Learning Ontology from Object-Relational Database

Andrejs Kaulins[1], Arkady Borisov[2]
[1, 2] *Riga Technical University*

Abstract **– This article describes a method of transformation of object-relational model into ontology. The offered method uses learning rules for such complex data types as object tables and collections – arrays of a variable size, as well as nested tables. Object types and their transformation into ontologies are insufficiently considered in scientific literature. This fact served as motivation for the authors to investigate this issue and to write the article on this matter. In the beginning, we acquaint the reader with complex data types and object-oriented databases. Then we describe an algorithm of transformation of complex data types into ontologies. At the end of the article, some examples of ontologies described in the OWL language are given.**

Keywords **– Complex data types, object-relational model, ontology, ontology learning.**

I. Introduction

Many systems use object-oriented databases in the architecture [1]–[4]. The process of ontology creation from such databases is insufficiently well described in scientific articles and literature. Object-oriented database (OOD) is a database in which data are modelled in the form of objects, their attributes, methods and classes.

This article describes a transformation algorithm of object-relational model into ontology [8]–[12]. The algorithm is constructed on application of compliance rules for complex data types, which are implemented in a relational database from Oracle Corporation. In the beginning, we consider complex data types, then the algorithm of transformation, and in conclusion we describe some examples of transformation of complex data types into ontologies.

II. Complex Data Types

For modelling real objects, for example, such as clients, orders and payments, complex data types are used in databases. Oracle company in its own database product allows one to implement the following types of data: object tables, collections – variable length arrays and nested tables.

The created object types may contain the built-in data types, earlier defined object types, references to objects and collections. Object data types can be used for processing video, audio and graphic information.

The model of object types is similar to the mechanism of classes in object-focused programming implemented in such languages as C++, Java. Likewise classes, object types can also be reused, which makes the process of application programming of databases more effective and fast.

Let us consider the main characteristics of object-oriented databases.

III. Characteristics of Object-Oriented Databases

Object-oriented databases are usually recommended for those cases when high-performance data processing is required, having complex data structure.

Characteristics of OOD are considered in the OOD manifesto [5]. Their choice is based on two criteria: the system has to be object-oriented and represent a database.

In what follows, we will list obligatory characteristics of object-oriented databases.

Support of complex objects. A possibility of creation of compound objects due to application of constructors of compound objects has to be provided in the system. It is required that constructors of objects are orthogonal, that is, any constructor would be possible to apply to any object.

Support of identity of objects. All objects have to have the unique identifier, which does not depend on the values of object attributes.

Encapsulation support. Correct encapsulation is reached because programmers have the right of access only to the specification of the interface of methods, and data and implementation of methods are hidden in objects.

Support of types and classes. It is required that in the OOD at least one concept of distinction between types and classes was supported. (The term “type” rather corresponds to the concept of abstract type of data. The variable appears in programming languages with the indication of its type. The compiler can use this information to check the operations, which are carried out from a variable on compatibility with its type that allows one to guarantee software correctness. On the other hand, the class is a certain template for the creation of objects and provides methods, which can be applied to these objects. Thus, the concept “class” belongs to the time of execution rather than to the time of compilation.)

Support of inheritance of types and classes from their ancestors. The subtype, or a subclass, has to inherit attributes and methods from its super type, or a super class, respectively.

Overload in combination with full binding. Methods have to be applied to objects of different types. Implementation of a method has to depend on type of objects, to which this method is applied. For ensuring this functionality, binding of names of methods in the system should not be carried out until execution of the program.

Computing completeness. Data manipulation language has to be a programming language of general purpose.

The data types have to be expandable. The user has to have tools for new types of the predetermined system types given on the basis of the existing set. Moreover, there should be no distinctions between ways of use of the system and user types of data.

Apart from obligatory characteristics, there are also optional ones:

1. Multiple inheritance;
2. Check of types;
3. Distribution;
4. Design transactions.

IV. Object Data Types

Object data types in application are similar to simple types, such as NUMBER or VARCHAR2. They consist of attributes and methods [6]. If we have preset values of attributes, the object type is called an exemplar (object instance). In Fig. 1, an example of object type and several exemplars is shown.

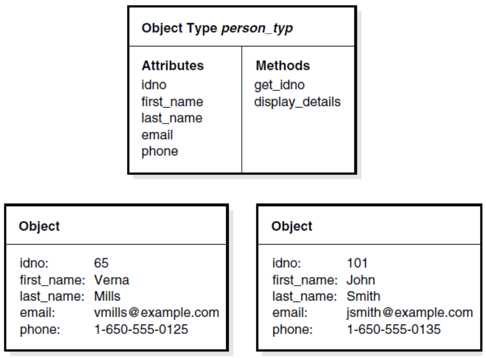

Fig. 1. Object type and exemplars.

Program code by which it is possible to set an object type in a database is shown below:

```
CREATE TYPE person_typ AS OBJECT (
 idno NUMBER,
 first_name VARCHAR2(20),
 last_name VARCHAR2(25),
 email VARCHAR2(25),
 phone VARCHAR2(20),
 MAP MEMBER FUNCTION get_idno RETURN NUMBER,
 MEMBER PROCEDURE display_details (SELF IN
 OUT NOCOPY person_typ));
/
```

Methods of object type are defined as follows:

```
CREATE TYPE BODY person_typ AS
   MAP MEMBER FUNCTION get_idno RETURN NUMBER
IS
   BEGIN
    RETURN idno;
   END;
   MEMBER PROCEDURE display_details ( SELF IN
   OUT NOCOPY person_typ ) IS
   BEGIN
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' '
    || first_name || ' ' || last_name);
    DBMS_OUTPUT.PUT_LINE(email || ' ' ||
   phone);
  END;
  END;
```

The variable of object type is an exemplar. Object types consist of attributes and methods (Fig. 2).

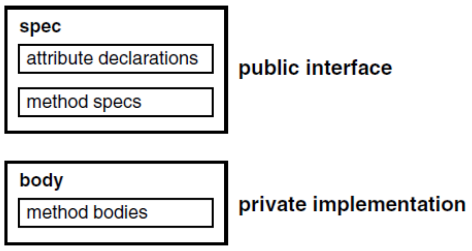

Fig. 2. Object attributes and methods.

After definition of object type, it is possible to create the tables having one or several columns of object type. An example of this kind of table is given below:

```
CREATE TABLE contacts (
     contact person_typ,
     contact_date DATE
);
```

The table *contacts* contains a column of earlier defined object type – person_type. The objects contained in the columns of the table are called column objects.

Such a table can be filled in with data by means of a standard data modification language (DML), for example, with operator INSERT:

```
INSERT INTO contacts VALUES (
  person_typ(
    65,'Verna','Mills',
    'vmills@example.com','1-650-555-0125'
  ),
to_date('24 Jun 2003','dd Mon YYYY')
);
```

Methods of object are performed functions or procedures determined by the user. The methods can be of several types:

Member methods. These methods implement access to attributes of object, and also the procedure and operation over attributes.

Static methods. Implement operations over exemplars of objects. For example, it is possible to define operation of object comparison. Such operations are defined by static methods.

Constructor methods. By means of exemplar constructor, exemplars of the set type are created. The constructor is set implicitly for any object; however, the user can make necessary changes that influence the process of creation of exemplars.

External implemented methods. External methods can be implemented in C/C++ and are stored in a type of libraries outside a database.

An example of using a method for the previously considered person_typ types and the table contacts is given below:

```
select c.contact.get_idno() from contacts c;
```

This query will print on display attribute of idno for all objects from the table *contacts*.

A. Object Tables

A table containing only objects is called object table. Every record in such a table represents the whole object. In such tables, it is possible to refer to objects from other objects or a program code. For such references in the database, special identifiers of objects are used.

Such object identifiers can be of two types:

1. Systemically generated (are applied by default).
2. Based on the primary keys.

For use of identifiers, the construction REF is used, which allows addressing an object.

An example of creation of the object table for the previously defined type person_typ is as follows:

```
CREATE TABLE person_obj_table OF person_typ;
```

Further, the object table can be filled in with data:

```
INSERT INTO person_obj_table VALUES (
 person_typ(101, 'John', 'Smith',
'jsmith@example.com', '1-650-555-0135')
);
```

A reference to object (REF) can be used as follows:

```
CREATE TYPE emp_person_typ AS OBJECT (
name VARCHAR2(30),manager REF emp_person_typ
);
/

CREATE TABLE emp_person_obj_table OF
emp_person_typ;

INSERT INTO emp_person_obj_table VALUES (
emp_person_typ ('John Smith', NULL));

INSERT INTO emp_person_obj_table SELECT
emp_person_typ ('Bob Jones', REF(e))
FROM emp_person_obj_table e WHERE e.name =
'John Smith';

select * from emp_person_obj_table e;

NAME    MANAGER
---------- ------------------------------
John Smith
Bob                                  Jones
0000220208424E801067C2EABBE040578CE70A070742
4E801067C1EABBE040578CE70A0707
```

For reference types it is possible to use restrictions:

```
CREATE TABLE contacts_ref ( contact_ref REF
person_typ SCOPE IS person_obj_table,
contact_date DATE );
```

According to the references, it is possible to address objects (implicit dereferencing):

```
SELECT e.name, e.manager.name FROM
emp_person_obj_table e
WHERE e.name = 'Bob Jones';

SELECT DEREF(e.manager) FROM
emp_person_obj_table e;

DEREF(E.MANAGER)(NAME, MANAGER)
----------------------------------------
EMP_PERSON_TYP('John Smith', NULL)
```

A PL/SQL code example is presented below:

```
DECLARE
  person_ref REF person_typ;
  person person_typ;
BEGIN
  SELECT REF(p) INTO person_ref FROM
person_obj_table p WHERE p.idno = 101;
  select deref(person_ref) into person
from dual;
  person.display_details();
END;
```

B. Oracle Collections

Collections in the databases are used for modelling multiple attributes (multi-valued attributes) and relations “many to many”.

Collections can be of two types – variable length arrays (VARRAY) and nested tables. Collections can serve as type of data in objects or simple tables (as certain columns).

We will give an example of collection:

```
CREATE TYPE people_typ AS TABLE OF
person_typ;
```

Now this complex type can be used for the definition of other type:

```
CREATE TYPE dept_persons_typ AS OBJECT (
     dept_no CHAR(5),
     dept_name CHAR(20),
     dept_mgr person_typ,
     dept_emps people_typ);
/
```

The attribute of dept_emps is the table supporting the employees working at a certain department.

V. Complex Data Type Transformation Algorithm

The algorithm is an expansion of the method of transformation of relational data model [7]. For each complex data type of object-relational model, the group of learning rules is defined, which maps objects from the database into ontology. The general algorithm is shown in Fig. 3.

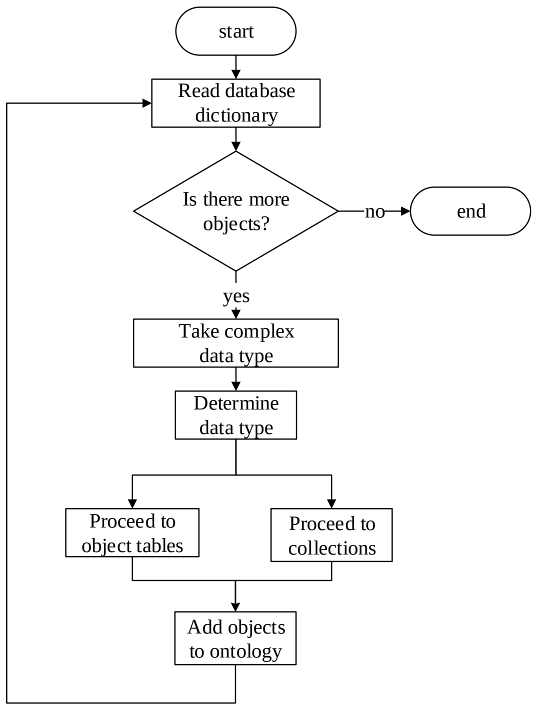

Fig. 3. Complex data type transformation steps.

VI. Learning Rules

A. Learning Rules for Object Tables

For object types we use several learning rules, which allow us to create classes, attributes and instances in ontology.

Let us consider an example:

```
CREATE TYPE person_typ AS OBJECT (
   idno NUMBER,
   first_name VARCHAR2(20),
   last_name VARCHAR2(25),
   email VARCHAR2(25),
   phone VARCHAR2(20),
   MAP MEMBER FUNCTION get_idno RETURN NUMBER,
          MEMBER PROCEDURE display_details (
       SELF IN OUT NOCOPY person_typ ));
  /

CREATE TABLE contacts (
       contact person_typ,
       contact_date DATE
  );

  INSERT INTO contacts VALUES ( person_typ
(65, 'Verna', 'Mills', 'vmills@example.com',
'1-650-555-0125'), to_date('24 Jun 2015', 'dd
Mon YYYY'));

  INSERT INTO contacts VALUES ( person_typ
(66, 'John', 'Smith',
'jsmith@example.com', '1-650-555-0325'),
to_date('26 Jun 2015', 'dd Mon YYYY'));
```

Rule 1. Every certain type defined by a user will be transformed into an ontology class.

```
  Ontology(<http://www.my.example.com/example>
Declaration( Class( a:Person_Type)))
```

Rule 2. Attributes of object type will be transformed into attributes of a class of ontology.

```
  Ontology(<http://www.my.example.com/example>
DataPropertyDomain (a:hasIdno a:Person_type)
  DataPropertyDomain (a:hasFirstName
a:Person_type)
  DataPropertyDomain (a:hasLastName
a:Person_type)
  DataPropertyDomain (a:hasEmail
a:Person_type)
  DataPropertyDomain (a:hasPhone
a:Person_type)
  DataPropertyRange (a:hasIdno xsd:int)
  DataPropertyRange (a:hasFirstName
xsd:string)
  DataPropertyRange (a:hasLastName xsd:string)
  DataPropertyRange (a:hasEmail xsd:string)
  DataPropertyRange (a:hasPhone xsd:string)
  )
```

Rule 3. Concrete values of attributes form instances in ontology.

```
  Ontology(<http://www.my.example.com/example>
  DataPropertyAssertion(a:hasIdno
a:Person_type "65" ^^ xsd:int)
  DataPropertyAssertion(a:hasFirstName
a:Person_type "Verna" ^^ xsd:string)
  DataPropertyAssertion(a:hasLastName
a:Person_type "Mills" ^^ xsd:string)
  DataPropertyAssertion(a:hasEmail
a:Person_type "vmills@example.com" ^^
xsd:string)
  DataPropertyAssertion(a:hasPhone
a:Person_type "1-650-555-0125" ^^ xsd:string)
  DataPropertyAssertion(a:hasIdno
a:Person_type "66" ^^ xsd:int)
  DataPropertyAssertion(a:hasFirstName
a:Person_type "John" ^^ xsd:string)
  DataPropertyAssertion(a:hasLastName
a:Person_type "Smith" ^^ xsd:string)
  DataPropertyAssertion(a:hasEmail
a:Person_type "jsmith@example.com" ^^
xsd:string)
  DataPropertyAssertion(a:hasPhone
a:Person_type "1-650-555-0325" ^^ xsd:string)
  )
```

B. Learning Rules for Collections

Let us define object like a Course:

```
CREATE TYPE Course AS OBJECT (
  course_no NUMBER(4),
  title     VARCHAR2(35),
  credits    NUMBER(1));
```

Now define TABLE type CourseList which contains objects of Course:

```
CREATE TYPE CourseList AS TABLE OF Course;
```

We will create the table *department* which contains a CourseList column:

```
CREATE TABLE department (
  name     VARCHAR2(30),
  director VARCHAR2(40),
  office   VARCHAR2(40),
  courses  CourseList)
  NESTED TABLE courses STORE AS courses_tab;
```

An example of filling the table with data is provided below:

```
INSERT INTO department
VALUES('Psychology', 'Irene Friedman', 'Fulton
Hall 133',           CourseList(Course(1000,
'General Psychology', 5),
  Course(2100, 'Experimental Psychology', 4),
  Course(4320, 'Cognitive Processes', 4),
  Course(4410, 'Abnormal Psychology', 4)));
```

Rule 1. Nested tables will be transformed to a separate class of ontology.

Ontology(<http://www.my.example.com/example>
Declaration(Class(a:Department))
Declaration(Class(a:Course))
)

Rule 2. Attributes of the nested table will be transformed into attributes of this class of ontology.

```
Ontology(<http://www.my.example.com/example>
DataPropertyDomain (a:hasCourseno a:Course)
DataPropertyDomain (a:hasTitle a:Course)
DataPropertyDomain (a:hasCredits a:Course)
DataPropertyRange (a:hasCourseno xsd:int)
DataPropertyRange (a:hasTitle xsd:string)
DataPropertyRange (a:hasCredits xsd:int)
)
```

Rule 3. Concrete values of attributes form instances in ontology.

```
  Ontology(<http://www.my.example.com/example>
  DataPropertyAssertion(a:hasCourseno a:Course
"2100" ^^ xsd:int)
  DataPropertyAssertion(a:hasTitle a:Course
"General Psychology" ^^ xsd:string)
  DataPropertyAssertion(a:hasCredits a:Course
"5" ^^ xsd:int)
  DataPropertyAssertion(a:hasCourseno a:Course
"1000" ^^ xsd:int)
  DataPropertyAssertion(a:hasTitle a:Course
"'Experimental Psychology" ^^ xsd:string)
  DataPropertyAssertion(a:hasCredits a:Course
"4" ^^ xsd:int)
  DataPropertyAssertion(a:hasCourseno a:Course
"4320" ^^ xsd:int)
  DataPropertyAssertion(a:hasTitle a:Course
"Cognitive Processes" ^^ xsd:string)
  DataPropertyAssertion(a:hasCredits a:Course
"4" ^^ xsd:int)
  DataPropertyAssertion(a:hasCourseno a:Course
"4410" ^^ xsd:int)
  DataPropertyAssertion(a:hasTitle a:Course
"Abnormal Psychology" ^^ xsd:string)
  DataPropertyAssertion(a:hasCredits a:Course
"4" ^^ xsd:int)
  )
```

The built ontology can be represented graphically (Fig. 4).

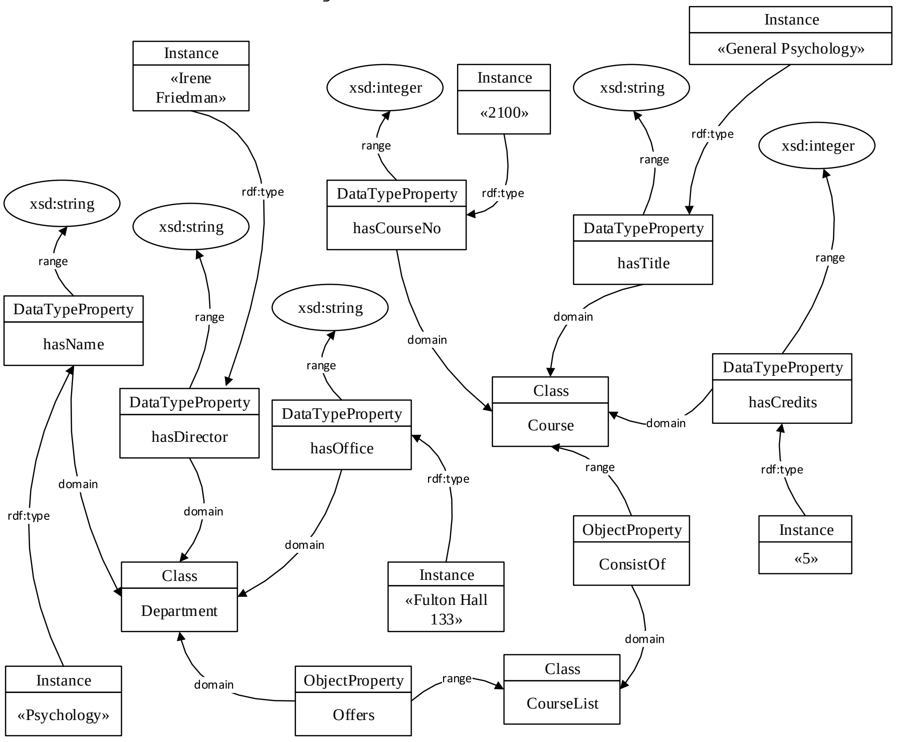

Fig. 4. Ontology example built using learning rules.

VII. Conclusion

This article has introduced the complex data type transformation algorithm from a database into ontology. In the beginning, definitions of object-oriented databases have been given. The main characteristics of such database types have been considered.

To be specific, examples of complex data types implemented in the Oracle database have been provided. Then steps of transformation of object data types into ontology are shown.

For each complex data type, a set of learning rules has been provided, which is used for the transformation of database objects into ontology.

In conclusion, examples of transformation from a database into ontologies described in the OWL language have been considered, which confirms the applicability of the algorithm.

References

[1] M. Rapaport, "Object-Oriented Data Bases: The Next Step in DBMS Evolution," *Comp. Lang.* vol. 5, no. 10, 1988, pp. 91–98.

[2] M. Stonebraker, "Future Trends in Database Systems", *IEEE Trans. Knowledge and Data Eng.* vol. 1, no. 1, 1989, pp. 33–44. http://dx.doi.org/10.1109/69.43402

[3] O. Nierstrasz, "*A Survey of Object-Oriented Concepts*", ACM Press and Addison-Wesley, 1989, pp. 3–21.

[4] M. A. Garvey, M. S. Jackson, "Introduction to Object-Oriented Databases", *Inf. and Software Technol.* vol. 31, no. 10, 1989, pp. 521–528. http://dx.doi.org/10.1016/0950-5849(89)90173-0

[5] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, S. Zdonik, "The Object-Oriented Database System Manifesto", In *Proc. of the First Int. Conf. on Deductive and Object-Oriented Databases*" , Kyoto, Japan, 1989, pp. 223–240.

[6] J. Greenberg, *Oracle Database Object-Relational Developer's Guide 12c Release 1*, Oracle corp., 2014.

[7] A. Kaulins, A. Borisovs, "A. Building Ontology from Relational Database", *Information Technology and Management Science*, vol. 17, 2014, pp. 45–49.

[8] S. Staab, R. Studer, "Handbook on Ontologies", *International Handbooks on Information Systems*, Springer Science and Business Media, 2013.

[9] B. Bennett, C. Fellbaum, "Formal Ontology in Information Systems", *Proc. of the Fourth Int. Conf. FOIS 2006*, IOS Press, 2006.

[10] L. W. Lacy, *Owl: Representing Information Using the Web Ontology Language*, Trafford Publishing, 2005.

[11] R. Poli, M. Healy, A. Kameas, "Theory and Applications of Ontology", *Computer Applications*, Springer Science and Business Media, 2010. http://dx.doi.org/10.1007/978-90-481-8847-5

[12] K. Munn, B. Smith, "Applied Ontology: An Introduction", *Metaphysical Research*, vol. 9, Walter de Gruyter, 2008. http://dx.doi.org/10.1515/9783110324860

Andrejs Kaulins received the *B. Sc.* and *M. Sc.* degrees in 1993 and 2002 from Riga Technical University and the University of Latvia. Since 2013 he has been studying at Riga Technical University to obtain a Doctoral degree in Computer Science. Currently, major field of study is complex IT system design based on ontologies.
E-mail: andrejs.kaulins@rtu.lv

Arkady Borisov received his Doctoral degree in Technical Cybernetics from Riga Polytechnic Institute in 1970 and *Dr. habil. sc. comp.* degree in Technical Cybernetics from Taganrog State Radio Engineering University in 1986. He is a Professor of Computer Science at the Faculty of Computer Science and Information Technology, Riga Technical University (Latvia). His research interests include fuzzy sets, fuzzy logic and computational intelligence. He has 235 publications in the field. He has supervised a number of national research grants and participated in the European research project ECLIPS. He is a member of IFSA European Fuzzy System Working Group, Russian Fuzzy System and Soft Computing Association, honorary member of the Scientific Board, member of the Scientific Advisory Board of the Fuzzy Initiative Nordrhein-Westfalen (Dortmund, Germany).
E-mail: arkadijs.borisovs@cs.rtu.lv