Automatic Migration of Files into Relational Databases

Uwe Hohenstein & Andreas Ebert
Siemens AG, Corporate Technology, ZT SE 2
D-81730 München (GERMANY)
<firstname>.<lastname>@mchp.siemens.de

ABSTRACT

In order to provide database-like features for files, particularly for searching in Web data, one solution is to migrate file data into a relational database. Having stored the data, the capabilities of SQL can be used for querying, provided, the data has been given some structure. To this end, an adapter must be implemented that converts data from files into the database.

This paper proposes a specification-based automation for this procedure: Given some descriptive specification of file contents, those file adapters are generated. An adequate specification language provides powerful concepts to describe the contents of files. In contrast to similar work, directory structures are taken into account because they often contain useful semantics.

1. INTRODUCTION

In spite of the variety of different database technologies, there is still a huge amount of data kept in ordinary files. For example, a few years ago, engineering applications were forced to take files as the performance of relational DBSs was quite bad for handling complex structures with traversal operations. It is still today the case that CAD tools keep their data in files. Even though object-oriented databases may be a well-suited platform for these kinds of applications, the files still exist due to the legacy problem [13]. Other examples for maintaining persistent data in files are electronic documents and gene databases. Files also get new importance in the context of the world-wide web: WWW pages are stored in files, and generated pages can also be seen as files.

Being spoilt by database systems, DBS-like requirements are arising for files. In fact, files are different to databases: They do not possess an explicit schema, and they provide only a simple interface for reading and storing data. Particularly, the field of semi-structured data [4] investigates solutions for this conflict. There is a lot of work dedicated to querying the web [6,14,18,23]. For example, [16] provides a SQL-like querying for the web by means of Unix services for analyzing and filtering semi-structured data. WebSQL [20] helps exploiting the structure and topology of the document. [1] proposes an extension of OQL [8] as a query language for the OEM model, and [3] describes how to evaluate queries by means of query rewriting and view materialization.

WIDM 99 Kansas City Mo USA

Most work in this context is relying on light-weight data models [7,21] to infer and model the structure of file data for an adequate querying. The approaches assume that data is already arranged in such a model, and that file data is wrapped by a layer that enables access. Transferring data manually, e.g., by implementing parsers [2], is a complex task that requires a lot of knowledge in compiler technology, even if tools like Yacc are used. Moreover, a manual implementation of adapters is often impractical because the format of sources changes frequently [5], especially in the WWW.

Only few approaches yield an effective support to this step. [5] suggests a semi-automatic way to generate adapters for internet data sources. The adapters then allow users to query web documents in a database like fashion. The approach takes keywords as tokens of interest, and derives the nested document structure from the source. The paper presents regular (Lex) expressions for usual representations such as headings and emphasizing tags. These are heuristics, whereupon a Yacc-program is generated. However, the overall approach is tailored very much to WWW pages. Similarly, [17] proposes wrapper generation, too. But the paper makes more assumptions about the data to be looked for. A powerful approach has been developed in the TSIMMIS project [22,9]. On the basis of specified templates that model file data in OEM, wrappers are generated.

This paper describes how file data can be migrated into a relational database in a comfortable manner. Database-like features are then for free since using SQL is powerful. This avoids the problem of limited query capabilities as discussed by [19].

In contrast to other proposals [5,9], our approach is very general and suited for any kind of files. The concepts are particularly adequate for WWW sources, too. Moreover, we are able to incorporate the directory structure. This is important since files and directories carry semantics being also useful for queries. Furthermore, our approach takes into account unparsing the data, i.e., writing data back into files after modifications. These important points are mostly neglected by others.

The remainder of this paper is structured as follows. Section 2 is concerned with the generative approach. We are presenting the basics of a language ODL_{File} for specifying file contents. Any such specification is an intuitive input to a generator that produces relational tables and a file adapter. This adapter converts files into tables in a structured manner, and stores data back into files. This releases one from the tremendous task of implementing Yacc programs as it is necessary in [2]: We *generate* those programs. Several files in any directories can be handled.

Section 3 presents several relevant examples that demonstrate the expressiveness of the specification language. Particularly, some aspects related to handling WWW sources are discussed.

In Section 4, we outline some future work we are planning to do.

2. GENERATIVE APPROACH

2.1 Principle

The generative principle for the handling of files relies on a generator producing a file adapter. The generator requires input by means of a specification in a language ODL_{File}. A specification defines a "schema" for file contents in a certain sense. The syntax of ODL_{File} is a mixture of the ODMG ODL [8] and an enhancement of Yacc. ODL is used to describe the schema of the data file in an object-oriented way. File contents are modeled independently of the physical file structure. The Yacc-part then defines a grammar. The grammar precisely describes the contents of files. Using a Yacc-style eases the later generation of a parser by means of this compiler-compiler. Parsing a file, the collected information has to be organized according to the ODL schema. Hence a third part of ODL_{File} relates the grammar to the schema.

The generator uses the grammar of the specification and produces a Yacc program that is able to parse data files. Using the ODL schema and the assignments, the Yacc-program is enriched with semantic actions that build objects according to the ODL schema. These are the objects to be stored in a relational database. Tables are installed according to some standard mechanism that map the ODL_{File} schema onto tables. Afterwards complex ODL objects are transferred into the relational database.

The unparser within the adapter is responsible for writing modified data back to files. As there is no tool support, the generator for unparsers must be implemented from scratch.

The data collected by the generator is stored in a meta-database. The meta-data consists of any information found in an ODL_{File} specification, i.e., the ODL schema and the structure of the grammar. Producing the Yacc program by means of the meta-database is easier as it does not need to be done during parsing an ODL_{File} specification. Moreover, the metadata is used for producing the unparser and the relational tables.

2.2 Example

Figure 1 describes a simple migration scenario with a data file containing departments and employees: Lines starting with 100 denote departments, those starting with 200 mark employees. The sequence of the records within the file is relevant: All employees following a department record work for that department. Furthermore, the first employee record is assumed to be the head of the department.

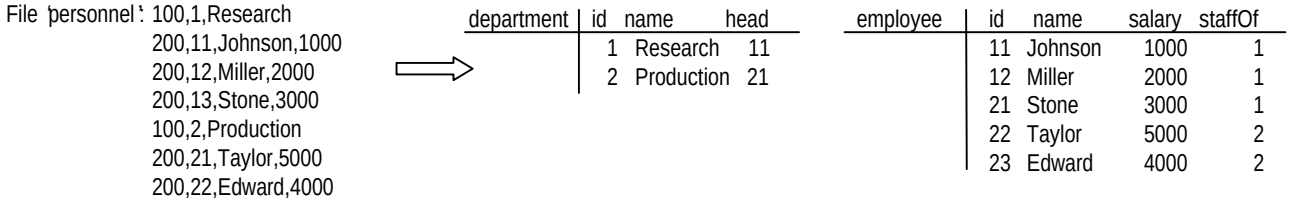
The first part of an ODL_{File} specification defines an *ODL-schema* for the data. The syntax is based on ODMG ODL [8]. In Figure 1, the data is modeled as object types Department and Employee.

ODL_{File} supports only a subset of ODMG ODL; subtypes and explicit relationships between object types are omitted. Although useful from the point of modeling file data, we cannot really benefit from subtypes and relationships, because we have to map them afterwards onto flat tables. Moreover, the representations of subtypes and relationships in files are much more manifold than those discussed in the field of semantic enrichment of relational databases [10,23]. Such an enrichment of files should better be done by a successive semantic enrichment step, e.g., using our specification-based approach [10] that perfectly fits. Anyway, some concepts are necessary because we will not be able to rebuild semantics in the relational database otherwise. This is why complex objects can be modeled by means of multi-valued attributes: staff in interface Department models a member relationship. We need the information for establishing correct foreign keys in relational tables (cf. staffOf in table employee). Those nested structures occur quite often in files [5], e.g., the nesting of headings in WWW pages or \(sub)section{...} in Tex files.

We also enhance ODL in order to be able to describe file contents as precisely as possible. We add an order by clause specifying that an object type must be ordered: Departments occur in the file with ascending id's. It is essential to express this, because otherwise the adapter has no chance to write back the file correctly in this order!

The second part of the specification consists of a *context-free grammar* for the data file in an extended Yacc style. At first, file(personnel,FILE) relates the file 'personnel' to the nonterminal FILE; FILE is the starting symbol for analyzing the file.

We extended Yacc in order to ease the writing of specifications.



ODL schema + Assignments:

```
interface Department  from DEPT[ DNUM<10 (short,ET_ASCII) ]
   (key  id)
    order by id asc;
{ attribute short  id        = DNUM (ET_ASCII);
   attribute String name  = DNAME;
   attribute Employee head          = MGR;
   attribute Set<Employee> staff   = EMP[1-];       };
interface Employee from EMP
{  attribute short  id       = ENUM;
   attribute String name  = ENAME;
   attribute String salary = SALARY;   };
```

Grammar:

```
file(personnel,FILE);    // start symbol
FILE:     STAFF[0-];
STAFF: DEPT   MGR   EMP[1-];
DEPT:   "100,"   DNUM   ","   DNAME   _EOL;
MGR :  EMP;
EMP:    "200,"   ENUM   ","   ENAME   ","   SALARY   _EOL;
DNUM:     _NUMBER;
DNAME:    _ID;
ENUM:     _NUMBER;
ENAME:    _ID;
SALARY:   _NUMBER;
```

Figure 1: ODL_{File} Specification

The specification language offers the possibility to use repetition groups <nonterminal>[n-m] known from SGML. Those repetitions clearly improve the readability and the maintainability of a specification. FILE should here contain one or more bulks ([0-]) of STAFF information. Each STAFF has one DEPT record, one MGR record, and several EMP records. DEPT records possess the form 100, comma, DNAME, comma, DNUM, and end-of-line. The structure of EMP and MGR records is similar.

Terminal symbols denote the characters of the data file. They can be defined either by embedding the characters in quotation marks such as "100", or by using predefined nonterminals. For example, the predefined symbol _NUMBER stands for an optionally signed integer number. Similarly, _ID represents a sequence of letters, _DIGIT any digit, _BYTE any byte character, and so on. The predefined symbols ensure shorter specifications. Particularly, files with fixed-sized records can effectively be described by using _NUMBER[6] (exactly 6 digits) and so on.

The grammar describes the structure of data files. A corresponding parser can then collect the information from files. *Assignment rules* now relate the parsed information to objects in the ODL schema. In fact, those rules provide some abstract information that allows our generator to enrich a Yacc-parser with semantic actions building objects from parsed file records.

Assignment rules extend the ODL schema by means of a from clause and attribute equations. These parts are put in italics in Figure 1. Department from DEPT determines that the nonterminal DEPT characterizes Department objects. Any time the rule for DEPT is passed by the parser, a new Department object is created. This defines the boundaries of objects and fixes an object as "current": Equations then describe how to fill attributes of the current object.

Any grammar rule gives a nonterminal a certain value that is used for filling attributes. The value of DNUM is exactly the number that is parsed, e.g., "1" after analyzing the first department record. This value is exported to set the attribute id of Department by means of id = DNUM. The value of MGR is the one of EMP which is composed as follows: "200," is a terminal symbol which naturally has the value "200,". ENUM, ENAME, and SALARY are other nonterminals which are later assigned the values "21", "Taylor" and "5000" after analyzing a manager record. Hence, MGR has the value "200,21,Taylor,5000". Equations describe that head receives the value of MGR. Similarly, the value of EMP[1-] is computed by concatenating the EMP values. staff = EMP[1-] specifies how to establish the attribute staff for the current department. The concept is very powerful, as we can express nested structures in any depth.

The approach integrates some other important features such as encoding and filtering. It is a difference, whether an integer occurs in a binary format or as an ASCII-number in a file. Hence, it is important to specify how the data in the file is encoded. ODL_{File} offers the possibility to distinguish between different *encodings*. For example, ET_ASCII can be used for ASCII-text, while ET_IEEE is used for IEEE binary encoding. id = DNUM (ET_ASCII) then specifies that the parsed DNUM-value should be ASCII-decoded before assigning to the id attribute of the current Department-object. ASCII-decoding is the default.

Filtering is important as there are some file formats that only mark records in a file as deleted, but do not really remove them. Consequently, it is no good to take those records for objects. To this end, filters can be specified in ODL_{File} for the object types constructed from a data file. [DNUM<10 (short,ET_ASCII)] specifies that Department records are only exported if the value of DNUM is lower than 10. The condition requires a data type, since the nonterminal DNUM might not be used as attribute value.

Using an ODL_{File} specification, a relational database can be installed easily. The tables are generated according to some standard mechanism to map the ODL_{File} schema onto tables. The handling of multi-valued attributes is straightforward: For any such attribute, e.g., staff, a table is created that takes the values entry by entry. An artificial foreign key staffOf is introduced for employee in order to refer to the "base" table department. Please note that the generator has to know the key of the base tables. This is why a key clause is necessary for the department interface.

The objects created by the analyzer are mapped to tuples according to the table structures. Running our migration procedure, the file data will be stored in tables department and employee.

3. SPECIFICATION LANGUAGE

In order to demonstrate the expressive power of ODL_{File}, we present some examples that show the handling of specific points.

3.1 Table Structures in Files

Table structures in files can be handled quite easily. The most common format is the "dbase" format, which stores objects horizontally value by value:

emp	id	name	salary
	11	Johnson	1000
	12	Miller	2000
	13	Stone	3000

The following specification produces an object type Emp.

```
interface Emp from TAB1          file(f,TAB1);
{ attribute int id      = ATTR1;  TAB1: "emp" _BLANK _ID BLANK
  attribute int name = ATTR2;          _ _ID_BLANK _ID EMP[1-];
  attribute int salary = ATTR3;  EMP:  ATTR1 _BLANK ATTR2
};                                     _BLANK ATTR3 _EOL;
                                 ATTR1:  _NUMBER;
                                 ATTR2:  _STRING;    ...
```

The grammar asks for a token "emp" and introduces a sequence of _BLANK _ID just to skip the headline; the _IDs are not used as information. Afterwards the structure of the table is described.

Please note this is a style similar to defining tables in HTML. There, a description of table looks as follows in a HTML source.

```
<TABLE...> <TR ALIGN=left>
  <TH> id </TH> <TH> name </TH> <TH> salary </TH> </TR>
  <TR ALIGN=left> <TD> 11 </TD> <TD> Johnson </TD>
  <TD> 1000 </TD> </TR>  ...
</TABLE>
```

A specification must handle bracketing such as <TD>...</TD> etc. as terminals in addition. This is precisely expressible in any depth in a context-free grammar.

A file may contain several tables in this manner. A corresponding specification then starts with FILE: TAB1 & TAB2 & TAB3. ODL_{File} provides an operator '&' in order to express in a comfortable manner that tables TAB1, TAB2 and TAB3 may occur in any order within the data file.

Just to demonstrate the power of our approach, we show another representation of tables that stores attribute values vertically: id 11 12 13 ... . An entry is built from one column instead of a row. This table form sometimes occurs in HTML files, if the number of entries is fixed, but the number of rows is subject to changes. It has advantages as the record structure can grow vertically.

The following specification is a little tricky because the creation of Emp objects is bound to ATTR1: Any time ATTR1 is passed, a new Emp object has to be created.

```
interface Emp from ATTR1
{ attribute integer id      = ATTR1;
  attribute integer name = ATTR2;
  attribute integer salary = ATTR3;
};
```

```
FILE: "emp" VALUES1
      _EOL  VALUES2
      _EOL  VALUES3;
VALUES1:  _ID _BLANK
          ATTR1[1-];
...
```

3.2 Hyperlinks

Handling hyperlinks, which occur as <A HREF=http:address> in WWW pages, requires some specific concepts. We assume a file with several entries of the form dept emps eol where any emps (underlined) is a link to another file containing the names of employees in this department. In principle, we could describe such a hyperlink by means of "<A HREF=" LINK "> ..." in a grammar. A nonterminal LINK defines the structure of the file referred to. But the parser will not be able to analyze LINK, because it is part of a different file. We introduce a predefined keyword _REF to let the parser take LINK as a string, the filename.

```
interface Dept from DEPT
{ attribute String name = DNAME;
  attribute Set<Emp> staff = LINK;   };
```

```
interface Emp from EMP
{ attribute name = EMP; };
```

```
FILE:   DEPT [1-];
DEPT: DNAME   _BLANK   "<A HREF=http:"
      _REF  LINK  ">"  NAME  "</A>";
DNAME:   _ID;
LINK:    EMP[1-];
EMP:     _ID;
NAME:    _ID;  // name of the link in browser
```

The parser ignores the nonterminal LINK (due to _REF LINK) and collects the http address of the link instead. NAME obtains the name of the link. Later on, LINK is used as a starting symbol for the addressed file. Then the rule LINK: EMP[1-] is applied.

3.3 Handling File Systems

Let us now take into account several files organized in a directory hierarchy. The directory structure may contribute to the modeling of file contents. As far as we perceived, there has been no solution in the literature considering this kind of semantics.

We start with isolated files, each one containing some objects o*i* of one type O*i*. Hence, *a file f_i represents one object type O_i*.

```
Files:   f1: o1 ... o1        file(f1,FILE1); FILE1: OBJ1[1-]; ...
         f2: o2 ... o2        file(f2,FILE2); FILE2: OBJ2[1-]; ...
```

The principle of the specification consists of giving each file f*i* a starting symbol of its own. The file clause determines which grammar (start symbol) is used to analyze the file. OBJ*i* rules represent the object structure.

It might happen that the objects of one type are spread over several files, e.g., *each file* in a certain directory /dir *is one (complex) object* of that type. Hence the directory denotes an object type.

```
Files:  f1: o1o2o3...o3o2o3...o3...     file(/dir/*,DATA1);
        f2: o1o2o3...o3o2o3...o3...     DATA1: OBJ1 DATA2[1-];
        f3: o1o2o3...o3o2o3...o3...     DATA2: OBJ2 OBJ3[1-];
                                        OBJ3: ...
```

Since the objects possess the same structure, one grammar can be used to define the structure of several files. file determines the files to be analyzed by the start symbol, e.g., file(dir/*,DATA1) takes all the files in /dir into account. They are all parsed by the same grammar starting with nonterminal DATA1. The grammar extracts exactly one complex object of type OBJ1, including subobjects of types OBJ2 and OBJ3. The files can be scoped by usual Unix wildcards '*' (any sequence of symbols) and '?' (an arbitrary symbol) to denote several files, e.g., file(/dir?/*/f*, DATA1) qualifies all the files starting with an f in any directory under /dir1, /dir2, ..., /dira, /dirb and so on. Several of those file specifications are possible within file.

If we want to use the filename as an attribute value for the object type O1, we can add <attr> = _NAME; a predefined function _NAME computes the filename in assignments.

A directory can represent an object type. For example, each file f*ij* in directory dir*i* represents a (complex) object of type O*i* below:

```
Files:  dir1 = { f11,...,f1m }    file(/dir1/*, DATA1); DATA1: OBJ1[1-];...
        dir2 = { f21,...,f2n }    file(/dir2/*, DATA2); DATA2: OBJ2[1-];...
```

All the files in directory dir*i* are parsed by the same grammar with start symbol DATA*i*. DATA*i* defines the structure of type O*i*.

3.4 Directories Representing Relationships

We found some file systems where a directory represents some kind of relationship between the objects contained in its files. Let us assume directories that contain three files, f1, f2 and f3, each one related to a type O*i*. Those directories can represent a ternary relationship between the objects in files. If a file contains several objects, then the relationships consists of the "cross product": Any object in f1 is in relationship with any object in f2 and in f3.

```
dir1 = { f1, f2, f3 }    f1: {2,3}   ;  f2 : {1,2}   ;  f3 : {1,2,4}
dir2 = { f1, f2, f3 }    f1: {3,4}   ;  f2 : {3,4,5} ;  f3 : {3}
```

The following specification again makes use of predefined functions for accessing file and directory names. A function _DIR determines the directory of the current file, _NAME yields the name of a file or directory, and _PATH gives out the complete path as a string. They can be used for assignments like nonterminals. This enables us to capture this additional source of semantics.

```
interface O1 from DATA1                 file(dir*/f1, DATA1);
{ attribute attr1 = ...;                DATA1: OBJ1[1-];
  attribute  relship = _DIR._NAME; };   ...

interface O2 from DATA2                 file(dir*/f2, DATA2);
{ attribute attr1 = ...;                DATA2: OBJ2[1-];
  attribute  relship = _DIR._NAME; };   ...

interface O3 from DATA3                 file(dir*/f3, DATA3)
{ attribute attr1 = ...; ...            DATA3: OBJ3[1-];
  attribute  relship = _DIR._NAME; };   ...
```

Please note the relationship is not directly expressed in the ODL schema. This is because it is difficult to describe the association between a relationship and its various directory representations. Nevertheless, we use the directory name as an attribute relship.

O1	id	relship
	2	dir1
	3	dir1
	3	dir2
	4	dir2

O2	id	relship
	1	dir1
	2	dir1
	3	dir2
	4	dir1

O3	id	relship
	2	dir1
	3	dir1
	3	dir2
	4	dir2

relship occurs in the tables. Now it is possible to join the tables via relship to compute the relationship in a SQL-like manner. This principle is also applicable for directory hierarchies, since it is possible to navigate along the hierarchy by means of _DIR._DIR ....

4. CONCLUSIONS

This paper presented a generative, specification-based approach to migrate data from several files into a relational database. A generator automatically installs adequate tables and fills them with file data; it also stores modified data back into files.

The approach is embedded in a federation framework FIHD (Flexible Integration of Heterogeneous Data sources) [11]. Similar to TSIMMIS [22] and Information Manifold [15], FIHD has the goal to provide an easy access to heterogeneous data without knowing the exact source and type of source. A generative principle produces ODMG adapters that homogenize object-oriented and relational databases. Those adapters can be plugged in a federation framework that gives a transparent ODMG access. Generators have been built to produce adapters for commercial relational and object-oriented DBSs. Files must be transferred to a relational database to be plugged in a federation.

In the context of this paper, semantic enrichment [10], one of the building blocks of FIHD, is worth mentioning: The semantics inherent to relational tables, is made explicit by using object-oriented modeling concepts. A real object-oriented view, including subtyping and relationships, is achieved. A generator produces an ODMG2.0 conforming manipulation and querying interface. This gives users the opportunity to see relational data in an object-oriented way, to manipulate relational data in terms of C++ objects and thus hiding the relational structure. Moreover, data can be queried in a more powerful way by means of the object-oriented extension OQL of SQL. Using this in addition to file migration, the file contents are manageable by an ODMG access interface [8]. This opens the door to migrate file data in an object-oriented database by means of our federation approach.

Future work will be dedicated to integrating file systems into the federation framework directly, avoiding the detour via relational databases. It then makes sense to incorporate the ideas of semantic enrichment into generating file adapters. We also think of other types of front-end interfaces for the federation framework. Owing to our FIHD architecture, it is easily possible to generate any type of interface instead of ODMG2.0, e.g., ActiveX components, or HTML pages to visualize information obtained from the federation. Finally, we feel the need for a comfortable graphical support for defining specifications similar to [10].

5. REFERENCES

[1] S. Abiteboul: Querying Semi-Structured Data. In [12]

[2] S. Abiteboul, S. Cluet, T. Milo: Querying and Updating the File. In Proc. Conf. on Very Large Databases (VLDB) 1993

[3] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, Y. Zhuge: Views for Semistructured Data. In: [4]

[4] ACM SIGMOD: Workshop on Management of Semistructured Data. Tucson (Arizona) 1997. Superseded by ACM SIGMOD Record 26(4), Dec. 1997

[5] N. Ashish, C. Knoblock: Wrapper Generation for Semi-structured Internet Sources. In [4]

[6] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu: A Query Language and Optimisation Techniques for Unstructured Data. ACM SIGMOD Conf. on Management of Data, Montreal 1996

[7] P. Buneman, S. Davidson, M. Fernandez, D. Suciu: Adding Structure to Unstructured Data. In [12]

[8] R. Cattell (ed.): The Object Database Standard: ODMG2.0. 2nd edition, Morgan-Kaufmann Publishers, San Mateo (CA) 1997

[9] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, A. Crespo: Extracting Semistructured Information from the Web. In [4]

[10] U. Hohenstein, C. Körner: A Graphical Tool for Specifying Semantic Enrichment of Relational Databases. In: 6th IFIP Conf. on Data Semantics (DS-6) "Database Applications Semantics", Atlanta 1995

[11] U. Hohenstein, V. Pleßer: A Generative Approach to Database Federation. 16th Int. Conf. on Conceptual Modeling - ER'97, Los Angeles 1997, Springer LNCS 1331

[12] ICDT'97: Int. Conf. on Database Theory, Delphi 1997

[13] IEEE'95: Legacy Systems. Special Issue of IEEE Software 12(1), 1995

[14] V. Kashyap, M. Rusinkiewicz: Modeling and Querying Textual Data using E-R models and SQL. In: [4]

[15] T. Kirk, A. Levy, Y. Sagiv, D. Srivastava: The Information Manifold. In Proc. of the AAAI Spring Symp. Series, March 1995

[16] D. Konopnicki, O. Shmueli: W3QS: A Query System for the World Wide Web. In 21st Int. Conf. on Very Large Databases (VLDB95), Zurich 1995

[17] N. Kushmerick, D. Weld, R. Doorenbos: Wrapper Induction for Information Extraction. In Int. Joint Conf. on Artificial Intelligence, Nagoya (Japan) 1997

[18] L. Lakshamanan, F. Sadri, I. Subramanian: A Declarative Language for Querying and Restructuring the Web. 6th Int. Workshop on Research Issues in Data Engineering (RIDE96), New Orleans 1996

[19] A. Levy, A. Rajaraman, J. Ordille: Heterogeneous Information Sources Using Source Descriptions. In 22nd Int. Conf. on Very Large Databases, 1996

[20] A. Mendelzon, G. Mihaila, T. Milo: Querying the World Wide Web. In Symp. on Parallel and Distributed Information Systems, Miami 1996

[21] S. Nestorov, S. Abiteboul, R. Motwani: Inferring Structure in Semistructured Data. In: [4]

[22] Y. Papakonstantinou, H. Garcia-Molina, J. Widom: Object Exchange Across Heterogeneous Information Sources. IEEE Conf. on Data Engineering 1995

[23] W. Premerlani, M. Blaha: An Approach for Reverse Engineering of Relational DBs. CACM 37(5), 1994

[24] Y.-H. Wu, Y.-H. Chen, A.L.P. Chen: Querying and Browsing the Resources in Internet. In Proc. of Int. Conf. on Distributed Systems, Software Engineering, and Database Systems, 1996