

Reverse Engineering of OO constructs in Object-Relational Database Schemas

Jordi Cabot¹, Cristina Gómez², Elena Planas¹ and M.Elena Rodríguez¹

¹*Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya
[jcabot,eplanash,mrodriguezgo]@uoc.edu*

²*Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
cristina@lsi.upc.edu*

Abstract. Reverse engineering applied to databases permits to extract a conceptual schema that represents, at a higher level of abstraction, the database implementation. This resulting conceptual schema may be used to facilitate, among others, system maintenance, evolution and reuse. In the last years, the use of object-relational constructs was incorporated into database development. However, reverse engineering techniques for these specific constructs have not been yet provided. In this sense, the main goal of this paper is to present a method that considers these new constructs in the reverse engineering of an existing object-relational database. As a result of the process, our method returns an equivalent conceptual schema specified in UML (extended with a set of OCL integrity constraints) that represents, at a conceptual level, the database schema. We provide a prototype tool that implements our method for the Oracle9i database management system.

Keywords: Reverse Engineering, Object-relational database, UML

1 Introduction

Software reverse engineering is the process of analyzing a subject system to identify its components and their interrelationships and to create representations of the system in another form or at higher level of abstraction [1]. Reverse engineering applied to databases (DB) permits to obtain a conceptual schema (CS) that represents, at a higher level of abstraction, a DB implementation.

This CS may be helpful in several situations. For instance, when documenting the system to ease its maintainance and evolution. Usually, changes on the DB schema (DBS) are poorly documented once the DB has been created. Therefore, reverse engineering of the DBS from time to time is needed to keep its documentation up to date. The CS also facilitates sytem integration and reuse. The integration process requires data understanding of the systems to be integrated. This data understanding is better achieved when we raise its abstraction level to omit all implementation details, as done during the DB-to-CS transformation.

Reverse Engineering has received a lot of attention during the last decades in the DB field [2], [3]. For relational databases, several processes have been defined to extract a conceptual object model from a relational one. Concretely, [4] presents a flexible and interactive approach to obtain an OMT model [5] with classes, associations, generalizations and key and referential constraints from a relational DBS. In the same way, [6] states a method for translating a relational DBS to an ORM [7] model instance. Some work has also been done in the context of OODBs. [8] defines a series of transformations to derive an OMT model from the source code of an existing OO system with an underlying OODB. [9] proposes a process of rebuilding an abstract documentation of the

object classes with attributes, inheritance hierarchies and methods which are declared and manipulated in OO application programs.

Nevertheless, although techniques for forward engineering of Object-Relational databases (ORDB) have been developed [10], [11], as far we know, reverse engineering for ORDB has not been addressed. Given that most vendors (as Oracle or IBM) and open source DBs (as PostgreSQL) incorporate nowadays OO constructs in their flagship products, we believe that specific methods for reverse engineering of ORDBs are required, especially because these new OO constructs (combined with the previous relational constructs) introduce multiple alternatives in DB design, alternatives that must be considered when reverse engineering this kind of databases.

The main goal of this paper is to present a new method to obtain a UML-based conceptual schema representation (UML 2.1 [12]) of a given object-relational database schema (ORDBS). When appropriate, this UML schema is extended with a set of OCL integrity constraints [13].

Our method is composed of different steps, where each step tackles the transformation of some of the OO constructs that may appear in an ORDBS. In this paper, we only consider the reverse engineering of these constructs since, as we mention above, reverse engineering for relational database constructs has been deeply studied in the literature, and thus, our method can be regarded as an extension of these previous ones. Note that, even if OO constructs in ORDBs seem similar to those in object-oriented databases, they cannot be directly compared (the first ones have been designed to be compatible with the relational model while the second ones with OO programs). Therefore, a specific method for ORDBs must be developed.

Additionally, we provide a prototype tool that implements our reverse engineering method for the Oracle9i database management system (DBMS) one of the relational DBMS that best supports these new OO constructs.

The rest of the paper is structured as follows. Section 2 presents the OO constructs admitted by the SQL:1999 plus the additional ones provided in the Oracle9i DBMS. Section 3 presents an overview of our method. The transformation rules for each construct and the refinement of the resulting CS are presented in sections 4 and 5. Section 6 presents the prototype. Some conclusions and future work are sketched in section 7.

2 OO constructs in ORDBSs

Standard SQL, since SQL:1999 [14] offers a set of OO constructs that can be combined with classical relational constructs. SQL:2003, the latest version of standard SQL, maintains with few extensions these constructs. In this section we briefly examine some OO constructs, specifically we concentrate on those present in SQL:1999.

The main OO characteristic is the possibility of defining complex types, i.e. types that have no directly associated atomic values. Such complex types can be collections or structured compound types. Both can be placed wherever a built-in SQL data type could be used such as, for example, in table column definitions. This contrasts with the classical relational data model which explicitly forbids this possibility, given that a table with decomposable columns violates first normal form.

The SQL:1999 data type to specify collections is the ARRAY type; an ARRAY has a maximum cardinality and it can be built over any SQL:1999 data type except over another ARRAY type.

For defining structured compound types, SQL:1999 offers the ROW type and the user defined types (UDTs). We only examine UDTs, given that ROW types could also be represented by means of UDTs. A UDT allows users to express new types, as complex as required, through the definition of their properties (attributes) and their intended behaviour (methods). In its turn, the type of a UDT's attribute could be also a complex type. UDTs can be extended and/or redefined, i.e. they can participate in taxonomies. Then, we are able to indicate if they are instantiable, that is, whether new instances of the UDT can be created. An instance of a UDT is a set of values. To be objects (in OO paradigm), instances of UDTs must be stored as rows of typed tables, a new element of SQL:1999. A typed table is always based on a previously defined UDT. It can be seen

as a table having a single compound column (that follows the structure of its associated UDT), or as a table having as many columns as attributes the related UDT has.

Rows of typed tables are globally unique identified. References between rows can be defined using the new REF type. Therefore, the REF type can be used for implementing relationships as an alternative to the use of foreign keys. With REF types we can navigate across DB objects, saving join operations needed when using foreign keys.

SQL:1999 also allows table taxonomies. When a table is defined under its supertable, it inherits not only the column structure, but also integrity constraints, triggers, indexes, etc.

Up to this point we have examined SQL:1999, we now comment on the OO constructs offered by commercial ORDBMSs. We will focus on Oracle, which is the commercial product that better fits (and extends) OO characteristics defined by SQL:1999, although there are slightly differences in nomenclature and syntax. ARRAY, UDTs and typed tables are respectively called, in Oracle, VARRAY, object types and object tables. REF type is also supported by Oracle. In addition to VARRAYs, Oracle also offers nested tables (not included in SQL:1999) for defining collections. A nested table is an unordered collection of elements of a given type; it can be used as a column data type in another table; therefore a nested table can be seen as a table embedded in another table.

Table 2.1. Examples of OO constructs in Oracle.

UDTs definitions	addr Address, mail VARCHAR2(30), MEMBER PROCEDURE set_mail (newMail VARCHAR2), MEMBER PROCEDURE incr_age)NOT INSTANTIABLE NOT FINAL;
<pre>-- Array of 2 phones CREATE TYPE phones_list IS VARRAY(2) OF INTEGER; -- New sybtypes from Address are not allowed CREATE TYPE Address AS OBJECT(street VARCHAR2(40), num INTEGER, floor VARCHAR(10), town VARCHAR(30), postal_code INTEGER)INSTANTIABLE FINAL; -- Type ref to Organizer CREATE TYPE Organizer_REF_VARRAY AS OBJECT(ref_Organizer REF Organizer)INSTANTIABLE FINAL; -- Array of 6 Organizers CREATE TYPE Organizer_REFS_VARRAY IS VARRAY(6) OF Organizer_REF_VARRAY;</pre>	<pre>-- Type Organizer (Person subtype) CREATE TYPE Organizer UNDER Person(phone phones_list, supervises REF Checkpoint)INSTANTIABLE FINAL; -- Type Hiker (Person subtype) CREATE TYPE Hiker UNDER Person(ages_experience INTEGER, num_excursions INTEGER)INSTANTIABLE FINAL;</pre>
Typed table definitions	
<pre>-- Type Checkpoint CREATE TYPE Checkpoint AS OBJECT(id INTEGER, description VARCHAR2(100), excursion_id INTEGER, supervised_by Organizer_REFS_VARRAY, MEMBER PROCEDURE set_information (id INTEGER, descr VARCHAR))INSTANTIABLE FINAL; -- Person is an abstract type CREATE TYPE Person AS OBJECT(id INTEGER, dni VARCHAR2(9), name VARCHAR2(30), age INTEGER,</pre>	<pre>-- Table Checkpoint CREATE TABLE Checkpoint_OT OF Checkpoint(id PRIMARY KEY, description NOT NULL)OBJECT ID PRIMARY KEY; -- Table Person CREATE TABLE Person_OT OF Person(id PRIMARY KEY NOT NULL, name UNIQUE, CONSTRAINT check_age CHECK (age > 10))OBJECT ID PRIMARY KEY; -- Table Organizer CREATE TABLE Organizer_OT OF Organizer; -- Table Hiker CREATE TABLE Hiker_OT OF Hiker;</pre>

Table 2.1 shows examples of the previous described constructs, according to Oracle9i syntax [15]. These examples intend to model data required for a hiking centre interested in storing data about its members and organized activities. We will use these examples to illustrate our reverse engineering method in the following sections. The complete case study can be found in the Appendix.

3 Method Overview

Given a running ORDB, our method extracts its DBS and returns a possible equivalent CS specified in UML. By equivalent we mean that the CS and the DBS accept the same information (i.e., all data that can be stored in the DBS is a valid instantiation of the CS and conversely). We ensure this equivalence by choosing an appropriate construct in the CS for each element of the DBS.

Our method consists of three different phases (Fig. 3.1). In the first one, the DBS is retrieved from the running ORDB by means of executing a set of predefined queries against the data dictionary. The second phase addresses the transformation of the OO constructs of this DBS and returns a preliminary UML CS. Finally, in the third phase, the user may optionally intervene in the process to refine and improve the obtained CS.

By far, the main part of the reverse engineering process is carried out in the second phase. This second phase can be split up into the following basic transformation steps, where each step transforms a subset of the elements that may appear in the DBS. The transformation rules involved in each step are provided in the next section.

The ordered list of steps is the following:

1. *UDTs' transformation*: Each UDT gives rise to a class or data type in the CS.
2. *Built-in data types transformation*: Each data type is mapped to one of the predefined UML data types.
3. *UDTs' attributes transformation*: Given an attribute of a UDT, our method may generate an attribute in a UML class or data type, an association, a composition or an association class, depending on the attribute characteristics.
4. *UDTs' methods transformation*: Each UDT method produces a new method in the corresponding class or data type in the CS.
5. *Constraints transformation*: The constraints of the DBS bring about minimum association multiplicities and/or textual OCL integrity constraints in the CS.
6. *Taxonomies creation*: clauses UNDER and [NOT] INSTANTIABLE generate new generalization relationships in the CS.

This ordering facilitates the creation of the CS (new elements are defined upon previously created ones), though alternative orderings could be used instead.

We would like to remark that our method is not limited to the reverse engineering of SQL:1999 constructs; specific Oracle constructs are also considered.

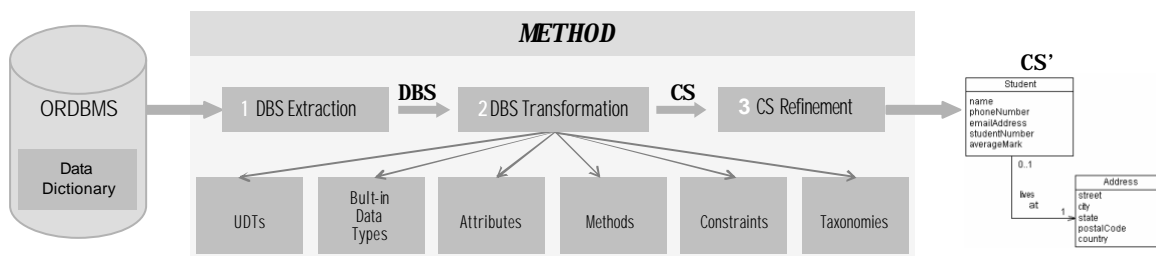


Fig 3.1. Method Overview.


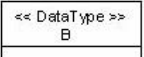
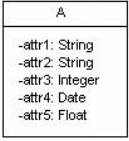
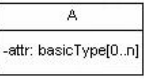
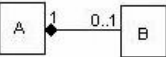
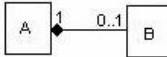

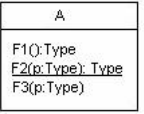
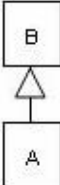
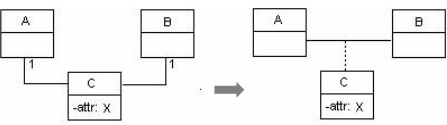
4 Transformation Rules

In what follows we define the different transformation rules presented above.

For each rule we describe its applicability conditions and include a transformation template that shows the DBS elements addressed by the rule and the corresponding UML elements, generated when processing the DBS with that particular rule. To indicate the elements of the DBS tackled by each rule, we show the SQL code excerpt (in Oracle9i syntax) that was used to create them in the running DB. See rule 1 for a more detailed explanation.

The application of all rules over the whole case study introduced in section 2 is shown in Appendix. Due to space limitations, in this section we will just present some relevant examples.

Table 4.1. Templates of a subset of Transformation Rules.

<p>Rule 1</p> <pre>CREATE TYPE A AS OBJECT(); CREATE TABLE A_ot OF A;</pre> 	<p>Rule 2</p> <pre>CREATE TYPE B AS OBJECT();</pre> 
<p>Rule 4</p> <pre>CREATE TYPE A AS OBJECT(attr1 VARCHAR2(40), attr2 CHAR(9), attr3 INTEGER, attr4 DATE, attr5 NUMBER(2,2)); CREATE TABLE A_ot OF A;</pre> 	<p>Rule 5</p> <pre>CREATE TYPE List IS VARRAY(n) OF basicType; CREATE TYPE A AS OBJECT(attr List); CREATE TABLE A_ot OF A;</pre> 
<p>Rule 7</p> <pre>CREATE TYPE A AS OBJECT(attr B); * Where A and B are classes</pre> 	<p>Rule 10</p> <pre>CREATE TYPE A AS OBJECT(attr B); * Where A and B are classes</pre> 
<p>Rule 13</p> <pre>CREATE TYPE A AS OBJECT(attr B); * Where A is a class and B is a data type</pre> 	<p>Rule 14</p> <pre>CREATE TYPE A AS OBJECT(MEMBER FUNCTION F1 RETURN Type, STATIC FUNCTION F2 (p Type) RETURN Type, MEMBER PROCEDURE F3 (p Type)); * Where A is a class</pre> 
<p>Rule 18</p> <pre>CREATE TYPE B AS OBJECT(); CREATE TABLE B_ot OF B; CREATE TYPE A UNDER B(); CREATE TABLE A_ot OF A;</pre> 	<p>Refinement (Transforming classes into associations)</p> 

4.1 UDTs' transformation

A UDT corresponds to either a class or a data type in the CS. As a criterion to distinguish both cases, we take into account whether the instances of the UDT may have an existence on their own or must always exist as part of another complex object.

Rule 1. Each UDT associated with a typed table is transformed into a UML class. The template for this rule should be read as follows. **If** the DBS contains an arbitrary UDT *A* **and** there is a typed

table that uses that same UDT *A* as a base type **then** a UML class with the same name will be generated in the CS to represent both the UDT and the typed table.

Rule 2. UDTs not appearing as the underlying type of any typed table are represented as data types in the CS.



Fig 4.1. Set of UML classes and data types resulting from the application of rules 1 and 2 over the DBS example shown in section 2.

4.2 Built-in Data Types transformation

A mapping between built-in data types in the DBS and the predefined set of data types permitted in the UML language must be determined before being able to proceed with the reverse engineering of the UDTs' attributes.

Rule 3. The correspondences between built-in data types in Oracle and the UML predefined types are:

The Char(*n*) type is transformed into a String type with an additional OCL constraint over all attributes of that type stating that the length of their String value must be exactly *n*.

The Varchar2(*n*) type is transformed into a String type with an additional constraint verifying that the maximum length of the String value is *n*.

Integer, Float and Date types are transformed into their equivalents UML Integer, Real and Date types.

The Number(*precision, scale*) is transformed into an Integer data type when *precision* is zero and into a Real type otherwise.

4.3 UDTs' Attributes transformation

UDTs' attributes may be represented by a variety of elements in the UML CS. The exact representation will depend on the complexity of the attribute (simple, complex, REF) and cardinality (univalued or multivalued).

Rule 4. Simple attributes of a UDT *A* are transformed into simple attributes of the class or data type corresponding to *A* in the CS (see rules 1 and 2). The type of the attribute in the CS is decided according to the rules seen in the previous section (plus the corresponding constraints).

Rule 5. Attributes defined as a VARRAY of one of the built-in data types (VARRAY of Integer, Float, etc), are represented as multivalued attributes in the CS. Its minimum multiplicity is zero and its maximum one coincides with the VARRAY maximum length.

Rule 6. Attributes defined as a nested table of a built-in data type are handled in the same way except for its maximum multiplicity which is now unlimited.

Rule 7. A REF attribute defined in a UDT *A* (that has been transformed into a class) and referencing a UDT *B* (that has been transformed into a class) is represented by means of an association between *A* and *B* with a 0..1 multiplicity on the *B* role. Multiplicity on the *A* role is left undefined (at least for now). Later rules may provide additional information to precise the multiplicity. If not, * multiplicity is stated. The name of *B* role is the same as REF attribute name.

Rule 8. When *A* contains an attribute that is a VARRAY of REFs to *B*, the association between *A* and *B* has as maximum multiplicity on the *B* role the one defined in the VARRAY.

Rule 9. For nested tables of REF types, the maximum multiplicity is unlimited.

Rule 10. When a UDT *A* (that has been transformed into a class) contains an attribute *attr* of type *B* and *B* has been previously transformed into a UML class in the CS, *attr* is represented as a composition relationship between *A* and *B* with a 0..1 multiplicity on the *B* role. *A* is the container, *B* the contained and *B* instances are related to exactly one *A* instance. With the composition

relationship we are able to link the lifespan of *A* instances with one of related *B* instances. As in rule 7, the name of *B* role is the same as *attr* of type *B*.

Rule 11. When *attr* is a VARRAY of objects of type *B* then the composition association from *A* to *B* has as a maximum multiplicity on the *B* side the maximum length of the VARRAY.

Rule 12. When *attr* is a nested table the maximum multiplicity is left unlimited.

Rule 13. When the type of the attribute *attr* in *A* is the UDT *B* but *B* is represented as a data type in the CS, *attr* is kept as an attribute of the class corresponding to *A*.

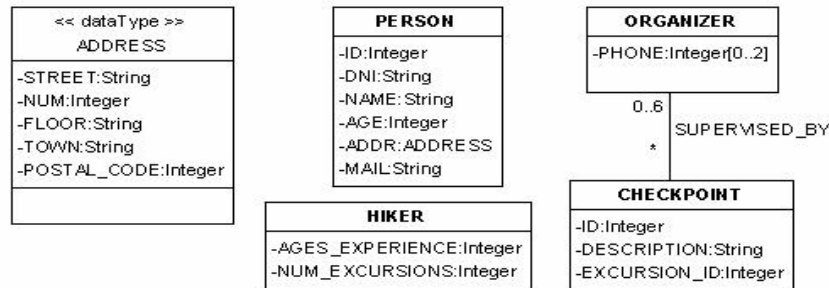


Fig 4.2. Subset of attributes and associations resulting from the application of rules 3 to 13 over the DBS example shown in section 2.

4.4 UDTs' Methods transformation

Rule 14. Each method (including static methods) in the UDT is represented as a method in the data type or class in the CS (see rules 1 and 2) with the same signature. Currently, our method does not address the reverse engineering of the method body itself.

4.5 Constraints transformation

Typed tables may include several integrity constraints that restrict the possible values that the attributes of the underlying UDT may hold. All these constraints must also be included in the CS to ensure that the CS does not permit to instantiate objects with values that would be forbidden in the DBS.

Rule 15. A NOT NULL constraint over an attribute is represented as a minimum 1-multiplicity in the corresponding element in the CS, either an attribute or an association (see section 4.3).

Rule 16. UNIQUE and PRIMARY KEY constraints are represented as an OCL constraint in the CS. The context *C* of the new constraint is the class representing the UDT over which the typed table is defined. The body of the constraint is "*C*::allInstances()->isUnique(*attr*)" where *attr* is the attribute marked as UNIQUE or PRIMARY KEY in the table. For UNIQUE or PRIMARY KEY constraints involving more than one attribute, the alternative constraint body should be used: "*C*::allInstances()->forAll(*x,y* | *x*<>*y* implies (*x*.*att1*<>*y*.*att1* or ... or *x*.*att_n*<>*y*.*att_n*))" where *att₁*...*att_n* are the set of attributes restricted by the constraint.

Rule 17. Each CHECK constraint generates a new OCL invariant (constraint) with the same expression in the constraint body (the expressivity allowed in the definition of CHECK constraints is rather limited and all possible operators have an equivalent counterpart with the same name in OCL). As an example, the OCL constraints generated for the example of section 2 are:

```

context Checkpoint inv:
  Checkpoint.allInstances->isUnique(id)
context Person inv: self.age > 10
context Person inv:
  Person.allInstances->isUnique(name)
context Address inv:
  self.floor->size()<=10
  
```

4.6 Taxonomies creation

Typical *abstract* and *complete* properties of generalizations are represented in the DBS by means of a combination of the clauses UNDER and [NOT] INSTANTIABLE. In UML, generalizations can also have the *disjoint* property (stating that a same object cannot be instance of more than a subclass at the same time). However, this property cannot be defined at the DBS level¹.

Rule 18. A UDT *A* defined as being UNDER a UDT *B* generates a generalization relationship between class *A* and class *B* with *B* as a supertype in the CS, only when UDT *A* and UDT *B* have been transformed into a class.

Rule 19. Classes for NOT INSTANTIABLE UDTs are marked as *abstract* classes. By default all UDTs are considered INSTANTIABLE.

Rule 20. When a UDT *A* is defined as UNDER of UDT *B* and *B* is NOT INSTANTIABLE, then the generalization relating *A* and *B* (generated by rule 18) is marked as *complete*. We assume that every class is superclass of at most one generalization set.

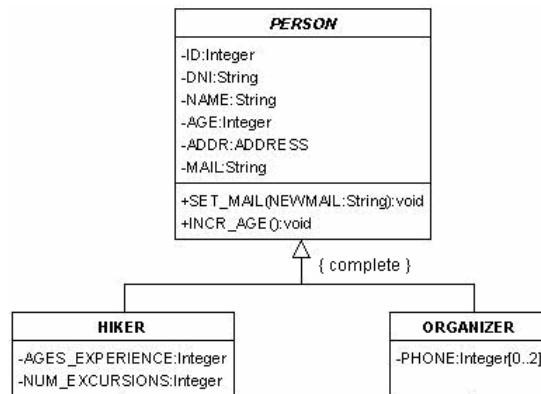


Fig 4.3. Taxonomies resulting from the application of rules 18 to 20 over the example DBS of Table 2.1.

5 Refinement of the CS

Our method allows the user to refine the resulting CS when the information from the DBS does not suffice to decide if the previous automatically obtained CS is the only (or the best) option. In those cases, our method presents several alternative options and let the user select the one he/she prefers.

In what follows we sketch the two main scenarios in which user intervention may help to improve the CS:

- *Merging associations (and compositions).* Associations at a conceptual level can be implemented in the DBS as unidirectional associations (i.e. only one UDT in the DBS will contain an attribute that references the associated object/s of the other association participant) or bidirectional ones (both UDTs will cross-reference each other). However, in this latter case and by just looking at the DBS, it is not possible to determine if the two REF attributes refer to the same bidirectional association or to two completely different associations between the same elements. The later is the safest option, and thus, the one used in our method. However, when this situation is detected, both generated associations are shown to the user so that he/she can decide to merge them into a single one.
- *Transforming classes into associations.* Any class *C* with 2 or more 1-multiplicity associations with other classes (see the pattern below) could also be represented as a new n-ary association between them. If *C* had attributes, the new association is represented as an association class.

¹ *Disjoint* property could be enforced by defining a set of triggers over each subclass table in charge of checking that the new object did not exist in the other tables.

6 Tool Implementation

A prototype of the reverse engineering method presented in this paper has been developed.

Given a set of parameters identifying an Oracle9i DB connection, our tool connects to the DB, extracts the DBS (by means of executing a set of SQL queries on its data dictionary) and generates the corresponding UML CS through an iterative application of the transformation rules, as defined in section 4.

The obtained CS is textually shown in the tool (Fig. 6.1) and it may also be stored into an output XMI (XML Metadata Interchange) [16] file to be opened and graphically displayed within a CASE tool. Due to interoperability problems among different XMI formats, we stick to the particular XMI format used by *Poseidon* [17].

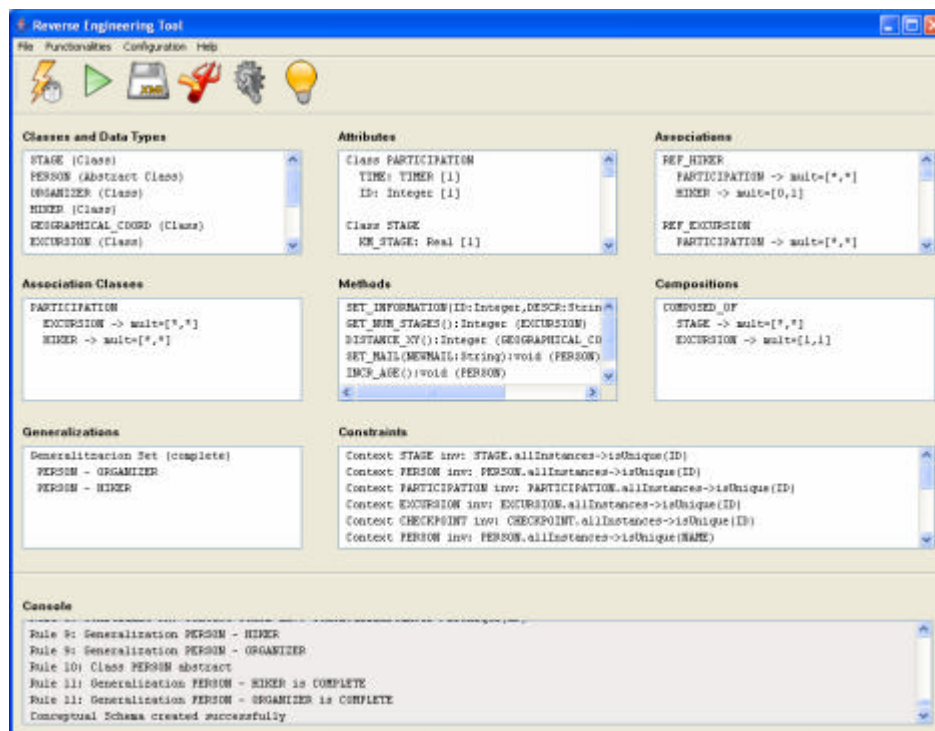


Fig 6.1. Tool interface.

7 Conclusions and Future Work

We have proposed a new method to reverse engineering ORDBS and the prototype tool supporting it. Our method is based on the execution of a set of transformation rules over the DBS extracted from the running DB by means of executing the appropriate set of queries over its data dictionary. As a result, the method provides a UML-based CS that represents the current DBS at a higher-level of abstraction.

This UML-based representation can be used to reason on and analyze the DBS more easily since all implementation details are omitted. Even when there already was an initial UML representation for the DBS, our method will be needed to resynchronize both representations (which usually diverge rapidly).

In this way, our method facilitates the maintenance, evolution and reuse of systems using an underlying ORDB to store their persistent data (in fact, most of them, since all major database vendors follow now the object-relational model).

We are planning to extend our method to address the reverse engineering of the *active* part of the database: triggers and stored procedures.

References

1. E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," in *IEEE Software*, vol. 7, pp. 13, 1990.
2. J. Hainaut, M. Chandelon, C. Tonneau and M. Joris, "Contribution to a Theory of Database Reverse Engineering," in *WCRE'03: Proc. of the Working Conference on Reverse Engineering*, 1993.
3. K. Hogshead and P. H. Aiken, "Data Reverse Engineering: A Historical Survey," in *WCRE'00: Proc. of the 7th Working Conference on Reverse Engineering* (), pp. 70, 2000.
4. W. J. Premerlani and M. R. Blaha, "An approach for reverse engineering of relational databases," in *Communications of the ACM*, vol. 37, pp. 42, 1994.
5. J. Rumbaugh, M. R. Blaha, W. J. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
6. D. W. Embley and M. Xu, "Relational database reverse engineering: a model-centric, transformational, interactive approach formalized in model theory" in *DEXA'97: Database and Expert Systems Applications*, pp. 372, 1997.
7. T. Halpin, *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. California, U.S.A, Morgan Kaufmann, 2001.
8. L. Theodoros, H. M. Edwards, A. Bryant and N. Willis, "ROMEO: reverse engineering from OO source code to OMT design" in *Proc. of the Working Conf. on Reverse Engineering (WCRE'98)*, pp. 191, 1998.
9. J. L. Hainaut, V. Englebert, J. M. Hick, J. Henrard and D. Roland, "Contribution to the Reverse Engineering of OO Applications - Methodology and Case Study" in *7th Conference on Database Semantics (DS-7)*, pp. 131, 1997.
10. E. S. Grant, R. Chennamaneni and H. Reza, "Towards analyzing UML class diagram models to object-relational database systems transformations" in *DBA'06: Proc of the 24th IASTED Int. Conference on Database and Applications*, pp. 129, 2006.
11. J. M. Vara, B. Vela, J. M. Cavero and E. Marcos, "Model transformation for object-relational database development" in *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pp. 1012, 2007.
12. OMG, "Unified modeling language: Superstructure. version 2.1" Tech. Rep. ptc/06-04-02, 2006.
13. OMG, "UML 2.0 OCL specification" Tech. Rep. ptc/03-10-14, 2003.
14. J. Melton, *Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, 2003.
15. K. Loney and G. Koch, *Oracle9i. The Complete Reference*, McGraw-Hill/Osborne, 2002.
16. T. J. Grose, G. C. Doney and S. A. Brodsky, *Mastering XML: Java Programming with XML, XML, and UML*. Wiley Computer Publishing, 2002.
17. Poseidon for UML. [Http://www. Gentleware. Com/](http://www.Gentleware.Com/)

Appendix: Case Study

1 List of SQL DDL sentences for the case study

```
-----
----- USER DEFINED TYPES -----
-----
-- Type Address
CREATE TYPE Address AS OBJECT(
    street    VARCHAR2(40),
    num       INTEGER,
    floor     VARCHAR(10),
    town      VARCHAR(30),
    postal_code INTEGER
) INSTANTIABLE FINAL;

-- Type Timer
CREATE TYPE Timer AS OBJECT(
    hours     INTEGER,
    minutes   INTEGER
) INSTANTIABLE FINAL;

-- Type Geographical_coord
CREATE TYPE Geographical_coord AS
OBJECT(
    coordX    NUMBER(2),
    coordY    NUMBER(2),
    MEMBER FUNCTION distance_XY RETURN
NUMBER
) INSTANTIABLE FINAL;

-- Type Person
CREATE TYPE Person AS OBJECT(
    id        INTEGER,
    dni       VARCHAR2(9),
    name      VARCHAR2(30),
    age       INTEGER,
    addr      Address,
    mail      VARCHAR2(30),
```

```

MEMBER PROCEDURE set_mail (newMail
VARCHAR2),
MEMBER PROCEDURE incr_age
) NOT INSTANTIABLE NOT FINAL;

-- Array of 2 phones
CREATE TYPE phones_list IS VARRAY(2) OF
INTEGER;

-- Type Checkpoint declaration
CREATE TYPE Checkpoint;

-- Type Organizer
CREATE TYPE Organizer UNDER Person(
phone phones_list,
supervises REF Checkpoint
) INSTANTIABLE FINAL;

-- Type ref to Organizer
CREATE TYPE Organizer_REF_VARRAY AS
OBJECT(
ref_Organizer REF Organizer
) INSTANTIABLE FINAL;

-- Array of 6 Organizers
CREATE TYPE Organizer_REFS_VARRAY IS
VARRAY(6) OF Organizer_REF_VARRAY;

-- Type Stage declaration
CREATE TYPE Stage;

-- Type Checkpoint
CREATE OR REPLACE TYPE Checkpoint AS
OBJECT(
id INTEGER,
description VARCHAR2(100),
excursion_id INTEGER,
supervised_by
Organizer_REFS_VARRAY,
belongs_to REF Stage,
located_in REF
Geographical_coord,
MEMBER PROCEDURE set_information (id
INTEGER, descr VARCHAR)
) INSTANTIABLE FINAL;
-- Type Excursion declaration
CREATE TYPE Excursion;

-- Type Stage
CREATE OR REPLACE TYPE Stage AS OBJECT(
id INTEGER,
km_stage NUMBER(2),
begins_at REF
Geographical_coord,
ends_at REF
Geographical_coord,
is_part_of REF Excursion
) INSTANTIABLE FINAL;

-- Nested Table of Stages
CREATE TYPE Stage_NT AS TABLE OF Stage;

-- Type Excursion
CREATE OR REPLACE TYPE Excursion AS
OBJECT(
id INTEGER,
day DATE,
excursion_km NUMBER(2),
composed_of Stage_NT,
MEMBER FUNCTION get_num_stages
RETURN INTEGER
) INSTANTIABLE FINAL;

-- Type Hiker
CREATE TYPE Hiker UNDER Person(
ages_experience INTEGER,
num_excursions INTEGER
) INSTANTIABLE FINAL;

-- Type Participation
CREATE TYPE Participation AS OBJECT(
id INTEGER,
time Timer,
ref_Excursion REF Excursion,
ref_Hiker REF Hiker
) INSTANTIABLE FINAL;

```

2 Obtained OCL Constraints

```

context Participation inv:
Participation.allInstances
->isUnique(id)

context Excursion inv:
Excursion.allInstances
->isUnique(id)

context Stage inv:
Stage.allInstances->isUnique(id)
context Checkpoint inv:
Checkpoint.allInstances
->isUnique(id)

context Checkpoint inv:
self.description->size() <= 100

context Person inv:
Person.allInstances
->isUnique(id)
context Person inv: self.age > 10

context Person inv:
Person.allInstances
->isUnique(name)

context Person inv:
self.mail->size() <= 30

context Person inv:
self.name->size() <= 30

```

context Person inv:
self.dni->size() <= 9

context Address inv:
self.floor->size() <= 10

context Address inv:
self.town->size() <= 30

context Address inv:
self.street->size() <= 40

3 Obtained CS

