Uniformly Querying Knowledge Bases and Data Bases

Paolo Bresciani
IRST, I-38050 Trento Povo, TN, Italy
bresciani@irst.it

Abstract

Present KL-ONE-like knowledge base management systems (KBMS), whilst offering highly structured description languages aside efficient concepts classification, have limited capability to manage large amounts of individuals. Data base management systems (DBMS) can, instead, manage large amounts of data efficiently, but give scarce formalism to organize them in a structured way, and to reason with them.

This paper shows how assertional knowledge of KBMS and data of DBMS can be uniformly accessed. The query answering capability of an arbitrary KBMS is augmented with the possibility of accessing external databases (DB) as a supplemental source of extensional knowledge.

The techniques presented in this paper can be easily adapted to several sources of information. From a knowledge acquisition perspective, we believe that they can be usefully applied in all those applications where several sources of informations are available independently from the knowledge bases.

1 Introduction

The two basic components of a KBMS of the KL-ONE family are the terminological box (TBox) and the assertional box (ABox). One of the tradeoff of these KBMS is between the expressiveness of the description languages characterizing their TBox and the inefficiency in managing large amounts of data in the ABox, even when they have a quite schematic form and their classification is completely a priori given. DBMS, instead, are suited to manage data efficiently, with little concern about their dimension, but their formalism for organizing them in a structured way is quite absent, as it is the capability to infer new information from the existing ones.

Here we propose to cope with both KBMS and DBMS together, using them in an integrated way to manage with several kinds of information. Of course a uniform way to retrieve information from a mixed KBMS/DBMS is needed.

In the present paper it is shown how assertional knowledge of KBMS and data of DBMS can be uniformly accessed. A technique to tightly couple KBMS with DBMS [Borgida and Brachman,1993] is described. As in [Devanbu,1993; Borgida and Brachman,1993] we let primitive concepts and relations in a KB correspond respectively to unary and binary tables/views in a DB. Unlike [Devanbu,1993; Borgida and Brachman,1993] we provide a *tight coupling* between KBMS and DBMS, i.e., a *on demand* access to the DB, instead of a *loose coupling*, that requires a pre-loading of the data from the DB into the KB. In this way we obtain the following advantages:

- more complex queries than simply asking for the instances of concepts can be done; just as an example, in our system queries like $C(x) \wedge R(x,y) \wedge D(y)$ can be made.
- no memory space is wasted in the KBMS to keep descriptions of DBMS instances.
- answers are given on the basis of the current state of the KB and the DB.
- no periodical updating of the KB with new or modified data from the DB is needed.

Basically, in our system the query answering capability of an arbitrary KBMS[1] is augmented with the possibility of accessing external information as a supplemental source of extensional knowledge. In particular a database is seen as an extension of the KBMS ABox.

2 DBox as Extension of the ABox

The ABox is the component of a DBMS where assertions about single individuals are stated. In the present paper we describe how the ABox can be extended with an external source of extensional data. We call this extension a 'DBox'. In the following, we adopt the notation of [Nebel,1990] and call a set of term descriptions (concepts and roles) a *terminology* $\mathcal{T}$[2], and a set of individual assertions a *world*

[1] Even if we implemented the ideas presented here as an extension of LOOM [MacGregor,1991], they can be easily applied to any KL-ONE-like KBMS system with a first-order-logic query-language.

[2] An important task of a KBMS is to organize the terms in a taxonomy accordingly with a specialization relation, i.e., to *classify* them; in the following, we often use $\mathcal{T}$ to denote just the set of atomic terms appearing in $\mathcal{T}$, and consider them correctly classified in the taxonomy on the basis of their definitions.

description $\mathcal{W}$[3]; we say also that the set of data expressed in a DBox constitutes a *data base* $\mathcal{D}$. Assuming that two complete query answering functions separately exist, for both the ABox and the DBox, a *knowledge base* $\mathcal{KB} = \langle \mathcal{T}, \mathcal{W}, \mathcal{D} \rangle$ can be defined in such a way that a uniform query function, based on the two answering functions, can be implemented. We do not require any special capability from the DBox, except the one of (quickly) retrieving lists of tuples of items satisfying requested conditions. These conditions are of the kind of *being in a class* – i.e., belonging to a unary table/view – or *being in relation with other items* – i.e., belonging to a binary table/view – and logical combinations of these, as it can be, for example, expressed in SQL. Since our implementation relies, in fact, on a DBMS with SQL, we assume that $\mathcal{D}$ is somehow represented by means of a relational database, and queries to the DBox can be done in SQL. Therefore in the following we will refer to tables/views – or simply tables – as they usually are intended in relational databases, and to the query answering function of the DBox as to those of SQL.[4]

¿From the point of view of users of KBMS, our experience [Bresciani,1992] suggests that, in realistic applications, knowledge bases not only can be complex, but can also involve a large number of individuals: often most of them are already completely and suitably described in some database. We faced several times, in the past, the task of transferring data from these databases to our knowledge bases. Using the techniques described in the present article it is, of course, much more recommended to *link* the databases to the knowledge bases. Using a DBox paradigm we obtain the advantage of reducing to the minimum the effort of transferring data and, moreover, they are automatically kept updated as far as the linked databases are. But also when data must be collected *ex-novo* it can be convenient[5] to manage most of them by means of a DBMS.

3 Coupling

As mentioned, in our $\mathcal{KB}$, $\mathcal{D}$ is assumed to contain no structural knowledge, but just raw data, and is not supposed to have any inferential power. The single instances are, therefore, already placed in the right tables. That is, speaking in KR terms, they are completely a-priori *realized* under the right concept. This corresponds to having each table of $\mathcal{D}$ associated with a *primitive* term[6] of $\mathcal{T}$. We will show how mixed (KBMS/DBMS) queries can be answered in a coherent way, but, to this extent, we need to *couple* the terminology $\mathcal{T}$ in $\mathcal{KB}$ with the data base $\mathcal{D}$. This coupling consists in the association of some particular terms of $\mathcal{T}$ with tables, in the DB representing $\mathcal{D}$, where the extension of these terms are to be found.

For sake of simplicity we adopt, next, some restrictions on the form of $\mathcal{KB}$, even if, as it will be noted later, they can be, at least in part, released. Given $\mathcal{KB} = \langle \mathcal{T}, \mathcal{W}, \mathcal{D} \rangle$, we assume that the following conditions are satisfied:

- **non intermediate db extension**: every $\mathcal{D}$ individual must be realized under a leaf term in $\mathcal{T}$, i.e., a term in $\mathcal{T}$ specialized by no other terms in $\mathcal{T}$.
- **homogeneous extension**: for each leaf term of $\mathcal{T}$ its associated instances are either all in $\mathcal{W}$ or all in $\mathcal{D}$.
- **db isolation**: all the leaf terms of $\mathcal{T}$ whose instances are in $\mathcal{D}$ are primitive and are not used in any other term definition in $\mathcal{T}$.

Consider that it is not difficult to design $\mathcal{KB}$ in such a way that a primitive term is introduced in $\mathcal{T}$ for each class of individuals present in $\mathcal{D}$: by this the **homogeneous extension** hypothesis can always be satisfied. The **db isolation** and the **non intermediate db extension** conditions reflect the hypothesis that $\mathcal{D}$ is just a flat collection of unstructured tables of records of data, without any reasoning capability.

Under these assumptions, all the information needed to correctly drive the query mechanism is the association of those terms in $\mathcal{T}$ whose extensions are in $\mathcal{D}$ with the corresponding tables in the DB. To this extent it is enough to know this association for those terms that are leaf, for the **non intermediate db extension** condition above. Therefore, we assume that a partial mapping $PM : \mathcal{PT} \rightarrow DBtable$ is given, where $\mathcal{PT}$ is the set of primitive terms in $\mathcal{T}$, and $DBtable$ the set of tables in the DB. So, we can define the *marking* function $M : \mathcal{T} \rightarrow 2^{DBtable}$, s.t. $M(t) = \{PM(x) \mid x \in subs(t) \text{ and } PM(x) \text{ is defined}\}$, where $subs(t)$ is the set of all the terms classified under t in $\mathcal{T}$ (including t). The marking function gives the (possibly empty) set of tables necessary to retrieve all the instances (pairs) of a given concept (relation). Therefore, it is an important part of our $\mathcal{KB}$, whose definition, to be more precise, has now to be rephrased: $\mathcal{KB} = \langle \mathcal{T}, \mathcal{W}, \mathcal{D}, PM \rangle$.

4 Query Answering

We are now ready to describe the task of answering a query. Here we will assume that a *query* to $\mathcal{KB} = \langle \mathcal{T}, \mathcal{W}, \mathcal{D}, PM \rangle$ is an expression of the kind $\lambda \overline{x}.P_1 \wedge \ldots \wedge P_n$, where $P_1, \ldots, P_n$ are predicates of the form $C(x)$ or $R(x, y)$, where C and R are respectively a concept and a relation in $\mathcal{T}$ and each of x and y appears in $\overline{x} = \langle x_1, \ldots, x_m \rangle$ or is an instance in $\mathcal{W} \cup \mathcal{D}$. As a first, informal example, let us consider the case in which all the $P_1, \ldots, P_n$ can be managed by the DBox only, that is: $M(t) \neq \emptyset$ for each $t \in subs(P_1) \cup \ldots \cup subs(P_n)$.

[3] By $\mathcal{W}$ we denote also the set of individuals described in $\mathcal{W}$.

[4] Of course, the external source of information where the DBox searches data can be of any kind, provided only that it can be accessed via a first-order-logic query language.

[5] if not necessary, considering that most KBMS cannot cope with more than some hundreds or few thousands of individuals.

[6] A primitive term is a term whose definition gives only necessary but not sufficient conditions; individuals cannot be realized under one of these terms unless it is not explicitly asserted that they belong to it or to a more specific term in the taxonomy.

4.1 Translating Queries into SQL

When each predicate in a query $q = \lambda\overline{x}.P_1 \wedge \ldots \wedge P_n$ can be made correspond to a set of tables in the DB, where the answers have to be found, it can be translated into an equivalent SQL query. Of course, the sets of tables can be easily found via the marking function M. At this point we have just to cope with the union set of tables $\{T_1, \ldots, T_h\}$ and their bindings via the variables in $\overline{x}$. For simplicity, let us suppose that the tables returned by M are composed by one column in the case of a concept (let it be called `left`), and two in the case of a relation (let them be called `left` and `right`). The SQL translation is of the kind:

`SELECT DISTINCT` *select-body*
`FROM` *from-body*
`WHERE` *where-body*

where the *select-body* is a list of column names of the kind $M(P_{x_i})$.`left` or $M(P_{x_i})$.`right`, one for each variable x_i in $\overline{x}$, according to the fact that the variable x_i appears for the first time in the predicate P_{x_i} in the first place[7] or in the second place, respectively. The *from-body* is the list of all the tables involved – i.e., all the $M(P_i)$. The *where-body* is a list of SQL `where`-conditions of the kind `field2=field1` or `field2=constant`, where the first form has to be used for each variable that is used more than once, each time it is reused, and the second form occurs for each use of constants. In both the forms `field2` is a selector similar to those in *select-body*, corresponding to positions in the query where the variable is further used or where the constant appears, respectively; `field1` corresponds to the first occurence of the variable.

4.2 The General Case

In general answering, a query is more complex and requires the merging of results from the DBMS and the KBMS. *Answering* a query in $\mathcal{KB}$ means finding a set $\{\overline{x}^1, \ldots, \overline{x}^m\}$ of tuples of instances s.t., for each tuple $\overline{x}^i$, $\lambda\overline{x}.(P_1 \wedge \ldots \wedge P_n)[\overline{x}^i]$ holds in $\mathcal{KB}$. We call such tuples *answers* of the query and the set of all of them its *answer set*.

Due to the definition of answer of a query, it is obvious that, in order to avoid the generation of huge answer sets, free variables should not be used, i.e., each variable appearing in $\overline{x}$ must appear also in the query body (i.e., the part at the right of the dot). Indeed, we adopt a stronger restriction, because the former one still allows for some undesired situations. Let us consider, for example, the query: $\lambda\langle x, y, z\rangle.A(x) \wedge R(x, y) \wedge C(z)$. All the variables appear in the body, but, nevertheless, the answer set of the query can be unreasonably large, due to the fact that all the answers of the sub-query $\lambda\langle x, y\rangle.A(x) \wedge R(x, y)$ have to be *combined* with all the answers of the sub-query $\lambda\langle z\rangle.C(z)$. We say that such a query is *unconnected*. More in general, we say that a query is *unconnected* when it can be split into two or more sub-queries s.t. all the variables appearing in each of them does not appear in any other. We call these sub-queries *clusters*. It is obvious that the relevant result of answering an unconnected query is equivalent to the union of the single results of separately answering the clusters, in the sense that all the information is included in it. But, if we consider the formal definition of answer, we must consider the fact that the overall result must contain tuples longer than those resulting by submitting the single clusters; to obtain all the tuples satisfying the definition of answer the single answers have to be combined by a sort of *Cartesian product*. More exactly, if, after having reordered the variables, un unconected query is written as $\lambda\overline{x}.\varphi_1(\overline{x}_1) \wedge \ldots \wedge \varphi_n(\overline{x}_n)$ – where $\overline{x}$ is the concatenation of the other vectors ($\overline{x} = \overline{x}_1 \cdot \ldots \cdot \overline{x}_n$), and $\varphi_1(\overline{x}_1), \ldots, \varphi_n(\overline{x}_n)$ corresponds to the single clusters – and given that the asnwers sets of a generic cluster $\lambda\overline{x}_i.\varphi_i(\overline{x}_i)$ is $S_i = \{\overline{I}_i^1, \ldots, \overline{I}_i^{j_i}\}$, the answer set of the whole query is $S = \{\overline{I}_1^{j_1} \cdot \ldots \cdot \overline{I}_n^{j_n} \mid \overline{I}_1^{j_1} \in S_1, \ldots, \overline{I}_n^{j_n} \in S_n\}$.

The case of a *connected* (i.e., non unconnected) query $\lambda\overline{x}.\varphi(\overline{y})$ with unbound variables can be reduced to the case of an unconnected query $\lambda\overline{x}.\varphi(\overline{y}) \wedge T(\overline{z})$, where $\overline{z} = \langle z_1, \ldots, z_k\rangle$ contains all the variables appearing in $\overline{x}$ but not in $\overline{y}$, and $T(\overline{z}) = top(z_1) \wedge \ldots \wedge top(z_k)$, where *top* correspond to the most generic concept in $\mathcal{T}$.

It is now clear that unconnected queries and queries with unbound variables may have unreasonably large answer sets, without giving any further capability to the system. Therefore, we consider only connected queries with only bound variables.

To afford the answering of a query we need to split it into sub-queries that can be answered by the two specialized query answering functions of the KBMS and the DBMS. To this extent we need, as a first step, to *mark* all the possible atomic predicates, corresponding to the terms in $\mathcal{T}$, and say that a term P is:

- DB-marked iff for each $t \in subs(P) \cap \mathcal{PT}$ $PM(t)$ is defined .
- KB-marked iff for each $t \in subs(P) \cap \mathcal{PT}$, $PM(t)$ is undefined.
- Mixed-marked otherwise.

These three markings reflect the fact that the instances (pairs) of P are all in $\mathcal{W}$, all in $\mathcal{D}$, or part in $\mathcal{W}$ and part in $\mathcal{D}$, respectively. The strategy for answering to a query is based on this information. Let us, first, observe that it is easy to answer to an atomic query where the predicate is a KB-marked or a DB-marked term. In the first case it is enough to submit it to the KBMS. In the second it is enough to translate the query in a SQL equivalent, as shown above, and submit it to the associated DB. Moreover, if the query is not atomic, but made up by atomic subexpression all with the same marking, the same strategy is applied. More difficult is the case of queries with Mixed-marked predicates. Even the atomic case is quite difficult; it is necessary to transform the atomic query into the (possibly non atomic) one whose predicates correspond to all the leaf terms that specialize the only term in the original atomic query, proceed as before, and collect all the results.

Let us now consider a generic non atomic query: $\lambda\overline{x}.P_1^{KB} \wedge \ldots \wedge P_{l_{KB}}^{KB} \wedge P_1^{DB} \wedge \ldots \wedge P_{l_{DB}}^{DB} \wedge P_1^{M} \wedge \ldots \wedge P_{l_M}^{M}$

[7] or the only one in the case of concept.

where the P_i^{KB} corresponds to the KB-marked terms, the P_i^{DB} to the DB-marked terms, and the P_i^M to the Mixed-marked terms. The query can be split in the sub-queries: $q^{KB} = \lambda\overline{x}.P_1^{KB} \wedge \ldots \wedge P_{l_{KB}}^{KB}$, $q^{DB} = \lambda\overline{x}.P_1^{DB} \wedge \ldots \wedge P_{l_{DB}}^{DB}$, and $q^M = \lambda\overline{x}.P_1^M \wedge \ldots \wedge P_{l_M}^M$.

4.3 The Algorithms

As we said, the sub-queries q^{KB}, q^{DB}, q^M can be easily processed. The only difficulty is that some of the variables in $\overline{x}$ could be unbound in a sub-query. In this case, as shown before, the answer sets have to be completed, that is, the unbound variables should be made correspond to each instance in $\mathcal{KB}$, for all the found answers, by all the possible combinations. But, in this way, huge answer sets are generated, as in the following sketch of the query-answering algorithm:

1 split the query as sketched above into q^{KB}, q^{DB} and q^M.

2 submit q^{KB} to KBMS, q^{DB} to SQL (after translation) and transform each of the atomic sub-queries q_i^M of q^M into a set of atomic queries corresponding to the leaf terms in $\mathcal{T}$ that specialize q_i^M; submit them to the specific retrievers.

3 collect all the answers respectively in the answer sets $AS^{KB}_{\overline{x}_{KB}}$, $AS^{DB}_{\overline{x}_{DB}}$, and $AS^{M}_{\overline{x}_{M}}$, and *complete* them with the whole domain in the place of unbound variables, as mentioned above, generating $AS^{KB}_{\overline{x}}$, $AS^{DB}_{\overline{x}}$, and $AS^{M}_{\overline{x}}$.

4 the overall answer set is just $AS^{KB}_{\overline{x}} \cap AS^{DB}_{\overline{x}} \cap AS^{M}_{\overline{x}}$.

Of course this first algorithm is widely space wasting. Moreover, in step 3 it is not clearly stated how to collect the answers of the sub-queries q_i^M. We try here to shortly describe this operation and to show how the completions of $AS^{KB}_{\overline{x}_{KB}}$, $AS^{DB}_{\overline{x}_{DB}}$, and $AS^{M}_{\overline{x}_{M}}$ in step 3, and their following intersection in step 4, can be obtained more efficiently. To solve these problems, from step 3 ahead a compact representation for $AS^{KB}_{\overline{x}}$, $AS^{DB}_{\overline{x}}$, and $AS^{M}_{\overline{x}}$ is needed. Let a generic partial answer set be written as $AS_{\overline{y}}$, where the variables of the original complete variable tuple $\overline{x}$ missing in $\overline{y}$ are, $x_{p_1}, \ldots, x_{p_k}$. Its completion can be represented in a compact way with $AS_{\overline{x}} = \bigcup_{\overline{I} \in AS_{\overline{y}}} \{\overline{I^\star}\}$, where $\overline{I^\star}$ are equivalent to $\overline{I}$ except that are lengthened by filling the k missing positions $p_1, \ldots, p_k$ with any marker, e.g., a star '$\star$'. The star stands for all the individuals in $\mathcal{KB}$. Using this representation for the completion in step 3, it is now easy to rephrase step 4 of the algorithm as a merging operation. In fact answer sets $AS^{KB}_{\overline{x}}$, $AS^{DB}_{\overline{x}}$, and $AS^{M}_{\overline{x}}$ can be *merged* into a single answer set as follow:

4.1 let `result-list`$=\{AS^{KB}_{\overline{x}}, AS^{DB}_{\overline{x}}, AS^{M}_{\overline{x}}\}$

4.2 choose two answer sets, AS_1 and AS_2, in `result-list`, where answers have at least one common position filled by individuals, i.e., not $\star$.[8]

4.3 merge AS_1 and AS_2 by collecting only those answers in AS_1 where each non-$\star$ filled position is filled by the same individual or by $\star$ in some answers in AS_2, and replace in the collected answers each $\star$ with the individuals in the corresponding position in all the matching answers of AS_2

4.4 replace AS_1 and AS_2 in `result-list` with their merging computed in step 4.3

4.5 REPEAT from step 4.2 UNTIL only one item is left in `result-list`.

4.6 RETURN the only item left in `result-list`.

Now it is easy to explain how to collect the answers of the sub-queries q_i^M of step 2. It is enough, for each $q_i^M \in \{q_1^M \ldots q_h^M\}$, to collect all the answers of all its *descendant queries*, and complete these answer sets generating $AS^{M}_{\overline{x},1}, \ldots, AS^{M}_{\overline{x},h}$, as described above; it is now clear that, in the above algorithm for step 4, step 4.1 has to be so rephrased:

4.1-bis

let

`result-list`$=\{AS^{KB}_{\overline{x}}, AS^{DB}_{\overline{x}}, AS^{M}_{\overline{x},1}, \ldots, AS^{M}_{\overline{x},h}\}$.

The resulting algorithm, composed by steps 1, 2, 3 (modified as shown), 4.1-bis, and 4.2 to 4.6 has been implemented. In our system the KBMS currently in use is LOOM [MacGregor,1991], and the database query language is SQL, but, as mentioned, also other systems could be easily used.

5 Conclusion and Future Developments

We have shown how a third component, a DBox – allowing for the extensional data to be distributed among the ABox and the DBox – can be added to the traditional TBox/ABox architecture of KBMS. By means of the DBox is possible to couple the KBMS with, for example, a DBMS, and use both the systems to uniformly answering queries to knowledge bases realized by this extended paradigm. The presented query language has some restrictions, and some constraints have been imposed to the form of the knowledge bases. To overcome these limitations, some extensions of the present work can be proposed.

5.1 Constraints on the Form of $\mathcal{KB}$

In section 3 we assumed that some constraints should be imposed on the form of $\mathcal{KB}$. Indeed they can be in part released, even if this more general approach would require a deeper discussion and a reformulation of the algorithms. Here we try to give a very short account on possible developments in this direction. First, consider the **homogeneous extension** condition. It is important because it allows to make the search of the answers simpler, giving the basis for a neat separation between KB-marked, DB-marked, and Mixed-marked predicates[9]. But it

[8] Such two sets do always exist, otherwise the query would be unconnected, while we assumed to deal only with connected queries.

[9] and giving also the way to decompose the Mixed-marked predicates in sets of KB-marked and DB-marked ones.

is even more important when considered in conjunction with the **db isolation** condition. In fact we can easily cope with leaf terms having instances from both $\mathcal{W}$ and $\mathcal{D}$ by submitting the corresponding subqueries to both the specialized retrieving functions, and then proceeding with the merging as usual. But, allowing this ambiguity would make more complex the formulation of the **db isolation** condition, that could become:

- **db isolation**: all the leaf terms of $\mathcal{T}$ whose instances are even only in part in $\mathcal{D}$ are primitive and are not used in any other term definition in $\mathcal{T}$.

Indeed we can, at least in part, give up also with this condition. In fact, while keeping the fact that such term must be primitive – this is pragmatically coherent with the fact that the raw information coming from the DB cannot be inferred – we can allow such term to be used inside new, eventually even non primitive, definition. To this extent we need a much more complex schema for translating queries on DB-marked term into SQL. For example, if the query is of the kind $\lambda\langle x\rangle.C(x)$ where $C \doteq \mathbf{some}(R, D)$, its SQL translation could be:

```
SELECT  M(R).left
FROM    M(R)
WHERE   M(R).right IN M(D)
```

Similarly, a translation for the **all** operator could be given, as in [Borgida and Brachman,1993], but in this case some extra considerations about the adequacy of the standard extensional semantics of this operator, when used in a database context, would arise. In fact, the empty satisfiability of an **all** clause would be hardly suited for a DB.[10]

In the example above D is supposed to be a primitive atomic DB-marked concept. Another extension to be explored is about releasing this constraint. Again, some concerns about semantics adequacy should probably be adressed.

Also the **non intermediate db extension** condition has, after the considerations above, to be revised. In fact, even if we must still consider the informaton of $\mathcal{D}$, as they are given, as being *a priori* fully realized in the leaves of the taxonomy, because the tables in the DB, where the instances of $\mathcal{D}$ are described, are not structured in a hierarchy, it could happen that non primitive concepts specialize the DB-marked ones, as in the previous example on the **some** operator.

5.2 The Query Language

Another iussue to be explored regards the query language. Currently our system support existentially quantified conjuntions of atomic formulae.

We plan to expand its capability with the possibility of answering any first-order-logic query. We foresee that, to this extent, much attention has to be paid on the optimization of the queries.[11]

[10] As we argued even for standard knowledge bases [Bresciani,1991] the **every** operator [Franconi,1992] would be more adequate in this case.

[11] Because in our system queries to KB and to DB are managed in a uniform way, the approach of [Buchheit *et al.*,1994] can be usefully applied.

5.3 Aknowledgments

Our thanks must be addressed to Enrico Franconi, for his careful reading of several preliminary copies of the present paper and the useful suggestions he made about it. We also thank Fabio Rinaldi, for his implementation of the SQL interface.

References

[Borgida and Brachman, 1993] Alex Borgida and Ronald J. Brachman. Loading data into description reasoners. In *Proceeding of ACM SIGMOD '93*, 1993.

[Bresciani, 1991] Paolo Bresciani. Logical account of a terminological tool. In *Proc. of the IX Conference on Applications of Artificial Intelligence*, Orlando, FL, 1991.

[Bresciani, 1992] Paolo Bresciani. Representation of the domain for a natural language dialogue system. Technical Report 9203-01, IRST, Povo TN, 1992.

[Buchheit *et al.*, 1994] Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, and Martin Staudt. Subsumption between queries to object-oriented databases. *Information Systems*, 19(1):33–54, 1994.

[Devanbu, 1993] Premkumar T. Devanbu. Translating description logics to information server queries. In *Proceedings of Second Conference on Information and Knowledge Management (CIKM '93)*, 1993.

[Franconi, 1992] E. Franconi. Extending hybridity within the YAK knowledge representation system. *AI*IA notizie, the Italian Association for Artificial Intelligence Journal*, 5(2):55–58, June 1992.

[MacGregor, 1991] R. MacGregor. Inside the LOOM description classifier. *SIGART Bulletin*, 2(3):88–92, 1991.

[Nebel, 1990] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, New York, 1990.