Rule Based Database Integration in HIPED : Heterogeneous Intelligent Processing in Engineering Design

Shamkant B. Navathe Sameer Mahajan Edward Omiecinski

College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332-0280, USA.
{sham,sameer,edwardo}@cc.gatech.edu

Abstract

In this paper [1] we describe one aspect of our research in the project called HIPED, which addressed the problem of performing design of engineering devices by accessing heterogeneous databases. The front end of the HIPED system consisted of interactive KRITIK, a multimodal reasoning system that combined case based and model based reasoning to solve a design problem. This paper focuses on the backend processing where five types of queries received from the front end are evaluated by mapping them appropriately using the "facts" about the schemas of the underlying databases and "rules" that establish the correspondance among the data in these databases in terms of relationships such as equivalence, overlap and set containment. The uniqueness of our approach stems from the fact that the mapping process is very forgiving in that the query received from the front end is evaluated with respect to a large number of possibilities. These possibilities are encoded in the form of rules that consider various ways in which the tokens in the given query may match relation names, attrribute names, or values in the underlying tables. The approach has been implemented using CORAL deductive database system as the rule processing engine.

1 Introduction

Heterogeneity of databases is becoming a necessary factor to contend with in the design of new applications because of the proliferation of database management systems that used diverse data models over the last three decades. Among widely implemented data models we have the hierarchical, network, relational and object oriented data models. A large body of work exists that deals with the mapping of these models among one another (e.g. see the mapping of models using the entity relationship model as an intermediate model in [1] [3]. While vendors are also providing middleware solutions to draw data from these legacy systems, the semantic problems of resolving, naming, scale, structure etc. that were pointed out several years ago [5] [6] still remain. The purpose of the present research was to develop a technique to dealing with the semantic differences in data by taking a flexible rule based approach. Another goal of the project was to tie a set of heterogeneous databases to an "intelligent front end application" which would make requests for data without any knowledge of the schemas of the target databases. To limit the degree of difficulty we assume that we are dealing with data in relational databases only. This assumption is reasonable in the sense that of the data is coming from a hierarchical or a network DBMS, we can first convert the schema to a relational one before treating it for purposes of integration.

The database integration problem we discuss here is couched in the context of engineering design which, like any other design application, relies on extracting data from existing databases containing material data, components, existing designs etc. The exact context and the application scenario will be explained in the next section.

We assume that relevant data for the design application is stored in relations (tables) whose schemas are available at "design time" to construct a rule-base. It is conceivable that to support large scale engineering designs, data from a variety of databases, i.e., from multiple schemas would be required. To facilitate integration of data among these databases we assume that the "correspondances", i.e., the similarities and differences among the (meaning of) attributes is encoded in the form of rules. Furthermore, for our application context, the front end of HIPED issues certain queries looking for relevant design information. We show in this paper how a query may have several interpretations, each one of which is encoded in the form of rules again.

Because of these two kinds of rules involved in the integration approach we have termed our approach a rule based approach to database integration. The present approach is an improvement over previous approaches where we handled integration by using the correspondance information to derive the process [2] [6] [7] [8].

2 Application Context

In this section we will provide the overall architecture of the HIPED system and point out the need for heterogeneous database processing which will be de-

[1] To appear in the Proceedings of International Symposium on Cooperative Database Systems for Advanced Applications, Heian Shrine, Kyoto, Japan, World Scientific Press, 1996.

scribed and illustrated in the next two sections.

2.1 Overall Architecture of HIPED

Our main objective in the HIPED project is to build and demonstrate an intelligent interface to a set of (possibly autonomous) information sources including structured databases, knowledge bases, and unstructured data. The approach we have selected involves the development of a mediator which utilizes meta-knowledge of the underlying information stores to aid a user in browsing data or to enable an application front-end to retrieve specific relevant information for problem solving.

The overall architecture of HIPED is described in Figure 1. We look at only the "Database Bachend" in this paper. The data is organized at two levels namely, (1) the metadata repository : consisting of information about various databases and tables in them and (2) the actual data which is distributed in various heterogeneous databases. This organization reduces the data to be dealt with at the first level to get to the appropriate database(s) and table(s). It also allows heterogeneity in the various databases involved. The Querying Interface is as described in section 3.1. The "data" together with its "wrapper" forms a database system. "Wrapper" simply defines the access methods to the data for reading purposes. A wrapper can be designed for each target database management system. A user query would be translated into the corresponding query, as understood by the corresponding "wrapper", for each of the relevant tables. This query would then be routed to the corresponding database, that contains this table. The metadata repository is consulted in determining these relevant tables and finding the corresponding database. The user would get the result, obtained after running the query against the table through the concerned "Output Data" channel(s).

2.2 Interactive KRITIK Front End

We developed the HIPED architecture by assuming a frontend system called Interactive Kritik [4]. This system is a multimode reasoning system which works like a design assistant for the design of devices such as acid coolers, electrical devices. In its current form the system uses "hard-wired" knowledge in the form of LISP data structures. The goal was to extend the capability of interactive Kritik to make it scalable to real-life design problems by incorporating databases of relevant design data as the back end. We therefore abstracted different forms of generic query types which would be used as requests to the back end. By coupling an intelligent front end application to a set of heterogeneous databases, we can thus extend the scope of problem solving by a large measure. For engineering device design, the above front end generates a number of requests for data from the underlying design databases such as design prototypes, properties of devices and components, material data, design specifications and tolerances etc. For illustrative purposes we have chosen five generic types of queries that are most commonly presented by the front end. They will be explained in detail in the following section.

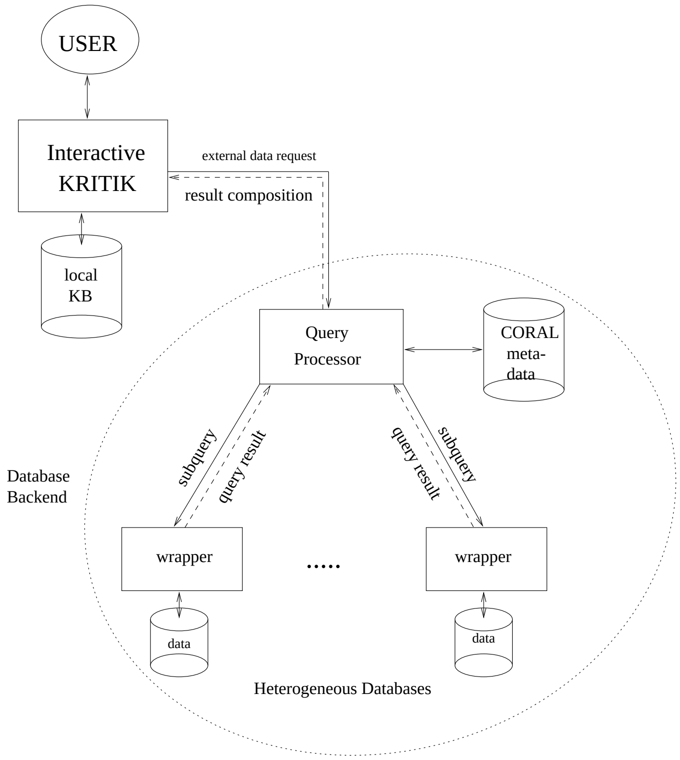

Figure 1: The High Level View of HIPED

3 Rule Based Approach to Database Integration

As explained earlier the main contribution of this research is the use of the two types of rules to accomplish access to the underlying heterogeneous information sources. The first set of rules deals with establishing various types of relationships among relation names and among attribute names across databases. The second set deals with the interpretation of queries from the front end so that various possible mappings to the interface of underlying target databases may be considered. We will explain both these types of rules when we discuss the generic queries and their mappings.

3.1 Five generic types of queries

The user is assumed to use this system as an Engineering Database for device design. Let us limit the application domain for illustrative purposes. We assume that during the design process, he would typically like to find components that satisfy his requirements (e.g. batteries with voltage rating higher than 10V and cheaper than \$10). Keeping this user's perspective in mind, the Engineering data is thought to be made up of various "Prototypes". Each Prototype has various "Properties". Each Property takes up some "Value" for every Prototype. We can compare the Values of various properties using the relations : $==, <, >, <=, >=, <>$ etc. The queries can be classified into the following five generic types,

1. (Prototype <proto_name>) : here the user is looking for all the prototypes identified by "proto_name". It is implicit that the user wants to see the various values for various properties (attributes) of these prototypes.

2. (Property <prop_name>) : the user is interested in all the prototypes having the specific Property identified by "prop_name". It is implicit that the user wants to see the values taken by this property for the various prototypes, that would be listed.

3. (Prototype <proto_name>)
(Property <prop_name>) : the user wants to see all the prototypes identified by "proto_name" and having property identified by "prop_name". It is implicit that the user also wants to see the corresponding value that the property takes for the particular prototype.

4. (Prototype <proto_name>)
(Property <prop_name>)
(Value<value>) (Rel-op <op>) : the user is interested in prototypes identified by "proto_name" having a property identified by "prop_name". In addition to this he wants only those prototypes for which the property takes a value which is related to the given "value" or a constant in the query by the operator "op" (i.e. it is equal to "value" or greater than "value" etc.)

5. (Property <prop_name>) (Value <value>)
(Rel-op <op>) : the user is interested in all the prototypes for which the property identified by "prop_name" takes a value which is related to the given "value" by the operator "op.

Data is distributed among various databases and various tables in each of those databases. The only assumption that we make about any database system is that it has an SQL access method. It is a reasonable assumption and is made to contain the complexity of the problem.

The system needs to find out which databases and which tables in these databases have the relevant data to answer a particular query. It then translates the query into a corresponding SQL query for every table. This SQL query is run against that table to get an answer. As we made an assumption of a uniform SQL interface to all the databases, we can simply translate a request for data into a set of SQL queries in each of these cases.

3.2 Rules for Interpretation of Queries

For better understanding of the following discussion, let us take up an example query. Let the four components of the query be,
(Prototype Battery) (Property Voltage)
(Value 10) (Relation ==).
As there can be various tables with different schema, we need to run this query with only those tables that might give meaningful results for the query. We can easily observe that any of "Prototype", "Battery", "Property" and "Voltage" can be a table or a column of a table. The "Battery" and "Voltage" can also be values in the columns (e.g. those labeled as "Prototype" and "Property" respectively). Of course there are a lot of dependencies amongst these components - e.g. if "Prototype" is a table then "Battery" has to be a column of this table. On the other hand if there is a table called the "Battery", then we are looking for values in the column "voltage" or "volts" - so that the query would generate meaningful results with the table. Now we take up an example query for each of the five types listed above. For every query we list the possible interpretations according to our scheme.

1. (Prototype Battery). The user typically means that he wants all the batteries with their properties and their corresponding values. Hence we will have to run this query against all the tables which,

   - are equivalent to "Prototype Table" and have a column equivalent to "Battery" or
   - are equivalent to "Battery Table"
   - have a column equivalent to "Prototype" (and only the tuples with Prototype as "Battery" would be considered).

   if and only if these tables have columns equivalent to "Property" and "Value" each.

2. (Property Voltage). The user is interested in listing all the Prototypes having "Voltage" as their one of the Properties. The Values of these Properties would also be significant from his standpoint. Hence we consider all the tables which,

   - are equivalent to "Property Table" and have a column equivalent to "Voltage" or
   - are equivalent to "Voltage Table"
   - have a column equivalent to "Property" (and only the tuples with Property as "Voltage" would be considered).

   if and only if they have "Prototype" equivalent column.

3. (Prototype Battery) (Property Voltage). The user wants all the batteries with special interest in their voltages. Hence we will run the query against all the tables which,

   - are equivalent to "Prototype Table" and have "Battery", "Property" and "Value" equivalent columns and we would be interested only in the tuples having an entry of "Voltage" in the "Property" equivalent column or
   - are equivalent to "Prototype Table" and have "Battery", "Voltage" equivalent columns or
   - are equivalent to "Battery Table" and have "Property" and "Value" equivalent columns. We would be interested only in those tuples having Property "Voltage" or

- are equivalent to "Battery Table" and have a column equivalent to "Voltage" or
- are equivalent to "Property Table" and have columns equivalent to "Voltage", "Prototype" and "Value". We would be interested in tuples with Prototype as "Battery".
- are equivalent to "Property Table" and have "Voltage" and "Battery" equivalent columns.
- are equivalent to "Voltage Tables" and have "Prototype" and "Value" equivalent columns. We would look for only tuples with Prototype as "Battery".
- are equivalent to "Voltage Table" with "Battery" and "Value" equivalent columns.
- have "Prototype" and "Property" equivalent columns as far as they have "Value" equivalent column. Only the tuples with Prototype as "Battery" and Property as "Voltage" would be considered.

4. (Prototype Battery) (Property Voltage) (Value 10) (Relation ==). Here the interest is indicated in all the batteries having Voltage as "10". The query can be run with all the tables as indicated as above with an added constraint that only those tuples which have an entry of "10" in the "Voltage" or "Value" column - whichever is applicable - (Note the table can have only one of these columns at a time) will be considered.

5. (Property Voltage) (Value 10) (Relation ==). All the Prototypes having voltage of "10" are being considered. Thus all the tables that,

   - are equivalent to "Property Table" and have a column equivalent to "Voltage"
   - are equivalent to "Voltage Table" and have a column equivalent to "Value"
   - have "Property" and "Value" equivalent columns along with "Prototype" column. (only tuples with Property "Voltage" and Value "10" would be taken into consideration).

   would be considered if and only if they have a column equivalent to "Prototype". All the tuples with "Voltage" or "Value" being 10 would be taken into account.

3.3 Rules to establish Data Correspondance

We need to relate various attributes and tables, within and across databases. The relationship could be of equivalence, subsumption, overlap, disjointness or containment. The relationship between attributes needs to be supplied by the schema developer. e.g. Attributes called "volt" and "voltage" in different tables are actually equivalent. The relationship between tables can either be supplied or can be deduced by the relationships of their individual attributes. A simple deduction rule can be that two tables are equivalent if all their attributes are equivalent.

4 Use of CORAL for rule representation and query processing

The metadata is stored in the form of CORAL [10] [11] facts and rules. CORAL is a deductive database system which stores data as facts and rules, and allows for that data to be queried. By using CORAL the mediator can decide which database(s) and table(s) are useful in answering any given query. In particular, CORAL is used in deriving relationships like equivalence; between attributes, tables and databases. Any creation, deletion or modification of a table results in a change in the metadata repository. This dynamic behavior can be easily captured by CORAL. In essence, CORAL provides us with the facility for database integration through the facts and rules specified about tables and databases. However, this integration can be considered implicit rather than explicit since no global conceptual schema is explicitly formed. Also the C++ interface provided by CORAL makes writing general purpose programs easy.

We explain the implementation with the help of an example. One more sample system for a single database environment is given in Table 5. Some sample input queries and the corresponding output SQL queries are shown in Tables 6 and 7 respectively.

4.1 A Simple Example

Consider the query,

```
(Prototype Battery) (Property Voltage).
```

Let us assume that there are two databases - db1 and db2. Let db1 have tables : Table 1 and Table 2. and

CompNo	Prototype	Property	Value
B101	Battery	Voltage	10
M101	Motor	Voltage	10
B110	Battery	Voltage	100
B111	Battery	Current	100
⋮			

Table 1: "Components" Table in db1

let db2 have Table 3 and Table 4.

We observe that according to the discussion in section 3.2 only the tables in db1 would produce meaningful results with the query under consideration.

BatteryNo	Voltage
B101	15
B102	30
⋮	

Table 2: "Battery" Table in db1

BatteryNo	Current
B101	15
B102	30
⋮	

Table 3: "Battery" Table in db2

BatteryNo	Supplier No
B101	4567
B102	4568
⋮	

Table 4: "Supplier" Table in db2

4.2 Schema Representation

It is stored as CORAL facts and rules. The advantage of such a storage is that we can utilize the strong deductive power of CORAL (e.g. deducing equivalence of attributes, equivalence of tables etc.). The various components of the repository are described below.

- First we list all the tables in all the databases as CORAL facts :

```
% For the first database, db1.
belongsTo(components,db1).
belongsTo(battery,db1).

% For the second database, db2.
belongsTo(battery,db2).
belongsTo(supplier,db2).
```

- Then we list attributes of individual tables as CORAL facts. The first argument of these predicates is the database name. It is so because the same table may have different attributes in different databases. e.g. the "battery" table in the two databases "db1" and "db2" as shown below.

```
% for db1
hasAttribs(db1,components,
   [compName,prototype,property,value]).
hasAttribs(db1,battery,[bName,voltage]).

% for db2
hasAttribs(db2,battery,[bName,current]).
hasAttribs(db2,supplier,[bName,sName]).
```

- We also need facts to list what attributes are equivalent. The equivalence of tables can be either given by facts or can be deduced by the rules (e.g. two tables with equivalent attributes are equivalent). But we do not need them in this particular example.

- To find whether a table has a particular attribute in a given database we define a CORAL rule as,

```
module isAttrib.
export isAttrib(bff).
isAttrib (Db,Table,Attri) :-
        hasAttribs(Db,Table,Attribs),
        iselem (Attri,Attribs).
end_module.

% Module ''iselem'' is defined for the
% sake of completeness.
module iselem.
export iselem(bb).
@pipelining+.    % Solve in a
                 % top-down fashion

iselem(X, [X|_]).
iselem(X, [_|Z]) :- iselem(X,Z).
end_module.
```

4.3 Sample Query Mapping Algorithm

The mapping of input requests into SQL queries is done according to the scheme suggested in section 3.2. We use the C++ interface of CORAL for this matter. In fact, an imperative interface (e.g. in C) would have been enough for the purpose. We check for the various conditions given in the scheme and generate the appropriate SQL queries for the existing tables. We run through the algorithm for the example query under consideration,

```
begin
For every ''table'' equivalent to
''prototype table''
  for every attribute equivalent
  to ''battery'', say attrib1
    for every attribute equivalent
    to ''property'', say attrib2
      if ''table'' has ''attrib1''
      as well as ''attrib2''
        for every attribute
        equivalent to ''value'',
        say attrib3
          if ''table'' has attrib3
            select the corresponding
            database and fire SQL query,
            SELECT * FROM table
            WHERE attrib2 == voltage or
                  some equivalent value.
            goto next table
        for every attribute equivalent
        to ''voltage'', say attrib4
          if ''table'' has attrib4
            select the corresponding
            database and fire SQL query,
            SELECT * FROM table
```

```
For every ``table'' equivalent to
``battery table''
  for every attribute equivalent
  to ``voltage'', say attrib1
    if ``table'' has attrib1
        select the corresponding
        database and fire SQL query,
        SELECT * FROM table
        goto next table
  for every attribute equivalent
  to ``property'', say attrib2
    for every attribute equivalent
    to ``value'', say attrib3
      if ``table'' has attrib2
      and attrib3
        select the corresponding
        database and fire SQL query,
        SELECT * FROM table
        WHERE attrib2 == voltage or
              some equivalent value.

For every ``table'' equivalent
to ``property table''
  for every attribute equivalent
  to ``voltage'', say attrib1
    for every attribute equivalent
    to ``prototype'', say attrib2
      if ``table'' has ``attrib1''
      as well as ``attrib2''
        for every attribute
        equivalent to ``value'',
        say attrib3
          if ``table'' has attrib3
            select the corresponding
            database and fire SQL query,
            SELECT * FROM table
            WHERE attrib2 == battery or
                  some equivalent value.
            goto next table
        for every attribute equivalent
        to ``battery'', say attrib4
          if ``table'' has attrib4
            select the corresponding
            database and fire SQL query,
            SELECT * FROM table

For every table equivalent to
``voltage table''
  for every attribute equivalent
  to ``battery'', say attrib1
    if ``table'' has attrib1
      select the corresponding
      database and fire SQL query,
      SELECT * FROM table
      goto next table
  for every attribute equivalent
  to ``prototype'', say attrib2
    for every attribute equivalent
    to ``value'', say attrib3
      if ``table'' has attrib2
      and attrib3
        select the corresponding
        database and fire SQL query,
        SELECT * FROM table
        WHERE attrib2 == battery or
              some equivalent value.

For every table having columns
equivalent to each of
prototype, property and value
  select the corresponding
  database and fire SQL query
  SELECT * FROM table
  WHERE prototype equivalent column
          == battery equivalent value
        AND
        property equivalent column
          == voltage equivalent value
end
```

4.4 The Result

Let us say that the wrapper of db1 can handle SQL queries. In that case we first select that database and then simply run a query,

```
SELECT *
FROM components
WHERE prototype == ``battery''
        AND property == ``voltage''
```

against the first ("components") table in the database. We take similar actions for the other table in (possibly various) databases. The other query in this case would be,

```
SELECT *
FROM battery
```

again with the same database namely, db1. The result is presented to the user as displayed by the corresponding "wrapper".

5 Conclusions and Future Work

In this paper we illustrated the implementation of a rule-based database integration scheme by considering two types of rules : (1) Rules to establish the "correspondence" among underlying component databases and (2) Rules to interpret data requests in an "open-ended" fashion where no knowledge of the component database schemas is expected from the application front end. We also described an interface to heterogeneous databases in which a user may directly access the back end data by making use of the rules of data correspondance and an SQL-like syntax for the queries.

The system makes an assumption that all the databases involved provide an SQL interface. This condition can be relaxed. In this case we need to generate different queries, as understood by each of the databases involved. This work was predicated on the assumption that the data relevant to our application was stored in relational tables. An extension of the present work involves relaxing this assumption and illustrating the utility of the approach by actually providing wrappers for hierarchical and network databases and sequential files. That would establish

```
% CORAL facts
isTable(battery).
hasAttribs(battery,
   [bname,voltage,current,life]).

isTable(compTable).
hasAttribs(compTable,
   [no,prototype,property,value]).

isTable(dummy).
hasAttrib(dummy,[prototype,property]).

isTable(prototype).
hasAttribs(prototype,
   [motor,property,value]).

isTable(motor).
hasAttribs(motor,[property,value]).

isTable(property).
hasAttribs(property,
             [rps,prototype,value]).

isTable(rps).
hasAttribs(rps,[prototype,value]).

isTable(voltage).
hasAttribs(voltage,[battery,value]).

% CORAL rules

module isAttrib.
export isAttrib(ff).
isAttrib (X,Y) :- hasAttribs(X,Z),
                  iselem (Y,Z).
end_module.
```

Table 5: A Single Database System

```
prototype battery property voltage
prototype battery property current
prototype motor property rps
prototype sheet property size
```

Table 6: Sample Input Queries

```
++++++++ for the first data request ++++++
SELECT * FROM battery;
SELECT * FROM voltage;
SELECT * FROM compTable
WHERE prototype == battery
        AND property == voltage;
++++++++ for the second data request +++++
SELECT * FROM battery;
SELECT * FROM compTable
WHERE prototype == battery
        AND property == current;
++++++++ for the third data request ++++++
SELECT * FROM prototype
WHERE property == rps;
SELECT * FROM motor
WHERE property == rps;
SELECT * FROM property
WHERE prototype == motor;
SELECT * FROM rps
WHERE prototype == motor;
SELECT * FROM compTable
WHERE prototype == motor
        AND property == rps;
+++++++ for the fourth data request +++++
SELECT * FROM compTable
WHERE prototype == sheet
        AND property == size;
```

Table 7: The corresponding SQL queries

the practical utility of the approach in a significant way. The next step would be to work on a query optimization by introducing a stage after the query interpretation phase to evaluate possible orderings of sub queries and cross subquery reduction of redundant processing.

From the engineering design standpoint, the problem horizon can be extended to include additional types of design problems. The current implementation can be initially enhanced by considering additional types of design queries.

Currently only the individual tables are checked to see whether they provide satisfactory data to answer a particular query. But it is possible that two or more tables taken separately do not have enough information to answer a query. At the same time, when taken together (e.g. their join), they provide data to answer the query. Consider that there are two tables - which might be in the same database or in different databases - one with columns "Component Number" and "Prototype". The other with columns "Component Number" and "Voltage". Then neither of them provides enough information for the query,

(Prototype Battery) (Property Voltage)

But their equijoin with the additional condition of "Prototype == Battery" for the tuples is of interest to us. The extended solution can exhaustively take care of all such cases.

In essence, the overall rule based approach appears promising in the context of Navathe's long standing investigation of the database integration problem [5] [6] [7] [8] [9].

Acknowledgement

We would like to thank Ashok Goel and William Murdock for their work on Interactive KRITIK. The work crystallized the HIPED front end to our system. We are also grateful to Jeff Donahoo, Ashok Savasere and Byong-soo Jeong for initializing the implementation work. The idea of using CORAL as a deductive database system was generated by Ed Omiecinski. The implementation was completed by Siddharth Bajaj. Support from ARPA grant no. F33615-93-1-1338 is greatly appreciated. Prof. Navathe's work is also partially supported by Center of Excellence in Information Science, Clark Atlanta University, Contract No. OSP-93-09-400-002.

References

[1] C. Batini, S. Ceri, and S.B. Navathe, **Conceptual Database Design: An Entity Relationship Approach**, Benjamin Cummings, August 1991, 470 pp.

[2] C. Batini, M. Lenzerini and S.B. Navathe, A Comparative Analysis of Methodologies for Database Schema Integration, *ACM Computing Surveys*, 18, 4, December 1986, pp. 323-364.

[3] R. Elmasri, S Navathe, **Fundamentals of Database Systems**, Addision Wesley Computer Science, 2nd Edition, 1994.

[4] Ashok Goel, Andres Gomez, Nathalie Grue, William Murdock, Margaret Recker, and T. Govindaraj. Design Explanations in Interactive Design Environments. In *Proc. Fourth International Conference on AI in Design*, Palo Alto, June 1996.

[5] J. Larson, S. B. Navathe, and R. Elmasri A Theory of Attribute Equivalence in Databases with Application to Schema Integration, *IEEE Transactions on Software Engineering*, Vol. 15, No. 4, April 1989.

[6] S.B. Navathe, R. Elmasri and J.A. Larson, Integrating User Views in Database Design, *IEEE Computer*, Vol. 19, No. 1, January 1986, pp. 50-62.

[7] S. B. Navathe and A. Savasere, "A Practical Schema Integration Facility using an Object Oriented Approach," *Multidatabase Systems (A. Elmagarmid and O.Bukhres, Eds.)*, Prentice Hall, 1996.

[8] S. Prabhakar, J. Srivastava, S.B. Navathe, et al., Federated Autonomous Databases: Project Overview, *Proceedings of the International Conference on Interoperability in Multidatabase Systems (IMS'93)*, Vienna, Austria, April 19-20, 1993.

[9] A. Sheth, S.K. Gala, S.B. Navathe, *On Automatic Reasoning for Schema Integration, Int. Journal of Intelligent Co-operative Information Systems*, Vol. 2, No.1, March 1993.

[10] R. Ramakrishnan, D. Srivastava and S. Sudarshan, CORAL: Control, Relations and Logic, *Proceedings of the International Conference on Very Large Databases*, 1992.

[11] R. Ramakrishnan, D. Srivastava, S. Sudarshan and P. Seshadri, Implementation of the CORAL deductive database system, *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.