Journal of Theoretical and Applied Information Technology
28th February 2018. Vol.96. No 4
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

EXTRACTING UML MODELS AND OCL INTEGRITY CONSTRAINTS FROM OBJECT RELATIONAL DATABASE

[1]TOUFIK FOUAD, [2]BAHAJ MOHAMED

[1] PhD, LITEN Laboratory, University Hassan I, FSTS Settat, Morocco

[2] Professor, LITEN Laboratory, University Hassan I, FSTS Settat, Morocco

E-mail: [1]toufik.fouad@gmail.com, [2]mohamedbahaj@gmail.com

ABSTRACT

Database reverse engineering is the process of extracting and transforming database metadata to a rich set of models. These models must be able to describe data structure at different levels of abstraction, starting from physical to conceptual schema. The obtained schema may be used to ease, among others, database structure update, evolution and maintenance. In the past few years, the object oriented construct was merged into relational database. Nevertheless, a few methods of object relational database (ORDB) reverse engineering was presented. In this sense, the main goal of this article is to present an approach of database reverse engineering which cover the transformation of new added object construct. At the end of transformation we obtain a conceptual schema (CS) expressed as UML class diagram. The returned CS is extended with a set of OCL (Object Constraint Language) clauses which represent at a higher level of abstraction, the database integrity constraints. We provide a program that implements our approach for ORACLE 11g database management system.

Keywords: *UML, OCL, ORDB, SQL, Reverse Engineering.*

1. INTRODUCTION

Reverse engineering a piece of software consists in reconstructing its functional and technical documentation, starting mainly from the source text of the programs. Recovering these specifications is generally intended to convert, restructure, maintain or extend old applications [1]. Database reverse engineering is the process of generating a description of database content in high level terms that are natural for users. The process produce a schema expressed in a conceptual modelling notation.
The conceptual schema represent an abstract definition of database tables and their relationships by using a human oriented natural language, independent of any implementation, respecting clarity and simplicity criteria. This CS can facilitate the comprehension of the data structure, business rules implemented as integrity constraints and triggers. The CS also may help for integration, evolution, data migration and system reuse.
Vendors like Oracle, Microsoft and IBM have moved object-oriented database features (classes, encapsulation, inheritance …) into their relational DBMSs to win the challenge of representing complex data. Therefore a lot of companies use the hybrid solution for database development which adopt the new object constructs. To facilitate the comprehension of the database and the enforced rules and their evolution, we believe the database tables and the rules must be described using an homogeneous representation and at a higher abstraction level. In this sense, this paper present a new reverse engineering approach capable of extracting a conceptual schema (CS) from a running database where the obtained CS is expressed as an UML class diagram. The class diagram present database tables as classes and table relationships as associations. Business rules implemented in the database as integrity constraints and triggers are transformed to OCL expressions. Each OCL expression is a transformation at the conceptual level of either one of the database constraints (CHECK constraints, constraints enforced by triggers).
Furthermore our method has been implemented in a prototype tool for ORACLE 11g database management system (DBMS) one of relational DBMS that support new object oriented features.
This paper is organized as follows: section 2 presents the state of the art; section 3 presents an

overview of our approach; section 4-5 describe in detail our approach; section 6 present implementation and validation; section 7 conclude this paper and discuss future work.

2. STATE OF THE ART

Database reverse engineering is a well-studied subject. Much of the research conducted in this field has focused on the relational model. Several approach have been proposed to extract a conceptual schema from relational database. [2] adopts Object Modeling Technique (OMT) notation for modeling data out of running database, [3] propose a new methodology for extracting an Extended Entiy-Relationship (EER) from Relational Database (RDB), [4] for extracting a conceptual schema from RDB, the author present a method based on analysis of data manipulation statements in the code of an application using a relational database schema, [5] maps a relational schema into an object-oriented schema by taking into consideration various types of RDB design optimizations. [6] presents a method for translating a relational database to an Object Relationship Model(ORM) ,[7] show how the notion of a relational database view can be correctly expressed as a derived class in UML/OCL,[8] presents a method to define OCL as query language for UML data models, [9] present an approach for automatically extracting structural business rules from legacy databases and it application on a specific legacy system, [10] presents a database reverse engineering approach that support extracting an extended entity-relationship diagram from a legacy database based on tables, the work in [11] addresses database reverse engineering by extracting the extended entity-relationship schema from relational schema, [12] presents a model-based reverse engineering approach able to extract a Conceptual Schema (CS) expressed as an UML class diagram extended with a set of Object Constraint Language (OCL) from RDB. For forward engineering of Object Relational Database (ORDB), [13] presents a method that defines new UML model elements to design the object relational database, [14] describes a method of UML models transformation, the method contains two phases; the first one present the transformation of class diagram (static aspect) to database schema, the second transform state chart diagram (dynamic aspect) to database triggers. For reverse engineering [15] describe an approach to recover schemas from ORDB. The main objective of this approach is to recover conceptual schemas, represented as UML diagrams, based on the analysis of the data dictionary. In comparison, our approach have some similarity in model extraction phase with [15].

3. APPROACH OVERVIEW

Our reverse engineering method has two main phases. The main goal of the first one is model extraction; in this step we focus on the structural part of Object Relational Schema, we retrieve all needed information from the database dictionary, such as User Defined Type UDT (NAME, SUPER TYPE, Final or not, INSTANTIABLE …), types tables, table dependencies, we also identify collections (NESTED TABLE, VARRAY) and simple attributes basic types (CHAR, DATE, VARCHAR2 …), in the next section we present in detail the transformation steps from ORACLE object constructs to UML models.

The second phase is constraints extraction, at the beginning we focus on declarative integrity constraints like (PRIMARY KEY, UNIQUE, CHECK …). These constraints are transformed to OCL expressions to enforce the generated UML Class Diagram obtained in the first phase. Triggers play very important role to define complex business rules and to enforce integrity constraints. Therefore it's necessary to present all possible triggers as OCL expressions. Since triggers merge between SQL queries and procedural code PL/SQL, we transform (SQL, PL/SQL) code to OCL expressions. The above figure describes our approach with two main phases mentioned before:

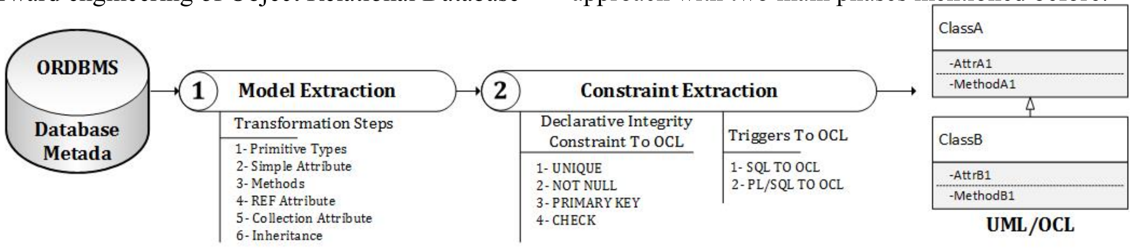


Figure 1 : Approach Overview

4. MODEL EXTRACTION

The model extraction phase transforms User Definition Type (UDT) to an equivalent set of class and associations in a UML class diagram. This phase contains a list of steps and rules transformation.

Each UDT generates a class in the CS; the stereotypes *abstract* and *final* are added from *notInstantiable* and *final* respectively if they are mentioned in the creation type statement.

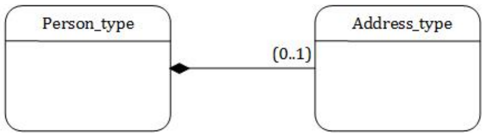
Primitive data type transformation: Integer, Float and Date types are mapped to Integer, Real and Date UML types respectively. The Number (*precision*, *Scale*) is transformed into an Integer data type when precision is zero and into a Real type otherwise. Characters data type (CHAR (n), VARCHAR2 (n), etc.) are transformed to String type, in the case of CHAR data type, the length attribute should be equal *n*, on the other hand in the case of VARCHAR data type, the length attribute cannot exceed *n*. to verify these conditions, we use OCL constraints.

UDT Attribute transformation: UDT in ORDB can have one of four different representations (simple UDT, REF to simple UDT, collection of UDT, collection of REF to UDT)

The example below represent the create object statement for a simple UDT (*person_type*) with nested UDT attribute (*address_type*):

Create type address_type as object(…)
Create type person_type as object(address address_type ...)

The transformation results are shown in the class diagram below:



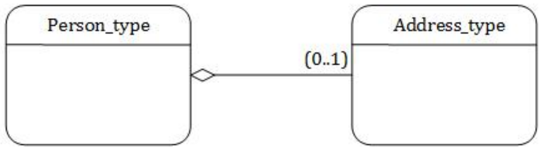
The transformation generate a new composition relationship, a strong association between the two types (*address_type*, *person_type*) with ***(0..1)*** multiplicity in address_type entity side.

By using references to different types in the database, another representation of UDT is possible in ORDB. References permit to create complex object and retrieving data easily without using JOINs between tables. References also allow creating weak relationship between two types

The example shown before present a strong relationship between *person_type* and *address_type*, by declaring the attribute address as reference to *address_type*, the transformation generate an aggregation relationship with the same properties mentioned before.

Create type address_type as object(…)
Create type person_type as object(address REF address_type ...)

The transformation results are shown in the class diagram below:

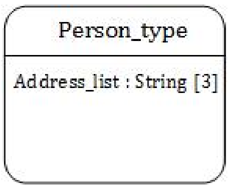


Oracle supports *varray* and *nested table* collection data types. *Varray* is an ordered set of data elements, all elements of a given *varray* are the same data type or a subtype of the declared one, *varray* is a limited collection, the maximum number of items is specified in the creation statement. *Nested table* is an unordered set of data elements, all of the same datatype, nested table is an unlimited collection.

Attributes defined as collection (*nested table*, *varray*) of primitive types (integer, float, varchar …), are transformed to list attribute in the CS. Its minimum length is zero and its size equals the length of the *varray*, in the case of nested table the size is unlimited. The example above show the transformation of an UDT which contains tow collection attribute of primitive type.

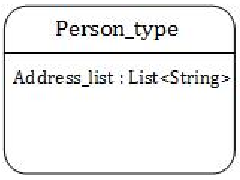
- Case 1

create type address_type as varray(3) of varchar2
create type person_type as object(address_list address_type,…)



- Case 2

create type address_type as table of varchar2
create type person_type as object(address_list address_type,…)

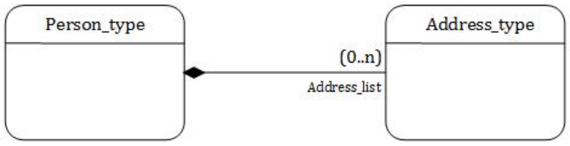


The attribute *address_list* defined in person_type is a collection of varchar2, the mapping process of the target attribute generate a String collection inside *person_type* class in the CS, limited in the case of *varray(3)* (case 1) where 3 is the maximum length of the collection and unlimited in the case of *nested table* (case 2).

When attribute is a collection of UDT, the transformation gives a rise to a new composition, the example above show the strong relationship between the two types:

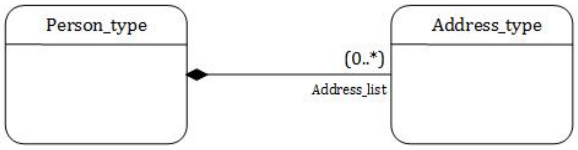
- Case 1

create type address_type as object(city varchar,..)
create type address_type_varray as varray(n) of address_type
create type person_type as object(address_list address_type_varray,…)



- Case 2

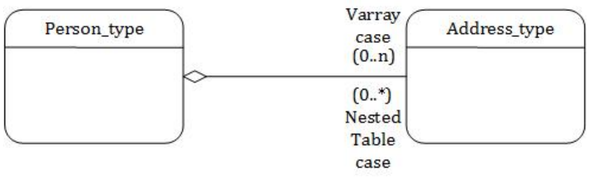
create type address_type as object(city varchar,..)
create type address_type_nested as table of address_type
create type person_type as object(address_list address_type_nested,…)



The attribute *address_list* defined in *person_type* is a collection of *address_type*, the transformation generate a composition relationship between *person_type* and *address_type* with ***(0..*)*** multiplicity in *address_type* side and *address_list* as role name in case of *nested table*, in the case of *varray* we change the multiplicity to ***(0..n)*** where ***n*** is the maximum length of the collection.

When attribute is a collection of REF to UDT, we keep all properties and result of the previous transformation and we change the association between the two types to weak relationship and change composition association to aggregation.

Create type address_ref as object(addr_r REF address_type)
Create address_varray as varray(n) of address_ref;
Create address_nestes as table of of address_ref;
create type person_type as object(address_list1 address_varray, address_list2 REF address_nested,…)

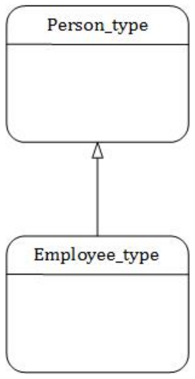


- Inheritance Transformation

Oracle offers the mechanism of inheritance which connects subtypes in a hierarchy to their supertypes. Subtypes automatically inherit attributes and methods of their parent type. Any attributes or methods updated in a supertype are updated in subtypes as well. The example above show the inheritance between two types, *person_type* and *employee_type*.

create type person_type as object(…) not final;
create type employee_type under person_type

Employee_type is a subtype of *person_type*, the transformation generate a generalization relationship between these classes in the CS.



- Methods Transformation

Database tables contains only data, objects can include the ability to perform operations on that data. Objects methods are functions or procedures declared in an object definition type to implement behavior wanted from objects of that type to perform. There are three general kinds of methods that can be declared in a type definition (member methods, static methods and constructor methods).

After retrieving information about method's (parameters and returned result), the transformation process map all kind of UDT's methods and add the same signature to the targeted class in the CS.

5. CONSTRAINT EXTRACTION

5.1. Declarative Integrity Constraints Transformation

Constraints are the rules enforced on data columns on table. If there is any violation between the constraint and the data action, the action (insert, update, delete) is aborted by the constraint. Constraints are used to ensure the accuracy and consistency of data stored in a typed table in object relational database.

To enrich the conceptual schema and make it complete and comprehensible, all these constraints must be included in the CS, in order to transform and present these constraints we use OCL (Object Constraint Language). The Object Constraint Language [15] is a textual specification language, designed especially for the use in the context of diagrammatic specification languages such as UML. OCL was always used to add well-formedness rules on both the model and the metamodel levels within UML. OCL is strongly connected to UML diagrams, as it is used as textual constraints within diagrams. OCL uses the elements defined in the UML diagrams, such as classes, methods and attributes. The language is based on types. Each OCL expression evaluates to a type either predefined by the language or defined by the model on which the expression is built.

To present all possible transformations of declarative integrity constraints applied to a specific object relational table we focus on the example below.

```
create type person_type as object(person_id int,
        passport_id int, name varchar(20), age int,
        city varchar(20));
create table person_table of person_type(person_id
        primary key,passport_id unique, name not
        null, city default 'casa', check (age<100) ;
```

The *DEFAULT* constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified. After transforming the targeted UDT to a class, now we have the possibility to use object concepts like constructors. Constructor is a special non-static member function of a class that is used to initialize objects of its class type. Constructors are invoked when initialization takes place, and are selected according to the rules of initialization. Constructors can assign values to any accessible fields or properties of an object at creation time. The example below present an initialization constructor which can assign default values to more than one attribute.

```
Public person_type (String city) {
        this.city=city;
}
Person_type person=new person("Casablanca");
```

The *NOT NULL* constraint enforces a field to always contain a value. This means that the insert or update of records not authorized without adding a value to this field. To present this constraint as an OCL condition in the CS we create an invariant that use the method *oclIsUndefined()* which return true if the value equal invalid or null. The invariant context is the class transformed from an UDT in model extraction phase.
The OCL instructions are:

```
Context person_type inv:
        Not self.name.oclIsUndefined()
```

The *UNIQUE* and *PRIMARY KEY* constraints both provide a guarantee for uniqueness for a column or a set of columns. A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on

it. A *primary key* column cannot contain *NULL* values.
The transformation of UNIQUE constraint is:

```
Context person_type inv:
 Person_type.allInstance()->
   forAll(person1,person2 | person1 <> person2
      Implies person1.passport_id <>
     person2.passport_id)
```

allInstances() is a feature associated with any type that returns the set of all instances of the given type. *forAll()* operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection.
As *PRIMARY KEY* constraint has a *UNIQUE* constraint we take the same transformation result of *UNIQUE* constraint and we add *oclIsUndefined()* method for each object in the returned collection retrieved by *allInstances()* to prevent the storage of null values

```
Context person_type inv:
  Person_type.allInstance()->
   forAll(person1,person2 | person1 <> person2
  Implies person1.person_id <> person2.person_id
  And not person1.person_id.oclIsUndefined())
```

CHECK constraints allow specifying a condition in each row in a table. Every *CHECK* constraint generate an OCL invariant with the same expression presented in the constraint body, the example below show the transformation of CHECK constraint:

```
Context class_name inv:
 Self.attribute_name.<condition SQL-TO-OCL Mappnig>

Context person_type inv:
            Self.age<100
```

In classical relational database RDB we use foreign keys to manage relationships between tables. To extract data we use multiple JOINs in queries to get the adequate result. In Object Relational Database ORDB, new concepts have been implemented to manage relationships like nested objects and references to row objects by using REFs and OIDs.
(OID) Object Identifier uniquely identifies row objects in object tables. A REF is a logical pointer or reference to a row object that we can construct from an object identifier. We can use the REF to obtain, examine or update the object and also we can change a REF so that it points to a different object of the same object type hierarchy or assign it a null value. REFs are Oracle database built-in types. REFs and collection of REFs model associations among object, particularly many-to-one relationships, thus reducing the need for foreign keys. REFs provide an easy mechanism for navigation between objects. The example below presents a relationship between two objects *command_type* and *client_type* using REF:

```
create type client_type as object(client_id int,name
           varchar(20),...);
create type command_type as object(command_id
           int,command_date,    client_ref    REF
           client_type);
```

We can constrain a column type, collection element, or object type attribute to reference a specified object table by using the SQL constraint subclause SCOPE IS when declaring REF. Scoped REF types require less storage space and allow more efficient access than unscoped REF types.
A REF can be scoped to an object table or of any subtype of the declared type. If a REF is scoped to an object table of a subtype, the REF column is effectively constrained to hold only references to instances of the subtype and its subtypes if existed in the table (mechanism of inheritance in ORDB).

```
create table client_table of client_type;
create table command_table(client_ref REF
           client_type SCOPE IS client_table);
```

The OCL instructions are:

```
Context command_type inv:
 Self.allInstances()->forAll(Command_type cmd |
 Client_type.allInstances()->collect(clt.getOID)
      ->exist(cmd.getClientRef))
```

As shown in the result above, the main context is the class *Command_Type*. We take each element in allInstances collection using forAll() function, to verify each instance of *command_type*. The next step is checking the existence of command client REF in the client dataset.

5.2. Sql To Ocl Transformation

This section presents the transformation steps from SQL SELECT statements to OCL expressions. In particular we describe the mapping for SQL projections, selections, methods, order by, group by and having clauses. This transformation is needed to extract constraints implemented as part of trigger definition.in the following we present examples of SQL statements transformation.

SELECT {DISTINCT | UNIQUE} colName, methodName()
FROM tableName
WHERE ({coleName, methodName} condition)
ORDER BY colName

The OCL instruction are:

```
Context className inv
  self.allInstances() ->
  select({colName | mehodName} condition)->
  collect(colName,methodName) ->
  asSet()->asSequence()
OR
 asOrderedSet()
```

The transformation starts by the *FROM* clause. *tableName* is the targeted class which transformed in Model Extraction phase to *className*. The context of the OCL expression is className which specifies the entity in the UML model for which the expression is defined. *allInstances()* method return the set of all instances of the given class.
The *WHERE* clause is transformed to an OCL select iterator which return a collection of all elements that validate the condition. The *colNames* in the *WHERE* clause are mapped to the corresponding attribute and association names. SQL functions are translated into their OCL counterparts (if existing otherwise new OCL operations must be previously defined [16]).
The *SELECT* clause is transformed to OCL collect iterator that creates a collection of objects according to the structure defined in the tuple definition. Each attribute present in the OCL expression corresponds to a column in the *SELECT* statement.
To retrieve different and unique data from a given table we use *DISTINCT*. This clause is mapped by adding *asSet()* method after transforming the *SELECT* clause.
ORDER BY is used to sort the result-set by one or more columns. To order data in OCL we use *asSequence()* which return an ordered collection. This collection may contain duplicates elements.
If *DISTINCT* and *ORDER BY* clauses are both present we can use the method *asOredered\Set()* to get an ordered collection with unique elements.

Inheritance is the mechanism that connects subtypes in a hierarchy to their supertypes. Subtypes automatically inherit the attributes and methods of their parent type. A subtype can be derived from a supertype either directly or indirectly through intervening levels of other subtypes. A supertype can have multiple sibling subtypes, but a subtype can have at most one direct parent supertype (single inheritance).
With object types in a type hierarchy, we can model an entity such as a *person_type*, and also define different specializing subtypes of *person_type* like *professor_type* and *student_type*. The example below show the creation statements of types mentioned before:

```
create type person_type as object(person_id,
          person_name varchar(20),…) not final;
Create type professor_type under person_type(…);
Create type student_type under person_type(…);
Create table person_table of person_type;
```

Person_type is declared *NOT FINAL* to create subtypes. Object relational table *person_table* is created to hold data of the three types.
The example below show the use of the function *value()* which help to select *professor_type* rows from *person_table*. *Value()* takes as its argument a correlation variable (table alias) associated with a row of an object table and returns object instances stored in the object table. The type of the object instances (include subtypes) is the same type as the object table.

```
Select value(p) from person_table p
          Where value(p) IS OF (professor_type);
```

The transformation of the inheritance query generate the OCL invariant as described below:

```
Context person_type inv:
          Self.allInstances()->select(p : Person_type
                    | p.oclIsTypeOf(professor_type))
```

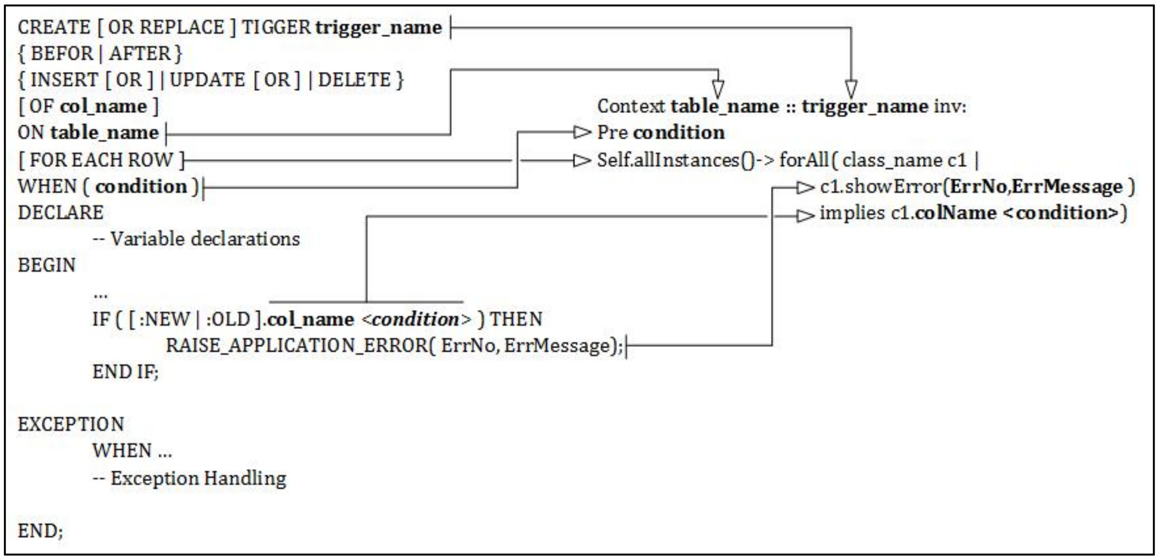
The OCL function *oclIsTypeOf()* check the type of each instance and return the desired object based on the parameter passed to function. In this example we select all instances of type *professor_type*.

5.3. Triggers To OCL

As declarative integrity constraints, triggers can constrain data input and enforce any type of integrity rule. A trigger always applies to new data only. For example, a trigger can prevent a DML (Data Manipulation Language) statement from inserting a *NULL* value into a database column, but the column might contain *NULL* values that were

inserted into the column before the trigger was defined or while the trigger was disabled.
Constraints are easier to write and less error-prone than triggers that enforce the same rules. However, triggers can enforce complex business or referential integrity rules that we cannot define with constraints.

Triggers are similar to stored procedures. A trigger stored in the database can include SQL and PL/SQL statements to run as a unit and can be invoked repeatedly. Unlike a stored procedure, we can enable and disable a trigger, but we cannot explicitly invoke it. It is composed by triggering event or statement, trigger restriction and finally trigger action. The triggering event is a SQL statement, database event, or user event that causes a trigger to fire. The trigger restriction is a Boolean expression that must be true for the trigger to fire. The trigger action is a PL/SQL block that contains SQL and procedural code to be run when the triggering event occurs and the restriction condition is true.
To identify triggers enforcing a complex business rules defined by the user, we use the proposition defined in [12]: all triggers embedding in their action section PL/SQL statement raising an exception are classified as constraint-enforcing-triggers .
The transformation of such triggers begins with the context of the OCL invariant, which is the UML class corresponding to the table where the trigger is defined (the targeted class is already retrieved from first phase), the body of the OCL invariant is composed by the trigger restriction condition, if defined, and the OCL instructions generated from the PL/SQL block defined in the trigger body. The (figure 2) present the generic and basic transformation for a simple trigger to OCL.
The trigger is fired when the condition in WHEN clause is true. To transform a conditional trigger, and execute the OCL instructions after verifying the condition, we use precondition. In OCL Pre- and postconditions are constraints that define a contract that an implementation of the operation has to fulfill. A precondition must hold when an operation is called, a postcondition must be true when the operation returns. As mentioned before, triggers are similar to stored procedures in some characteristic, for that, triggers play the role of operation in object oriented world. The context of the OCL invariant is the operation (*trigger_name*) of the class already generated from table (*table_name*) on which the trigger is created. *FOR EACH ROW* clause is transformed to *allInstance()* OCL operation, the condition which handle the exception is transformed to *forAll()* function, this function show the error message for each instance if the Boolean expression is true.

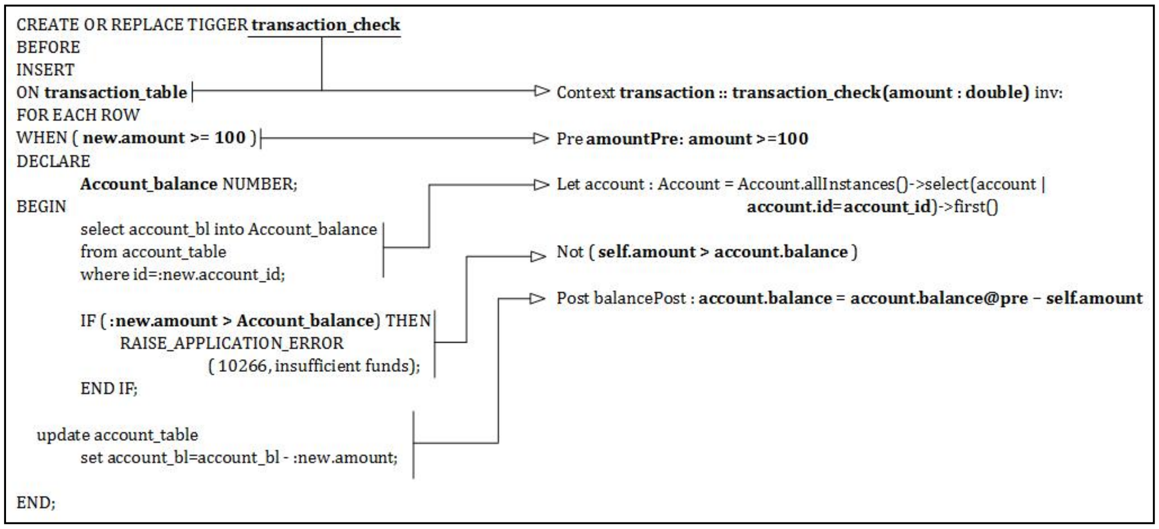


```
CREATE [ OR REPLACE ] TIGGER trigger_name
{ BEFOR | AFTER }
{ INSERT [ OR ] | UPDATE [ OR ] | DELETE }
[ OF col_name ]
ON table_name
[ FOR EACH ROW ]
WHEN ( condition )
DECLARE
          -- Variable declarations
BEGIN
          ...
          IF ( [ :NEW | :OLD ].col_name <condition> ) THEN
                    RAISE_APPLICATION_ERROR( ErrNo, ErrMessage);
          END IF;

EXCEPTION
          WHEN ...
          -- Exception Handling

END;
```

```
Context table_name :: trigger_name inv:
Pre condition
Self.allInstances()-> forAll( class_name c1 |
          c1.showError(ErrNo,ErrMessage )
          implies c1.colName <condition>)
```

Figure 2 : Basic Trigger Transformation

In figure 3, we show an example of the transformation of *transaction_check* constraint. The trigger is executed when the new inserted value amount is greater or equal 100. This trigger raises an exception when the amount withdrawn of the new transaction is greater than the account balance. The account balance is stored in Account table. This variable is retrieved using *Account_balance* variable by means of a *SELECT INTO* clause. The transformation of the declared variable in *transaction_check* trigger, gives a rise to an object of an Account type, defined by let expression.
The desired Account object is retrieved by searching the *account_id* using the *select()* function. *Select()* returns a collection with all elements of class that validate the OCL condition. In PL/SQL *SELECT INTO* statement can only return one single row, as *select()* function return collection of object, we use *first()* function to get the first element. The account balance is compared according to a Boolean expression that map the negation of the PL/SQL if-statement condition. Finally we use OCL post-condition (*balancePost*) to verify the new value of account balance. The new value should equal the previous value of account balance minus amount. We use *@pre* modifier to get the value of account balance in OCL precondition.



```
CREATE OR REPLACE TIGGER transaction_check
BEFORE
INSERT
ON transaction_table
FOR EACH ROW
WHEN ( new.amount >= 100 )
DECLARE
        Account_balance NUMBER;
BEGIN
        select account_bl into Account_balance
        from account_table
        where id=:new.account_id;

        IF ( :new.amount > Account_balance) THEN
             RAISE_APPLICATION_ERROR
                        ( 10266, insufficient funds);
        END IF;

   update account_table
        set account_bl=account_bl - :new.amount;

END;
```

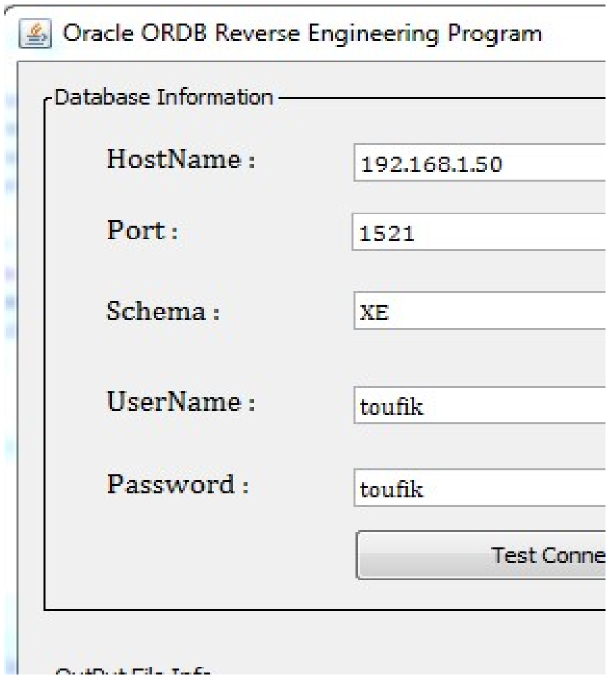
```
Context transaction :: transaction_check(amount : double) inv:
Pre amountPre: amount >=100
Let account : Account = Account.allInstances()->select(account |
                                    account.id=account_id)->first()
Not ( self.amount > account.balance )
Post balancePost : account.balance = account.balance@pre – self.amount
```

Figure 3 : Trigger Transformation Example

6. IMPLEMENTATION AND VALIDATION

To demonstrate the validity of our approach, a tool have been developed (Figure 4) to present the reverse engineering method proposed in this paper.
To develop our prototype, we use java as a programming language, and to create UML class diagram we use graphviz. Graphviz is open source graph visualization software initiated by AT&T Labs Research for drawing graphs specified in DOT (DOT is a plain text graph description language).
This tool takes a set of parameters as input to establish connection with oracle instance, specifically the user schema. The input parameters are: ip address of the server hosting the Oracle database, port, schema, username and password.
After establishing connection, a set of SQL queries are executed on the data dictionary to get information about types, attributes, methods, associations and other components.
The obtained conceptual schema CS is expressed as UML class diagram as show in figure 5. The example below present a set of SQL create statements of types with attributes and methods. This case study presents a simple scenario to demonstrate the transformation process from an object relational database ORDB to UML class diagram.

```
CREATE TYPE address AS OBJECT(
        int address_id, city varchar(20), country
        varchar(20));

CREATE TYPE job AS OBJECT (int job_id, title
        varchar(20), double minSal, double
        maxSal);

CREATE TYPE job_ref AS OBJECT(job_r REF
JOB);

CREATE TYPE job_list AS VARRAY(3) of job_ref;
CREATE TYPE transaction AS OBJECT
        (transaction_id int,
        transaction_date date, transaction_amount
        double,
        MEMBER FUNCTION get_last_trans_id
        RETURN VARCHAR2)

CREATE TYPE transaction_ref AS
        OBJECT(transaction_r REF transaction);

CREATE TYPE transaction_list AS TABLE OF
        transaction_ref;

CREATE TYPE account AS OBJECT(account_id
        int, account_balance double,
        transaction_lst transaction_list);

CREATE TYPE account_list AS VARRAY(5) of
account;

CREATE TYPE person AS OBJECT(person_id int,
        first_name varchar(20),
         last_name varchar(20), birth_date date,
        addr address,
        MEMBER FUNCTION get_age RETURN
                INT,
        MEMBER PROCEDURE
        show_information )
        NOT INSTANTIABLE NOT FINAL;

CREATE TYPE client UNDER person(category
        varchar(20), inscription_date date,
        Account_lst account_list,
        MEMBER PROCEDURE get_client_info);

CREATE TYPE employee UNDER person(salary
        double, hire_date date, job_lst job_list,
        MEMBER FUNCTION
                calcul_salary(worked_houres int,
                price_per_hour double)
                RETURN double);
```


Oracle ORDB Reverse Engineering Program

Database Information

HostName : 192.168.1.50

Port : 1521

Schema : XE

UserName : toufik

Password : toufik

Figure 4 : ORDB Reverse Engineering Program

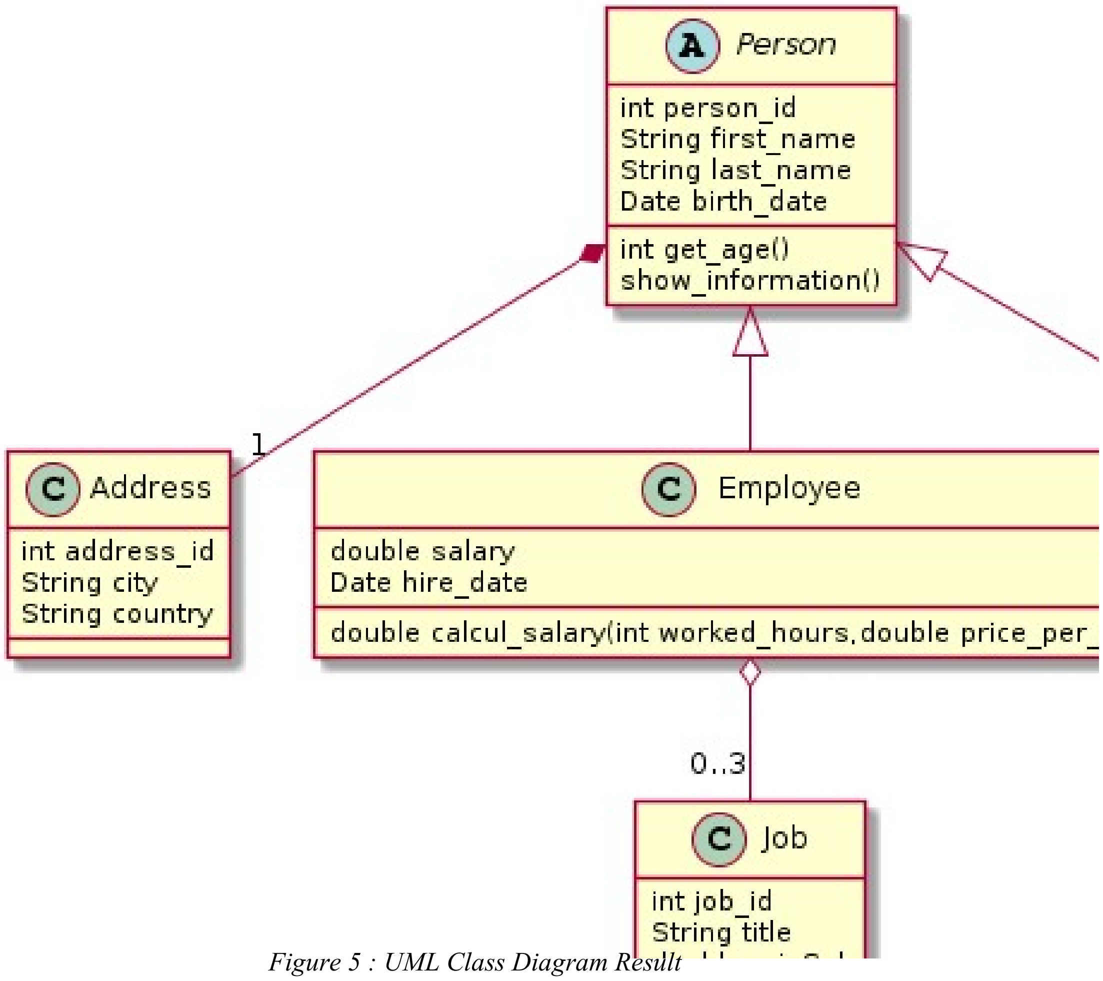


Figure 5 : UML Class Diagram Result

7. CONCLUSION

In this paper, we have proposed a new reverse engineering method to obtain a Conceptual Schema (CS) extended with a set of OCL integrity constraint out of running Object Relational Database (ORDB), by executing a set of queries on database dictionary. This CS facilitate the comprehension of the integrity constraints and can help for database migration, maintenance and evolution for systems using ORDB.
Our method is based on two main phases, model extraction and constraints extraction; each phase contains a set of transformation rules and steps.
As further work, we would like to extend our reverse engineering method to address object methods (member function and member procedure) and add triggers transformation to our prototype tool.

REFRENCES:

[1] Hainaut, J-L, Database Reverse Engineering, Models, Techniques and Strategies, in Preproc. of the 10th Conf. on Entity-Relationship Approach, San Mate0 (CA), 1991

[2] Premerlani, W.J., Blaha, M.R: An approach for reverse engineering of relational databases. In Working Conference on Reverse Engineering. (May 1993) pages 151-160.

[3] Roger H. L. Chiang, Terence M.Barron, Veda C. Storey: Reverse Engineering of relational databases: extraction of an EER model from relational database. In Data & Knowledge Engineering (March 1994) pages 107-142

[4] Anderson, M: Extracting an entity relationship schema from a relational database through

reverse engineering. In Entity Relationship approach (1994) pages 403-419

[5] Ramanathan, S. Hodges, J: Extraction of object-oriented structures from existing relational databases. In ACM Sigmod Record (March 1997) pages 59-64

[6] David W. Embely: Relational database reverse engineering: A model-centric, transformational, interactive approach formalized in model theory. In DEXA'97 Database and Expert Systems Applications (1997) pages 372-377

[7] H. Balsters, Modelling Database Views with Derived Classes in the UML/OCL-framework in P. Stevens, ≪UML≫2003 - The Unified Modeling Language. Modeling Languages and Applications, Volume 2863 of Lecture Notes in Computer Science; Springer, 2003, pp 295-309.

[8] D. H. Akehurst, B. Bordbar, On Querying UML Data Models with OCL in M. gogolla, ≪UML≫ 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Volume 2185 of Lecture Notes in Computer Science ; Springer, 2001, pp 91-103.

[9] Chaparro. O, Aponte. J, Ortega. F, Towards the Automatic Extraction of Structural Business Rules from Legacy Databases in IEEE Working conference on reverse engineering, 2012, pp 479 – 488.

[10] D. Yeh, Y. Li, and W. Chu, "Extracting entity-relationship diagram from a table-based legacy database," Journal of Systems and Software, vol. 81, no. 5, pp. 764 – 771, 2008.

[11] Alhajj, R., "Extracting the extended entity-relationship model from a legacy relational database", Information Systems, Elsevier Science Ltd., 2002, pp.597-618.

[12] Cosentino and S. Martinez: Extracting UML-OCL Integrity Constraints and Derived Types From Relational Databases. In the 13th International Workshop on OCL, Model Constraints and Query Languages, Miami, United States, (2013) pages 43-52.

[13] Rajani Chennamaneni, Emanuel S. Grant : Comparison and Evaluation of Methodologies for Transforming UML Models to Object-Relational Databases, University of North Dakota. 2002

[14] Wai Yin Mok, David P. Paper: On Transformations from UML Models to Object-Relational Databases. In the 34th Annual Hawaii International Conference on system science. (HICSS-34)-Volume 3 (Jan 2001) p.3046

[15] Cabot J., Gomez C., Planas E. and Rodriguez M. E., Reverse Engineering of OO Constructs in Object-Relational Database Schemas, JISBD 2008, (2008)

[16] Object Constraint Language Specification v 2.4 OCL.2.4 URL:http://www.omg.org/spec/OCL/2.4/PDF/