Translating Relational Schemas into Object-based/XML Schemas

Abstract

This paper proposes an approach for translating an existing relational database (RDB) schema into relatively newer database (i.e., object-oriented, object-relational and XML) schemas, based on available standards. The approach is superior to existing proposals as it generates three different and equivalent target schemas. The solution takes an RDB as input, enriches its metadata representation with required semantics, and generates an enhanced canonical data model, which captures essential characteristics of the target schemas, and is appropriate for translation into any of the three target schemas. To demonstrate the effectiveness and validity of the approach, a prototype has been developed, which realises the concepts and algorithms of our approach, and generates target schemas successfully. A set of experiments has been conducted, the results of which are very encouraging showing that the proposed method is feasible, efficient and correct.

Keywords: Database models, Reverse Engineering, Object-oriented databases, Schema translation, Relational databases, Object-relational databases, XML Schema

1. Introduction

The traditional Relational DataBases (RDBs) are dominant in the marketplace as most of the data nowadays are still stored and maintained in relational database systems. However, RDBs have their shortcoming in supporting complex structures and user-defined data types provided by object-based and World Wide Web (WWW) technologies. Consequently, new breed of database management systems (DBMSs) have started to emerge in the market, providing more functionality and flexibility. The relatively newer databases, i.e., Object-Oriented DataBases (OODBs), Object-Relational DataBases (ORDBs) and XML, which all support various diverse concepts, have been proposed in order to fulfill the demands of complex applications (e.g., multimedia, geographical information systems, computer aided design, etc.) that require rich data types. Therefore, it is expected that the need to convert such RDBs into the technologies that have emerged recently will grow substantially [13, 8]. The problem is how to effectively translate an existing RDB schema as a source into more than one richer database as a target.

A general scenario for a seamless integration/conversion is to translate RDB schemas into richer schemas. Existing work does not appear to provide a solution for more than one target database. Most of existing database conversion techniques are concerned with the process of converting schema with constraints (with or without data) from a source RDB, as a one-time conversion, into one target database to be managed and handled in its new environment. This paper aims to devise a method for translating an RDB schema into object-based and XML schemas.

Schema translation process aims to map a schema of a source data model into an equivalent schema expressed in the target data model through applying a set of mapping rules [19]. The generation of a well-designed target schema depends on the flexibility of these rules. Each rule
maps a specific construct, e.g., attribute or relationship. Both schemas should hold equivalent semantics.

In this paper, we propose an approach, which takes an existing RDB as an input, enriches its metadata representation with required semantics, and constructs a Canonical Data Model (CDM) in order to generate three different output schemas. The CDM captures the essential characteristics of the target data models, for the purpose of translation. The CDM so produced is enriched with the RDB’s constraints and data semantics that may not have been explicitly expressed in the source metadata. The CDM is then mapped into target schemas according to internationally recognised standards such as ODMG 3.0, SQL4 and XML Schema, and data definition languages, leading to more portability and flexibility. Algorithms have been developed, each of which consists of a set of rules, which describe how to translate each construct of the CDM into a specific construct in the target schema. The translated target schemas might help in the heterogeneous systems, e-commerce and publishing and sharing business data. Our approach is more beneficial compared to the existing solutions as it produces three different output schemas based on the user choice, and exploits the range of powerful features provided by target data models and standards. A prototype has been implemented to demonstrate the migration process and provide proof of concept. An experimental study has been conducted to evaluate the prototype by checking the results it generates ensuring correctness and completeness of the solution and its concepts.

This paper is structured as follows. An overview of the related work is presented in Section 2. Section 3 describes how an existing RDB metadata is enriched in the form of CDM. The translation of CDM into target schemas is presented in Section 4. Section 5 evaluates our approach and reviews its results, and Section 6 concludes the paper.

2. Related Work

Significant research, to address the problem of RDB translation into newer databases, focused on the translation from RDBs into a single target database, e.g., OODB or XML. Each proposal has made certain assumptions to facilitate the process, which might be a point of a drawback or limitation. Semantic information is extracted by an in-depth analysis of relations in the relational schema together with their data dependencies into a conceptual schema model such as Entity-Relationship (ER) and its varieties, the object-oriented and XML data models. Target data models may be generated based on their conceptual schema or other representations as an intermediate stage.

Existing work for translating RDB schemas into OODBs has been extensively studied [4, 18, 19, 1]. Another group of methods for transforming UML diagrams to Oracle 9i/SQL3 ORDB schemas has been studied recently [22, 15, 21]. Most of the research on translating RDB schemas to XML is focused on generating a DTD schema [7, 13, 9, 8]. While existing works for migrating into OODBs focus on schema translation using one-to-one translation, we have noted that most works for generating XML documents have used an intermediate stage (e.g., ER model) during the translation process, focusing on generating a DTD schema. Moreover, all research on the generation of ORDBs has focused on design rather than schema translation. It could be concluded that research into the translation of RDB schema into object-based/XML schemas is still underdeveloped, therefore, several areas are in need of further consideration.

Some semantics (e.g., inheritance, aggregation) are not considered in some work. This is mainly due to their lack of support for such semantics either in the source or target data models, e.g., the ER model and DTD lack support for inheritance. Despite the ability of UML to model
data semantics such as aggregation and inheritance, UML is still weak and unsuitable for handling the hierarchical structure of the XML data model [9]. Although inheritance relationships could be indirectly realized in an RDB, they have been either ignored or only briefly considered with the exception of [12], which extracts inheritance hierarchies by means of database reverse engineering (DBRE) process.

DBRE is a process for enriching a source schema using semantics that might have not been clearly expressed by acquiring as much information as possible about objects and the relationships that exist among them [4]. This process is also known as semantic enrichment. Inferring conceptual schema from a logical RDB schema via DBRE has been extensively studied [5, 10, 14, 2]. Such conversions are usually specified by rules, which describe how to derive RDB constructs (e.g., relations, attributes, data dependencies, keys), classify them, and identify relationships among them. Semantic information is extracted by an in-depth analysis of relations in an RDB schema together with their data dependencies into a conceptual schema such as ER, UML, OO and XML data models. Data and query statements have also been used in some studies to extract data semantics. Some proposals consult expert users or use data dictionaries to provide metadata, whereas other proposals employ database design techniques. However, some of these proposals could be combined together to form a more comprehensive solution.

Inferring a conceptual schema based on the analysis of DDL and SQL queries embedded in applications has been suggested by some authors [6, 3]. [6] proposes a method called MeRCI which concentrates on schema de-optimisation. The process starts with RDB physical schema containing de-normalised relations, and then a set of appropriate rules is applied to de-optimize the schema through the analysis of application source codes (DDL, DML) and data mining techniques. Relational operators such as join, project and restrict in a physical schema are detected and used for de-normalisation of relations. [3] focuses on extracting generalisation hierarchies in an RDB using DDL, DML and data analysis. Database forward engineering (DBFE) process also known as schema mapping uses a conceptual schema generated from the DBRE process and translates it into a high level data model through the application of a set of rules, called schema mapping rules. Several proposals have been made for transforming conceptual schemas, e.g., ER, EER, UML or other specific models into object-based and XML schemas [7, 11, 15, 21]. These proposals and many others have been used as a basis for middlewares, gateways and CASE tools.

There have been fewer efforts to use standards such as the ODMG 3.0, SQL4 and XML Schema as target models [17]. The adoption of standards is essential for better semantics preservation, portability and flexibility. Furthermore, the majority of work so far has generated target schemas that are either like flat relational, in which object-based model features and the hierarchical form of the XML model are usually missed, or have deep levels of clustering/nesting, which may cause data redundancy. It would be desirable to avoid the flattened form and to reduce the levels of clustering in object structures as much as possible in order to increase the utilizations of the target models and to avoid undesirable redundancy. This requires the preservation of the semantics of the source database into a single conceptual model, which takes into account the relatively richer data model of the target database environment. In fact, the existing work on schema translation does not provide a comprehensive solution for generating more than one target schema.
3. Semantic Enrichment of Relational Database

This section provides an overview of semantic enrichment process of an RBD, which involves the extraction of its data semantics, to be enriched and converted into a much enhanced CDM. For this task, we have applied the approach in [16] for semantically enriching RDBs. The process starts by extracting the basic metadata information about an existing RDB, including relation names, attribute properties (i.e., attribute names, data types, length, default values, and whether the attribute is nullable), Primary Keys (PKs), Foreign Keys (FKs) and Unique Keys (UKs). We assume that data dependencies are represented by PKs and FKs as for each FK value there is an existing, matched PK value, which can be considered as a value reference. To get the best results, it is preferable that the process is applied to a schema in 3rd Normal Form (3NF). A relation that is not in 3NF may have redundant data, update anomalies or no clear semantics of whether it represents an entity or a relationship type. These problems may affect the real world meaning materialized in object-relational models. The next step is to identify the CDM constructs based on a classification of relations, attributes and relationships, which may involve accessing the database. Lastly, the CDM structure is generated.

Definition 1: The CDM is defined as a set of classes: \( CDM := \{ C | C = (cn, cls, abs, A_{cdm}, Rel, UK) \} \), where each class \( C \) has a name \( cn \), is given a classification \( cls \), and is flagged \( abs \) if it is abstract. Each \( C \) has a set of attributes \( A_{cdm} \), a set of relationships \( Rel \), and a set of unique keys \( UK \).

Classification (cls): our classification scheme divides classes into the following three categories:

1. Main classes (classes forming base types in the target database)
   - Regular Strong Class (RST): a class whose PK is not composed of any FKs.
   - Secondary Strong Class (SST): an inherited RST class.
   - Sub-class (SUB): a class that inherits another super-class, but is not inherited by other sub-classes.
   - Secondary Sub-class (SSC): a sub-class that is inherited by other sub-classes.
   - Secondary Relationship Class (SRC): a referenced RRC class, an M:N relationship class with attributes, or n-ary relationships where \( n \geq 2 \).
   - Regular Component Class (RCC): a weak class that participates in a relationship with other classes rather than its parent class.

2. Component classes (classes representing multi-valued/composite attributes)
   - Multi-valued Attribute Class (MAC): a class that represents a multi-valued attribute.
   - Composite Attribute Class (CAC): a class that represents a composite attribute.

3. Relationship class (a class describing an M:N relationship between two classes)
   - Regular Relationship Class (RRC): an M:N relationship class without attributes.

Abstraction (abs): A super-class is abstract (i.e., \( abs = \text{true} \)) when all of its objects are members of its sub-type objects. Instances of an abstract type cannot appear in the database extension, but are subsumed into instances of its sub-types.
Attributes ($A_{cdm}$): A class $C$ has a set of attributes $A_{cdm}$. $A_{cdm} := \{a \mid a = (a_n, t, \text{tag}, l, n, d)\}$, where each attribute $a$ has a name $a_n$, data type $t$ and a tag, which classifies $a$ as a non-key ‘NK’, ‘PK’, ‘FK’ or both PK and FK ‘PF’ attribute. Each $a$ can have a length $l$ and may have a default value $d$ where $n$ indicates whether or not $a$ is nullable (‘y’/’n’).

Relationships ($Rel$): A class $C$ has a set of relationships $Rel$. Each relationship $rel \in Rel$ between $C$ and class $C'$ is defined in $C$ to represent an association, aggregation or inheritance. $Rel := \{rel \mid rel = (RelType, dirC, dirAs, c, invAs)\}$, where $RelType$ is a relationship type, $dirC$ is the name of $C'$, and $dirAs$ denotes a set containing the attribute names representing the relationship from the $C'$ side. The $invAs$ denotes a set of inverse attribute names representing the inverse relationship from the $C$ side, and $c$ is the cardinality constraint of $rel$ from the $C$ side. $RelType$ can have the following values: ‘associated with’ for association, ‘aggregates’ for aggregation, and ‘inherits’ or ‘inherited by’ for inheritance. Relationships have two cases: 1:1 and 1:M, and $c$ is defined by $\min..\max$ notation to indicate the minimum and maximum occurrences of objects of $C'$ within objects of $C$. Based on $c$, the object(s) of $C'$ can be single-valued where $c = 0..1$ (optional) or $c = 1..1$ (required), or set-valued where $c = 0..*$ (optional) or $c = 1..*$ (required).

Unique keys ($UK$): A class $C$ may have a set of UKs (that are preserved in $UK$: $UK := \{\delta \mid \delta = \{(ua, s)\}\}$, where $\delta$ represents a key, $ua$ is an attribute name, and $s$ is a sequence number.

3.1. Generation of CDM from RDB

Using key matching, relations and their attributes are classified, relationships among relations are identified and their cardinalities are determined. All these are translated into equivalents in the CDM. The semantically enriched CDM forms the starting point for the remaining steps of the migration process that leads to the generation of the target schema and then the conversion of relational data into target data. Each relation $R$ is classified based on the comparison of its PK with the PKs of other relations, and mapped into one of the nine CDM classes above. After class $C$ is classified, it is necessary, if $C.ccls = (“SSC”)$, to check whether $C$ is concrete or abstract. $C$ is a concrete class (i.e., $abs = \text{false}$) when all (or some) of its corresponding RDB table rows are not members of other sub-tables, and abstract otherwise. Attributes of $R$ are identified and mapped along with other properties into attributes of $C$. The keys of $R$ are used to generate the relationships $Rel$ of $C$. Using this information, the relationships among relations are identified, their cardinalities determined, and a mapping is made into $Rel$ as association, inheritance or aggregation. Using the corresponding data, each relationship in which $R$ participates is identified and mapped into an equivalent relationship $rel$ and added to $Rel$.

Example 1:. Consider the RDB shown in Figure 1. PKs are in bold and FKs are in italics. Table 1 shows (partly) the resulting CDM. Each RDB relation is mapped into a class in CDM. For instance, the relation $\text{Department}$ is mapped into the CDM class $\text{Department}$, which is concrete RST class, and has the attributes: $\text{code}$, $\text{name}$ and $\text{chair}$. Other properties of the attributes (e.g., types, tags) are shown. The class is ‘associated with’ the classes: $\text{Faculty}$ (twice) and with $\text{Student}$. The cardinality $c$ and the other properties for each class are also given.

4. Schema Translation

This section explains the schema translation process. Schema translation aims to translate the CDM, produced by the semantic enrichment phase into the target schemas as shown in Figure 2.
Person (pID, dob, fname, lname)
Student (pID, status, major): pID → Person, major → Department
Faculty (pID, rank, dept): pID → Person, dept → Department
Department (code, name, chair): chair → Faculty
CampusClub (cID, name, phone, location, advisor): advisor → Faculty
Clubs (pID, cID): pID → Student, cID → CampusClub

Figure 1: Relational Schema of School Database [23]

<table>
<thead>
<tr>
<th>cn</th>
<th>cls</th>
<th>abs</th>
<th>A absorbing</th>
<th>Rel</th>
<th>RelType</th>
<th>dirC</th>
<th>dirAs</th>
<th>c</th>
<th>mAs</th>
<th>UK</th>
</tr>
</thead>
</table>
| Person         | SST         | true | pID name dob
               | int char date    | PK   | 25 | n   | n   | inherBy: Student
               |                |     | 40 | n   |   | Faculty: pID        |
| Student        | SUB         | false | pID status major
               | int char int    | PK   | 25 | n   | n   | inherBy: Person
               |                |     | 1  | y   |   | Department: pID code|
| Department     | RST         | false | code name chair
               | int char int    | PK   | 40 | n   | n   | asso: Faculty
               |                |     | 25 | n   |   | Student: dept major|

Table 1: Results of CDM generation

The target schemas hold equivalent semantics to that of the existing RDB, which are enhanced and preserved in the CDM.

4.1. Common CDM Translation Functions

The following functions are used by the three algorithms described in this section:

- `mapAttrType(tdb, a)` translates the CDM data type of an attribute `a` into an equivalent data type according the target database kind `tdb`, where `tdb` = (‘OODB’ | ‘ORDB’ | ‘XML’).
- `getCDMclass(cn)` returns the CDM class that corresponds to the class name `cn`.

Figure 2: Schematic view of translating CDM into target schemas
getRelationshipName \((x, X, y, Y)\) returns a concatenation of a class name \(x\) and a set of attribute names \(X\) with a class name \(y\) and a set of attribute names \(Y\) to derive a unique name for a relationship.

setMtoNrelationship\((tdb, rel)\). Given that an RDB cannot implement M:N relationships directly, this function resolves an M:N relationship into two 1:Ms. That is, if a CDM class \(C\) has a 1:M relationship \(rel\) \((i.e., \, rel \in C.REL)\) with a class \(C'\), where \(C'.cls = 'RRC'\) and \(C\) has a 1:M relationship with a class \(C''\), then depending on \(tdb\), the function defines a new M:N relationship between the target classes corresponding to \(C\) and \(C''\). This means that there will be no target class corresponding to \(C'\).

mapNonFKtype\((tdb, A_{cdm})\) translates the data type of a non-foreign key attribute \(a \in A_{cdm}\) into a target equivalent type in \(tdb\).

mapAttAndType\((tdb, A_{cdm})\) takes the name and data type of every non-foreign key attribute \(a \in A_{cdm}\) and translates them into the equivalent target types in \(tdb\) in the form of a structured type, e.g., \struct{\text{when } tdb = 'OODB' \text{ or row when } tdb = 'ORDB'}\.

4.2. Translating CDM into OODB Schema

Definition 2: A target ODMG 3.0 schema is defined as a set of classes:

\[
\text{OODSchema} := \exists \{ \text{Class}_{oo} \mid \text{Class}_{oo} = \{c_{oo}, \text{spr}_{oo}, k_{oo}, \text{REL}_{oo}\}\}, \quad \text{where } c_{oo} \text{ is the name of a class } \text{Class}_{oo}, \text{ spr}_{oo} \text{ is the name of its super-class, } k_{oo} \text{ is its key, } A_{oo} \text{ is a set of its attributes, and } \text{REL}_{oo} \text{ is a set of relationship types in which } \text{Class}_{oo} \text{ participates. The sets } A_{oo} \text{ and } \text{REL}_{oo} \text{ are defined as follows:}
\]

- \(A_{oo} := \{a_{oo} \mid a_{oo} = (a_n, t, m)\}, \text{ where } a_n \text{ is the name of an attribute } a_{oo}, t \text{ is its data type, and } m \text{ denotes whether } a_{oo} \text{ is single-valued ('sv') or collection-valued ('cv')}\).
- \(\text{REL}_{oo} := \{\text{Rel}_e \mid \text{Rel}_e = (rel_e, \text{dirC}_e, m, \text{invRel}_e)\}, \text{ where } \text{rel}_e \text{ is the name of the relationship } \text{Rel}_e, \text{ dirC}_e \text{ is the name of the referenced class, } m \text{ indicates the multiplicity of } \text{Rel}_e, \text{ and } \text{invRel}_e \text{ is the name of the inverse relationship.}\)

4.2.1. The ProduceOODBSchema Algorithm

Given the CDM as defined in Section 3 and the ODMG 3.0 ODL as defined above, this section explains how the CDM is translated into an equivalent target OODB schema. A set of rules have been designed for translating CDM classes into corresponding ODL constructs. The ProduceOODBSchema algorithm shown in Figure 3 implements these translation rules. Each rule is aimed at a specific construct, e.g., class creation or attribute translation, which may be supported by mapping functions, e.g., mapAttType.

Translating Classes: For each CDM class \(C \in cdm\), where \(C.cls \neq ('MAC' \mid 'CAC' \mid 'RRC')\), an OODB class \(C_{oo}\) (of type \text{Class}_{oo}) is created with its own properties, and added to the target schema targetSchema. The properties of \(C_{oo}\) are extracted from \(C\), include its key \(k\), super-class name \text{spr}_{oo}\, attributes \(A_{oo}\), and relationships \text{REL}_{oo}, whereas the name of \(C_{oo}\) is assigned to be that of \(C\). The \(C_{oo}\) is defined under its super-class \(spr\) if \(C.cls = ('SUB' \mid 'SSC')\) as described in translating inheritance. The \(k\) is specified for strong classes, i.e., \(C.cls = ('RST' \mid 'SST')\) using the defineClassKey function (line 9), when one or more attributes of \(C\) are tagged with ‘PK’. Sub-classes inherit their super-class keys. However, the OODB class must have an ‘extent’ to have a key. In ODMG 3.0, an extent defines the set of all instances of a given OODB class. The class name is appended by the letter ‘s’ to represent the class extent, e.g., Person.7
```
1: algorithm ProduceOODBschematic (cdm: CDM) return OOschema
2:  targetSchema: OOschema := ∅ // a set to represent the OODB target schema
3:  foreach class C ∈ cdm do
4:    if C.cls ≠ ('MAC' | 'CAC' | 'RRC') then
5:      Coo::Classoo := define an OODB class as (cn, spr, k, Aoo, RELoo)
6:      mlt: string := 'sv'
7:      relname, invRelname: string := ''
8:      Coo.cn := C.cn
9:      Coo.k := defineClassKey(C, cn, C.cls)
10:     foreach attribute att ∈ C.Acdm do
11:       if att.tag ≠ ('FK' | 'PF') then
12:         Coo.Aoo := {⟨att.name, mapAttAndType('OODB', att), mlt⟩}
13:       end if
14:     end for
15:     foreach relationship rel ∈ C.REL do
16:       C′ := getCDMclass(rel.dirC)
17:       relname := getRelationshipName(rel.dirC, rel.dirAs, C.cn, rel.invAs)
18:       if rel.c = (0..1 | 1..1) then
19:         mlt := 'sv'
20:       else
21:         mlt := 'cv'
22:       end if
23:       if rel.relType = 'associated with' then
24:         if C′.cls ≠ 'RRC' then
25:           Coo.RELoo := setMtoNrelationship('OODB', rel)
26:         else
27:           invRelname := getRelationshipName (C.cn, rel.invAs, rel.dirC, rel.dirAs)
28:           Coo.RELoo := {⟨rel.name, rel.dirC, mlt, invRelname⟩}
29:         end if
30:       else if rel.relType = 'aggregates' then
31:         if C′.cls ≠ 'MAC' then
32:           Coo.Aoo := mapNonFKtype('OODB', C′.Acdm, mlt)
33:         else
34:           Coo.Aoo := mapAttAndType ('OODB', C′.Acdm, mlt)
35:         end if
36:       else if rel.relType = 'inherits' then
37:         Coo.spr := rel.dirC
38:       end if
39:     end for
40:     targetSchema := targetSchema ∪ {Coo} // add the class to OODB schema
41:   end if
42:  end for
43: return targetSchema
44: end algorithm
```

Figure 3: The ProduceOODBschematic Algorithm
Translating Attributes. Attributes are mapped according to two rules: a basic rule and a complex rule. The basic rule applies to the primitive attribute types. Each CDM attribute \( att \in C.A_{cdm} \), where \( att.tag \neq \{ \text{FK} \mid \text{PF} \} \), is translated into an equivalent OO attribute and placed in the attribute set \( A_{oo} \) of \( C_{oo} \) with the same name as that of \( att \) using the mapAttrType function (lines 10-14). The complex rule has been designed for mapping aggregation relationships, resulting in simple/composite multi-valued attributes as described in translating aggregations.

Translating Relationships. Each relationship \( rel \) defined in a CDM class \( C \) (\( rel \in C.REL \)) is translated into an equivalent relationship in the corresponding OODB class \( C_{oo} \) (lines 15-39). The type of the target relationship and its multiplicity are determined by the classification of the CDM class \( C' \) (i.e., \( C'.cls \)) related to \( C \) and the properties of \( rel \) (e.g., \( rel.relType \)). OO association, aggregation and inheritance relationships are derived from \( rel \) when \( rel.relType= \\text{‘associated with’}, \text{‘aggregates’} \) and \text{‘inherits’}, respectively. Moreover, the cardinality \( c \) of \( rel \) is mapped into a single-valued multiplicity \( \text{‘sv’} \) when \( c = (0..1 | 1..1) \), or a collection-valued multiplicity \( \text{‘cv’} \) otherwise.

- **Association**: Each relationship \( rel \in C.REL \), where \( rel.relType= \text{‘associated with’} \), is translated into a corresponding bi-directional OO association relationship (lines 23-29). The relationship is represented by a pair of inverse references to ensure navigation in both directions and to preserve the referential integrity constraints. The direct relationship name \( rel\text{name} \) of the new relationship and also its inverse relationship name \( invRel\text{name} \) are obtained using the getRelationshipName function, as described above. The \( rel \) is mapped into a single-valued \( \text{‘sv’} \) relationship referencing the related OO class, corresponding to \( C' \) if \( rel.c = (0..1 | 1..1) \), and into a collection-valued \( \text{‘cv’} \) relationship if \( rel.c = (0..* | 1..*) \).

- **Aggregation**: Each relationship \( rel \in C.REL \) where \( rel.relType= \text{‘aggregates’} \) is translated, in \( C_{oo} \), into an OO literal attribute (lines 30-35). The type of the attribute is mapped from the component \( C' \) that participates in the relationship with \( C \), whereas its multiplicity \( mlt \) is derived from \( rel.c \). The classification of \( C' \), i.e., \( C'.cls \) determines the type of the attribute. When \( C'.cls = \text{‘MAC’} \), then \( rel \) is mapped into a multi-valued attribute, where its type is obtained by the mapNonFKtype function. However, if \( C'.cls = \text{‘RRC’} \), then \( rel \) is mapped into an M:N relationship using the setMtoNRelationship function, in which a pair of 1:M relationships is defined in each of the OODB classes corresponding to the two CDM classes that participate in the relationship with \( C' \) (lines 24-25).

- **Inheritance**: Each relationship \( rel \in C.REL \) where \( rel.relType= \text{‘inherits’} \) is mapped into a simple inheritance, by which the sub-class \( C_{oo} \), mapped from \( C \), inherits all of the properties of its super-class mapped from \( C' \) (lines 36-37). The super-class name \( C_{oo}.spr \) is assigned to \( C'.cn \) to realise the inheritance between \( C_{oo} \) and its super-class. Additional attributes and relationships of \( C \) are mapped to \( C_{oo} \) in the usual way.

4.3. Translating CDM into ODBR Schema

**Definition 3.** The SQL4 ORDB schema is denoted as 3-tuple:

\( \text{ORschema} := (UT, TT, UK_{oo}) \), where \( UT \) is a set of user-defined types (UDTs), \( TT \) is a set of typed tables, and \( UK_{oo} \) is a set of unique keys. The sets \( UT \) and \( TT \) are defined as follows:
Each non-foreign key attribute is translated into the target ORDB schema as association, aggregation and inheritance. Each relationship \( rel \in C.REL \) is translated, based on \( rel.relType \) and the classification of the related CDM class \( C' \), into a relationship attribute and added into \( A_{ut} \) of the corresponding type, or mapped into an inheritance relationship (lines 19-43). The relationship attribute name \( rel.name \) for association/aggregation relationships is generated using the \( getRelationshipName \) function (line 21), whereas its multiplicity \( mt \) is single-valued ‘sv’ when \( rel.c = (0..1 | 1..1) \), or collection ‘cv’ when \( rel.c = (0..m | 1..m) \) (lines 22-26).
1: algorithm ProduceORDBschema (cdm: CDM) return ORschema
2: aUT: UT := ∅
3: aTT: TT := ∅
4: aUKinf, UKsup := ∅
5: foreach class $c \in$ cdm do
6:   if $c.cls \neq$ (‘RRC’ | ‘MAC’ | ‘CAC’) then
7:       udt: udtType // define a UDT as ($uut$, $suc$, $Au$)
8:       $T_{ut}$: CreateTable // define a typed table as ($tu$, $utuc$, $puc$, $pk$, $uoid$)
9:       $mt$, $rename$, $nl$: string := ‘’
10:      udt.$utuc$, $T_{ut}.utuc$ := $C.cn$+ ‘$J$’
11:      $T_{ut}.utuc$ := $C.cn$
12:      $T_{ut}.pk$ := definePKconstraint($ccls$, $C.Acdn$)
13:      $T_{ut}.uoid$ := defineUserDefinedOID($ccls$) // specify uoid
14:   end if
15: end foreach
16: end algorithm
• **Association:** Each relationship \( rel \in C.\text{REL} \) where \( rel.\text{relType} = \text{‘associated with’} \) is translated, in the \( udt \) corresponding to \( C \), into an attribute where its type is a \( \text{ref} \) or a collection of \( \text{refs} \) (depending on \( rel.c \)), referencing the related UDT mapped from the corresponding \( C’ \) (lines 27-32). A \( \text{ref} \) attribute is constrained to be scoped (i.e., using a \textit{scope} clause) to a specific table, so that the \( \text{ref} \) values stored in that attribute points to objects of the specified table. Each association is defined bi-directionally between its two types. However, unlike the OODB systems that support ODMG 3.0, neither SQL4 nor any ORDB products allow the user to rely on the system to automatically enforce inverse relationships. The inverse direction of the relationship is defined in related UDT. Depending on the cardinalities \( c \) of \( rel \) and the classification of associated class \( C’.\text{cls} \), \( rel \) is translated into 1:1, 1:M or M:N relationships. The \( rel \) is mapped, in \( udt \), into a single-valued attribute of \( \text{ref} \) type, pointing to a pre-defined type corresponding to \( C’ \) (e.g., \( udt’ \)) when \( rel.c = (0..1 \mid 1..1) \). In addition, \( rel \) is mapped into a collection-valued attribute, containing a collection of \( \text{refs} \), pointing to a related type \( udt’ \) if \( rel.c = (0..m \mid 1..m) \). However, \( rel \) is mapped into an M:N relationship if \( C’.\text{cls} = \text{‘RRC’} \) using the \textit{setMtoNrelationship} function. As \( C’ \) participates in only two M:1 association relationships with \( C \) and another CDM class \( C” \), \( rel \) is mapped, in \( udt \), into an attribute that contains a collection of \( \text{refs} \), pointing to a pre-defined \( udt” \), corresponding to \( C” \). Similarly, a collection of \( \text{refs} \) is defined inside \( udt” \) when mapping from the \( C” \) side.

• **Aggregation:** Each relationship \( rel \in C.\text{REL} \) where \( rel.\text{relType} = \text{‘aggregates’} \) is translated, in \( udt \) mapped from \( C \), into a literal type attribute, representing \( C’ \) (lines 33-38). Depending on \( c \) of \( rel \) and \( C’.\text{cls} \), \( rel \) is translated into multi-valued attributes or \textit{row} types. If \( C’.\text{cls} = \text{‘MAC’} \), \( rel \) is mapped into a multi-valued attribute, where the \textit{mapNonFKtype} function returns the data type of the attribute from \( C’.A_{\text{cls}} \). However, if \( C’.\text{cls} = \text{‘CAC’} \), \( rel \) is mapped into a \textit{row} type attributes when \( rel.c = (0..1 \mid 1..1) \), or as a collection of \textit{rows} when \( rel.c = (0..m \mid 1..m) \). The attributes of the \textit{row} type are mapped from the non-foreign key attributes in \( C’.A_{\text{cls}} \) using the \textit{getAttAndType} function.

• **Inheritance:** Each relationship \( rel \in C.\text{REL} \) where \( rel.\text{relType} = \text{‘inherits’} \) is mapped as a single inheritance, where \( udt \) translated from \( C \) inherits all of the properties of its super-type mapped from \( C’ \). The name of \( C’ \) is appended by the string ‘.\ ‘ and assigned to super-type name attribute \( s_{\text{tag}} \) of \( udt \) to realise the inheritance, i.e., \( udt.s_{\text{tag}} = C’. cn+’ .\ ‘ \) (line 40). Additional properties of \( udt \) are defined in the usual way. Creating a sub-type under its super-type is considered while creating the super-type by specifying the \textbf{not final} phrase at the end of the super-type definition, which is \textbf{final} by default. Specifying \textbf{not final} for a super-type in the \textit{create type} statement means that other types can inherit from it.

Creating Typed Tables. A typed table \( T_{\text{or}} \) (of type \textit{iTable}) is defined for each declared \( udt \) and labeled with the same name as the corresponding CDM class \( C \), from which the \( udt \) has been translated, i.e., \( T_{\text{or}}.t_{\text{or}} = C.cn \). Creating \( T_{\text{or}} \) is based on UDT specifications, representing instances for each row in the table, i.e., \( T_{\text{or}}.u_{\text{or}} = C.cn+’ .\ ‘ \). The primary key \( pk \) of \( T_{\text{or}} \) is defined using the \textit{definePKconstraint} function, which produces \( pk \) from the attributes in \( C.A_{\text{cls}} \), where \( \text{tag} = \text{‘PK’} \) (line 12). Because \( T_{\text{or}} \) would contain objects that can be referenced by other objects, an identifier column \( T_{\text{or}}.u_{\text{oid}} \) is specified as user-generated OID to facilitate cyclic referencing among pre-created objects during data loading. When inserting a tuple in \( T_{\text{or}} \), the \( uoids \) of objects can be generated from the primary key values of the corresponding RBD table. The function...
defineUserDefinedOID is used to define the uoid (line 13). However, sub-tables inherit primary keys and uoids from their super-tables. Unique keys for each table are extracted from the CDM equivalents and placed in $aUK_n$ using the function defineUKconstraints (line 46). Sub-tables are created under their super-tables via $T_{or,$$n} = C'.$cn, where $C'.$cn is the name of the corresponding super-class $C'$ (line 41).

4.4. Translating CDM into XML Schema

**Definition 4.** The target XML Schema is denoted as 2-tuple: $\langle $Root, $GT$ $\rangle$, where $Root$ is a global element declared under the schema with its direct local elements and constraints, representing the XML document tree, and $GT$ is a set consisting of global complex types. The types in $GT$ are defined as types of the elements declared in $Root$, or to be referenced by other complex types in $GT$. The $Root$ and the set $GT$ are defined as follows:

- $Root = \langle root_n, LE, PK_x, FK_x, UK_x \rangle$, where $Root$ has a name $root_n$, a set of elements $LE$, and three sets of identity-constraints $PK_x, FK_x$ and $UK_x$.
  
  - $LE$ represents the complex type of $Root$ that involves a set of local sub-element declarations: $LE := \{ e \mid e = \langle e_n, e_i, min, max \rangle \}$, where each element $e$ has a name $e_n$, a type $e_i$, and a minimum $min$ and maximum $max$ occurrences. The $e_i$ is defined globally under the schema, i.e., in the set $GT$.
  
  - $PK_x$ is a set of primary keys for the elements defined in the $Root$, where $PK_x := \{ pk \mid pk = \langle pk_n, selector, PKfield \rangle \}$. Each key $pk$ has a name $pk_n$, an element set $selector$ as a scope within which the key is defined, and a set of related sub-elements $PKfield$ selected to be unique.
  
  - $FK_x$ is a set of foreign keys, where $FK := \{ fk \mid fk = \langle fk_n, ref, selector, FKfield \rangle \}$. Each foreign key $fk$ has a name $fk_n$, an element set scope $selector$, a reference constraint name $ref$ that points to a matched primary key name, and a set of related sub-elements $FKfield$.
  
  - $UK_x$ is a set of unique keys, where $UK := \{ uk \mid uk = \langle uk_n, selector, UKfield \rangle \}$. Each unique key $uk$ has a name $uk_n$, an element set scope $selector$, and a set of related sub-elements $UKfield$ selected to be unique.

- $GT := \{ compType \mid compType = \langle ct_n, base, abst, LE \rangle \}$, where $ct_n$ is the name of a complex type $compType$, $base$ is the name of its super-type (if it is derived from another type), $abst$ denotes whether or not $compType$ is abstract type, and $LE$ is a set of elements that are declared locally within $compType$.

$LE := \{ e \mid e = \langle e_n, e_i, min, max \rangle \}$ is defined as for $Root$; however, $e_i$ can be a built-in data type (e.g., a string) or a complex type pre-defined in the set $GT$.

4.4.1. The ProduceXMLschema Algorithm

An XML Schema file (.xsd) can be created based on the additional semantics captured in CDM. This section explains how to translate CDM constructs into an XML Schema. The translation process involves a set of mapping rules, which translate CDM into XML Schema annotations. An algorithm called ProduceXMLschema, given in Figure 5, has been developed for this purpose and represents an integration of these rules. The main steps of the algorithm are described as follows:
Figure 5: The ProduceXMLschema Algorithm
**Defining XML Namespaces.** XML Schema documents have main components, e.g., complex type definitions, and secondary components, e.g., namespaces, annotations and language used. The secondary must be defined in the first step in order to create an XML Schema document. A namespace is defined according to the standard for schema commands and assigned to the variable, e.g., `xs:xml:lang` as an XML Schema description using the attribute `xml:lang` (XML namespace). All schema tags are prefixed by `xs:` to indicate the XML Schema namespace. However, the namespace can be defined as a default, where the use of such a prefix is not needed. Any necessary annotations used in the document have to be specified for the user and machine, e.g., `English Language (xml:lang = 'en')`.

**Declaring Schema Root and its Elements.** An XML Schema is defined based on a tree data model, which has two main components. The first component is the declaration of the root element as a complex type that contains a sequence of elements and integrity constraints. The second component includes the definitions of the complex types of the elements declared under the root. Three common approaches are available to make the decision on how to define the schema's components, locally or globally. These approaches are Salami Slice, Russian Doll and Venetian Blind designs [20]. Each approach can be adopted based on the application's requirements. In this paper, the target XML Schema is produced according to the Venetian Blind design, which defines complex types globally and elements locally. This offers flexible component reusing and nest element declarations within type definitions.

After defining the namespace and annotations, the root element `aRoot` of the schema document is created and given a name `rootn` with the same name as an existing RDB schema (or alternative provided by the user). The subsequent steps of the algorithm define the set of elements of the root `aRoot.LE` and its identity-constraints `PK`, `FK`, and `UK`, and then specify the set of global complex types `aGT`. The target XML Schema document is generated from `aRoot` and `aGT`. Each CDM main and concrete class `C ∈ cdm` (i.e., `C.abs` and `C.cls ≠ ('MAC' | 'CAC' | 'RRC')`) is translated as an empty first-level element under the root – placed in `aRoot.LE`. Each element `∈ aRoot.LE` is named with the same name as the corresponding CDM class, i.e., `C.cn` and has a type specified by adding the `.'t'` string to its name, i.e., `C.cn + '.'t'`. The type name is used as a reference to a global complex type that is defined separately in the set `aGT`. The occurrence of each element is specified using the occurrence attributes `mn = “0”` and `mx = “unbounded”`. The `PK`, `FK`, and `UK`, are defined for each element as described in Section 4.4.1.

**Defining Complex Types.** Each CDM class `C`, where `C.cls ≠ ('MAC' | 'RRC')` is translated into a global complex type `ct` (of type `CompType`) (lines 6-48). The name `ct`, of `ct`, is specified from the corresponding CDM class name `C.cn`, concatenated with the string `.'t'`. For example, the `Person` class is mapped into the `Person` element that has the `Person.t` complex type. If `C` is abstract, i.e., `C.abs = true`, then `ct` is specified as abstract, i.e., `ct.abs = C.abs`. The set of elements `ct.LE` is constructed from `CA_{cdm}` and `C.REL`. Each attribute `att ∈ CA_{cdm}` is translated into a local element and added to `ct.LE` (lines 11-23). Each element is given a name as the same name of the corresponding `att`, and a type translated via the `mapAttrType` function. The occurrences `mn` and `mx` of the element, are set to default values where each is “1” since they are all of the primitive type. However, `mn` is set to “0” if `att` accepts nulls (i.e., `att.n = ‘y’`). Foreign key attributes (i.e., tagged as ‘FK’ or ‘PF’) are defined in `ct` as normal simple attributes if they are specified in `PK`, and `FK`, sets; otherwise, they are dropped from the definition of `ct` (e.g., foreign key attributes in CAC classes). In other words, each attribute `att ∈ CA_{cdm}` is mapped into a local
An XML Schema represents relationships among elements, where $C.cls = ('CAC' \mid 'SUB' \mid 'SSC')$ and $att.tag \neq 'PF$'; whereas all attributes of $C$, where $C.cls = ('RST' \mid 'RCC' \mid 'SRC$) are mapped in $ct$ into local elements. Other non-primitive elements are mapped from $C.REL$ as described next.

Translating Relationships and Constraints. An XML Schema represents relationships among elements by specifying nested complex types, or constraints using the `key/keyref`. On the one hand, the definition of nested types follows the parent-child containment technique, which in most cases causes data redundancy, even though it may speed up the processing of queries by avoiding join operations. Moreover, nesting the elements under each other requires the user’s help, during the translation. On the other hand, defining relationships using `key/keyref` may result in a flat document, even though the documents generated using this technique contain less redundancy. Therefore, we follow each of these two techniques with the aim to produce less data redundancy. Therefore, we follow each of these two techniques with the aim to produce less data redundancy. Thus, relationships among main CDM classes are mapped into identity constraints using the `key/keyref`, whereas the MAC, CAC and RRC classes are translated as nested elements under their parent elements.

Identity Constraints: The sets $PK_x$, $FK_x$ and $FK_t$ are declared for candidate elements defined in $aRoot$ from each corresponding CDM class $C$ using $C.A_{cdm}$ and $C.REL$ (lines 25-27).

The following functions are defined to return the three sets:

- **definePK(C)** returns the primary key $pk$ for each element defined under $aRoot$ in the form $\langle pk_a, selector, PK field \rangle$, and adds it into the $PK_x$ set. The $pk$ element is translated from each attribute $att \in C.A_{cdm}$, where $C.cls = ('RST' \mid 'SRC' \mid 'RCC$) and $att.tag = ('PK' \mid 'PF$). To guarantee their uniqueness each key name $pk_a$ is formed by concatenating $C.cn$ with each $att.a_n$ and the string ‘PK’. A selector is assigned $C.cn$ as a constraint element scope, whereas $PK field$ is specified from each $att \in C.A_{cdm}$ when $att.tag = ('PK' \mid 'PF$) as related element(s) to the selected selector. The $PK field$ can have more than one element in the case of a composite key. For example, the primary key code of Department class is translated into the XML primary key as "departmentCodePK", Department, {code}). However, if $C.cls = ('SUB' \mid 'SSC$), the corresponding $PK field$ contains the key attributes of the top super-type of the inheritance hierarchy.

- **defineFKs(C)** returns foreign keys for each element defined under $aRoot$ in the form $(fk_a, ref, selector, FK field)$ and adds them to the $FK_x$ set. XML foreign keys are mapped from each CDM relationship $rel \in C.REL$, where $rel.relType = 'associated with'$ and the attribute names in $rel.invAs$ are either tagged as ‘FK’ or ‘PF’. The foreign key name $fk_a$ is formed by the concatenation of $C.cn$, with the name of each attribute in $rel.invAs$, and the string ‘FK’. The refer is formed from concatenating the $rel.dirC$, with the attribute names in $rel.dirAs$ and the string ‘PK’, whereas the selector is named as $C.cn$, and each element in $FK field$ is assigned an attribute in $rel.invAs$.

- **defineUKs(C)** returns the unique keys for elements defined under $aRoot$ and adds them to the set $UK_x$, based on their equivalents in CDM, i.e., $C.UK$.

Nested Elements: The following rules are used to translate CDM relationships into XML as sub-elements embedded within their parent elements (lines 29-44).

- Each association $rel \in C.REL$ between $C$ and another CDM class $C'$, where $C'.cls = 'RRC$', is translated using the `setMtoNrelationship` function into a multi-valued element in
a complex type ct of the element translated from C. As C′ participates in only two M:1 associations with C and another CDM class C′′, rel is mapped in ct into a multi-valued element, referencing a complex type ct′′, corresponding to C′′. A foreign key is defined for the new sub-element through its parent element and added to the set FKx, referencing the primary key of the element corresponding to C′′. Similarly, a multi-valued element with its foreign key is defined in ct′′, referencing ct.

- There are two different mapping rules that can be applied when rel ∈ C.REL represents an aggregation relationship, i.e., where rel.relType = ‘aggregates’, between a parent class C and a component class C′. A sub-element corresponding to C′ is defined and embedded in the parent complex type ct, mapped from C, representing this relationship. The name of the sub-element is generated by the getRelationshipName function, and its occurrences mn and mx are declared according to the corresponding cardinality rel.c. If C′.cls = ‘MAC’, rel is mapped into a simple multi-valued sub-element, the type of which is extracted via the mapNonFKtype function. However, if C′.cls = ‘CAC’, rel is mapped into a multi-valued sub-element. The type of the sub-element is defined as a global complex type ct′ and added into aGT, as described in Section 4.4.1.

Inheritance:. Each relationship rel ∈ C.REL defined in a class C that inherits another CDM class C′ where rel.relType = ‘inherits’, is mapped as an inheritance. A complex type ct corresponding to C is defined as an extension of its a complex type ct′ corresponding to C′. This realises an XML inheritance between ct and ct′, where ct.base = C′.cn (lines 41-42).

5. Validation

To demonstrate the effectiveness and validity of our approach, a prototype has been developed to realize its algorithms. The algorithms were implemented using Java 1.5. To evaluate the scalability and correctness of the approach, an experiment has been designed to observe any differences between the source RDB schemas and target schemas produced by the prototype. This section describes the experiment, which tests schema information preservation by comparing target schemas generated from the prototype with that translated from the same source schemas using an existing manual schema mapping technique (i.e., [23]). The evaluation includes comparisons of the schema structures, data semantics and integrity constraints.

5.0.2. OODB schemas

Urban and Dietrich [23] used UML features to illustrate mapping alternatives from UML to relational, object-oriented and object-relational data models. The approach used an RDB called School to compare and contrast mapping techniques specific to each model. Figure 1 shows the RDB logical School schema they used.

Figure 6 shows the ODMG 3.0 ODL schema mapped from [23] and Figure 7 shows the equivalent schema generated by our prototype. In the figures, it can be observed that, apart from methods associated with classes, which are not assumed in our approach, the results of the prototype and Urban and Dietrich approach were similar in translating the RDB schema into its equivalent OODB schema, including classes, attributes, single/collection-based relationships, inheritance, and keys. However, association relationships are mapped in our approach bi-directionally and aggregation relationships uni-directionally. Such relationships were modelled as bi-directional and sometimes uni-directional relationships in Urban and Dietrich work, e.g., the deptChair attribute.
defined in the Department class. Another example is that the major association relationship in Student class is translated by our prototype bi-directionally, with its inverse the students relationship in the Department class. Such a relationship is mapped by Urban and Dietrich as major attribute in the Student class whose type is a single instance of the Department class, and the students attribute in the Department class was defined as a collection of students.

```plaintext
class Person( extent people key pID) {
    attribute string pID; attribute date dob;
    attribute string firstName; attribute string lastName; . . . );
class Faculty extends Person(extent facultyMembers) {
    attribute string rank; attribute Department dept;
    relationship set<CampusClub> advisorOf inverse CampusClub::advisor;
    Department getChairOf(); . . . );
class Student extends Person(extent students) {
    attribute string status; attribute Department major;
    relationship set<CampusClub> memberOf inverse CampusClub::members; . . .
    class Department( extent departments key code) {
        attribute string code; attribute string name; attribute faculty deptChair;
        attribute set<Student> students; attribute set<Faculty> deptFaculty; . . .
    class CampusClub( extent campusClubs key cID) {
        attribute string cID; attribute string name; attribute string phone;
        attribute string location;
        relationship set<Student> members inverse Student::memberOf;
        relationship Faculty advisor inverse Faculty::advisorOf.
    class department (extent departments key code) {
        attribute string code; attribute string name;
        relationship set<faculty> deptFaculty inverse faculty::dept;
        relationship set<Student> students inverse student::major;
        class Faculty extends person (extent facultys) {
            attribute string rank;
            relationship set<CampusClub> advisorOf inverse campusclub::advisor;
            relationship department chairOf inverse department::deptchair;
            relationship department dept inverse department::deptfaculty);
        class person (extent persons key pid) {
            attribute string pid; attribute date dob; attribute string firstname;
            attribute string lastname;}
        class student extends person (extent students) {
            attribute string status;
            relationship set<CampusClub> memberof inverse campusclub::members;
            relationship department major inverse department::students;
}
```

Figure 6: Fragment of OODB School schema mapped from [23]

```plaintext
class campusclub (extent campusclubs key cid) {
    attribute string cid; attribute string name; attribute string phone;
    relationship set<student> members inverse student::memberof;
    relationship faculty advisor inverse faculty::advisorof.
    class department (extent departments key code) {
        attribute string code; attribute string name;
        relationship set<faculty> deptFaculty inverse faculty::dept;
        relationship set<Student> students inverse student::major;
        class Faculty extends person (extent facultys) {
            attribute string rank;
            relationship set<CampusClub> advisorOf inverse campusclub::advisor;
            relationship department chairOf inverse department::deptchair;
            relationship department dept inverse department::deptfaculty);
        class person (extent persons key pid) {
            attribute string pid; attribute date dob; attribute string firstname;
            attribute string lastname;}
    class student extends person (extent students) {
        attribute string status;
        relationship set<CampusClub> memberof inverse campusclub::members;
        relationship department major inverse department::students;
}
```

Figure 7: Fragment of OODB School schema generated by our prototype

5.0.3. ORDB schemas

Figure 8 shows the ORDB SQL3 schema mapped by Urban and Dietrich approach and Figure 9 shows the equivalent schema generated by our prototype.

Considering the differences in SQL syntax between Oracle and the SQL3 standard, the same schemas, including relations and constraints are translated by our approach and the approach of Urban and Dietrich into equivalent ORDB object types, including attributes, references among objects and nested tables/arrays, and object tables. Similarly in both approaches, object types,
which may be defined as inheritance hierarchies, have been used to create object tables and primary keys are defined for those tables. However, RDB relationships are translated by our approach into object-based relationships bi-directionally. The 1 side of relationships is translated as ref and the M side as nested tables. Nested tables are especially appropriate for modelling association/aggregation collection data types. Urban and Dietrich used an SQL3 array of refs to map the multi-valued relationships [23], and proposed using triggers for referential integrity maintenance in the ORDB schema [23].

In summary, in this experiment, we have compared our approach with existing manual mapping approaches. These techniques give the user an opportunity to use all features of target models and their conceptual schemas, resulting in well-designed physical schemas. By comparing the results of these techniques with the results of our prototype, we found that both sets of results were comparable. The resulting schemas show our algorithms and existing manual algorithms to be equivalence preserving translations. Furthermore, our approach is fully-automatic and has the ability to generate more accurate and intuitively-correct target schemas. In addition, it is more comprehensive than its manual counterparts because it generates (if desired by the user) one target schema or up to three different schemas. Therefore, the CDM, which preserves an enhanced structure of an existing RDB, is translatable into any of the three target database schemas. The algorithms of schema translation are correct in the sense that they have all preserved the original information of the RDBs. Various different types of semantics have been converted from an RDB into the target schemas, e.g., association, aggregation and inheritance. Moreover, the main types of constraints that can be extracted from an RDB, including key constraints, constraints on NULLs and entity and referential integrity constraints, are all translated explicitly into the
create type student_t;
create type faculty_t;
create type campusclub_t;
create or replace type student_ntt as table of ref student_t;
create or replace type faculty_ntt as table of ref faculty_t;
create or replace type campusclub_ntt as table of ref campusclub_t;
create or replace type campusclub_t as object (cid char(10), name char(20), phone char(10), location char(30),
members student_ntt, advisor ref faculty_t) not final;
create or replace type department_t as object (code char(3), name char(20), deptfaculty faculty_ntt, students student_ntt,
depchair ref faculty_t) not final;
create or replace type person_t as object (pid char(10), dob date, firstname char(10), lastname char(10)) not final;
create or replace type faculty_t under person_t (rank char(5), advisorof campusclub_ntt, chairof ref department_t, dept ref department_t) final;
create or replace type student_t under person_t (status char(10), memberof campusclub_ntt, major ref department_t) final;
create table campusclub of campusclub_t
nested table members store as members_ntt;
create table department of department_t
nested table deptfaculty store as deptfaculty_nt
nested table students store as students_ntt;
create table faculty of faculty_t
nested table advisorof store as advisorof_ntt;
create table student of student_t
nested table memberof store as memberof_ntt;
alter table campusclub add constraint campusclub_pk primary key (cid);
alter table department add constraint department_pk primary key (code);

Figure 9: Fragment of ORDB School schema generated by our prototype

equivalent target schemas.

6. Conclusion

This paper contributes a solution to the problem of translating RDB schemas into object-based and XML schemas. The solution is superior to existing work as it generates three different schemas, and take advantages of the variety of powerful features provided in internationally recognized standards. A prototype has been developed to realize the solution, and evaluated using detailed experiments. When our approach is evaluated by comparing the output schemas of its prototype with the ones produced by existing work, it is found that both sets of schemas are comparable. The CDM preserves and enhances the metadata of existing RDBs, and is translatable into any of the three target database schemas. Therefore, the algorithms for translating CDM into the target schema are empirically shown to be correct. In addition, our approach is more comprehensive than its counterparts since it generates up to three different databases.

References


21