304

SECURITY AND INTEGRITY IN LOGIC DATA BASES USING QUERY-BY-EXAMPLE

M H WILLIAMS, J C NEVES and S O ANDERSON

Department of Computer Science

Heriot-Watt University

Edinburgh

Scotland

KEYWORDS: Security, Integrity, Query-by-Example,

Prolog, Logic data base.

Abstract

Security and integrity are two important and inter-related aspects of data base systems, and data base management languages must make provision for the specification and enforcement of such constraints. In the case of the data base language Query-by-Example a style for handling certain types of security and integrity constraints has been developed by Zloof.

An alternative approach to integrity in QBE is presented here which is based on the idea of consistency of the data in the data base. This approach allows for a more general type of constraint which includes the handling of functional, multivalued and embedded-multivalued dependencies, as well as the more conventional and simpler type of integrity constraints in a uniform manner.

Both security and integrity constraints have been implemented in Prolog as part of a logic data base.

1. INTRODUCTION

One of the important functions of any data base management system is to preserve the integrity of any data stored within the data base by ensuring that it is consistent with the prescribed properties of such data (integrity constraints). Integrity constraints can be classified into three types (Ullman [1], Nicolas and Yazdanian [2]) :

(a) Value-based constraints. These are conditions which the values of the domain elements must satisfy. They are usually restrictions on the range of values which a field can assume or are concerned with non-structural relationships amongst various fields. For example in the set of relations given in Appendix 1 one might wish to impose restrictions such as:

(i) The weight of a part is always less than 100 units (simple restriction on range).

(ii) An entry may only appear in the supplier_parts table if an entry for the supplier concerned exists in the supplier table (existence check).

(iii) Any supplier from Vienna or Athens must have a status which is at least 20 (non-structural relationship), etc.

(b) Structural or "Value-oblivious" constraints. These are restrictions concerned not with the value in any particular field of a tuple but with whether certain fields of one tuple match those of another. Three specific types of structural constraints are addressed in this paper:

(i) Functional Dependencies. If X and Y are two sets of attributes from some relation scheme, then X functionally determines Y (or Y functionally depends on X), written "X -> Y", if any pair of tuples which agree in the components for all attributes in set X must likewise agree in all components corresponding to attributes in set Y.

Examples of functional dependencies in the set of relations in Appendix 1 include:

> sno -> sname (corresponding to each supplier number is a unique name),
>
> sno, pno -> qty (corresponding to each supplier/part number combination is associated an unique quantity),

and so on. It has been shown [3] that any set of functional dependencies can be transformed to an equivalent set in which all functional dependencies have the form "X -> Y" where Y is a singleton set.

(ii) Multivalued Dependencies. If X and Y are two sets of attributes from some relation scheme then X multidetermines Y (or there is a multivalued dependency of Y on X), written "X ->-> Y", if corresponding to a given set of values for the attributes of X there is a set of zero or more associated values for the attributes of Y, and this set of Y-values is independent of the values of any attributes not contained in X U Y.

An example of a multivalued dependency taken from the relation scheme in Appendix 2 (taken from Ullman [1]) is:

course ->-> period, room, teacher

that is, associated with each "course" is a set of "period-room-teacher" triples which does not depend on any other attributes. For example, given the pair of tuples:

cs2a	3	601	jones j	adams a	42
cs2a	5	302	smith t	zebedee e	67

one would expect to be able to exchange (3, 601, jones j) with (5, 302, smith t) and obtain two valid tuples, viz:

cs2a	5	302	smith t	adams a	42
cs2a	3	601	jones j	zebedee e	67

However, it is not possible to exchange one or two fields of the triple without exchanging all of them, eg:

cs2a	5	601	smith t	adams a	42

is not in the data base since "course ->-> room" does not hold.

(iii) Embedded Multivalued Dependencies. These are multivalued dependencies which do not apply in the full set of data but which become applicable when the data set is reduced by projection. Formally, given a relation scheme R, an embedded multivalued dependency is one which holds only when any relation r in R is projected onto some subset X [

R. For example, in the relation scheme presented in Appendix 2, the multivalued dependency "course ->-> prerequisite" does not hold since tuples such as:

cs2a zebedee e cs1b 1978

are not present in the data base. However, if the data in progresstable is projected onto the subset {course, student, prerequisite} giving:

cs2a	adams a	cs1a
cs2a	adams a	cs1b
cs2a	zebedee e	cs1a
cs2a	zebedee e	cs1b

then "course ->-> prerequisite" does hold, as does "course ->-> student".

(c) Transition constraints. These are restrictions on the way in which the data base may change; or, more specifically, the relationship between the states of the data base before and after any change is made. They include restrictions on the way in which:

(i) Values in a single field may change, e.g. values such as age or salary may only increase, marital status may only change in a particular way, etc.

(ii) Values in a set of fields (possibly in different relations) may change, e.g. the amount of special low-interest-rate loan may be increased only if the grade of the employee is above a certain level, etc.

Security, on the other hand, is concerned with who may access what information in the data base and what operations may be performed. The distinction between security and integrity constraints is not always clear as will be seen in later sections.

Zloof [4] has developed mechanisms for handling security and integrity constraints within the data base language Query-by-Example (QBE). The approach used for handling integrity constraints is a trivial extension of the concept of transition constraints in which constraints may be placed on insert, delete and update operations as well as on print operations. The problem with such an approach is that it is not possible to make any general statements about the data in the data base without a detailed history of the data base.

The object of this paper is to present a slightly different approach which includes all three types of constraints, and which does lend itself to statements about the properties of data in the data base.

The following section gives a brief introduction to Query-by-Example, while section 3 looks briefly at the specification of security constraints (a slight variation from Zloof's approach). The remainder of the paper is devoted to the integrity constraints and implementation details.

2. QUERY-BY-EXAMPLE - THE BASIC LANGUAGE

Query-by-Example [5] is a two-dimensional language which is designed for use at a terminal and makes use of a special-purpose screen editor to compose queries. On striking a particular key, the user is presented with the skeleton of a table as follows:

```
              |
--------------|-------------------------------
              |
```

The four areas delimited by this skeleton are:

```
     (1)      |    (2)
--------------|-------------------------------
     (3)      |    (4)
```

(1) Table name field,

(2) Column name field,

(3) Tuple command field, and

(4) Tuple entry field.

Using the screen editor the user may position the cursor in any of these four areas in order to insert a command and/or a variable or constant element. The formulation of queries is achieved by setting up tuples containing variables, constants and conditions. An attribute which is to be displayed is indicated to the system by typing "p.", followed possibly by a variable name and possibly by a condition, in the column corresponding to that attribute. In our implementation lower-case letters have been used in place of upper-case letters for the basic operations.

For example, to print the status of a particular supplier, say "clark", given the data base of Appendix 1, the user may enter the table name "suppliers" in the table name field, viz (the parts which the user might enter are underlined "___"):

```
suppliers    |
_________    |
-------------|--------------------------------
             |
```

Since the relation already exists in the data base the column headings (attributes of suppliers) can be generated by the system, i.e.:

```
suppliers    | sno   sname   status   city
_________    |
-------------|----------------------------
             |
```

One can now enter "clark" in the sname field and "p.X" in the status field as follows:

```
suppliers    | sno   sname   status   city
_________    |
-------------|----------------------------
             |       clark   p.X
                     _____   ___
```

Any character sequence beginning with a lower case letter, such as "clark", is taken to be a constant representing a specific value, while one beginning with an upper case letter or an underline symbol "_" is taken to be a variable. Thus this is interpreted as a request to print the status of any supplier whose name is "clark".

Similarly to print the details of any supplier whose status exceeds 10, one may enter:

suppliers	sno	sname	status	city
	p.X	p.Y	p.A::A>10	p.C

or one may write the command "p." in the tuple command field as follows:

suppliers	sno	sname	status	city
p.	X	Y	A::A>10	C

where the infix operator "::" is used as a syntactic aid and is to be read as "such that".

A query may require more than one relation in which case appearances of the same variable name in different parts of a query represent the same value. For example, to display the names of all suppliers who supply parts which are red, one may enter:

parts	pno	pname	colour	weight
	X		red	

supplier_parts	sno	pno	qty
	Y	X	

suppliers	sno	sname	status	city
	Y	p.Z		

Complex conditions are handled by use of a separate condition box. For example, suppose that one wishes to display the names of all suppliers for whom the quantity of part number 2 lies between 100 and 300. One may enter:

suppliers	sno	sname	status	city
	X	p.Y		

supplier_parts	sno	pno	qty
	X	2	Z

CONDITIONS
Z>99 and Z<301

Besides the query operator "p." there are three other operators: "i." (Insert), "d." (Delete) and "u." (Update). As an illustration of the use of "i.", consider the addition of a new part tuple to the relation parts:

parts	pno	pname	colour	weight
i.	7	washer	red	10

3. SECURITY IN QUERY-BY-EXAMPLE

Security constraints take the form of an authorization for a user to perform certain operations on a relation. For example, if one wishes to permit a user John to perform print, update and insert operations on the relation suppliers, this may be specified as follows:

suppliers	sno	sname	status	city
i.autr(p.,u.,i.).john	A	B	C	D

where once again lower case letters have been used and the final "i." omitted [4].

The presence of a variable in each field of the relation indicates that John has access to that field. If the variable C had been omitted and the status field left blank, this would indicate that John does not have access to the status field. Just as in other QBE statements, one may add conditions to these variables or link them to fields in other relations.

A more complex example which illustrates this imposes the constraint that John may only read details from the supplier_parts relation if the status of the supplier is less than 30 or the supplier comes from Paris. This is specified as follows:

supplier_parts	sno	pno	qty
i.autr(p.).john	A	B	C

suppliers	sno	sname	status	city
	A		E	F

CONDITIONS
E<30 or F=paris

In each case the entry in the tuple-command-field has the form:

i.autr(<access rights lists>).<user>

The <access rights list> is a list of one or more of the four rights "p.", "i.", "u." or "d." while <user> is the name of the user to whom access is to be granted. In generalizing these two items, following the philosophy of QBE, variables may be used. Similarly if the keyword "all." is used in the table-name-field it will refer to all relations. Thus the constraint:

```
all.            |
____            |
----------------|-------------------
i.autr(X).Y     |
___________
```

will allow any user to perform any operation on any relation.

4. REALIZATION IN PROLOG

In our initial implementation of QBE in Prolog [6], each QBE request (insertion, deletion, update, print, constraints) was translated directly into Prolog and applied to the data base. However, when we changed our approach to integrity constraints and adopted the approach which will be described in the next section, a different implementation strategy was called for.

In the current system (which runs both on a PDP 11/34 and a DEC 10 machine), each QBE request is translated into a clause in a meta-language which is then interpreted using the remainder of the data base.

The following notation is used to express object-level knowledge in the meta-language:

(1) A rule clause is represented as:

```
p <- [q1, q2, ..., qn, {s}].
```

which stands for p :- q1,q2, ...,qn. while the string s in braces {} is used to store information for recreating the original QBE request.

(2) A goal clause is represented by:

<- [q1,q2,...,qn].

which stands for ?- q1,q2,...,qn.

(3) A fact or assertion is represented as:

p.

which stands for p.

The usual interpretations are to be understood for rules, goals and assertions [7]. The use of the meta-language at the object-level has the great advantage of allowing one to use clauses and predicates as terms.

5. EXTENSION TO HANDLE INTEGRITY CONSTRAINTS

The general philosophy behind the approach described here is that any constraint which is currently operative must apply to all data in the data base. Thus whenever a new constraint is defined, it is immediately checked against the data in the data base. If any of the data does not satisfy the constraint, the exceptions are reported and the user is given the opportunity of either updating the data or revising the constraint. If all the data does satisfy the new constraint, it is stored and used to check all insertions and update operations conducted in the future. Three new operators are introduced for this purpose:

ic. - insert a new constraint

dc. - delete an existing constraint

pc. - print constraints

The form of a constraint definition is similar to that of a query. As a simple example, consider the insertion of the constraint that the value in the quantity field of each supplier_parts tuple should be greater than zero. To do this one may enter:

```
supplier_parts   | sno    pno    qty
________________ |
-----------------|------------------
ic.              |            X::X>0
___                           ______
```

or one may use the condition box as follows:

```
supplier_parts   | sno    pno    qty
________________ |
-----------------|-----------------
ic.              |                X
___                               _

|-----------------|
|   CONDITIONS    |
|-----------------|
|      X > 0      |
       _____
```

which is translated by the system to yield:

```
supplier_parts(_, _, X) <-
      [ X>0,
        {X>0}
      ].
```

To ensure that a tuple may only exist in the supplier_parts relation if a tuple for the supplier concerned exists in the suppliers relation, one may have:

supplier_parts	sno	pno	qty
ic.	X		

suppliers	sno	sname	status	city
	X			

which is translated by the system to yield:

```
supplier_parts(X, _, _) <-
        [ suppliers(X, _, _, _),
          {}
        ].
```

A more complicated value-based constraint is the restriction that any supplier from Vienna or Athens must have a status which is at least 20. To specify this, one has:

suppliers	sno	sname	status	city
ic.			X	Y

CONDITIONS
(Y = vienna or Y = athens) implies (X >= 20)

which is translated by the system to yield:

```
suppliers(_, _, X, Y) <-
   [ not (Y=vienna or X>=20) and
     not (Y=athens or X>=20),
     {(Y=vienna or Y=athens) implies (X>=20)}
   ].
```

Functional dependencies are specified in the condition box using the format:

<var> -> <var>

or (<varlist>) -> <var>

For example, in the parts relation, suppose that "pno -> weight". This can be specified as a constraint as follows:

```
parts    | pno   pname   colour   weight
---------|-------------------------------
ic.      | X                      Y

|-----------------|
|   CONDITIONS    |
|-----------------|
|     X -> Y      |
```

which is translated as follows:

```
parts(X, _, _, Y) <-
     [ parts(X, _, _, U),
       Y=U,
       {1->4}
     ].
```

This can be read as:

for all X, A, B, Y:
if there exists R, S, U such that
if parts(X, R, S, U) and Y=U are true
then parts(X, A, B, Y) is true.

When this command is given, the data base will be checked immediately to ensure that the data already present satisfies this condition. Provided it does, the constraint will be added to the data base. Thereafter whenever the user inserts or updates a tuple in the parts relation it attempts to deduce "weight" from "pno" and fill it in automatically for the user.

Multivalued dependencies are specified in a similar way using the format:

<X> ->-> <Y>

where <X> and <Y> each stand for either a single variable or a variable list enclosed in parentheses. Thus in the example from Appendix 2 one might express the constraint:

timetable	course	period	room	teacher	student	mark
ic.	W	X	Y	Z		

CONDITIONS
W ->-> (X, Y, Z)

which is formalized as follows:

```
timetable(W, X, Y, Z, R, S) <-
      [ timetable(W, A, B, C, M, N),
        timetable(W, A, B, C, R, S),
        timetable(W, X, Y, Z, M, N),
        {1->->(2,3,4)}
      ].
```

Once again when this command is given the data base is checked for any violations. If violations arise they are reported, if not the constraint is added to the data base. Thereafter whenever an insertion or update operation causes this constraint to be invoked, the system generates (and displays) the full set of tuples which need to be added to the data base in order to maintain consistency. If the user is content with the set of tuples generated, the system adds the full set to the data base, otherwise the insertion/update operation is abandoned.

Embedded multivalued dependencies are specified using the format:

<X> ->-> <Y>/<Z>

where <X>, <Y> and <Z> each stand for either a single variable or a variable list in parentheses. This is interpreted as X multidetermines Y if the set of attributes Z is removed. For example, to express the fact that "course ->-> prerequisite" if the relation "progresstable" in Appendix 2 is projected onto the subset {course, student, prerequisite}, one may enter:

progresstable	course	student	prerequisite	year
ic.	X		Y	Z

CONDITIONS
X ->-> Y/Z

which is translated by the system to yield:

```
progresstable(X, A, Y, Z) <-
      [ progresstable(X, B, C, E),
        progresstable(X, A, C, R),
        progresstable(X, B, Y, S),
        {1->->3/4}
      ].
```

When this command is given, the data base is checked for consistency. If violations arise the user is prompted to correct them or abort the constraint. Once the constraint is added to the data base, any further insertions or update operations are checked against the constraint and where required the system will generate the full set of tuples needed to fulfil any particular operation, prompting the user for the additional information (year) required to complete each tuple.

Transition constraints, which are concerned with the way in which values in the data base may change, are expressed using a pair of entries for the relation in question. The field in this relation which is to be controlled, will be represented by two different variables -

the one occurring in the line with the ic command in the tuple-command-field represents the new value of the variable, the other the old value.

For example, suppose that one wishes to place a constraint on the status of a supplier whereby it can only increase, one might enter:

```
suppliers   | sno    sname   status   city
_________   |
------------|-----------------------------
ic.         | N              X
__          | _              _
            | N              Y
              _              _
```

```
|-----------------|
|   CONDITIONS    |
|-----------------|
|      X>=Y       |
       ____
```

which is translated by the system to yield:

```
suppliers(N, _, X, _) <-
      [ suppliers(N, _, Y, _),
        X>=Y,
        {X>=Y}
      ].
```

Similarly one might impose a constraint on the age or salary of an employee whereby the values of these fields for a particular employee can only increase. In the case of marital status the only permissible transitions may be:

```
single    ---> married
married   ---> divorced or widowed
divorced  ---> married
widowed   ---> married
```

which may be specified as follows:

employee	empno	ename	salary	status	grade
ic.	N			X	
	N			single	

CONDITIONS
X=married or X=single

employee	empno	ename	salary	status	grade
ic.	N			X	
	N			married	

CONDITIONS
X=married or X=divorced or X=widowed

and so on. This is translated by the system to yield:

```
employee(N, _, _, X, _) <-
      [ employee(N, _, _, single, _),
        X=married or X=single,
        {X=married or X=single}
      ].

employee(N, _, _, X, _) <-
      [ employee(N, _, _, married, _),
        X=married or X=divorced or X=widowed,
        {X=married or X=divorced or X=widowed}
      ].

...
...
...
```

Alternatively the four constraints may be combined into a single one using two variables.

As an example of a more complex form of constraint, consider the restriction that the value of a loan may only increase (or decrease) if the grade of the employee is greater than 5. This might be specified as follows:

```
loantable    | empno   loan
-------------|-------------
ic.          | X       NL
             |
             | X       OL
```

```
employee    | empno   ename   salary   status   grade
------------|-----------------------------------------
            | X                                 Y
```

```
|-----------------------------|
|         CONDITIONS          |
|-----------------------------|
|   (Y<=5) implies (NL=OL)    |
```

This is translated by the system to yield:

```
loantable(X, NL) <-
     [ loantable(X, OL),
       employee(X, _, _, _, Y),
       not Y<=5 or NL=OL,
       {(Y<=5) implies (NL=OL)}
     ].
```

The complete syntax of these constraints is given in Appendix 3.

6. OVERLAP OF INTEGRITY AND SECURITY CONSTRAINTS

The transition constraints discussed in the previous section deal only with the way in which data in the data base may change (i.e. be updated). It does not cater for transitions involving insertion or deletion.

Thus suppose one wishes to impose the constraint that a loan may only be granted to an employee with grade between 5 and 8, but once an employee has been granted a loan, if his grade changes to a value outside the range 5-8, he will not lose his existing loan. This type of constraint is not a simple property of the data (i.e. one cannot conclude that any employee who has a loan, must have a grade in the range 5 to 8). However, it can be handled using a security constraint, eg.

```
loantable              | empno   loan
_________              |
---------  ------------|-------------
i.autr(i.).X           | A       B
_____________            _       _
```

```
employee     | empno   ename   salary   status   grade
________     |
-------------|------------------------------------------
             | A                                 C
               _                                 _
```

```
|------------------------|
|       CONDITIONS       |
|------------------------|
|  (C>=5) and (C<=8)
   __________________
```

Likewise the example considered by Nicolas and Yazdanian [2] in which a constraint needs to be placed on the system to prevent employees whose income is less than some value (say 5000) from being deleted, can be treated as follows:

employee	empno	ename	salary	status	grade
i.autr(d.).X	A	B	C	D	E

CONDITIONS
C>=5000

7. CONCLUSIONS

The specification and enforcement of integrity constraints in a data base system is essential in order to guarantee the consistency of data within the data base. The role of security constraints is to control the types of operations which individual users may perform on the data base. The two types of constraints overlap to some extent.

This paper presents an integrated approach for specifying generalized integrity and security constraints within the data base management language Query-by-Example.

The important aspects of this approach are:

(a) It caters for all three types of integrity constraints in a generalized and consistent manner.

(b) It treats integrity constraints as properties of the data applying to all data in the data base, rather than as properties of particular operations (as proposed by Zloof [4]).

(c) It ensures that the user is aware of the implications of any operation producing changes in the data base which affect fields involved in multivalued or embedded-multivalued dependencies.

ACKNOWLEDGEMENTS

The work of one of the authors, J C NEVES, was supported by the Calouste Gulbenkian Foundation under grant 14/82 and by an ORS award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

J C NEVES is on leave from Minho University, Largo do Paco, 4700, BRAGA, PORTUGAL.

8. REFERENCES

[1] J. D. Ullman, Principles of Database Systems, London: Pitman, 1980.

[2] J. M. Nicolas and K. Yazdanian, Integrity Checking in Deductive Data Bases, in: Logic and Data Bases, Plenum Press, 1978, 325-344.

[3] W. W. Armstrong, Dependency Structures of Database Relationships. Proc. IFIP 74, 1974, 580-583.

[4] M. M. Zloof, Security and Integrity within the Query-by-Example Database Management Language, IBM RC6982, Yorktown Heights, N. Y., 1978.

[5] M. M. Zloof, Query-by-Example: A Data Base Language, IBM Systems J., Vol. 16, No. 4, 1977, 324-343.

[6] J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog Implementation of Query-by-Example, in: Proceedings of the 7th International Computing Symposium, March 22-24, 1983, Nurnberg, Germany.

[7] W. F. Clocksin, and C. S. Mellish, Programming in Prolog, Springer-Verlag, 1981.

[8] K. Bowen, and R. A. Kowalski, Amalgamating Object Language and Metalanguage in Logic Programming. To appear in Logic Programming (K. L. Clark and S. -A. Tarnlund Eds.)

Academic Press,1982.

[9] M. H. Williams, A Flexible Notation for Syntatic Definitions, ACM Trans. on Prog. Lang. and Syst., Vol. 4, No. 1, 1982, 113-119.

Appendix 1: A simple business data base

Consider a simple business data base which contains:

(i) A relation "parts" with attributes (columns): pno, pname, colour and weight.

(ii) A relation "suppliers" with attributes: sno, sname, status and city.

(iii) A relation "supplier_parts" with attributes: sno, pno and qty.

(iv) A relation "employee" with attributes: empno, ename, salary, status and grade.

(v) A relation "loantable" with attributes: empno and loan.

Suppose that the current content of each relation is:

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	30	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

employee	empno	ename	salary	status	grade
	12	morley	6500	married	10
	7	warren	7135	single	7
	15	exner	4475	single	4
	17	berry	5345	married	12
	5	john	6725	widowed	9

Table 1.4 - The employee relation

loantable	empno	loan
	7	570
	17	1500

Table 1.5 - The loantable relatio

Appendix 2: A simple departmental data base

Consider a simple departmental data base which contains:

(i) A relation "timetable" with attributes: course, period, room, teacher, student, grade.

(ii) A relation "progresstable" with attributes: course, student, prerequisite, year.

Suppose that the current content of the data base is:

timetable	course	period	room	teacher	student	grade
	cs2a	3	601	jones j	adams a	42
	cs2a	5	302	smith t	zebedee e	67

Table 2.1 - The timetable relation

progresstable	course	student	prerequisite	year
	cs2a	adams a	cs1a	1978
	cs2a	adams a	cs1b	1979
	cs2a	zebedee e	cs1a	1978
	cs2a	zebedee e	cs1b	1979

Table 2.2 - The progresstable relation

Appendix 3: Concrete syntax of the data base query language Extended-Query-by-Example

The basic Extended-Query-by-Example (EQBE) format is as follows:

```
Table-name-field          |  Column-name-field
--------------------------|--------------------
Tuple-command-field       |  Tuple-entry-field
                          |

|-----------------------------|
|         CONDITIONS          |
|-----------------------------|
|   Condition-entry-field     |
```

where the syntax of each of these components is defined as:

```
table-name-field ::= ("i." | "u.") string-constant |
                     ["p." | "d."] [string-constant] |
                     "all."

column-name-field ::= ["p."] [string-constant]

tuple-entry-field ::= ["p."] [example-element
                      ["::" relation] | p-relation]
                      | string-constant | integer

authorization ::= "autr" ["(" access-rights-list ")"] "."
                  user-list

access-rights-list ::= access-right ("," access-right)* |
                       example-element

access-right ::= "p." | "i." | "d." | "u."

user-list ::= list | example-element | string-constant

list ::= "(" string-constant ("," string-constant)* ")"
```

```
tuple-command-field ::= ["ic." | "dc." | "pc." |
                         ("i." | "d." | "u." | "p.")
                         [authorization]]

condition-entry-field ::= functional-dependency
                          | multivalued-dependency
                          | embedded-multivalued-dependency
                          | boolean-expression

functional-dependency ::= set "->" example-element

multivalued-dependency ::= set "->->" set

embedded-multivalued-dependency ::= set "->->" set "/" set

set ::= "(" example-element ("," example-element)* ")"
        | example-element

boolean-expression ::= boolean-secondary
                       ("implies" boolean-secondary)*

boolean-secondary ::= boolean-term ("or" boolean-term)*

boolean-term ::= boolean-factor ("and" boolean-factor)*

boolean-factor ::= ["not"] boolean-primary

boolean-primary ::= boolean-constant | relation
                    | "(" boolean-expression ")"

boolean-constant ::= "true" | "false"

relation ::= numeric-exp relational-op numeric-exp
             | string-exp relational-op string-exp

p-relation ::= relational-op (numeric-exp | string-exp)

numeric-exp ::= [add-op] numeric-term
                (add-op numeric-term)*

numeric-term ::= factor (multiply-op factor)*

factor ::= [function-designator] numeric-variable
```

```
          | numeric-constant | "(" numeric-exp ")"
multiply-op ::= "*" | "/"
add-op ::= "+" | "-"
function-designator ::= "max." | "min." | "ave."
                        | "cnt." | "sum."
string-exp ::= string-primary ("+" string-primary)*
string-primary ::= string-variable | string-constant
integer ::= digit +
string-constant ::= (""""non-quote-character*"""")+ |
                    lower-case-letter letter-or-digit*
string-variable ::= example-element
numeric-variable ::= example-element
example-element ::= capital-letter letter-or-digit*
                    | underscore letter-or-digit*
letter-or-digit ::= lower-case-letter | digit
capital-letter ::= "A" | "B" | "C" | "D" | "E" | "F"
                   | "G" | "H" | "I" | "J" | "K" | "L"
                   | "M" | "N" | "O" | "P" | "Q" | "R"
                   | "S" | "T" | "U" | "V" | "W" | "X"
                   | "Y" | "Z"
lower-case-letter ::= "a" | "b" | "c" | "d" | "e" | "f"
                      | "g" | "h" | "i" | "j" | "k" | "l"
                      | "m" | "n" | "o" | "p" | "q" | "r"
                      | "s" | "t" | "u" | "v" | "w" | "x"
                      | "y" | "z"
digit ::= "0" | "1" | "2" | "3" | "4" | "5"
          | "6" | "7" | "8" | "9"
```

where the notation used is that given by Williams [9].