UDC: 004.42
Original scientific paper

DATA SOURCE WRAPPERS BASED ON DEFINITE CLAUSE GRAMMARS

Alen Lovrenčić
University of Zagreb, Faculty of Organization and Informatics, Varaždin, Croatia
E-mail: alovrenc@foi.hr

The problem of the integration of heterogeneous data sources can be divided into two separated subproblems. The first one is the problem of solving semantic inconsistencies and conflicts between independent sources. This problem is dealt with in [4],[5],[6] and [12]. In this paper we shall deal with the second subproblem – the problem of translation among different query languages that data sources may use. Grammar templates will be created for some of them, using DC grammars. We will show that DC grammars are good enough to represent all the properties of query languages that are syntactically of the first or a higher order.

Keywords: heterogeneous sources, integration, wrapper, query processing, grammars.

1. INTRODUCTION

In this paper, we will deal with one of the problems with the integration of heterogeneous data sources. The basic terms for this field are defined in [12],[13] and [4]. A system for heterogeneous data sources integration is made up of two basic parts. One part – the *mediator* is described in detail in [4],[5],[6] and [12]. The other part, well actually – parts, are known as *wrappers*. The wrapper is an interface system whose role is to translate queries from the language that is used by the mediator into languages that are used by underlying sources, and to translate answers from the form in which the underlying sources represent into the form the mediator can understand.

We will show that definite clause grammars (DCG) are a good framework for building language templates. In this paper, we are interested in specific subclass of programming languages – the class of query languages. This is because our aim is to build a *wrapper*, i.e. a query language converter.

The problem that we are going to address in this paper is that of language translation. We will not examine the other problems of structural heterogeneity (such as type mismatches, formats, codes etc.), and semantic heterogeneity (such as synonyms, homonyms, etc.), addressed in [1] here, and we are not going to address the problems of inconsistency of sources, addressed in [4],[5],[6] and [12]. The problem we will examine and try to solve is dealt with in [7],[8] and [9]. But in these papers a language called QDTL is proposed for the query language description. The QDTL is a descriptive language that describes the syntax of some logic-based language. However, in this paper we will use grammars for the syntax description.

We will use DCG as the grammar for the language syntax representation. DCG is, in fact, a context-free grammar (CFG) that is implemented in Prolog-like languages. As we well know, CFGs define context-free languages (CFL), which can be recognised by pushdown automata. Readers who are interested in the types of grammars, languages and automata that represent them can refer to [3]. CFLs are closely connected to Turing machines, the famous mathematical model for an algorithm. So, it is not surprising that DCGs are good enough to represent query languages, which are functional languages, and do that not contain any problematic constructs, such as *if...then...else* structures, *goto* command, or loops etc.

The language that will be the host language for the DCG will be HiLog ([2]). HiLog is a Prolog-like language of higher-order syntax and first-order semantics. This higher-order syntax makes HiLog more appropriate for meta-programming, so it will be easier to implement DCG query templates and its semantics. The second reason we choose HiLog is that there is a HiLog implementation developed by SUNY at Stony Brook called XSB ([11], [14]), and, what is also important is that this implementation has tabled, SLG resolution, which is safe for partially stratified programs. So HiLog with SLG resolution, as the procedural semantics, has great capabilities.

2. THE HETEROGENEOUS DATA SOURCE SYSTEM

Firstly, we have to define what a system that integrates heterogeneous data sources actually looks like. As was said before, this system has two fundamental parts. The first part is the *mediator* or the *supervisor*. The mediator is a knowledge base that contains rules that describes ways of using data sources. The mediators role is to resolve conflicts and inconsistencies between underlying data sources, as they arise. In other words, the mediator is the system that acts as the interface between the user and the underlying sources as an interface. It allows the users to see the whole system with all the underlying data sources, as a single, consistent data source.

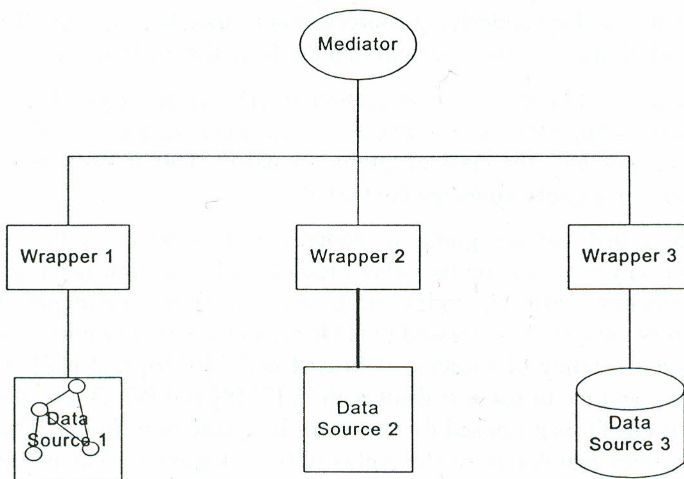

Figure 1: The Heterogeneous Data Source System

In Figure 1 the simple heterogeneous data source system is presented. It contains one mediator for the query language that the users use to access all the underlying data sources and it also contains the wrappers that are the interfaces between the mediator and underlying data sources.

Because of the difference between the syntax, the semantics and the power of query languages for the underlying data sources, wrappers are needed to translate the mediator's query from the language used on the mediator to the query languages of the underlying data sources, and to translate answers given by the underlying source into the language of the mediator.

But, a wrapper is not only a translator. Its other powerful function is to empower underlying data sources by extending their query capabilities. This is possible because a wrapper can have some selection, projection and joining capabilities that can be performed on data that is given as an answer from the underlying data source. The wrapper's role is to equalise the query capabilities of the underlying data sources with the query capabilities of the mediator query language. In this way, the mediator is obligated to distinguish between the differences in the query capabilities of underlying data sources.

We will describe now what the data sources in such heterogeneous data source system could actually be. It is obvious that any type of a database could be a data source. In [4], an example is given in which a relational database and a deductive object-oriented database are data sources. Legacy sources can be data sources in the heterogeneous data source system, too. That means that any for example say a UNIX file with UNIX command *grep*, can play a role of the data source. Computer programs that take input and give an output could be data sources for this system, too. Web pages could also be data sources. So, various data sources can be integrate into the system. It is important to say that the designer of the heterogeneous data source system is not necessarily the designer of the underlying sources. We will think of data sources as independent databases, that we are generally not able to administrate. This means that we do not necessarily have any other grants on the underlying data sources, except read-only access. So, we are not suppose to change the data sources schema and we are not able to accommodate data sources with our system. We should use them as they are, read data from them, and make the best of the answers they give.

More complex heterogeneous data sources integration systems are possible, than just the system that is shown in Figure 1. Other heterogeneous data source integration systems can be used as data sources for a bigger system of this type. Figure 2 shows another, more complex heterogeneous data source system.

In the system shown in Figure 2, *Mediator 1* the mediator of the main system, and *Mediator 2* is mediator of heterogeneous data source system and it serves as the data source for the main system. Regardless of the form of *Mediator 2*, (which can be the same as the form of *Mediator 1*), the system to which *Mediator 2* is the supervisor, has a wrapper. This is because, regardless of similarity with the mediator of the main system, the syntax of these two systems are not the same, and the translation between these two syntax has to be done.

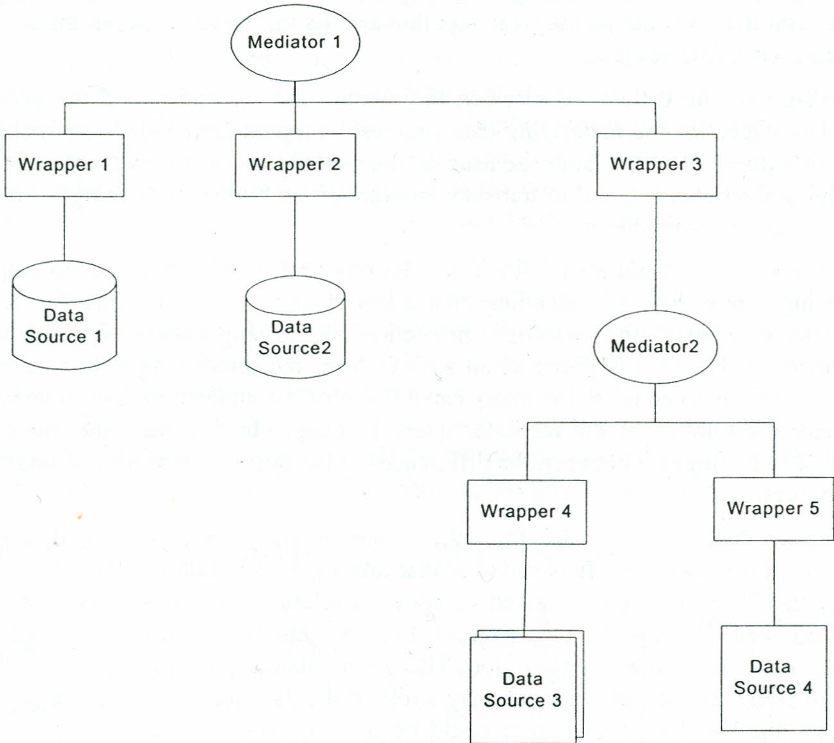

Figure 2: A Complex Heterogeneous Data Source System

3. WRAPPER CONSTRUCTION

As was shown in the figures from the previous section, every data source in the heterogeneous data source integration system has its own wrapper. A heterogeneous data source integration system can have many underlying sources and it is necessary to build a wrapper for each one of them. So, a lot of programming could be necessary in order to build wrappers for all the data sources. Because of this, a theory for wrapper building is necessary. The main goal of this theory is to define, as much as possible, things that are common to all the wrappers, and to build those base things together for all the wrappers in the system.

The Following figure shows main components of the wrapper and the data that flows through it.

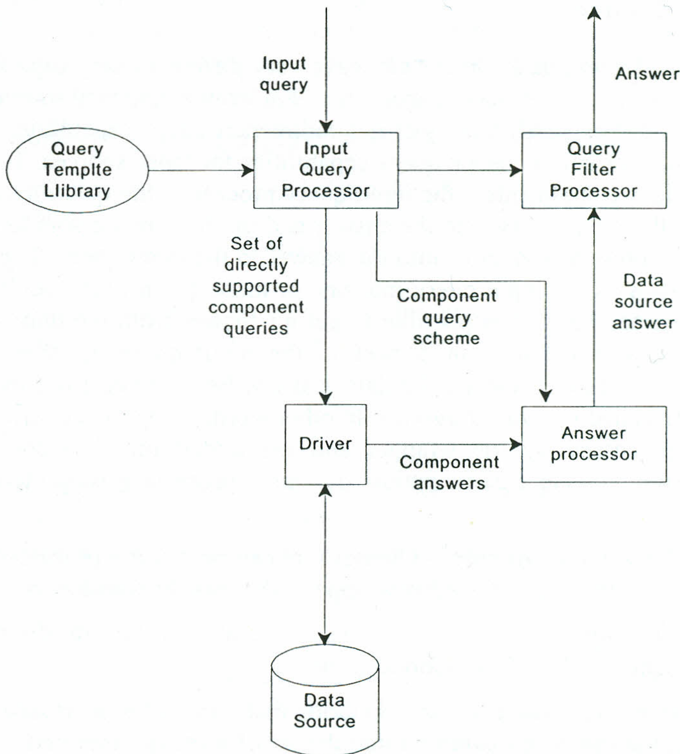

Figure 3: A Data Source Wrapper

A query is accepted by the *input query processor* first. The input query processor transforms the query into a form that can be compared with query templates from the *query template library*. According to the result of that comparison, the input query processor transforms the input query into one or more component queries that the underlying source is able to process, and sends these component queries to the *driver*. The *driver* transforms queries into the query language that the underlying source understands, and sends these queries to the underlying data source, then it gets the answers from the data source, and it transforms answers into a form that the mediator understands. The answers are proceeded to the *answer processor*. The answering processor combines component answers, according to the component query scheme, into a single answer. This answer is sent to the *filter processor* where the answer is filtered. The filtered answer is the answer to the input query.

It is easy to see that in this concept of a wrapper only a driver and a query template library are data source dependent, that and all other components can be built globally, for all of the wrappers together.

3.1. Supported Queries

As was said before, data sources can have very different query capabilities. Some of them can have very poor query capabilities. For example, legacy sources (data files with a searching command) have just got rudimental query capabilities. One of the aims of the wrapper is to enhance query capabilities for those sources. To do that, the wrapper uses four components – the input query processor, the query template library to decompose the input query, and the answer and the filter processors to combine the answers to the component queries into the answer to the input query. The input query processor transforms the input query into one or more queries that can be run on the underlying data source, if it is possible to get an answer from the underlying source that can be transformed into an answer to the input query. Queries that can be processed on the wrapper and on the data that can be obtained from the underlying data source are called *supported queries*. In other words, supported queries are queries that can be processed by the wrapper and the underlying data source together. According to the wrapper's participation, the query processing supported queries can be divided into:

- *Directly supported queries* – Queries that can be directly processed by the data source. For this type of queries wrapper only does the translation.
- *Logically supported queries* – Queries that can be transformed into an equivalent and directly supported query.
- *Indirectly supported queries* – Queries that cannot be processed by the data source, but can be transformed into the set of directly supported queries whose answers can be combined, by selection, projection and join operations, into an answer to the input query.

It is obvious that directly supported queries make a subclass of the class of logically supported queries. In the same manner, logically supported queries form a subclass for the class of the indirectly supported queries.

To process the indirectly supported query a definition of a *query plan* is needed. The query plan defines the way we process queries. The plan contains a set of directly supported queries that are called *component queries*, which can be processed by the data source, and the *filter*. The filter is a query that is processed on the wrapper using selection, projection and join operations, in answer to the input query.

Generally, the filter could be any query that uses the answers of component queries. But it is not necessary to define the filter so widely. It is sufficient to define the filter as a query that uses the answers to the component queries as relations, and projection, selection and join as operations on them. This means that the filter cannot use any other data that is not given in the answer to some component query.

Now, we shall give definitions that define the terms informally introduced above.

Definition 1: A query **q** is a *logical consequence* of a query **q'** if an answer to the query **q** can be produced from an answer to the query **q'** by a projection and a selection on the database that supports both queries directly.

Directly supported queries are queries to which the database can give an answer. Definition 1 will allow us to formalize indirectly and logically supported queries. It is obvious that every indirectly supported query has to be the logical consequence of the join of some directly supported queries.

To define indirectly supported queries we need another definition:

Definition 2: Let **q** be a query. A *filter query* f_q is every query that has the following form:

$answer_q(Y)$, <*cond*(*Y*)>

where $answer_q$ is an answer to the query **q**, and *cond* is a conditional query on the variables from the vector *Y*.

Let us define clause:

$answer_q$(Y):- <q>.

which describes s predicate $answer_q$ whose intention is an answer to the query **q**. In other words, $answer_q$ describes an answer to the query **q**. Then the filter query on the query **q** is every query that contains an instance of the $answer_q$ for some vector Y, and conditional atoms. Conditional atoms are atoms of the form

<*term1*>Θ<*term2*>

where $\Theta \in \{<, >, =, \backslash=, >=, =<\}$. It has to be said that the vector Y can contain unnamed variables _, which embody projections in Prolog-like languages. This is a Prolog-like definition of the filter query. In the relational algebra-like language, a conditional query can be defined as a query that contains only selections and projections as operators.

Definition 3: Let $\mathbf{q_1}$ and $\mathbf{q_2}$ be queries. We can say that the query $\mathbf{q_1}$ is *indirectly supported by the query* $\mathbf{q_2}$ if there is a filter query $\mathbf{f_{q2}}$ that is equivalent to the query $\mathbf{q_1}$.

Now we will define the meaning of indirectly supported query by some other query. But, this is not enough to cover all of the indirectly supported queries. Namely, there are queries that are not indirectly supported by one directly supported query, but by the join of several directly supported queries. Definition 3 can easily be extended into a definition that includes this case too.

Definition 4: Let $\mathbf{q_1}, ..., \mathbf{q_n}$ be queries, and let **q** be a join of these queries. Then we can say that the query $\mathbf{q}^*$ is indirectly supported by the set of queries $\mathbf{q_1}, ..., \mathbf{q_n}$ if it is indirectly supported by **q**.

3.2. Query templates

In this section, we will introduce query templates – templates that describe the query capabilities of the query language of the data source. Many ways of defining query templates can be found in the literature related to this area, but we will use DCG to define them. The reasons for this are given in the introduction to this paper.

To define the query capabilities of some data source, we have to define the queries that the source can process. Normally, a data source supports some class of queries that can be described by looking at several syntactic roles. So the natural way to define the query capabilities of a query language is to define its grammar. But, in our case, wrappers will get the queries from the mediator in the mediator query language. It would not be rational to translate every query from the mediator and then check if this query is supported by the data source. It is more rational to define the grammar of the sublanguage of the language of the mediator which, when translated, is directly supported by the data source. This means that we will define the grammars of the sublanguages of HiLog. The translation of the Prolog-like language into some standard query language (relational algebra, for example) is described in detail in [1].

With regard to the language capabilities of Prolog-like languages, a problem arises, namely that the grammar should include, and recognise, many variants of the same query. The problems are that the order of the atoms in Prolog-like languages are irrelevant, and the join and selection operations could be defined in an implicit way. For example, if we have the relations **P(A,B)**, and **Q(B, C)**, and we want to make the natural join of these two relations, we could write the following query:

?- p(X,Y), q(Y,Z). (1)

In this query, the join is defined implicitly, by the repeating of variable **Y** from the first atom, in the second atom. If we want to avoid an implicit definition of the join in this query, we could write this query in the following way:

?- p(X,Y), q(U,Z), U=Y. (2)

The query (2) is equivalent to the query (1), but it is much easier to analyse because the join operation is defined explicitly by the third atom.

In the same way, if we want to introduce the selection into the query (1), so that we would get only those answers that are build from the records from the **Q** relation, in which the value of the second argument is equal to 1, the query would then have to be:

?- p(X,Y), q(Y,1). (3)

In the query (3), both operations, the join and the selection operatoins are defined implicitly. To define them explicitly, we need to write the following query:

?- p(X,Y), q(U,Z), U=Y, Z=1. (4)

It is obvious that every query can be written in such way that all of the relation operations are defined explicitly. The following definition gives a formalisation of the queries where all the operations are defined explicitly.

Definition 5: We said that the query is in the *descriptive normal form* (DNF) if the following conditions are met:

(1) In the atoms of the form $\mathbf{p(t_1, ..., t_n)}$, all $\mathbf{t_i}$, $\mathbf{i=1, ..., n}$ are variables or queries in the descriptive normal form.
(2) In atoms of the form $\mathbf{p(t_1, ..., t_n)}$, every variable occurs only once.

It is obvious that every query could be rewritten in DNF. The following proposition proves that and gives an algorithm for transformation of queries into DNF.

Proposition 1: For every HiLog query there is an equivalent HiLog query in DNF.

Let query **q**

$$?\text{-}\ \mathbf{p_1(t_{11}, ..., t_{n1}), ..., p_m(t_{1m}, ..., t_{nm})}, \langle cond \rangle.$$

be a query, in which *cond* is some conditional query that uses constants and variables which appear in the first part of the query. We can always write a query in such a way that allb the conditional atoms come at the end of the query. This is because the order of the atoms in a query is irrelevant as long as the query has a finite answer.

We will read the query from left to right, then examine it to see if it is in the DNF or not, and if it is not we will transfer the query so that it will be in the DNF.

The algorithm is as follows:

1. Set **i:=1**
2. Examine if the atom $\mathbf{p_i(t_{1i}, ..., t_{mi})}$ is in DNF. If it is not. Transfer the query to the **q'** so $\mathbf{p_i(t_{1i}, ..., t_{mi})}$ is in the DNF.
3. Set **i:=i+1**. Go to the step 2.

It is obvious that this algorithm will have exactly **m** steps for the query **q**, and that, after it has finished, the transformed query will be in the DNF. The only thing that is not clear is the possibility of performing one step of the algorithm in the finite time. The following procedure will examine if the atom $\mathbf{p_i}$ is in DNF and, if it is not, transform the query **q** in which the atom $\mathbf{p_i}$ is the first atom that is not in the DNF into the query **q'**, in which the atom $\mathbf{p_i}$ is in the DNF.

1. Set **j:=1**
2. If $\mathbf{t_{ij}}$ is the constant **c**, then transform the atom $\mathbf{p_i}$ by introducing a new variable $\mathbf{X_{ij}}$ in place of the term $\mathbf{t_{ij}}$ and add a conditional atom $\mathbf{X_{ij}=c}$ at the end of thequery.
3. If the term $\mathbf{t_{ij}}$ is the variable **Y** that appears in the query for the second time, then transform the atom $\mathbf{p_i}$ by introducing a new variable $\mathbf{X_{ij}}$ in place of the term $\mathbf{t_{ij}}$ and add a conditional atom $\mathbf{X_{ij}=Y}$ at the end of the query.
4. If the term $\mathbf{t_{ij}}$ is an atom, then repeat this procedure for that atom.
5. Set **j:=j+1**. Go to step 2.

To prove a finiteness of step 2, we should be able to introduce a new variable name in a finite time, and in fact we can do just that. For step 3 we should be able to find out if the variable appears in the query for the first time and to introduce a new variable name in a finite time, and it is also possible to do this.

To prove the finiteness of step 4 we should recall that HiLog formulae are of finite length, so after, at most the finite number of recursive steps, we will reach the first order atom, i.e. the atom whose arguments are variables and constants only.

The only thing that remains to be shown is that the transformations that are defined in the algorithm are proper. This means that they transform a query into an equivalent query, and that it is obvious. □

It is shown that the algorithm is finite and correct. The complexity of this algorithm is $\mathbf{O(2^n)}$ where **n** is the number of variables that appears in the query. To see that we should recall that the algorithm for Skölemization, since predicate calculus is NP-complete because of the task of introducing new variables into the formulae. Namely, to introduce new variables into the formula, exponential time regarding the number of the variables in the formula is needed.

Let us suppose that we have data source that has one relation, say **R(A,B)**, and query language for the data source is implemented and it can do selections on the attribute **A**. Plus, let the query language of the data source only do selections based on the equality of the value of the attribute **A** to the constant.

In this case, the query capabilities of this data source in the relational algebra-like language can be described by the following query template:

$\sigma_{A=<constant>}(R)$

The DCG query template for this source is

```
query -->[[r,[X,Y],t],[=,[X,Z]]],
        {var(X),var(Y) ,anot([A],[]),not(X==Y),atomic(Z)}.
```

Another data source that we will examine is an extension of the first one. Let the data source allow all possible selections on the relation **R**. Then a query can be described by the following query template

```
query -->[[r,[X,Y],t]], cond,            {var(X),var(Y), not(X==Y)}.
```

Where the **cond** defines the conditional query. The **cond** is defined by the grammar below

```
scond --> [[Comp,[Term1,Term2]]],
        {comp([Comp],[]),term([Term1],[]),term([Term2],[])}.
scond --> [[';',[Cond1,Cond2]]],            {cond([Cond1],[]),cond([Cond2],[])}.
scond --> [[not,Cond]] ,
        {cond(Cond,[])}.
```

```
cond --> scond.
cond --> scond,cond.

term --> [X],{var(X)}.
term --> [X],{not(var(X)),atomic(X)}.
comp --> ['<'].
comp --> ['>'].
comp --> ['=<'].
comp --> ['>='].
comp --> ['='].
comp --> ['\='].
```

In this grammar, the definition for the variable **cond** describes the recursively conditional query as a simple conditional query (**scond**), or as a conjunction of a simple a conditional query and a conditional query (that is not necessary a simple one). The simple conditional query is defined by three grammar clauses. The first one defines a binary comparison condition, where comparison operators are given by the definition of the **comp** variable. That clause defines a simple conditional query as a binary operator and two terms. The term in this definition is constant (atom or number) or a variable. The second clause defines a simple conditional query as a disjunction of two conditional queries. This is necessary because of way that HiLog (and other Prolog-like) languages process the DCG. Only the conjunction is an ordinary connective in DCG, so a disjunction has to be defined explicitly. The third clause defines the negation of the conditional query as a simple conditional query.

And at the end of report we will give the DCG template for the data source that allows all possible selections, projections and joins.

```
query(X,Y) --> rel(X,Y).
query(X,Y) --> rel(X1,Y1), cond(X2,Y2),  {append([','],[X1],X3),
                                           append(X3,[X2],X4),
                                           X ^=.. X4,append(Y1,Y2,Y)}.
query(X,Y) --> cond(X,Y).

rel(X,X1) --> [[R,X1,A]],             {relation(R,N), length(X1,N), anot(A),
                                        append([R],X1,X)}.
rel(X,Y) --> [[R,X1,A]],rel(X2,Y1), {relation(R,N), length(X1,N),
                                        anot(A), append([R],X1,X3),
                                        append([','],[X3],X4),
                                        append(X4,[X2],X5),X ^=.. X5,
                                        append(X1,Y1,Y)}.

scond(X,Y) --> [[Usp,[Term1,Term2]]],       {comp([Usp],[]),
                                              term(Y1,[Term1],[]),
                                              term(Y2,[Term2],[]),
                                              append([Usp],[Term1],X1),
                                              append(X1,[Term2],X2),
                                              X ^=.. X2, append(Y1,Y2,Y)}.
```

```
scond(X,Y) --> [[';',[Cond1,Cond2]]],   {cond(X1,Y1, [Cond1],[]),
                                          cond(X2,Y2,[Cond2],[]),
                                         append([';'],[X1],Y1),
                                         append(Y1,[X2],Y2),
                                          X ^=.. Y2,append(Y1,Y2,Y)}.
scond(X,Y) --> [['not',Cond]] ,          {cond(X1,Y,Cond,[]),
                                         append(['not'],[X1],Y2),
                                         X ^=.. Y2}.

cond(X,Y) --> scond(X,Y).
cond(X,Y) --> scond(X1,Y1), cond(X2,Y2),   {append(X1,X2,X),
                                             append(Y1,Y2,Y)}.

term([X]) --> [X],   {var(X)}.
term([]) --> [X],    {not(var(X)),atomic(X)}.

comp --> ['<'].
comp --> ['>'].
comp --> ['=<'].
comp --> ['>='].
comp --> ['='].
comp --> ['\='].
```

In addition to the query template above, we have to define the predicate **relation/2**, which contains names of relations in the data source, and the arity of these relations. In this way, we have built a more general template that can be easily used for any data source that has these query capabilities. It is very interesting to do so, because this query template defines the selection, projection and join capabilities of relational databases. Also, the predicate **append/3**, that concatenates two lists given in the first and the second argument into the single list, is used.

3.3. Plans

Our goal is to build a wrapper system that extends the query possibilities of the underlying data source and translates queries from the language of the mediator to the language of the underlying data source and vice versa. This means that part of the query will be processed in the underlying source and part of the query will be processed in the wrapper. We said before that a wrapper can do selections, projections and joins on data from the answers to the queries processed in the underlying data source. But the data source may allow some projections, selections and joins itself. So, it is obvious that sometimes there is more than one way to process any indirectly supported query. The question arises: which of these ways is the best? Generally, the tendency is to proceed as many operations as possible in terms of the data source and to minimise the wrapper's participation in query processing.

The formalisation of the possible ways of processing a query is query plan defined by the following definition:

Definition 6: A *plan* **P** for the indirectly supported query **q** on the data source **d** is the set of the queries $\mathbf{q_i}, \ldots, \mathbf{q_n}$ that are directly supported by **d**, and the filter query **f**, so that query **q** is equivalent to the query

$$?\text{-}\ \mathbf{q_i}, \ldots, \mathbf{q_n}, \mathbf{f}.$$

In this set, $\mathbf{q_i}, \ldots, \mathbf{q_n}$ are called the *component queries* of the plan **P**.

It is easy to see that the plan for the directly supported query is that query alone. Plus, the plan for the logically supported query is a directly supported query that is equivalent to that query.

The problem is that there could be more then one plan for a query. We have to define which plan is the best, so that we can choose it. In [4], algebraic optimal plans are introduced to solve this problem, but this is out of the range of this paper. We will use any plan here. Every plan should process a query correctly, and choosing between these plans is the task of the optimization of query processing.

The second problem that arises here is the problem of generating plans for the query. The plans for the query are built in the same way as the plans for the query execution for the relational databases. Any reader interested in this field can refer to [10]. In [4], a HiLog program that creates an algebraic optimal plan is given, and it can be used with the query templates that have been introduced in this paper.

4. CONCLUSION

In this paper, the term of data source wrapper has been introduced. We have shown how the wrapper can be built and integrated into the heterogeneous data source integration system.

A term of query template was introduced, so more general wrappers can be built. For the building of query templates definite clause grammars can be used. It was shown that DCGs are suitable for the task of building query templates.

To make queries easier to process a descriptive normal form was defined, we showed that every HiLog query can be transformed into an equivalent HiLog query in DNF. The transforming algorithm is given.

There are some open questions that were not examined in this paper. The classification of query plans that was introduced in [7] and [4] can be used for the plan for indirectly supported query building. The building plan for HiLog queries with regards to DCGs as query templates was not examined in the literature and could well be an interesting theme to examine.

Building query templates for some common data sources, such as deductive, object-oriented and temporal databases, could be another interesting extension of this paper. Building of the wrapper, based on the ODBC driver for the XSB HiLog is the practical extension that could be very useful. A wrapper that would be created in this way could be used for the wide class of data sources that could allow an ODBC connection.

REFERENCES

[1] S. Ceri,G. Gottlob, L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[2] W. Chen, M. Kifer, S.D. Warren. HiLog: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, Vol. 15, No. 3, 1993, pp. 187-230.

[3] J.E. Hopcroft, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, 1979.

[4] A. Lovrenčić. *Amalgamacija baza znanja*. Master thesis, Faculty of Organization and informatics, 1999.

[5] A. Lovrenčić. Knowledge Base Amalgamation using Higher-Order Logic-Based Language HiLog. *Proceedings of 10th International Conference on Information and Intelligent Systems*, Varaždin, Croatia, 1999.

[6] A. Lovrenčić, M. Čubrilo. Amalgamation of Heterogeneous Data Sources Using Amalgamated Annotated HiLog. *Proceedings of 3rd IEEE Conference on Intelligent Engineering Systems*, INES'99, Stara Lesna, Slovakia, 1999., pp. 285-290.

[7] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, J.D. Ullman. A Query Translation Scheme or Rapid Implementation of Wrappers. *Conference on Deductive and Object-Oriented Databases*, Singapore, 1995.

[8] Y. Papakonstantinou, A. Gupta, L. Haas, Capabilities-Based Query Rewriting in Mediator systems (extended version). *Distributed and Parallel Databases* vol. 6, no. 1, 1998.

[9] A. C. Rakesh, R. Chandra, A. Segev. *Processing Matching-Joins in Multidatabases*. technical paper, 1996.

[10] R. Ramakrishan. *Database Management Systems*. McGrow-Hill, Boston, 1999.

[11] K. Sagonas, T. Swift, D.S. Warren, J. Freire, P. Rao. *The XSB System Version 2.2 Volume 1: Programmers Manual*, State University of New York, Stony Brooks, 2000

[12] V.S. Subrahmanian. Amalgamating Knowledge Bases. *ACM Transactions on Database Systems*, Vol. 19, No. 2., 1994. pp. 291-331

[13] J.D. Ullman. Information Integration Using Logical Views, *International Conference on Databases Theory*, invited paper, Delphi, Greece, 1997., pp. 19-40.

[14] D.S. Warren. *Programming in Tabled Prolog*, Draft Version, SUNY. Stony Brook, 1995.

Received: 14 January 2000
Accepted: 25 May 2000

Alen Lovrenčić

OMOTAČI IZVORA ZNANJA TEMELJENI NA GRAMATIKAMA DEFINITNIH KLAUZULA

Sažetak

Ovaj rad se bavi drugim od dvaju osnovnih problema koji se javljaju pri integraciji heterogenih izvora znanja u jedinstveni izvor, koji sve podatke sa izvora nad kojim je izgrađen, prezentira korisniku na jedinstven način – prevođenjem upitnog jezika integratora u upitne jezike izvora, I obrnuto, prevođenjem formata odgovora koje daju izvori u format razumljiv integratoru. Prvi problem integracije heterogenih izvora znanja – razrješavanje konflikata između podataka s raznih izvora obrađen je u [4][5] i [6].

U ovom se radu uvode omotači izvora znanja, koji rješavaju gore spomenuti problem. Štoviše, omotači omogućuju rješavanje još jednog bitnog problema koji se javlja kod propitivanja heterogenih izvora znanja – prevladavanje razlika u upitnim mogućnostima izvora. Drugim riječima, omotači izvora znanja ujednačavaju upitne mogućnosti izvora koje omataju.

Kako bi se olakšao i smanjio posao koji je potrebno uložiti za izradu omotača izvora znanja, razvija se jezgra koja je zajednička omotačima za sve izvore, a upitne se mogućnosti izvora definiraju pomoću biblioteka upitnih obrazaca, koji se grade korištenjem gramatika definitnih klauzula.

Ključne riječi: heterogeni izvori znanja, integracija, omotač, obrada upita, gramatike.