A GENERATIVE APPROACH FOR BUILDING DATABASE FEDERATIONS

Uwe Hohenstein
Corporate Technology
Siemens AG, ZT SE 2
D-81730 München (GERMANY)
E-mail: Uwe.Hohenstein@mchp.siemens.de

ABSTRACT

A comprehensive, specification-based approach for building database federations is introduced that supports an integrated ODMG2.0 conforming access to heterogeneous data sources seamlessly done in C++.
The approach is centered around several generators. A first set of generators produce ODMG adapters for local sources in order to homogenize them. Each adapter represents an ODMG view and supports the ODMG manipulation and querying. The adapters can be plugged into a federation framework. Another generator produces an homogeneous and uniform view by putting an ODMG conforming federation layer on top of the adapters.
Input to these generators are schema specifications. Schemata are defined in corresponding specification languages. There are languages to homogenize relational and object-oriented databases, as well as ordinary file systems. Any specification defines an ODMG schema and relates it to an existing data source. An integration language is then used to integrate the schemata and to build system-spanning federated views thereupon.
The generative nature provides flexibility with respect to schema modification of component databases. Any time a schema changes, only the specification has to be adopted; new adapters are generated automatically.

INTRODUCTION

Federated database systems (DBSs) provide solutions to give a uniform and integrated access to data stored in several autonomous sources (Sheth and Larson (1990), Conrad et al. (1997)). 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. Users are not aware of the location of items in a particular system. With all that, the autonomy of the constituent systems is preserved. Each DBS still exists, and especially existing applications are not affected.
Research in the field of federated DBSs has brought up several prototypes (e.g., Rafii et al. (1991), Busse et al. (1994), Kuno and Rundensteiner (1996)) and results (Kambayashi et al. (1991), IMS (1993)) that tackle fundamental problems such as global transaction management and query processing. Several papers describe integration methodologies (Reddy et al. (1994) or Schmitt and Saake (1996)) and languages for view definitions (Kaul et al. (1990)). Despite the variety of approaches, some important points are often neglected.

1. Only few approaches such as Pegasus (Rafii et al. (1991)) incorporate *object-oriented systems*, and if they do, they use an own system instead of arbitrary commercial ones. IRO-DB from Busse et al. (1994) is one exception as it incorporates the commercial systems O_2 and Ontos. Indeed, plugging object-oriented systems in an – even object-oriented – federation framework is not as trivial as it seems to be. For instance, it is not clear how to handle databases that use object versions. There are no solutions discussed so far for those advanced concepts. Moreover, incorporating files is almost neglected in spite of the relevance of huge amounts of file data.

2. The adapters (e.g., in Huck et al. (1994), Radeke (1995)) for incorporating existing data sources must be *hand-coded* for any schema again and again. This is particularly a problem in case of schema evolution. Any change of a local schema requires a corresponding re-implementation of the adapter. In case of proprietary file systems, parsers must be implemented for each file (Abiteboul et al. (1993)); the parser has furthermore to be enhanced with semantic actions that build objects at the federation layer.

3. A lot of approaches such as Kaul et al. (1990) rely on querying only, or use own stand-alone manipulation languages. For example, Pegasus uses an extension of OSQL, the language of HP's object-oriented system, for manipulation. Data manipulation *embedded in a standard language such as C++* is neglected. There are two important points that cause trouble: First, federated schemata must be represented in the programming language. This is quite critical in C++, because its semantics is sometimes odd. Second, problems with view updates must be avoided. In IRO-DB (Busse et al. (1994)) federated views are defined by means of the query language OQL of ODMG (Cattell and Barry (1997)). It is extremely doubtful, whether updates on those views can be executed unambiguously.

Our research prototype FIHD Flexible Integration of Heterogeneous Database Systems) provides a comprehensive solution to these aspects. FIHD is a tightly coupled approach in the sense of Sheth and Larson

(1990). An open federation framework allows plugging in relational and object-oriented DBSs as well as files. The overall approach of FIHD is generative. Section 2 describes how generators produce adapters that "wrap" a data source and provide a uniform access according to the ODMG2.0 standard (Cattell and Barry (1997)) for object-oriented DBSs. Those adapters are implemented *automatically* on top of the component systems (cf. Point 2). There are generators to wrap relational and object-oriented databases as well as file systems (Point 1). A homogenization layer is achieved in this way.

All these generators require input, some mapping information that describes an ODMG schema and relates it to the local schema. Section 3 is concerned with specification languages that serve this purpose. We introduce three languages according to the kind of data source. A specification language for relational databases enables remodeling tables in the ODMG2.0 object model, thereby using object-oriented concepts intensively in order to express semantics as much as possible. This language incorporates ideas developed in the context of reverse engineering and semantic enrichment in the sense of Castellanos (1993), Hainault et al. (1993), Chiang et al. (1994), or Premerlani and Blaha (1994). Another language allows defining ODMG views on top of object-oriented DBSs, handling specific concepts that are not available in ODMG. A third language enables one to specify an ODMG view of files and to describe file contents in a grammar-like fashion.

Section 4 presents a specification language for defining federated schemata that yield system-spanning views. A corresponding generator creates federation layers giving a uniform ODMG access to the federation. Each layer supports an object manipulation and querying, seamlessly done in C++ (Point 3). This is especially important in view of manipulating federations in C++.

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

GENERATIVE APPROACH TO DATABASE FEDERATION

Our approach follows Sheth and Larson (1990) who define a reference architecture for federation frameworks. Two steps in their architecture are essential for us, homogenization and data integration.

Homogenization eliminates syntactic heterogeneity resulting from different data models and access interfaces. Each local schema, expressed in its native data model, is translated into a *canonical* data model. We rely on ODMG2.0 (Cattell and Barry (1997)) as canonical model because it defines a complete database interface consisting of an Object Model, an Object Definition Language (ODL), an Object Manipulation Language (OML), and an Object Query Language (OQL). Hence, each homogenized schemata offers manipulation and querying according to this standard for object-oriented DBSs.

Data integration then deals with integrating such homogenized schemata into *federated* ones. A federated schema provides a unified, consistent, and transparent view of all the integrated data. Particularly, semantic discrepancies between homogenized schema as described by Saltor et al. (1992) or Spaccapietra and Parent (1994) are eliminated. We again define federated schemata in the canonical model, in ODMG2.0.

From a functional point of view, the homogenization layer requires adapters that convert local data into the canonical model and map operations onto the local systems. Data integration has to support special operational aspects such as global transaction management and global query processing.

In our federation approach, homogenization and data integration are done in a *generative manner*, i.e., the homogenizing adapters and the federation layers are generated automatically by means of generators.

Each generator requires user input that defines a homogenized or a federated ODMG schema. To this end, we adopt the ODMG Object Definition Language (ODL) and extend it to capture explicit information according to the respective purpose. Languages ODL_x define a homogenization of relational databases ($x \equiv$ 'R'), files ($x \equiv$ 'File') and object-oriented systems O ($x \equiv$ 'O'). Any ODL_x specification defines how the specified schema is related to the underlying data source.

Data integration is supported by an integration language ODL_{Int}. ODL_{Int} provides means to merge homogenized schemata into federated ones defining an object-oriented and system-spanning view. In fact, the integration language helps dissolve semantic heterogeneity between schemata, i.e., resolves conflicts between homogenized schemata.

Figure 1 illustrates the process of generation. Given as input an ODL_x homogenization specification, corresponding generators GEN_x produce ODMG interfaces Interface_x automatically. Each interface provides an ODMG wrapper that maintains an ODMG view of the local data and implements OML operations on top of local systems. Other federation approaches generally require implementing a *hand-coded* adapter for each database to be incorporated in a federation.

A generator GEN_{Int} produces a corresponding interface Interface_Int to operate on the federation.

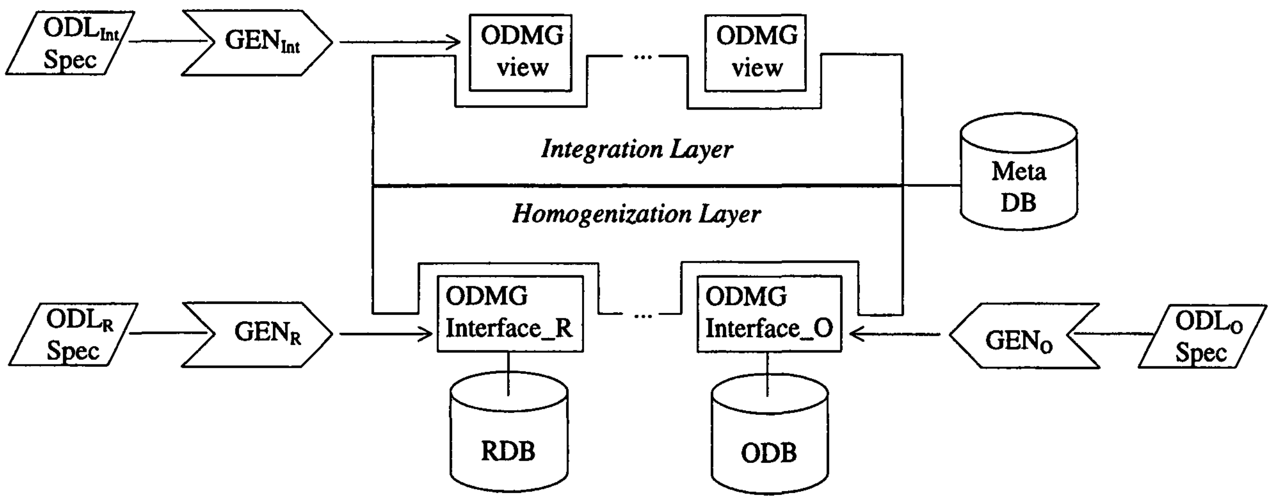


Figure 1: Generative Approach

Similar to Roantree and Murphy (1997), the generators use a meta database to maintain information about all the specifications. The meta database is pre-filled with data from the dictionaries of local DBSs, for example, what tables are available in a relational database and what types are in an object-oriented one. Parsing a specification, each generator fills the meta database with information about how homogenized schemata are related to local ones, and how federated schemata are composed of homogenized ones.

The real kernels of the generators read the meta information and generate the ODMG2.0 conforming interfaces. The outcome consists of C++ classes including methods for manipulation. Naturally, the implementations of interfaces for relational, object-oriented databases and file systems must call the original component source, while Interface_Int is implemented on top of all those homogenizing interfaces.

Each of the interfaces (Interface_R etc.) builds an adapter that already supports OML/OQL for the heterogeneous data sources. These adapters can be plugged together in order to access several databases in parallel with one common manipulation language. Using the homogenization layer, an application can open the databases and access them in a homogeneous ODMG way. However, the individual databases must be handled separately according to the respective disjoint schemata. There is no integrated view on all the data. It is the task of federated schemata to provide a system-spanning view with an integrated Interface_Int in addition.

HOMOGENIZATION

Homogenizing Relational Databases with Semantic Enrichment

The task of homogenizing relational databases is to convert relational tables into ODMG schemata and to provide an ODMG conforming access to tables. Relational DBSs, since having rather primitive modeling structures, do not carry much semantics. This is rather bad, since data integration is a complex task, as it requires a deep knowledge about the semantics of data. Wrong semantics could lead during integration to inadequate and defective federated schemata. In our approach homogenizing relational systems is thus combined with ideas developed in the context of *semantic enrichment* (Castellanos (1993), Hohenstein and Körner (1995)) and *reverse engineering* (Hainault et al. (1993), CACM (1994), Chiang et al. (1994), Premerlani and Blaha (1994)). Implicit knowledge is expressed explicitly by using object-oriented concepts extensively. The price we pay for comprehensive remodeling capabilities is an explicit specification. Indeed, the advantage of our approach lies in the fact that semantic enrichment is *precisely* specified. This gives us the opportunity to regain any implicitly given semantics and to remodel schemata in object-oriented terms in various ways. In automatic types of reverse engineering, as against, there is a danger of expressing wrong semantics, a fact we definitively want to avoid.

We adopt the Object Definition Language ODL of ODMG2.0 and extend it to capture explicit remodeling information. Our specification language ODL_R enables one to remodel relational tables in an object-oriented manner.

Figure 2 presents a relational schema representing a part hierarchy: It models atomic and complex parts, the latter may be composed of either parts. Table A contains atomic parts, and C complex ones. Since atomic and complex parts may be part of a complex part, there is a table P that consists of the keys of A and C and maintains common properties such as name and the build date. The hierarchy is expressed by a foreign key father in P that refers to the cid of the father part in C.

Semantic enrichment should produce the ODMG schema presented in UML. There are an object type PART and two subtypes ATOMIC and COMPLEX. The composition hierarchy of parts is explicitly modeled by means of a relationship Components/PartOf.

Figure 2 also presents a corresponding ODL$_R$ specification. Object types are defined by interface declarations, as usual in ODMG ODL. The clause from relation is an extension to ODL. It relates object types to tables. Type PART is built from table P directly. [pid] denotes the relational key of P. Each tuple, which is uniquely identified by its key value, refers to one object. Composite keys are possible and denoted as [pid,name,...]. COMPLEX : PART from relation C[cid = P.pid] specifies that subtype COMPLEX is basically found in table C. Moreover, each tuple in C is related to a tuple in P by means of equal id values. [cid=P.pid] is in fact a join condition that is necessary to access inherited P attributes.

An equation Long MId = M.mid connects an object type's attribute MId to a relational attribute mid of table M.

The build date of a part is stored in several relational attributes day, month and year, in tables A and C, since both contain parts. Date BuildDate = (A.day,A.month,A.year) + (C.day,C.month,C.year) combines those relational attributes to a predefined ODMG type and merges the values obtained fot the tables.

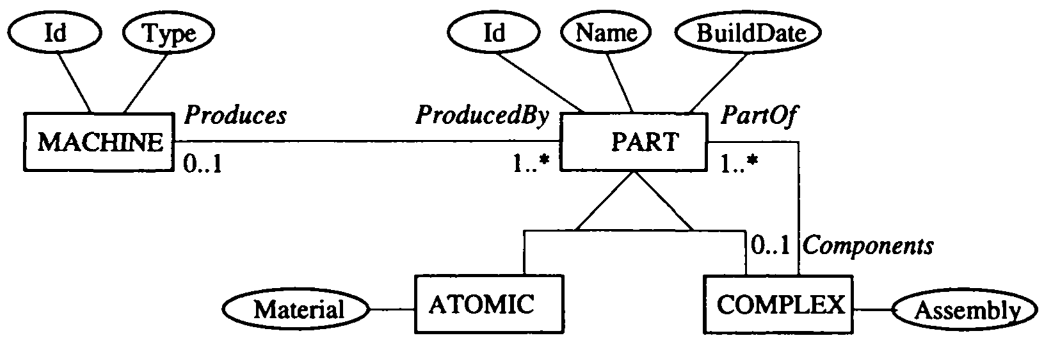
Tables:

A	pid	name	day	month	year	father	machine	material
	4	ABD	4	4	1994	2	20	Steel
	5	ABE	5	5	1995	2	20	Silicon
	6	ABF	6	6	1996	3	30	Steel
	7	ACG	7	7	1997	3	30	wood

C	pid	name	day	month	year	father	machine	assembly
	1	A	1	1	1999	NULL	10	Glue
	2	AB	2	2	1998	1	20	weld
	3	AC	3	3	1998	1	30	screw

M	mid	Type
	10	aaa
	20	bbb
	30	ccc

Object-oriented schema:



ODL_R specification:

```
interface PART from relation A[pid] + C[pid]
( extent parts  key id )
{  attribute Long Id = P.pid + C.pid;
   attribute String Name = A.name + C.name;
   attribute Date BuildDate = (A.day, A.month, A.year) + (C.day, C.month, C.year);
   relationship MACHINE ProducedBy inverse Produces  = (M I M.mid = A.machine + C.machine);
   relationship COMPLEX PartOf inverse Components = ( C I C.cid = A.father + C.father);
};

interface MACHINE from relation M[mid]
( extent machines  key MId )
{  attribute Long MId =  M.mid; ...
  relationship Set<PART> Produces inverse ProducedBy = { A+C I A.machine + C.machine = M.mid };
};

interface COMPLEX : PART from relation C[pid]
{  attribute String Assembly = C.assembly;
  relationship Set<PART> = Components inverse PartOf = { A+C I A.father+C.father = C.pid };
};

interface ATOMIC : PART from relation A[pid]
{  attribute Material = A.material;
};
```

Figure 2: Specification for Semantic Enrichment

The relationship Components is expressed by Set<PART> Components inverse PartOf = { A+C I A.father + C.father = C.cid }. The set of component parts consists of those tuples in A and C that have the complex parts' cid as value of father. The inverse relationship ProducedBy is analogously computed. Round brackets convert a tuple in C into a corresponding object of type COMPLEX, while curly brackets convert a set of tuples into a set of objects. Composite attributes are again possible to establish associations.
The specification language offers more concepts to express semantics. For instance, several relational representations for subtypes (vertical, a horizontal and a complete materialization strategy as well as a flag approach) can be rebuild. For more details about the specification language, the reader is referred to Hohenstein and Körner (1995) as well as Hohenstein (1996).

Homogenization of Object-Oriented Databases

In contrast to the one language ODL_R handling *all* relational DBSs, there are several languages ODL_O because each ODBMS possesses some peculiarities that must be taken into account. Each specification language ODL_O is naturally simpler than ODL_R since it is easy to express how ODMG types are built from existing object types. But every object-oriented DBS provides special modeling concepts such as versioning that are not available in ODMG. Even if commercial systems completely supported the ODMG standard, they would surely offer specific add-ons to beat competitors. Those advanced constructs cause trouble. There may be existing databases that use those concepts, e.g., databases that contain versioned objects. In particular, some concepts possess a specific semantics that must be maintained by ODMG operations at the homogenized level! We have to find ODMG ways to remodel peculiarities adequately without falsifying semantics.
In the following, we discuss some problems that occur when homogenizing the commercial object-oriented DBS Objectivity/DB. However, the critical points are similar for other systems. We present an adequate specification language ODL_{Objy} that reflects advanced concepts in ODMG2.0.

Storage hierarchy

Objectivity has a storage hierarchy that consists of federated databases [2], containers, and objects. A federated database contains several databases and gives them a common schema. The databases can be located on different sites in a network. Databases are partitioned into containers. Newly created objects are assigned to one of the containers. Containers are thus useful for clustering objects.

[2] Please note that the term "federated database" is here used in an Objectivity context.

Concerning homogenization, federated databases and databases do not give rise to problems. An Objectivity application program can open only one federated database in a process. Consequently, the names of a database and its associated federated database can be considered together as a database name from an ODMG point of view. Opening a database at the ODMG level requires then opening both the federated database and the database.
But containers cause some trouble. If we did not support containers, we would give up clustering, an important aspect of tuning object-oriented databases. A possible solution could be to define new types C_i in the ODMG schema, one for each container. Objects of type T in a certain container are put into a type TC_i that is subtype of both T and C_i. Accessing type T yields all objects, while type TC_i contains only those that are in container C_i. Unfortunately, the amount of containers is dynamic, as they can be created and deleted in Objectivity programs. Hence, the schema must be modified every time a new container is created and deleted. This is an inadequate solution!
We suggest establishing a predefined type d_Container as a subtype of ODMG type d_Database: Each container becomes an instance of d_Container. Containers can thus be created and deleted dynamically, as in Objectivity. Being objects of supertype d_Database, containers can be used instead of db when creating objects with new(db). Clustering becomes possible. Moreover, d_Container inherits the typical d_Database functionality such as open and close, now having the Objectivity specific container semantics. Certainly, we have slightly extended ODMG, as there is a new predefined type, but the real user-defined schema is not affected. Supporting clustering is worth it anyway.

Short relationships

Objectivity's object model offers so-called short relationships that reduce storage. Short relationships must refer to objects in the same container. If short relationships are not supported in the homogenized schema, establishing new relationships may violate this restriction regarding containers. Since Objectivity recognizes violations and issues an error message, it is not necessary to designate short relationships in an ODL_{Objy} specification.

Complex objects

Objectivity allows one to define propagation of operations copy, delete, and lock. This is done by means of so-called relationship specifiers: The operation is propagated to objects related by that relationship. We suggest no propagated copy and lock, because ODMG does not support any explicit copy and lock operations; this means no great loss of functionality. But otherwise we would extend OML.
If propagation of delete is not supported at the ODMG level, the contents of an Objectivity database may be corrupted: Propagation of delete may define an implicit integrity constraint; an object cannot exist without participating in a relationship. A propagated deletion can be achieved by changing the effect of ODMG delete_object. The ODL_{Objy} syntax should indicate propagation so that the implementation of delete takes propagation into account.

```
interface ARTICLE from object type ObjyArt
{ ...
  relationship ORDER OrderedBy = ObjyArt.orderedBy : propagate delete;
};
```

Versioning

In Objectivity, any object in the database may occur in several versions. Versionable classes need not be defined in the schema. Objects can dynamically become versionable by invoking a method setVersStatus(on/off). If versioning is enabled for an object, any modification (more precisely, any invocation of ooUpdate in order to explicitly notify an update, comparable to ODMG's mark_modified) creates a new version. A version graph maintains all versions of an object and possesses predefined relationships to navigate in the graph, i.e., going to previous or next versions. There exists a predefined *Genealogy* class the objects of which are surrogates for version graphs. Besides normal relationships pointing to a concrete version of an object, it is possible to establish relationships referring to genealogies, i.e., whole version graphs. Those relationships are *floating* in a certain sense. If the target is a genealogy, the current version within the graph can be accessed by means of predefined methods.
We cannot omit the concept of versioning if there are existing databases containing versioned objects and floating relationships. We propose a predefined class d_VersionGraph. This class supports special methods to

handle versions, e.g., to turn versioning on and off, to designate a referenced version in a version graph as current one etc. Any versionable type must inherit from d_VersionGraph. Objects then inherit versioning methods. Furthermore, it is possible to mark relationships as floating. Finally, the semantics of mark_modified is altered to create new versions if versioning is switched on. Since all these points affect the "semantic enrichment" of object-oriented database systems, ODL_{Objy} must reflect them in the following way:

```
interface ARTICLE : d_VersionGraph from object type ObjyArt
{ ... };

interface ORDER from object type ObjyOrder
{ ...
  relationship Set<ARTICLE> Orders = ObjyOrder.orders floating;
};
```

Please note the whole functionality is concentrated in one additional predefined class. Neither the schema nor OML is really extended.

Object manipulation

Homogenizing implies emulating ODMG OML on top of Objectivity. It is no problem to bridge the differences in both manipulation languages. But again, there are advanced Objectivity manipulation concepts that exceed ODMG2.0. Omitting those manipulation features causes a lack of Objectivity functionality. Anyway, this will not exclude existing databases from a federation.
We propose in general omitting those functions that do not endanger usability, e.g., copy propagation.
Sometimes simulating operations is easily possible by giving ODMG operations an extended effect (propagational delete with delete_object, creating new versions with mark_modified).
Keeping advanced functionality can also be done by adding new classes. The extensions are then encapsulated in classes, staying outside the original OML. This principle was applied previously to support versioning and containers, e.g., classes d_VersionGraph and d_Container. Similarly, we introduce a subclass d_LongTransaction of d_Transaction to support Objectivity's checkIn/checkOut functions.
We strictly avoid introducing new OML methods. Hence, we do not support scoped queries in the sense that searching objects can be done in a particular container, in one database, or in the whole federation. This would require a new type of query method in ODMG.

Homogenization of File Systems

The 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 and an enhancement of Yacc. ODL is used to describe the schema of the data file in an object-oriented way. The Yacc-part defines a grammar. This 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.
Figure 3 presents the three parts of an ODL_{File} specification. We assume a data file containing machines and the parts they produce. Lines starting with "100" denote machines, those starting with "200" mark parts. Please note the sequence of the records within the data file is relevant; it defines an implicit relationship between the objects: All parts following a machine record are produced by that machine.

File content:

```
100,Caster,1
200,AB,4
200,ABC,5
200,ABD,6
100,Press,2
200,ACG,7
```

// Part 1 of ODL$_{File}$ specification: ODL schema

```
interface PRODUCT
{
  attribute short No;
  attribute String Name;
};

interface MACHINE
{
  order by Name asc;
  attribute short No;
  attribute String Name;
  attribute Set<PRODUCT> Partlist;
};
```

// Part 2 of ODL$_{File}$ specification: Grammar

```
startsymbol(File);                      // start
File: Data[1-];                         // rules of the grammar
Data: Machine Part[1-];
Machine: "100" _COMMA  MName _COMMA MNum _EOL;
Part:    "200" _COMMA  PName _COMMA PNum _EOL;
PName: _IDENTIFIER;      // nonterminals using
PNum:  _NUMBER;                         // predefined terminal
MName: _IDENTIFIER;
MNum:  _NUMBER;
```

// Part 3 of ODL$_{File}$ specification: Assignment Rules

```
Machine class2use(MACHINE);
MName  export(Name,ET_ASCII);
MNum   export(No,ET_ASCII) condition(short,ET_ASCII,LT,10);
Data   export(Partlist);
Part class2use(PRODUCT);
PName  export(Name,ET_ASCII);
PNum   export(No,ET_ASCII);
```

Figure 3: ODL$_{File}$ Specification

Interface Definition with ODL Subset

The first part of an ODL$_{File}$ specification defines an ODL-schema. The file content is here modeled as two object types MACHINE and PRODUCT. Nested structures can be expressed: Each machine produces a Partlist, a set of PRODUCTs. It is important to express those nested structures because they occur quite often in files as shown by Ashish and Knoblock (1997).

In addition to ODMG ODL, it can be specified that an object type must be ordered in a certain way. Here, machines occur in the file with ascending names. It is essential to express this, because otherwise the adapter has no chance to write back the file correctly in this order! In other words, the specification would be incomplete.

Definition of the File Grammar

The second part of the specification consists of a grammar for the data file. The grammar rules describe, how the data file is structured. Figure 3 also shows the grammar part for our example. The syntax of rules is similar to the rules of the compiler-compiler Yacc.

```
nonterminal: item1 item2 | item3 item4 ...;
```

is a rule that defines a nonterminal symbols by means of items. Alternatives in rules are denoted as '|' similar to Yacc. Each item can be either a nonterminal that must be defined by other syntax rules, or a terminal symbol. Terminal symbols can be defined in two different ways:

- Embedding the characters in quotation marks such as "100".
- Usage of predefined terminal symbols.

Terminal symbols represent the characters in the data file. There are several predefined terminal symbols. For example _NUMBER stands for an optionally signed integer number. Similarly, _IDENTIFIER represents a sequence of letters, _BYTE any byte character, _COMMA a comma, _1 the digit "1", and so on. The predefined terminal symbols are provided because they ensure shorter specifications and thus improve readability. Furthermore, fixed-sized attributes can be described easily. This is necessary to effectively describe files with fixed-sized records.
The specification language offers the possibility to use repetition groups to bypass recursions. This clearly improves the readability and the maintainability of a specification and also has advantages for unparsing a file. For example, FILE : DATA[1-] specifies that DATA can occur more than once.
In some cases, a data file contains a certain set of tokens which can occur in any order. It is cumbersome to describe this variable order by only using alternatives, as it has to be done with Yacc. That is why $\mathrm{ODL}_{\mathrm{File}}$ provides an operator &:

```
Article: Author & Title & Journal & Year;
```

specifies that an article consists of an author, a title, a journal and a year, whereby these components may occur in any order within one record of the data file.

Assignment Rules

The grammar only describes the structure of the data file. A parser can then collect the information from the file. Assignment rules now relate the parsed information to objects in the ODL schema.
A nonterminal always has a certain value. This value is a string, which arises as a concatenation of the values of the nonterminals on the right side of the rule: The value of Machine is composed as follows: "100" is a terminal symbol which naturally has the value "100". _COMMA is a predefined terminal symbol with the value ",". MName and MNum are other nonterminals which are later assigned the values "Caster" and "1" after analyzing the first machine record. Hence, Machine has the value "100,Caster,1". The values of the nonterminals are assigned to the attributes of the objects in the following way.
A nonterminal is connected with an object type by the keyword class2use. For example, Machine class2use(MACHINE) specifies that the nonterminal Machine is related to the object type MACHINE. That is, the value of the nonterminal is used to build MACHINE-objects.
An export rule fills an attribute of the current object type with the value of a nonterminal. Thereby, it has to be specified how the data in the file is encoded. It is a difference, whether an integer number is encoded in a binary format or as an ASCII-number. $\mathrm{ODL}_{\mathrm{File}}$ offers a 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. PNum export (No, ET_ASCII) then specifies that the parsed PNum-value should be ASCII-decoded before assigning to the No attribute of the current PRODUCT-object. Similarly, the set-valued attribute Partlist is filled by using the DATA rule that describes the nesting.
Condition rules contain simple comparisons in order to be able to filter certain records. In Figure 3, machine records are only exported if the value of MNum is lower than 10. A condition rule always effects a whole object type. The current record is only exported, if all condition rules affecting the appropriate object evaluate to "true". Condition rules need a data type, since the comparison value need not be part of the object type.

Condition rules are very useful because some file formats use a flag to mark a record as deleted in order to increase performance. Using the condition rules, a specification can be built which only exports undeleted records to the DBS.
Such an ODL_{File} specification is input to a generator. This generator produces a Yacc program that is able to parse the data file. Moreover, the Yacc-program contains semantic actions that build objects according to the ODL schema. This releases one from the tremendous task of building Yacc programs as it is necessary in Abiteboul et al. (1993). The approach is similar to Ashish and Knoblock (1997), but more general.
At the moment, the file adapter is not directly embedded in the federation framework; the adapter stores file data into a relational database it has previously installed. But the relational database can be included into the federation as described in Subsection 3.1. An unparser is generated that writes data from the database back to files.

DATA INTEGRATION

A flurry of activities developed languages to specify federated views for heterogeneous DBSs. However, most proposals rely on relational and functional canonical models. Others like Radeke (1995) do rely on object-orientation, but present only simple examples, not showing how to mix subtype hierarchies, as discussed in Schmitt and Saake (1996) from a methodological point of view. Some approaches seem to be too powerful with regard to view updates. For example, Busse et al. (1994) use ODMG OQL queries to describe federated schemata, a natural and flexible mechanism. Although they claim in Huck et al. (1994) to support ODMG OML, it is not clear how they map updates onto local databases.
As far as we perceived, updates are a general problem of federation approaches. Queries are often preferred. What is neglected, is how to represent federated views in a language such as C++, and how to embed object manipulation.
These points led us designing a good compromise that is powerful enough, but is also able to support updates invoked from C++. ODMG2.0 helps us since it defines a C++ binding for the OML. What we still have to do is to care for a C++ representation of federated views.
The specification language ODL_{Int} we propose, provides syntactic constructs to handle typical problems (Reddy et al. (1994)) and aspects of schema integration such as:

- Naming conflicts such as homonyms/synonyms.
- Type conflicts of attributes.
- Scaling conflicts (Dollar vs. DM).
- Structural discrepancies (Spaccapietra and Parent (1994)), e.g., if some unit is modeled by an attribute in one component schema, but as an object type elsewhere.
- Interdatabase connections, i.e., to specify logical links between so far disjoint databases, i.e., objects are built by "joining" objects from different databases.
- Objectification of values and relationships (Busse et al. (1994)).
- Generalization to bring together objects of same type, but from different, heterogeneous databases, disjoint or overlapping.
- Relating types of different databases in a subtype hierarchy, in general merging subtype hierarchies etc. (Schmitt and Saake 1996).

The language follows the way we pursue for homogenization. Given an ODL_{Int} specification as input, object classes, which represent the federated schema and provide OML, are automatically generated.
In the following, we stress important aspects that are mainly centered around subtyping. We feel subtyping has not been investigated enough, especially because C++ possesses an odd semantics in this respect, impeding a class representation.
Let us take the homogenized schemata defined in Figures 2 (schema S1) and 3 (schema S2). We first want to "join" PARTs and PRODUCTs. We suppose that only complex parts are produced. Hence, the semantics for integration is as follows: COMPLEX = PRODUCT. We define an interface JOINED_PART that generalizes COMPLEX and PRODUCT.

```
interface PART from PART@S1
{
  Id = PART@S1.Id; ...
};

interface JOINED_PART : PART from PRODUCT@S2[No] = COMPLEX@S1[Id]
{
  attribute Name = COMPLEX@S1.Name = PRODUCT@S2.Name;
  attribute Assembly = COMPLEX@S1.Assembly;
  relationship MACHINE ProducedBy inverse Produces
                              = ( MACHINE@S2 | PRODUCT in MACHINE@S2.Partlist );
};
```

Similar to the ODL$_x$ homogenization languages, the basic bricks are existing types (related to homogenized schemata by means of '@'), means to identify objects ('[ ]'), and set operators '+' (disjoint union), '$\cup$', and others. The from clause defines how the (virtual) extent is built.
At first, type PART is directly taken from schema S1. Objects of type JOINED_PART are essentially found in COMPLEX, however, information is also needed from PRODUCT. This is particularly necessary to retain the attribute Assembly from COMPLEX and the relationship ProducedBy from PRODUCT: An object of type JOINED_PART should possess both. The clause ... from PRODUCT@S2[No] = COMPLEX@S1[Id] indicates this fact and demands for an equality of extents. [No] and [Id] are used to relate objects in both types. This is indeed some kind of instance integration that is absolutely necessary, but almost forgotten by other approaches. If the keys are not homogeneous with regard to data type and value, functions can be defined to relate the objects.
Since objects of PRODUCT and COMPLEX are "joined", common attributes must be handled. The names of products and complex parts should be the same in both schemata. Hence, we require an equation COMPLEX@S1.Name = PRODUCT@S2.Name. If value conflicts between names were allowed, we could omit the part = PRODUCT@S2.Name, just saying to take the Name of COMPLEX. If the names were homonyms, we could specify

```
Name1 = PRODUCT@S2.Name
Name2 = COMPLEX@S1.Name
```

The relationship Produces is computed by using the set-valued attribute Partlist. The embedded Partlist of MACHINE is converted to an explicit relationship.
A second example shows a generalization of types: We now suppose PART and PRODUCT be disjoint, and we want to generalize them to one type PART_SUM the extent of which should receive all the objects:

```
interface PART_SUM  from PART@S1[Id] + PRODUCT@S2[No]
    (extent sum)
{
  attribute No = PART@S1.Id + PRODUCT@S2.No;
  ...
}
```

Please note keys [...] are not necessary here. If they are omitted, PART and PRODUCT will be assumed to be disjoint due to '+', but there is no mean to control it then. Key specifiers [Id] and [No] allow the federation layer to check for disjoint sets of numbers.
Let us now suppose PART and PRODUCT overlapping: There are parts that possess the same Id and No, stored in both databases. The following interface definition is sufficient at a first glance:

```
interface PART_UNION  from  PART@S1[Id] ∪ PRODUCT@S2[No] ... ;
interface PART_S1      : PART_UNION  from  PART@S1 ...
interface PRODUCT_S2 : PART_UNION  from  PRODUCT@S2 ...
```

'$\cup$' now allows non-disjoint unions, in contrast to '+'. [Id] and [No] *must* specify how to relate identical parts in PART and PRODUCT. PART_UNION allows handling all parts uniformly, independently of their location. interfaces can be added in order to access not only all parts, but also the parts in S1 (as PART_S1), and the parts in S2 (as PRODUCT_S2), too.

But this specification is only partly correct, owing to the odd semantics of C++, and ODMG2.0, too: Subtypes are always disjoint in C++ with regard to real instances. Even if PART and PRODUCT contain common objects, the federated view is not aware of them. There is no means in OML, no object type, to insert one instance in both PART_S1 and PRODUCT_S2. This problem can be solved by introducing an additional subtype PART_INTERSECTION of PART_S1 and PRODUCT_S2:

```
interface PART_INTERSECTION : PART_S1, PRODUCT_S2 from PART@S1[Id], PRODUCT @S2[No] ...
```

PART_INTERSECTION is now the type to insert common objects; it has the purpose to hold the intersection of PART and PRODUCT. Hence an identification [ ] of objects in both types is demanded. Considering the shallow extents, PART_S1 contains the real parts (that are not products), PRODUCT_S2 the real products, and PART_INTERSECTION all the parts that are also products. The shallow extent of PART_UNION is empty as there no instances that are neither parts nor products; nevertheless, the deep extent contains all the parts. Other approaches such as Busse et al. (1994) and Radeke (1995) that also rely on ODMG2.0 do not discuss this important fact! Only approaches that use their own manipulation language, e.g., Kaul et al. (1990), can make things easy to handle disjoint and overlapping generalizations.
Mixing several subtype hierarchies is another problematic case that requires special concepts. The literature contains methodologies how to mix hierarchies, for example Schmitt and Saake (1996). But less effort has been spent on languages to define mixed subtype hierarchies and their relationships to the original ones.
Let us assume two databases: A university library (UN1) contains a type PUBLication with a subtype JOURNAL. A computer science library (CS2) has a type PUBL possessing a subtype PROCeedings. The following conditions should hold with regard to extents:

$$PUBL@UN1 \supseteq PUBL@CS2 \supseteq JOURNAL@UN1,$$
$$PUBL@UN1 \supseteq PUBL@CS2 \supseteq PROC@CS2,$$
$$JOURNAL@UN1 \cap PROC@CS2 \neq \emptyset$$

The following specification defines an integrated subtype hierarchy that covers these conditions.

```
interface UNI_PUBL from PUBL@UN1 ...
interface CS_PUBL  : UNI_PUBL from  PUBL@CS2 [isbn=PUBL@UN1.isbn] ...
interface JOURNAL  : CS_PUBL  from  JOURNAL@UN1 [isbn=PUBL@CS2.isbn] ...
interface PROC     : CS_PUBL  from  PROC@CS2 ...
interface J_P : JOURNAL,PROC from  JOURNAL@UN1[isbn],PROC@CS2[isbn] ...
```

Types of different schemata can be put in a subtype relationship by "join" conditions such as [isbn=PUBL@UN1.isbn]. This is necessary to access inherited attributes.
The definition of federated schemata must obviously be done in accordance with the extents of the existing types. In principle, we can derive different federated schemata from one and the same set of subtype hierarchies. Methodologies such as Schmitt and Saake (1996) and Spaccapietra and Parent (1994) help finding the right semantics.
To sum up, ODL_{Int} has the power to handle complex situations. Anyway, the semantics is compatible with C++ so that manipulations become possible. The above examples can certainly give only an impression of the power and flexibility of our approach.

CONCLUSIONS

In this paper, we suggested a comprehensive approach to database federation. Building federations is done in a completely generative manner: Generators automatically implement access layers to a set of heterogeneous data sources and provide integrated views and uniform access conforming to the ODMG standard (Cattell and Barry (1997)).
The approach is based on a set of specification languages that build the input to code generators. A first set of languages is concerned with homogenization. Each object-oriented database system acquires a language of its own due to the diversity of systems and concepts. Relational systems are handled by one single language. So are files. The languages allow making semantics explicit. An additional language allows one to merge the homogenized schemata to federated ones. A federated schema then defines a system-spanning, integrated and consistent view of all the databases.
Giving specifications in those languages as input, corresponding generators produce code to provide the schemata with an ODMG conforming object manipulation. Defining a homogenization specification, a generator produces a schema-dependent adapter that automatically maps ODMG2.0 operations and queries onto

the data source. Each adapter can be used stand-alone. For instance, the adapter for a relational database provides an object-oriented access to relational data. A set of generated ODMG adapters can be used to access several, heterogeneous databases in parallel, without implementing any glue! A programmer just needs to compile and link the homogenized schemata into an application program in order to operate in an ODMG manner on the data sources. Plugging adapters in a global federation framework provides full functionality. There is now a database-spanning view, and global querying and transaction management.

The generative nature of database federation has a significant advantage with regard to schema evolution: Any time a local schema is modified, the corresponding generator produces a new adapter for the local database without any implementation effort. Just the specification has to be adapted.

The presented generative approach has partly been implemented. At the moment, there are generators for a homogenization of files, any relational database system and the commercial object-oriented databases Objectivity/DB and Versant. The specification language for defining federated schemata and the corresponding generator for producing federation layers is still under development.

Future work will be dedicated to some improvements. Homogenizing file systems should take into account directory information as discussed in Höding (1996). Furthermore, file systems must be integrated into the federation framework directly, avoiding the detour via relational databases. Finally, we feel the need for a comfortable graphical support for defining schemata similar to Hohenstein and Körner (1995). Anyway, some more support is necessary for guiding users in the integration process.

REFERENCES

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

Ashish, N. & Knoblock, C. (1997) "Wrapper Generation for Semi-structured Internet Sources", **ACM SIGMOD** Workshop on Management of Semi-structured Data, Tucson (Arizona) 1997, Superseded by ACM SIGMOD Record 26(4), Dec. 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. (1993) "Semantic Enrichment of Interoperable Databases", in **IMS** (1993)

Cattell, R. & Barry, D. (1997) (eds.) "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

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

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

Hohenstein, U. (1996) "Bridging the Gap Between C++ and Relational Databases", Proc. of 10th European Conf. on Object-Oriented Programming (**ECOOP'96**), Linz (Austria) 1996, Springer LNCS 1098

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

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

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

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

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

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

Markowitz, V. & Makowsky, J. (1990) "Identifying Extended ER Object Structures in Relational Schemas",

IEEE Transactions on Software Engineering 16(8), 1990

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

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

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

Rafii, R., Ahmed, R., DeSmedt, P., Kent, B., Ketabchi, M. & Litwin, W. (1991) "Multidatabase Management in Pegasus", Kambayashi et al. (1991)

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

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

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

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** 1994, 6(2)