

Conceptual Modeling for Customized XML Schemas

Ramez Elmasri, Yu-Chi Wu, Babak Hojabri, Charley Li, and Jack Fu

Department of Computer Science and Engineering
The University of Texas at Arlington
P.O. Box 19015, Arlington, TX 76019, U.S.A.
{elmasri, yuwu, hojabri, qingli, shengfu}@cse.uta.edu

Abstract. XML was initially developed for document management, but it is becoming increasingly used for storing and exchanging all kinds of data on the Internet. In this paper, we introduce a design methodology for XML schemas that is based upon well-understood conceptual modeling methodologies. Because XML is hierarchical (tree-structured), many different XML schemas (or document structures) can be generated from the same conceptual database schema. We describe algorithms for generating customized hierarchical views from EER model, creating XML schemas from hierarchical views, and creating XML instance documents.

1 Introduction

The Extensible Markup Language (XML) [1] was initially developed for document management, but it is becoming increasingly used for storing and exchanging all kinds of data on the Internet. Some have even hypothesized that the XML format will replace databases in the future [2], although this is not very likely. XML has features from the various traditional data models, including the legacy hierarchical data model [3], as well as the relational and object oriented data models [4]. Much work has already been done related to how to store XML using relational [5,6], object-oriented, and object-relational databases [7,8,9], but little work has been done in the area of methodologies for designing XML schemas for specific applications. In this paper, we introduce a design methodology for XML schemas that is based upon well-understood conceptual modeling methodologies. Our approach is system-independent, because it is based on conceptual modeling techniques that are independent of any specific commercial system.

In general, because XML is hierarchical (tree-structured), many different XML schemas (or document structures)¹ can be generated from the same conceptual database schema, and hence from its underlying database. Therefore, there is a need for an XML schema design methodology that helps users in specifying specific XML document structures for their applications. A lot of current data sources use relational databases, whose relational schemas can be reverse engineered [10] into Extended

¹ We will use the terms *XML schema* and *XML document structure* interchangeably.

Entity Relationship (EER) schemas [4,6] (or other similar conceptual data models such as UML class diagrams [11]). By applying our methodology to EER schemas, data sources from relational databases can be converted to a variety of XML document structures, which can then be used by different user applications.

Although standard mappings from relational schemas to XML schemas exist [12,13], we will argue and demonstrate that those mappings rarely produce what the user needs. In fact, default mappings often produce XML schemas that conceptually mirror the relational schemas. In our methodology, users will be able to easily customize XML document structures for their specific applications. In particular, different hierarchical views corresponding to different XML document structures can be created by our methodology. In addition, the tedious part of writing XML Schema documents² will be automated from graphical EER diagrams that are easily understood and manipulated by the users.

The rest of this paper is organized as the following. In section 2 we briefly introduce the architecture of XML Schema Designer, a visual tool for designing XML Schema and instance documents using EER conceptual modeling, and describe the various ways it can be used by presenting several user scenarios. In section 3, we describe algorithms for generating customized hierarchical views from EER model, creating XML Schema documents from hierarchical views, and creating XML instance documents. In the last section of the paper, we discuss the benefits of this methodology and possible future work.

2 XML Schema Designer Architecture and Modules

XML Schema Designer is a visual XML schema designing tool that allows the user to generate XML Schema documents using well-understood conceptual modeling techniques, including EER modeling and UML class diagrams. It also automates the generation of SQL query scripts, the extraction of relational data, and the creation of XML instance documents. Only the EER model is supported in the current version of the XML Schema Designer, but future versions will support UML class diagrams. Figure 1 shows the overall architecture of XML Schema Designer.

XML Schema Designer consists of six components as shown in Figure 1. The following is a brief description of each module.

EER Reverse Engineering. This module extracts an existing relational schema from a relational database using the Data Access module, and creates a corresponding EER schema, which can then be used for designing XML schemas for parts of the relational database. The EER metadata, stored in an internal format, is used as the basis for GUI editing in the EER Designer module.

RDB Designer. Given an EER schema created using EER Designer, this module will create a corresponding RDB schema using the Data Access module. It also facilitates loading the relational data.

² An XML Schema document describes the structure of a set of XML documents, and conforms to the XML Schema recommendation [14].

EER Designer. This module is the visual component (GUI) that a user interacts with to create and edit EER schema diagrams. The user can either retrieve the EER schema using the EER Reverse Engineering module, or they can create the EER schema from scratch. The module initiates creation of the RDB schema through RDB Designer if the EER schema is a new one.

Hierarchy/XML Schema Designer. This module allows a user to create various XML document structures based on the underlying database. The user first chooses the entity types and relationships of interest, which are passed to Hierarchy/XML Schema Designer module as input to create a hierarchy and XML schema. The user first selects a subset of the entity types and relationships from an EER diagram, then chooses a root entity type. This module creates an entity hierarchy, which represents an XML document structure, and also creates the corresponding XML Schema document.

XML Instance Document Generator. Given an XML schema, this module generates a corresponding SQL query script to retrieve data from the relational database, and it also generates the corresponding XML instance document from the query result.

Data Access. This module is the general database access component that provides access to relational database, possible operations including querying RDB metadata, creating RDB schemas, loading RDB data, querying RDB data, and saving EER schemas.

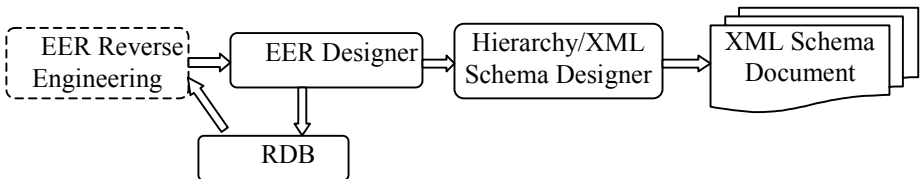
There are basically three types of users that will benefit from using this tool.

User 1: Relational Database Schema Designer



A RDB Schema Designer can use this tool to design the EER diagram graphically. The tool will automatically create the corresponding RDB schema on the RDB server. Future enhancement of this tool will also provide facilities for loading RDB data.

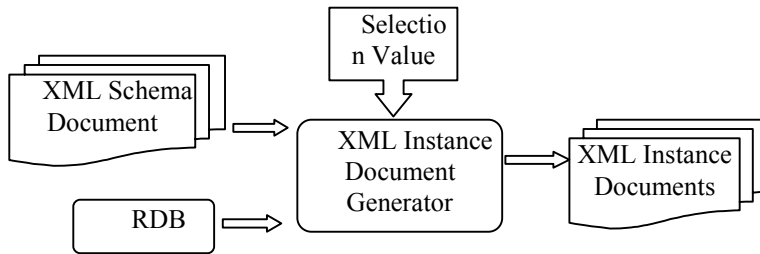
User 2: XML Schema Designer



This type of user can generate any number of XML schemas based on a specific RDB/EER schema. The user will either retrieve the EER schema from an existing RDB using EER Reverse Engineering module, or design it from scratch

using EER Designer module, or retrieve a saved version of some previously created EER schema diagram. The user can then select entities and relationships of interest for a particular XML document structure, and a root entity. The tool will automatically generate an entity hierarchy and convert the hierarchy into an XML schema.

XML Data End User



This type of user can generate XML instance documents from an existing RDB loaded with data. The user first chooses an existing database and a saved XML schema, then enters selection values. The tool will automatically generate an SQL query, execute the query to extract the data from the RDB, and format the extracted data in XML document format.

3 Algorithms for XML Schema Designer

XML Schema [14] is the new standard for specifying the format and constraints of XML documents. One use of XML is for extracting information from relational databases and converting the information into a hierarchical XML document format [12, 8]. This makes it possible to exchange data between databases and XML documents for use on the Internet [15]. XML documents are hierarchically structured. In general, it is possible to create different hierarchical views of a relational database [16]. Since different users may need to see a specific document extracted through a particular hierarchical view, the need to extract different XML Schema documents becomes significant. This section presents three algorithms used in our tool that support the conceptual modeling of XML document structure.

3.1 Algorithms for Generating Conceptual Hierarchical View from EER Model

As a first step, we generate a conceptual hierarchical view for the subset of the entities and relationships that user has already selected for a particular XML document structure. It is likely that the user selection is an EER graph, which contains one or more cycles. In this case, we first eliminate the cycles before we proceed to generate the hierarchical view. First, we present an algorithm to break up the cycles. Then, we present the algorithm for hierarchical view formulation.

Algorithm for Eliminating Cyclic Entities/Relationships in User Selection. The purpose of this algorithm is to eliminate potential graph cycles in the EER diagram subset selected by the user. Prior to this step, the user has already selected a subset of the entities and relationships that are of interest, and the user would have already identified a root entity for the selection. In essence, this algorithm first categorizes all nodes, except root, into different node groups according to their shortest distance from the root (level). It then tries to find the furthest entity node away from the root that participates in a cycle. The algorithm then breaks the cycle by duplicating that entity node (only the entity node, not the subgraph attached to it), so that it still maintains each relationship it originally has with other nodes. At this point, we have a new graph, and we repeat the process until no cycles are detected.

Input: User Selection (G, subgraph of EER), with root R of the graph identified

Output: A tree with the same root, without any cycles

Let G be Graph with n nodes and R be selected root.

Graph2Tree()

```

{
    While (there is a cycle in graph G)
    {
        G = Break_Cycle(G);
        G = new generated graph
    }
    Output the graph G;
}
Graph Break_Cycle(graph G)
{
    1. Use Breath First Search (BFS) to define total ordering on the
    nodes based on distance from root, use Depth First Search (DFS) to
    define edge categories (tree edge or back edge).
    2. Start from the node with deepest level to top level, and do
    cycle elimination:
    Let V denote current node
    For each V {
        For each node that connect to V (NBR-neighbors) from the node with
        deepest level to top level of neighbors. {
            If there is a back edge between V and NBR {
                //Search the cycle path until back edge connected node.
                While cycle_node (the node is moving to search the cycle
                path) is not back edge connected node. {
                    Find the node connected to cycle_node with tree edge in
                    the search path and add cycle_node to cycle_list (the
                    list contain all nodes in the cycle path).
                }
                Construct a list(c_depth_list) and sort it according the
                level of nodes in cycle_list.
                Let Copy_V be the node that is going to duplicate.
                Copy_V = the node that is deepest node in c_depth_list.
                Pre_Copy_V = find the node that is one position ahead the
                Copy_V in term of level in c_depth_list
                G = new_graph (n, Pre_Copy_V, Copy_V )
                Return G;
            }
        }
    }
}
Return G;
}

```

The process to eliminating cycles is the preprocessing procedure for generating a hierarchical view. This algorithm has been implemented and sample results are shown in Figure 6.

Algorithm for Hierarchical View Formulation. This Algorithm is for generating XML schema documents based on users' choices. In the front end, the user-friendly tool will use an EER graphical representation of the database schema to specify the criteria for document generation. The user identifies the root entity type to choose a particular hierarchical view for a specific XML Schema document. Based on the relationships between the root and other entity types, the corresponding hierarchy will be recognized by the system and the XML Schema document will be automatically generated.

Given a portion of interest of a source EER schema that has already been preprocessed to eliminate cycles, and a root entity R, we construct the hierarchical view by invoking the recursive procedure "Generate_Hierarchy (E1,AE1)". There are two input parameters for this procedure.

First parameter is for the current entity that needs to be transferred to hierarchical view.

Second parameter is the ancestor's entity. It is used when the child relationship of current entity is N:1 or 1:1.

In the procedure, we define that

The relationship that is connected to E1 be "L".

The entity type that is related to E1 though L be "E2"

The procedure Generate_Hierarchy (R, R) creates the hierarchical view. It basically merges child entity types that participate in N:1 or 1:1 relationships with their parent entity type. For 1:N or M:N relationships, the relationship attributes are migrated to the child entity type. This keeps the number of entity types in the hierarchical view to a minimum. The pseudocode of the procedure is as follows:

```

Generate_Hierarchy( EntityType E1, AE1, RelationshipName Re)
{
  Let P be each path that has relationship with E1
  Let the path Re be the relationship between E1 and it's parent.
  If E1 is not R(root) { remove the path Re from P }
  For each path P
  Do{
    Let the relationship that is connected to E1 be "L".
    Let the entity type that is related to E1 though L be "E2"
    If L is M:N or 1:N relationship
    {
      Make E2 be the child of E1 in the hierarchy.
      If L is M:N relationship
      {Change L to 1:N relationship (1 to E1 and N to E2) }
      If L has it's own attributes
      { merge L's attributes to E2 }
      // Mark the attribute originally belongs to L with color 1
      Generate_Hierarchy (E2, E2, L)
    }
    else if L is N:1 or 1:1 relationship
    {
      if E1 and AE1 are not the same
      { E1 <= AE1 // Replace E1 With AE1 }
      Merge E2's attributes to E1.
      If L has it's own attribute
      { merge L's attributes to E1 }
      // mark the attributes originally belong to E2 with color 2
      // mark the attributes originally belong to L with color 1
      Generate_Hierarchy (E2, E1, L)
    }
  }
}
}

```

In the procedure, we color the attributes that originally belong to a relationship with color1 and the attributes that originally belong to a merged child entity with color2. As a result, we can distinguish each attribute as to whether it originally belonged to another entity or relationship.

An Example to Illustrate Hierarchical View Formulation. Consider an application that needs XML/Schema documents for student, course and grade information, to be extracted from the university database shown in Figure 2. The user is only interested in the entity sets, COURSE and STUDENT, and the relationships between them via the SECTION entity type. Then, in step 1, these three entity types and their attributes are selected, as well as the relationships S-S and C-S. The EER diagram after this step is shown in Figure 3. In step 2, based on user's choices for XML document structure, it is possible to formulate at least three possible hierarchies:

First, a user can choose COURSE as root, as illustrated in Figure 4a. Since the relationship cardinalities along the path from parent to child in this case are 1:N relationships, there is no need to merge entities. The "Grade" attribute in the S-S relationship is migrated to the STUDENT entity. This is because, by choosing COURSE as root, SECTION becomes a child of COURSE, and STUDENT becomes a child of SECTION in the hierarchy. All STUDENT elements that are children of a specific section are related to that SECTION, and hence can have a specific grade in that SECTION. In this hierarchy, a STUDENT taking more than one SECTION will have several replicas, one under each SECTION, and each will have the specific grade given in that particular section.

Second, the user can choose STUDENT as root (Figure 4b). In this view, in step 3 each section is related to one course, so the relationship between SECTION and COURSE is N:1. We can hence merge COURSE and SECTION entities as shown in Figure 4b. In addition, the "Grade" attribute is migrated to the SECTION entity. In this hierarchy, COURSE/SECTION information is replicated under each separate STUDENT who completed the section.

The third possible way is to choose SECTION as root, as shown in Figure 4c. Similar to the hierarchical view, the COURSE merges into the SECTION entity and the "Grade" attribute is migrated to the STUDENT entity.

Discussion of Hierarchical View Formulation. As we can see, even in this simple example, there can be numerous hierarchical views, each corresponding to a different XML document structure. We illustrated the process of hierarchical formulation. The advantage of this process within our tool is to provide a user-friendly interface and let the user formulate a specific hierarchical view without knowledge of the detailed structure and content of the database or XML schema constructs. The completed hierarchical view is the first stage for designing a customized XML Schema document. We next discuss the algorithm to convert a hierarchical view into an XML Schema document.

3.2 Generating XML Schema and Instance Documents from Conceptual Hierarchical View

In this section, we present the algorithm to build an XML Schema document from a hierarchical view. We will then apply the algorithm to an previous example and generate the corresponding XML Schema document.

Algorithm for Generating XML Schema Document From Conceptual Hierarchical View. The algorithm steps are as follows, and are illustrated using the hierarchy in Figure 4b:

1. Define a root element for the XML Schema document, which will be used to give a name to the entire schema document. The type of this element will be “rootType”, which is a complex type element. In our example, we choose the name StudentTranscriptDoc for the root element.
2. Call the recursive procedure “Generate_XML_Schema”, using the root entity of the hierarchy as its argument. In our example (Figure 4b), STUDENT is the root of the hierarchy, so we pass STUDENT as input to “Generate_XML_Schema”.

We now describe the recursive procedure “Generate_XML_Schema”:

```

Generate_XML_Schema (EntityTypename E1)
{
  Generate a complex type element for E1 and its type, "E1Type"
  Under the E1Type element
  Do{
    For each EER attribute A of E1
    {
      Create a complex type XML element e corresponding to A
      Define a simpleContent3 element within e
      Create four XML attributes for e
        FIELD_NAME, FIELD_TYPE, FIELD_LEN, NULLABLE.
    }
    For each entity E2 that is a child of E1, in the
    hierarchy
    Do { Call Generate_XML_Schema (E2) }
  }
}

```

An Example of Generating XML Schema Document. Now that we have described the algorithm, we use the example given in section 3.1.2 with STUDENT as root. We first define a root element for this XML Schema document, and we choose the name “StudentTranscriptDoc” to represent that this schema contains student transcript documents. Since we use STUDENT as the root of our hierarchical view, STUDENT will be the initial current entity type (E_I) that is input to “Generate_XML_Schema”.

In “Generate_XML_Schema”, we generate a complex element for STUDENT and its type, “STUDENTType”. Under the STUDENTType element, there are three EER attributes, Ssn, Name, and Class of the STUDENT entity type, so we generate three complex elements S_SSN, S_NAME and S_S_CLASS, respectively in XML Schema document for each EER attribute, and each element will have four XML attributes that describe their properties from the database system. Next, we apply the same

³ A SimpleContent element contains either extensions or restrictions on a complexType element with character data, or contains a simpleType element as content and contains no elements [14].

scheme recursively to the SECTION entity type, which is a child of STUDENT in the hierarchical view. Figure 5 illustrates the brief view of the generated XML Schema document.

Formulating SQL Query. In order to generate XML instance documents based on a particular hierarchical view of the database, we need to have a SQL query that extracts data from the database, in addition to some information about the hierarchy. The SQL query text can be stored within the XML Schema document or in a separate text file. This query must be formulated such that it selects all the attributes of the hierarchy ordered by the primary keys of the entities. The tables must be joined together using outer join starting from the top-level entity and in a breadth-first order. The result of the query is used to generate the XML instance document.

Generating XML Instance Documents. Some information about the hierarchy is used for generating the XML instance documents. This includes:

1. Data about the hierarchy including:
 - numEntities: Number of entities in the hierarchy including the "Root" entity
 - numLevels: Number of levels in the hierarchy including the "Root" entity
2. Data about each entity including:
 - number: Entity's number, starting from 0 for the "Root" in a breadth-first order
 - name: Entity's name, used as the XML tag name
 - level: Entity's level, starting from 0 for the "Root" entity
 - parent: Entity's parent number
 - firstAttribute: Column number corresponding to the entity's first attribute
 - numAttributes: Number of attributes for the entity
 - keyIndex: Column number corresponding to the entity's key attribute
 - firstChild: Entity's first child entity
 - numChildren: Number of children for the entity
 - tagNames: Names of the entity's attributes, used as XML tag names

To generate the XML instance documents, method "elementBuilder" is called recursively to process all the entities of the hierarchy in a depth-first order, starting from the "Root" entity. For each entity, there is a set of rows in the query result to be processed. These rows are specified by the entity's *startRow* and *endRow*. Initially, the *startRow* of each entity is the *startRow* of its parent, where the *startRow* for "Root" is 0. The *endRow* for each entity is determined by testing the *keyIndex* value of the entity, except the *endRow* for "Root" that is assigned the last row of the query result. The entity's *endRow* is determined as follows: The last row that still has the same *keyIndex* value as the entity's *startRow* will be the entity's *endRow*; where it must be less than the *endRow* of the entity's parent. Each entity has a vector named *tempVector* that is used to keep track of the entity's *keyIndex* values. Method "getEndRow" is used to find the row of the query result that must be processed as the last row for that instance of the entity. This method tests the entity's *keyIndex* value with its *tempVector* vector to find the entity's *endRow*. If it encounters a NULL value, the method returns a flag meaning that there are no instances of the entity. If it encounters a value that already exists in the entity's *tempVector*, it returns a flag indicating that there are no more instances of the entity. When the process for one instance of the entity is complete, the entity's *startRow* will be set to one after its

endRow, and the same process will be repeated for the next instance of the entity. The algorithm terminates when the "Root" entity is processed completely. The algorithms for "elementBuilder" and "getEndRow" methods are shown in Figures 7 and 8 respectively.

4 Conclusion and Future Work

We have illustrated the process for generating customized XML schemas from EER models. The user selects the entity types and relationships of interest, and identifies the root to form a hierarchical view. According to this hierarchical view, our tool generates the corresponding XML Schema document. We described some of the algorithms used by our tool to automate the breaking of cycles, generate a hierarchy by merging entities and migrating relationship attributes, creating XML schemas, generating SQL queries, and generating XML instance documents.

Our approach for designing XML Schema documents reduces the large amount of work that is needed to access databases and transform their data to XML format. With the power of XML, users will be able to easily customize their XML document structure for their specific application. Consequently, it also establishes the interface between various databases and applications.

Our tool will also store the XML Schema documents for use by applications to create instance documents. We are currently enhancing the tool to provide more flexibility, such as providing the meaning of selecting a root by rephrasing in English, and giving the user options as to how to break the cycles in a graph.

References

- [1] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E.: Extensible Markup Language (XML) 1.0, W3C Recommendation, October 2000.
- [2] Wadler, P.: Et tu, XML?, 27th VLDB Conference, Roma, Italy, 2001.
- [3] Ullman, J., Widom, J.: A First Course in Database Systems, Prentice Hall (1997).
- [4] Elmasri, R., Navathe, S.: Fundamentals of Database Systems, 3rd Edition, Addison-Wesley (2000).
- [5] Fong, J., Pang, F., Bloor, C.: Converting Relational Database into XML Document, IEEE 12th International Workshop on , 2001.
- [6] Kappel, G., Kapsammer, E., Rausch-Schott, S., Retschitzegger W.: X-Ray-Towards Integrating XML and Relational Database Systems, International Conference on Conceptual Modeling (ER), LNCS 1920, Springer-Verlag (2000).
- [7] Ha, S., Kim, K.: Mapping XML Documents to the Object-Relational Form, Proceedings of The 2001 International Symposium on Industrial Electronics, June 2001, IEEE.

- [8] Shanmugasundaram, J., Shekita, E., et al: Efficiently Publishing Relational Data as XML Documents, Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, Sept. 2000.
- [9] Klettke, M., Meyer., H.: XML and Object-Relational Database Systems—Enhancing Structural Mappings Based on Statistics, International Workshop on Web and Databases (WebDB), Dallas, TX, May 2000.
- [10] Vermeer, M., Apers, P.: Reverse Engineering of Relational Database Applications, Proceedings 14th International Conference on OO/ER Modeling (ER '95), LNCS 1021, Springer-Verlag (1995).
- [11] Object Management Group:
<http://www.omg.org/technology/documents/formal/uml.htm>.
- [12] Bourret, R.: Mapping W3C Schemas to Object Schemas to Relational Schemas
<http://www.rpbouret.com/xml/SchemaMap.htm>.
- [13] Florescu, D., Kossmann, D.: Storing and Querying XML Data Using an RDBMS, IEEE Data Eng. Bulletin 22(3), Sep.1999.
- [14] XML Schema Part 1 & 2 W3C Recommendation, May 2001:
<http://www.w3.org/TR/xmlschema-1/>, <http://www.w3.org/TR/xmlschema-2/>.
- [15] Schmidt, A., Kersten, M.L., Windhouwer, M., Waas, F.: Efficient Relational Storage and Retrieval of XML Documents, International Workshop on the Web and Databases (WebDB), Dallas, TX, May 2000.
- [16] Elmasri, R., Larson, J.: A Graphical Query Facility for ER Databases, Proceedings of the 4th International Conference Entity-Relationship Approach, Chicago, Illinois, October, 1985, IEEE.

Figures

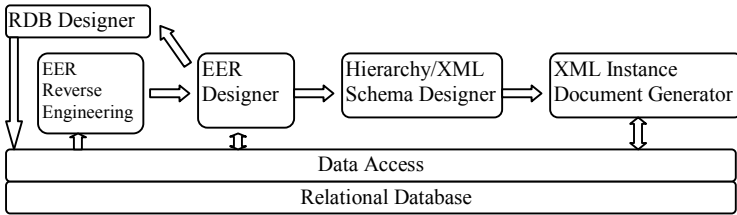


Fig. 1. Architecture Overview

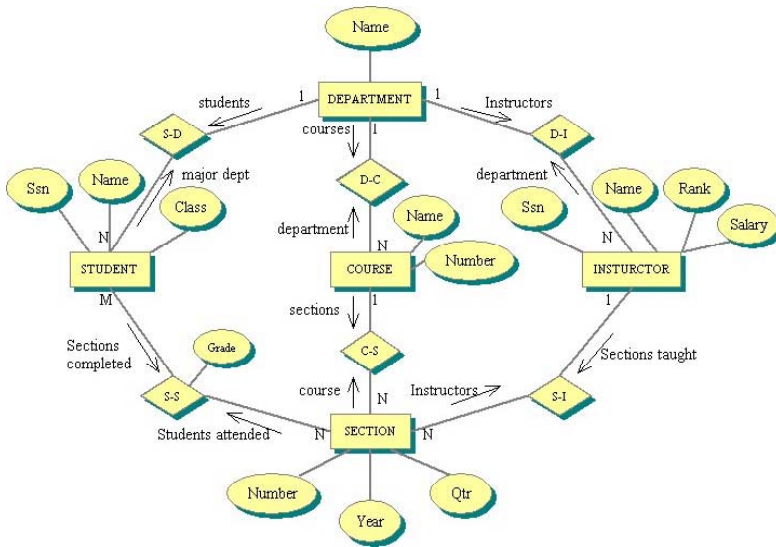


Fig. 2. EER Schema for a University Database

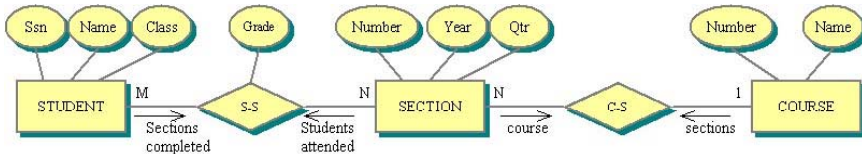


Fig. 3. The EER Diagram after User Selects Relevant Entities/Relationship Attributes

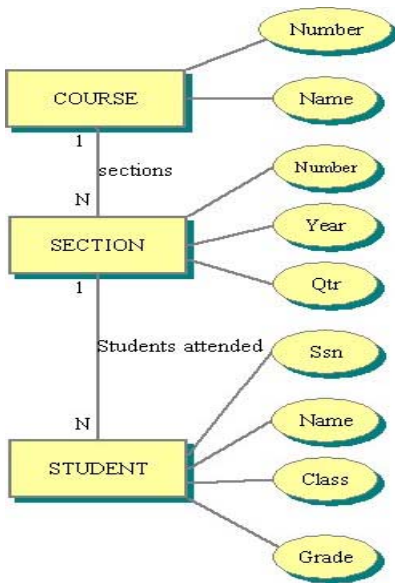


Fig. 4a. Hierarchical View with COURSE as the Root

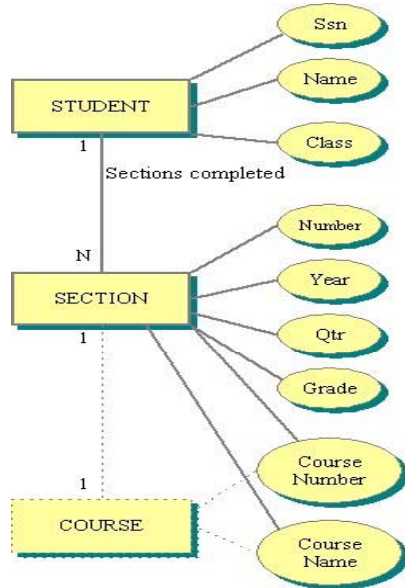


Fig. 4b. Hierarchical View with STUDENT as the Root

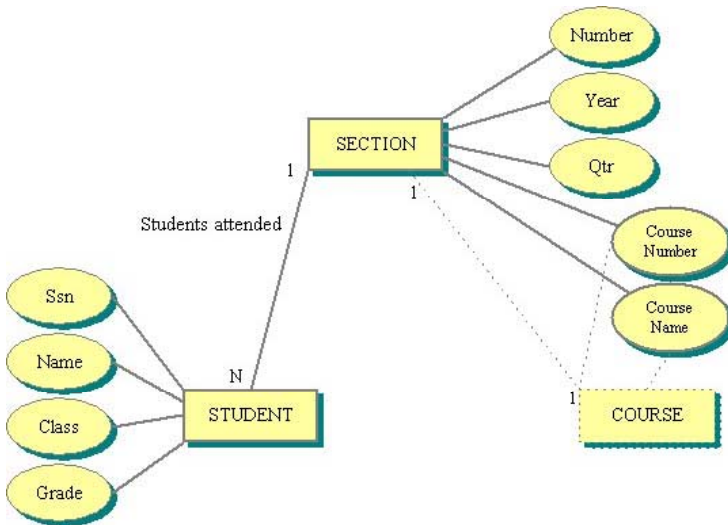


Fig. 4c. Hierarchical View with SECTION as the Root

```

<?xml version="1.0" encoding="utf-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="StudentTranscriptDoc" type="rootType"/>
  <xsd:complexType name="rootType">
    <xsd:sequence>
<xsd:element name="STUDENT" type="STUDENTType" minOccurs="0"
maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="STUDENTType">
      <xsd:sequence>
        <xsd:element name="S_SSN">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:string">
                <xsd:attribute name="FIELD_NAME" type="xsd:string" fixed="S_SSN"/>
                <xsd:attribute name="FIELD_TYPE" type="xsd:string" fixed="CHAR"/>
                <xsd:attribute name="FIELD_LEN" type="xsd:string" fixed="11"/>
                <xsd:attribute name="NULLABLE" type="xsd:string" fixed="false"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        +<xsd:element name="S_NAME">
        +<xsd:element name="S_CLASS">
<xsd:element name="SECTION" type="SECTIONType" minOccurs="0"
maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
    +<xsd:complexType name="SECTIONType">
</xsd:schema>

```

Fig. 5. XML Schema corresponding Hierarchical View that STUDENT as the Root

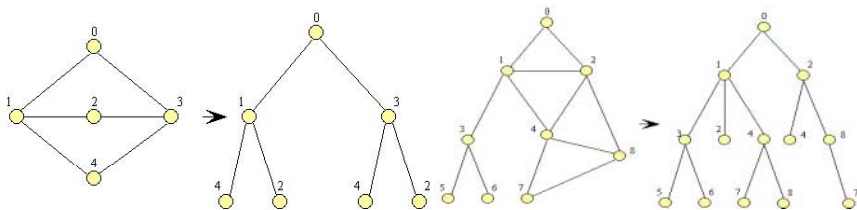


Fig. 6. Example of Algorithms for Eliminating Cycles

```

while (e's startRow <= e's parent's endRow)
  Print the opening tag for e
  Print all the simple elements of e
  if (e has at least one child)
    for (all the e's children as i)
      Empty i's tempVector
      i's startRow = i's parent's startRow
      tempEndRow = getEndRow(i)
      if (tempEndRow == -1 meaning NULL value)
        continue to the next loop iteration
      else
        i's endRow = tempEndRow
        elementBuilder(i)
  Print the closing tag for e
  if (e == 0)
    Exit, since "Root" is processed completely
  else /* search the next rows for new instances of e */
    e's startRow = e's endRow + 1
    tempEndRow = getEndRow(e)
    if (tempEndRow == -1) continue
    else if (tempEndRow == -2) return
    else e's endRow = tempEndRow

```

Fig.7. EementBuilder Algorithm

```

nextRow = e's startRow
if (e's startRow <= numRows - 1)
  if (the e's keyIndex value is NULL) return -1
  else if (the e's keyIndex is in its tempVector) return -2
  Add the e's keyIndex value to its tempVector
nextRow = nextRow + 1
while (nextRow <= endRow of the e's parent)
  if (the e's keyIndex value in nextRow is NULL) return -1
  else if (e's keyIndex value in nextRow is different
    from the e's keyIndex value in the e's startRow)
    break out of the loop
  else nextRow = nextRow + 1
return nextRow - 1
else return numRows - 1

```

Fig.8. getEndRow Algorithm