifier applies.

An `EFromItem` element represents the *table* or *subselect* from which the select-statement retrieves information. It is an interface. An `EFromItem` element can be either an `ETable` or a `SubSelect`. The former represents a *table*. The later represents a *subselect*.

A `where` element of type `EExpression` represents a *where*-clause.

An `EJoin` element represents a *join* with a `rightItem` of type `EFromItem`, possibly according to an `onExpression` of type `EExpression`.

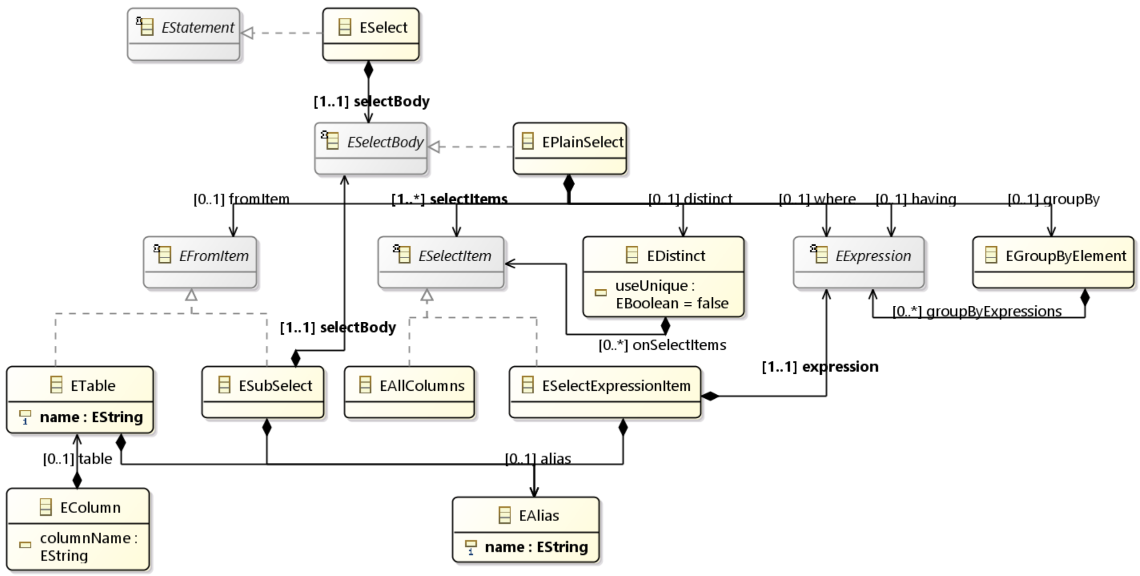

Figure 3: OCL2PSQL metamodel for SQL select-statements.

An `EGroupByElement` element represents a *groupby*-clause. It contains a list `groupByExpressions` of expressions of type `EExpression` that defines how the rows are to be *grouped by*.

A `having` element of type `EExpression` represents a *having*-clause.

2.2.2 *Output metamodel for SQL expressions.*

`EExpression` is the root element. It is an interface. Next, we describe the realizations of this expression that OCL2PSQL currently considers.

An `ELongValue`, `EStringValue`, and `ENullValue` represent, respectively, a literal integer, a literal string, and the value `NULL` in SQL.

An `EColumn` represents a column in SQL.

An `EBinaryExpression` represents a binary expression in SQL. It contains a `leftExpression` and a `rightExpression`, both of type `EExpression`. `EBinaryExpression` is an abstract class. It can be either a logical expression, namely `EOrExpression` and `EAndExpression`, or a comparison expression, namely `EEqualsTo`, `ENotEqualsTo`, `EMinorThan`, `EMinorThanEquals`, `EGreaterThan`, and `EGreaterThanEquals`.

An `ENotExpression` represents a `NOT`-expression in SQL. It contains an `expression` of type `EExpression`.

An `EIsNullExpression` represents an `IS NULL`-expression in SQL. It contains a `leftExpression` of type `EExpression`.

An `EFunction` represents a functional expression in SQL. It contains the `name` of the function, a Boolean value `allColumns` (to indicate whether the function applies to all columns or not), and a Boolean value `distinct` (to indicate whether the `DISTINCT` modifier applies or not). It also contains a list of `parameters` of type `EExpressionList`. An `EExpressionList` simply contains a list of `EExpressions`.

An `ECaseExpression` represents a `CASE`-expression in SQL. It contains a list `whenClauses` of `EWhenClause`, representing `CASE-WHEN`-clauses in SQL. It may also contain a `switchExpression` of type `EExpression`, representing a `CASE`-switch in SQL, as well as an `elseExpression` of type `EExpression`, representing a `CASE-ELSE` clause in SQL. An `EWhenClause` contains a `whenExpression` and a `thenExpression`, both of type `EExpression`.

An `ESubSelect` represents a subselect-expression in SQL. It contains a `selectBody` of type `ESelectBody`, which is introduced in the OCL2PSQL metamodel for SQL select-statements.

3 MAIN TASK

The main task for the participants in the TTC 2020 OCL2PSQL Case consists of implementing, using their own model-transformation methods, the OCL2PSQL mapping as it is defined in [1] (with the "fixes" included in Appendix B). Participants are free to extend or modify the OCL2PSQL mapping, or even to propose their own mapping from OCL to SQL, in which case they should also provide convincing arguments that their solution is correct with respect to the semantics of OCL and SQL.

During the contest, the participants will be presented with different *challenges* of increasing complexity. Each challenge will be an OCL2PSQL OCL expression, i.e., an instance of the OCL2PSQL metamodel for OCL. The *context* for all the challenges will be an OCL2PSQL data model, i.e., an instance of the OCL2PSQL metamodel for data models. Then, the participants will be asked to generate the *solutions* for these challenges, applying their own transformation rules. Very importantly: (i) each solution should be an SQL select-statement valid in the database schema corresponding to the given data model, according to the definition of the OCL2PSQL mapping; moreover, (ii) each solution should be an SQL select-statement returning a table with (at least) a column

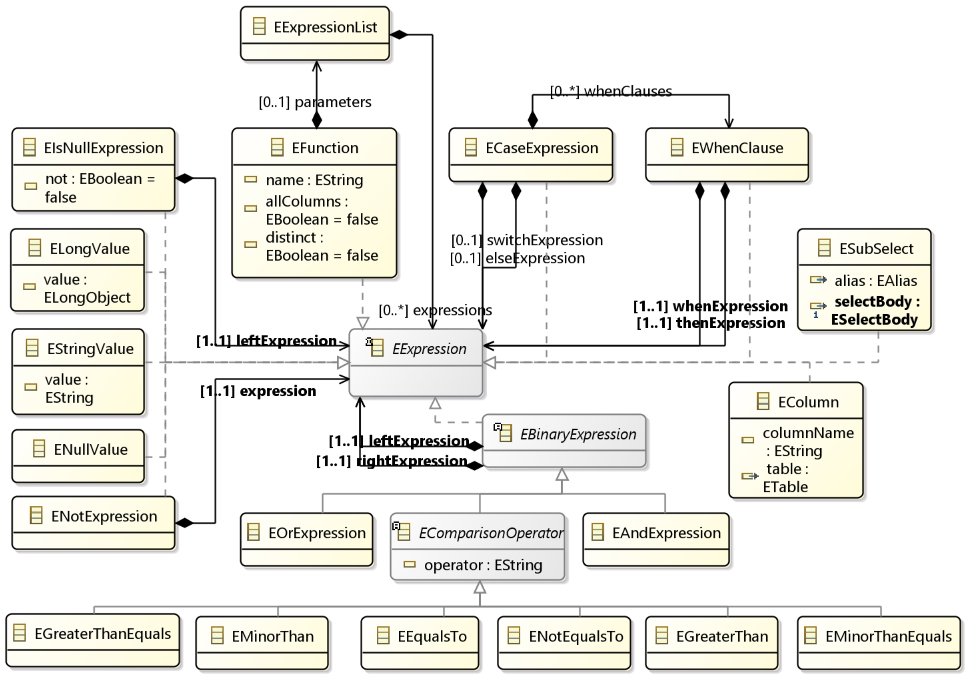

Figure 4: OCL2PSQL metamodel for SQL expressions.

res. Then, when *executing* the solution for a challenge on a given scenario, this column res will be interpreted as holding the result of *evaluating* the given challenge in the same scenario. Finally, the solutions will be checked for *correctness*, using a set of selected *scenarios*.

For the purpose of testing, the participants can find the following material in the ttc2020 repository:

- In the docs folder, the file challenges.txt contains a list of *challenges* grouped in *stages*. Each stage has a unique number, and each challenge within a stage has also a unique number. The greater the number of a stage, the greater its complexity. The *context* for all the challenges in challenges.txt is the data model CarPerson shown in Figure 5.
  In this same folder the file carperson.sql contains the MySQL database schema that corresponds to CarPerson, according to definition of the OCL2PSQL mapping.
  Also in the docs folder, the file scenarios.txt contains a list of *scenarios*. Each scenario describes an instance of the database carperson.sql. Then, for each scenario, and each (relevant) *stage*/challenge listed in challenges.txt, the file scenarios.txt contains the *correct* result: i.e., the result-table that corresponds to the evaluation of the given *stage*/challenge in the given scenario.
- The folder input contains the challenges listed in challenges.txt in XMI-format. More specifically, each file Stage*i*-Challenge*j*.xmi contains the represention, in XMI-format, of the challenge *j* within the stage *i* in the file challenges.txt.
  In the same folder, the file CarPerson.xmi contains the data model CarPerson in XMI-format.
- In the folder metamodels, the file ocl.ecore contains the EMF implementation of the OCL2PSQL metamodel for OCL expressions. Also in the same folder, the file sql.ecore contains the EMF implementation of the OCL2PSQL metamodel for SQL-select statements.

Finally, the participants can use, also for the purpose of testing:

- The OCL2PSQL REST API, described in Appendix C, to check the *solutions* generated by OCL2PSQL for their own *challenges* in the context of the data model CarPerson.
- The OCL2PSQL-TEST REST API, described in Appendix D, to test the correctness of their own *solutions* for the *stages* and *challenges* in challenges.txt, with respect to the scenarios in scenarios.txt.

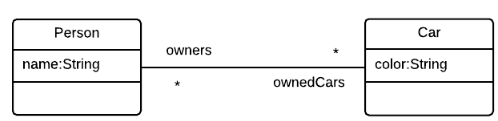

Figure 5: The CarPerson data model.

4 BENCHMARK FRAMEWORK

The case resources on Github include an automated benchmark framework for systematic measurement of the performance and

Listing 1: `solution.ini` file for the `ReferenceXMI` solution

```
[build]
default=mvn compile
skipTests=mvn compile

[run]
cmd=mvn −f pom.xml −quiet −Pxmi exec:exec
```

correctness of the various solutions. It is based on the framework of the TTC 2017 Smart Grid case [6], without the visualisation components. Solution authors are heavily recommended to adapt their solutions to this framework, to allow for the easier integration and comparison of the various solutions.

The configuration of the benchmark framework for the TTC 2020 OCL2PSQL CASE is stored in the file `config.json` inside the folder `config`. This file includes the definitions of the various stages and challenges, the names of the tools to be run, the number of repetitions to be used, and the timeout in milliseconds for each execution. Currently, for purpose of testing, the file `config.json` contains the stages and challenges listed in the file `challenges.txt`. But, for the final contest, the stages and challenges may be different.

4.1 Solution requirements

All solutions must be forks of the main Github project, and should be submitted as pull requests after the descriptions have been uploaded to EasyChair.

All solutions should be in a subdirectory of the `solutions` folder, and inside this subdirectory they should include a `solution.ini` file describing how the solution should be built and how it should be run. As an example, Listing 1 shows the file for the reference solution. The build section provides the `default` and `skipTests` fields for specifying how to build and test, and how to simply build, respectively. In the `run` section, the `cmd` field specifies the command to run the solution.

Solutions should print to their standard output streams a sequence of lines with the following fields, separated by semicolons:

- **Tool**: name of the tool.
- **Stage**: integer with the stage within the case whose challenge is being solved.
- **Challenge**: integer with the challenge within the stage which is being solved.
- **RunIndex**: integer with the current repetition of the transformation.
- **MetricName**: may be "TransformTimeNanos", "TestTimeNanos", or "Scenario*ID*" where *ID* is the identifier of the scenario under test.
- **MetricValue**: the value of the metric:
  – For "TransformTimeNanos", an integer with the nanoseconds spent performing the transformation.
  – For "TestTimeNanos", an integer with the nanoseconds spent testing the correctness of the transformation through the TEST REST API (refer to Appendix D).
  – For metrics following the "ScenarioID" pattern, a string equal to either "passed" or "failed" (as provided by the TEST REST API).

The repetition of the transformation is handled by the framework. Moreover, for every repetition, the framework provides, through environment variables, the following information: the run index, stage number and challenge number, as well as the OCL expression, in XMI-format, corresponding to the challenge, and the context of the challenge, also in XMI-format. More specifically, the available environment variables are:

- **ChallengeIndex**: the index of the challenge within the stage which will be run.
- **Debug**: true if and only if the `-debug` flag has been used to run the main `run/script.py` script.
- **PathToOCLXMI**: the absolute path to the file containing the OCL expression, in XMI-format, corresponding to the challenge to be run.
- **PathToSchemaXMI**: the absolute path to the file containing the SQL schema, in XMI-format, corresponding to the context (data model) of the challenges to be run.
- **RunIndex**: the index of the repetition to be run.
- **Runs**: the total number of repetitions planned.
- **StageIndex**, **StageName**: the index and name of the stage whose challenge is to be run.
- **Tool**: the name of the tool (the name of the `solutions` subfolder).

Solution authors may wish to consult the reference solution for guidance on how to use the TEST REST API to test the correctness of their transformations, and how to use the various environment variables. Solution authors are free to reuse the source code of this reference solution for these aspects (e.g. the `CaseLauncher` and `Configuration` classes), as well as the classes related to the communication with the TEST REST API (in the `reference.api` Java package). The reference solution uses Maven to retrieve the appropriate libraries for communicating with the TEST REST API. In the case of Java, it is using the Apache CXF 3.3.5 JAX-RS implementation, with the Jackson JAX-RS JSON provider in its 2.10.2.1 version.

4.2 Running the benchmark

The benchmark framework needs Python 3.3 or later to be installed, and the reference solution requires Maven 3 and Java 8 or later. Solution authors are free to use alternative frameworks and programming languages, as long as these dependencies are explicitly documented. For the final evaluation, it is planned to construct a Docker image with all solutions, and this will require installing those dependencies into the image.

If all dependencies are installed, the benchmark can be run with `python scripts/run.py` (potentially `python3` if Python 2.x is installed globally in the same system).

5 EVALUATION

The benchmark framework will provide independent measurements of the correctness, completeness, and time usage of the solutions provided by the participants. Attendees to the contest will evaluate the usability, conciseness, and understandability of the transformation rules that define the different solutions, as well as the other attributes of interest that the solution providers may want to focus in. In this regard, although some solutions may not be entirely complete or may be hard to understand, they may still serve as examples of active research areas within model transformations that the community may wish to showcase. To recognize these contributions, an audience-driven "Most Promising" award will be given.

REFERENCES

[1] H. Nguyen Phuoc Bao and M. Clavel. 2019. OCL2PSQL: An OCL-to-SQL Code-Generator for Model-Driven Engineering. In *Future Data and Security Engineering - 6th International Conference, FDSE 2019, Proceedings (Lecture Notes in Computer Science)*, T. Khanh Dang, J. Küng, M. Takizawa, and S. Ha Bui (Eds.), Vol. 11814. Springer, 185–203.

[2] M. Clavel and H. Nguyen Phuoc Bao. 2019. Mapping OCL into SQL: Challenges and Opportunities Ahead. In *19th International Workshop in OCL and Textual Modeling (OCL 2019) co-located with MODELS 2019 (CEUR Workshop Proceedings)*, A. D. Brucker, G. Daniel, and F. Jouault (Eds.), Vol. 2513. CEUR-WS.org, 3–16.

[3] M. Egea and C. Dania. 2019. SQL-PL4OCL: an automatic code generator from OCL to SQL procedural language. *Software and Systems Modeling* 18, 1 (2019), 769–791.

[4] M. Egea, C. Dania, and M. Clavel. 2010. MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. *ECEASST* 36 (2010).

[5] F. Heidenreich, C. Wende, and B. Demuth. 2008. A Framework for Generating Query Language Code from OCL Invariants. *ECEASST* 9 (2008).

[6] Georg Hinkel. 2017. An NMF solution to the Smart Grid Case at the TTC 2017. In *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017 (CEUR Workshop Proceedings)*, Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.), Vol. 2026. CEUR-WS.org, 13–17. http://ceur-ws.org/Vol-2026/paper5.pdf

[7] Object Management Group. 2014. *Object Constraint Language specification Version 2.4.* Technical Report.

[8] Object Management Group. 2017. *Unified Modeling Language.* Technical Report.

[9] International Organization for Standardization. 2011. *ISO/IEC 9075-(1–10) Information technology – Database languages – SQL.* Technical Report.

A THE MAPPING OCL2PSQL IN A NUTSHELL

The mapping OCL2PSQL is defined recursively over the structure of OCL expressions. To describe the key idea underlying its definition, and to illustrate it with the presentation of some recursive cases, we need to introduce first some notation.

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

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

Let e be an OCL expression, such that $\mathrm{FVars}(e) = \emptyset$. Let $\mathcal{O}$ be a scenario. Then, we denote by $\mathrm{Eval}(e, \mathcal{O})$ the result of *evaluating* e in $\mathcal{O}$.

Finally, let $\mathcal{D}$ be a data model. Then, we denote by $\mathrm{map}(\mathcal{D})$ the SQL database schemata corresponding to $\mathcal{D}$, according to OCL2PSQL. Let $\mathcal{D}$ be a data model, and let $\mathcal{O}$ be a scenario of $\mathcal{D}$. Then, we denote by $\mathrm{map}(\mathcal{O})$ the instance of $\mathcal{D}$ corresponding to $\mathcal{O}$, according to OCL2PSQL. Let e be an OCL expression, let e' be a subexpression of e. Then, we denote by $\mathrm{map}_e(e')$ the SQL query corresponding to e', according to OCL2PSQL.

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

We illustrate the recursive definition of OCL2PSQL with the cases corresponding to literal strings, variable, attribute-expressions, `allInstances`-expressions, `<>`-expressions, and `exists`-expressions.

Literal strings

Let e be an OCL expression. Let e' be a subexpression of e. Let $e' = s$, where s is a literal string. Then,

$\mathrm{map}_e(s) =$
`SELECT` s `as res, 1 as val.`

Variables

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

$\mathrm{map}_e(v) =$
`SELECT`
  `TEMP_dmn.res as res,`
  `TEMP_dmn.res as ref_`v`,`
  `TEMP_dmn.val as val,`
  `TEMP_dmn.ref_`v' `as ref_`v', for each $v' \in \mathrm{SVars}_e(\mathrm{src}(v))$
`FROM (`$\mathrm{map}_e(\mathrm{src}(v))$`) as TEMP_dmn.`

Attribute-expressions

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

$\mathrm{map}_e(v\,.att) =$
`SELECT`
  c`.`att as res,
  `TEMP_obj.val as val,`
  `TEMP_obj.ref_`v' `as ref_`v', for each $v' \in \mathrm{SVars}_e(v)$
`FROM (`$\mathrm{map}_e(v)$`) as TEMP_obj`
`LEFT JOIN` c
`ON TEMP_obj.ref_`v = c`.`c`_id AND TEMP_obj.val = 1.`

`AllInstances`-expressions

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

$\mathrm{map}_e(c\ \mathrm{.allInstances()})$=
```
SELECT c_id as res, 1 as val FROM c.
```

<>-expressions

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

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

  $\mathrm{map}_e(l <> r)$ =
  ```
  SELECT
    TEMP_left.res <> TEMP_right.res as res,
    1 as val
  FROM
    (map_e(l)) AS TEMP_left,
    (map_e(r)) AS TEMP_right
  ```
- $\mathrm{FVars}(l) \neq \emptyset$, $\mathrm{SVars}(r) \subseteq \mathrm{SVars}(l)$. Then,

  $\mathrm{map}_e(l <> r)$ =
  ```
  SELECT
    TEMP_left.res <> TEMP_right.res as res,
    CASE
      TEMP_left.val = 0 OR TEMP_right.val = 0
      WHEN 1 THEN 0
      ELSE 1 END as val,
    TEMP_left.ref_v as ref_v, for each v ∈ SVars_e(l)
  FROM (map_e(l)) AS TEMP_left
  [LEFT] JOIN (map_e(r)) AS TEMP_right
  [ON TEMP_left.ref_v = TEMP_right.ref_v,
      for each v ∈ SVars_e(l) ∩ SVars_e(r)].
  ```
- $\mathrm{FVars}(r) \neq \emptyset$, $\mathrm{SVars}(l) \subseteq \mathrm{SVars}(r)$. As before, but swapping the order of the elements in the left-join.
- $\mathrm{FVars}(l) \neq \emptyset$, $\mathrm{FVars}(r) \neq \emptyset$, $\mathrm{SVars}(l) \not\subseteq \mathrm{SVars}(r)$, and $\mathrm{SVars}(r) \not\subseteq \mathrm{SVars}(l)$. Then,

  $\mathrm{map}_e(l <> r)$ =
  ```
  SELECT
    TEMP_left.res <> TEMP_right.res as res,
    CASE
      TEMP_left.val = 0 OR TEMP_right.val = 0
      WHEN 1 THEN 0
      ELSE 1 END as val,
    TEMP_left.ref_v, for each v ∈ SVars_e(l),
    TEMP_right.ref_v, for each v ∈ SVars_e(r)
  FROM
    (map_e(l)) AS TEMP_left,
    (map_e(r)) AS TEMP_right
  ```

Exists-expressions

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

- $v \in \mathrm{FVars}(b)$ and $\mathrm{FVars}(e') = \emptyset$. Then
  ```
  SELECT
    COUNT(*) > 0 as res,
    1 as val
  FROM (map_e(b)) as TEMP_body
  WHERE TEMP_body.res = 1
  ```
- $v \in \mathrm{FVars}(b)$ and $\mathrm{FVars}(e') \neq \emptyset$. Then
  ```
  SELECT
    CASE TEMP_src.ref_v IS NULL
      WHEN 1 THEN 0
      ELSE TEMP.res END as res,
    1 as val,
    TEMP_src.ref_v′ as ref_v′,
      for each v′ ∈ SVars(s),
    TEMP_body.ref_v′ as ref_v′,
      for each v′ ∈ SVars(b) \ SVars(s) \ {v}
  FROM (map_e(s)) as TEMP_src
  LEFT JOIN (
    SELECT COUNT(*) > 0 as res,
      TEMP_body.ref_v′ as ref_v′,
        for each v′ ∈ SVars(b) \ {v}
    FROM (map_e(b)) as TEMP_body
    WHERE TEMP_body.res = 1
    GROUP BY TEMP_body.ref_v′,
      for each v′ ∈ SVars(b) \ {v}
  ) as TEMP_body
  ON TEMP_src.ref_v′ = TEMP_body.ref_v′,
    for each v′ ∈ SVars(s)
  ```
- $v \notin \mathrm{FVars}(b)$. Similarly, but the source and the body would need to be *joined* using a `JOIN`-clause.

B CORRIGENDUM

In [1, Section 4.3], in the second case considered in the definition of the mapping for `Exists`-expressions instead of:

- $v \in \mathrm{FVars}(b)$ and $\mathrm{FVars}(e') \neq \emptyset$. Then
  ```
  SELECT
    CASE TEMP_src.ref_v IS NULL
      WHEN 1 THEN 0
      ELSE TEMP.res END as res,
  ...
  LEFT JOIN (
    SELECT COUNT(*) > 0 as res,
      TEMP_body.ref_v′ as ref_v′,
        for each v′ ∈ SVars(b) \ {v}
  ```

it should read:

- $v \in \mathrm{FVars}(b)$ and $\mathrm{FVars}(e') \neq \emptyset$. Then
  ```
  SELECT
    CASE TEMP_body.ref_v IS NULL
      WHEN 1 THEN 0
      ELSE TEMP_body.res END as res,
  ...
  LEFT JOIN (
    SELECT COUNT(*) > 0 as res,
      TEMP_body.ref_v′ as ref_v′,
        for each v′ ∈ SVars(b)
  ```

And similar errors should be corrected in [1, Section 4.3], in the second case considered in the definition of the mapping for `ForAll`-expressions.

C OCL2PSQL REST API

Description. Given an OCL expression in the context of the data model `CarPerson`, it returns the SQL select-statement corresponding to this expression, according to the definition of the OCL2PSQL mapping.

URL. http://researcher-paper.ap-southeast-1.elasticbeanstalk.com/rest/map/carperson.

Parameters. It can take an optional parameter `sqlAsXmi`. If `sqlAsXmi` is `true`, then the output is returned in XMI-format; otherwise, it is returned as a string.

Input. It takes a JSON object as input. This object has two fields:

- `contentType`: the type of the input content. It can be either `expression/text` or `expression/xml`.
- `content`: an OCL expression either given as a string (when `contentType` is `expression/text`) or in XMI-format (when `contentType` is `expression/xml`).

Output. It returns a JSON object. This object has the following fields:

- `status`. The status of the response: `200` indicates success. `400` indicates invalid input.
- `contentType`. The type of the output: it can be either `statement/text` or `statement/xml`, depending on the value of `sqlAsXmi`; by default, its value is `statement/text`.
- `content`. The SQL select-statement corresponding to the *input*'s `content`, according to the definition of the OCL2PSQL mapping. This *output*'s content is given in the format indicated in `contentType`.
- `transformationTime`. The time spent in generating the *output*' content from the *input*'s content, measured in nanoseconds.

D OCL2PSQL-TEST REST API

Description. Given an OCL expression in `challenges.txt` and an SQL select-statement, it tests the correctness of the former as a translation of the latter, using the scenarios in `scenarios.txt`.

URL. http://researcher-paper.ap-southeast-1.elasticbeanstalk.com/rest/ttc.

Parameters. It takes two parameters: `stage` and `challenge`. The former is the number of a stage in the list `challenges.txt`. The latter is the number of a challenge of the stage given in `stage` in the list `challenges.txt`.

Input. It takes a JSON object as input. This object has two fields:

- `contentType`: the type of the input content. It should be `statement/xml`.
- `content`: an SQL-select statement (in XMI format).

Output. It returns a JSON object. It has only one field `scenarii` whose value is an array of JSON objects, one for each of the scenarios listed in `scenarios.txt` for the stage/challenge given in `stage`/`challenge`. Each object reports if the SQL select-statement given in `content` returns the expected results, as a translation of the OCL expression given in `stage`/`challenge`, when executed over the scenario corresponding to the object. In particular, each of the objects in the output contains the following fields:

- `scenario`. The number of the scenario.
- `status`. The result of the test: either `passed` or `failed`.
- `executionTime`. The execution time, measured in nanoseconds.