Designing SQUIRREL: an extended SQL for a deductive database system

K. G. WAUGH, M. H. WILLIAMS, Q. KONG, S. SALVINI AND G. CHEN
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

One of the problems facing the designers of a deductive database is that of the choice of query language. The deductive database model is seen as a natural progression from the relational model; the query language should reflect this. This paper reports the decisions taken while designing an extended form of SQL, called SQUIRREL, as the query language for a Prolog-based deductive database. The extensions relate to the inclusion of both rules and incomplete information in the query language and result in changes to the data definition, data manipulation and query languages of SQL. The extensions were constrained by the desire to retain the existing SQL texture in the new language while introducing concepts such as rules, rule manipulation and incomplete information which are alien to the relational philosophy. The language we describe is being used as the interface language for an implementation of a deductive database which will run on a version of Prolog developed to handle database applications.

Received May 1989, revised December 1989

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

1. INTRODUCTION

The relational model which was introduced by Codd,[1] and underlies relational databases, is now well established, and a number of different query languages have been developed for relational systems. More recently the logic database model has been proposed[2-4] as a natural successor to the relational model. This model is an upwardly compatible extension of the relational model. However, as yet there has been little work done on designing query languages for deductive or logic database systems based on the logic database model.

One significant difference between the approaches lies in the basic unit used by each. In the case of the relational model the basic unit is the **tuple**, sets of which are grouped together to form **relations**; in the logic database model the unit is the **clause**, sets of which make up **procedures**. A clause may be a ground fact which corresponds to the notion of a tuple, or it may be some form of rule. This difference is a fundamental one as it affects the basic strategy for query evaluation in the two models. In the relational model all tuples are stored explicitly and require no evaluation; by contrast the clauses stored in a deductive database are formulae which generate explicit tuples when they are evaluated. Thus the query language for a deductive database must permit a distinction to be made between the clauses (rules and facts) as objects stored in the database and the objects which are produced when these clauses are evaluated.

The functional characteristics of a logic database system make it an appropriate vehicle for realising some aspects of knowledge-based systems. The distinction between a logic database and a knowledge base is a fine one. Database systems have been developed for the storage and retrieval of large volumes of data in a safe and efficient manner. Interfaces to such systems are generally based on a simple (preferably high-level) approach which permits different applications to access data simultaneously, returning the data in the form in which each application needs it. An important function of a Database Management System (DBMS) is to ensure the security and integrity of data stored in the database through appropriate authorization and concurrency controls. A logic database system is a natural successor to the relational database system, based firmly on the theory of first-order logic as opposed to the relational theory typically found in DBMSs. Traditional relational databases permit only data (information) to be manipulated; by using first-order logic it becomes possible for more complex units of information (the foundation for a knowledge base) to be manipulated.

A Knowledge Base Management System (KBMS) is similar in concept to the DBMS. Just as a DBMS is responsible for the storage and management of data (in one or more internal databases), a KBMS is a repository for knowledge which is stored in one or more knowledge bases. In the case of a KBMS, different users and application programs should be able to access the internal knowledge bases it maintains in order to retrieve, store and update knowledge. The KBMS contrasts with the idea of a Knowledge Base (KB), which frequently refers only to a single user system for storing and retrieving knowledge. Such KBs often use rigid knowledge formats appropriate only to the application for which they were designed. If the KBMS is to parallel the DBMS approach the knowledge base should be representation (format)-independent, capable of returning 'knowledge' in a format usable directly by the application that requests it. This suggests that the applications using such a system would be able to access the contents of a KBMS in a high-level, representation-independent manner, and specify the representations to be used for the knowledge returned by the KBMS. Clearly, such a requirement will depend upon agreement as to the useful encoding of knowledge units. At this stage the ideas of KBMSs are still being formulated, and there is no clear theoretical framework on which the representation formats should be based.

Returning to the idea of a logic or deductive database, one obvious factor in designing a query language for such a database is the desirability for it to be an upwardly compatible extension of a relational database query language. In particular, the one relational query language which has emerged as a standard for communicating with relational database systems is SQL,[5] and compatibility with SQL would clearly be advantageous.

This paper describes some of the design decisions taken in producing a query language for a deductive database system. The language, SQUIRREL, is a natural extension of SQL. The paper examines how the addition of rules to the SQL language requires a number of significant alterations to the SQL philosophy in order to permit a compact, manageable language without compromising the deductive database capabilities. Alternative views of the database contents are considered and the dual nature (termed syntactic and semantic) of a relation introduced. The problem of indexing and cursor use is examined briefly in the light of this dual nature.

Finally, this paper shows how incomplete information has been incorporated into the SQUIRREL language. The underlying database implementation will permit the manipulation of incomplete values under different evaluation strategies; the language has been designed to facilitate these.

2. THE DEDUCTIVE DATABASE POPULATION

In a relational database each tuple consists of a set of explicit values, such as strings, characters, integers, reals, etc. A relation may be represented by a view, which is a projection of values found in other relations and views. As an example consider the database consisting of family details (Table 1) in which a relation **person** is defined by a table and a relation **parent** by a view. A reference to values in a view or a table is always to some value stored explicitly in some table.

Table 1. Relational database population

Relation: person

name	child	age	address
Mary	John	103	Newcastle
Milly	Billy	33	Sunderland
John	Jack	72	Hartlepool
Bill	NULL	22	Newcastle
Jack	Paula	45	Sunderland
Paul	Karen	34	Sunderland

```
VIEW definition
 CREATE VIEW parent(adult, child)
 AS SELECT name, child
 FROM person
 WHERE child IS NOT NULL
```

In a deductive database a relation may be populated with explicit facts and with rules which express the logical relation that holds between values in the database, therefore defining the condition that a tuple must satisfy for its inclusion in that relation.[2] For example, consider a database of family details in which the relation **person** is defined by a set of tuples and the relation **grandparent** by a set of rules and facts (Table 2). For this section the rules are represented as Prolog clauses, while later the SQUIRREL syntax for rules will be introduced. Notice that the relation grandparent is stored as a set of rules and facts, but represents a set of tuples.

Table 2. Deductive database population

Relation: person(name,child,age,address)

```
person(Mary,John,103,Newcastle).
person(Milly,Billy,33,Sunderland).
person(John,Jack,72,Hartlepool).
person (Bill,NULL,22,Newcastle).
person(Jack,Paula,45,Sunderland).
person(Paul,Karen,34,Sunderland).
```

Relation: grandparent(elder,younger)

```
grandparent(Old,Young):-
 person(Old,Middle,_,_),
 person(Middle,Young,_,_).
grandparent(George,Michael).
grandparent(Susan,Michael).
```

A distinction is drawn between the two types of objects that are involved in query evaluation: syntactic and semantic objects. A relation is populated with syntactic objects (rules, facts and views) which define the semantic objects under some resolution strategy. The semantic objects are those tuples produced by the evaluation of the syntactic objects in the database under some inference mechanism. This is illustrated in Table 3. This categorisation is somewhat different from that of Extensional and Intensional databases (EDB and IDB). In the latter case[6,7] a set of closed formulas can be viewed as having two components, the extensional part which contains all the ground unit formulas and the intensional part which contains all the nonground or non-unit formulas. The combined EDB and IDB are equivalent to the set of syntactic objects of the database. It is essential for the database query language to be able to distinguish between the internal storage of the objects and the objects they represent in addition to being able to distinguish between internal objects.

Table 3. Syntactic vs. semantic objects in a deductive database

Syntactic objects

```
person(Mary,John,103,Newcastle).
person(Milly,Billy,33,Sunderland).
person(John,Jack,72,Hartlepool).
person(Bill,NULL,22,Newcastle).
person(Jack,Paula,45,Sunderland).
person(Paul,Karen,34,Sunderland).

grandparent(Old,Young):-
 person(Old,Middle,_,_),
 person(Middle,Young,_,_).
grandparent(George,Michael).
grandparent(Susan,Michael).
```

Semantic objects

```
person(Mary,John,103,Newcastle).
person(Milly,Billy,33,Sunderland).
person(John,Jack,72,Hartlepool).
person(Bill,NULL,22,Newcastle).
person(Jack,Paula,45,Sunderland).
person(Paul,Karen,34,Sunderland).

grandparent(George,Michael).
grandparent(Susan,Michael).
grandparent(John,Paula).
grandparent(Mary,Jack).
```

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

In a relational database the distinction between syntactic and semantic objects vanishes, and reduces to two different interpretations of the same underlying object. On the other hand, a deductive database management system must provide the facilities whereby a user can manipulate both the syntactic objects stored in the database, and the semantic objects produced by evaluating those stored objects. However, the ways in which these two types of object are manipulated are not fundamentally different: one still wants to select, insert, delete and update syntactic objects in the same way that semantic objects are manipulated in SQL. For example, one may select for display the rules stored in a named relation (excluding any facts stored). One may even wish to combine syntactic and semantic objects in a query. A restriction may be imposed on the syntactic objects (as in the selection above) and, rather than display the resulting rules, one may require the constrained relation to be evaluated in a further subquery.

2.1 Indexing and cursor use in a deductive database

The dual nature of the deductive database population complicates the relational concepts of indexing and cursor use. In a relational database it is the explicit storage of base values that permits indexing and cursor operations. The base representation permits the designation of an ordering of tuples in a relation and the selection of tuples based on positions relative to that ordering. When all tuples in the relations are explicitly present (or referenced through views) this ordering can be achieved, and each tuple has an identifiable next tuple and previous tuple.

However, a deductive database permits the combination of rules and facts to represent a relation. Given a rule which generates a set of tuples, this raises questions such as whether one indexes the rule bodies or the tuples. If one imposes a cursor on a relation containing rules, how does one insert, delete or retrieve tuples relative to that cursor when the cursor currently identifies a tuple generated by a rule? In the light of these difficulties it was felt that an alternative to cursors should be investigated.

2.2 Clause and domain variables

When selecting objects, either for display (`SELECT`) or for update (`DELETE`, `UPDATE`), one must be able to distinguish clearly between the selection of a syntactic or a semantic object as target of the manipulation. To facilitate this, two different types of variables are introduced. These are referred to as clause variables and domain variables, respectively. A clause variable ranges over the syntactic objects stored in a relation, whereas a domain variable ranges over semantic objects and as such is identical to the traditional SQL table and column name reference. The use of these variables will be shown in later sections of this paper.

The domain variable is represented by the SQL column reference of [*table name*.]*column name* or (for the complete relation) *table name*.

The clause variable is formed by adding @ to the front of the table name. Thus, using the database of Table 2, the domain variable **grandparent** will range over the **tuple** set:

```
{(George, Michael), (Susan, Michael),
 (John, Paula), (Mary, Jack)}.
```

whereas the clause variable @**grandparent** ranges over the **clause** set:

```
{grandparent(George,Michael).,
 grandparent(Susan,Michael).,
 grandparent(Old,Young):-
 person(Old,Middle,_,_),
 person(Middle,Young,_,_).}
```

These two sets can be displayed by the statements

```
SELECT*
FROM grandparent
```

and

```
SELECT @grandparent
FROM grandparent
```

3. INSERTING SYNTACTIC OBJECTS INTO THE DATABASE

All objects entered into the database, whether facts or rules, are entered as syntactic objects. This means that the existing SQL insertion of values is seen as inserting facts which are syntactic objects rather than tuples which are semantic objects. Apart from this reclassification of the inserted values, no other alternation is made to the existing insertion statement of SQL.

The major extension to the data manipulation language lies in the statements for the insertion of rules into the database. A number of alternative representations for the rule syntax were examined. They fall broadly into two categories, those that retain the SQL reference by name to manipulate values and those that introduce variables for use in the rule body.

3.1. Reference-by-name approach

SQL is a reference-by-name language, that is, a language in which each data reference has an explicit (and universal) name. For example, the **student** relation (Table 5) contains a column **age** which is referenced as *student.age* or simply as *age* when no ambiguity can arise.

If this reference-by-name philosophy is maintained, it leads to unwieldy rule bodies. To illustrate this, consider the following examples.

Table 4. Examples of possible clause restriction representations

RESTRICT Statement

```
SELECT elder, age
FROM person, grandparent
RESTRICT rules (@grandparent)
WHERE age > 40
```

Clause Restriction Statement

```
SELECT elder, age
FROM person, grandparent: rules
WHERE age > 40
```

Using the relations in Table 2, restrict the grandparent relation to rules only and select the names and ages of the grandparents with ages over 40.

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

Table 5. Database schema

Relation	Column names				
staff	name	age	sex	dept	grade
student	name	sex	age	year	dept
marks	dept	name	num	mark	
record	dept	name	total_courses	avg_mark	
result	dept	name	grade		
teaching_staff	name	dept			
research_staff	name	dept			
ancestor	anc	man			
father	name	dad			

Table 6. Extended cartesian product of parent and benefit relations

parent		benefit	
name	child	name	amount
mark	william	william	10
mark	william	simon	20
mark	william	tom	30
mark	william	mary	40
mary	simon	william	10
mary	simon	simon	20
mary	simon	tom	30
mary	simon	mary	40
mark	tom	william	10
mark	tom	simon	20
mark	tom	tom	30
mark	tom	mary	40

(1) Assuming relations **ancestor** and **father** are defined as *ancestor*(*person*,*elder*) and *father*(*dad*,*son*), the ancestor rules can be inserted as:

```
INSERT INTO ancestor
USING father, ancestor
DEFINE ancestor IF SELECT son, dad
 FROM father
DEFINE ancestor IF SELECT son, elder
 FROM father, ancestor WHERE dad=person
```

The rule body has the form of a select statement to allow multiple assignment to columns in the target relation. The alternative is to have explicit assignments to *ancestor.person* and *ancestor.elder* (*ancestor.person* = *father.son* and *ancestor.elder* = *father.dad* in the first rule body). This would inhibit the reading of a rule as an assignment to a tuple.

(2) Using the **research_staff, teaching_staff** and **staff** definitions in Table 5, the definition of **research_staff** as a rule can be inserted as:

```
INSERT INTO research_staff(name, dept)
USING staff, teaching_staff
DEFINE research_staff IF
  SELECT staff.name, staff.dept
  FROM staff
  WHERE staff.grade <> 'technician' AND
  NOT EXISTS (SELECT name
    FROM teaching_staff
    WHERE staff.name = teaching_
     staff.name AND staff.dept =
     teaching_staff.dept
    )
```

3.2 Prolog-like approach

An alternative approach for the rule formats is to base these on the logic programming language Prolog. Prolog is a reference-by-position language, that is, one which permits a variable to be declared which takes its value from some positional information supplied when it is used. For example, *student*(_,_,*Age*,_) would associate the variable *Age* with the values found in the third argument position of the **student** tuples.

Using this approach, the above examples can be coded in Prolog in the following manner.

(1) The Prolog definition of **ancestor,** assuming *ancestor*(*elder*,*younger*) and *father*(*dad*,*son*) are defined as for Example 1 above, is as follows:

```
ancestor(X,Y):- father(X,Y).
ancestor(X,Y):- father(X,Z),
 ancestor(Z,Y).
```

(2) The Prolog definition of **research_staff,** assuming the schema defined in Table 5, is

```
research_staff(N,D):-
 staff(N,_,_,D,G),
 not (G = 'technician'),
 not (teaching_Staff(N,D)).
```

However, this approach is possible only when the number of columns in a relation and their ordering is known. Clearly in a relational database language this cannot be guaranteed. A user may not have authorized access to all columns of a relation, may not know in what order the columns were defined or what the names of all the columns are; or a relation may be unwieldy in this form, having a large number of columns of which only a few are of interest.

3.3 Intermediate approach

A solution to the difficulty of using positional representations in a relational language which takes advantage of the shorthand offered by the use of variables and unification can be found in SQL. The existing SQL INSERT statement permits the use of an insert column list to define the target columns and the order in which

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

they will be referenced in the query. There is then no need to use all columns or to depend on some default ordering. The insert column list can be used to override the default information.

Example:

```
INSERT INTO staff(name,age,grade)
VALUES ('clare',23, 'technician')
```

Note that columns not referenced in the INTO statement default to the NULL value provided that the relation permits this; the column names that are referenced can be in any order. Thus if the relation actually contains two further fields, *sex* and *dept*, this would be interpreted as if it were the statement:

```
INSERT INTO
 staff(name,age, sex, dept, grade)
VALUES
 ('clare',23,NULL,NULL, 'technician')
```

In SQUIRREL the two reference systems have been combined by permitting the declaration of source and target columns in a reference-by-name manner; this declares the order of those columns for the reference by position employed in the rule bodies. In order to allow the definition of source tables for the rules, a USING clause is added to the INSERT statement. The following examples show how the INSERT USING statement is employed to insert rules into a relation using both reference by name and reference by position. The first example shows how the declaration of source and target relations can be used to override the default order of the columns of the relations used. The second shows how the number of columns used in the rule definition can be restricted to those actually used in the rule bodies. The final example shows how correlation names can be employed, in this instance for the insertion of a recursive rule. (The schema description of the tables used in these examples can be found in Table 5.)

Examples:

(1) Insert rules into the **result** relation.

```
INSERT INTO result(grade,name,dept)
USING   record(dept,name,total_courses,
         avg_mark)
        DEFINE result('excellent',D,S,)
         IF record(D,S,N,A) and N>3
         and A>=80
        DEFINE result('good',D,S)
         IF record(D,S,N,A) and N>3
         and A<80 and A>=60
        DEFINE result('pass',D,S)
         IF record(D,S,N,A) and N>3
         and A<60 and A>=40
        DEFINE result('fail',D,S,)
         IF record(D,S,N,A) and N>3
         and A<40
        DEFINE result ('incomplete',D,S)
         IF record(D,S,N,A) and N<=3
```

(2) Insert a rule defining the **research_staff** relation.

```
INSERT INTO research_staff(name,dept)
USING   staff(name,dept,grade),
         teaching_staff
        DEFINE research_staff(N,D) IF
         staff(N,D,G) and G<>'technician'
         and not teaching_staff(N,D)
```

(3) Insert the recursive rule pair for the **ancestor** relation.

```
INSERT INTO ancestor(anc,man)
USING  father(name,dad),
        ancestor(man,anc) anc1
       DEFINE ancestor(person,elder)
        IF father(person,elder)
       DEFINE ancestor(person,elder)
        IF father(person,parent) and
        anc1(parent,elder)
```

A number of constraints are imposed on the rule bodies, including some on the use of variables in the INSERT and USING statements. The following restrictions apply:

(1) Variables must agree both in position and type with the lists of columns appearing in the relation descriptions in the INSERT and USING statements.

(2) All variables must have a defining use, that is one in which the value of a variable can be bound to a constant. A defining use declares the variable name and (due to its positional use) a type. A defining use of a variable occurs in some primitives and under certain conditions. The primitives permitting a defining use are assignment and use in an argument position in a table reference. Terms of the form, *table_name*(*argument list*), offer defining use for all variables used in the argument list (unless already defined). All variables must agree in type and number with the column types declared in the USING list. Use of a variable in the head of a rule is not a defining use as the head of a rule is considered to be the target of the variables. This prevents unbound variables appearing in the head of a rule and hence in a column of the relation.

(3) Assignment (⟨*var*⟩? ⟨*value expr*⟩) is non-destructive; that is, it assigns the result of the ⟨*value expr*⟩ to the variable on the left of the operator (?) which should not have previously been used in a defining position.

(4) A comparison predicate does not offer a defining use. For example, the comparison expression, $Var < 10$, would offer a set of values for *Var* if it were not already bound to a value.

(5) In a rule body containing a disjunction the defining use of a variable is global if and only if it occurs in both branches of the disjunction and the defining use agrees with respect to type. For example, in '*parent*(*X*,*Y*) *OR grandparent*(*X*,*Z*)', *X* is a defining use if the types of the first columns in parent and grandparent agree. *Y* and *Z* are not global defining uses.

(6) No variables can be defined globally under negation. If a defining use does occur within a negation, this is treated as a local defining use. For example, in '*NOT*(*parent*(*X*,*Y*) *AND* $Y > 10$)', *X* and *Y* are treated as local defining uses. Furthermore, if the negation appears in '*grandparent*(*X*,*Z*) *AND NOT* (*parent*(*X*,*Y*) *AND* $Y > 10$)', *Y* cannot be used elsewhere in the rule, whereas *X* may be used, as its defining use occurs in the argument list of grandparent, which is not negated.

4. QUERY LANGUAGE

Since the query language in SQUIRREL must permit the retrieval of both syntactic and semantic objects, this requires the ability to distinguish between these two aspects of a relation. Furthermore, it is useful to be able

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

to constrain syntactic objects before a relation is evaluated. This is achieved with the clause restriction.

Clause and domain variables are used to indicate the type of result required from the SELECT statement. However, a few constraints must be imposed on the use of clause and domain variables. In the select list of the SELECT statement clause and domain values cannot be mixed, since it does not make sense to permit the mixing of rules and the objects they generate. Similarly, it would be difficult to select clause–variable objects if the table expression produced domain-variable values, so that the use of a clause variable in the select list is not permitted when a table expression (`WHERE`, `GROUP BY` and `HAVING` clause) is present in the query.

In this section of the paper consideration is given to the syntax for manipulation of the syntactic population of the database, before showing how to form queries which return the syntactic objects. The evaluation of a query over a constrained relation is shown and an explanation given of how syntactic constraints and semantic conditions can be combined in a single query. It remains possible to pose typical relational queries, which select semantic objects from the database as if it were a standard relational database.

4.1 Restricting the syntactic objects

Just as the SQL `WHERE` clause selects semantic objects on the basis of some specified condition, one needs to be able to select syntactic objects on the basis of syntactic properties. This can be achieved by a clause restriction; this either takes the form of a separate clause, the `RESTRICT` clause (which is similar to `WHERE`, `GROUP BY`, etc.) or by adding the restriction to the `FROM` clause.

The two forms for restricting objects on the basis of their syntactic structure are illustrated in Table 4. While the first (`RESTRICT` clause) approach appears most SQL-like there are a number of interpretation problems which arise if one needs to restrict more than one relation from within a single clause. The second form removes these problems while strengthening the concept of the `FROM` clause as the source of the syntactic objects used in the evaluation of the user query.

There are two basic problems with the `RESTRICT` clause: inter-relation dependencies and conditions involving independent variables.

(*a*) *Inter-relation dependencies*. The following description of the dependency problem gives some background on the SQL-evaluation of inter-relation dependencies and shows how this would result in problems if it were carried into the clause restriction.

The SQL-processing of the `FROM` list involves the production of the extended cartesian product of the relations. So that for each condition in the table expression the values in the extended cartesian product can be used to examine dependency information. For example, consider the relations *parent*(*name*,*child*) and *benefit*(*name*,*amount*) with the values:

```
parent(mark,william).
parent(mary,simon).
parent(mark,tom).

benefit(william,10).
benefit(simon,20).
benefit(tom,30).
benefit(mary,40).
```

and the query:

```
SELECT   parent.name,amount
FROM     parent,benefit
WHERE    child = benefit.name
```

which lists the benefit amounts for each child and the person to whom it is payable.

The `FROM` list is evaluated to give the extended cartesian product of the two relations; this is shown in Table 6. The evaluation of the `WHERE` condition is then performed by examining each row in the cartesian product table to produce the constrained result (Table 7). The first and fourth columns of Table 7 are projected to give the result of the query.

The effect of the inter-relation dependencies is best demonstrated by examining the result of the (badly formed) query:

```
SELECT   parent.name
FROM     parent,employee
WHERE    parent.child IS NOT NULL
```

which produces the results shown in Table 8.

If this were paralleled in the clause restriction it would greatly increase the complexity of the subsequent evaluation of a constrained relation. Consider that the `RESTRICT` clause is intended to constrain the syntactic objects in the database to allow subsequent evaluation of the constrained rule set. A clause condition statement modelled on the above evaluation strategy would not result in a single set of syntactic objects for each relation, but a collection of sets of syntactic objects.

Consider the following query fragment:

```
FROM       P,Q
RESTRICT   rules(P) and rules(Q) and
                @P.a = @Q.x
```

where P is defined as having columns a and b; Q as having columns x and y.

The intention of the `RESTRICT` statement is to apply restrictions to each **set** of clauses (P and Q). However, the above fragment appears to impose a dependency condition between the sets (that of equality between two columns). Thus, given the initial clauses $P(1,2){:-}A(1,2)$, $P(2,3){:-}A(2,3)$, $P(3,4){:-}A(3,4)$, $Q(1,2){:-}B(1,2)$, $Q(2,3){:-}B(2,3)$ and $Q(3,4){:-}B(3,4)$ the `FROM` statement would produce the cartesian product shown in Table 9.

The `RESTRICT` statement would then eliminate the clauses not satisfying the dependency condition and would produce the full six clauses, but paired into the sets:

```
{P(1,2):-A(1,2), Q(1,2):-B(1,2)}
{P(2,3):-A(2,3), Q(2,3):-B(2,3)}
{P(3,4):-A(3,4), Q(3,4):-B(3,4)}
```

Table 7. Result of applying the WHERE condition to Table 6

parent		benefit	
name	**child**	**name**	**amount**
mark	william	william	10
mary	simon	simon	20
mark	tom	tom	30

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

Table 8. Result of the badly formed query

parent name
mark
mark
mark
mark
mary
mary
mary
mary
mark
mark
mark
mark

Table 9. Cartesian product produced by RESTRICT clause

P	Q
P(1,2):-A(1,2)	Q(1,2):-B(1,2)
P(1,2):-A(1,2)	Q(2,3):-B(2,3)
P(1,2):-A(1,2)	Q(3,4):-B(3,4)
P(2,3):-A(2,3)	Q(1,2):-B(1,2)
P(2,3):-A(2,3)	Q(2,3):-B(2,3)
P(2,3):-A(2,3)	Q(3,4):-B(3,4)
P(3,4):-A(3,4)	Q(1,2):-B(1,2)
P(3,4):-A(3,4)	Q(2,3):-B(2,3)
P(3,4):-A(3,4)	Q(3,4):-B(3,4)

This would require evaluation of 'each' set of clauses to produce the exhaustive set of semantic objects for the clauses satisfying the dependency. The effect of this is not easy to predict, nor is it easily implementable. Thus, any inter-relation dependency information does not introduce a useful restriction on the clause sets. For this reason the clause restriction operates on clauses taken one at a time from the source clauses. This prevents dependency information such as explicit equality being used in the clause restriction.

(*b*) *Independent variables.* If the RESTRICT clause is constrained to conditions with independent variables (that is, inter-relation dependencies are prevented), the RESTRICT clause can be written as either

```
FROM P,Q
RESTRICT F(P) or G(Q)
```

or

```
FROM P,Q
RESTRICT F(P) and G(Q)
```

where F and G are conditions over P and Q respectively. Since F and G are independent and are evaluated for each single-source clause taken from P and then Q, the conjunction of $F(P)$ and $G(Q)$ cannot be satisfied. For this to occur the source clause would need to be a syntactic object in relation P and in relation Q. With independent conditions only the disjunction can be permitted.

For these reasons the RESTRICT clause would require a complex formulation of syntax rules to ensure the above code fragments did not occur. Notice in Table 4 that in the augmented FROM statement it is impossible to contravene the above requirements. The source relation and its restriction are bound tightly, which prevents interdependencies and forces the separation of independent conditions.

The clause restriction is used in the delete and update statements as well as in selection for display purposes. Examples of these statements are given in a later section.

4.2 Clause restriction predicates and functions

A new set of functions and predicates applicable to syntactic objects in the database is required. The following gives an indication of the form that these might take.

4.2.1. Predicates

A useful set of predicates might include:

facts: true for those clauses which are facts.

rules: true for those clauses which are rules.

recursive: true for those clauses which can lead to a recursive call of the relation, under a static search of the call tree.

uses(relation name): true for those clauses which include a call to the named relation.

match(string): true for those clauses that contain the named string in the head or body.

var(column name): true for those clauses with a variable in the named column position in the object head.

null(column name) or **column name is [not] null**: true for those clauses with null in the named column position in the head of the rule.

4.2.2. Functions

The function set should include:

head(clause variable): returns the head structure of the clauses selected by the clause variable.

body(clause variable): returns the body of the clauses selected by the clause variable.

count_clauses(clause variable): returns a count of all syntactic objects represented by the clause variable.

count_rules(clause variable): returns a count of the rules represented by the clause variable.

count_facts(clause variable): returns a count of the facts represented by the clause variable.

4.3 Selecting syntactic objects

Consider how syntactic objects can be retrieved from the database. The clause variable and the clause restriction can be combined to constrain a relation population and then display the constrained set of syntactic objects.

The simplest retrievals are those without any clause restriction imposed on the selection. This is comparable with the 'select*from relation' of SQL. The objects returned are the representations of the syntactic objects stored under a given relation name. An example is:

```
SELECT   @student
FROM     student
```

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

Similarly, one can apply the functions mentioned earlier to the result of the SELECT statement, e.g.

```
SELECT  count_rules(@student)
FROM    student
```

A retrieval may restrict the clauses chosen before display. An example of this is

```
SELECT  @result
FROM    result: grade = 'good'
```

which selects the objects in which the named column position (*grade*) in the head of the clause has the specified value.

The restriction may be more complex than this, e.g.

```
SELECT  @result
FROM    result: grade = 'good' and
          var(dept)
```

in which the *dept* column position in the head must be occupied by a variable and the *grade* position must have the value '*good*'.

Notice (as stated earlier) that since the target of the above queries is a clause variable one cannot have a semantic condition (table expression) as this would result in the evaluation of the syntactic objects.

4.4 Evaluation under the clause restriction

As mentioned previously, it is useful to be able to evaluate a relation which is constrained to syntactic objects satisfying a clause restriction condition. That is, one may constrain a relation (as shown above) and then require that the constrained relation is evaluated to produce semantic objects to be used in a table expression.

In implementing the clause restriction one must ensure that the restriction does not lead to side-effects in other aspects of the query. Thus one must ask how restricting the set of syntactic objects for a particular evaluation of a relation affects other relations that depend on the values produced by the restricted relation. Two alternative approaches are the following.

(1) Temporarily remove from the database all clauses that do not satisfy the restriction. Consider a database *DB* and a query $Q(p,e)$ (i.e. the query is over relations *p* and *e*). If *Q* restricts the relation *p* to p' then $Q(p,e)$ is evaluated over the database $DB-p+p'$. Hence, if *e* contains a procedure using relation *p*, the evaluation of *e* is restricted to those clauses that use the clauses in p'.

(2) Create temporary relations and fill these with the restricted clauses. Consider a database *DB* and the query $Q(p,e)$. If *Q* restricts *p* to p', then copy p' into the database in a marked form so that $DB' = DB+p'$. Now rename the explicit use of relation *p* in *Q* to p', i.e. $Q(p,e) \rightarrow Q(p',e)$ and execute this query over the database DB'. After the execution of the query, remove p' from the database, i.e. $DB' \rightarrow DB$. Now, if *e* contains a rule which uses relation *p* it is able to use all of relation *p* contained in the database.

For example, if the initial database consists of Prolog clauses:

```
ancestor(X,Y) if father(X,Y).
ancestor(X,Z) if father(X,Y),
 ancestor(Y,Z).
father(....etc.
```

Under the first implementation a query *ancestor*(A,B) in which the relation *ancestor* is restricted to only recursive clauses, would result in the evaluation of the query *ancestor*(A,B) over the database:

```
ancestor(X,Z) if father(X,Y),
 ancestor(Y,Z).
father(....etc.
```

Under the second implementation the same query would result in the evaluation of *ancestor_marked*(A,B) over the database:

```
ancestor_marked(X,Z) if
 father(X,Y), ancestor(Y,Z).
ancestor(X,Y) if father(X,Y).
ancestor(X,Z) if father(X,Y),
 ancestor(Y,Z).
father(....etc.
```

The second implementation strategy has been employed in this project for the following reasons.

(*a*) Recursion remains safe under this interpretation. Suppose that under the Prolog resolution strategy the example database above will execute the query *ancestor*(A,B) correctly and will terminate correctly. If the first implementation were used to constrain the database, the *ancestor* query would not terminate. Under the second implementation the query will be executed and terminate correctly.

(*b*) In the database one does not want the effects of a restriction on the syntactic objects of one relation to alter the semantic objects in other relations. That is, in the example query above one does not want a restriction on *p* to affect the semantic objects produced by evaluating *e*. This is similar to an SQL query in which a view over a base table and the base table itself are included in a query. If a *where* statement imposes a condition on the base table the view is not affected.

4.5 Selecting semantic objects

Having seen how a constrained relation can be evaluated, consider how domain variables, clause restriction and the SQL table expression can be combined in the retrieval of semantic objects (tuples).

In these cases one wants the tuple sets of traditional relational database queries to be returned by a query. The simplest example is that of a normal SQL query such as

```
SELECT    name, mark, dept
FROM      marks
GROUP BY  name,dept,mark
HAVING    max(mark) = mark
```

No distinction is necessary when using tables which are defined by rules, for example

```
SELECT  name,depth
FROM    result
WHERE   grade = 'excellent'
```

From the preceding section it has been seen that SQUIRREL will permit a query to be evaluated over a set of clauses taken from a relation definition by using a clause restriction. An example is

```
SELECT  name
FROM    result: facts
WHERE   grade = 'excellent'
```

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

This restricts the evaluation of the query to those syntactic objects which are facts (it excludes any rules used to define the relation **result**).

This can be used to calculate the restricted **ancestor** relation described in the database population section above.

```
SELECT  anc
FROM    ancestor: uses(ancestor)
```

or

```
SELECT  anc
FROM    ancestor: recursive
```

The use of the clause restriction means that some queries can be expressed in two ways, using the clause restriction or using the WHERE clause. For example, the query

```
SELECT  name, dept
FROM    result: grade = 'incomplete'
```

has the same result as the query

```
SELECT  name, dept
FROM    result
WHERE   grade = 'incomplete'
```

However, in the second case the entire relation for **result** must be generated and examined, tuple at a time, in order to compare the grade with the constant '*incomplete*'. In the first case, only those objects that will generate the constant '*incomplete*' in the *grade* position will be used, hence no extra tuples are generated then excluded. (Obviously this depends on the *grade* position being filled with constants, a slightly safer but less efficient approach would be to include the rules which have variables in the *grade* position. That is, `SELECT name, dept FROM result: grade = 'incomplete' or var(grade)`.)

5. OTHER OPERATIONS

The changes to the data definition language and the delete and update statements are introduced in this section. The problems of the update of semantic objects are considered, and parallels are drawn with the difficulty of update through relational views.

5.1 Data definition language

The CREATE TABLE statement is used to define a table. The only change to this statement is the addition of an optional restriction which permits a table to be defined WITHOUT RULES. An example (taken from the database schema in Table 5) is:

```
CREATE TABLE staff(
  name char(20) NOT NULL UNIQUE,
  age integer NOT NULL,
  sex char(1),
  dept char(5),
  grade char(20) NOT NULL
  ) WITHOUT RULES.
```

The main reason for this is for use in optimization processes which are applicable during query evaluation.

5.2 Deletion

There are two possible forms for the deletion statement, either deleting a syntactic object or deleting a semantic object.

(*a*) For the purposes of deleting a syntactic object from a relation, the relation is identified using the clause variable and then the optional clause restriction is applied to restrict the deletion to those clauses one actually wants to remove. To illustrate this, consider the following examples.

(1) Delete all syntactic objects in the **result** relation,

```
DELETE FROM @result
```

(2) Delete all *facts* from the **result** relation,

```
DELETE FROM @result: facts
```

(3) Delete all *rules* from the **research_staff** relation that use the **student** relation and the *facts* that have NULL in the *dept* position.

```
DELETE FROM @research_staff:
 (rules and uses(student)) or
 (facts and null(dept))
```

A semantic condition (WHERE clause) cannot be applied if the target of the deletion is a clause variable. The reason for this is the same as the reason that the table expression statement is not permitted if the target of the select statement is a clause variable.

(*b*) The semantic objects for deletion can be identified using a combination of domain variable, clause restriction and semantic condition (WHERE clause). The syntax for SQUIRREL permits the expression of statements to delete semantic objects. Some examples are considered below.

(1) Delete all semantic objects from relation **result**.

```
DELETE FROM result
```

(2) Delete the semantic objects from relation **result** which have a *grade* column value of '*fail*'.

```
DELETE FROM result
WHERE  grade = 'fail'
```

(3) Delete all semantic objects that are generated by the *facts* for the relation **result**.

```
DELETE FROM result: facts
```

(4) Delete the semantic objects that are generated by *rules* and have a *grade* column value of '*pass*'.

```
DELETE FROM result: rules
WHERE  grade = 'pass'
```

5.3 Update

Updates to the contents of a relation can similarly be divided into those with syntactic or semantic targets.

(*a*) The update of syntactic objects is identified by the use of the clause variable as target for the update. As with select and delete, one cannot have a semantic condition if the target of the update request is a syntactic object. Rule update is currently performed using a simple string-replacement operation. Thus, for a given clause the operation **replace(string1,string2)** will replace all occurrences of *string1* in the clause body with *string2*. Examples include,

(1) Update the *facts* in relation **student** replacing the *age* of all '*kevin's* with the *null value*,

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

```
UPDATE   @student: facts and
          name = 'kevin'
SET      age = NULL
```

(2) Update the *rules* in **research_student** that use the relation **student**, replacing the '*student*' string with '*staff*',

```
UPDATE   @research_student:
          rules and uses(student)
SET      replace ('student', 'staff')
```

(*b*) Semantic updates are identified by the use of the domain variable as target. The SQUIRREL syntax permits the statement of queries that update semantic objects. The problems of implementing such updates are considered in the next section. Examples include,

(1) Update all semantic objects in relation **student** replacing the *age* field with *NULL*.

```
UPDATE   student
SET      age = NULL
```

(2) Update the semantic objects in relation **student** which have the *name* field equal to '*kevin*' setting the *age* field to 23.

```
UPDATE   student
SET      age = 23
WHERE    name = 'kevin'
```

(3) Update the semantic objects generated by the *facts* in the syntactic object set for the relation **student** setting the *dept* field to '*remedial*'.

```
UPDATE   student: facts
SET      dept = 'remedial'
```

(4) Update the semantic objects generated by *rules* for the **student** relation that have the *name* field equal to *kevin*, setting the *age* value to 23.

```
UPDATE   student: rules
SET      age = 23
WHERE    name = 'kevin'
```

5.4 Update and deletion of semantic objects

The use of views in relation databases leads to problems in updating and deleting tuples in relations represented as views. In a deductive database this problem is exaggerated by the generality permitted in the rule bodies.[8] This problem is partly ascribable to the fact that in logic the world is static.[9] That is, once a rule is defined there can be no exceptions to that rule at a later instant.

Consider the following syntactically valid request to delete a semantic object generated by a relation which contains rules:

```
DELETE FROM grandparent
WHERE elder = 'kevin'
```

There are two possible ways in which tuples containing '*kevin*' can be generated: either the tuple appears in the set of facts in the syntactic set, or it appears in the set of semantic objects generated by a rule in the syntactic set. In the first instance an implementation can delete the corresponding facts from the syntactic set. In the second case (as has already been identified in respect of the indexing problem), the rule generating the tuple cannot be removed as this action will have the side-effect of removing all other semantic tuples generated by the rule. The same arguments apply to update operations.

The question then arises as to the interpretation of a request to delete a semantic object. Should it result in an alteration to the syntactic objects so that the rules when evaluated cannot generate the deleted tuple, or should this form of deletion be forbidden (allowing only deletion or update of syntactic objects)? The first of these suggestions is unsatisfactory, as any implementation that achieves this effect would need to be free from side-effects and would lead to the possibility of inefficient rules being created automatically without the database user being aware of this. The second is also unsatisfactory from the user's perspective, since if one wants to delete a tuple one should not need to be aware of how that tuple is generated internally.

In fact a preferred reading of such a deletion request would be to create facts in the set of syntactic objects which are interpreted as positive exceptions to the relation, that is, as tuples that the rules cannot generate but which should be included in the relation. The request to delete a tuple would then be interpreted as creating an exception to the relation, that is, a tuple which can be generated by the syntactic set but which should not be included in the relation.

This suggests that the problem of a general update to deductive databases parallels the negative information problem. Much research has been conducted on the representation of negative information in databases. Gallaire[9] summarizes this research, but the results reported are generally difficult or expensive to implement.

6. INCOMPLETE INFORMATION IN SQUIRREL

In many database applications a value of an attribute might be unknown. A special value called **null** is introduced to represent the missing value.[10] In some situations one may have some partial information about an attribute even though its exact value is unknown. If the null value is used for this case the information one does have will be discarded. To handle this problem the concept of **incomplete information** is introduced. This kind of value falls between exact values and the null value. The handling of incomplete information in relational databases has been studied by many researchers.[11,12,14,17]

The manipulation of incomplete information in a logic database is much more difficult than it is in a relational database, because the complexity of deductive operations is substantially increased.[18] A model has been put forward by Williams and Kong[15] to account for the behaviour of incomplete information in a logic database.

In this model, a student called *John Smith*, whose age is not known exactly but is known to be one of 18, 19 or 20, can be expressed as:

```
student('John Smith', {18, 19, 20}).
```

which is equivalent to the following expression

```
student('John Smith', 18) OR
student('John Smith', 19) OR
student('John Smith', 20).
```

The implementation problems associated with incomplete information in a logic database have been discussed in Refs 13 and 16.

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

6.1 Inserting incomplete constants

The values that can be stored in a deductive database are expanded to include incomplete values, and the manipulation strategies applicable to the incomplete values are similarly expanded. The evaluation strategy employed for incomplete information can be expensive in both memory usage and execution time. For this reason the algorithm is capable of selectively applying the evaluation strategy. If, for instance, no incomplete information is employed, the normal deductive database evaluation strategy is applied. To permit maximum use of this conditional strategy the definition of a column in a relation will default to complete information. A database designer will be required to specify that a column in a relation may include incomplete information when defining the relation. The following example shows this.

Define a table for relation **staff** which is declared to contain no rules but may have incomplete information in attributes *age* and *dept*.

```
CREATE TABLE staff(
  name char(2) NOT NULL UNIQUE,
  age integer NOT NULL WITH INCOMPLETE,
  sex char(1) NOT NULL,
  dept char(20) WITH INCOMPLETE,
  grade char(20) NOT NULL
  ) WITHOUT RULES
```

The insertion of incomplete values into the above table is demonstrated in the following example:

```
INSERT INTO staff(name,age,sex,dept,grade)
VALUES ('Bill Stone',{31,32,33},
'm',{'Physics', 'Chemistry'}, 'Lecturer')
```

A rule may contain incomplete information, for example: a statement that *a child has blood group A or O if one of his parents has blood group A and the other has blood group O* can be expressed as:

```
INSERT   blood_group(name,type)
USING    father(name,child),
 mother(name,child),
 blood_group(name,type) bg
DEFINE   blood_group(NameC,{'A', 'O'}) IF
                 father(NameF,NameC) AND
                 mother(NameM,NameC) AND
                 bg(NameF, 'A') AND
                 bg(NameM, 'O')
DEFINE   blood_group(NameC,{'A', 'O'}) IF
                 father(NameF,NameC) AND
                 mother(NameM,NameC) AND
                 bg(NameF, 'O') AND
                 bg(NameM, 'A')
```

6.2 Manipulating incomplete values

The evaluation strategies defined in Ref. 16 allow two types of interpretation to be employed in query evaluation. These are *definite* and *possible* evaluation. To these modes is added a default evaluation strategy which treats all incomplete values as if they were the null value. The user is able to control the type of evaluation strategy employed in a similar way to the ALL and DISTINCT selections of SQL which either take all values in a result or restrict the result to unique values. The query and subquery specifications are given an optional DEF or POS clause to specify the form of evaluation strategy to be employed. Examples include:

(1) Select tuples in the **staff** relation which definitely have the value 21 in the *age* column,

```
SELECT DEF*
FROM staff
WHERE age = 21
```

(2) Select tuples which possibly have the value 21 in the *age* column,

```
SELECT POS*
FROM staff
WHERE age = 21
```

(3) Select the tuples with 21 in the *age* column, treating incomplete information in the *age* column as if it were a *NULL* value,

```
SELECT*
FROM staff
WHERE age = 21
```

A new predicate is added for use in the where clause of the table expression: the incomplete data predicate. This is true if the column referenced contains a value that is in the incomplete data item or is a subset of that data item.

(4) Select tuples where the *age* value is wholly contained in the incomplete constant {21,22,23,24},

```
SELECT DEF*
FROM staff
WHERE age IS {21,22,23,24}
```

Since for update and deletion queries the contents of the database are evaluated, it is necessary for the user to specify which evaluation strategy is employed in these statements. This is achieved by including the option DEF or POS (or default) in the update and delete statements. The update statement also permits an incomplete constant to be used in the set clause. Examples show how this is used:

(1) Delete all possible staff members whose *age* may be above 65:

```
DELETE FROM POS staff
WHERE age > 65
```

(2) Update the staff members who are definitely in the Physics department making them all technical *grade*,

```
UPDATE DEF staff
SET grade = 'technical'
WHERE dept = 'Physics'
```

(3) Update all staff members making those with *ages* possibly below 21 have the *grade* {'*junior*','*apprentice*'},

```
UPDATE POS staff
SET grade = {'junior', 'apprentice'}
WHERE age < 21
```

7. SUMMARY

The SQUIRREL language presented in this paper is a variant of SQL, designed for access to a deductive database. It permits the definition and manipulation of relations containing logic statements. The distinction

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

between the two interpretations of the database populations (syntactic and semantic) is made, and language features to distinguish between the database populations described. Examples of the definition, insertion and manipulation of syntactic objects are given. A brief description is given of the implementation strategy that permits syntactic and semantic manipulations to be combined in a query. The logic database implementation permits the expression and manipulation of incomplete information. Examples are given which show how such information is incorporated in the SQUIRREL language.

Acknowledgement

The authors wish to acknowledge the support received from the Science and Engineering Research Council, who are funding the project under the Logic Programming Initiative of the Alvey Programme, and from Mr J. R. Lucking of STC, who is industrial uncle to the project.

REFERENCES

1. E. F. Codd, A relational model of data for large shared data banks. *Communications of the ACM* **13**, 377–387 (1970)
2. H. Gallaire, J. Minker and J. M. Nicholas, An overview and introduction to logic and databases. In *Logic and Databases*, edited H. Gallaire and J. Minker, pp. 3–30. Plenum, New York (1978).
3. J. W. Lloyd and R. W. Topor, A basis for deductive database systems. *Journal of Logic Programming*, no. 2, pp. 93–109 (1985).
4. J. W. Lloyd and R. W. Topor, A basis for deductive database systems. II. *Journal of Logic Programming*, no. 1, pp. 55–67 (1986).
5. ISO, Final Draft ISO 9075-1987(F), *Database Language SQL*. International Standards Organization (1986).
6. R. Reiter, Deductive Q-A on relational databases. In *Logic and Databases*, edited H. Gallaire and J. Minker, pp. – . Plenum, New York (1978)
7. R. Reiter, On closed world databases. In *Logic and Databases*, edited H. Gallaire and J. Minker. Plenum, New York (1978).
8. S. Manchanda and D. S. Warren, A logic based language for database update. In *Foundations of Deductive Databases and Logic Programming*, edited J. Minker, pp. 363–394. Morgan Kaufmann, New York (1988).
9. H. Gallaire, Bridging the gap between AI and databases: logic approach. Proceedings of the IFIP TC2 *Working Conference on Knowledge and Data* (*DS-2*), *Aldeia das Acoteias*, *Portugal* (1986).
10. E. F. Codd, Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems* 4 (4), 397–434 (1979).
11. J. Grant, Partial values in a tabular database. *Information Processing Letters* 9 (2), 97–99 (1979).
12. M. Gray, Views and imprecise information in databases. *Ph.D. Thesis*, *Cambridge* (1982).
13. Q. Kong and M. H. Williams, Evaluating different strategies for handling incomplete information in a logic database. *Workshop on Logic Programming – Expanding the Horizons*. Imperial College (1989).
14. W. Lipski, Informational systems with incomplete information. In *Proc. 3rd Int. Symp. on Automata, Languages and Programming*, edited S. Michaelson and R. Milner, pp. 120–130. Edinburgh University Press, Edinburgh (1976).
15. M. H. Williams and Q. Kong, Incomplete information in a deductive database. *Data and Knowledge Engineering* **3**, 197–220 (1988).
16. M. H. Williams, Q. Kong and G. Chen, Handling incomplete information in a logic database. *Proceedings of UK IT 88 Conference*, pp. 224–227 (1988).
17. M. H. Williams and K. A. Nicholson, Implementation of incomplete information in databases. *The Computer Journal* **31** (2), 133–140 (1988).
18. A. Yahya and L. J. Henschen, Deduction in non-Horn databases. *Journal of Automated Reasoning* **1**, 141–160 (1985).

Downloaded from https://academic.oup.com/comjnl/article/33/6/535/351018 by guest on 10 April 2024

Announcements

***Applied Intelligence**: The international journal of artificial intelligence, neural networks, and complex problem-solving technologies*

Call for Papers

Editor-in-Chief: **Moonis Ali**, The University of Tennessee Space Institute, Tullahoma, TN 37388, U.S.A.

This journal will address issues involving solutions of real-life manufacturing, defence, management and industrial problems which are too complex to be solved through conventional approaches and which require the simulation of intelligent thought processes, heuristics, applications of knowledge, and distributed and parallel processing. The integration of these multiple approaches in solving complex problems will be of particular importance.

The objective of *Applied Intelligence* is to provide a medium for exchanging scientific research and technological achievements accomplished by the international community – academia, industry and government – in the areas associated with intelligent systems.

The emphasis of the reported work will be on new and original research and technological developments covering methodologies as well as applications. Special topic issues will be published from time to time.

Applied Intelligence is a new international journal, published four times a year, beginning in January 1991.

Instructions for Authors

Potential authors should write to the following address for 'instructions for authors':

Karen S. Cullen, Kluwer Academic Publishers, 101 Philip Drive, Norwell, MA 02061, U.S.A.