PLUGGING FILES IN DATABASE FEDERATIONS

Uwe Hohenstein & Andreas Ebert
Siemens AG, Corporate Technology, ZT SE 2, D-81730 München, GERMANY
E-mail: {Uwe.Hohenstein,Andreas.Ebert}@mchp.siemens.de

ABSTRACT

In order to use files in database federations, we propose to migrate file data into a relational database. Integrating the database in a federation is then well-understood. This procedure has the advantage that a „real" database is handled supporting transactions and SQL; files obtain a high-level interface for free.
This paper presents a specification-based approach to support the process of migrating file data into a relational database, and vice versa. A specification language provides powerful concepts to describe the contents of files and directories. In contrast to similar work, directory structures are taken into account, because they contain useful semantics. Given some descriptive specification of file contents, an adapter is generated moving data from several files into the database, and the other way round.

INTRODUCTION

Federated DBSs provide solutions to handle data stored in several autonomous sources (Sheth and Larson (1990), Conrad et al. (1997), Conrad et al. (1999)), giving a user the illusion of a homogeneous „database system" (DBS). A unified and consistent view of the stored data resolves discrepancies (Saltor et al. (1992)) and conflicts (Spaccapietra and Parent (1994)) between database schemata, which result from representing real world situations in different ways. A transparent access ensures that users are not aware of the location of items in a particular system. With all that, the autonomy of the constituent systems is preserved.
Research in the field of federated DBSs has brought up several prototypes (Huck et al. (1994), Kuno and Rundensteiner (1996), Roantree et al. (1999)) and results such as integration methodologies (Reddy et al. (1994), Schmitt and Saake (1996)), languages for view definitions (Kaul et al. (1990)), global transaction management and query processing. Most effort is dedicated to „real" database systems; incorporating files is neglected. But there are huge amounts of interesting data kept in ordinary files. For example, engineering applications could not live with the bad performance of relational DBSs for handling complex structures. In absence of object-oriented DBSs, those applications were forced to keep persistent data in files, and they still do due to the legacy problem. Besides legacy applications, files also get new importance in the context of the world-wide web (WWW): Web pages are stored in files, and generated pages can also be seen as files. Other examples for maintaining persistent data in files are electronic documents and genome databases. These all are important information sources, and thus candidates for being plugged in database federations.
According to the architecture of Sheth and Larson (1990), any data source in a federation requires an adapter that is responsible for homogenizing the data according to a canonical data model. An adapter maintains a view of the data in the canonical model. Moreover, the adapter has to provide a corresponding interface which must be implemented on top of the local source (Roantree et al. (1999)). Implementing an adapter for files is very cumbersome because files are somewhat different to databases: They do not possess an explicit schema, and they provide only a simple interface for reading and storing data. Nowadays as object-oriented canonical data models are frequently used, e.g., Huck et al. (1994), Radeke (1995), Hohenstein and Ebert (1999), Roantree et al. (1999), the gap to bridge has become even huger; particularly if the semantic power of the canonical model is used, and if a complete interface such as ODMG OQL (Cattell and Barry (1997)) is supported.
Papers such as Höding (1996) emphasize the necessity to take files into account for federations and analyze the problems that occur. But there are still no adequate solution to ease the integration of files. Abiteboul et al. (1993) show how to generate a parser that provides an object view of file data. But this parser must be manually enhanced with semantic actions that collect information in a structured way and build objects. This is a complex task that requires a lot of knowledge in compiler technology, even if compiler-compilers like Yacc are used. So far, this is a manual task to be done for each file again.
Other helpful solutions are coming from the field of semi-structured data. Most work here relies on light-weight data models (Buneman et al. (1997), Nestorov et al. (1997)) that define a merely untyped node structure being less strong than classical models. In order to organize file data according to that data model, files are wrapped.
A manual implementation of wrappers is often impractical if the format of sources changes frequently, as it is especially the case for the WWW. Hence, Ashish and Knoblock (1997) suggest a semi-automatic way to generate wrappers around internet data sources. Keywords are taken as tokens of

interest, and the nested document structure is derived from the source. Ashish and Knoblock (1997) present regular (Lex) expressions for detecting usual representations such as headings and emphasizing tags. These are heuristics, whereupon a wrapper is generated. The wrapper then allows users to query web documents in a database-like fashion. However, the overall approach is tailored very much to internet sources.

Kushmerick et al. (1997) propose wrapper generation, too. But the paper makes more assumptions about the data to be looked for. A powerful approach has also been developed in the TSIMMIS project (Papakonstantinou et al. (1995), Hammer et al. (1997)). On the basis of specified templates that model file data in OEM (Object Exchange Model), wrappers are generated. They claim that regular expressions are simpler to use for specifying templates. Sattler and Höding (1999) propose a similar solution and discuss how to extract relational schemata from web sources. Besides being tailored to WWW sources, all those approaches lack of two important points:

- They take into account single files and do not cover the semantics of directory structures. But sometimes, records in *several* files constitute an object type of interest. Moreover, directories are often used to express relationships.
- They stress on querying (mostly the web) and are not concerned with updates and transactional support. This limits their usage in database federations.

This paper now provides an effective solution to incorporating files in a database federation thereby abolishing the above deficits. We suggest using a relational database as intermediate storage for handling files. This seems to be an unnecessary indirection, but has some obvious advantages:

- Provided we store file data in a fine-granular manner in tables, the federation can use SQL to query and manipulate the data. Similarly, the transactions of the database system can be taken. Users of the federation are synchronized by the DBS, as the database is just a „cache“ for file data. Certainly, external users of files must be locked until modified data is written back to files. That is, although the locking unit is a file, federation users can work concurrently. Using DBS features thus makes the implementation of an adapter much easier.
- Furthermore, the step of expressing the semantics of file data in a high-level canonical data model can be divided into two simpler steps:
  (a) Giving file data a simple relational structure first, and then
  (b) enhancing the semantics in a semantic enrichment process (Hohenstein and Körner (1995))
  Afterwards, the federation framework of Hohenstein and Ebert (1999) can take benefit from the higher semantic level.

The remainder of this paper discusses both steps (a) and (b) in more detail and presents an adequate support. The next section elaborates on the underlying generative principle, which relies on two corresponding generators: The first one produces a file wrapper that converts file data into relational tables in a structured manner. The wrapper is also responsible for writing data back into files after modifications (Abiteboul et al. (1995)). This releases one from the tremendous task of building Yacc programs as it is necessary in Abiteboul et al. (1993). A second generator produces an ODMG adapter for the relational database to be plugged in our federation approach (Hohenstein and Ebert (1999)). Both generators perfectly cooperate and provide together a real object-oriented view of data and a corresponding ODMG access to files.

However, the generators need some input. This is done by means of logic-based specifications. The third section presents the essentials of a first specification language ODL_{File} for specifying file contents. An ODL_{File} specification is input to the generator that produces the migration into the database. Several examples will demonstrate the expressiveness of the specification language. Relevant file formats are discussed, and it is shown how to handle them. In addition to our previous work in Ebert et al. (1999), we introduce some new features to incorporate several files and their directory structure. It is important to keep this kind of semantics. Furthermore, we present concepts to handle WWW sources. In contrast to other proposals, e.g., Ashish and Knoblock (1997), and Hammer et al. (1997), our language is very general and suited for any kind of files.

The fourth section presents a second language ODL_R that enables specifying semantic enrichment of relational databases. We discuss how the language enhances the semantic level of a relational database. The focus lies on concepts that are important for giving tables with file data a further upgrade.

In the conclusions, we summarize our ideas and outline some future work we are planning to do.

GENERATIVE PRINCIPLE

Figure 1 gives an overview of the principle of having a generator for migrating data from files in a relational database and plugging the database in a federation. There are two generators with corresponding specification languages.

A generator GEN_{File} produces a file adapter, consisting of a parser and an unparser. The generator requires some input given by a specification. A corresponding specification language ODL_{File} has the goal to define a „schema" for file contents in a certain sense. The syntax of ODL_{File} is a mixture of the ODMG ODL (Cattell and Barry (1997)) and an enhancement of Yacc.

1. A *subset of ODL* is used to describe the schema of data files in an intuitive way. File contents are modelled independently of the physical file structure.
2. A Yacc-part then defines a *context-free grammar.* The grammar precisely describes the contents of files and is the basis for a parser. Using a Yacc-style eases the later generation of a parser by means of this compiler-compiler. We extended Yacc to ease the writing of specifications. For example, the specification language offers repetition groups known from SGML, and concepts to express hyperlinks between files.
3. Parsing a file, the collected information has to be organized according to the ODL schema. So-called *assignment rules* now relate the parsed information to objects in the ODL schema. It is important to note that objects can be built from several files in several directories. Assignment rules handle that, too.

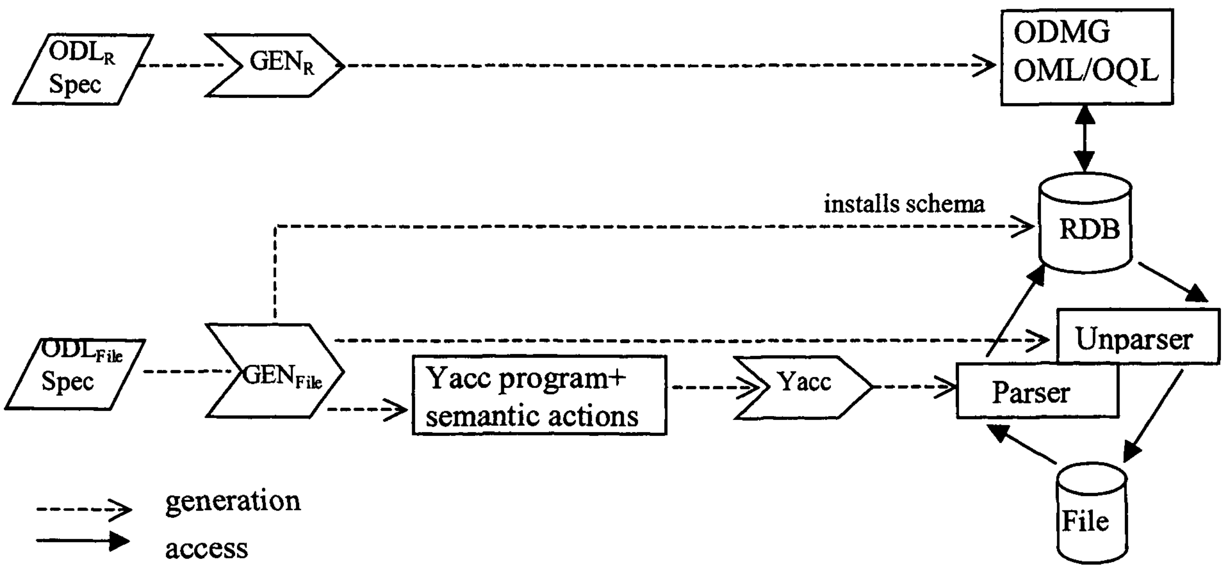


Figure 1: Generating Parser and Unparser

The ODL part (1) of an ODL_{File} specification is used to derive adequate relational tables. It may be surprising why we use the higher ODL to format file data in relational tables, and although ODL is used, why not the full power but only a subset. On the one hand, we rely on ODL because multi-valued attributes are necessary to specify nested structures. Using only flat tables, the association between nesting and foreign keys is difficult to express, as we will see later. On the other hand, we use only „complex objects"; subtypes and explicit relationships between object types are omitted. Although useful from the point of modeling file data, we cannot really benefit from the object-oriented schema, because we have to map it afterwards onto flat tables. Moreover, subtypes and relationships are difficult to handle because their representations in files are much more manifold than those discussed by Castellanos and Saltor (1991) and Chiang et al. (1994) in the field of semantic enrichment of relational databases. Such an enrichment of files should better be done by a successive semantic enrichment step, e.g., using the specification-based approach discussed later in this paper.

The generation of tables is very straightforward. There are no big problems as an ODL_{File} schema does not contain subtypes and relationships. Each object type is mapped to a base table, and any multi-valued attribute gets an additional table in the sense of normalization. This table takes the values entry by entry. Each entry possesses a foreign key to its related base table.
The generator uses the grammar (2) of the specification and produces a Yacc program that is able to parse files. But a parser is not enough, since it only complains if a file does not satisfy the specified grammar. Assignment rules (3) provide some abstract information that allows the generator to enrich a Yacc-parser with semantic actions. Those semantic actions build objects from parsed file records according to the ODL schema. Afterwards transferring complex ODL objects into the relational database takes place.
Unfortunately, there is no tool support for building unparsing. Consequently, the part of GEN_{File} for generating unparsers must be implemented from scratch.
Aa second generator GEN_R produces an ODMG adapter that presents a real object-oriented view of the relational data and provides an OML/OQL access. In order to control the building of the view, a semantic enrichment specification in a language ODL_R is necessary. The ODMG adapter can be plugged in a federation as discussed in Hohenstein and Ebert (1999).
The data collected by GEN_{File} and GEN_R is stored in a meta-database similar to (Roantree and Murphy (1997)). The meta-data consists of any information found in specifications, i.e., ODL schemata, the structure of the grammar, the assignments as well as the mapping of an ODL schema to a relational schema. This makes the implementation of the generators easier, as producing the Yacc program does not need to be done during parsing a specification.

SPECIFICATION LANGUAGE ODL_{File}

This section presents several schematic examples that sketch out in a systematic manner how to handle important file formats by ODL_{File}. We start with commonly used organizations of objects in a single file and then switch to directory structures, before discussing some special topics such as hyperlinks.

Handling Single Files

a) Table structures in files

We start with a single file that contains records in tabular form to outline the basic principle of ODL_{File}. The most common format is the „dbase“ format, which stores objects horizontally value by value. The first line of the file introduces the structure of the data: A name O1 for the table and three attributes Attr1, Attr2, and Attr3. Each row is then an object*i* = (Value*i*1, Value*i*2, Value*i*3). A marker "-" should denote records that are deleted.

File:

```
O1   Attr1     Attr2     Attr3
     Value11   Value12   Value13
–    Value21   Value22   Value23
     Value31   Value32   Value33
```

Specification 1:

```
/* ODL part + assignment rules */
interface O1 from OT1[OK!="-"]
{ attribute int          attr1 = ATTR1 ;
  attribute String       attr2 = ATTR2 ;
  attribute int          attr3 = ATTR3 ;
}

/* grammar */
file(O1.tab,DATA1);
DATA1:   _ID " " _ID " " _ID " " _ID _EOL OT1[1-];
OT1:     OK[0-1] " " ATTR1 " " ATTR2 " " ATTR3 _EOL;
OK:      "-";
ATTR1:   _NUMBER;
ATTR2:   _ID;
ATTR3:   _NUMBER;
```

The ODL schema defines a corresponding interface O1. The structure of O1 represents the table directly. This part is the logical view of file contents. It does not depend on how objects are organized in the file. According to our standard mapping strategy, the ODL part is mapped on a relation O1(attr1 int, attr2 varchar, attr3 int) directly.
The grammar defines the structure of the file. It first introduces a start symbol DATA1 for the file „O1.tab" by means of a file clause. DATA1 possesses a rule DATA1: ... that defines the file contents, starting with _ID and a sequence of " " _ID just to skip the headline; the _IDs are not used as information. After an end-of-line (_EOL), one or more ([1-]) OT1-records can occur. The rule for OT1 describes the structure of a single row of the table. Such a record consists of an optional marker "–" (OK), a number (ATTR1), a string (ATTR2), and a number (ATTR3), all separated by a blank " ". Terminal symbols denote the characters found in the data file. They are embedded in quotation marks. Furthermore, there are several predefined non-terminal symbols. For example, _NUMBER stands for an optionally signed number. Similarly, _ID represents a sequence of letters, _DIGIT any digit, _BYTE any byte character, _EOL an end-of-line, and so on. The predefined non-terminals are useful because they make specifications shorter and thus improve readability.
We extended Yacc in order to ease the writing of specifications. The specification language offers repetition groups known from SGML. <non-terminal>[n-m] determines that <non-terminal> may occur at least n, but at most m times. [n-] means at least n times, [-m] atmost m times, and [n] exactly n times. The optional marker can thus be expressed by OK[0-1].
Assignment rules now relate the parsed information to objects in the ODL schema. The rules extend the ODL schema by means of a from clause and attribute equations. These parts are put in italics in the above example. O1 from OT1 determines that the non-terminal OT1 characterizes O1 objects. Technically speaking, any time the rule for OT1 is passed by a parser, a new object of type O1 is created. Such a rule fixes an object as „current". Equations then describe how to fill attributes of the current object by using the *values* of non-terminals. Each non-terminal has a certain value. This value is a string, which arises as a concatenation of the values of the non-terminals on the right side of its rule: The value of ATTR1 is exactly the number that is parsed, e.g., "11" after analyzing the second row. This value is exported to set the attribute attr1 of O1 by means of attr1 = ATTR1. The value of OT1 (if used) is computed by concatenating the value of ATTR1, a blank (which naturally has the value " "), the value of ATTR2, a blank, the value of ATTR3, and an end-of-line.
The approach allows one to filter records. Filtering is important as there are some file formats that do not really remove records. Instead they only mark records in a file as deleted to increase performance. Consequently, it is no good to export those records because they should not be there. ODL_{File} offers special conditions that restrict exporting records. For instance, OT1[OK!= "–"](int) determines to export only those records that have no marker "-". Each condition is related to a non-terminal and requires a data type, since the comparing non-terminal is usually not used as an attribute value.
Please note the above file content has a style similar to defining tables in HTML:

```
<TABLE ... >
<TR ALIGN=left> <TH> Attr1    </TH> <TH> Attr2    </TH> <TH> Attr3    </TH> </TR>
<TR ALIGN=left> <TD> Value11</TD> <TD> Value12 </TD> <TD> Value13 </TD> </TR>
. . .
</TABLE>
```

A specification must handle <TABLE> etc. as terminals in addition to Specification 1.
Just to demonstrate the power of our approach, we show another representation of tables that stores objects vertically. An object*i* = (Value*i*1, Value*i*2, Value*i*3) is built from one column. This table form sometimes occur in HTML files, if the number of entries is fixed, but the number of columns is subject to changes. This form has advantages as the record structure can grow vertically.

File:

```
Attr1  Value11   Value21   Value31   . . .
Attr2  Value12   Value22   Value32   . . .
Attr3  Value13   Value23   Value33   . . .
```

Specification 2:

```
interface O1 from ATTR1
{ attribute int         attr1 = ATTR1;
  attribute String      attr2 = ATTR2;
  attribute int         attr3 = ATTR3;
}
file(vertical.tab,FILE);
FILE:   _ID " " ATTR1[1-] _EOL
        _ID " " ATTR2[1-] _EOL
        _ID " " ATTR3[1-];
ATTR1: _NUMBER " ";
 ...
```

This specification is a little tricky because the creation of O1 objects is bound to ATTR1: Any time ATTR1 is passed, a new O1 object has to be created.

b) Several objects of several types in one file
There are several possibilities how records of different structure or type can be found in files. The next example discusses several ways to separate records (objects) and fields. We assume a file f

```
        o1 ... o1
        o2 ... o2
        o3 ... o3
```

where objects are arranged according to their type. o1 should be an object of type O1, o2 an object of type O2 etc.

Specification 3:

```
interface O1 from OT1
{  attribute int attr1 = ATTR11;
   attribute int attr2 = ATTR12;
};
interface O2 from OT2
{  attribute String   attr1  = ATTR21 [EBCDIC];
   attribute int      attr2  = ATTR22;
   attribute String   attr3  = ATTR23;
};
interface O3 from OT3
{  attribute int attr1 = ATTR31;
   attribute int attr2 = ATTR32;
};
file(f,FILE);
FILE:    OT1[1-] _EOL OT2[1-] _EOL OT3[1-];
OT1:     ATTR11 " " ATTR12 " ";                  // (1) separator blank
OT2:     ATTR21 ATTR22 ATTR23;                   // (2) fixed-size
OT3:     VAL31 & VAL32[0-1];                 // (3) token-based
ATTR11:  _NUMBER;
ATTR12:  _NUMBER;
ATTR21:  _BYTE[4];
ATTR22:  _DIGIT[5];
ATTR23:  _LETTER[2];
VAL31:   "Attr31: " _COLON ATTR31 _EOL;
ATTR31:  _NUMBER;
VAL32:   "Attr32: " _COLON ATTR32 _EOL;
ATTR32:  _NUMBER;
```

The ODL schema models three object types O1, O2 and O3, each one possessing certain attributes.
The non-terminals OT1, OT2, and OT3 directly correspond to the object types: A rule OT1 = ... describes the structure of O1 objects, and so on. The example presents three approaches for separating fields in a record. Again, each interface O*i* describes a corresponding table O*i*.

1. Using separators, e.g., blanks (" ") between values for OT1; the blank is necessary because otherwise we do not know where the first integer stops.

2. OT2 records have a fixed size: ATTR21, ATTR22, ATTR23 have a fixed length defined by _BYTE[4], _DIGIT[5] and _LETTER[2]. That is, predefined terminal symbols are useful to describe fixed-sized records.
3. OT3 is token-based: Tokens "Attr31:" etc. denote that the subsequent values belong to this attribute. Typical examples for this style are bibtex files and WWW pages. '&' expresses that the ordering of occurrences is irrelevant, i.e., VAL31 and VAL32 may occur in any order within the data. It is cumbersome to describe this variable order by only using alternatives '|' as provided by Yacc.

The separation of records in the file is just done by ordering the objects and indicating a switch of type by means of _EOL.

If the structures defined by OT1, OT2 and OT3 characterize the records of each type uniquely, then no _EOL is necessary. Above, the records are not identifiable because a record 111□22223□ [18] causes trouble: Is it an object (111,22223) of type OT1, or an object ("111□",2222, "3□") of type OT2?

If several clusters of object types occur in a file, e.g., "O1" o1...o1 "O2" o2...o2 "O1" o1...o1 "O3" o3...o3, a token has to introduce the change of object type. A specification then looks like:

```
FILE:   DATA[1-];
DATA:   TYPE1 & TYPE2 & TYPE3;
TYPE1: "O1" OT1[1-];
...
```

Again, if the structure of types is unique, then no special tokens are needed to distinguish the objects in the file.

Sometimes, brackets are used to separate records, e.g.:

```
FILE: "(" OT1[1-] ")(" OT2[1-] ")(" OT3[1-] ")".
```

Those context-free dependencies are easily expressible in a grammar. This type occurs in HTML files that for instance mark tables by means of <TABLE>... </TABLE>.

ODL$_{File}$ offers a possibility to distinguish between different *encodings* such as ASCII or EBCDIC. An assignment attr1 = ATTR21 [EBCDIC] could specify that the parsed ATTR21-value should be EBCDIC-decoded before being assigned to attr1 of the current O1-object. ASCII-decoding is the default.

c) Nested objects

The following example shows a file that expresses a nested structure as it occurs quite often (Ashish and Knoblock (1997)).

File:

```
o1
  o2
    o3o3...o3
  o2
    o3o3...o3
  ...
o1
  ...
```

Specification 4:

```
interface O1 from OT1
  (key attr1)
{  attribute int attr1 = ...;
   attribute Set<O2> setO2 = DATA2[1-];
};
interface O2 from OT2
  (key attr1 )
{  attribute int attr1 = ...;
   attribute Set<O3> setO3 = OT3[1-];
};
interface O3 from OT3
```

[18] The symbol '□' should denote a blank.

```
{ attribute int attr1 = ...;
};
file(f,FILE);
FILE:     DATA1[1-];
DATA1:    OT1 DATA2[1-];
DATA2:    OT2 OT3[1-];
OT1: ...
OT2: ...
OT3: ...
```

The indentation denotes the nesting, i.e., o1 contains several o2 objects that in turn possess o3 components. The ODL schema defines the nesting by means of multi-valued attributes setO2 and setO3, each one referring to a set of subordinated objects. Any of the collections (Set, Bag, List, Array) can be used as attribute domains.

The grammar uses repetition groups, e.g., DATA2[1-], to express the nesting. Those terms are directly taken in assignment rules for filling the set-valued attributes. Hence, setO2 = DATA2[1-] means that the value of DATA2[1-] is taken to fill setO2. The value itself is computed by concatenating the DATA2 values.

This concept is very powerful, as we are able to express nested structures in any depth. For example, it is easy to analyze the nesting of headings in WWW pages or \(sub)section{...} in Tex files.

The generated tables to keep the file data will then look like O1 (attr1 int, ...), O2 (attr1 int, ..., setO2fk int), and O3 (attr1 int, ..., setO3fk int). setO2fk is a foreign key that refers back to the key of table O1 to establish the relationship setO2, and similar for setO3fk. In order to insert values for foreign keys in tables, we need a primary key. That is why interface O1 and O2 must fix a key attribute by means of key. The key can certainly be composite.

Here we see why we use complex objects in the ODL part instead of flat relational structures. It is difficult to specify a nesting by having some kind of „flat table" specification. A first try may look in pseudo notation like

```
table O2 from OT2
{ attribute int attr1 = ...;
  attribute int setO2fk = ??? }
```

We are confronted with the problem of how to find the foreign key value for setO2fk since the context of rule OT2 is progressed too far; no right side of OT2 carries this information. Using multi-valued attributes is thus much easier.

Handling File Systems

So far, we have considered one file containing the objects of one or more types. Then it is sufficient to specify a grammar for the file, and to associate the filename with this grammar. But it might happen that the objects of one type are spread over several files. In order to collect objects from several files and to take into account the directory structure for modeling file contents, advanced concepts are required. As far as we perceived, the literature has not presented a solution to use this kind of semantics.

a) one file = one object type

We start with a simple extension of the previous discussion. There are isolated files, each one containing the objects of one type. Hence, a file represents one object type.

Files:

```
f1: o1 ... o1
f2: o2 ... o2
f3: o3 ... o3
```

Specification 5:

```
interface O1 from OT1 { ... };
interface O2 from OT2 { ... };
interface O3 from OT3 { ... };

file(f1,FILE1); FILE1: OT1[1-]; ...
file(f2,FILE2); FILE2: OT2[1-]; ...
file(f3,FILE3); FILE3: OT3[1-]; ...
```

The above lines present the principle of a specification. Each file fi obtains a starting symbol of its own, defined by the file clause. The objects of each type may certainly be complex (in the sense of Specification 4). Then OT*i* must represent the complex structure in the known manner.

b) one file = one object

In contrast, the objects of one type can be spread over several files. We assume that there are files in a common directory dir; each file should contain one complex object. The directory denotes de-facto an object type.

Files:

```
File f1: o1o2o3...o3o2o3...o3...
File f2: o1o2o3...o3o2o3...o3...
File f3: o1o2o3...o3o2o3...o3...
```

Specification 6:

```
interface O1 from OT1
{ attribute String filename = _NAME;
  ...
};

file(dir/*, DATA1);
DATA1: OT1 DATA2[1-];
DATA2: OT2 OT3[1-]; ...
```

Since the objects possess the same structure, one grammar can be used for handling several files. The file clause above determines to take all the files in dir into account. They are all parsed by the same grammar starting with non-terminal DATA1. The grammar extracts exactly one complex object of type O1 from each file, including subobjects of types O2 and O3.
The general form of the file-clause is file(<file>, ..., <file>, <non-terminal>). It determines the files to be analyzed by the start symbol <non-terminal>. Each <file> possesses the form <directoryspec>/<filespec> with <directoryspec> describing an absolute path /<directory>/<directory>/... . <filespec> and <directoryspec> can be scoped by usual Unix wildcards '*' (any sequence of symbols) and '?' (an arbitrary symbol) to denote several files. For example, file(/dir?/*/f*, DATA1) qualifies all the files that start with an "f" and are in any directory beneath /dir1, /dir2, ..., /dira, /dirb, ... This is a simple, but very effective mechanism.
Furthermore, there are predefined functions for accessing *meta-information* such as file and directory names. They can be used for assignments in the same way as non-terminals. In the above example, each object of type O1 receives the name fi of its file as value for the attribute filename; a predefined function _NAME computes the filename. This enables us to capture this important source of additional semantics. A function _DIR determines the directory of the current file, _PATH gives out the complete path as a string etc. These functions enable one to navigate in the directory structure. Naturally they can be appended, e.g., _DIR._NAME to compute the name of the directory of the current file.

c) one directory = one object type

This is an extension of b). It considers several directories dir*i*, each one containing several files. Each file f*ij* in directory di*ri* represents a (complex) object of type O*i*:

Files:

```
dir1 = {f11,...,f1m }
dir2 = {f21,...,f2n }
...
```

Specification 7:

```
file(/dir1/*, DATA1); ...
file(/dir2/*, DATA2); ...
...
```

The essential feature here again is the file clause. All the files in directory dir*i* are parsed by the same grammar with start symbol DATA*i*. DATA*i* defines the (complex) structure of type O*i*.

d) 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 several directories dir*k* that all contain three files f1, f2 and f3 each one related to a corresponding type O1, O2 and O3, respectively. Those directories dir*k* can represent a ternary relationship, the „cross product" between the objects in files: Any object in f1 is in relationship with any object in f2 and in f3.

Files:

```
dir1 = { f1, f2, f3 }
dir2 = { f1, f2, f3 }
...
```

Specification 8:

```
interface O1 from OT1
{ attribute attr1 = ...;
  attribute dirname = _DIR._NAME;
};
interface O2 from OT2
{ attribute attr1 = ...;
  attribute dirname = _DIR._NAME;
};
interface O3 from OT3
{ attribute attr1 = ...;
  attribute dirname = _DIR._NAME;
};

file(dir*/f1, DATA1);
DATA1: OT1[1-]; ...
file(dir*/f2, DATA2);
DATA2: OT2[1-]; ...
file(dir*/f3, DATA3);
DATA3: OT3[1-]; ...
```

Please note that the relationship is not directly expressed in the ODL schema, because it is difficult to describe the association between a relationship and its directory representation. Nevertheless, we use the directory name as an attribute dirname. Using dirname, we can join the tables to compute the relationship in SQL. Below we present some sample files and how they result in tables.

dir1:	**dir2**:
f1: records 2,3	f1: records 3,4
f2: records 1	f2: records 3,4,5
f3: records 1,2,4	f3: records 3

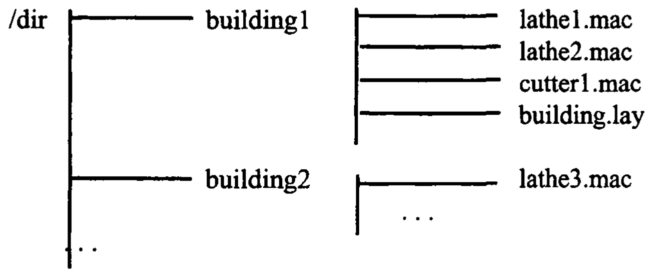
O1	**id**	**dirname**
	2	dir1
	3	dir1
	3	dir2
	4	dir2

O2	**id**	**dirname**
	2	dir1
	3	dir2
	4	dir2
	5	dir2

O3	**id**	**dirname**
	1	dir1
	2	dir1
	4	dir1
	3	dir2

Let us now consider a typical constellation with directories representing some hierarchical relationship.

Files:



Specification 9:

```
interface Machine from MACHINE
{  attribute int       mid        = _NAME._TAIL(1);
   attribute String    type       = _NAME._FRONT(1,_LEN-1);
   attribute String    building   = _DIR._NAME;
};
interface Layout from LAYOUT
{  attribute String    name       = ...;
   attribute String    building   = _DIR._NAME;
};
file(dir/building*/*.mac, MACHINE);
MACHINE: ...
file(dir*/building*/*.lay, LAYOUT);
LAYOUT: ...
```

There are several directories building*i*. Each of those directories defines a building with layout data (file extension .lay) and machines (.mac) to be placed in the building. The machines have a certain type (e.g., lathe or cutter) and a number. Both together form a file name.
The file clause determines that all files with an extension .mac (independent of their directory) should be analyzed by the same grammar with starting symbol MACHINE. Similarly, the files with extension .lay are handled by LAYOUT.
Specification 9 defines interface's for machines and layouts, essentially using the MACHINE and LAYOUT non-terminals, respectively. Since there are several machine files, each one results in a Machine object of its own. In order to extract the type of a machine, we take the file name by means of _NAME and decompose the name in two parts, type and number, by predefined functions. For both machines and layouts, we use the directory name of each file as an attribute building. This enables us to express the relationship by attribute building.
The object types are directly mapped onto tables Machine(mid int, type varchar, building varchar), Layout(name, ..., building varchar) with the same structure.

Hyperlinks in Files

A final example discusses the main principles of handling hyperlinks. We assume a file with several entries of the form

country <u>cities</u> eol

where any cities (underlined) is a link to another file containing the names of cities in this country (and maybe some other information). The following lines describe the HTML source:

Specification 10:

```
interface Country from COUNTRY
{ attribute String cname = COUNTRY;
  attribute Set<City> cities = CITIES;
};
 interface City from CITY
{ attribute name = CITY;
};
```

```
file(f.html, FILE);
FILE:   DATA[1-];
DATA:           COUNTRY " <A HREF=http:" _REF CITIES ">" LINK "</A>" _EOL;
COUNTRY:        _ID;
CITIES:         CITY[1-];
CITY:           _ID;
LINK:           _ID;     // name of the link in browser
```

Hyperlinks occur as <A HREF=http:address > in WWW pages. It is not enough to describe such a hyperlink in a grammar by means of "<A HREF=" CITIES "> ..." with a non-terminal CITIES defining the structure of the addressed file. The parser will not be able to recognize the CITIES part, because it is part of a different file referred to by http:address. The parser tries to map the address to CITIES. This fails!
We introduce a predefined keyword _REF to remedy this: _REF CITIES tells the parser to take CITIES as a string, the http address, during parsing. Later on, CITIES is used as a starting symbol for the addressed file. Then the rule CITIES: CITY[1-] is applied for this file.

SEMANTIC ENRICHMENT

Having migrated data from files in a relational database, the database can be plugged in a federation. In the framework FIHD of Hohenstein and Ebert (1999), this is done by generating an ODMG adapter that homogenizes the relational database. Homogenizing means to remodel tables in the ODMG object model. In the FIHD approach, we incorporate ideas developed in the context of *reverse engineering* and *semantic enrichment* (Chiang et al. (1994), Premerlani and Blaha (1994), Castellanos and Saltor (1991), Hohenstein and Körner (1995)). Hence, tables are not just converted to object types that possess the simple table structure, i.e., object types without relationships and without subtyping. Instead, the semantic level of the relational schema is upgraded: Implicit knowledge is expressed explicitly by using higher modeling concepts.
This is advantageous, since data integration is a complex and ambitious task. Providing an integrated access to DBSs requires a deep knowledge about the semantics of data so that inter-database semantic relationships can be detected. Any support that can be provided previously during homogenization is useful to let the integration concentrate on its real task.
The overall approach of homogenization with semantic enrichment resembles the one for our file handling. It again uses a generator, relying on a specification language called ODL_R (Hohenstein and Körner (1995)): The language allows specifying how to homogenize the relational database in the ODMG model. The advantage of our approach lies in the fact that semantic enrichment is explicitly – *precisely* – specified. This gives us the opportunity to remodel schemata in various ways, making any semantics explicit. The generator produces a corresponding ODMG OML/OQL wrapper to access the database. Since the ODMG adapter provides an object-oriented view of relational data including relationships and subtyping, the semantics is usable in form of high-level ODMG operations, manipulating objects and relationships instead of isolated tuples.
The concepts for semantic enrichment are very powerful. For instance, several attributes can be joined together to make an attribute of a predefined ODMG type or an object type nested into another. New object types may be introduced, e.g., to bring together common attributes in a *generalized* object type. It is possible to remodel typical forms of relational representations for relationships and subtypes. Particularly, several representations for subtype hierarchies can be handled. In Hohenstein and Körner (1995) we discuss a vertical, a horizontal and a complete materialization strategy as well as a flag approach. Each one has pros and cons in regard to fast access, redundancy, and easy update. These substantial strategies are orthogonal and can be applied in a mixed manner within one subtype hierarchy. For more details about the specification language, the reader is referred to Hohenstein and Körner (1995).
The following specification gives an example of enriching the tables obtained by Specification 9. Let us assume we want to see an additional object type Building, two subtypes OdmgCutter and OdmgLathe of Machine according to the type of machines, and some explicit relationships.

```
interface OdmgMachine
  from relation Machine[mid,type]
{ attribute int Id = Machine.mid;
  ...
}
interface OdmgCutter : OdmgMachine
  from relation Machine[type= "Cutter"]
  { }
interface OdmgLathe : OdmgMachine ...
  from relation Machine[type= "Lathe"] ...
interface OdmgBuilding
   from relation group(Machine[building])
{   attribute String Name = KEY;
    relationship Set<OdmgMachine> machines =  { Machine | building = KEY };
}
interface OdmgLayout
   from relation Layout[name]
{   attribute String Name = ...;
    relationship OdmgBuilding belongsTo = group { Machine | building = Layout.building } ;
}
```

In contrast to ODL$_{File}$, the full ODMG ODL is used here to express semantics. We again introduced some amendments to express connections between object-oriented and relational schemata. The clause from relation relates the specified
object types to tables. It is defined in what table the deep extent of a type (including objects of subtypes) is found. The easiest case is OdmgLayout from relation Layout[name]: An object type is built from one relation directly. name is the relational key of Layout. Keys are necessary to identify objects in the object-oriented runtime system: The key value is directly taken as object identifier. Each tuple, which is uniquely identified by its key value, refers to one object.
Tables without keys cannot correspond to object types. They can be used as bags nested in other types, nevertheless. Composite keys are possible and denoted as (mid,type), cf. OdmgMachine.
The objects of subtype OdmgCutter are also found in table Machine, which contains all the machines, cutters and lathes. However, cutters are marked in the table by a flag type which is used to separate cutters from other machines: [type="Cutter"]. Hence, subtyping by flag is made explicit.
Concerning buildings, the attribute building of Machine is „objectified". Building objects are built from table Machine. All the tuples in Machine are grouped by building, and each such group builds an object. The keyword group expresses such a grouping. The number of objects corresponds to the number of different building values. building is the key of a group and can be referred to by KEY.
Equations '=' occurring behind the attributes relate attributes and relationships of object types to relational attributes. The simplest form is int Id = Machine.mid and relates an object type attribute Id of OdmgMachine with domain int to a relational attribute mid of table Machine directly.
Relationships are specified similarly. OdmgBuilding belongsTo = group { Machine | building = Layout.building } expresses a reference to OdmgBuilding for interface OdmgLayout. Remember, buildings are part of table Machine, and the relationship is relationally represented in tables Machine and Layout by an attribute building. belongsTo is computed as follows: Take the set of tuples in Machine that have a building-value equal to the layout's building, and then group the set according to building (that is the key of OdmgBuilding).
Similarly, the relationship machines is expressed as all the tuples in Machine that possess a building equal to the key.

CONCLUSIONS

In this paper, we presented a specification-based approach to incorporate files in a federation approach based on the ODMG standard (Cattell and Barry (1997)). The procedure consists of two steps: Migrating data from several files and directories into a relational database, thereby giving the data a first structure. And then building an ODMG adapter on top of the database; the adapter can be plugged in a federation. The two steps together make intensive use of the ODMG object model, i.e., the federation layer can in total benefit from a real object-oriented view of file data.
Both steps are supported by specification languages and generators. A first language allows making semantics of files and directories explicit, i.e., giving files some structure. Such a specification is input to a generator that produces a file adapter. The adapter is responsible for parsing and unparsing

(modified data is stored back into files). Moreover, an adequate relational database is installed, the tables of which are automatically filled with parsed file data.
Providing an ODMG adapter for the relational database is done by a generator, too. Defining semantic enrichment in a specification language makes the semantics of relational tables 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 query interface. This gives the federation the opportunity to see relational data in an object-oriented way, to manipulate relational data in terms of C++ objects.
Either step is applicable for querying the WWW by now using SQL or ODMG/OQL, respectively.
The approach is embedded in a federation framework FIHD (Flexible Integration of Heterogeneous Data sources), which is described Hohenstein and Ebert (1999). Similar to TSIMMIS (Papakonstantinou et al. (1995)), Information Manifold (Kirk et al. (1995)), Efendi (Radeke (1995), OASIS (Roantree et al. (1999)) or SIGMA$_{FDB}$ (Höding et al. (1999)), FIHD has the goal to provide a global view of several, heterogeneous data sources, and to support an easy access to all the data without knowing the exact source and type of source. Relying again on a generative principle, our federation approach uses the ODMG adapters that homogenize databases and plugs them in a federation framework, which then gives a transparent ODMG access. Generators have been built for producing adapters for commercial systems such as Oracle, SQLServer and Informix, or object-oriented database systems such as Objectivity/DB and Versant.
Future work will be dedicated to a comfortable graphical support for defining specifications similar to Hohenstein and Körner (1995). Integrating file systems into the federation framework directly, avoiding the detour via relational databases, is another point. 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, HTML pages, or XML to visualize information obtained from the federation.

REFERENCES

Abiteboul, S., Cluet, S. & Milo, T. (1993) „Querying and Updating the File“, **Proc. Conf. on Very Large Databases** (VLDB) 1993

Abiteboul, S. Cluet, S. & Milo, T. (1995) „A Database Interface for File Update“, Proc. **ACM SIGMOD'95**, San Jose, CA USA 1995

Ashish, N. & Knoblock, C. (1997) „Wrapper Generation for Semi-structured Internet Sources", Proc. of **ACM SIGMOD** Workshop on Management of Semi-structured Data, Tucson (Arizona) 1997, Superseded by ACM SIGMOD Record 26(4), Dec. 1997

Buneman, P., Davidson, S., Fernandez, M. & Suciu, D. (1997) „Adding Structure to Unstructured Data“, Proc. of Int. Conf. on Database Theory **(ICDT97)**, Delphi 1997

Busse, R., Fankhauser, P. & Neuhold, E. (1994) „Federated Schemata in ODMG", Proc. of 2nd **East/West Database Workshop** 1994

CACM (1994) „Reverse Engineering", Special Issue of **Communications of the ACM** 37(5), 1994

Castellanos, M. & Saltor, F. (1991) „Semantic Enrichment of Database Schemas: An Object-Oriented Approach“, in Y. Kambayashi, M. Rusinkiewicz, A. Sheth (eds.): Proc. of 1st Int. Workshop on **Interoperability in Multidatabase Systems**, Kyoto (Japan), 1991

Cattell, R. & Barry, D. (1997) „The Object Database Standard: ODMG2.0", 2nd edition, **Morgan-Kaufmann Publishers**, San Mateo (CA) 1997

Chiang, R., Barron, T. & Storey, V. (1994) „Reverse Engineering of Relational Databases: Extraction of an EER model from a Relational Database", **Data&Knowledge Engineering** 12, 1994

Conrad, S., Eaglestone, B., Hasselbring, W., Roantree, M., Saltor, F., Schönhoff, M., Sträßler, M. & Vermeer, M. (1997) „Research Issues in Federated Database Systems", **SIGMOD RECORD** 12/1997, 26(4)

Conrad, S., Hasselbring, W., Heuer, A. & Saake, G. (1997) „Proc. of the Int. CAiSE97 Workshop Engineering Federated Database Systems **EFDBS'97**", Barcelona 1997

Conrad, S., Hasselbring, W., Saake, G. (1999) „Proc. of the 2nd Workshop **Engineering Federated Information Systems**“, Kühlungsborn (Germany) 1999

Ebert, A., Hohenstein, U. & Höding, M. (1999) „An Approach for Generating File Interfaces“, Proc. of Database Systems for Advanced Applications (**DASFAA**), Taipeh 1999

Hainault, J.-L., Tonneau, C., Joris, M. & Chandelon, M. (1993) „Schema Transformation Techniques for Database Reverse Engineering", 12th Int. Conf. on **Entity-Relationship Approach**, Karlsruhe 1993

Hammer, J., Garcia-Molina, H., Cho, J., Aranha, R. & Crespo, A. (1997) „Extracting Semistructured Information from the Web", Proc. of **ACM SIGMOD** Workshop on Management of Semi-structured Data, Tucson (Arizona) 1997, Superseded by ACM SIGMOD Record 26(4), Dec. 1997

Höding, M. (1996) „An Approach to Integration of File Based Systems into Database Federations", Proc. of 10th European Research Consortium for Informatics and Mathematics (**ERCIM'96**) on Heterogeneous Information Management, Prague 1996

Höding, M., Schwarz, K., Conrad S. et al. (1999) „SIGMA$_{FDB}$: Overview of the Magdeburg-Approach to Database Federations", in **Conrad et al.** (1999)

Huck, G., Fankhauser, P., Busse, R. & Klas, W. (1994) „IRO-DB: An Object-Oriented Approach towards Federated and Interoperable DBMS", Advances in Databases and Information Systems (**ADBIS'94**), Moscow 1994

Hohenstein, U. & Ebert, A. (1999) „A Comprehensive Toolkit for Building Database Federations", **Australian Journal of Information Systems**, Vol.7, No. 1, September 1999

Hohenstein, U. & Körner, C. (1995) "A Graphical Tool for Specifying Semantic Enrichment of Relational Databases", 6th IFIP WG 2.6 Work. Group on Data Semantics (**DS-6**) "Database Applications Semantics", Atlanta 1995

Hohenstein, U. & Pleßer, V. (1997) „A Generative Approach to Database Federation", in D. Embley, R. Goldstein (eds.): 16th Int. Conf. on Conceptual Modeling – **ER'97**, 1997, Los Angeles, Springer LNCS 1331

IMS (1993) „Proc. of Conf. on Research Issues in Data Engineering: Interoperability in Multidatabase Systems" (**RIDE-IMS'93**). Vienna 1993

Kaul, M., Drosten, K. & Neuhold, E. (1990) „ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views", Proc. Of 6th Int. Conf. on **Data Engineering**, Los Angeles 1990

Kambayashi, Y., Rusinkiewicz, M. & Sheth, A. (1991) (eds.) „Proc. of 1st Int. Workshop on **Interoperability in Multidatabase Systems**", Kyoto (Japan), 1991

Kirk, T., Levy, A., Sagiv, Y. & Srivastava, D. (1995) „The Information Manifold", Proc. of the **AAAI Spring Symposium Series**, March 1995

Kuno, H. & Rundensteiner, E. (1996) „The MultiView OODB View System: Design and Implementation", **Theory and Praxis of Object Systems** 2(3), 1996

Kushmerick, N., Weld, D. & Doorenbos, R. (1997) „Wrapper Induction for Information Extraction", Int. Joint Conf. on **Artificial Intelligence**, Nagoya (Japan) 1997

Markowitz, V. & Makowsky, J. (1990) „Identifying Extended ER Object Structures in Relational Schemas", **IEEE Trans. on Software Engineering** 16(8), 1990

Nestorov, S., Abiteboul, S. & Motwani, R. (1995) „Inferring Structure in Semistructured Data", Proc. of **ACM SIGMOD** Workshop on Management of Semi-structured Data, Tucson (Arizona) 1997, Superseded by ACM SIGMOD Record 26(4), Dec. 1997

Papakonstantinou, Y., Garcia-Molina, H. & Widom, J. (1995) „Object Exchange Across Heterogeneous Information Sources", Proc. of IEEE Conf. on **Data Engineering**, Taipeh 1995

Pitoura, E., Boukres, O. & Elmagarid, A. (1995) „Object-Orientation in Multidatabase Systems", **ACM Computing Surveys** 27(3), 1995

Radeke, E. (1995) „Efendi: Federated Database System of Cadlab", ACM SIGMOD Conf. on Management of Data 1995, **SIGMOD RECORD** 24(2)

Reddy, M., Prasad, B., Reddy, P. & Gupta, A. (1994) „A Methodology for Integration of Heterogeneous Databases", **IEEE Trans. on Knowledge and Data Engineering** 8(6), 1994

Premerlani, W. & Blaha, M. (1994) „An Approach for Reverse Engineering of Relational Databases", **Communications of the ACM** 37(5), May 1994

Roantree, M. & Murphy, J. (1997) „An Architecture for Federated Database Metadata", in **Conrad et al.** (1997)

Roantree, M., Murphy, J. & Hasselbring, W. (1999) „The OASIS Multidatabase Project", **ACM SIGMOD** Record 28(1), March 1999

Saltor, F., Castellanos, M. & Garcia-Solaco, M. (1992) „Overcoming Schematic Discrepancies in Interoperable Databases", in D.K. Hsia, E.J. Neuhold, R. Sacks-Davis (eds.): Proc. of the IFIP WG 2.6 Database Semantics Conf. (**DS-5**) on Interoperable Database Systems, Lorne (Australia), 1992

Sattler, K.-U. & Höding, M. (1999) „Adapter Generation for Extracting and Querying Data from Web Sources", Proc. of 2nd ACM SIGMOD workshop **WebDB'99**, 1999

Schmitt, I. & Saake, G. (1995) „Integrating of Inheritance Trees as Part of View Generation for Database Federations", 15th Int. Conf. on Conceptual Modeling (**ER'96**), Cottbus 1996, Springer LNCS 1157

Sheth, A. & Larson, J. (1990) „Federated DBSs for Managing Distributed, Heterogeneous and Autonomous Databases", **ACM Computing Surveys 1990**, 22(3)

Spaccapietra, S. & Parent, C. (1994) "View Integration: A Step Forward in Solving Structural Conflicts", **IEEE Transactions on Knowledge & Data Engineering** 6(2), 1994