[Página intencionalmente en blanco]

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

IMPLEMENTING CONSISTENT QUERY ANSWERS IN INCONSISTENT DATABASES

ALEXANDER CELLE TRAVERSO

A thesis submitted in conformity with the requirements for the degree of
Master of Science in Engineering

Supervising professor:
LEOPOLDO BERTOSSI D.

Santiago de Chile, 2000

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Departamento de Ciencia de la Computación

IMPLEMENTING CONSISTENT QUERY ANSWERS IN INCONSISTENT DATABASES

ALEXANDER CELLE TRAVERSO

A thesis submitted in conformity with the requirements for the degree of
Master of Science in Engineering

Supervising professor:
LEOPOLDO BERTOSSI D.

Santiago de Chile, 2000

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Departamento de Ciencia de la Computación

IMPLEMENTING CONSISTENT QUERY ANSWERS IN INCONSISTENT DATABASES

ALEXANDER CELLE TRAVERSO

Thesis presented to the comission formed by the following professors :

LEOPOLDO BERTOSSI D.

JAVIER PINTO B.

RICARDO BAEZA-YATES

MIGUEL RIOS O.

to complete the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, 2000

It's strange. The gulls who scorn perfection for the sake of travel go nowhere, slowly. Those who put aside travel for the sake of perfection go anywhere, instantly.

Chiang

To my parents and my beloved Nonno Aldo

ACKNOWLEDGEMENTS

I would like to begin by expressing my sincere thanks to my supervisor, Leopoldo Bertossi, for his guidance, advice, and assistance for preparing this thesis. The completion of this work would not have been possible without him sharing his time, experience, and knowledge with me.

A special thank to Marcelo Arenas for his time, advice and illuminating conversations.

I would also like to thank Baoqiu Cui, from the XSB development team, for supplying the needed patch for the XSB–ODBC interface to work.

Finally, I would like to thank my parents, whose lifelong encouragement and support made this work possible.

CONTENTS

Page

Page

LIST OF TABLES

Page

LIST OF ALGORITHMS

Page

RESUMEN

Consultar una base de datos inconsistente es una situación peligrosa, especialmente si consideramos que, desde un punto de vista lógico, podemos concluir cualquier cosa de un conjunto de fórmulas inconsistente. Es por ello que la manera usual de enfrentar bases de datos inconsistentes sea repararla, es decir, llevarla de vuelta a un estado consistente para así poder obtener información significativa (consistente) a partir de ella. Esto, sin embargo, tiene un gran inconveniente: generalmente incluye descartar los datos inconsistentes, y por lo tanto se pierde información potencialmente útil.

Para evitar este derroche, se presenta el algoritmo $QUECA$, el cual, dada una consulta de primer orden Q, genera una nueva consulta $QUECA(Q)$ tal que, al consultar la base de datos, sus respuestas corresponden sólo a las respuestas consistentes de Q. El algoritmo fue probado correcto, de terminación finita y completo para una clase de restricciones de integridad que incluye a la mayoría de las restricciones encontradas en sistemas de bases de datos relacionales. La complejidad del algoritmo también fue abordada.

Por último, la implementación del algoritmo en XSB es presentada. Se aprovechó la funcionalidad de XSB como un lenguage de programación lógico con capacidad de 'tabling', además de la posibilidad de conectarlo a bases de datos relacionales, para crear así una aplicación de consulta interactiva.

ABSTRACT

Querying an inconsistent database is a dangerous situation, specially if we consider that, from a logical point of view, we can conclude anything from an inconsistent set of formulas. Thus, when facing an inconsistent database the usual approach is to repair it, i.e. take it back to a consistent state to be able to retrieve meaningful (consistent) data. This, however, has one serious drawback: it generally involves discarding inconsistent data, hence losing potentially useful information.

To avoid this discarding, we present algorithm *QUECA* which, given a first order query Q, generates a new query $QUECA(Q)$ such that, when posed to a database, its answers correspond to the consistent answers of Q. The algorithm is proven to be sound, terminating and complete for a class of integrity constraints that includes most of the constraints found in traditional relational database systems. Complexity issues are also addressed.

Finally, the implementation in XSB of the algorithm is described. We take advantage of the functionalities of XSB, as a logic programming language with tabling capabilities, and the possibility of coupling it to a relational database system, to create an interactive application.

I. INTRODUCTION

It is usually assumed that data stored in a database is consistent; and not having this consistency is considered a dangerous situation. However, it often happens that this is not the case and the database reaches an inconsistent state in the sense that the database instance does not satisfy a given set of integrity constraints IC. This situation may arise due to several reasons. The initial problem was due to poor design of the database schema itself or a malfunctioning application that made the system reach the inconsistent state after an update was performed.

Nowadays, other sources of inconsistencies have appeared. For example, in a datawarehouse context (Chaudhuri and Dayal, 1997) inconsistencies may appear, among other reasons, to integration of different data sources, in particular, in the presence of duplicate information; and to delayed updates of the datawarehouse views.

Either case, having a consistent database or not, the information stored in it remains relevant to the user and is potentially useful, as long as the distinction between consistent and inconsistent data can be made, and they can be separated when answering queries.

The common solution for the problem of facing inconsistent data is to repair the database and take it back to a consistent state, by deleting or inserting tuples, before posing a query to it. However, this approach is very expensive in terms of computing power, complexity and, because restoring consistency involves discarding information (i.e. making choices between information), in some cases, we might lose potentially relevant data. In addition, a particular user, without control on the database administration, might want to impose his/her particular, soft or hard constraints on the database (or some views). In this case, the database cannot be repaired.

Example 1 *Consider the inclusion dependency stating that a purchase must have a corresponding client:* $\forall(u, v),\ (Purchase(u, v) \Rightarrow Client(u))$. *The following database instance* r *violates the IC:*

Purchase	
c	e_1
d	e_2
d	e_1

Client
c

When repairing the database we might be tempted to remove all the purchases done by client d*, which provide us with useful information about a client's behavior, no matter whether he is a valid client or not.* ■

A promising alternative to restoring consistency is to keep the inconsistent data *in* the database and modify the queries in order to retrieve only consistent information. By using this kind of approach one can still use the inconsistent data for analysis (purchases of customer d in Example 1). This is often acceptable in a belief base domain because agents may have contradictory beliefs (Cholvy, 1990), but is not common from the classical database point of view.

In (Arenas et al., 1999) a semantic notion of consistent answer to a query was given. In essence, a tuple answer $\bar{t}$ is a consistent answer to a query $Q(\bar{x})$ if $Q(\bar{t})$ becomes true in every repair of the inconsistent database instance r that can be obtained from r by a minimal set of changes. Of course, the idea is not to construct all possible minimal repairs and then query them; this would impossible or too complex. It is necessary to search for an alternative mechanism.

Example 2 *Consider the alternative of computing all possible repairs and a functional dependency stating that every client's second attribute is uniquely determined by the first one, that is:* $\forall u, v, w,\ Client(u, v) \wedge Client(u, w) \Rightarrow v = w$. *The following database instance violates the IC:*

Client	
1	*0*
1	*1*
2	*0*
2	*1*
⋮	⋮
n	*0*
n	*1*

In this case we have 2^n possible repairs, making the alternative of computing all possible repairs and then querying them unpractical. ■

In this context, an operator T_ω was presented in (Arenas et al., 1999) which does not repair the database, but that, given the query $Q(\bar{x})$, computes a modified query $T_\omega(Q)(\bar{x})$ to be posed to the original database instance r, in such a way that its answers are consistent in the semantic sense, that is they belong to all possible repairs. The operator was proven to be sound, complete and terminating for interesting syntactic classes of queries and constraints (Arenas et al., 1999). However, this operator has some drawbacks: it is hard to implement due to its recursive nature and semantic termination condition.

Here we address the problem of designing and implementing an alternative operator inspired by T_ω. The new operator corresponds to an algorithm called *QUECA*, for "*QUE*ry for *C*onsistent *A*nswers"(Celle and Bertossi, 2000). This algorithm, given a first order query[1] Q, again generates a new query $QUECA(Q)$, whose answers in r are consistent with IC, but as opposed to T_ω, it guarantees termination, soundness and completeness for a larger set of integrity constraints.

The implementation is done in XSB (Sagonas et al., 1994), a powerful logic programming system, which is provided with useful functionalities for the right implementation and operation of the consistent query answering algorithm.

[1]Aggregate queries are being treated in (Arenas et al., 2001).

A number of possible applications motivate the development of such an implementation. These include:

Data warehousing (Data Cleaning). A data warehouse contains data coming from many different sources. Some of them, or the integration of them, may not satisfy the given integrity constraints. The usual approach is thus to *clean* the data by removing inconsistencies before the data is stored in the warehouse (Chaudhuri and Dayal, 1997). Our results make it possible to determine which data is in fact clean. Moreover, a different scenario becomes possible, in which the inconsistencies are not removed but rather query answers are marked as "consistent"or "inconsistent". In this way, information loss due to data cleaning may be prevented.

Database integration. Often many different databases are integrated together to provide a single unified view for the users. Database integration is difficult, however, because it requires the resolution of many different kinds of discrepancies of the integrated databases. One possible discrepancy is due to different sets of integrity constraints. Moreover, even if every integrated database *locally* satisfies the same set of integrity constraints, the same constraints may be globally violated. For example, different databases may assign different names to the same student number. Such conflicts may fail to be resolved at all. Therefore, it is important to be able to find out, given a set of local integrity constraints, which query answers returned from the integrated database are consistent with the constraints and which are not.

Data Mining. Often data mining tools are used at the initial stage of the process of building applications on top of an unknown database schema. In these cases relationships between tables (i.e. foreign keys) must be established by intuition. If we consider that this involves joining tables in all possible ways, selecting that that delivers best results (i.e. more tuples), we may think of a tremendous amount of work, not to mention the computational complexity of seeking every possible combination. Even more, if the database were in an inconsistent state, we might be tempted to remove the inconsistent data, not always knowing which tuple(s) is(are) the inconsistent one(s). With our approach the user can rebuild the DB schema incrementally. This process can be done on a trial–and–error fashion, without fear of losing data due to the definition of wrong relationships because inconsistent data is never discarded.

Legacy Data. When dealing with legacy data it is often desirable to impose semantic constraints on it. This usually involves low–level coding, making it a lengthy task which prevents "toying around"with such conditions. With our approach, these constraints may be modified easily at the application level, thus facilitating the experimenting with legacy data.

The rest of this thesis continues as follows. In Chapter 2 we show the most relevant characteristics of the operator T_ω and what makes it difficult to implement. We also give a description of what we will understand by a database repair, query, integrity constraint and consistent answer. Next, in Chapter 3, we present the algorithms which generate a query $QUECA(Q)$ for a given first order query Q. Then in Chapter 4, the properties of these algorithms are analyzed, namely: runtime complexity, termination, soundness and completeness. Also, the set of integrity constraints and queries covered by the solution are characterized. In Chapter 5 we describe issues regarding the implementation done in XSB. Finally, in Chapter 6 we draw some concluding remarks and propose future directions of research.

II. PRELIMINARIES

2.1 Basic Notions

We start from a finite fixed set, IC, of integrity constraints associated to fixed relational database schema. A database instance r is consistent if it satisfies IC, that is, $r \models IC$. Otherwise, we say that r is inconsistent. We assume that IC is consistent in the sense that there is DB r that satisfies IC.

If r is inconsistent, its repairs are database instances (wrt the same schema) that, each of them, satisfy IC and differ from r by a minimal set of inserted or deleted tuples. A tuple $\bar{t}$ is a *consistent* answer to a query $Q(\bar{x})$ wrt IC and we denote this with $r \models_c Q(\bar{t})$, if for every repair r' of r, $r' \models Q(\bar{t})$ (Arenas et al., 1999).

Example 3 *Consider a product database.* $Product(u, v)$ *and* $RetailStore(u, v)$ *mean that product* u *has code* v *in the master product table and the retail store table respectively. The following* ICs *state that products must be present in both tables at the same time and that a product may not have more than one code.*

$$\forall u, v.\ (Product(u, v) \Rightarrow RetailStore(u, v))\ ,$$
$$\forall u, v.\ (RetailStore(u, v) \Rightarrow Product(u, v))\ ,$$
$$\forall u, v, z.\ (Product(u, v) \wedge Product(u, z) \Rightarrow v = z)\ .$$

The following database instance r*, which violates* IC*,*

Product	
a	1
a	2
b	2

RetailStore	
a	1
a	2
b	2

has two repairs:

r' :

Product	
a	1
b	2

RetailStore	
a	1
b	2

r'' :

Product	
a	2
b	2

RetailStore	
a	2
b	2

Here, the only consistent answer to the query $Product(u, v)$*? in the database instance* r *is* $(b, 2)$*:* $r \models_c Product(b, 2)$. ■

2.2 The T_ω operator

The T_ω operator (Arenas et al., 1999) is defined based on a previous residue calculation stage which generates the necessary rules to feed the operator. Intuitively, residues show the interaction between an integrity constraint and a given literal[2]. Generally speaking, it is defined for a query Q, as a sequence of queries $\mathrm{T}_\omega := \{\mathrm{T}_0(Q), \mathrm{T}_1(Q), \mathrm{T}_2(Q) \ldots\}$. If $\mathrm{T}_n(Q) \Rightarrow \mathrm{T}_i(Q)$ for all $i \geq n$, we say that n is the *finiteness point* and computation is stopped. We illustrate the application of this operator by means of an example.

Example 4 *With the set of integrity constraints of Example 3, we will show a computation of* $T_\omega(P(u, v))$*, letting* P *stand for Product and* R *for RetailStore.*

$$
\begin{aligned}
\mathrm{T}_0(P(u,v)) =& P(u,v) \ . \\
\mathrm{T}_1(P(u,v)) =& P(u,v) \wedge (R(u,v) \wedge (\neg P(u,z) \vee v = z)) \ . \\
\mathrm{T}_2(P(u,v)) =& P(u,v) \wedge ((R(u,v) \wedge P(u,v)) \wedge ((\neg P(u,z) \wedge \neg R(u,z)) \vee v = z)). \\
\mathrm{T}_3(P(u,v)) =& P(u,v) \wedge ((R(u,v) \wedge P(u,v) \wedge (R(u,v) \wedge (\neg P(u,w) \vee v = w))) \wedge \\
& ((\neg P(u,z) \wedge \neg R(u,z) \wedge \neg P(u,z)) \vee v = z)) \ .
\end{aligned}
$$

It seems as if T_3 *is very different from* T_2*, however, if we rewrite them by hand we have*

[2]See Sections 3 and 3.1 for a description of what residues are and how to obtain them.

$$\begin{aligned}
\mathrm{T}_2(P(u,v)) =& P(u,v) \wedge (R(u,v) \wedge P(u,v) \wedge ((\neg P(u,z) \vee v = z) \wedge \\
& (\neg R(u,z) \vee v = z))) \ . \\
\mathrm{T}_3(P(u,v)) =& P(u,v) \wedge (R(u,v) \wedge P(u,v) \wedge ((R(u,v) \vee \neg P(u,w)) \wedge \\
& (R(u,v) \vee v = w) \wedge (\neg P(u,z) \vee v = z) \wedge (\neg R(u,z) \vee v = z) \wedge \\
& (\neg P(u,z) \vee v = z))) \ .
\end{aligned}$$

Where we can easily see that $\mathrm{T}_2(P(u,v)) \equiv \mathrm{T}_3(P(u,v))$*, therefore the finiteness point is 2 and the modified query is* $\mathrm{T}_0(P(u,v)) \wedge \mathrm{T}_1(P(u,v)) \wedge \mathrm{T}_2(P(u,v))$ *This query is to be posed to the original database, and its answers should be consistent answers to the original query,* $P(u,v)$*.* ■

Although operator T_ω is sound and complete for non fact-oriented binary integrity constraints, that is, ICs which have at most two database literals plus built-ins and do not generate explicit knowledge, termination is only assured for *uniform* ICs (Arenas et al., 1999). Needless to say that, when thinking of a possible implementation, the termination issue is critical. Detecting the finiteness point in operator T_ω can be very difficult, even in simple examples like the one above (or may be an undecidable problem). An initial approach consisted in using Otter (McCune, 1994) to detect this semantical termination point, but it turned out to be cumbersome and sometimes it did not deliver the expected results. For instance, it was not able to solve the previous example. Furthermore, even if it does work, the *offline* nature of such process makes it unsuitable for a real world implementation where a user should interact directly with the query answering system.

Thus, we need to modify the previous approach to improve the results regarding termination, and possibly extending completeness as well. In consequence, we face the problem of modifying T_ω, providing a new, more practical mechanism, but preserving the nice properties T_ω has in terms of soundness and completeness. We need to add a stronger termination property which makes the new mechanism more likely for implementation. The basic approach involves identifying a stronger syntactical condition to achieve semantically correct results.

2.3 Integrity Constraints

Traditionally, relational databases may be defined as a collection of facts (E_{DB}) and integrity constraints (IC). Our interest in this thesis is for *static integrity constraints*, which account for most of the ICs found in relational databases. In particular, we will deal with *static non-aggregate constraints*.

2.3.1 Semantics

Static integrity constraints are closed first order formulas (Lloyd, 1987). We will assume the set of integrity constraints of a given database, IC, is consistent, that is, there is at least one database instance that makes IC true. When arguing what ICs represent, although we adopted the *logical entailment* approach when defining consistency of a database instance r as $r \models IC$ (Reiter, 1984), this does not fully capture our intuition in the sense that ICs should only serve to validate transactions (query answers). Furthermore, because we are dealing with logic formulas, we could be tempted to treat them as rules to generate data. This, however, again violates our convention as is discussed in (Godfrey et al., 1998):

> ICs are meant as knowledge about the domain of the database and are not intended to generate data as do the rules in the I_{DB}[3], nor do they represent specific data, as do the facts in the E_{DB}.

For example, if IC has an integrity constraint as

$$bakery(Name, Address, Phone, Owner, CloseTime) \Rightarrow store(Name, Address, Phone). \quad (2.1)$$

it is assumed that both *store* and *bakery* relations are obtained separately and that the integrity constraint only establishes the necessary relationship between them. Thus, ICs are used to check the soundness of the answer to a query and not to gen-

[3]Set of rules defined in a deductive database.

erate the answer itself, although they do contribute to construct indirectly inferred answers by eliminating invalid models.

2.3.2 Representation

As in (Arenas et al., 1999), we will only consider *universal* constraints that can be transformed into a standard format

Definition 2.1. *An integrity constraint is in* standard format *if it has the form*

$$\forall(\bigvee_{i=1}^{m} P_i(\bar{x}_i) \vee \bigvee_{i=1}^{n} \neg Q_i(\bar{y}_i) \vee \psi) \ ,$$

where $\forall$ *represents the universal closure of the formula,* $\bar{x}_i$*,* $\bar{y}_i$ *are tuples of variables, the* P_i*'s and* Q_i*'s are atomic formulas based on the schema predicates that do not contain constants, and* ψ *is a formula that mentions only built–in predicates.* ■

Notice that in these ICs, if constants are needed, they can be pushed into ψ. Notice, as well, that equality is allowed in ψ.

Because of implementational issues we shall negate the ICs in standard format, representing ICs as denials, that is range restricted (Nicolas, 1982) goals of the form

$$\Leftarrow l_1 \wedge \cdots \wedge l_n \ , \tag{2.2}$$

where each l_i is a literal and variables are assumed to be universally quantified over the whole formula. We must emphasize the fact that this is just notation and from now on we shall speak of ICs assuming they are in denial form in the sense of classical logic and not that of logic programming.

We shall note, however, that not all integrity constraints may be transformed into standard format, and therefore they are not considered. For example, we leave aside ICs with existential quantifiers like referential ICs as

$$\forall \bar{x} \exists y. \ (P(\bar{x}) \rightarrow Q(\bar{x}, y)) \ . \tag{2.3}$$

III. QUERY GENERATION FOR CONSISTENT ANSWERS

The whole process of query generation for consistent answers relies on the concept of *residues* developed in the context of semantic query optimization (Chakravarthy et al., 1990). Residues, simply put, show the interaction between an integrity constraint and a literal name.[4] Thus, a literal name which does not appear in any constraint does not have any (non–maximal (Chakravarthy et al., 1990)) residues associated to it, i.e. there are no restrictions applied to that literal. Similarly, a literal that appears more than once in an IC or set of ICs, may have several residues, which may or may not be redundant. For example, consider the integrity constraint (2.1) and, as a query, the literal $bakery(x, y, z, u, v)$. The residue supplied by the mentioned integrity constraint for this literal is $store(x, y, z)$, stating that when retrieving a *bakery* as answer, there must also exist a related *store*. In consequence, to make sure we retrieve consistent answers wrt to this integrity constraint, we must consider the residue $store(x, y, z)$ as appended via a conjunction to the query $bakery(x, y, z, u, v)$, emulating a join operation between the two tables.

To calculate the residues in a database schema, we will introduce Algorithm 1, which shows how to systematically obtain residues for a given literal name. Because only literal names appearing in an integrity constraint generate (non–maximal) residues, the algorithm will only be applied to them, and not to every relation in r.

Once we have calculated all the residues associated to a literal name appearing in IC, we shall present a second algorithm, $QUECA$, that will generate the queries for consistent answers based on the residues that have been already computed. We will also show how this algorithm differs from the operator T_ω presented in (Arenas et al., 1999), not only in the delivered results in terms of termination, but in the operation itself and the necessary conditions for sound execution.

[4]Literal names denote relations, so different literals may have the same literal name, e.g. $P(u)$ and $P(v)$ have the same literal name P. Literal names may be negative, e.g. $\neg P$, where P is a predicate name; and have an associated arity that further differentiates them (Prolog convention), so from now on, when talking about a literal, say $P(u, v)$, we are really talking about its literal name, $P/2$.

3.1 Residue calculation

The first step in the residue calculation determines for whom they are to be calculated. In our case, it is for every literal name appearing in an integrity constraint. Because of this we must first build a list of ICs and a list of the distinct literal names L_P appearing in *IC*. This list of integrity constraints L_{IC} will only include the bodies of the ICs (represented in the form (2.2)). That is, given the set of integrity constraints *IC*, we build $L_{IC} = \{[l_1 \wedge \ldots \wedge l_n] \mid \forall(\leftarrow l_1 \wedge \ldots \wedge l_n) \in IC\}$. It should be noted that when negating a member of L_{IC} we obtain a clause.

Example 5 *Let IC be the following set of integrity constraints taken from Example 3 expressed in the form (2.2).*

$$\leftarrow P(u,v) \wedge \neg R(u,v) \ .$$
$$\leftarrow \neg P(u,v) \wedge R(u,v) \ .$$
$$\leftarrow P(u,v) \wedge P(u,z) \wedge y \neq z \ .$$

From this we would generate $L_{IC} = \{[P(u,v) \wedge \neg R(u,v)], [\neg P(u,v) \wedge R(u,v)], [P(u,v) \wedge P(u,z) \wedge y \neq z]\}$ *and* $L_P = \{P(u,v), R(u,v), \neg P(u,v), \neg R(u,v)\}$. *We should recall that in* L_P *we have the following literal names:* $P/2$, $R/2$, $\neg P/2$ *and* $\neg R/2$. ■

Next, to calculate the residues coming from $l \in L_P$, and $ic \in L_{IC}$, we use the subsumption algorithm presented in (Chakravarthy et al., 1990). However, because we are dealing with an implementation, we need a systematical procedure to obtain residues. The method utilized is formalized as Algorithm 1.

Algorithm 1 Compute $residues(l)$

Require: Set of integrity constraints in denial form IC.

Ensure: $residues(l)$ is a formula in CNF that contains all the residues associated to a literal l.

1: Create list L_{IC} of integrity constraint bodies and a list L_P of distinct literal names in L_{IC}.
2: **for all** $l \in L_P$ **do**
3: $i = 1$
4: **for all** $ic \in L_{IC}$ **do**
5: **for** each occurrence of l in ic **do**
6: delete l from $ic \mapsto \overline{ic}$
7: negate $\overline{ic}$ {Now $\overline{ic}$ is in clausal form}
8: $residue_i(l) := \overline{ic}$
9: $i := i + 1$
10: **end for**
11: **end for**
12: $n(l) := i$ {the number of residues associated to l}
13: **end for**
14: **for all** $l \in L_P$ **do**
15: $residues(l) := \varnothing$
16: **for all** $i := 1$ to $n(l)$ **do**
17: **if** $residue_i(l)$ is not redundant **then**
18: $residues(l) := residues(l) \wedge residue_i(l)$
19: **end if**
20: **end for**
21: **end for**

Example 6 *(example 5 continued) Applying Algorithm 1 up to line 13, to* $l = P(u, v)$ *and every member of* L_{IC}*, we would obtain one residue for each occurrence of* $P/2$*:* $residue_1(P(u, v)) := R(u, v)$*,* $residue_2(P(u, v)) := \neg P(u, z) \vee v = z$ *and* $residue_3(P(u, v)) := \neg P(u, w) \vee w = v$*.* ■

Finally, a conjunction of all the residues associated to a given $l \in L_P$ is created and denoted by *residues*(l). In this process, we take care of eliminating redundant residues as we build the conjunction (steps 14–21 in Algorithm 1) in order to reduce complexity in the following phase (*QUECA*). The notion of redundant residues is formalized below.

Definition 3.1 (Residue Redundancy). *Let $R \wedge \varphi$ be a conjunction of residues associated to a literal l, where R is a clause and φ a conjunction of clauses. We will say R is redundant in $R \wedge \varphi$ if there exists a clause $R' \in \varphi$ and a substitution $\sigma : (Var(R')^5 \setminus Var(l)) \rightarrow (Var(R) \setminus Var(l))$, such that $R'\sigma \equiv R$.* ■

Note that, in the definition above, if R is redundant in $R \wedge \varphi$, then $R \wedge \varphi$ is logically equivalent to φ. The elimination of redundant residues is based on unification and is done in steps 14–21 of Algorithm 1.

Example 7 *(example 6 continued) By Definition 3.1, we have that $residue_3(P(u, v))$ is a redundant residue, because there exists a substitution $\sigma : z \mapsto w$, such that $residue_2(P(u, v))\sigma = residue_3(P(u, v))$. Thus, we have $residues(P(u, v)) = [R(u, v)] \wedge [\neg P(u, z) \vee v = z]$.* ■

Note that the definition does not state that it detects *all* redundancies, but only those subject to the sufficient condition presented. For example, if we consider the following residues for $R(x)$: $residue_1(R(x)) = P(x) \vee x > 100$ and $residue_2(R(x)) = P(x) \vee x > 50$. Clearly $residue_1$ includes the information in $residue_2$, so $residue_2$ would be redundant. However, Definition 3.1 does not detect it. This occurs mainly when ICs are redundant, which can easily be avoided for cases like these. As shown in Example 7, functional dependencies are a common case of ICs which generate redundant residues according to Definition 3.1. The reason why residue redundancy is not treated further is due to the complexity of implementation, which could be far higher than the performance improvement we could get in the next stage (*QUECA*). Besides, residue redundancy can become such a large subject that it would deviate the central point of attention of this work, which is to build the queries for consistent answers.

[5] *$Var(E)$ is the set of all (quantified or unquantified) variables in the expression E.*

Example 8 *Finally, by applying Algorithm 1 to the set IC presented in Example 5, we would obtain:*

$$\begin{aligned} \mathit{residues}(P(u,v)) =& (R(u,v)) \wedge (\neg P(u,z) \vee v = z) \ , \\ \mathit{residues}(\neg P(u,v)) =& (\neg R(u,v)) \ , \\ \mathit{residues}(R(u,v)) =& P(u,v) \ , \\ \mathit{residues}(\neg R(u,v)) =& \neg P(u,v) \ . \end{aligned}$$

■

3.2 Query generation ($QUECA$)

Once all the residues have been computed, and given a query Q, we can generate the query, $QUECA(Q)$, which will deliver consistent answers from a consistent or inconsistent database. This query only differs from Q when Q has residues, so $QUECA(Q)$ should be only executed for literal names appearing in IC.

Initially the query $QUECA(Q)$ is equal to Q, plus a list of pending residues which are the residues associated to Q calculated by Algorithm 1.[6] These residues are not yet part of the query, they form a list of pending clauses that must be resolved via some condition if they should belong to the query. This condition is, informally, if they add new information to it or not. If they do not, they are discarded; but if they do, they must be added to the query and their residues appended at the end of the residue list. This procedure is iterated until no residues are left to resolve, i.e. either we run out of residues or they have all been discarded. We will see later that the procedure does not always terminate.

Example 9 *Consider the following hypothetical pairs of queries and residues:*

Query :	1.	$S(u)$	*Residues* :	$S(u)$,
	2.	$M(u)$		$N(u)$,
	3.	$P(u,v)$		$\forall z \ (P(u,v) \vee \neg Q(u,z))$.

[6] The residues are in CNF, we will treat every clause as an element of a list.

Clearly in the first case the residue can be discarded because it adds no new information to the query. However, in the second and third cases the residues must be added to the corresponding query and their residues to the Pending Residue List. So we would have:

$$\begin{array}{lll} \textit{Query}: & 1.\ S(u) & \textit{Residues}: \ \varnothing\ , \\ & 2.\ M(u) \wedge N(u) & \textit{residues}(N(u))\ , \\ & 3.\ P(u,v) \wedge \forall z\ (P(u,v) \vee \neg Q(u,z)) & \textit{residues}(P(u,v) \vee \neg Q(u,z))\ . \end{array}$$

The residues that were added to the respective Pending Residue Lists are only mentioned (e.g. $\textit{residues}(N(u))$) to avoid cluttering the example. ■

This method works fine when only conjunctions are involved (case 2 in Example 9), because determining if a residue should be part of the query or not is easy. However, most of the residues are clauses (case 3 in Example 9), so we must somehow deal with disjunction.

The way to solve this problem is by keeping conjunctions together, i.e. working in Disjunctive Normal Form (DNF). To do so, when a clausal residue adds new information to a query, we make as many copies of the query as literals in the residue we are adding, and append to each of them exactly one of the literals in the residue. The pending residue list of each of these new copies must then be the existing list, minus the *resolved* residue (the one being considered), plus the residues coming from the newly appended literal. We shall informally call this a *split* operation. These copies with the added residues, connected together by disjunctions, would constitute the final query $QUECA(Q)$.

Example 10 *(example 9 continued) In the third case of the previous example we would then have*

$$\begin{array}{lll} \textit{Query}: & & \textit{Residues}: \\ & 3.\ E_1 : P(u,v) \wedge P(u,v) & R_1 : \textit{residues}(P(u,v))\ , \\ & \quad E_2 : P(u,v) \wedge \neg Q(u,z) & R_2 : \textit{residues}(\neg Q(u,z))\ . \end{array}$$

where each E_i simply denotes a copy of the original query after a split operation. So we have $QUECA(Q) = \forall z E_1 \vee E_2$ where each E_i is a disjunctionless formula.

Also, after the split operation, new pending residue lists R_i are generated. They are associated to their respective conjunctive formulas E_i. ■

We must, somehow, keep track of the correspondence between the mentioned copies (E_i's in Example 10) and their associated Pending Residue List (R_i's in Example 10). Recall that each E_i will, in the end, form the final query $QUECA(Q)$ by connecting them via disjunctions, once all the pending residues have been resolved (more about this later).

To maintain the correspondence, we need a new notation that will enable us to keep track of the residues involved in building each E_i. Furthermore, this notation should not only include the literals in E_i and its associated Pending Residue List R_i, but it should also "remember" the last residue that provoked one of these split operations, in order to avoid inserting a residue whose information was already inserted earlier. For these purposes we define a *Temporary Query Unit* (TQU).

Definition 3.2. *A temporary query unit (TQU), $D : E \bullet R$, consists of a set of clauses D, a conjunction of literals E and a conjunction of residues R. A disjunction of temporary query units shall be written as TQUs (note that the final 's' differentiates it from a single TQU).* ■

Both symbols in an arbitrary TQU, ':' and '$\bullet$' in $D : E \bullet R$, are only used to separate D, E and R from each other. D represents the last residues involved in building E (those that must be "remembered") and R is the conjunction of residues $\phi_1 \wedge \cdots \wedge \phi_n$ yet to be resolved, associated to that particular E. We shall note that all variables coming from a residue appear universally quantified in D and E. Also, both symbols have higher precedence that any other connective. In the following example, we illustrate how a TQU is formed, the composition of TQUs and what will constitute the final query $QUECA(Q)$.

Example 11 *(example 10 continued) Using the new notation for the third case of Example 10, we would have:*

$$TQUs = \overbrace{\overbrace{P(u,v) \vee \neg Q(u,z)}^{D_1} : \overbrace{P(u,v) \wedge P(u,v)}^{E_1} \bullet \overbrace{residues(P(u,v))}^{R_1}}$$
$$\vee$$
$$\underbrace{P(u,v) \vee \neg Q(u,z)}_{D_2} : \underbrace{P(u,v) \wedge \forall z \neg Q(u,z)}_{E_2} \bullet \underbrace{residues(\neg Q(u,z))}_{R_2} ,$$

which can be rewritten as $TQUs = D_1 : E_1 \bullet R_1 \ \vee \ D_2 : E_2 \bullet R_2$*, and/or, equivalently,* $TQUs = TQU_1 \vee TQU_2$*. The final query should eventually be formed by the* E_i*'s, i.e.* $QUECA(Q) = \bigvee_i E_i$*, once all the residues have been resolved.* ■

As may be expected, the remaining Pending Residue Lists, R_1 and R_2 in Example 11 (and R_i's in general), must again be resolved against their corresponding E_i's and D_i's in an iterative fashion. This iteration will stop when we run out of pending residues, that is, $R_i = \varnothing$ for every i. The naive way to detect this is by observing when the $\bullet$ symbol reaches the right end of a given TQU, that is, all its pending residues have been resolved.

The critical step is then, determining when a residue should be added to the query and when its information is already in it, i.e. it should be discarded. It is easy to see that when $E \vDash \phi_1$[7] or $D \vDash \phi_1$, then ϕ_1 can be discarded. If either condition is not satisfied, the residue must be included in the query.[8] In Example 11, we have that $D_1 \vDash residues(P(u))$ (see the residues for the third case in Example 9), thus they can be discarded and the iteration would have ended for TQU_1. This is the semantic result we want to obtain via syntactical means. The usual way to attain this is by unification.

In our case we will define a sort of one way unification in which only certain types of variables will be involved: New Variables in a TQU and Free Variables in a Residue.

---

[7] The sufficient condition is that every term in ϕ_1 belongs to E.

[8] We will see that sometimes only part of the residue must be included.

Definition 3.3. *Given a $TQU = D : E \bullet R$ and a query Q, the set of New Variables in the TQU, denoted by $newVar(TQU)$, is defined as the set of universally quantified variables in $Var(E) \setminus Var(Q)$.* ■

Definition 3.4. *Given a $TQU = D : E \bullet R$ and a residue $\phi \in R$, the set of Free Variables in the Residue, denoted by $freeVar(\phi)$, is defined as the set of universally quantified variables in $Var(\phi) \setminus Var(E)$.* ■

Because D in a TQU consists of a recently resolved residue, it also behaves as one and has *Free Variables* in the sense of Definition 3.4. For instance, in Example 11, we have $newVar(TQU_2) = \{z\}$ and $freeVar(D_1) = \{z\}$. From these definitions it is clear that we can substitute a *freeVar* for any other variable because they occur no where else than in that residue.

Having identified the variables that we will use in the unification process, we can now formally define the meaning of *the information of a residue already in a TQU*. This notion will enable us to determine when a residue must me discarded or not.

Definition 3.5. *We will say the information of a residue $\phi = l_1 \vee \cdots \vee l_n$ is already in a $TQU = D : E \bullet R$, and will write $\phi \,\widetilde{\in}\, D : E$, whenever there exists a substitution $\sigma : freeVar(\phi) \to newVar(TQU) \cup freeVar(D)$, such that $\phi\sigma \in D$ or, for all i, $l_i\sigma \in E$. Otherwise we will write $\phi \,\widetilde{\notin}\, D : E$. However, in case only some $l_i\sigma \in E$, we will say the information of a residue is already partially in a TQU, and we will write $\phi \;{}_P\widetilde{\in}_\sigma\; D : E$* ■

Example 12 *Let us recall the third case of Example 9, in which we had:*

$$\textit{Query}: \;\; 3. \;\; P(u,v) \qquad \textit{Residues}: \;\; \forall z \; (P(u,v) \vee \neg Q(u,z)) \;\; .$$

This way we build TQUs as

$$TQUs = \varnothing : P(u,v) \bullet \forall z \; (P(u,v) \vee \neg Q(u,z)) \;\; .$$

Suppose, as well, that we have that $residues(\neg Q(x,y)) = P(x,y)$. *Now, we must check if the first (clausal) residue's information is already in* $[\varnothing : P(u,v)]$[9] *or not. Clearly we have that* $\forall z\ (P(u,v) \vee \neg Q(u,z)) \ \widetilde{\notin}\ [\varnothing : P(u,v)]$. *However, we do have that* $\forall z\ (P(u,v) \vee \neg Q(u,z)) \ {}_P\widetilde{\in}_\sigma\ [\varnothing : P(u,v)]$ *for* $\sigma = \varepsilon$ *(the identity, i.e.* $P(u,v)\sigma \in P(u,v)$*). Thus, the literal* $P(u,v)$ *in the residue can be discarded, and we must keep an instance of* $P(u,v)$. *The other member of the residue,* $\forall \neg Q(u,z)$, *is appended to a copy of* $P(u,v)$, *and its residues appended to the Pending Residue List. So we have:*

$$\begin{aligned} TQUs = \quad & P(u,v) \vee \neg Q(u,z) : P(u,v) \bullet \varnothing \\ & \qquad\qquad \vee \\ & P(u,v) \vee \neg Q(u,z) : P(u,v) \wedge \forall z \neg Q(u,z) \bullet P(u,z) \ . \end{aligned}$$

We can see how the iteration has reached an end for the first member of $TQUs$. *On the other hand, iterations must continue for the second member of the disjunction. We have that* $P(u,z) \ \widetilde{\notin}\ [P(u,v) \vee \neg Q(u,z) : P(u,v) \wedge \forall z \neg Q(u,z)]$, *so the residue must be added to* $P(u,v) \wedge \forall z \neg Q(u,z)$, *and its residues to the Pending Residue List. We then have:*

$$\begin{aligned} TQUs = \quad & P(u,v) \vee \neg Q(u,z) : P(u,v) \bullet \varnothing \\ & \qquad\qquad \vee \\ & P(u,z) : P(u,v) \wedge \forall z \neg Q(u,z) \wedge P(u,z) \bullet \forall w (P(u,z) \vee \neg Q(u,w)) \ . \end{aligned}$$

Now we have that $\forall w(P(u,z) \vee \neg Q(u,w)) \ \widetilde{\in}\ [P(u,z) : P(u,v) \wedge \forall z \neg Q(u,z) \wedge P(u,z)]$ *for* $\sigma : w \to z$. *Thus the residue is discarded and iteration terminates with:*

$$\begin{aligned} TQUs = \quad & P(u,v) \vee \neg Q(u,z) : P(u,v) \bullet \varnothing \\ & \qquad\qquad \vee \\ & P(u,z) : P(u,v) \wedge \forall z \neg Q(u,z) \wedge P(u,z) \bullet \varnothing \ . \end{aligned}$$

At this point, all the residues have been resolved and we have that the final query for consistent answers is $QUECA = P(u,v) \vee (P(u,v) \wedge \forall z(\neg Q(u,z) \wedge P(u,z))$ ■

[9] *The square brackets are only used for clarity reasons.*

We can summarize the general procedure illustrated in Example 12 as follows. When verifying whether to add a residue $\phi = l_1 \vee \cdots \vee l_m$ to a query, if $\phi \,\widetilde{\in}\, D : E$, then ϕ is discarded. Otherwise, it must be added to E and one of the mentioned split operations must take place. However, if $\phi \;{}_P\widetilde{\in}_\theta\; D : E$, then we must keep a copy of $D : E \bullet R$ adding ϕ to D; and for all the cases in which $l_i\theta \notin E$, $l_i\theta$ must be appended to a copy E_i of E, its residues must be added at the end of a copy R_i of R and D_i must be replaced by ϕ. The procedure just described is formalized as Algorithm 2.

Algorithm 2 Generate a QUEry for Consistent Answers for a literal l: $QUECA(l)$

Require: Algorithm 1 has been executed.

Ensure: $QUECA(l)$ contains the expected results.

1: $QUECA(l) := \varnothing$
2: $TQUs := \varnothing : l \bullet residues(l)$
3: **while** $TQUs \neq \varnothing$ **do**
4: select(extract) first TQU from $TQUs \mapsto (D\colon E \bullet R)$
5: **if** $R = \varnothing$ **then**
6: $QUECA(l) := QUECA(l) \vee E$
7: **else**
8: select(extract) first residue(clause) from $R \mapsto \phi$ $\{\phi = l_1 \vee \cdots \vee l_m\}$
9: **if** $\phi \,\widetilde{\in}\, D : E$ **then**
10: $TQUs = D\colon E \bullet R \vee TQUs$
11: **else**
12: **if** $\phi \;{}_P\widetilde{\in}_\theta\; D : E$ **then**
13: $append(D, \phi) \mapsto D_0$
14: $E_0 := E$
15: $R_0 := R$
16: **else**
17: $\theta = \varepsilon$ (identity)
18: **end if**
19: **for all** $i \in [1, m]$ **do**
20: **if** $l_i\theta \notin E$ **then**
21: $D_i := \phi$
22: $E_i := E \wedge l_i\theta$
23: $R_i := R \wedge residues(l_i\theta)$
24: **else**
25: Do nothing
26: **end if**
27: **end for**
28: $TQUs := \bigvee_{i=0}^{m}(D_i\colon E_i \bullet R_i) \vee TQUs$
29: **end if**
30: **end if**
31: **end while**

Example 13 *(example 4 continued) We will show how Algorithm 2 computes $QUECA(P(u,v))$, which is equivalent to $\mathrm{T}_2(P(u,v))$, being 2 the finiteness point. To do so, we will show the state of variables QUECA and TQUs at the beginning of every iteration of the while loop in Algorithm 2.*

1^{st} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \varnothing \\ TQUs = &\ \varnothing : P(u,v) \bullet (R(u,v)) \wedge (\neg P(u,z) \vee v = z) \ , \end{aligned}$$

2^{nd} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \varnothing \\ TQUs = &\ R(u,v) : P(u,v) \wedge R(u,v) \bullet (\neg P(u,z) \vee v = z) \wedge (P(u,v)) \ , \end{aligned}$$

3^{rd} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \varnothing \\ TQUs = &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \bullet \\ &\ (P(u,v)) \wedge (\neg R(u,z))] \vee \\ &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))] \ , \end{aligned}$$

4^{th} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \varnothing \\ TQUs = &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \bullet \\ &\ (\neg R(u,z))] \vee \\ &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))] \ , \end{aligned}$$

5^{th} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \varnothing \\ TQUs = &\ [\neg R(u,z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z) \bullet \\ &\ (\neg P(u,z))] \vee \\ &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))] \ , \end{aligned}$$

6^{th} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \varnothing \\ TQUs = &\ [\neg R(u,z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z) \bullet] \vee \\ &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))] \ , \end{aligned}$$

7^{th} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ [P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)] \\ TQUs = &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))] \ , \end{aligned}$$

8^{th} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ [P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)] \\ TQUs = &\ [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet] \ , \end{aligned}$$

9^{th} *iteration*

$$\begin{aligned} QUECA(P(u,v)) = &\ \forall z \, [[P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)] \vee \\ &\ [P(u,v) \wedge R(u,v) \wedge v = z]] \ . \end{aligned}$$

By rearranging the result by hand, we obtain

$$\begin{aligned} QUECA(P(u,v)) = &P(u,v) \wedge R(u,v) \wedge \forall z \, [(\neg P(u,z) \wedge \neg R(u,z)) \vee v = z] \ , \\ QUECA(P(u,v)) = &P(u,v) \wedge R(u,v) \wedge \forall z \, [(\neg P(u,z) \vee v = z) \wedge \\ &(\neg R(u,z) \vee v = z)] \ . \end{aligned}$$

Notice, how the constraints are propagated towards the related literals according to the nature of IC. In this case the functional dependency of the second argument of $P/2$ has generated a functional dependency for the second argument of $R/2$. This example was shown to be non terminating for T_ω, but is now solved by QUECA. ■

In the previous example we can also see how the $\bullet$ symbol works as a separator between the residues that have been included in the final query and those that are to be resolved. It graphically shows when a *TQU* is ready to be included in *QUECA*, this is when the $\bullet$ reaches the end of R, put in other words, when no residues are left to be resolved.

IV. SOLUTION ANALYSIS

This section describes the properties of *QUECA*. We will begin by restricting the ICs covered by the solution to a set that happens to be larger than that covered by operator T (Arenas et al., 1999). Then we will analyze the runtime complexity of both algorithms presented in this thesis, and we will prove that termination is guaranteed for the restricted class of ICs. Next, we will prove that the soundness and completeness results of T can be extended to *QUECA*. Finally, we will show the class of queries for which *QUECA* is intended to be applied.

4.1 Restriction on Integrity Constraints

As was informally stated in Section 2.3.1, ICs should not generate data nor do they represent specific data. Thus, they are not *fact-oriented* in the sense of Definition 4.1.

Definition 4.1 (Fact-Oriented Constraints). *A set of integrity constraints, IC, is fact-oriented if there is a tuple $\bar{a}$ and a literal name L, such that $IC \vDash L(\bar{a})$.* ■

Usually ICs are not fact-oriented. Having set them aside, the following definition describes the class of ICs for which *QUECA* behaves properly.

Definition 4.2 (Binary Integrity Constraint - BIC). *A binary integrity constraint (BIC) is a range restricted (Nicolas, 1982) denial of the form (2.2), that is, $\forall\ (\leftarrow l_1(\bar{x}_1) \wedge l_2(\bar{x}_2) \wedge \psi(\bar{x}))$, where $\forall$ represents the universal closure of the formula, l_1 and l_2 are database literals and ψ is a formula that only contains built-in predicates.* ■

BICs account for most of the integrity constraints found in relational databases. In this class we find functional dependencies, inclusion dependencies and symmetry constraints. Furthermore, as a particular case of BICs, we obtain *unary integrity constraints* (UICs), which have just one database literal and possibly a formula with built-in predicates. UICs include domain and range constraints. In

consequence, by considering BICs (and UICs), we are covering most of the static integrity constraints found in traditional relational databases, excluding (existential) referential ICs, transitivity constraints, and possibly other constraints that might be better expressed as rules or views at the application layer.

4.2 Algorithm Runtime Complexity

Theorem 4.1. *The runtime complexity for the worst case of Algorithm 1, which computes residues for literal-names in a set of integrity constraints, is* $O(n^2)$*, where* n *represents the number of* ICs*.*

Proof. The numbers on the left represent the corresponding line numbers in the algorithm.

1–1: $F_1(n) = (c_1 - c_3)n + c_2n^2$.

Read the ICs and insert them into a list L_{IC} : c_1n. Next we must build a list of distinct literal names appearing in the integrity constraints L_P. Because we accept only BICs, the maximum literal names per constraint is 2, so in the worst case (when they are all different) we have $2n$ literal names. The number of comparisons performed is $\sum_{i=1}^{2n}(i-1) = c_2n^2 - c_3n$. So $F_1(n) = c_1n + c_2n^2 - c_3n = (c_1 - c_3)n + c_2n^2$.

2–13: $F_2(n) = c_4kn^2$.

By considering that $|L_P|$ is bound by $2n$, we have that the loop executed between lines 5–10 is repeated a maximum of $2n^2$ times. The maximum number of comparisons performed in line 5 is k, where k is the maximum number of terms allowed per integrity constraint, so $F_3(n) = c_4kn^2$.

14–21: $F_3(n) = c_5(k-1)^2n^2 - c_6(k-1)^2n$.

If all literal names are different we have 1 residue for each, and if they are all equal we have $2n$ residues for that single one. So, either case, the redundancy elimination process must be executed $2n$ times. This process involves checking that every literal belongs to a single residue already in the final list for a given substitution and if it succeeds, the reversal must be performed. Now the worst

case would be that no two residues are redundant and the checking process fails in the last step, that is, the last term of the reversal process , so all the possible checks must be made. So for a residue of $(k-1)$ terms we would have to do $(k-1)^2$ checks to determine if it is redundant or not with another residue of $(k-1)$ terms. This can be formulated as $\sum_{i=1}^{2n}(c(i-1)(k-1)^2)$. So $F_3(n) = c_5(k-1)^2n^2 - c_6(k-1)^2n$.

The resulting runtime complexity of Algorithm 1 is $F(n) = F_1(n) + F_2(n) + F_3(n)$. That is, $F(n) = (c_1 - c_3)n + c_2n^2 + c_4kn^2 + c_5(k-1)^2n^2 - c_6(k-1)^2n$. Grouping and rearranging terms we have:

$$F(n) = \left(c_5(k-1)^2 + c_4k + c_2\right)n^2 - \left(c_6(k-1)^2 + c_3 - c_1\right)n \ .$$

Because we are dealing with real world databases, the maximum number of terms in an arbitrary integrity constraint, k, is bounded by a sufficiently large constant (i.e. because we only consider BICs it would usually be 3). This way we have that the runtime complexity for the residue calculations is quadratic, that is:

$$F(n) = O(n^2) \ .$$

■

Before analysing the runtime complexity of Algorithm 2 we must define a new notion related to the critical decision point in the algorithm's execution. This is what will be called a *failure*.

Definition 4.3 (Failure). *A failure in Algorithm 2 occurs when the condition in line 9 fails and the one in line 20 succeeds at least once. Otherwise it is called a success.* ■

Consequently, a failure increases the number of terms in *TQUs* by a maximum of $k-1$, being k the number of terms per integrity constraint. On the other hand, a success decreases the number of terms in *TQUs* by 1. The maximum number of failures to occur must now be bounded to be able to calculate the runtime complexity of Algorithm 2.

Lemma 4.1. *Restricting IC to BICs with maximum of k terms each, and being* $|IC| = n$*, the maximum number of failures in Algorithm 2 is upper bounded by* $\frac{k^{4n+1}-1}{k-1}$.

Proof. Algorithm $QUECA(Q)$'s execution can be represented as a tree in which Q is the root and the literals from the first residue are its sons, the literals from the second its grandsons and so forth. Following this interpretation, what delivers children is a failure, so the number of inner nodes of the tree indicates how many failures have occurred.

We know that by restricting IC to BICs we have a maximum of $2n$ different literal names each having a maximum of 1 literal name per residue. Because of the condition in line 24 and the fact that ICs are range restricted denials (2.2), we can have a maximum of $2n$ predicates or negated predicates in any path from root to leaf, the rest are built-ins. These built-ins are either part of the path or will be when the pending residues are resolved. Only literal names generate residues, so there is a maximum of $(2n)^2$ residues that will supply built-ins ($2n$ residues per each of the $2n$ literal names). However, because of the condition in line 13, once the first $2n$ built-ins have been added, all the residues are stored in D, so no more failures will occur. Thus we have that the maximum depth of the tree would then be $4n$. An consequently the number of failures (inner nodes) is upper bound by $\sum_{j=0}^{4n} k^j = \frac{k^{4n+1}-1}{k-1}$. ■

Having an upper bound for failures it is possible to calculate the runtime complexity of Algorithm 2.

Theorem 4.2. *Restricting IC to BICs with a maximum of k terms each, the runtime complexity for the worst case of Algorithm 2, which computes QUECAs for literal-names in a set of integrity constraints, is* $O(nk^{8n})$*, where n represents the number of* ICs.

Proof. Before starting this calculation we must introduce a new useful quantity: the maximum number of residues associated to any literal r. It is easy to see that $0 \leqslant r \leqslant 2 \cdot n$. Again, the numbers on the left represent the corresponding line numbers in the algorithm.

1–2: $F_1(n) = c_1 + c_2 r$.

The runtime complexity of line 2 is clearly linear wrt the number of residues associated to the literal.

3–31: $F_2^{(f)}(n) = c_3(k-2)(r-1)f^2 + c_3 k(r-1)f + r$.

Clearly the availability of terms in $TQUs$ determines how many executions of the while loop will take place. Furthermore, the number of executions is really bound by the number of pending residues at the right of $\bullet$ in every term appearing in $TQUs$. The upper bound of this number will be given by the maximum number of terms in $TQUs$ times the maximum number of residues one of them has at a given moment. So, once the algorithm is initialized a first check takes place in line 5. We will assume the worst case, which is that this condition is never met and so the algorithm never stops. Next, if the residue checks succeed in line 9, no new residues are added and the number of residues of the first term of $TQUs$ is reduced by one, so whatever the upper bound we picked it still remains as such. On the other hand if a failure occurs the upper bound must be modified to include at most $(k-2)$ new terms, and only one of them adds at most $r-1$ new residues (because we are dealing with BICs the rest of the residue are only built-ins which do not add residues so now they have one residue less). Now we are ready to calculate these upper bounds. In the beginning we have $TQUs = \varnothing : l \bullet residues(l)$, that is, only one term in $TQUs$ ($|TQUs| = 1$) and a maximum of r residues yet to be resolved in that term.

$$F_2^{(0)}(n) = 1 \cdot r \ .$$

Next, a *failure* occurred, so now the maximum number of terms is $1 + (k-2) = (k-1)$, with the first term having a maximum of $r + (r-1) = 2r - 1$ residues.

This procedure can be iterated to obtain

$$
\begin{aligned}
F_2^{(1)}(n) =& (2r-1) + (k-2)(r-1) \ . \\
F_2^{(2)}(n) =& 1 \cdot (2r-1+r-1) + (k-2)(2r-2) + (k-2)(r-1) \ . \\
=& (3r-2) + (k-2)(2r-2) + (k-2)(r-1) \ . \\
F_2^{(3)}(n) =& 1 \cdot (3r-2+r-1) + (k-2)(3r-3) + (k-2)(2r-2) + \\
& (k-2)(r-1) \ . \\
=& (4r-3) + (k-2)(3r-3) + (k-2)(2r-2) + (k-2)(r-1) \ . \\
F_2^{(4)}(n) =& ...
\end{aligned}
$$

This way we arrive to the following equation:

$$F_2^{(f)}(n) = (f+1)r - f + (k-2)(r-1) \sum_{i=1}^{f} i \ ,$$

for $f \geqslant 0$. By eliminating the sum and rearranging terms we have $F_2^{(f)}(n) = (r-1)f + r + 1/2(k-2)(r-1)f^2 + 1/2(k-2)(r-1)f$, which is then

$$F_2^{(f)}(n) = c_3(k-2)(r-1)f^2 + c_3k(r-1)f + r \ .$$

In this equation f represents the maximum number of *failures* of the algorithm until completion, k is the maximum number of literals per integrity constraint and $r \leqslant k \cdot n$ is the maximum number of residues a literal may have for a given set of n ICs. In this case constants were disposed because they are not relevant and only clutter the calculation.

The resulting runtime complexity of Algorithm 2 is $F(n) = F_1(n) + F_2^{(f)}(n)$. That is,

$$F(n) = c_1 + c_2r + c_3(k-2)(r-1)f^2 + c_3k(r-1)f + r \ ,$$

which clearly depends on k, r and f. Because r is upper bound by $k \cdot n$ we have that it now depends on k, n and f. Next, by using the same argument presented in the

runtime complexity calculation of $residues(l)$ (k is upper bound by a constant) we can discard k and this way the runtime complexity of $QUECA(l)$ for one literal is

$$F(n) = O(nf^2) \ ,$$

where f represents the maximum number of *failures* of the algorithm until completion and n is the number of integrity constraints in IC. But from Lemma 4.1 we know that the number of *failures* is bound by $O(k^{4n})$ so the runtime complexity for $QUECA(l)$ is

$$F(n) = O(nk^{8n}) \ .$$

■

Despite this complexity result, we will see in Chapter 5 (see its introduction and Section 5.3.2) that the process of computing *QUECAs* (and residues) is done beforehand (i.e. offline), so it should not affect the performance from a user's point of view. This means, that once the *QUECAs* have been computed for a given set of integrity constraints IC (associated to some database), the user can pose the new queries (*QUECAs*) directly to the database, regardless of whether the database contents have changed or not. Of course, we rely on the fact that integrity constraints do not change often.

4.3 Termination

Recall from Section 2.1 that the set of integrity constraints IC is finite. Thus, termination is guaranteed for Algorithm 1. For Algorithm 2 we have the following:

Theorem 4.3 (Termination). *Given a finite set of non fact-oriented BICs, Algorithm 2, which computes QUECAs for literal names in a set of integrity constraints, terminates in a finite number of steps.*

Proof. From Lemma 4.1 ■

The termination property is based on the fact that by restricting execution to BICs only, residues contain one literal name at most, which in the worst case generates an infinite sequence of single literals. The infiniteness of this sequence is then limited by the condition in line 12 of Algorithm 2 and the fact that we only consider range–restricted ICs of the form (2.2), conditions which ensure that at a given point, pending residues add no new information to the resulting query, thus being discarded. Notice that this result extends the termination results presented in (Arenas et al., 1999), where semantic termination was only ensured for uniform binary constraints.

4.4 Soundness and Completeness

It is possible to prove that the *QUECA* algorithm's execution can be put in correspondence with the iterative application of operator T until the point where *QUECA* stops (see Examples 4 and 13). At that point we obtain a corresponding semantical termination point for T. The main difference is that, while T would perform split operations and add residues to the pending list (for the whole set of residues) whenever *at least one* of the residues adds new information to the resulting query, *QUECA* does this on a per-residue basis. This eliminates residues one by one, thus obtaining a much more efficient query (see the difference between $\mathrm{T}_3(P(u,v))$ in Example 4 and $QUECA(P(u,v))$ in Example 13). Having mapped *QUECA*'s execution to that of T, we may take advantage of soundness and completeness results for T.

In order to achieve soundness in the execution of *QUECA*, it must first be proved that the notion presented in Definition 3.5 implies logical consequence.

Lemma 4.2. *Given a TQU* $D : E \bullet R$. *Let* $\phi = l_1 \vee \cdots \vee l_n \in R$ *be a residue. If* $\phi \,\widetilde{\in}\, D : E$ *then either* $E \vDash \phi$ *or* $\bigvee E_i \vDash \phi$ *for all* i *such that* $D_i \in D$.

Proof. For a substitution σ as in Definition 3.5 we have two cases, when for all i, $l_i\sigma \in E$ and when $\phi\sigma \in D$.

a) for all i, $l_i\sigma \in E$. Since σ substitutes variables occurring nowhere else but in ϕ for variables which are universally quantified in E, and because we know that $l_1 \wedge l_2 \vDash l_1 \vee l_2$ we have that $E \vDash \phi$.

b) $\phi\sigma \in D$. We know that D contains the last residue that generated a split operation plus all those residues φ, if any, such that $\varphi \ {}_P\widetilde{\in}_\theta \ D : E$ for the case that no literal is added to its E (line 13). We also know that all the elements from *TQUs* whose D share elements have been involved in a split operation, so they have a common stem e and are just differentiated by the last portion of E. This way $\bigvee E_i$ for all i such that $D_i \in D$ can be rewritten as $e \wedge D$. Since σ substitutes variables occurring nowhere else but in ϕ for variables which are universally quantified in D we have that $\phi\sigma \in D \Rightarrow D \vDash \phi$. Consequently $e \wedge D \vDash \phi$, which implies that $\bigvee E_i \vDash \phi$ for all i such that $D_i \in D$.

■

Now it is possible to prove the soundness of Algorithm 2. To do so, the execution of *QUECA* will be mapped to that of T in order to take advantage of T's soundness and completeness results.

Definition 4.4. *Residues can be classified into generations according to the following inductive definition:*

- *Given a query Q, residues(Q) are first generation residues.*
- *Residues of n^{th} generation residues are $(n+1)^{th}$ generation residues.*

■

Lemma 4.3. *Restricting IC to BICs , the execution of Algorithm 2 for a query Q delivers QUECA(Q), such that QUECA(Q) $\equiv \mathrm{T}_n(Q)$, being n is the finiteness point as defined in (Arenas et al., 1999).*

Proof. The execution of Algorithm 2 follows a DFS strategy in the sense that it systematically selects the first element from $TQUs$ to resolve its first pending residue. However, this is not essential for sound execution because *all* branches of the tree *must* be followed and completely evaluated. This way the elements of $TQUs$ can be selected in *any* order without compromising soundness.

Having said this, and taking into account the residue generations presented in Definition 4.4, the elements of $TQUs$ can be selected so all the elements having first generation residues are selected in first place. Then, when no first generation residues are left in any element belonging to $TQUs$, elements having second generation residues are selected. This choosing scheme continues until complete execution. As auxiliary notation, when all k generation residues have been resolved, it will be be written as $QUECA(\cdot)^k$, being $QUECA$ formed by the disjunction of all the E's present at that moment in $TQUs$. This way we can map $QUECA$'s execution to that of operator T, presented in (Arenas et al., 1999), in the following way:

$$
\begin{aligned}
&\mathrm{T}_0(Q) = Q \equiv QUECA(Q)^0 \ , \\
&\mathrm{T}_1(Q) = Q \wedge \mathit{residues}(Q) \equiv QUECA(Q)^1 \ , \\
&\dots \\
&\mathrm{T}_n(Q) = \dots \equiv QUECA(Q)^n = QUECA(Q) \ \mathit{for\ some}\ n \ .
\end{aligned}
$$

Because only BICs are being considered, stopping is guaranteed at some step, say n (see Theorem 4.3). Due to the equivalence shown above, the main difference between operator T and $QUECA$ would be that $QUECA$ is guaranteed to stop for a greater set of ICs.

Although not all residues are added to $QUECA$, Lemma 4.2 states that the information that the residues that were discarded before reaching step n represent, is already included in $QUECA$; thus is irrelevant. This information discarding is what makes the algorithm stop. However, this stopping condition (i.e. lines 9–18, 20 and 24–26) can be stripped off the algorithm, and the resulting formula at step n will be logically equivalent to the original one; it will also be equivalent to T_n. Even more, the execution of $QUECA$ could then be taken one step further, to when all $n+1$ generation residues are resolved. This step is not reached originally (with the

stopping conditions in place) because all the information added in it is redundant, that is $QUECA(Q)^n \equiv QUECA(Q)^{n+1} \equiv QUECA(Q)$ (see Lemma 4.2).

Consequently, by transitivity and the invariant described earlier, it is proved that $\mathrm{T}_n(Q) \equiv QUECA(Q)$ and $\mathrm{T}_n(Q) \equiv \mathrm{T}_{n+1}(Q)$, so n is the finiteness point as defined in (Arenas et al., 1999). ■

Having mapped the execution of *QUECA* to that of operator T, it is now possible to take advantage of the soundness and completeness results for Tpresented in (Arenas et al., 1999).

Theorem 4.4 (Soundness). *Let r be a database instance, IC a set of binary integrity constraints and $Q(\bar{x})$ a literal query, such that $r \models QUECA(Q)(\bar{t})$. If Q is universal or non-universal, but domain independent, then $\bar{t}$ is a consistent answer to Q in r, that is, $r \models_c Q(\bar{t})$.*

Proof. From Lemma 4.3 and Theorem 2 in (Arenas et al., 1999). ■

Theorem 4.5 (Completeness). *Let r be a database instance and IC a set of non fact-oriented binary integrity constraints, then for every ground literal $l(\bar{t})$, if $r \models_c l(\bar{t})$, then $r \models QUECA(l)(\bar{t})$.*

Proof. From Lemma 4.3 and Theorem 4 in (Arenas et al., 1999). ■

As will be shown in Section 4.5, the queries covered by these results are literals and conjunctions of literals free of existential quantifiers.

4.5 Restrictions on Queries

We will now illustrate the need to restrict the class of queries supported by the algorithms presented in this thesis. Initially, given a query Q, we want to compute a first order query $QUECA(Q)$, which will deliver consistent answers only. The query Q must then follow a simple rule: qualify to be the seed of Algorithms 1 and 2. Because both algorithms operate on integrity constraints that are written in

terms of predicates (tables), queries must also be based on database predicates, so they should be literals of the form $P(t_1, \ldots, t_n)$ or $\neg P(t_1, \ldots, t_n)$, where P represents a table name and each t_i an attribute of P.

QUECA can also handle conjunctive queries by distributing over the conjunction. That is:

$$QUECA(l_1 \wedge \ldots \wedge l_n) \equiv QUECA(l_1) \wedge \ldots \wedge QUECA(l_n) \ .$$

In case a given l_i involves a built-in predicate, *QUECA* does not operate on it and simply has no effect (e.g. $QUECA(x \geq 7) \equiv x \geq 7$). This distribution property makes it possible to apply *QUECA* to (partially) ground queries as well. A (partially) ground query such as $P(t_1, \ldots, c, \ldots, t_n)$, is really treated as $P(t_1, \ldots, t_i, \ldots, t_n) \wedge t_i = c$. So $QUECA(P(t_1, \ldots, c, \ldots, t_n))$ would really become $QUECA(P(t_1, \ldots, t_i, \ldots, t_n)) \wedge t_i = c$. All this process is invisible to the user, making the program friendlier and easier to use.

It is easy to see that a ground query Q will generate a $QUECA(Q)$ whose answer is *true*, i.e. the tuple belongs to every repair, or *false*, i.e. the tuple does not belong to every repair.[10] On the other hand, if at least one of the t_i's is not ground, then a set of consistent values for the non-ground t_i's, if any, shall be returned as answer. If no values satisfy the query, an empty set is returned.

Example 14 *Consider the set of integrity constraints (IC) presented in Example 5:*

$$\Leftarrow P(u, v) \wedge \neg R(u, v) \ ,$$
$$\Leftarrow \neg P(u, v) \wedge R(u, v) \ ,$$
$$\Leftarrow P(u, v) \wedge P(u, z) \wedge y \neq z \ .$$

The following database instance r, which violates IC,

[10] In the implementation we will use the traditional Prolog values `yes` and `no`.

P	
a	1
b	2
c	8
d	9

R	
a	1
a	2
b	2
c	8

has four repairs:

r' :

P	
a	1
b	2
c	8

R	
a	1
b	2
c	8

r'' :

P	
a	2
b	2
c	8

R	
a	2
b	2
c	8

r''' :

P	
a	1
b	2
c	8
d	9

R	
a	1
b	2
c	8
d	9

r'''' :

P	
a	2
b	2
c	8
d	9

R	
a	2
b	2
c	8
d	9

Using the QUECAs calculated in Example 13, we may pose the queries:

Query	*Answer*
$QUECA(P(x, y))$	$[b, 2]$; $[c, 8]$
$QUECA(P(x, 2))$	$[b]$
$QUECA(P(a, 1))$	*false*
$QUECA(P(x, y) \wedge y > 7)$	$[c]$
$QUECA(P(b, 2) \wedge R(c, 8))$	*true*

■

All the results shown previously (e.g. termination, soundness, etc.) are applicable to queries that are conjunctions of literals without existential quantifiers. Therefore, we may perform join operations between two *QUECA*s by simply querying the database with the conjunction of their *QUECA*s, taking care to use the proper variable names where we want the join to be performed.

Other types of queries, specifically those involving disjunction and existential quantifiers, may not be covered by this solution. We will now show that in general, completeness is not obtained for queries that are not conjunctions of literals.

Example 15 *(Existential query) Consider the query* $\exists x P(a, x)$. *The database instance* r *of Example 14 has* true *as a consistent answer because in every repair exists a tuple with* a *as its first argument. However, it is easy to see that* $QUECA(\exists x P(a, x))$ *is logically equivalent to* $\exists x(P(u, x) \wedge R(u, x) \wedge \forall z\,[(\neg P(u, z) \vee x = z) \wedge (\neg R(u, z) \vee x = z)])$. *Thus, we have that* $r \not\models QUECA(\exists x P(a, x))$, *and the consistent answer* true *is not captured by* $QUECA$. ■

Example 16 *(Disjunctive query) Consider the query* $P(a, 1) \vee P(a, 2)$. *When posed to the database instance* r *of Example 14, it should return* true *as a consistent answer because in every repair it succeeds. However, the query* $QUECA(P(a, 1)) \vee QUECA(P(a, 2))$ *returns* false. ■

This last example shows that the $QUECA$ of a disjunction is not logically equivalent to the disjunction of the $QUECA$s. In (Arenas et al., 2000) a methodology to retrieve consistent answers has been proposed that can be applied to existential and disjunctive queries.

V. IMPLEMENTATION

To achieve the goal of generating queries whose answers correspond to the consistent information stored in a database, we need a common framework for data, rules, queries and integrity constraints to be able to perform operations on them and elaborate the mentioned queries. Logic programming languages provide this framework and XSB (Sagonas et al., 1994) seems an adequate candidate. Generally speaking we prefer a LP language because the algorithms described in this thesis need the ability to perform unifications, substitutions and detecting subsumption.

Some of the main characteristics that make XSB suitable for this application are:

Tabling. Dramatically improves performance by being able to store intermediate results efficiently and thus avoid redundant subcomputation. If queries are to be posed directly from the XSB interpreter, tables can be also used as a cache for the calculated queries which would have been generated at compile time. These queries correspond to those which deliver consistent answers only, according to what was presented in Section 2.1.

Relational DBMS interface. XSB comes with an a general ODBC interface which allows data stored in the databases to be accessed from XSB's environment as thought they existed as facts. For more details on features and operation see (Sagonas et al., 1999).

Foreign language interface. XSB may be accessed from different languages, including C and Visual Basic, which enables building powerful applications which would use XSB as a subroutine processor, or to run as a complete initialization module.

Multiplatform. XSB currently supports a number of different platforms which makes it specially convenient when working from different locations. For more details see (Sagonas et al., 1994).

Perhaps what makes XSB a better candidate that any other LP language is its *tabling* capabilities that improve its efficiency over other systems that would, for example, have to recalculate the residues every time they are needed by Algorithm 2. Without this tabling ability, whenever the user posed a query to the database, its corresponding *QUECA* should be calculated on the fly. This would diminish performance notoriously because of its complexity (see Theorem 4.2). Consequently, when querying a database, the high runtime complexity of the algorithm does not affect the performance of the system as an interactive database querying system. To achieve complete persistency of the computed *QUECAs* we should use XSB's I/O facilities to write the results to a file, so they (*QUECAs*) could be used even when the database contents have been updated.

5.1 Program Overview

Because the set of integrity constraints, *IC*, seldom changes for a given database, we can compute the *QUECAs* beforehand (i.e. offline). Therefore, when a user poses a query Q to a database, all the system does is, fetch $QUECA(Q)$, pose it to the database instance and return its answers, which correspond to the consistent answers of Q. Notice that the database contents may be updated and the *QUECAs* still serve their purpose. This way we avoid possible performance problems derived from the complexity results obtained for Algorithm 2. The reason why the computation may be done offline is that, as could be noted in Chapter 3, the only input for the algorithms presented in this thesis is *IC*. Therefore, *IC* must be defined in an appropriate manner and somewhere the system is able to find it.

Next, when evaluating an expression (i.e. posing a query to a database), we would like to take advantage of XSB's ODBC interface and access the data stored in the database directly. To be able to do so we must take care to adjust the database schema to a certain framework that will allow the system to operate properly. These two issues are described next.

5.1.1 Integrity Constraints

Integrity constraints are to be defined in a file named `ics` in conformity with the syntax of (2.2), that is:

```
<- [ ...  denials ...  ].                                    (5.1)
```

The left arrow '`<-`' and square brackets '`[ ]`' are necessary for correct parsing into the program. Also, each constraint must be ended with a period '.' and on a separate line. Inside the brackets, as in traditional Prolog lists, terms must be separated by commas which represent conjunctions (see Example 17). The built-in operators allowed in an integrity constraint and their syntactical form are described in Table 5.1.

Table 5.1 Built-in operators allowed in Integrity Constraints

System's Built-in	Represents
~F	$\neg F$
T_1 == T_2	$T_1 = T_2$
~(T_1 == T_2)	$T_1 \neq T_2$
T_1 < T_2	$T_1 < T_2$
T_1 =< T_2	$T_1 \leq T_2$
T_1 > T_2	$T_1 > T_2$
T_1 >= T_2	$T_1 \geq T_2$

Terms must be written according to the standard Prolog convention of capitalizing variables and leaving object constants in lower case.

Example 17 *The* ICs *in Example 5 would be represented as:*

```
<- [p(U,V),∼r(U,V)].
<- [∼p(U,V),r(U,V)].
<- [p(U,V),p(U,Z),∼(V==Z)].
```

Other examples of ICs*:*

```
<- [m(X,Y),X=<10].
<- [m(X,Y),∼n(Y)].
<- [q(X),X==apple].
<- [q(X),X==orange].
<- [t(X),s(Y),X>Y].
```

■

This file *must* reside in the same directory as XSB's binary executable for the system to find it.

5.1.2 Data Objects

By taking advantage of XSB's RDBMS interface[11] the program avoids the need to transform the data into a suitable format for processing. Consequently, having an adequate ODBC driver for the database in which the data is stored, lets us access it directly from the program. Some considerations must be taken into account for a proper execution:

[11] See Section 5.4.2 for some considerations regarding XSB 2.1 and ODBC.

– Table names must begin with a lower case letter. This is necessary because otherwise the system, as in traditional Prolog environments, will treat the table names as variables and not as predicates. Also, column names must begin with c1 and be numerated consecutively (e.g. c1, c2, etc.). A table should then be defined as:

```
CREATE TABLE "p"(c1 varchar(1),c2 varchar(1));
```

Watch for the pair of double quotes (`" "`), which force the table name's case to hold (SQL standard).

– A Dummy table *must* be present in the database, and it *must* contain a single column and a single element 'X'. It should be defined as follows (watch for lower case 'dummy'):

```
CREATE TABLE "dummy"(c1 (varchar(1)));
INSERT INTO "dummy"VALUES ('X');
```

This table is used for complex queries, and makes it possible to use the system in any known database without having to rely on proprietary system tables (i.e. Oracle's DUAL or IBM's DB2 sysibm.sysdummy1).

– The database must have an ODBC alias with its corresponding user name and password. These three parameters must be set in the file `main.P` as follows:

```
database('mydatabase').
username('myself').
password('mypassword').
```

No other part of the file `main.P` must be modified.

5.2 Modules

The implementation presented in this thesis comprises 4 modules. A brief description of each follows.

5.2.1 Main Module (`main`)

As the name says, this is the main module in the system. It must be loaded into XSB's interpreter with '`[main].`' . It provides the predicates `install/0`, `init/0`, `end/0`, `query/2`, `list_all/2` and `neg_query/2`. No other file must be loaded into the system by the user. Also in this file, the parameters mentioned in Section 5.1.2 (i.e. database ODBC alias, username and password) must be set.

5.2.2 Queca Generation Module (`queca`)

This is the heart of of the implementation. It contains the implementation of the *QUECA* algorithm and several other useful routines. It provides three predicates which are imported into 'main' and are visible to the user: `residues/2`, `queca/2` and `qca2sql/2`.

5.2.3 Query Transformation Module (`qca2fol`)

It defines the directive `qca2fol/4`. This predicate is not visible by the user but we consider it is important to know its location for reference purposes.

5.2.4 SQL Generation Module (`fol2sql`)

This module is part of (Bertossi et al., 1996 1998), and after some minor modifications was included in this system. It provides the predicate `fol2sql/2` which is imported into 'main'.

5.3 Using the Program

In this section we will describe how to use the application. We will begin by defining the syntax of the queries to be posed to the system. Next, we will show how to retrieve *consistent* information from the database. Some other useful predicates will be presented before analyzing some particular issues regarding a system module of XSB and the 'allowedness'of queries.

5.3.1 Queries

The queries supported by the system are those described in Section 4.5. However, the syntax must be changed to the standard Prolog convention of capitalizing variables and keeping object constants in lower case. When dealing with conjunctive queries, we write them as a list of terms.

Example 18 *Some valid queries would then be:*

```
p(X,Y).
p(a,Y).
[p(X,Y),Y>7].
[p(X,Y),t(X,Y)].
```

As long as tables `p` *and* `t` *are defined in the database.* ■

In the case of (partially) ground queries, the program uninstantiates them to fetch the corresponding non-ground *QUECAs*, and then re-instantiates them (the *QUECAs*) with the original constants. This is hand coded and is invisible to the user. However, future versions of XSB (probably 2.3 onwards) will support a special feature that allows, for a given table call, return as answer that of a more general call. This way we would avoid the overhead of the procedure just described.

5.3.2 Program Initialization

To correctly initialize the system, the user must do following:

a) Integrity Constraints must be defined in the file `ics`. They must obey the syntactical form (5.1), and built-in predicates (when needed) must be entered according to Table 5.1.

b) Database, username and password parameters must be modified in file `main.P` to meet the needs of a particular user. These parameters are those described in Section 5.1.2.

c) The XSB interpreter must be started and `| ?- [main].` must be executed to consult the main module of the program.

d) `| ?- install.` must be executed. This instruction compiles the rest of the modules.

e) `| ?- init.` must be executed. This predicate consults the rest of the modules and then connects to the database according to the parameters set in step b). During this process, when module `queca.P` is consulted the whole *QUECA* calculation routine is executed:

   i) File `ics` is read and its ICs are incorporated into a list. A list L_P of distinct predicates appearing in *IC* is also created.

   ii) Residues are calculated for every member of L_P (`residues/2`) and are written into file `results`.

   iii) *QUECA*'s are generated for every member of L_P using predicate `queca/2`.

   iv) `qca2fol` is executed for every *QUECA* generated. This predicate transforms a list of list (representing DNF) into a First Order Logic formula. This is done using the predicates defined in Table 5.2.

Table 5.2 Predicates and their meaning

Predicates	Formula
$no(F)$	$\neg F$
$and(F_1, F_2)$	$F_1 \wedge F_2$
$or(F_1, F_2)$	$F_1 \vee F_2$
$equal(T_1, T_2)$	$T_1 = T_2$
$T_1 =< T_2$	$T_1 \leq T_2$
$T_1 < T_2$	$T_1 < T_2$
$T_1 >= T_2$	$T_1 \geq T_2$
$T_1 > T_2$	$T_1 > T_2$
$all(X, F)$	$(\forall x)\ F$

Thus, when asking for a given *QUECA* the user will obtain a first order formula in prefix form. It is in this format that the *QUECA*s are written into file `results`.

v) Using predicate `qca2sql`, an SQL string is generated for every *QUECA* already computed. These SQL strings are also written into file `results`.

It is important to mention that because predicates `residues/2`, `queca/2` and `qca2sql/2` are declared tabled, their results are kept in memory and file `results` does not have to be consulted. This file is kept for reference purposes only.

After all computations have been performed, the connection to the database is established and its tables are mapped to XSB predicate names. This is done so they can be used by predicates `query/2` and `list_all/2`.

Once the initialization procedure is completed, the *QUECAs* and their equivalent SQL queries are stored in XSB's tables, and are also available in the file `results` for further reference.

f) Initialization is complete and the system is ready to be used.

5.3.3 Retrieving Consistent Information

Having successfully completed initialization procedure the user may execute:

- `| ?- query(Q,R).` Given a query `Q`, it returns in `R` a *consistent* tuple. It consults the database directly using the corresponding *QUECA* for `Q`. We may obtain *all* the consistent tuples, using `findall(G,query(Q,R),L)`, as a list of tuples `L` or via backtracking.

5.3.4 Other Useful Predicates

Other predicates available to the user after initialization are:

- `| ?- residues(Q,R).` Given a query `Q`, its residues are returned in `R` as a list in CNF (i.e. a inner list represents a clause and the list of lists is a conjunction of them).
- `| ?- queca(Q,R).` Given a query `Q`, its calculated $QUECA(Q)$ is returned in `R` as a first order formula, according to the notation defined in Table 5.2.
- `| ?- qca2sql(Q,R).` In case the formula calculated by `queca(Q,R)` is *allowed* (see Section 5.4.1), it returns in `R` the SQL query equivalent to the $QUECA(Q)$ already computed. Otherwise, it returns the string *Formula not allowed* (see Section 5.4.1).
- `| ?- list_all(Q,R).` As the name implies, given a query `Q`, it returns in `R` a tuple belonging to `Q`, no matter wether consistent or not. Again all the tuples may be obtained as a list of tuples `L` using `findall(G,list_all(Q,R),L)` or via backtracking. This predicate comes handy when posing more complex queries to the database.
- `| ?- neg_query(Q,R).` Given a query `Q`, it returns in `R` an *inconsistent* tuple. It is not very efficient as it is defined as `:-list_all(Q,R),\+ query(Q,R).`

5.3.5 Ending the Application

To terminate the application the following predicates are included:

- `| ?- end.` Disconnects from the database.
- `| ?- halt.` Exits XSB.

Example 19 *Assuming we have modified the file* `ics` *to contain the integrity constraints defined in Example 17, we would obtain the following results:*

```
| ?- [main].

yes
| ?- install.

yes
| ?- init.

yes
| ?- queca(p(X,Y),Q).
Q = and(p(id1,id2),all(u1,and(r(id1,id2),or(and(no(p(id1,u1)),
    no(r(id1,u1))),equal(id2,u1)))))

yes
| ?- qca2sql(p(X,Y),Q).
Q =  SELECT a0.c1,a0.c2 FROM  "p"  a0 WHERE NOT EXISTS
    (SELECT '*' FROM  "r"  a3 WHERE a0.c2<>a3.c2 AND a3.c1=a0.c1)
        AND NOT EXISTS
    (SELECT '*' FROM  "p"  a2 WHERE a0.c2<>a2.c2 AND a2.c1=a0.c1)
        AND NOT EXISTS
    (SELECT '*' FROM  "dummy"  WHERE NOT EXISTS
    (SELECT '*' FROM  "r"  a1 WHERE a1.c1=a0.c1 AND a1.c2=a0.c2))

yes
| ?- end.

yes
```

To get the consistent tuples of $P(x, y)$ *we would use* `query(p(X,Y),Q)`. *For an explanation of what table* `dummy` *represents see Section 5.1.2.* ■

5.4 Special Considerations

5.4.1 On Domain Independence

Given a query `Q` (see Section 5.3.1), `queca(Q,R)` delivers a first order formula in `R`, that, when posed as a query to a database, delivers *consistent* information only. However, due to the greater expressive power of first order logic (against traditional query languages as SQL), only some formulas can be used as queries in ordinary databases. This subset contains the so-called *domain independent* (Abiteboul et al., 1995, Ullman, 1988) formulas. As the name implies, the outcome of domain independent formulas does not depend on the domain over which variables range. For example, the formula $x > 1950$ is not domain independent because its answer set depends on the domain over which x ranges. For instance, if x is to range over $\mathbb{Z}$, its result set is infinite, on the other hand, if it ranges over $[0, 3000]$ it now has a finite answer set. If we pose the query $Year(x) \wedge x > 1950$, we now have a domain independent query because we have restricted x to the domain of valid years. So, no matter what is the domain of *Year*, only tuples stored in it (which are supposedly finite) shall be returned as answers.

Example 20 *(taken from (Topor and Sonenberg, 1988)) Some more examples of domain independent formulas:*

$$P(x);\ \exists x \exists y(P(x) \vee Q(y));\ \exists x \exists y(P(y) \rightarrow Q(x, y))$$

On the other hand, the following formulas are not domain independent:

$$\neg P(x);\quad P(x) \vee Q(y);\quad \forall y(P(y) \rightarrow Q(x, y));\quad \exists x P(x) \wedge \exists x \neg P(x)$$

■

Domain independent formulas were, however, shown to be equivalent to the class of *definite* formulas defined in (Nicolas and Demolombe, 1982), which were proved to be recursively unsolvable in (DiPaola, 1969) and in (Vardi, 1981). Because of this, decidable subclasses of domain independent formulas were proposed. Perhaps

the most common of these classes is *range restricted* formulas (Nicolas, 1982), which is equivalent to the notion of *allowed* formulas defined in (Topor and Sonenberg, 1988). For a deeper discussion on other decidable subclasses of domain independent formulas see (Abiteboul et al., 1995, Böhlen, 1994, Van Gelder and Topor, 1991).

To translate FOL formulas into SQL queries, we will take advantage of a module coded for SCDBR (Bertossi et al., 1998), an automated reasoner on specification of database updates. This way we avoid coding a translator between FOL and SQL from scratch. The condition used here to ensure domain independence is that of *allowed* queries (Böhlen, 1994, Burse, 1992). This notion of allowedness is slightly different from the one in (Topor and Sonenberg, 1988), in the sense that it is based on a procedural reading of formulas from left to right. The main advantage of this notion (the one described in (Böhlen, 1994, Burse, 1992)) is that recognizing an allowed formula and then translating it into relational algebra is fairly simple.

Although in (Van Gelder and Topor, 1991) it is argued that *evaluable* formulas (Demolombe, 1982) are the largest decidable subset of domain independent formulas (in fact, allowed formulas are a proper subset of them), its implementation is extremely complicated and the gain is not too big (specially for program-generated queries).

In consequence, due to the syntactic nature of the algorithm that determines whether a formula is domain independent or not, some of the computed *QUECA*s will not generate a SQL statement, even though, in some cases, they should.

5.4.2 XSB and ODBC

Although XSB 2.1 is supposed to connect easily to databases via ODBC, a bug prevented this from becoming true. To solve this problem, the system must be rebuilt using a special patch (see Appendix B) and some files must be modified. The modifications are as follows:

– In file `lib/odbc_call.P`, the line

```
:- dynamic attribute/4.
```

must be added at the beginning of the file, and line

```
attribute(1,'FLIGHT','FLIGHT_NO',string).
```

must be eliminated. This modification forms part of the whole set of alterations needed for the ODBC connection to work properly.

– In file `lib/odbc_call.H` the following lines must be added:

```
:- export odbc_open/3.
:- export odbc_close/0.
:- export odbc_attach/2.
:- export odbc_sql_select/2.
:- export odbc_get_schema/2.
```

This modification is done specifically for this implementation to allow the module `main.P` to import such predicates.

VI. CONCLUSIONS AND FURTHER WORK

In this thesis we faced the problem getting meaningful answers when querying an inconsistent database. In particular, we provided an algorithm that, given a first order query Q, computes a query $QUECA(Q)$, such that its answers correspond to the consistent answers of Q.

We established the method to be sound, complete and terminating for an interesting set of integrity constraints (BICs). The algorithm's complexity was also analyzed, obtaining satisfactory results.

Finally, the procedure was implemented in XSB, using standard LP techniques, and coupled into an ODBC compliant database to conform an interactive querying system.

To our knowledge, this is the first implemented system for retrieving consistent information out of a possibly inconsistent database. It should serve as a basis for a more powerful application in which some of the following extensions should be included:

a) Optimize the resulting queries, both syntactically and semantically. Because all the semantically relevant information associated to the query Q is already included in $QUECA(Q)$, it is possible that by just performing syntactical optimization we perform both optimizations at once. For this purpose there are plenty of commercial systems available that might be coupled to our system.

b) Incorporating existential (and/or) disjunctive queries. The difficulty of handling these queries was shown in Section 4.5. In the case of existential queries, the problem involves propagating the existential quantifier accordingly through the residues. In some cases it is possible to obtain correct results, but getting a general method seems quite difficult. Regarding disjunctive queries, we have not dealt with it yet.

c) Incorporating existential constraints into the solution, namely referential integrity constraints. This is closely related to existential queries. In essence,

they both involve propagation of existential quantifiers through a universally quantified query. The only difference, we note, is that in this case (referential ICs) the original query is built directly with existential quantifiers, so we have to take care of unifications and (in some way) modify the notion of information already in a TQU to be able to deal with existential quantifiers. Once that is done, the problem might be equivalent to that of existential queries. Another approach, which might prove to be easier, is to deal with a special case in which we exclude functional dependencies, as we believe they are the most problematic ICs regarding the interaction with existential quantifiers.

d) Eliminating residue redundancy completely. This problem has been barely treated, just to cope with redundant residues coming from functional dependencies. Other redundant residues must still be characterized, and furthermore, redundant ICs should be detected and eliminated using a special purpose package.

e) Testing the system on a real (large) database. We consider it should only be done once we have a fully operational system that is able to cope with referential integrity constraints. At that point we would have a really useful tool for querying databases, integrating data sources, etc. However, this involves one critical issue: most of the generated queries are domain dependent. This enforces the implementation to deal with restricting the domain of the queries in order to properly benchmark the system.

f) Developing the applications suggested in the introduction, e.g. data cleaning, datawarehousing, data integration, data mining, etc.

BIBLIOGRAPHY

Abiteboul, S., Hull, R. and Vianu, V. (1995). *Foundations of Databases*, Addison-Wesley.

Arenas, M., Bertossi, L. and Chomicki, J. (1999). Consistent Query Answers in Inconsistent Databases, *Proceedings ACM Symposium on Principles of Database Systems (ACM PODS '99)*, ACM Press, pp. 68–79.

Arenas, M., Bertossi, L. and Chomicki, J. (2000). Specifying and Querying Database Repairs using Logic Programs with Exceptions, Submitted to Fourth International Conference on Flexible Query Answering Systems (FQAS'2000).

Arenas, M., Bertossi, L. and Chomicki, J. (2001). Scalar Aggregation in FD-Inconsistent Databases, Submitted to International Conference on Database Theory (ICDT'2001).

Bertossi, L., Arenas, M. and Ferretti, C. (1998). SCDBR: An Automated Reasoner for Specifications of Database Updates, *Journal of Intelligent Information Systems* **10**(3): 253–280.

Bertossi, L., Arenas, M., Ferreti, C., Delaporte, A., Saez, P., Siu, B. and Strello, M. (1996). El Razonador SCDBR: Manual de Uso e Instalación, *User's manual*, Pontificia Universidad Católica de Chile.

Böhlen, M. (1994). *Managing Temporal Knowledge and Data Engineering*, Phd. thesis #10802, ETH, Zürich.

Burse, J. (1992). ProQuel: Using Prolog to Implement a Deductive Database System, *Technical report*, Department Informatik, ETH Zürich.

Celle, A. and Bertossi, L. (2000). Querying Inconsistent Databases: Algorithms and Implementation, *in* John Lloyd et al. (ed.), *Computational Logic – CL 2000*, Vol. 1861 of *Lecture Notes in Artificial Intelligence*, Springer, London, UK, pp. 942–956.

Chakravarthy, U., Grant, J. and Minker, J. (1990). Logic-Based Approach to Semantic Query Optimization, *ACM Transactions on Database Systems (ACM TODS)* **15**(2): 162–207.

Chaudhuri, S. and Dayal, U. (1997). An Overview of Data Warehousing and OLAP Technology, *SIGMOD Record* **26**(1): 65–74.

Cholvy, L. (1990). Querying an Inconsistent Database, *Proceedings of Artificial Intelligence : Methodology, systems and applications (AIMSA)*, North Holland.

Das, S. K. (1992). *Deductive Databases and Logic Programming*, first edn, Addison-Wesley.

Demolombe, R. (1982). Syntactical Characterization of a Subset of Domain Independent Formulas, *Technical report*, ONERA-CERT.

DiPaola, R. (1969). The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas, *Journal of the ACM* **16**(2): 324–324.

Godfrey, P., Grant, J., Gryz, J. and Minker, J. (1998). Integrity Constraints: Semantics and Applications, *in* J. Chomicki and G. Saake (eds), *Logics for Databases and Information Systems*, Kluwer Academic Publishers, Boston, chapter 9, pp. 265–306.

Lloyd, J. (1987). *Foundations of Logic Programming*, Artificial Intelligence, second edn, Springer-Verlag, New York.

McCune, W. (1994). OTTER 3.0 Reference Manual and Guide, *Technical report ANL-94/6*, Argonne National Laboratory.

Nicolas, J. and Demolombe, R. (1982). On the Stability of Relational Queries, *Technical report*, ONERA-CERT.

Nicolas, J.-M. (1982). Logic for Improving Integrity Checking in Relational Data Bases, *Acta Informatica* **18**(3): 227–253.

Reiter, R. (1984). Towards a Logical Reconstruction of Relational Database Theory, *in* M. Brodie, J. Mylopoulos and J. Schmidt (eds), *On Conceptual Modelling*, Springer-Verlag, New York, pp. 191–233.

Sagonas, K. F., Swift, T. and Warren, D. S. (1994). XSB as an Efficient Deductive Database Engine, *Proceedings of SIGMOD 1994 Conference*, ACM Press, pp. 442–453.

Sagonas, K. F., Swift, T., Warren, D. S., Freire, J. and Rao, P. (1999). *The XSB System Version 2.0 Volume 2: Libraries and Interfaces*.

Topor, R. and Sonenberg, E. (1988). On Domain Independent Databases, *in* J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publishers, Los Altos, chapter 6, pp. 217–240.

Ullman, J. (1988). *Principles of Database and Knowledge-Base Systems*, Vol. I, Computer Science Press, Maryland.

Van Gelder, A. and Topor, R. (1991). Safety and Translation of Relational Calculus Queries, *ACM Transactions on Database Systems (ACM TODS)* **16**(2): 235–278.

Vardi, M. (1981). The Decision Problem for Database Dependencies, *Information Processing Letters* **12**(5): 251–254.

APPENDICES

A. SOURCE CODE

A.1 Module main

A.1.1 main.H

```
%####################################
%
% module : main
%
% author : Alexander Celle T.
% date   : Autumn-2000
%####################################
:- export init/0, install/0, end/0.
:- export query/2, list_all/2, neg_query/2.

:- import qca2sql/2 from queca.
:- import make_id/2 from queca.
:- import variables/2 from queca.

:- import fol2sql/2 from fol2sql.

:- import odbc_open/3 from odbc_call.
:- import odbc_close/0 from odbc_call.
:- import odbc_attach/2 from odbc_call.
:- import odbc_sql_select/2 from odbc_call.
:- import odbc_get_schema/2 from odbc_call.
```

A.1.2 main.P

```
%####################################
%
% module : main
%
% author : Alexander Celle T.
% date   : Autumn-2000
%
%   This file should be loaded into
% the interpreter with [main].
%
%   Next, install/0 must be called
% followed by init/0.
%####################################

%Database ODBC alias
database('tesis').

%Username with access to the database
username('acelle').
```

```
%Password
password('db2admin').

%###############################
% DO NOT MODIFY BELOW THIS LINE
%###############################


%#####################################################
%   Query(+Q,-R):-
%       given a query Q it returns in R only
%       the consistent answers to Q.  Q must be
%       a database table with its arguments (i.e.
%       p(X,Y)).  It may be non-ground, ground or
%       partially ground. In case it's ground it
%       returns yes or no.
%#####################################################
query(Q,R):-
    qca2sql(Q,Sql),
    odbc_sql_select(Sql,R).

%#####################################################
%   list_all(+Q,-R):-
%       given a query Q it returns in R all the
%       tuples in Q, consistent or not.  If it is
%       ground it returns yes or no.
%#####################################################
list_all(Q,R):-
    (is_list(Q) ->
        variables(Q,Vq);
        variables([Q],Vq)),
    make_id(Vq,0),
    make_and(Q,Q1),     %convert to fol (only and)
    fol2sql(Q1,Str),
    name(Sql,Str),
    odbc_sql_select(Sql,R).

make_and([Q|[]],Q):- !.
make_and([Q|Qs],and(Q,Rest)):- !,
    make_and(Qs,Rest).
make_and(Q,Q).


%#####################################################
%   neg_query(+Q,-R):-
%       given a query Q it returns in R only
%       the inconsistent answers to Q.  Q must be
%       a database table with its arguments (i.e.
%       p(X,Y)).  It may be non-ground, ground or
%       partially ground. In case it's ground it
%       returns yes or no.
%#####################################################
neg_query(Q,R):-
    list_all(Q,R),
    \+ query(Q,R).

%#####################################################
%   install:-
%       compiles all the modules in the system.
%#####################################################
install :-
    compile(queca),
```

```
    compile(qca2fol),
    compile(fol2sql).

%#####################################################
%   init:-
%       consults the modules, initializes the system
%       (when it consults queca) and connects to
%       the database.
%#####################################################
init:-
    consult(qca2fol),
    consult(fol2sql),
    consult(queca),
    connect.

%#####################################################
%   end:-
%       disonnects from the database.
%#####################################################
end:-
    odbc_close.

%#####################################################
%   connect:-
%       connects to the database and maps the
%       table names to XSB predicates.
%#####################################################
connect:-
    database(Name),
    username(User),
    password(Pwd),
    odbc_open(Name,User,Pwd),
    map_tables.

%#####################################################
%   map_tables:-
%       maps the user tables in the database (lower
%       case) to XSB predicates.
%#####################################################
map_tables:-
    odbc_get_schema(user,L),
    map_tables(L).
map_tables([]).
map_tables([L|Ls]):-
    (usertable(L)->
        odbc_attach(L, table(L));
        true),
    map_tables(Ls).
usertable(T):-
    atom_chars(T,[L|_]),
    a@=<L,
    L@=<z.
```

A.2 Module queca

A.2.1 queca.H

```
%####################################
%
% module : queca
%
% author : Alexander Celle T.
% date   : Autumn-2000
%####################################
:- export residues/2.
:- export queca/2.
:- export qca2sql/2.
:- export make_id/2.
:- export variables/2.
:- export difference/3.

:- import absmember/2 from listutil.
:- import delete_ith/4 from listutil.
:- import setof/3 from setof.
:- import append/3 from basics.
:- import member/2 from basics.
:- import select/3 from basics.
:- import memberchk/2 from basics.
:- import length/2 from basics.
:- import subsumes_chk/2 from subsumes.
:- import numbervars/1 from num_vars.
:- import abolish_all_tables/0 from tables.
:- import abolish_table_pred/1 from tables.
:- import table_state/2 from tables.
:- import qca2fol/4 from qca2fol.
:- import fol2sql/2 from fol2sql.
```

A.2.2 queca.P

```
%####################################
%
% module : queca
%
% author : Alexander Celle T.
% date   : Autumn-2000
%####################################

%#######################
% TABLED PREDICATES
%#######################
:- table ic_pred_list/2.
:- table residues/2.
:- table queca/2.
:- table qca2sql/2.


%#################
%   New symbols
%#################
:- op(700,fx,<-).   % for denials in file 'ics'
:- op(500,fx,~).    % symbol for negation

%################################################
%   ic__pred_list(-L,-P):-
%       Reads the integrity constraints stored
```

```
%   in the file 'ics' and creates a list of them
%   and the distinct literals in them.
%################################################
ic_pred_list(L,P):-
    see(ics),
    get_constraints([],L,[],Pr),
    eliminate_variants(Pr,[],P),
    seen.
get_constraints(Aux,L,Paux,Pr):-
    read(<- Term) ->
        (add_to_list(Aux,[Term],NewAux),
        add_to_list(Paux,Term,NewPaux),
        get_constraints(NewAux,L,NewPaux,Pr));
        (L=Aux,
        Pr=Paux).

%################################################
%   eliminate_variants(+InList,[],-OutList):-
%       given a list it eliminates duplicates and
%       variants.  Important to avoid repetition of
%       residues for a given predicate.
%       Without this there would be one residuelist
%       for each individual appearance of a single
%       predicate
%       It also eliminates predicates which involve
%       built-in operators.
%################################################
eliminate_variants([],Aux,Aux).
eliminate_variants([X|Xs],Aux,List_wodups):-
    ((\+member(X,Aux),not_builtin(X) )->
        append(Aux,[X],Newaux);
        Newaux = Aux),
    eliminate_variants(Xs,Newaux,List_wodups).

not_builtin(~X):- !,X=..[F|_],checklist(F).
not_builtin(X):- X=..[F|_],checklist(F).
checklist(F):-
    builtins(Bins),
    \+ member(F,Bins).

%################################################
% Built-in predicates
%################################################
builtins([==,>,<,=<,>=,<>]).

%################################################
%    add_to_list(X,Y,Z) :-
%        adds element Y to list X and returns Z;
%        X MUST be a list and list Z is FLAT!!
%################################################
add_to_list(X,Y,Z):-
    is_list(X),
    is_list(Y) ->
        append(X,Y,Z);
        append(X,[Y],Z).

%################################################
%   qsort:-
%       Sorts a list of lists in ascending
%   order of length of each sublist.
%################################################
qsort([X|T],R) :-
    part(X,T,U1,U2),
```

```
    qsort(U1,V1),
    qsort(U2,V2),
    append(V1,[X|V2],R).
qsort([],[]).

part(M,[E1|T],[E1|U1],U2) :-
    length(M,M1),
    length(E1,E1),
    E1 =< M1,
    part(M,T,U1,U2).
part(M,[E1|T],U1,[E1|U2]) :-
    length(M,M1),
    length(E1,E1),
    E1 > M1,
    part(M,T,U1,U2).
part(_,[],[],[]).


%#####################################################
%   assert_relevant_ics:-
%       walks through the set of IC and asserts
%       relevant(Element,IC) for each pair where
%       Element belongs to IC.
%#####################################################
assert_relevant_ics:-
    ic_pred_list(_,Plist),
    assert_relevant_ics(Plist).
assert_relevant_ics([]).    %this is to here to make it succeed
assert_relevant_ics([X|Xs]):-
    assert_ics(X),
    assert_relevant_ics(Xs).
assert_ics(Element):-
    ic_pred_list(Clist,_),
    assert_ics(Element,Clist).
assert_ics(_,[]).
assert_ics(Element,[Ic|IcRest]):-
    (memberchk(Element,Ic)->
        (copy_term(Ic,NewIc),       %refreshes variables
        assert(relevant(Element,NewIc)),
        assert_ics(Element,IcRest));
        assert_ics(Element,IcRest)).

%#####################################################
%    make_selections:-
%       selects all possible combinations of
%       residues from an IC.  It also forces out
%       the second residue of a FD for a given
%       predicate, thus excluding redundancy
%       according to Lemma 1.
%#####################################################
make_selections:-
    clause(relevant(E,List),true),      %CRUCIAL STEP
    select(E,List,NewList),             %selects all possible
    negate_list(NewList,NegList),       %combinations of residues
    assert(relevant_selected(E,NegList)).

%#####################################################
%   negate_list(+List,-Neglist):-
%       given a list it negates its elements
%       and returns a new negated list
%#####################################################
negate_list([],[]).
negate_list([X|Xs],[Y|Ys]):-
```

```
    negate(X,Y),
    negate_list(Xs,Ys).

%#####################################################
%   negate(+X,-Xneg):-
%       given a literal X it returns its negation
%#####################################################
negate(X,Xneg):-
    (X = ~L ->
        Xneg = L;
        Xneg = ~X).

%#####################################################
%   residues(+Element,-Residuelist):-
%       given literal (Element) it returns the
%       associated residues according to the existing
%       Integrity Constraints, already in DNF.
%#####################################################
residues(Element,Residuelist):-
%    copy_term(Element,E2),
    ((Element = ~E3) ->
        (functor(E3,Pred,L),
        functor(Q2,Pred,L),
        Q= ~Q2);
        (functor(Element,Pred,L),
        functor(Q,Pred,L))),
    setof(List,clause(relevant_selected(Q,List),true),Rlistcnf),
    Q=Element,
    append([[Element]],Rlistcnf,Original),
    copy_term(Original,Copy),
    delete_ith(1,Original,_,RestOriginal),
    delete_ith(1,Copy,CopyElement,RestCopy),
    numbervars(CopyElement),
    write('PREDICATE : '),
    writeln(Element),
    write('     RESIDUES with redundancy : '),
    writeln(Rlistcnf),
    clean(RestCopy,RestOriginal,[],[],Residuelist1),
    qsort(Residuelist1,Residuelist),
    write('     RESIDUES : '),
    writeln(Residuelist).


/*
residues(Element,Residuelist):-
    setof(List,clause(relevant_selected(Element,List),true),Rlistcnf),
    append([[Element]],Rlistcnf,Original),
    copy_term(Original,Copy),
    delete_ith(1,Original,_,RestOriginal),
    delete_ith(1,Copy,CopyElement,RestCopy),
    numbervars(CopyElement),
    write('PREDICATE : '),
    writeln(Element),
    write('     RESIDUES with redundancy : '),
    writeln(Rlistcnf),
    clean(RestCopy,RestOriginal,[],[],Residuelist1),
    qsort(Residuelist1,Residuelist),
    write('     RESIDUES : '),
    writeln(Residuelist).
*/
%#####################################################
%   clean(+Groundlist,+Originallist,[],-CleanAux,-Cleanlist):-
%       returns in Cleanlist, the Originallist without
```

```
%       the redundant residues.  REQUIRES ORDER
%       p(X,Y) -> p(a,b) only for variant checking.
%#####################################################
clean([],[],_,Aux,Aux).
clean([X|Xs],[Y|Ys],Aux,CleanAux,Cleanlist):-
    (redundant(X,Aux)->
        (clean(Xs,Ys,Aux,CleanAux,Cleanlist));
        (add_to_list(Aux,[X],NewAux),
        add_to_list(CleanAux,[Y],NewCleanAux),
        clean(Xs,Ys,NewAux,NewCleanAux,Cleanlist))).

%#################################################
%   redundant(+X,+L):-
%       checks for residue redundancy of X in
%       the list of residues L.  One feature
%       it has is it permutates a residue to
%       check for different order of appearance
%       of literals.
%#################################################
redundant(_,[]):-
    fail.
redundant(X,[Y|Ys]):-
    setof(Perms,perm(X,Perms),Permlist),
    check_each_perm(Permlist,Y) ->
        true;
        redundant(X,Ys).
check_each_perm([],_):-
    fail.
check_each_perm([X|Xs],Y):-
    my_variant(X,Y) ->
        true;
        check_each_perm(Xs,Y).

%#####################################################
%   perm:-
%       given a list it generates all possible
%   permutations of the elements in that list by
%   backtracking.  It may be partly instantiated.
%#####################################################
perm([],[]).
perm(L,[X|Xs]):-
    select(X,L,L1),
    perm(L1,Xs).

%################################################
%   my_variant(+Term1,+Term2):-
%       this is just a redefinition of variant/2
%   in order to correctly check for variants like
%   X==a and a==Y.
%################################################
my_variant(Term1,Term2):-
    my_subsumes_chk(Term1,Term2),
    my_subsumes_chk(Term2,Term1).
my_subsumes_chk([],[]).
my_subsumes_chk([X|Xs],[Y|Ys]):-
    (X = (A==B) ->
        (subsumes_chk(A==B,Y);subsumes_chk(B==A,Y));
        (X = ~(A==B)->
            (subsumes_chk(~(A==B),Y);subsumes_chk(~(B==A),Y));
            subsumes_chk(X,Y))),
    my_subsumes_chk(Xs,Ys).

%################################################
```

```
%   generate_residues, generate_quecas & generate_sql
%       These three procedures work in the same way.
%       They walk through the whole Predicatelist
%       without variants or duplicates or X==Y's
%       and generate the residues, queca or sqlstring
%       associated to each predicate respectively.
%#################################################
generate_sql:-
    ic_pred_list(_,Plist),
    generate_sql(Plist).
generate_sql([]).
generate_sql([X|Xs]):-
    qca2sql(X,Sql),
    write('PREDICATE : '),
    writeln(X),
    write('     SQL   : '),
    writeln(Sql),
    generate_sql(Xs).

generate_residues:-
    ic_pred_list(_,Plist),
    generate_residues(Plist).
generate_residues([]).
generate_residues([X|Xs]):-
    residues(X,_),
    generate_residues(Xs).

generate_quecas:-
    ic_pred_list(_,Plist),
    generate_quecas(Plist).
generate_quecas([]).
generate_quecas([X|Xs]):-
    queca(X,Queca),
    write('PREDICATE : '),
    writeln(X),
    write('     QUERY : '),
    writeln(Queca),
    generate_quecas(Xs).

%#################################################
%   queca(+Q,-Quecafol):-
%       given a query Q it returns the corresponding
%       queca in Quecafol as a First Order Formula.
%       Query Q can be a list of terms (conjunction).
%#################################################
queca([Q|[]],Quecafol):-!,
    copy_term(Q,Q1), %refresh variables
    ((Q1 = ~Q2) ->
        (functor(Q2,Pred,L),
        functor(E2,Pred,L),
        E= ~E2);
        (functor(Q1,Pred,L),
        functor(E,Pred,L))),
    (residues(E,R)->
        (make_tqu([],[E],R,T),
        tqus([T],Querylist,[]));
        (Querylist=[[E]])),
    E=Q1,       %for use with ground queries
    variables([E],Ve),
    variables_list(Querylist,Vl),
    make_id(Ve,0),
    difference(Vl,Ve,Uvars),
    make_u(Uvars,0),
```

```
    sel_exists(Querylist,Evars),
    qca2fol(Querylist,Uvars,Evars,Quecafol).
queca([Q|Qs],and(Quecafol,Restfol)):-!,
    copy_term([Q|Qs],[Q1|Qs1]), %refresh variables
    ((Q1 = ~Q2) ->
        (functor(Q2,Pred,L),
        functor(E2,Pred,L),
        E= ~E2);
        (functor(Q1,Pred,L),
        functor(E,Pred,L))),
    (residues(E,R)->
        (make_tqu([],[E],R,T),
        tqus([T],Querylist,[]));
        (Querylist=[[E]])),
    E=Q1,       %for use with ground queries
    variables([E],Ve),
    variables_list(Querylist,Vl),
    make_id(Ve,0),
    difference(Vl,Ve,Uvars),
    make_u(Uvars,0),
    sel_exists(Querylist,Evars),
    qca2fol(Querylist,Uvars,Evars,Quecafol),
    queca(Qs1,Restfol).
queca(Q,Quecafol):-
    copy_term(Q,Q1), %refresh variables
    ((Q1 = ~Q2) ->
        (functor(Q2,Pred,L),
        functor(E2,Pred,L),
        E= ~E2);
        (functor(Q1,Pred,L),
        functor(E,Pred,L))),
    (residues(E,R)->
        (make_tqu([],[E],R,T),
        tqus([T],Querylist,[]));
        (Querylist=[[E]])),
    E=Q1,       %for use with ground queries
    variables([E],Ve),
    variables_list(Querylist,Vl),
    make_id(Ve,0),
    difference(Vl,Ve,Uvars),
    make_u(Uvars,0),
    sel_exists(Querylist,Evars),
    qca2fol(Querylist,Uvars,Evars,Quecafol).

%#####################################################
%   qca2sql(+Q,-Sql):-
%       given a query Q it returns the corresponding
%       SQL string in Sql.
%#####################################################
qca2sql(Q,Sql):-
    queca(Q,T),
    qca2sql0(T,Sql).
qca2sql0(or(Q,_),Sql):-!,
    qca2sql0(Q,Sql).
qca2sql0(Q,Sql):-
    fol2sql(Q,L),
    name(Sql,L).


%#######################################
%   make_id(+Vars,0) and make_u(+Vars,0)
%       grounds variables to id1, id2,
%   etc. so they can be evaluated.
```

```
%   (u1, u2 for universally quants)
%######################################
make_id([],_).
make_id([V|Vs],Counter):-
    Newcounter is Counter + 1,
    name(Newcounter,N),
    append([105,100],N,M),
    name(V,M),
    make_id(Vs,Newcounter).
make_u([],_).
make_u([V|Vs],Counter):-
    Newcounter is Counter + 1,
    name(Newcounter,N),
    append([117],N,M),
    name(V,M),
    make_u(Vs,Newcounter).

%#####################################################
%   tqus(+[tqu(D,E,R)|Ts],-Querylist,[]):-
%       given a list of tqus it executes the
%       while loop in Algorithm 2 until completion.
%       This is the heart of the implementation,
%       here is where the list of lists is generated
%       and the stopping conditions are evaluated.
%#####################################################
tqus([],Aux,Aux).
tqus([tqu(_,E,[])|Ts],Querylist,Aux):-
    add_to_list(Aux,[E],Newaux),
    tqus(Ts,Querylist,Newaux).
tqus([tqu(D,E,[R|Rs])|Ts],Querylist,Aux):-
    information_in(tqu(D,E,[R|Rs]))->
        (tqus([tqu(D,E,Rs)|Ts],Querylist,Aux));
        (info_p_in(tqu(D,E,[R|Rs]))->
            (d_split(tqu(D,E,R),Rs,Newtqu),
            append(Newtqu,Ts,Newtqus),
            tqus(Newtqus,Querylist,Aux));
            (split(tqu(D,E,R),R,Rs,[],Newtqu),
            append(Newtqu,Ts,Newtqus),
            tqus(Newtqus,Querylist,Aux))).


%#####################################################
%   best_in
%       Basically chooses the substitution that
%       produces most information in E from the
%       residues in R. Use setof to keep the same
%       unification is CRUCIAL.  The criteria used
%       is that of shortest list.
%#####################################################
best_in(tqu(D,E,R),NewR):-
    setof(Goal,partially_in(tqu(D,E,R),Goal),Goallist),!,
    shortest(Goallist,R,NewR).

%#####################################################
%   patially_in(+tqu(D,E,R),-NewR):-
%       given that a residue is part of E, that
%       is, R is partially in R, then we choose
%       the best unification to minimize the
%       remaining residues in NewR.  We must take
%       care to consider only the free vars in R and
%       the new vars in E. (That is why grounding
%       is performed).
%#####################################################
```

```
partially_in(tqu(D,E,R),NewR):-
    copy_term(tqu(D,E,[R|_]),tqu(Dc,Ec,[Rc|Rsc])),
    Ec=[Q|_],
    numbervars(Q),      %grounds vars in Q
    freeVar(tqu(Dc,Ec,[Rc|Rsc]),Fvar),
    variables(Rc,VarsRc),
    difference(VarsRc,Fvar,Groundvars),
    numbervars(Groundvars),%ground vars in residue except freeVars.
    trim_residues(Ec,Rc,R,[],NewR).


%#####################################################
%   trim_residues(+E,+Rescopy,+Resreal,[],-NewR):-
%       given an E, and residues, we only add to
%       NewR those residues that do not unify with
%       a term in R.
%#####################################################
trim_residues(_,[],[],Aux,Aux).
trim_residues(E,[Rc|Rcs],[R|Rs],Aux,NewR):-
    (my_absmember(Rc,E)->
        Newaux=Aux;
        add_to_list(Aux,R,Newaux)),
    trim_residues(E,Rcs,Rs,Newaux,NewR).

%#####################################################
%   shortest(+L,+Max,-NewR):-
%       given a list of lists L, and a maximum length
%       Max, it returns the shortes list in NewR.
%#####################################################
shortest([],Aux,Aux).
shortest([L|Ls],Aux,NewR):-
    length(L,Ll),
    length(Aux,Laux),
    ((Ll @< Laux)->
        Newaux=L;
        Newaux=Aux),
    shortest(Ls,Newaux,NewR).

%#####################################################
%   d_split(+tqu(D,E,R),+Rs,-Newtqu):-
%       given a TQU it selects the best unification
%       for the partially belonging residues in R
%       and then splits the TQU, using te remaining
%       residues in Rs, to produce a new set of
%       TQUs in Newtqu.
%#####################################################
d_split(tqu(D,E,R),Rs,Newtqu):-
    best_in(tqu(D,E,R),NewR),
    difference(R,NewR,Rest),
    split(tqu(D,E,NewR),NewR,Rs,[],Newtqu1),
    dummy_split(tqu(D,E,Rest),Rest,Rs,[],Newtqu2),
    append(Newtqu1,Newtqu2,Newtqu).
dummy_split(tqu(_,_,[]),_,_,Aux,Aux).
dummy_split(tqu(D,E,[R1|R2]),R,Rs,Aux,Newtqu):-
   (is_list(E)->
        add_to_list(E,R1,Temp);
        Temp=[E,R1]),
    add_to_list(D,R,NewD),
    make_tqu(NewD,Temp,Rs,T),
    add_to_list(Aux,T,Newaux),
    dummy_split(tqu(D,E,R2),R,Rs,Newaux,Newtqu).

%#####################################################
```

```
%   split(+tqu(D,E,R),+Rs,-Newtqu):-
%       performs the split operation.  That is,
%       it generates copies of E and Rs, puts R in D,
%       puts one member of R in each copy of E and
%       adds the residue of each member of R to
%       the corresponding copy of Rs.
%#####################################################
split(tqu(_,_,[]),_,_,Aux,Aux).
split(tqu(D,E,[R1|R2]),R,Rs,Aux,Newtqu):-
    (is_list(E)->
        add_to_list(E,R1,Temp);
        Temp=[E,R1]),
%    copy_term(R1,Rd),
    ( R1 = ~Rd ->
        (functor(Rd,Pred,L),
        functor(Rd2,Pred,L),
        NewR= ~Rd2);
        (functor(R1,Pred,L),
        functor(NewR,Pred,L))),
    (residues(NewR,R1s)->
        (NewR = R1,
        add_to_list(Rs,R1s,Rr));
        Rr=Rs),
    make_tqu(R,Temp,Rr,T),
    add_to_list(Aux,T,Newaux),
    split(tqu(D,E,R2),R,Rs,Newaux,Newtqu).



%##########################
%   TQU constructor
%##########################
make_tqu(D,E,R,tqu(D,E,R)).

%################################################
%   information_in(+tqu(D,E,[R|Rs]):-
%       verifies whether the information of
%       residue R is already in E, according
%       to Definition 3.4.  This means that
%       the whole residue is contained in E.
%################################################
information_in(tqu(D,E,[R|Rs])):-
    copy_term(tqu(D,E,[R|Rs]),tqu(Dc,Ec,[Rc|Rsc])),
    Ec=[Q|_],
    numbervars(Q),      %grounds vars in Q
    freeVar(tqu(Dc,Ec,[Rc|Rsc]),Fvar),
    variables(Rc,VarsRc),
    difference(VarsRc,Fvar,Groundvars),
    numbervars(Groundvars),%ground vars in residue except freeVars.
    (in_E(Ec,Rc) ->
        true;
        (in_D(Dc,Rc) ->
            true;
            fail)).

%################################################
%   info_p_in(+tqu(D,E,[R|Rs]):-
%       verifies whether the information of
%       residue R is partially in E, according
%       to Definition 3.4.  This means that
%       the part of the residue is contained in E.
%################################################
info_p_in(tqu(D,E,[R|Rs])):-
```

```
    copy_term(tqu(D,E,[R|Rs]),tqu(Dc,Ec,[Rc|Rsc])),
    Ec=[Q|_],
    numbervars(Q),      %grounds vars in Q
    freeVar(tqu(Dc,Ec,[Rc|Rsc]),Fvar),
    variables(Rc,VarsRc),
    difference(VarsRc,Fvar,Groundvars),
    numbervars(Groundvars),%ground vars in residue except freeVars.
    (in_E2(Ec,Rc) ->
        true;
        fail).


%######################################
%   in_E, in_D, in_E2, in_D2
%       verify if a residue, or
%       part of it is in E or D
%       E2 checks for part of it only.
%######################################
in_E(_,[]).
in_E(E,[R|Rs]):-
    my_absmember(R,E),
    in_E(E,Rs).

in_D([],_):-
    fail.
in_D([D|Ds],R):-
    in_D2(D,R)->
        true;
        in_D(Ds,R).

in_D2(_,[]).
in_D2(E,[R|Rs]):-
    member(R,E),
    in_E(E,Rs).

in_E2(_,[]):-
    fail.
in_E2(E,[R|Rs]):-
    my_absmember(R,E)->
        true;
        in_E2(E,Rs).

%############################
% New var in a TQU
%   According to Def. 3.2
%   not used.
%############################
/*newVar(tqu(D,[Q|E],_),Nvar):-
    variables([Q],Vq),
    variables(E,Ve),
    difference(Ve,Vq,Nvar).
*/

%############################
% Free var in a residue
%   According to Def. 3.3
%############################
freeVar(tqu(_,E,[R|_]),Fvar):-
    variables(R,Vr),
    variables(E,Ve),
    difference(Vr,Ve,Fvar).

%############################
```

```
% Free var in D
%   According to Def. 3.3
%   not used.
%############################
/*freeVarD(tqu(D,E,_),Fvar):-
    variables(D,Vd),
    variables(E,Ve),
    difference(Vd,Ve,Fvar).
*/

%######################################
% variables(+Terms,-Varlist):-
%
%       Given a list of terms it builds
%       a list of distinct variables.
%######################################
variables(Terms,Varlist):-
    variables(Terms,[],Arglist),
    onlyvars(Arglist,[],Varlist).
variables([],Aux,Aux).
variables([T|Ts],Aux,Arglist):-
    (T= ~A->
        A=.. Temp;
        T=.. Temp),
    append(Aux,Temp,Newaux),
    variables(Ts,Newaux,Arglist).
onlyvars([],Aux,Aux).
onlyvars([A|As],Aux,Varlist):-
    ((var(A), \+ absmember(A,Aux))->
        add_to_list(Aux,A,Newaux);
        Newaux=Aux),
    onlyvars(As,Newaux,Varlist).

%######################################
% variables_list(+Lists,-Varlist):-
%
%       Given a list of lists of terms
%       it builds a list of distinct
%       variables.
%######################################
variables_list(Lists,Varlist):-
    variables_list(Lists,[],Arglist),
    onlyvars(Arglist,[],Varlist).
variables_list([],Aux,Aux).
variables_list([L|Ls],Aux,Arglist):-
    variables(L,Vl),
    append(Aux,Vl,Newaux),
    variables_list(Ls,Newaux,Arglist).


sel_exists(Lists,Elist):-
    sel_exists(Lists,[],All),
    only_exists(All,[],Elist).
only_exists([],Aux,Aux).
only_exists([A|As],Aux,Varlist):-
    ((is_exists(A), \+ absmember(A,Aux))->
        add_to_list(Aux,A,Newaux);
        Newaux=Aux),
    only_exists(As,Newaux,Varlist).

sel_exists([],Aux,Aux).
sel_exists([L|Ls],Aux,Elist):-
    s_exists(L,El),
```

```
    append(Aux,El,Newaux),
    sel_exists(Ls,Newaux,Elist).
s_exists(L,El):-
    s_exists(L,[],El).
s_exists([],Aux,Aux).
s_exists([L|Ls],Aux,El):-
    (L= ~A->
        A=.. Temp;
        L=.. Temp),
    append(Aux,Temp,Newaux),
    s_exists(Ls,Newaux,El).
is_exists(V):-
    atom(V)->
        (atom_chars(V,LV),
        append(TVL,LY,LV),
        number_chars(NY,LY),
        NY>0,
        atom_chars(TV,TVL),
        member(TV,[exists]));
        (fail).


%##############################
% difference(+X,+Y,-Dif):-
%
%   Given two sets X and Y it
% returns Dif = X - Y using
% absmember (i.e. ==/2).
%##############################
difference([],_,[]).
difference([X|Xs],Y,Dif):-
    absmember(X,Y),!,
    difference(Xs,Y,Dif).
difference([X|Xs],Y,[X|Dif]):-
    difference(Xs,Y,Dif).


%#####################################################
% my_absmember(+X,+L):-
%
%       Takes into account that X==Y <=> Y==X.  It
%       alse ensures that substitutions only consider
%       variables (i.e. are variants).
%#####################################################
my_absmember(X,L):-
    variables(L,Vl),
    (X = (A==B)->
        (status(A,Sa),status(B,Sb),
        (member(A==B,L);member(B==A,L)),
        status(A,Sa),status(B,Sb),
        still_vars(Vl));
        (X = ~(A==B) ->
            (status(A,Sa),status(B,Sb),
            (member(~(A==B),L);member(~(B==A),L)),
            status(A,Sa),status(B,Sb),
            still_vars(Vl));
            (variables([X],V),member(X,L),
            still_vars(V),still_vars(Vl)))).

still_vars([]).
still_vars([L|Ls]):-
    var(L),
```

```
    still_vars(Ls).
status(Var,Status):-
    var(Var)->
        Status='var';
        Status='nonvar'.


%########################################
%   INITIALIZATION
%########################################
:- abolish_all_tables.
:- ic_pred_list(L,P),fail.
:- tell('results').
:- writeln('-----------------------------------'),
    writeln('       Residues'),
    writeln('-----------------------------------').
:- !,assert_relevant_ics,fail.          %complete table
:- make_selections,fail.                %CRUCIAL step
:- !,generate_residues,fail.            %complete table
:- abolish(relevant/2),abolish(relevant_selected/2).    %freespace
:- writeln('-----------------------------------'),
    writeln('       Quecas'),
    writeln('-----------------------------------').
:- !,generate_quecas,fail.
:- writeln('-----------------------------------'),
    writeln('       SQL'),
    writeln('-----------------------------------').
:- !,generate_sql,fail.
:- table_state(ic_pred_list(X,Y),complete)->
     abolish_table_pred(ic_pred_list(X,Y)).
:- told.

%########################################
%   END
%
%   What is left:
%       1.- tables with residues
%       2.- tables with quecas
%       3.- tables with SQL strings
%       3.- File results with results
%########################################
```

A.3 Module qca2fol

A.3.1 qca2fol.H

```
%###################################
%
% module : qca2fol
%
% author : Alexander Celle T.
% date   : Autumn-2000
%###################################
:- export qca2fol/4.
```

```
:- import append/3, select/3 from basics.
```

A.3.2 qca2fol.P

```
%####################################
%
% module : qca2fol
%
% author : Alexander Celle T.
% date   : Autumn-2000
%
% IMPORTANT - DO NOT MODIFY THIS FILE
%####################################

:- op(500,fx,~).    % symbol for negation

%#################################################
%   qca2fol(+QUECA,+Uvars,-FOL):-
%       Given a QUECA as described above (DNF) and
%       a list of universally quantified variables
%       it generates in FOL a formula in FOL in
%       prefix notation. It factorizes common terms.
%
%       This prefix notation includes
%
%   all(x,F)
%   and(F1,F2)
%   or(F1,F2)
%   no(F)
%   equal(T1,T2)
%
%       Other built-in operators are used normally
%   T1=<T2
%   T1>=T2
%   T1>T2
%   T1<T2
%   T1<>T2
%
% QUECA is [ [...] [...] [...] ... [...] ] etc, a list
% of lists.  Each inner list is a conjunction of terms
% and the bigger list is a disjunction of inner lists.
% Thus it represents a formula in DNF.
%#########################################################
qca2fol([[E|Es]|Ls],_,_,E2):-  %No more elements in the query
    tuple(E,E2),
    eliminateE([[E|Es]|Ls],[],[[]]),!.
qca2fol([[E|Es]|Ls],G,X,and(E2,R)):-
    tuple(E,E2),
    eliminateE([[E|Es]|Ls],[],L),
    qca2fol0(L,G,X,R).

%#################################################
%   eliminateE(+L,[],-Final):-
%       eliminates the first element of every
%       sublist.  This is done because the
%       algorithm is supposed to put E at the
%       beggining of every branch.
%#################################################
```

```
eliminateE([],Aux,Aux).
eliminateE([[_|G2]|Gs],Aux,Final):-
    append(Aux,[G2],Newaux),
    eliminateE(Gs,Newaux,Final).
eliminateE([[_|G2]],Aux,Final):-
    append(Aux,[G2],Newaux),
    eliminateE([],Newaux,Final).

%###################################################
%   qca2fol0(+L,+Uvars,-R):-
%       continues the process of qca2fol/3.
%       Essentially it adds universal quantification
%       after the already factorized E.  Then it
%       calls for further factorization.
%###################################################
qca2fol0(L,[],X,R):-
    qca2fol0(L,X,R).
qca2fol0(L,[G|Gs],X,all(G,R)):-
    qca2fol0(L,Gs,X,R).

qca2fol0(L,[],R):-
    factqueca(L,R).
qca2fol0(L,[G|Gs],some(G,R)):-
    qca2fol0(L,Gs,R).


%###################################################
%   factqueca(+Listoflists,-R):-
%       factorizes common terms out of the list
%       of lists which represents DNF.
%###################################################
% Case 1: [[A],[A],[A]]
factqueca(Listoflists,E2):- fact(Listoflists,E,[],[]),!,
                            tuple(E,E2).
% Case 2: [[A,B],[A,C],[A,D]]
factqueca(Listoflists,and(E2,R2)):- fact(Listoflists,E,R,[]),!,
                                    tuple(E,E2),
                                    factqueca(R,R2).
% Case 3: [[A],[B],[C]]
factqueca(Listoflists,or(E2,Rs2)):- fact(Listoflists,E,[],Rs),!,
                                    tuple(E,E2),
                                    factqueca(Rs,Rs2).
% Case 4: [[A,_],[A,_],[B,_]]
factqueca(Listoflists,or(and(E2,R2),Rs2)):- fact(Listoflists,E,R,Rs),!,
                                    tuple(E,E2),
                                    factqueca(R,R2),
                                    factqueca(Rs,Rs2).

%###################################################
%   tuple and convert
%       transforms ~ into no and == into equal.
%       The rest of the predicates remain the same.
%###################################################
tuple(~N,no(N2)):- !,tuple(N,N2).
tuple(N,N2):- !,convert(N,N2).
convert(N,N2):- N=..[==|Ls],!,N2=..[equal|Ls].
convert(N,N).

%###################################################
%   fact(+L,-Element,-Innerlist,-Outerlist):-
%       Given a list of lists L it factorizes out
%       the common term Element.  The values of
%       Innerlist and Outerlist depend on if
```

```
%       such factorization exists and to what
%       extent.  See the 4 cases above.
%#####################################################
fact(L,[],[],L2):-
    select([],L,L2),!.
fact([L|Ls],E,R,Rs):-
    L= [E|Es],
    (Es = [] ->
        Temp=[];
        Temp=[Es]),
    fact(Ls,E,Temp,R,Rs).
fact([],_,Aux,Aux,[]).
fact([L2|L2s],E,Aux,R,Rs):-
    L2= [E2|E2s],
    (E2==E ->
        (append(Aux,[E2s],Newaux),
        fact(L2s,E,Newaux,R,Rs));
        (R=Aux,
        Rs=[L2|L2s])).
```

A.4 Module `fol2sql`

A.4.1 `fol2sql.H`

```
%###################################
%
% module : main
%
% author : Mauricio Strello
%          Small modifications done
%       by Alexander Celle T.
% date   : Autumn-2000
%###################################
:- export fol2sql/2.

:- import append/3,member/2 from basics.
:- import difference/3 from queca.
```

A.4.2 `fol2sql.P`

```
% Mapeando formulas en FOL a consultas en SQL.
% Mauricio Strello.
% Oto~no de 1997.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(700,xfx,[<>]).
:- op(700,xfx,[<=]).

is_atom(F) :- functor(F,N,_), \+current_op(_,_,N).
```

```
% Normalizacion de formulas en FOL.
% La utilizacion de "==>" es solo "sintactic sugar".
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(950,xfx,'==>').

implies(F1,F2)        ==> or(no(F1),F2).
all(VL,F)             ==> no(some(VL,no(F))).

no(and(A,B))         ==> or(no(A),no(B)).
no(or(A,B))          ==> and(no(A),no(B)).
no(no(F))            ==> F.
no(equal(T1,T2))     ==> T1<>T2.
no(T1<T2)            ==> T1>=T2.
no(T1>T2)            ==> T1=<T2.
no(T1=<T2)           ==> T1>T2.
no(T1>=T2)           ==> T1<T2.
no(T1<>T2)           ==> equal(T1,T2).

and(F1,or(F2,F3))     ==> or(and(F1,F2),and(F1,F3)).
and(or(F1,F2),F3)     ==> or(and(F1,F3),and(F2,F3)).
some(VL,or(F1,F2))    ==> or(some(VL,F1),some(VL,F2)).

%% repeated(+Var,+VarList,-VarNr)
repeated(X,[[X,N]|_],N) :- !.
repeated(X,[_|L],N) :- repeated(X,L,N).

%% try(+Form,+VarNr,+VarList,-Form,-VarNr,-VarList)
try(X,N,VL,var(N1),N1,VL1) :- variable(X),( repeated(X,VL,N1) -> VL1=VL ;
  N1 is N+1,append([[X,N1]],VL,VL1) ).
try(X,N,VL,Y,N,VL) :- X==>Y.
%try(X,N,VL,Z,N1,VL1) :- X==>Y,try(Y,N,VL,Z,N1,VL1).
try(X,N,VL,Y,N1,VL1) :-
  X=..[F|Args], try_list(Args,N,VL,NArgs,N1,VL1), Y=..[F|NArgs].

%% try_list(+FormList,+VarNr,+VarList,-FormList,-VarNr,-VarList)
try_list([X|T],N,VL,[Z|T],N1,VL1) :- try(X,N,VL,Z,N1,VL1).
try_list([X|T],N,VL,[X|T1],N1,VL1) :- try_list(T,N,VL,T1,N1,VL1).

%% normalize(+Form,+VarNr,+VarList,-Form,-VarNr,-VarList)
normalize(X,N,VL,Y,N2,VL2) :-
  try(X,N,VL,Z,N1,VL1), !, normalize(Z,N1,VL1,Y,N2,VL2).
normalize(X,_,_,X,_,_).

% "Allowedness" de formulas en FOL.
% Remitirse a la tesis de Bohlen.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% vars(+Formula,+VarList,?VarList)
vars(var(N),V,V1)      :- !, (member(var(N),V) -> V1=V; V1=[var(N)|V]).
vars(some(VL,A),V,V1)  :- !, vars(A,V,V2), difference(V2,[VL],V1).
vars([H|T],V,V1)       :- !, vars(H,V,V2), vars(T,V2,V1).
vars([],V,V1)          :- !, V=V1.
vars(A,V,V1)           :- A=..[_|L], vars(L,V,V1).

%% alwd(+Formula,+SafeVarList)
% Alexander Celle T.
%   Checks for allowedness of formulas according to Boehlen.
% Has one basic syntactic problem: and(a,and(b,c)) may be
% allowed, while and(and(a,b),c) may not.
% NO TRANSITIVITY !!
%
```

```
alwd(and(A,B),V)      :- !, alwd(A,V), vars(A,V,V1), alwd(B,V1).
alwd(or(A,B),_)       :- !, alwd(A,[]), alwd(B,[]), vars(A,[],V), vars(B,[],V).
alwd(no(A),V)        :- !, alwd(A,V), vars(A,V,V).
alwd(some(VL,A),V)    :- !, difference(V,[VL],V1), alwd(A,V1).
alwd(T1<T2,V)         :- !, vars(T1,V,V), vars(T2,V,V).
alwd(T1>T2,V)         :- !, vars(T1,V,V), vars(T2,V,V).
alwd(T1=<T2,V)        :- !, vars(T1,V,V), vars(T2,V,V).
alwd(T1>=T2,V)        :- !, vars(T1,V,V), vars(T2,V,V).
alwd(T1<>T2,V)        :- !, vars(T1,V,V), vars(T2,V,V).
alwd(equal(T1,T2),V)  :- !, (vars(T1,V,V); vars(T2,V,V)).
alwd(A,_)             :- is_atom(A), !.

% Traduccion de FOL a SQL.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

or_free_part(or(A,B), OFP) :- !, (or_free_part(A, OFP) ; or_free_part(B, OFP)).
or_free_part(OFP, OFP).

simplify(Q,Q1) :- simplify1(Q,Qx), !, simplify(Qx,Q1).
simplify(Q,Q).

simplify1(sel(S,F,W),sel(S,F1,W)) :-
  simplify2(F,F1).

simplify2([R-A|T],[R1-A|T]) :- simplify1(R,R1).
simplify2([R-A|T],[R-A|T1]) :- simplify2(T,T1).

remove(_,[],[]).
remove(QV,[Var-_|S1],S2) :- member(Var,QV), !, remove(QV,S1,S2).
remove(QV,[Var-V|S1],[Var-V|S2]) :- remove(QV,S1,S2).

unify(_-V,_-V,W,W) :- !.
unify(_-V1,_-V2,W,[V2=V1|W]).

trans_expr(var(N),S,var(N)-V,S) :- member(var(N)-V,S), !.
trans_expr(var(N),S,var(N)-V,[var(N)-V|S]) :- !.
trans_expr(E,S,_-E,S).

trans_args([],_,_,S,W,S,W).
trans_args([Arg|T],N,A,S,W,S2,W2) :-
  N1 is N+1,
  trans_args(T,N1,A,S,W,S1,W1),
  trans_expr(Arg,S1,VV,S2),
  Dummy=A-N,    %ACELLE, avoid syntax error
  unify(VV,_-Dummy,W1,W2).

%% add_sel(+Select,+Alias,+ColNumber,+Select,+Where,-Select,-Where)
% Alexander Celle T.  not used.
/*add_sel([],_,_,S,W,S,W).
add_sel([Var-_|T],A,N,S,W,S2,W2) :-
  trans_expr(Var,S,VV,S1),
  Dummy=A-N,    %ACELLE, avoid syntax error
  unify(VV,_-Dummy,W,W1),
  N1 is N+1,
  add_sel(T,A,N1,S1,W1,S2,W2).
*/

%% trans_form(+Formula,+AliasSelectStmt,-AliasSelectStmt)
%% Se supone que +Formula ya ha sido normalizada anteriormente
% Formulas del tipo A and B
trans_form(and(A,B),I,O) :- !,
  trans_form(A,I,H),
```

```
  trans_form(B,H,O).
% Formulas del tipo no E
trans_form(no(E),A-sel(S,F,W),
           A1-sel(S1,F,[no(sel([*],F1,W1))|W])) :- !,
  trans_form(E,A-sel(S,[],[]),A1-sel(S1,F1,W1)).
% Formulas del tipo exists QV: E
trans_form(some(QV,E),A-sel(S1,F1,W1),A1-sel(S2,F2,W2)) :- !,
  trans_form(E,A-sel(S1,F1,W1),A1-sel(S3,F2,W2)),
  remove([QV],S3,S2).
% Formulas del tipo E1 == E2
trans_form(equal(E1,E2),A-sel(S,F,W),A-sel(S2,F,W1)) :- !,
  trans_expr(E1,S,VV1,S1),
  trans_expr(E2,S1,VV2,S2),
  unify(VV1,VV2,W,W1).
% Formulas del tipo E1 < E2
trans_form(E1<E2,A-sel(S,F,W),A-sel(S2,F,[V1<V2|W])) :- !,
  trans_expr(E1,S,_-V1,S1),
  trans_expr(E2,S1,_-V2,S2).
% Formulas del tipo E1 > E2
trans_form(E1>E2,A-sel(S,F,W),A-sel(S2,F,[V1>V2|W])) :- !,
  trans_expr(E1,S,_-V1,S1),
  trans_expr(E2,S1,_-V2,S2).
% Formulas del tipo E1 <> E2
trans_form(E1<>E2,A-sel(S,F,W),A-sel(S2,F,[V1<>V2|W])) :- !,
  trans_expr(E1,S,_-V1,S1),
  trans_expr(E2,S1,_-V2,S2).
% Formulas del tipo E1 =< E2
%ACELLE Modified =< to <= (SQL syntax)
trans_form(E1=<E2,A-sel(S,F,W),A-sel(S2,F,[V1<=V2|W])) :- !,
  trans_expr(E1,S,_-V1,S1),
  trans_expr(E2,S1,_-V2,S2).
% Formulas del tipo E1 >= E2
trans_form(E1>=E2,A-sel(S,F,W),A-sel(S2,F,[V1>=V2|W])) :- !,
  trans_expr(E1,S,_-V1,S1),
  trans_expr(E2,S1,_-V2,S2).
% Formulas atomicas
trans_form(P,A-sel(S,F,W),A1-sel(S1,[Func-A|F],W1)) :-
  P=..[Func|Args],
  A1 is A+1,
  trans_args(Args,1,A,S,W,S1,W1).

trans_query(Form,A,union(SQLList),A1) :-
  % encuentra todas las partes libres de "or"
  findall(OFF, or_free_part(Form,OFF), FL),
  trans_query2(FL,A,SQLList,A1).
trans_query2([],A,[],A).
trans_query2([OFF|OFFList],A,[SQL1|SQLList],A1) :-
  trans_form(OFF,A-sel([],[],[]),A2-SQL),
  simplify(SQL,SQL1),
  trans_query2(OFFList,A2,SQLList,A1).


% Construccion de la consulta SQL (se devuelve un string)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% g_selstat(+TERM): ("generate_select_statement" o genera
% una sentencia SELECT): recibe un termino que representa
% la consulta y genera un string con la traduccion en SQL.


g_selstat(union(L))     --> g_unions(L).

g_selstat(sel(S,F,W))   --> "SELECT ", g_exprlist(S),
```

```
                          " FROM ", g_from(F),
                          g_where(W).

g_unions([A])           --> !, g_selstat(A).
g_unions([A|B])         --> g_selstat(A), " UNION ", g_unions(B).


g_expr(var(_)-A)        --> !, g_expr(A).
g_expr(A-C)             --> !, "a", g_atom(A), ".c", g_atom(C).
%################################################
% Alexander Celle T.
%g_expr(X)               --> !, g_atom(X).
g_expr(X)               --> !, "'",g_atom(X),"'".
% This is needed for the RDBMS to interpret a
% ground term as a constant and not a variable.
% At least in MS Access :)
%################################################

%#################################
% Alexander Celle T.
%g_exprlist([])          --> "1".
g_exprlist([])          --> "1".
% It might need a *, not sure
%#################################
g_exprlist([E])         --> !, g_expr(E).
g_exprlist([E|L])       --> g_expr(E), !, ",", g_exprlist(L).
g_exprlist([_|L])       --> g_exprlist(L).

%#################################################################
%g_from([])              --> "DUAL".
g_from([])              --> " ""dummy"" ".
%
% Alexander Celle T.
%   DUAL is exclusive for Oracle.  To make it portable to
% all DB's you must do the following in the database.
%       CREATE TABLE "dummy"(c1 (varchar(1));
%       INSERT INTO "dummy" VALUES ('X');
%
% On the other hand, tables must be created with lower case names
% such as the following statement
%
% CREATE TABLE "p"(c1 varchar(1),c2 varchar(1))
%
% This is to be able to use lower case names with XSB
% so the extra " """ and """ " where entered beside the
% table name
g_from([R-A])           --> !," """, g_atom(R),""" ", " ", g_alias(A).
g_from([R-A|T])         --> " """,g_atom(R),""" ", " ", g_alias(A), ",", g_from(T).
%#################################################################

g_where(CL)             --> " WHERE ", g_condlist(CL), !.
g_where(_)              --> "".

g_condlist([X])         --> g_cond(X), !.
g_condlist([H|T])       --> g_cond(H), " AND ", g_condlist(T), !.

g_cond(no(sel(S,F,W))) --> "NOT EXISTS (", g_selstat(sel(S,F,W)), ")".
g_cond(Cmp)             --> {Cmp=..[R,X,Y]}, g_expr(X), g_atom(R), g_expr(Y).

g_alias(A)              --> "a", g_atom(A).

g_atom(N)               --> {name(N,Str)}, Str.
```

```
%#####################################################
% Main predicate
%
% Alexander Celle T.
% little modifications, it now
% returns a list of ascii code 'Formula not Allowed'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% fol2sql(+Query,-SQLString)
fol2sql(Query,SQLString) :-
  normalize(Query,0,[],NQuery,_,_),
  (alwd(NQuery,[]) ->   %ACELLE V por []
    (trans_query(NQuery,0,SQLTerm,_),
    g_selstat(SQLTerm,SQLString,[]));
    (SQLString = [70,111,114,109,117,108,97,32,110,111,116,32,97,108,108,111,119,101,100])).

%#################################################
% Alexander Celle T.
%
% Predicates originally in other modules put here
% for convenience (other modules are not used).
%
% Some predicates have been stripped of
% unused parts.  Also, atom_chars in XSB aborts
% when failing so it had to be bypassed with atom/1.
%#################################################
variable(V):-
    type_variable(V,TV),
    term_types(TVs),
    member(TV,TVs).

type_variable(V,TV):- \+ number(V),once(type_variable_1(V,TV)).

type_variable_1(V,TV):-
    atom(V)->
        (atom_chars(V,LV),
        append(TVL,LY,LV),
        number_chars(NY,LY),
        NY>0,
        atom_chars(TV,TVL),
        term_types(TVs),
        member(TV,TVs));
        (fail).

%######################
% Alexander Celle T.
%
% Allowed terms:
%   id: free variables
%   u: universal vars
%######################
term_types([id,u,exists]).
```

B. ODBC PATCH

Thanks to Baoqiu Cui for his advice and supplying the following patch, which is a new version of file `XSB/emu/odbc_xsb.c`:

```
/* File:      odbc_xsb.c
** Author(s): Lily Dong
** Contact:   xsb-contact@cs.sunysb.edu
**
** Copyright (C) The Research Foundation of SUNY, 1986, 1993-1998
**
** XSB is free software; you can redistribute it and/or modify it under the
** terms of the GNU Library General Public License as published by the Free
** Software Foundation; either version 2 of the License, or (at your option)
** any later version.
**
** XSB is distributed in the hope that it will be useful, but WITHOUT ANY
** WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
** FOR A PARTICULAR PURPOSE.  See the GNU Library General Public License for
** more details.
**
** You should have received a copy of the GNU Library General Public License
** along with XSB; if not, write to the Free Software Foundation,
** Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
**
** $Id: odbc_xsb.c,v 1.4 1999/12/30 06:21:13 cbaoqiu Exp $
**
*/

#include "configs/config.h"

#ifdef WIN_NT
#include <windows.h>
#include <SQL.H>
#include <SQLEXT.H>
#include <odbcinst.h>
#endif

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#include "cinterf.h"
#include "cell_xsb.h"
#include "error_xsb.h"
#include "export.h"

#define MAXCURSORNUM                    20
#define MAXCOLS                         100
#define MAXNUMPRECISION                 15
#define MAXNUMSTRINGSIZE                (MAXNUMPRECISION + 5)
#define MAXCHARLEN                      100
#define MAXBINDVALLEN                   100
#define MAXI(a,b)                       ((a)>(b)?(a):(b))

static char     *str[4] = {"NULL","string","integer", "number"};
```

```
static int      serverConnected = 0;

HENV henv;
HDBC hdbc;
HSTMT hstmt;
UCHAR uid[128];

struct Cursor {
  int Status;           // status of the cursor
  int StmtNum;          // the number of the sql statement
  UCHAR *Sql;            // pointer to the sql statement
  HSTMT hstmt;           // the statement handler
  int VarListNum;       // distinct bind variable number
  UCHAR **VarList;       // pointer to array of pointers to the actual bind vars
  int *VarTypes;        // types of the distinct bind vars
  int VarCurNum;        // Current Bind Var Number in the distinct Bind var list
  int BListNum;         // number of total bind vars in the sql statement
  UCHAR **BList;        // pointer to array of pointers to the bind vars
  int *BTypes;          // and pointer to their types
  int BCurNum;          // Current Bind Var Number in the total Bind var list
  SWORD ColNum;          // number of columns selected
  SWORD *ColTypes;       // pointer to array of column types
  UDWORD *ColLen;        // pointer to array of column lengths
  UDWORD *OutLen;        // pointer to array of actual column lenghts
  UCHAR **Data;          // pointer to array of pointers to data
  SWORD ColCurNum;       // the cloumn number that's already fetched by xsb
};

// global cursor table
struct Cursor CursorTable[MAXCURSORNUM];

//----------------------------------------------------------------------------
//  FUNCTION NAME:
//      PrintErrorMsg()
//  PARAMETERS:
//      int i - index into the global cursor table
//  NOTES:
//      PrintErrorMsg() prints out the error message that associates
//      with the statement handler of cursor i.  if i is less than 0,
//----------------------------------------------------------------------------
int PrintErrorMsg(int i)
{
  UCHAR FAR *szsqlstate;
  SDWORD FAR *pfnativeerror;
  UCHAR FAR *szerrormsg;
  SWORD cberrormsgmax;
  SWORD FAR *pcberrormsg;
  RETCODE rc;

  szsqlstate=(UCHAR FAR *)malloc(sizeof(UCHAR FAR)*10);
  pfnativeerror=(SDWORD FAR *)malloc(sizeof(SDWORD FAR));
  szerrormsg=(UCHAR FAR *)malloc(sizeof(UCHAR FAR)*SQL_MAX_MESSAGE_LENGTH);
  pcberrormsg=(SWORD FAR *)malloc(sizeof(SWORD FAR));
  cberrormsgmax=SQL_MAX_MESSAGE_LENGTH-1;
  if (i >= 0)
    rc = SQLError(SQL_NULL_HENV, hdbc, CursorTable[i].hstmt, szsqlstate,
                  pfnativeerror, szerrormsg,cberrormsgmax,pcberrormsg);
  else
    rc = SQLError(SQL_NULL_HENV, hdbc, hstmt, szsqlstate,
                  pfnativeerror, szerrormsg,cberrormsgmax,pcberrormsg);
  if ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO)) {
    printf("ODBC SYSCALL ERROR:\n");
    printf("    ODBC Error Code: %s\n", szsqlstate);
```

```
    printf("    ODBC Error Message: %s\n", szerrormsg);
  }
  free(szsqlstate);
  free(pfnativeerror);
  free(szerrormsg);
  free(pcberrormsg);
  return 1;
}

//--------------------------------------------------------------------------
//  FUNCTION NAME:
//      SetCursorClose()
//  PARAMETER:
//      int i - index into the global cursor table
//  NOTES:
//      free all the memory resource allocated for cursor i
//--------------------------------------------------------------------------
void SetCursorClose(int i)
{
  int j;

  SQLFreeStmt(CursorTable[i].hstmt, SQL_CLOSE);    // free statement handler

  if (CursorTable[i].VarListNum) {                 // free bind variable list
    for (j = 0; j < CursorTable[i].VarListNum; j++)
      free((void *)CursorTable[i].VarList[j]);
    free(CursorTable[i].BList);
    free(CursorTable[i].VarList);
    free(CursorTable[i].BTypes);
    free(CursorTable[i].VarTypes);
  }

  if (CursorTable[i].ColNum) {                  // free the resulting row set
    for (j = 0; j < CursorTable[i].ColNum; j++)
      free(CursorTable[i].Data[j]);
    free(CursorTable[i].ColTypes);
    free(CursorTable[i].ColLen);
    free(CursorTable[i].OutLen);
    free(CursorTable[i].Data);
  }

  // free memory for the sql statement associated w/ this cursor
  if (CursorTable[i].Sql) free(CursorTable[i].Sql);
  // initialize the variables.  set them to the right value
  CursorTable[i].Sql = 0;
  CursorTable[i].ColNum =
    CursorTable[i].Status =
    CursorTable[i].VarListNum = 0;
}

//--------------------------------------------------------------------------
//  FUNCTION NAME:
//      ODBCConnect()
//  NOTES:
//      This function is called when a user wants to start a db session,
//      assuming that she doesn't have one open.  It initializes system
//      resources for the new session, including allocations of various things:
//      environment handler, connection handler, statement handlers and then
//      try to connect to the database indicated by the second parameter prolog
//      passes us using the third one as user id and fourth one as password.
//      If any of these allocations or connection fails, function returns a
//      failure code 1.  Otherwise 0.
//--------------------------------------------------------------------------
```

```
void ODBCConnect()
{
  int i;
  UCHAR *server;
  UCHAR *pwd;
  RETCODE rc;

  // if we already have a session open, then ...
  if (serverConnected) {
    xsb_error("A session is already open");
    ctop_int(5, 1);
    return;
  }

  // allocated environment handler
  rc = SQLAllocEnv(&henv);
  if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
    xsb_error("Environment allocation failed");
    ctop_int(5, 1);
    return;
  }

  // allocated connection handler
  rc = SQLAllocConnect(henv, &hdbc);
  if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
    SQLFreeEnv(henv);
    xsb_error("Connection Resources Allocation Failed");
    ctop_int(5, 1);
    return;
  }

  // get server name, user id and password
  server = (UCHAR *)ptoc_string(2);
  strcpy(uid, (UCHAR *)ptoc_string(3));
  pwd = (UCHAR *)ptoc_string(4);

  // connect to database
  rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
  if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    xsb_error("Connection to server %s failed", server);
    ctop_int(5, 1);
    return;
  }

  if (!((rc=SQLAllocStmt(hdbc,&hstmt))==SQL_SUCCESS) ||
      (rc==SQL_SUCCESS_WITH_INFO)) {
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    ctop_int(5, 1);
    return;
  }

  // initialize cursor table. it includes statement handler initialization
  memset(CursorTable, 0, sizeof(struct Cursor) * MAXCURSORNUM);
  for (i = 0; i < MAXCURSORNUM; i++) {
    if (!(((rc=SQLAllocStmt(hdbc,&(CursorTable[i].hstmt)))==SQL_SUCCESS) ||
          (rc==SQL_SUCCESS_WITH_INFO))) {
      int j;
      for (j = 0; j < i; j++)
        SQLFreeStmt(CursorTable[j].hstmt,SQL_DROP);
```

```
      SQLDisconnect(hdbc);
      SQLFreeConnect(hdbc);
      SQLFreeStmt(hstmt, SQL_DROP);
      SQLFreeEnv(henv);
      ctop_int(5, 1);
      return;
    }
  }

  serverConnected = 1;
  ctop_int(5, 0);
  return;
}

//--------------------------------------------------------------------------
//  FUNCTION NAME:
//      ODBCDisconnect()
//  NOTES:
//      Disconnect us from the server and free all the system resources we
//      allocated for the session - statement handlers, connection handler,
//      environment handler and memory space.
//--------------------------------------------------------------------------
void ODBCDisconnect()
{
  int i;

  if (!serverConnected) return;

  for (i = 0; i < MAXCURSORNUM; i++) {
    if (CursorTable[i].Status)
      SetCursorClose(i);
    SQLFreeStmt(CursorTable[i].hstmt,SQL_DROP);
  }

  SQLFreeStmt(hstmt, SQL_DROP);
  SQLDisconnect(hdbc);
  SQLFreeConnect(hdbc);
  SQLFreeEnv(henv);
  serverConnected = 0;
}

//--------------------------------------------------------------------------
//  FUNCTION NAME:
//      FindFreeCursor()
//  NOTES:
//      Find a free statement handler and return its index number into the
//      global cursor table.  It gives priority to a closed cursor with same
//      stantement number over ordinary closed cursors.  If there is no handler
//      left,  function returns -1. possible cursor status values are
//      0 - never been used - no resource associated w/ the cursor
//      1 - used before but having been closed-the cursor has all the resource
//      2 - reusing a used cursor w/ the same statement number, no resource
//          needs to be allocated
//      3 - using a cursor that has no resource - it needs to be allocated
//--------------------------------------------------------------------------
void FindFreeCursor()
{
  int i, j = -1, k = -1;
  int StmtNum = ptoc_int(2);

  // search
  for (i = 0; i < MAXCURSORNUM; i++) {
    if (!CursorTable[i].Status) j = i;        // a cursor never being used
```

```
    else {
      if (CursorTable[i].Status == 1) {    // a closed cursor
        // and it had the same statement number as this one. so grab it
        if (CursorTable[i].StmtNum == StmtNum) {
          if (StmtNum < 2) {
            SetCursorClose(i);
            CursorTable[i].Status = 3;
          } else
            CursorTable[i].Status = 2;
          ctop_int(3, i);
          return;
        } else k = i;                      // otherwise just record it
      }
    }
  }

  // done w/ the search and we didn't find a reusable one
  if ((j < 0) && (k < 0))                       // no cursor left
    i = -1;
  else {
    // we give the cursor that has never been used the priority
    if ((i = j) < 0) SetCursorClose(i = k);
    CursorTable[i].StmtNum = StmtNum;
    CursorTable[i].Status = 3;
  }
  ctop_int(3, i);
  return;
}

//-----------------------------------------------------------------------------
//  FUNCTION NAME:
//     SetBindVarNum()
//  NOTES:
//      set the number of different bind variables and their total number of
//      occurrances in the sql statement to VarListNum and BListMum
//      respectively and allocate memory for furture use, i.e. for holding the
//      bind variables' types and array of pointers to their value.  note that
//      the memory to store their values is not allocated here since we don't
//      know their type:
//      no information on how much memory is needed.  if we're reusing an old
//      statement handler we don't have to worry about these things.  all we
//      need to do is to make sure that the statement is in deed the same
//      statement w/ the same bind variable number.
//-----------------------------------------------------------------------------
void SetBindVarNum()
{
  int i = ptoc_int(2);

  if (CursorTable[i].Status == 2) {
    if (CursorTable[i].VarListNum != ptoc_int(3))
      xsb_exit("In SetBindVarNum: CursorTable[i].VarListNum != ptoc_int(3)");
    if (CursorTable[i].BListNum != ptoc_int(4))
      xsb_exit("In SetBindVarNum: CursorTable[i].BListNum != ptoc_int(4)");
    return;
  }

  CursorTable[i].VarListNum = ptoc_int(3);
  CursorTable[i].VarList = malloc(sizeof(UCHAR *) * CursorTable[i].VarListNum);
  if (!CursorTable[i].VarList)
    xsb_exit("Not enough memory for CursorTable[i].VarList!");
  CursorTable[i].VarTypes = malloc(sizeof(int) * CursorTable[i].VarListNum);
  if (!CursorTable[i].VarTypes)
    xsb_exit("Not enough memory for CursorTable[i].VarTypes!");
```

```
  CursorTable[i].BListNum = ptoc_int(4);
  CursorTable[i].BList = malloc(sizeof(UCHAR *) * CursorTable[i].BListNum);
  if (!CursorTable[i].BList)
    xsb_exit("Not enough memory for CursorTable[i].BList!");
  CursorTable[i].BTypes = malloc(sizeof(int) * CursorTable[i].BListNum);
  if (!CursorTable[i].BTypes)
    xsb_exit("Not enough memory for CursorTable[i].BTypes!");
  CursorTable[i].BCurNum = 0;
}

//---------------------------------------------------------------------------
//  FUNCTION NAME:
//      SetVar()
//  NOTES:
//      set the bind variables' values.
//        allocate memory if it is needed(status == 3)
//---------------------------------------------------------------------------
void SetVar()
{
  int i = ptoc_int(2);
  int j = atoi(ptoc_string(3)+4);

  if (!((j > 0) && (j <= CursorTable[i].VarListNum)))
    xsb_exit("Abnormal argument in SetVar!");

  // if we're reusing an opened cursor w/ the statement number
  if (CursorTable[i].Status == 2) {
    if (CursorTable[i].VarTypes[j-1] != ptoc_int(5))
      xsb_exit("CursorTable VarTypes error!");
    switch (CursorTable[i].VarTypes[j-1]) {
    case 0:
      *((int *)CursorTable[i].VarList[j-1]) = ptoc_int(4);
      break;
    case 1:
      *((float *)CursorTable[i].VarList[j-1]) = (float)ptoc_float(4);
      break;
    case 2:
      strncpy(CursorTable[i].VarList[j-1], ptoc_string(4), MAXBINDVALLEN);
      (CursorTable[i].VarList[j-1])[MAXBINDVALLEN - 1] = 0;
      break;
    default:
      xsb_exit("Unknown bind variable type, %d", CursorTable[i].VarTypes[j-1]);
    }
    return;
  }

  // otherwise, memory needs to be allocated in this case
  switch (CursorTable[i].VarTypes[j-1] = ptoc_int(5)) {
  case 0:
    CursorTable[i].VarList[j-1] = (UCHAR *)malloc(sizeof(int));
    if (!CursorTable[i].VarList[j-1])
      xsb_exit("Not enough memory for an int in SetVar!");
    *((int *)CursorTable[i].VarList[j-1]) = ptoc_int(4);
    break;
  case 1:
    CursorTable[i].VarList[j-1] = (UCHAR *)malloc(sizeof(float));
    if (!CursorTable[i].VarList[j-1])
      xsb_exit("Not enough memory for a float in SetVar!");
    *((float *)CursorTable[i].VarList[j-1]) = (float)ptoc_float(4);
    break;
  case 2:
    CursorTable[i].VarList[j-1] = (UCHAR *)malloc(sizeof(char) * MAXBINDVALLEN);
    if (!CursorTable[i].VarList[j-1])
```

```
      xsb_exit("Not enough memory for MAXBINDVALLEN chars in SetVar!");
    strncpy(CursorTable[i].VarList[j-1], ptoc_string(4), MAXBINDVALLEN);
    CursorTable[i].VarList[j-1][MAXBINDVALLEN - 1] = 0;
    break;
  default:
    xsb_exit("Unknown bind variable type, %d", CursorTable[i].VarTypes[j-1]);
  }
}

//---------------------------------------------------------------------------
//  FUNCTION NAME:
//      SetBind()
//  NOTES:
//      set the bind variables' values for each occurrance of the bind
//      variables in the sql statement.
//---------------------------------------------------------------------------
void SetBind()
{
  int i = ptoc_int(2);
  int j = atoi(ptoc_string(3)+4);

  if (!((j > 0) && (j <= CursorTable[i].VarListNum)))
    xsb_exit("Abnormal argument in SetBind!");

  if (CursorTable[i].Status == 2) return;                // did this already

  CursorTable[i].BList[CursorTable[i].BCurNum] = CursorTable[i].VarList[--j];
  CursorTable[i].BTypes[(CursorTable[i].BCurNum)++] = CursorTable[i].VarTypes[j];
}

int DescribeSelectList(int);

//---------------------------------------------------------------------------
//  FUNCTION NAME:
//      Parse()
//  NOTES:
//      parse the sql statement and submit it to DBMS to execute.  if all these
//      succeed, then prepare for resulting row fetching.  this includes
//      determination of column number in the resulting rowset and the length
//      of each column and memory allocation which is used to store each row.
//      Note indices -3 and -2 are reserved for transaction control(rollback
//      and commit) and index -1 is for those sql statements that don't need
//      cursor, i.e. they can be executed directly.
//---------------------------------------------------------------------------
void Parse()
{
  int j;
  int i = ptoc_int(2);
  RETCODE rc;
  UWORD TablePrivilegeExists;

  if (!((i >= -3) && ( i < MAXCURSORNUM)))
    xsb_exit("Abnormal argument in Parse!");

  switch (i) {
  case (-3):                // index = -3; special case for rollback
    if (((rc=SQLTransact(henv,hdbc,SQL_ROLLBACK)) == SQL_SUCCESS) ||
        (rc == SQL_SUCCESS_WITH_INFO)) {
      for (i = 0; i < MAXCURSORNUM; i++) {
        if (CursorTable[i].Status)
          SetCursorClose(i);
      }
      ctop_int(4,0);
```

```
  } else
    ctop_int(4, PrintErrorMsg(-1));
  return;
case (-2):               // index = -2; special case for commit
  if (((rc=SQLTransact(henv,hdbc,SQL_COMMIT)) == SQL_SUCCESS) ||
      (rc == SQL_SUCCESS_WITH_INFO)) {
    for (i = 0; i < MAXCURSORNUM; i++) {
      if (CursorTable[i].Status)
        SetCursorClose(i);
    }
    ctop_int(4,0);
  } else
    ctop_int(4,PrintErrorMsg(-1));
  return;
case (-1):          // index = -1; special case for odbc_sql; no return rows
  if (((rc=SQLExecDirect(hstmt,ptoc_string(3),SQL_NTS)) == SQL_SUCCESS) ||
      (rc == SQL_SUCCESS_WITH_INFO))
    ctop_int(4,0);
  else
    ctop_int(4,PrintErrorMsg(-1));
  return;
default: ;
}
switch (CursorTable[i].StmtNum) {
case (0):             // column information retrieval
  if (((rc=SQLColumns(CursorTable[i].hstmt,
                      NULL, 0,
                      NULL, 0,
                      ptoc_string(3), SQL_NTS,
                      NULL,0)) == SQL_SUCCESS) ||
      (rc == SQL_SUCCESS_WITH_INFO)) {
    ctop_int(4,DescribeSelectList(i));
  } else {
    ctop_int(4,PrintErrorMsg(i));
    SetCursorClose(i);
  }
  return;
case (-1):             // all the table names in this database
  if (((rc=SQLTables(CursorTable[i].hstmt,
                     NULL, 0,
                     NULL, 0,
                     NULL, 0,
                     NULL, 0)) == SQL_SUCCESS) ||
      (rc == SQL_SUCCESS_WITH_INFO))
    ctop_int(4,DescribeSelectList(i));
  else {
    ctop_int(4,PrintErrorMsg(i));
    SetCursorClose(i);
  }
  return;
case (-2):             // user accessable table names,
  // since some ODBC drivers don't implement the function SQLTablePrivileges
  // we check it first
  SQLGetFunctions(hdbc, SQL_API_SQLTABLEPRIVILEGES, &TablePrivilegeExists);
  if (!TablePrivilegeExists) {
    printf("Privilege concept does not exist in this DVMS: you probably can access any
     of the existing tables\n");
    ctop_int(4, 2);
    return;
  }
  if (((rc=SQLTablePrivileges(CursorTable[i].hstmt,
                              NULL, 0,
                              NULL, 0,
```

```
                                NULL, 0)) == SQL_SUCCESS) ||
        (rc == SQL_SUCCESS_WITH_INFO))
      ctop_int(4,DescribeSelectList(i));
    else {
      ctop_int(4,PrintErrorMsg(i));
      SetCursorClose(i);
    }
    return;
  default: ;
  }
  if (CursorTable[i].Status == 2) {
    rc = SQLCancel(CursorTable[i].hstmt);
    if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO)) {
      ctop_int(4, PrintErrorMsg(i));
      SetCursorClose(i);
      return;
    }
  } else {
    CursorTable[i].Sql = (UCHAR *)strdup(ptoc_string(3));
    if (!CursorTable[i].Sql)
      xsb_exit("Not enough memory for strdup in Parse!");

    if (SQLPrepare(CursorTable[i].hstmt, CursorTable[i].Sql, SQL_NTS)
        != SQL_SUCCESS) {
      ctop_int(4,PrintErrorMsg(i));
      SetCursorClose(i);
      return;
    }

    // set the bind variables
    for (j = 0; j < CursorTable[i].BListNum; j++) {
      if (CursorTable[i].BTypes[j] == 2)
        // we're sloppy here.  it's ok for us to use the default values
        rc = SQLSetParam(CursorTable[i].hstmt, (short)(j+1), SQL_C_CHAR, SQL_CHAR,
                         MAXCHARLEN, 0,(char *) CursorTable[i].BList[j], NULL);
      else if (CursorTable[i].BTypes[j] == 1)
        rc = SQLSetParam(CursorTable[i].hstmt, (short)(j+1), SQL_C_FLOAT, SQL_FLOAT,
                         0, 0, (float *)(CursorTable[i].BList[j]), NULL);
      else
        rc = SQLSetParam(CursorTable[i].hstmt, (short)(j+1), SQL_C_SLONG, SQL_INTEGER,
                         0, 0, (int *)(CursorTable[i].BList[j]), NULL);
      if (rc != SQL_SUCCESS) {
        ctop_int(4,PrintErrorMsg(i));
        SetCursorClose(i);
        return;
      }
    }
  }
  // submit it for execution
  if (SQLExecute(CursorTable[i].hstmt) != SQL_SUCCESS) {
    ctop_int(4,PrintErrorMsg(i));
    SetCursorClose(i);
    return;
  }
  ctop_int(4,DescribeSelectList(i));
  return;
}

//---------------------------------------------------------------------------
//  FUNCTION NAME:
//      DisplayColSize()
//  PARAMETERS:
//      SWORD coltype - column type which is a single word.
```

```
//      UDWORD collen - column length which is returned by SQLDescribeCol
//      UCHAR *colname - pointer to column name string
//  RETURN VALUE:
//      column length - the size of memory that is needed to store the column
//      value for supported column types
//      0             - otherwise
//-----------------------------------------------------------------------------
UDWORD DisplayColSize(SWORD coltype, UDWORD collen, UCHAR *colname)
{

  switch (coltype) {
  case SQL_CHAR:
  case SQL_VARCHAR:
    return(MAXI(collen, strlen((char *) colname)));
  case SQL_SMALLINT:
    return(MAXI(6, strlen((char *)colname)));
  case SQL_INTEGER:
    return(MAXI(11, strlen((char *)colname)));
  case SQL_DECIMAL:
  case SQL_NUMERIC:
  case SQL_REAL:
  case SQL_FLOAT:
  case SQL_DOUBLE:
    return(MAXI(MAXNUMSTRINGSIZE, strlen((char *)colname)));
  case SQL_DATE:
  case SQL_TIME:
  case SQL_TIMESTAMP: return 32;
  default: ;
  }
  return 0;
}

//-----------------------------------------------------------------------------
//  FUNCTION NAME:
//      DescribeSelectList()
//  PARAMETERS:
//      int i - cursor number, the index into the global cursor table
//  RETURN VALUES:
//      0 - the result row has at least one column and
//      1 - something goes wrong, we can't retrieve column information, memory
//          allocation fails (if this happens program stops).
//      2 - no column is affected
//  NOTES:
//      memory is also allocated for future data storage
//-----------------------------------------------------------------------------
int DescribeSelectList(int i)
{
  int j;
  UCHAR colname[50];
  SWORD colnamelen;
  SWORD scale;
  SWORD nullable;
  UDWORD collen;

  CursorTable[i].ColCurNum = 0;
  SQLNumResultCols(CursorTable[i].hstmt, &(CursorTable[i].ColNum));
  if (!(CursorTable[i].ColNum)) return 2;   // no columns are affected

  // if we aren't reusing a closed statement hand, we need to get
  // resulting rowset info and allocate memory for it
  if (CursorTable[i].Status != 2) {
    CursorTable[i].ColTypes =
      (SWORD *)malloc(sizeof(SWORD) * CursorTable[i].ColNum);
```

```
    if (!CursorTable[i].ColTypes)
      xsb_exit("Not enough memory for ColTypes!");

    CursorTable[i].Data =
      (UCHAR **)malloc(sizeof(char *) * CursorTable[i].ColNum);
    if (!CursorTable[i].Data)
      xsb_exit("Not enough memory for Data!");

    CursorTable[i].OutLen =
      (UDWORD *)malloc(sizeof(UDWORD) * CursorTable[i].ColNum);
    if (!CursorTable[i].OutLen)
      xsb_exit("Not enough memory for OutLen!");

    CursorTable[i].ColLen =
      (UDWORD *)malloc(sizeof(UDWORD) * CursorTable[i].ColNum);
    if (!CursorTable[i].ColLen)
      xsb_exit("Not enough memory for ColLen!");

    for (j = 0; j < CursorTable[i].ColNum; j++) {
      SQLDescribeCol(CursorTable[i].hstmt, (short)(j+1), (UCHAR FAR*)colname,
                     sizeof(colname), &colnamelen,
                     &CursorTable[i].ColTypes[j],
                     &collen, &scale, &nullable);
      colnamelen = (colnamelen > 49) ? 49 : colnamelen;
      colname[colnamelen] = '\0';
      if (!((CursorTable[i]).ColLen[j] =
            DisplayColSize(CursorTable[i].ColTypes[j],collen,colname))) {
        CursorTable[i].ColNum = j;
        // let SetCursorClose function correctly free all the memory allocated
        // for Data storage: CursorTable[i].Data[j]'s
        SetCursorClose(i);
        return(1);
      }
      CursorTable[i].Data[j] =
        (UCHAR *) malloc(((unsigned) CursorTable[i].ColLen[j]+1)*sizeof(UCHAR));
      if (!CursorTable[i].Data[j])
        xsb_exit("Not enough memory for Data[j]!");
    }
  }
  // bind them
  for (j = 0; j < CursorTable[i].ColNum; j++)
    SQLBindCol(CursorTable[i].hstmt, (short)(j+1), SQL_C_CHAR, CursorTable[i].Data[j],
               CursorTable[i].ColLen[j], (SDWORD FAR *)(&(CursorTable[i].OutLen[j])));
  return 0;
}

//---------------------------------------------------------------------------
//  FUNCTION NAME:
//      FetchNextCol()
//  NOTES:
//      fetch next result rowset.  if we're retrieving user accessable table
//      names, we fetch until we get next user accessable table name
//---------------------------------------------------------------------------
void FetchNextCol()
{
  int i = ptoc_int(2);
  RETCODE rc = SQLFetch(CursorTable[i].hstmt);

  // get user accessable table name
  if ((CursorTable[i].StmtNum == (-2))) {
    while (((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO))
           && (CursorTable[i].OutLen[1] != SQL_NULL_DATA)
           && (strncmp(CursorTable[i].Data[1], uid, strlen(uid))
```

```
              || (CursorTable[i].OutLen[1] != strlen(uid)))
          && (strncmp(CursorTable[i].Data[4], uid, strlen(uid))
              || (CursorTable[i].OutLen[4] != strlen(uid))))
      rc = SQLFetch(CursorTable[i].hstmt);
  }
  if ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO))
    ctop_int(3,0);
  else if (rc == SQL_NO_DATA_FOUND){
    CursorTable[i].Status = 1; // done w/fetching. set cursor status to unused
    ctop_int(3,1);
  }
  else {
    SetCursorClose(i);         // error occured in fetching
    ctop_int(3,2);
  }
  return;
}

//-----------------------------------------------------------------------------
//  FUNCTION NAME:
//      GetColumn()
//  NOTES:
//      get the next column.  special care is taken if table information is
//      needed(statement number -2, -1 and 0) since we actually fetch more than
//      what we need. unfortunately it's inevitable.  we discard unwanted
//      columns.  for column info of a table, we need the fourth and fifth
//      columns(statement no 0). for table names in the database and user
//      accessable table names, we only need the third column(statement -1 and
//      -2).
//-----------------------------------------------------------------------------
void GetColumn()
{
  int i = ptoc_int(2);
  int ColCurNum;
  UDWORD len;

  // if table information is retrieved, special care has to be paid
  // we set the ColCurNum to some appropriate value to get the columns we need
  // and discard unwanted
  switch (CursorTable[i].StmtNum) {
  case (0):
    if (CursorTable[i].ColCurNum <= 3)
      CursorTable[i].ColCurNum = 3;
    else {
      if (CursorTable[i].ColCurNum > 4)
        CursorTable[i].ColCurNum = CursorTable[i].ColNum;
    }
    break;
  case (-1):
  case (-2):
    if (CursorTable[i].ColCurNum <= 2)
      CursorTable[i].ColCurNum = 2;
    else
      CursorTable[i].ColCurNum = CursorTable[i].ColNum;
    break;
  default: ;
  }

  if (CursorTable[i].ColCurNum == CursorTable[i].ColNum) {
    // no more columns in the result row
    CursorTable[i].ColCurNum = 0;
    ctop_int(4,1);
    return;
```

```
}

// get the data
ColCurNum = CursorTable[i].ColCurNum;
if (CursorTable[i].OutLen[ColCurNum] == SQL_NULL_DATA) {
  // column value is NULL
  CursorTable[i].ColCurNum++;
  ctop_string(3,string_find(str[0],1));
  ctop_int(4,0);
  return;
}

// convert the column string to a C string
len = ((CursorTable[i].ColLen[ColCurNum] < CursorTable[i].OutLen[ColCurNum])?
       CursorTable[i].ColLen[ColCurNum]:CursorTable[i].OutLen[ColCurNum]);
*(CursorTable[i].Data[ColCurNum]+len) = '\0';

// pass the result to Prolog if statement is 0, the column type of a table
// is actually an integer, convert it to to corresponding string
if ((!CursorTable[i].StmtNum) && (ColCurNum == 4)) {
  switch (atoi(CursorTable[i].Data[ColCurNum])) {
  case SQL_DATE:
  case SQL_TIME:
  case SQL_TIMESTAMP:
  case SQL_CHAR:
  case SQL_VARCHAR:
    ctop_string(3, string_find(str[1],1));
    break;
  case SQL_SMALLINT:
  case SQL_INTEGER:
    ctop_string(3, string_find(str[2],1));
    break;
  case SQL_DECIMAL:
  case SQL_NUMERIC:
  case SQL_REAL:
  case SQL_FLOAT:
  case SQL_DOUBLE:
    ctop_string(3,string_find(str[3],1));
  }
  CursorTable[i].ColCurNum++;
  ctop_int(4,0);
  return;
}
// otherwise convert the string to either integer, float or string
// according to the column type and pass it back to Prolog
switch (CursorTable[i].ColTypes[ColCurNum]) {
case SQL_DATE:
case SQL_TIME:
case SQL_TIMESTAMP:
case SQL_CHAR:
case SQL_VARCHAR:
  ctop_string(3, string_find(CursorTable[i].Data[ColCurNum],1));
  break;
case SQL_SMALLINT:
case SQL_INTEGER:
  ctop_int(3,atoi(CursorTable[i].Data[ColCurNum]));
  break;
case SQL_DECIMAL:
case SQL_NUMERIC:
case SQL_REAL:
case SQL_FLOAT:
case SQL_DOUBLE:
  ctop_float(3,atof(CursorTable[i].Data[ColCurNum]));
```

```
  }

  CursorTable[i].ColCurNum++;
  ctop_int(4,0);
  return;
}
```