Extracting UML/OCL Integrity Constraints and Derived Types from Relational Databases

Valerio Cosentino, Salvador Martínez

AtlanMod team, Mines Nantes & INRIA & LINA, Nantes, France
{valerio.cosentino, salvador.martinez}@mines-nantes.fr

Abstract. Relational databases usually enforce relevant organizational business rules. This aspect is ignored by current database reverse engineering approaches which only focus on the extraction of the structural part of the conceptual schema. Other database elements like triggers, views, column constraints, etc. are not considered by those methods. As a result, the generated conceptual schema is incomplete since integrity constraints and derivation rules enforced by the database are not represented.

In this sense, this paper extends existing approaches by enriching the generated (UML) conceptual schema with a set of OCL integrity constraints and derivation rules inferred from the database schema. Our method has been implemented in a prototype tool for the Oracle database management system.

1 Introduction

Relational databases play a key role in most organizations storing the data they need for their business operations. To make sure the data is consistent, a set of integrity constraints is defined as part of the database schema. These constraints must be obviously aligned with the organizational policies and rules.

Therefore, it is very important these constraints evolve together with the own organization as it adapts to the changing needs of the market. Unfortunately, discovering and understanding the set of business rules enforced by a given database system is a time-consuming and error-prone activity since it implies querying/browsing the database dictionary to extract and analyze the retrieved data. Note that, even if SQL is a standard language, each vendor presents slight variations on the SQL language they support and use a completely different structure for their data dictionary which makes the process different for each database management system (DBMS).

To facilitate the comprehension of the enforced rules and their evolution, we believe the rules must be described using an homogeneous representation and at a higher-abstraction level. In this sense, this paper presents a new model-based reverse engineering approach able to extract a conceptual schema (CS) out of a running database where the CS is expressed as an UML class diagram extended with a set of Object Constraint Language (OCL) expressions to represent the integrity constraints and derivation rules contained in the database schema. Each OCL expression is the direct translation at the conceptual level of either one of the database constraints (e.g., CHECK constraints, keys or even constraints enforced by means of triggers) or view definitions. To the best of our knowledge, ours is the first reverse engineering method to cover these two aspects.

Without loss of generality, we particularized our method for the Oracle DBMS. Moreover, our solution is based on the principles of Model Driven Engineering (MDE) which facilitates reusing the plethora of available model-based tools for manipulating and processing the obtained models.

This paper is structured as follows: Section 2 presents the state of the art; Section 3-5 describes the proposed approach; Section 6 concludes this paper.

2 State of the Art

The integration of conceptual models and database applications is a research domain that has been explored extensively. On the one hand, approaches to translate UML class diagrams to database schemas and the embedded OCL constraints into integrity constraints have been proposed ([1], [2]). On the other hand, the extraction of class diagrams out of database schemas has been largely studied as well ([3], [4], [5], [6]). Nevertheless, these works do not cover the views or constraints (beyond primary and foreign keys) also included in the schema. The possibility of representing database views within UML models has been analysed in [7]. There, the author shows how the notion of relational database views can be correctly expressed as derived classes in UML using OCL constructs and relying on the expressiveness of OCL as query language[8], but no method is proposed to automatically infer those OCL expressions from the actual database view definition.

Reverse engineering of database constraints is partially covered in [9], where an approach for rephrasing constraints using the Semantics of Business Vocabulary and Business Rules (SBVR) notation is presented. Nevertheless, only a small subset of column constraints (CHECK and NOT NULL) is supported. Our work supports a much richer set of database constraints (including constraints enforced by means of triggers) and also view definitions.

3 Approach Overview

Our reverse engineering method (shown in Fig. 1) is composed by two steps: Model Extraction and Constraint Extraction.

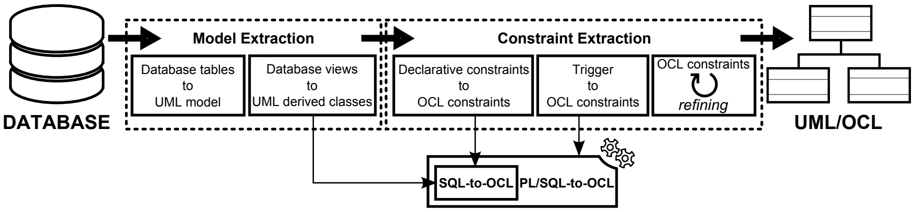

DATABASE
Model Extraction
Database tables to UML model
Database views to UML derived classes
Constraint Extraction
Declarative constraints to OCL constraints
Trigger to OCL constraints
OCL constraints refining
UML/OCL
SQL-to-OCL
PL/SQL-to-OCL

Fig. 1. Discovery process overview

Model Extraction focuses on the extraction of the structural elements of the schema. It is composed by two operations respectively dedicated to create the UML classes and associations corresponding to the database tables and their relations, and to extend the obtained model by adding a set of derived classes to represent the database views.

Constraint Extraction focuses on inferring the OCL expressions required to complement the UML model. Substeps of this method cover the declarative constraints (CHECK, UNIQUE,...) and the analysis of triggers [1] since, beyond other applications, triggers can also be used to enforce complex integrity constraints. Key elements in this step are the SQL-to-OCL and PL/SQL-to-OCL transformations, which are used to map SQL and PL/SQL constructs to OCL. Since triggers often merge SQL and procedural code, SQL-to-OCL is included in PL/SQL-to-OCL.

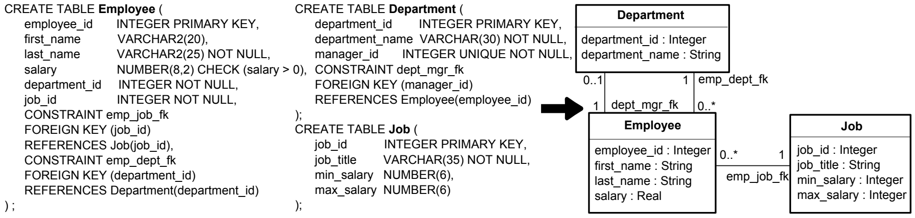
In the following, these steps are described in detail.

4 Model Extraction

The Model Extraction phase translates tables and views into an equivalent set of classes and associations in an UML class diagram.

As typically done in existing approaches, each table generates a class (or an association class) in the CS and table columns (except for foreign keys) are mapped into attributes of the corresponding class. Foreign keys are used to create associations between the classes, while the corresponding cardinalities can be calculated by performing SQL queries on the data stored in the database[10].

The type of the attributes depends on the type of the columns. Character data types (CHAR(n), VARCHAR2(n), etc.) are transformed to String types. Number and datetime data types (Integer, Float, Date, etc.) are transformed into their equivalent UML types: Integer, Real and Date. The Number(precision, scale) is transformed into an Integer data type when precision is zero and into a Real type otherwise. Other Oracle data types can be represented by defining new data types in the UML model.



```
CREATE TABLE Employee (
    employee_id     INTEGER PRIMARY KEY,
    first_name      VARCHAR2(20),
    last_name       VARCHAR2(25) NOT NULL,
    salary          NUMBER(8,2) CHECK (salary > 0),
    department_id   INTEGER NOT NULL,
    job_id          INTEGER NOT NULL,
    CONSTRAINT emp_job_fk
    FOREIGN KEY (job_id)
    REFERENCES Job(job_id),
    CONSTRAINT emp_dept_fk
    FOREIGN KEY (department_id)
    REFERENCES Department(department_id)
) ;
```

```
CREATE TABLE Department (
    department_id     INTEGER PRIMARY KEY,
    department_name  VARCHAR(30) NOT NULL,
    manager_id      INTEGER UNIQUE NOT NULL,
    CONSTRAINT dept_mgr_fk
    FOREIGN KEY (manager_id)
    REFERENCES Employee(employee_id)
);
CREATE TABLE Job (
    job_id        INTEGER PRIMARY KEY,
    job_title     VARCHAR(35) NOT NULL,
    min_salary  NUMBER(6),
    max_salary  NUMBER(6)
);
```

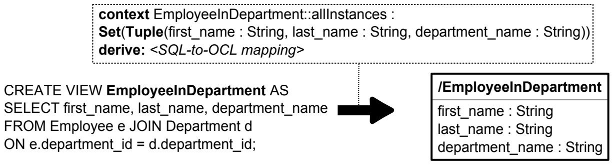
Fig. 2. Database schema to conceptual schema

As an example, Figure 2, shows the mapping of a simple human resources database system. Tables *Employee*, *Job* and *Department* are translated into the equivalent UML

[1] The SQL standard also includes a CREATE ASSERTION statement to specify complex constraints but none of the major database vendors support it

classes. An *Employee* in the UML model is defined by the attributes *employee ID*, *first name*, *last name* and *salary*. These attributes are derived from columns where no foreign key constraints are defined. The foreign keys are translated into UML associations; therefore an *Employee* has one *Job* (i.e., *emp_job_fk*) and belongs to one *Department* (i.e., *emp_dept_fk*). Accordingly to the previous mappings, a *Department* is described by an unique *department_id*, its *name* and the manager leading it (i.e., *dept_mgr_fk*). Finally, a *Job* is depicted by an unique *job_id*, by its *title* and the corresponding *maximum* and *minumum* salaries for that job.

Besides this basic process, the model extraction phase also creates an additional class for each database view. In this case, the UML class is derived and has as attributes the names and corresponding types of the columns that are selected in the view definition. The derivation rule for the class is created by translating the view select expression as described in the next section.



```
context EmployeeInDepartment::allInstances :
Set(Tuple(first_name : String, last_name : String, department_name : String))
derive: <SQL-to-OCL mapping>
```

```
CREATE VIEW EmployeeInDepartment AS
SELECT first_name, last_name, department_name
FROM Employee e JOIN Department d
ON e.department_id = d.department_id;
```

/EmployeeInDepartment
first_name : String last_name : String department_name : String

Fig. 3. Mapping of a database view

In Fig. 3, the mapping of a view is depicted. The view represents the projection of *Employees* (*first* and *last* names) per *Department*, that is identified by its *name*. The columns in the SELECT clause (i.e., *first_name*, *last_name* and *department_name*) are the attributes of the UML derived class; while their types (i.e., String in this case) are derived by mapping the built-in data type VARCHAR2 used to define those columns.

5 Constraint Extraction

Once we have generated the UML model we can enrich it with the set of OCL expressions to represent the constraints and derivation rules in the database schema.

5.1 Declarative Integrity Constraints to OCL constraints.

This step covers the declarative integrity constraints specified within the table creation statement. The context for all these constraints is the class representing the table in the UML diagram.

Figure 4 shows the patterns used to generate the OCL constraints corresponding to the PRIMARY KEY, UNIQUE, NOT NULL and CHECK constraints[2].

[2] Note that due to the high expressiveness of the OCL language, different OCL expressions can represent the same semantic constraint so other equivalent alternatives are also possible

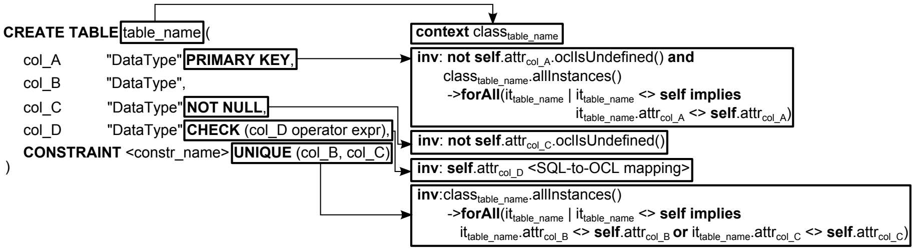

CREATE TABLE table_name (
col_A "DataType" PRIMARY KEY,
col_B "DataType",
col_C "DataType" NOT NULL,
col_D "DataType" CHECK (col_D operator expr),
CONSTRAINT <constr_name> UNIQUE (col_B, col_C)
)
context class_table_name
inv: not self.attr_col_A.oclIsUndefined() and
class_table_name.allInstances()
->forAll(it_table_name | it_table_name <> self implies
it_table_name.attr_col_A <> self.attr_col_A)
inv: not self.attr_col_C.oclIsUndefined()
inv: self.attr_col_D <SQL-to-OCL mapping>
inv:class_table_name.allInstances()
->forAll(it_table_name | it_table_name <> self implies
it_table_name.attr_col_B <> self.attr_col_B or it_table_name.attr_col_C <> self.attr_col_C)

Fig. 4. Mapping of Declarative integrity constraints

The name of the table (i.e., table_name) and columns (i.e., col_A) are mapped respectively to the corresponding UML class (i.e., $\text{class}_{\text{table\_name}}$) and attributes (i.e., $\text{attr}_{\text{col\_A}}$). In addition, the name of the table in lowercase is used as name of the iterator within the corresponding OCL operations.

5.2 SQL-to-OCL transformation

This section describes the mapping between SQL SELECT statements and the equivalent OCL expressions. In particular we describe the mappings for SQL projections, selections, joins, functions, group by and having clauses. These mappings are needed to create the derivation rules for views as described above and to be able to extract constraints implemented as part of trigger definitions as explained in the next subsection.

In the following, we present the list of mappings.

Projection and Selection. Projections and selections in SQL rely on SELECT, FROM and WHERE clause. The mapping of a generic projection/selection is shown in Fig. 5.

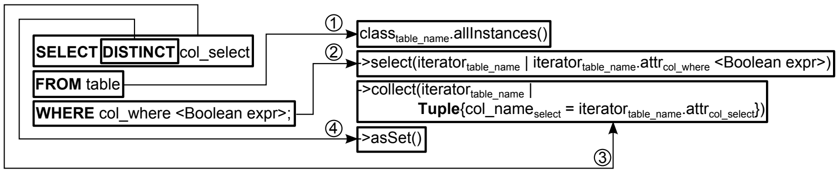

SELECT DISTINCT col_select
FROM table
WHERE col_where <Boolean expr>;
①
class_table_name.allInstances()
②
->select(iterator_table_name | iterator_table_name.attr_col_where <Boolean expr>)
->collect(iterator_table_name |
Tuple{col_name_select = iterator_table_name.attr_col_select})
④
->asSet()
③

Fig. 5. Mapping of projections/selections

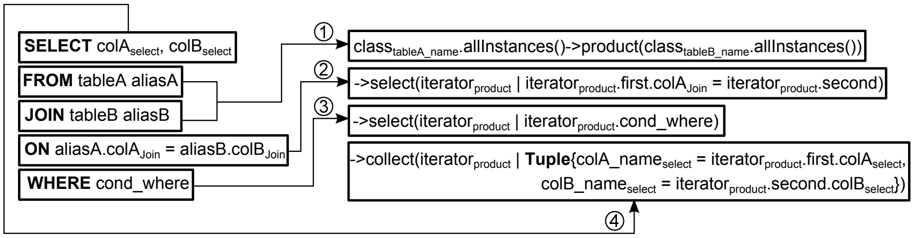
1. The mapping starts by translating the FROM clause. This is done by selecting all instances (i.e., method *allInstances()*) of the class in the CS that corresponds to table in the FROM.
2. The WHERE clause, if defined, is translated into an OCL *select* iterator. The condition in the *select* iterator is created by translating the conditions in the WHERE

clause. The mapping is basically a direct mapping since the references to the column names in the WHERE are replaced by the corresponding attribute and association names. SQL functions are translated into their OCL counterparts (if existing, otherwise new OCL operations must be previously defined, e.g., see [11]).

3. The SELECT clause is translated into an OCL *collect* iterator that creates a collection of objects according to the structure defined in the Tuple definition. Each field in the Tuple corresponds to a column in the SELECT clause. Fields are initialized with the value of the corresponding attributes.
4. Finally, the DISTINCT clause might be used in conjunction with a SELECT statement to return only the different (i.e., distinct) values in a given table. This clause is mapped adding the operation *asSet()* after the mapping of the SELECT clause.

Join. In SQL, the JOIN operation combines the values of two or more tables. Our transformation covers both inner and outer joins, but for the sake of conciseness we focus the explanation on the mapping for inner joins.

The inner join is by far the most common case of joining tables. Giving two tables *a* and *b* and according to the join conditions, it returns the intersection of the two tables. The mapping of a generic inner join is shown in Fig. 6.



SELECT colA$_{select}$, colB$_{select}$

FROM tableA aliasA

JOIN tableB aliasB

ON aliasA.colA$_{Join}$ = aliasB.colB$_{Join}$

WHERE cond_where

class$_{tableA\_name}$.allInstances()->product(class$_{tableB\_name}$.allInstances())

->select(iterator$_{product}$ | iterator$_{product}$.first.colA$_{Join}$ = iterator$_{product}$.second)

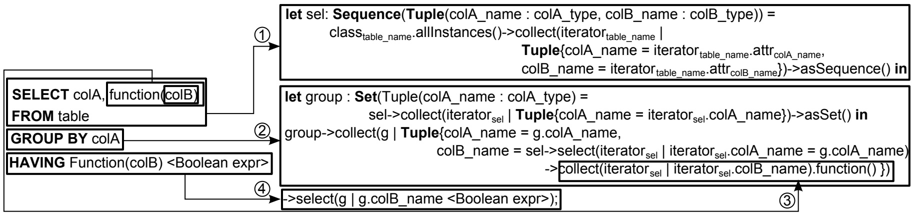
->select(iterator$_{product}$ | iterator$_{product}$.cond_where)

->collect(iterator$_{product}$ | **Tuple**{colA_name$_{select}$ = iterator$_{product}$.first.colA$_{select}$, colB_name$_{select}$ = iterator$_{product}$.second.colB$_{select}$})

Fig. 6. Mapping of inner joins

1. Firstly, we perform the Cartesian product of the population of all tables by retrieving all the instances of the corresponding classes in the CS and applying on them the Cartesian product (named *product* in OCL).
2. The join conditions are mapped to the body of a *select* operation, that iterates over the tuples of the Cartesian product (i.e., iterator$_{product}$), selecting those that satisfy the conditions. *First* and *second* are used to identify the classes in the Cartesian product (i.e., $class_{tableA\_name}$ and $class_{tableB\_name}$). Note that *colA* is compared with *second* (and not *second.colB*) since in the pattern we assume that *colA* is the name of the role of the association linking the two classes (i.e., the type of *colA* in the CS is $class_{tableB\_name}$).
3. In case the WHERE clause is defined (i.e., implicit joins or other conditions), a new *select* operation is created according to the patterns previously described.
4. If not all columns are selected, the collect operation is used as described in the previous pattern.

Group By, Having and Aggregate Functions. The GROUP BY clause is used to group the result-set of a given SELECT statement. Groups can be filtered by means of the HAVING clause. In Fig. 7, the mapping of GROUP BY and HAVING clauses is shown.



```
SELECT colA, function(colB)
FROM table
GROUP BY colA
HAVING Function(colB) <Boolean expr>
```

```
let sel: Sequence(Tuple(colA_name : colA_type, colB_name : colB_type)) =
        class_table_name.allInstances()->collect(iterator_table_name |
                                          Tuple{colA_name = iterator_table_name.attr_colA_name,
                                          colB_name = iterator_table_name.attr_colB_name})->asSequence() in
```

```
let group : Set(Tuple(colA_name : colA_type) =
              sel->collect(iterator_sel | Tuple{colA_name = iterator_sel.colA_name})->asSet() in
group->collect(g | Tuple{colA_name = g.colA_name,
                         colB_name = sel->select(iterator_sel | iterator_sel.colA_name = g.colA_name)
                                     ->collect(iterator_sel | iterator_sel.colB_name).function() })
->select(g | g.colB_name <Boolean expr>);
```

Fig. 7. Mapping of group by and having clauses

1. We first translate the SQL FROM, JOIN and WHERE clauses according to the previous mapping patterns. The result is used to initialize the variable *sel*.
2. The tuples in *sel* are then processed to create the grouped result set *group* that maps the GROUP BY clause. In short, *group* is created by iterating on the *sel* tuples and collecting together those tuples with an identical value on the attributes corresponding to the columns in the GROUP BY clause.
3. If an aggregate function is applied on one of the columns in the SELECT clause, an equivalent OCL operation (see [11] for more on aggregate functions in OCL) is added at the end of the GROUP BY clause mapping.
4. Finally, if the HAVING clause has been defined, it is translated into a *select* operation, where the body of the *select* is created by mapping the HAVING conditions.

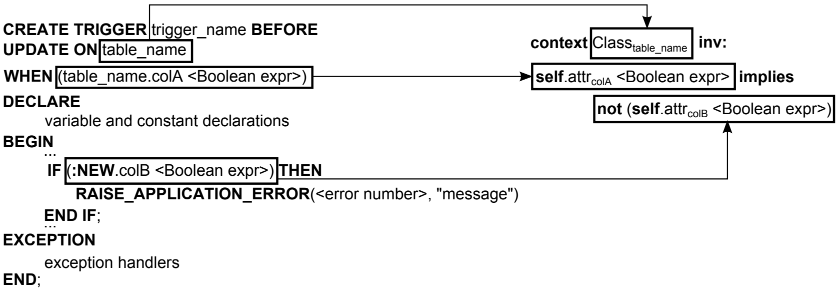
5.3 Triggers to OCL constraints.

A second source of integrity constraints are triggers that, among other application scenarios, can be in charge of enforcing complex constraints that go beyond the expressiveness of CHECK table constraints. In particular, we focus our analysis on row-level triggers, that are generally used to check the validity of the data and for this reason they are often used to embed business rules at database level.

A trigger is a procedural code that is automatically executed in response to certain events applied to a table. It is composed by the triggering event, an optional trigger condition (i.e., WHEN clause in Figure 8) and the triggered action. The triggering event is a SQL statement, database or user event. The restriction condition is a boolean expression related to the event clause. A triggered action is a PL/SQL block that contains SQL and procedural code to be run when the triggering event occur and the restriction condition is true. Finally, triggers can be executed *instead of*, *before* or *after* performing the triggering event.

To distinguish triggers enforcing a business rule from other kinds of triggers (e.g., devoted to log actions) we use the following heuristic: all triggers embedding in their action section a PL/SQL statement raising an exception defined by the user are classified as *constraint-enforcing-triggers*. For each of such triggers, an OCL invariant is generated. The context of the invariant is the UML class corresponding to the table where the trigger is defined; while its body is composed by the trigger restriction condition, if defined, and the output of the PL/SQL-to-OCL transformation described below.

PL/SQL-to-OCL Transformation The PL/SQL-to-OCL transformation is used to extract OCL constraints out of PL/SQL blocks (in our case part of trigger definitions). A stored procedure consists of three sections (see Fig. 8): an optional declaration section, identified by the keyword DECLARE and used to define variables; an execution section, wrapped between the keywords BEGIN and END and containing PL/SQL and SQL statements; and finally an optional exception handling section, used to handle the run-time error and identified by the keyword EXCEPTION. The OCL constraints are extracted starting from an analysis of the conditional statements that raise user exceptions in the execution section.



```
CREATE TRIGGER trigger_name BEFORE
UPDATE ON table_name
WHEN (table_name.colA <Boolean expr>)
DECLARE
      variable and constant declarations
BEGIN
      ...
       IF (:NEW.colB <Boolean expr>) THEN
          RAISE_APPLICATION_ERROR(<error number>, "message")
      END IF;
      ...
EXCEPTION
      exception handlers
END;
```

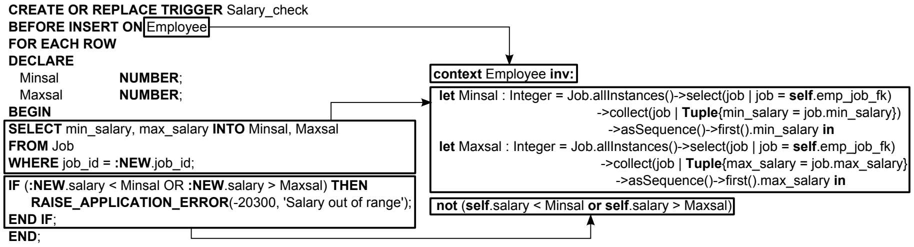
```
context Class_table_name inv:
self.attr_colA <Boolean expr> implies
      not (self.attr_colB <Boolean expr>)
```

Fig. 8. Mapping of constraints enforced by triggers

PL/SQL allows defining user exceptions in two ways. One is to override an already-defined exception. In this case, the exception must be declared in the declaration section and raised explicitly in the execution section using the statement RAISE. The other way concerns the command RAISE_APPLICATION_ERROR, that raises an user-defined exception that is used to communicate an application-specific error back to the user. Both kinds of exceptions are generally business relevant, since they represent a violation of the company's business and not a technology issue. Since these exceptions are raised explicitly, they are generally nested in conditional statements, that check if the business constraints are violated or not.

For each exception, the conditions triggering the exception are mapped to an equivalent OCL expression. Note that these conditions may include variables calculated from previous SQL queries in the trigger execution section. If that case, those expressions are also processed according to the SQL-to-OCL mappings described before.

In Fig. 9, we show the mapping of the *Salary_check* constraint. This trigger raises an exception when the salary of a new employee for a given job is outside the salary range expected for that job. The minimum and maximum salaries are stored in the



```
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT ON Employee
FOR EACH ROW
DECLARE
  Minsal        NUMBER;
  Maxsal        NUMBER;
BEGIN
SELECT min_salary, max_salary INTO Minsal, Maxsal
FROM Job
WHERE job_id = :NEW.job_id;
IF (:NEW.salary < Minsal OR :NEW.salary > Maxsal) THEN
    RAISE_APPLICATION_ERROR(-20300, 'Salary out of range');
END IF;
END;
```

```
context Employee inv:
 let Minsal : Integer = Job.allInstances()->select(job | job = self.emp_job_fk)
                                  ->collect(job | Tuple{min_salary = job.min_salary})
                                  ->asSequence()->first().min_salary in
 let Maxsal : Integer = Job.allInstances()->select(job | job = self.emp_job_fk)
                                  ->collect(job | Tuple{max_salary = job.max_salary}
                                   ->asSequence()->first().max_salary in
 not (self.salary < Minsal or self.salary > Maxsal)
```

Fig. 9. Example of a constraint enforced by a trigger

table *Job* (Fig. 2). They are retrieved using two variables (i.e., *Minsal* and *Maxsal*) by means of a SELECT INTO clause. The mapping of this clause extends the mapping for SQL projections seen before. Each variable is mapped to an OCL *let* expression, where the OCL variable takes the name and the type of the SQL variable defined in the DECLARE section of the trigger. Variable values are initialized with the result of the SQL-to-OCL mapping for the SQL query expression. Since in PL/SQL, SELECT INTO statements can only return one single row, the first elements of both projections are returned using the OCL operation *first()*. Finally, the values of the two variables are compared according to a boolean expression that maps the negation of the PL/SQL if-statement condition.

Refining OCL Constraints Constraints enforced by a trigger should be linked to the event that may fire the trigger. For instance, the *Salary* constraint extracted in Fig. 9 should only be checked when creating new employees not when updating them (there could be another trigger in charge of checking that updates also respect the constraint but this is not something we can deduce from this single trigger). This corresponds to the concept of creation-time constraints[12]. Therefore, when extracting the constraints we use the stereotype mechanism from the UML language to annotate each constraint with information of the events that apply to it.

Only when we find the set of equivalent constraints that altogether are annotated with all the relevant events for the kind of condition enforced by the constraint set, we can merge them and define the resulting constraint as a standard OCL invariant.

A complete procedure to identify semantically-equivalent constraints and to analyze which events can violate the constraint is out of the scope of this paper. They are needed respectively to create a single constraint annotated with the union of events for each individual constraint, and to guess if the set of database triggers are sufficient to ensure that the generated constraint should always hold (and not only at the creation, update or deletion time). The aforementioned procedure would rely on [13] for the detection of redundant constraints and [14] for the derivation of relevant events for each constraint.

6 Conclusion and Future Work

In this paper, we have presented a model-based reverse engineering approach for extracting an UML/OCL model out of a database implementation in order to facilitate the comprehension of the integrity constraints and derivation rules embedded in the database. As a proof of concept, the approach has been implemented as a model-based prototype[3] integrated in the Eclipse platform.

As further work, we would like to extend the kind of constraints we can infer by first extending the OCL language with additional libraries needed to properly represent time and calendar-based constraints. In addition, we would like to focus our analysis on other kind of triggers and their relations to discover more complete business rules. Finally, in order to have a global view of the current organization policies enforced in an Information System, we plan to compare (e.g., for consistency) and merge the obtained constraints with those ones enforced in other components of the system (to be extracted with other reverse engineering approaches, e.g., [15]).

References

1. Demuth, B., Hußmann, H.: Using UML/OCL Constraints for Relational Database Design. In: UML. (1999) 598–613
2. Demuth, B., Hußmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: UML. (2001) 104–117
3. Premerlani, W.J., Blaha, M.R.: An approach for reverse engineering of relational databases. In: WCRE. (1993) 151–160
4. Chiang, R.H.L., Barron, T.M., Storey, V.C.: Reverse Engineering of Relational Databases: Extraction of an EER Model from a Relational Database. Data & Knowledge Engineering **12**(2) (1994) 107–142
5. Andersson, M.: Extracting an entity relationship schema from a relational database through reverse engineering. In: ER. (1994) 403–419
6. Ramanathan, S., Hodges, J.: Extraction of object-oriented structures from existing relational databases. ACM Sigmod Record **26**(1) (1997) 59–64
7. Balsters, H.: Modelling database views with derived classes in the UML/OCL-framework. In: UML. (2003) 295–309
8. Akehurst, D.H., Bordbar, B.: On querying uml data models with ocl. In: UML. (2001) 91–103
9. Chaparro, O., Aponte, J., Ortega, F., Marcus, A.: Towards the Automatic Extraction of Structural Business Rules from Legacy Databases. In: WCRE. (2012) 479–488
10. Yeh, D., Li, Y., Chu, W.C.: Extracting entity-relationship diagram from a table-based legacy database. Journal of Systems and Software **81**(5) (2008) 764–771
11. Cabot, J., Mazón, J.N., Pardillo, J., Trujillo, J.: Specifying aggregation functions in multidimensional models with OCL. In: ER. (2010) 419–432
12. Olivé, A.: Integrity Constraints Definition in Object-Oriented Conceptual Modeling Languages. In: ER. (2003) 349–362
13. González, C.A., Buttner, F., Clarisó, R., Cabot, J.: Emftocsp: A tool for the lightweight verification of emf models. In: FormSERA. (2012) 44–50
14. Cabot, J., Teniente, E.: Determining the structural events that may violate an integrity constraint. In: UML. (2004) 320–334
15. Cosentino, V., Cabot, J., Albert, P., Bauquel, P., Perronnet, J.: A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application. In: RuleML. (2012) 17–31

[3] http://docatlanmod.emn.fr/IntegrityConstraints2OCL/intro.html