PUC

Series: Monografias em Ciência da Computação
Nº 5/82

DESIGN-BY-EXAMPLE
(Preliminary Report)

Claudio M. O. Moura
Marco A. Casanova

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC / RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação, Nº 5/82

Editor: Marco A. Casanova May, 1982

DESIGN-BY-EXAMPLE*

(Preliminary Report)

Claudio M. O. Moura

Marco A. Casanova

* This research was supported in part by FINEP and CNPq grant 402090/80

** IBM Latin American Systems Research Institute
C.P. 1830
22671 - Rio de Janeiro - RJ - Brazil

ABSTRACT

A constraint definition language, that provides a uniform notation for data dependencies commonly used, is introduced. Constraints are expressed in this language in much the same way as queries are defined in Query-by-Example. Dictionary facilities to manage constraint written in this language are also described. Finally, the problem of checking if a set of constraints captures the intended semantics of the enterprise is considered.

KEYWORDS:

Constraint definition language, data dependencies, data dictionary, constraint checking, database design.

RESUMO:

Uma linguagem para definição de restrições de integridade, provendo uma notação uniforme para as dependências de dados comumente usadas, é introduzida. Restrições são expressas nesta linguagem de forma semelhante a consultas em Query-by-Example. Um dicionário para restrições escritas nesta linguagem também é descrito. Finalmente, o problema de verificar se um conjunto de restrições capta a semântica pretendida para o empreendimento é considerado.

PALAVRAS CHAVE:

Linguagem de definição de restrições, dependência de dados, dicionário de dados, verificação de restrições, projeto de banco de dados.

1. Introduction

A database description consists of a set of data structures and a set of integrity constraints restricting what data values can be stored in the database. A database state is said to be consistent if it satisfies all integrity constraints. Users' transactions are then forced to preserve consistency, that is, to map the set of consistent states into itself.

The relational model of data adopts flat tables, or relations, as the basic data structure. Several classes of integrity constraints, usually called data dependencies, have been studied in connection with the relational model. A sample of data dependencies includes functional dependencies [Co, Ar], multivalued dependencies [Fa1, ZM, BHF], join dependencies [ABU, MMS], inclusion dependencies [Fa2, CFP] and template dependencies [SU, FMUY]. However, all these classes, and others [YP, GJ], are special cases of the extended embedded implicational dependencies (XEIDs) [Fa3].

The purpose of this paper is to explore XEIDs as a practical tool to model database semantics. Towards this end, a suitable constraint definition language is first introduced and then the problem of checking if a given set of XEIDs expresses the intended semantics of the enterprise is discussed.

Our constraint definition language follows the style of Query-by-Example (QBE) [Zl1, Zl2, Re] and, for this reason, it is called Design-by-Example. XEIDs are expressed in this language in much the same way as queries are defined in QBE. Hence, the language provides a uniform, easy-to-use notation to express XEIDs and, thus, the data dependencies commonly found in the literature.

A database description will then model a given enterprise through a set of tables, defined using QBE, and a set of XEIDs over these tables, specified through our language. The problem we address next is how to verify that the set of XEIDs captures the intended semantics of the enterprise. By this we do not mean the problem of proving that a set of XEIDs is logically equivalent to another formal description of the enterprise, but rather we mean the problem of checking if the behaviour of the database, as determined by the XEIDs, corresponds to the intuitive behaviour of the enterprise.

To conclude this introduction, we briefly describe the contents of each section. Section 2 reviews the basic concepts of the relational model,

defines XEIDs and shows that many other data dependencies are indeed special cases of XEIDs. Section 3 describes our constraint language with the help of a series of examples. Section 4 describes dictionary facilities to manage integrity constraints. Section 5 addresses the problem of verifying that a set of XEIDs captures the intended semantics of the enterprise. Finally, section 6 contains conclusions and suggests directions for future research.

2. Basic Concepts

A relation scheme is a statement of the form R[U], where R is a relation name and $U=(A_1,\ldots,A_n)$ is a finite sequence of attributes. A tuple t over U (from a given set $\mathcal{D}$) is a sequence $(a_1,\ldots,a_n)$ where $a_1,\ldots,a_n \in \mathcal{D}$. A relation over U (from $\mathcal{D}$) is a set of tuples over U from $\mathcal{D}$.

If s is a sequence, then $|s|$ denotes the length of s. If $s=(s_1,\ldots,s_n)$ is a sequence such that $|s|=|U|$, and $X=(A_{i_1},\ldots,A_{i_k})$, where $i_1,\ldots,i_k$ are distinct elements of $\{1,\ldots,n\}$, then s[X] denotes the sequence $(s_{i_1},\ldots,s_{i_k})$. If s and s' are two sequences, then s=s' indicates that s and s' have the same length and the same entries.

Now, if r is a relation over U, then $r[X]=\{t[X]/t\in r\}$.

A database scheme $D=\{R_1[U_1],\ldots, R_m[U_m]\}$ is a set of relation schemes. A database state I of D is a mapping that associates each relation scheme $R_i[U_i]$ with a relation r_i over U_i. Sometimes we will represent the database state simply as $r_1,\ldots,r_m$. We assume that $r_1,\ldots,r_m$ are relations from a given set $\mathcal{D}$, the domain of I.

Given a database scheme $D=\{R_1[U_1],\ldots,R_m[U_m]\}$, and a set of variables $x_1, x_2, \ldots$, a relational formula over D is a statement of the form $R_i(\bar{x})$, where $1\le i\le m$ and $\bar{x}$ is a sequence of variables such that $|\bar{x}|=|U_i|$. An equality is a statement of the form x=y, where x and y are variables. An atomic formula is either an equality or a relational formula.

Formulas (involving connectives and quantifiers) and sentences (formulas with no free variables) are defined as for first-order logic [En].

An extended embedded implicational dependency (XEID) [Fa3] over D is a sentence F of the form $\forall x_1\ldots\forall x_m((P_1\wedge\ldots\wedge P_k) \Rightarrow \exists y_1\ldots\exists y_r(Q_1\wedge\ldots\wedge Q_\ell))$ where $P_1,\ldots,P_k$ are relational formulas over D, the antecedents of F, and $Q_1,\ldots,Q_\ell$ are atomic formulas, the consequents of F, such that each x_i occurs in at least one P_j.

note: we do not require that $P_1,\ldots,P_k$ be typed and not inter-relational, unlike the original definition of XEIDs [Fa3].

Given a database state $I=\{r_1,\ldots,r_m\}$ of D with domain $\mathcal{D}$ and a set of variables $x_1,x_2,\ldots$, a valuation over $\mathcal{D}$ is a function v assigning to each variable x_i an element $v(x_i) \in \mathcal{D}$. We say that a relational formula $R_i(x_1,\ldots,x_m)$ over D is true in I for v iff $(v(x_1),\ldots,v(x_m)) \in r_i$. We also say that an equality x=y is true in I for v iff v(x)=v(y).

Now, given an XEID F over D of the form $\forall x_1\ldots\forall x_m((P_1\wedge\ldots\wedge P_k) \Rightarrow \exists y_1\ldots\exists y_r(Q_1\wedge\ldots\wedge Q_\ell))$ we say that F is true in a database state I iff, for any valuation v over $\mathcal{D}$, there is a valuation v' over $\mathcal{D}$ such that v' agrees with v on $x_1,\ldots,x_m$ and if $P_1,\ldots,P_k$ are true in I for v', then $Q_1,\ldots,Q_\ell$ are true in I for v'. In this case, we also say that I satisfies F.

For example, the XEID $\forall x_1\ldots\forall x_s(R_i(x_1,x_2,x_3)\wedge R_i(x_1,x_4,x_5)\Rightarrow x_2 = x_4)$ is true in a database state $r_1,\ldots,r_m$ iff any two tuples of r_i that agree on the first entry must also agree on the second entry. As a second example, the XEID $\forall x_1\forall x_2\forall x_3(R_i(x_1,x_2,x_3) \Rightarrow \exists x_4\ R_j(x_4,x_1,x_2))$ is true in $r_1,\ldots,r_m$ iff the projection of r_i on the first two columns is a subset of the projection of r_j on the last two columns.

We say that an XEID F is a logical consequence of a set $\mathcal{F}$ of XEIDs iff any database state I that satisfies all XEIDs in $\mathcal{F}$ also satisfies F.

We close this section by defining some of the familiar dependencies in terms of XEIDs. Let $D=\{R_1[U_1],\ldots, R_m[U_m]\}$ be a database scheme.

A functional dependency (FD) over D is an XEID F of the form $\forall x_1\ldots\forall x_n(R_i(\bar{u}) \wedge R_i(\bar{v}) \Rightarrow \bigwedge_{j=1}^{q} u_{k_j} = v_{k_j})$ where X is a sequence of distinct attributes of $R_i[U_i]$, as is $Y=(Y_{k_1},\ldots,Y_{k_q})$, and $\bar{u}$ is a sequence of distinct variables from the set $\{x_1,\ldots,x_n\}$, as in $\bar{v}$, such that $|\bar{u}|=|\bar{v}|=|U_i|$ and $\bar{u}[X]=\bar{v}[X]$. We abbreviate F as $R_i: X\rightarrow Y$.

A multivalued dependency (MVD) over D is an XEID F of the form $\forall x_1\ldots\forall x_n(R_i(\bar{u}) \wedge R_i(\bar{v}) \Rightarrow R_i(\bar{t}))$ where X is a sequence of distinct attributes of $R_i[U_i]$, as is Y, and Z is a sequence of the attributes of $R_i[U_i]$ not in X or Y; $\bar{u}$, $\bar{v}$ and $\bar{t}$ are each a sequence of distinct variables from the set $\{x_1,\ldots,x_n\}$ such that $\bar{u}[X] = \bar{v}[X] = \bar{t}[X]$, $\bar{v}[Y] = \bar{t}[Y]$ and $\bar{u}[Z] = \bar{t}[Z]$. We abbreviate F as $R_i: X\twoheadrightarrow Y|Z$ or, simply, $R_i: X\twoheadrightarrow Y$.

An inclusion dependency (IND) over D is an XEID F of the form

$\forall x_1 \ldots \forall x_p\ (R_i(\bar{u}) \Rightarrow \exists y_1 \ldots \exists y_q\ R_j(\bar{v})))$ where X and W are sequences of distinct attributes of $R_i[U_i]$ and $R_j[U_j]$, respectively, such that $|X|=|W|$, and $\bar{u}$ and $\bar{v}$ are sequences of distinct variables from $\{x_1,\ldots,x_p\}$ and $\{x_1,\ldots,x_p, y_1,\ldots,y_q\}$, respectively, such that $\bar{u}[Y] = \bar{v}[W]$. We abbreviate F as $R_i[X] \subseteq R_j[W]$.

This brief list covers all dependencies we will mention in later sections, but it could be extended to include other familiar dependencies (the reader is referred to [Fa3] for further discussion on the expressive power of XEIDs)

We close this section with an example illustrating the use of XEIDs. Consider the following database schema [Da]
D={SUPPLIER[S#, CITY, STATUS], SP[S#, P#, QTY]}

We define the following FDs over D (together with their abbreviations):

(a) $\forall s \forall c \forall t \forall c' \forall t'$(SUPPLIER(s,c,t) $\wedge$ SUPPLIER(s,c',t') $\Rightarrow$ c=c' $\wedge$ t=t')
- SUPPLIER: S# $\rightarrow$ CITY, STATUS

(b) $\forall s \forall c \forall t \forall s' \forall t'$(SUPPLIER(s,c,t) $\wedge$ SUPPLIER(s',c,t') $\Rightarrow$ t=t')
- SUPPLIER: CITY $\rightarrow$ STATUS

(c) $\forall s \forall p \forall q \forall q'$(SP(s,p,q) $\wedge$ SP(s,p,q') $\Rightarrow$ q=q')
- SP: S#, P# $\rightarrow$ QTY

We also define the following IND over D (together with its abbreviation) that should have been introduced in the original example:

(d) $\forall s \forall p \forall q$(SP(s,p,q) $\Rightarrow$ $\exists c\ \exists t$ SUPPLIER(s,c,t))
- SP[S#] $\subseteq$ SUPPLIER[S#] □

This example will be used in Section 3.2 to introduce our constraint definition language.

3. A Constraint Definition Language

We describe in this section a constraint definition language that extends the QBE DDL [Zℓ1, Zℓ2] to include XEIDs. In Section 3.1 we briefly discuss the major characteristics of QBE that are relevant to this paper. In Section 3.2 we then introduce our constraint definition language, illustrating the basic features with examples.

3.1 - Query-by-Example

Query-by-Example (QBE) is a relational query language of the family of Relational Calculus [Co2]. To formulate a query in QBE, the user first creates table frames with the relation names and attributes he will use. Then, the user fills these table frames with examples of possible answers to his query. For example, referring to the database scheme at the end of Section 2, the following query "list all part numbers supplied by some supplier located in a city with status 10" would be formulated as:

SUPPLIER	S#	CITY	STATUS
	s		10

SP	S#	P#	QTY
	s	P.	

where all underscored letters act as variables (or examples) and where P. indicates the desired answer. When some query involves a complex condition, it can be formulated with the help of a CONDITION box. For example, the query "list all cities with status less than 10 and greater than 5" would be formulated as:

CS	CITY	STATUS
	P.	s

CONDITION
s=(>5 and <10)

or, alternatively, as

CS	CITY	STATUS
	P.	s

CONDITION
s > 5
s < 10

These two examples suffice to give an indication of the syntax of QBE. We discuss here just two characteristics of the language. First, QBE has a unique bidimensional syntax, which gives users great freedom to formulate queries, since table entries can be filled in the order users feel is the most natural. Second, unlike SEQUEL [CB] or QUEL [HSW] or even Relational Calculus [Co2], variables range over table entries, not tuples. Hence, QBE is a domain-oriented language [Pi]. Our constraint definition language is based on these two characteristics of QBE.

3.2 - Design-by-Example

We first introduce our constraint definition language by way of a simple example. Then, we define the basic constructs of the language more precisely and, finally, we return to more examples. We construct the language on top of the basic commands and conventions used in QBE.

Consider again the database scheme of Example 2.1:

D={SUPPLIER[S#, CITY, STATUS], SP[S#, P#, QTY]}

together with the constraints

(a) $\forall s \forall c \forall t \forall c' \forall t' (SUPPLIER(s,c,t) \wedge SUPPLIER(s,c',t') \Rightarrow c=c' \wedge t=t')$

(b) $\forall s \forall c \forall t \forall s' \forall t' (SUPPLIER(s,c,t) \wedge SUPPLIER(s',c,t') \Rightarrow t=t')$

(c) $\forall s \forall p \forall q \forall q' (SP(s,p,q) \wedge SP(s,p,q') \Rightarrow q=q')$

(d) $\forall s \forall p \forall q (SP(s,p,q) \Rightarrow \exists c \exists t\ SUPPLIER(s,c,t))$

Constraints (a) and (c) actually say, respectively, that S# is a key of SUPPLIER and S#, P# is a key of SP. Hence, they can be both defined using the DDL of QBE [ZL2].

However, this is not the case with (b) and (d). Let us consider (d) first. We propose to define (d) in much the same way as a query is specified in QBE. The user would first invoke two table frames and fill these frames with the relevant relation names and attributes (those that will not be used can be omitted). In our running example, these operations would result in

SP	S#
I.A1	

SUPPLIER	S#

The operator field of the first line of the first frame is filled with I.A1 to indicate to the system that a new constraint, whose name is A1, is now being defined. The end of the definition of A1 is indicated to the system when the user presses ENTER.

The definition of the constraint would continue by filling the relevant table entries with variables (examples). Those rows that correspond to the right-hand side of XEID are indicated by placing C, the consequence operator, in the command field. Thus, the final specification of the constraint would look as follows:

SP	S#
I.A1	s

SUPPLIER	S#
C.	s

The above example illustrates how we would specify an XEID whose consequents are relational formulas. Consider now constraint (b) (the FD SUPPLIER: CITY →→ STATUS), which is an XEID whose right-hand side is an equality. To cope with these XEIDs, we introduce the CONSEQUENCE box, so that constraint (b) would be specified as follows:

SUPPLIER	S#	CITY	STATUS
I.A2		c	t
		c	t'

CONSEQUENCE
t = t'

These two examples give the general flavor of Design-by-Example. It should be clear that our constraint language follows quite closely the major characteristics of QBE pointed out in Section 3.1. We also observe that there is a direct mapping of the basic notation of XEIDs and our language. However, due to its bidimensional syntax, our language allows users greater flexibility in formulating constraints. Moreover, quantifiers and connectives are left implicit, which alleviates the syntax.

Let $D=\{R_1[A_{11}\ldots A_{1m_1}],\ldots, R_n[A_{n1}\ldots A_{nm_n}]\}$ be a database. In general, an XEID F over D of the form $\forall x_1\ldots\forall x_p(P_1\wedge\ldots\wedge P_r \Rightarrow \exists y_1\ldots\exists y_q(Q_1\wedge\ldots\wedge Q_s))$ is formulated in our language as follows:

(i) for each P_i, if P_i is of the form $R_k(u_1,\ldots,u_{m_k})$, a table of the form below is created:

R_k	A_{k1}	---	A_{km_k}
	u_1		u_{m_k}

By convention, if u_j is not used elsewhere in the XEID, the corresponding column may be omitted or u_j may be replaced by a blank;

(ii) for each Q_i, if Q_i if of the form $R_k(u_1,\ldots,u_{m_k})$, a table of the form below is created

R_k	A_{k1}	----	A_{km_k}
C.	u_1		u_{m_k}

(The same convention as in (i) applies here). Otherwise, Q_i is of the form u=v and a line of the form below is inserted in the CONSEQUENCE box:

CONSEQUENCE
⋮ u = v ⋮

Finally, the operator field of the first line of the first table is filled with insert constraint operator, I.<name>, indicating the name of the constraint being defined (<name> follows the same conventions as table names in QBE).

This concludes the description of the general case.

We now exhibit other examples of XEIDs defined in our language. Let E={EMP[NAME, SKILLS, PROJ, MGR], NP[NAME, PROJ], PM[PROJ, MGR]} be a database. Then the MVD EMP: NAME ↠ SKILLS | PROJ, MGR would be defined as:

EMP	NAME	SKILLS	PROJ
I.A3	n	-	p
	n	s'	
C.	n	s'	p

If we wanted to enforce that a manager must also be an employee, we would write

EMP	NAME	MGR
I.A4		m
C.	m	

As indicated by the attributes, we may assume that NP and PM are redundant in the sense that if NP(n,p) and PM(p,m) holds then there is s such that EMP(n,s,p,m) also holds. This is expressed as follows:

NP	NAME	PROJ
I.A5	n	p

PM	PROJ	MGR
	p	m

EMP	NAME	PROJ	MGR
C.	n	p	m

note: following our conventions, we left out the SKILLS column of EMP.

This concludes the description of the core of the constraint definition language. The next section discusses a dictionary facility to store and maintain constraints.

4. A Dictionary Facility to Manage Constraints

Since the description of a database may involve a large number of XEIDs, just cataloguing all constraints may be time-consuming. Thus, we introduce a constraint dictionary facility that helps the DBA in this task. We first describe the dictionary structure and then we introduce commands to catalogue and maintain constraints.

4.1 - The Dictionary of Constraints

Constraints will be maintained in much the same way as stored queries are kept in the QBE system. The dictionary of constraints then consists of:

(a) a file to store constraints, whose organization is totally transparent to users; (b) a system table CONSTRAINT, with attributes ANTECEDENT, CONSEQUENT, NAME, COMMENTS and USERID, which resembles the system tables of QBE.

The CONSTRAINT table contains the names of relations that participate in an antecedent of an XEID, the names of the relations that participate in the consequent of an XEID or an indication that the CONSEQUENCE box is used to specify the consequent, the name of the XEID, a column for comments and a column USERID to identify the specific database to which the XEID refers. The CONSTRAINT table is automatically updated whenever a new constraint is defined, except for the COMMENTS field, which is updated manually.

Thus, after the examples in Section 3.2 have been defined, the CONSTRAINT table will contain:

CONSTRAINT	ANTECEDENT	CONSEQUENT	NAME	COMMENTS	USERID
	SP	SUPPLIER	A1	SP.S# is sub of SUPPLIER.S#	D
	SUPPLIER	C.	A2	CITY→STATUS	D
	EMP	EMP	A3	NAME→→SKILLS	E
	EMP	EMP	A4	manager is an employee	E
	NP	EMP	A5	-	E
	PM	EMP	A6	-	E

Information may be retrieved from this table in the same way as it is retrieved from any table. But other operations on the dictionary are also provided, as discussed in the next section.

4.2 - Operations on the Dictionary of Constraints

We discuss in this section how to display, delete and modify constraints stored in the dictionary. Note that no explicit insertion operation is necessary, as discussed in Section 3.2.

As already mentioned, the CONSTRAINT table may be queried as any other table. Thus, returning to the example at the end of Section 4.1, the following query

CONSTRAINT	ANTECEDENT	NAME	USERID
	EMP	P.	P.

produces an output table showing the names of XEIDs where EMP occurs in an antecedent:

CONSTRAINT	NAME	USERID
	A3	E
	A4	E

To display constraint A_i, the user would enter in input mode the print constraint operator, P.CONSTR.A_i, in the COMMAND box. Thus, assuming that the user has USERID E, to display constraints A3 and A4, the user would enter

COMMAND
P.CONSTR.A3

which results in

EMP	NAME	SKILLS	PROJ
	n		P
	n	s'	
C.	n	s'	P

and

COMMAND
P.CONSTR.A4

which produces

SUPPLIER	S#	CITY	STATUS
		c	t
		c	t'

CONSEQUENCE
t = t'

The print constraint operator also offers the option of sequentially scanning the constraint file. After displaying a constraint, the system displays a message "PRESS ENTER TO SEE MORE"; if the user presses ENTER the next constraint on the constraint file is displayed (the order of constraints in the constraint file has no significance). To stop displaying constraints, the user should return to input state via COMMAND box and the QBE standard comand P.*.

To delete constraint A_i, the user simply enters D.CONSTR.A_i in the COMMAND box. The execution of a delete operation automatically updates the CONSTRAINT table.

Updates on constraints stored in the dictionary follow the same philosophy. To modify constraint A_i, the user enters U.CONSTR.A_i in the COMMAND box. Constraint A_i is then displayed on the screen to be edited by the user. The new version of A_i then replaces the old version when the user presses ENTER.

For example, suppose that the user wants to modify constraint A4 to mean that S#, CITY → STATUS (and not CITY→STATUS). He then enters

COMMAND
U.CONSTR.A4

which results in

SUPPLIER	S#	CITY	STATUS
		c	t
		c	t'

CONSEQUENCE
t = t'

The user would then edit the screen to

SUPPLIER	S#	CITY	STATUS
	s	c	t
	s	c	t'

CONSEQUENCE
t = t'

and press ENTER.

This concludes our description of the dictionary operations. The next section turns to the topic of checking if a set of XEIDs captures the intended semantics of the enterprise.

5. Experimenting with a Database Description

We address in this section the problem of verifying if a set of XEIDs captures the intended semantics of the enterprise. Our approach is not to verify if a given set of XEIDs is logically equivalent to another formal description of the enterprise. Rather, we sketch tools to check if a database description has certain characteristics. Section 5.1 introduces a new feature of the language to express logical inference. Section 5.2 discusses a design tool based on the idea of experimenting with test data.

5.1 - The Inference Operator

Suppose that we want to experiment with a set F of XEIDs. One possibility is to check if F implies a new XEID F that we believe must be true about any database state that satisfies F. We assume that F is the set of constraints stored in the dictionary under the current USERID. So, it suffices to define F. This is done exactly as in Section 3.2, except that the insert constraint operator is replaced by the logical consequence operator, L. (no name need be given, since F will not be stored).

For example, let $D=\{R_1[ABC], R_2[DE]\}$ be a database. To check if $\bar{F}$ logically implies F, where $\bar{F} = \{R_1[AB] \subseteq R_2[DE], R_1[AC] \subseteq R_2[DE], R_2: D \rightarrow E\}$ and F is $\forall a \forall b \forall c\ (R_1(a,b,c) \Rightarrow b=c)$, the user would enter $\bar{F}$ (in a fresh dictionary):

(a)

R_1	A	B
$I.a_1$	a	b

R_2	D	E
C.	a	b

(b)

R_1	A	C
$I.a_2$	a	c

R_2	D	E
C.	a	c

(c)

R_2	D	E
$I.a_3$	a	b

CONSEQUENCE
b = c

and then enter F:

(d)

R_1	B	C
L.	b	c

CONSEQUENCE
b = c

The system will answer "true" or "false" in the communication area at the botton of the screen.

The reader is invited to verify that indeed $\bar{F}$ logically implies F, but this should be clear from our choice of variables in the above example.

It should be noted that the inference capability has serious limitations. First, the decision problem for EIDs, which are a special case of XEIDs, is undecidable [CLM]. Hence, we cannot allow the logical consequence operator to be used with any set $\bar{F}$ of XEIDs. In fact, we do not even know if the decision problem is solvable for much simpler classes of dependencies, such as EMVDs [SW], or FDs in the presence of INDs [CFP].

However, we know that there is a polynomial decision procedure for MVDs [Sa]. Thus, the logical consequence operator can be efficiently supported

at least when $\bar{F}$ and F are MVDs.

We conclude this section by observing that the idea of validating a set $\bar{F}$ of XEIDs using logical inference can be generalized through the concept of an Armstrong database state [Fa3] for $\bar{F}$, that is, a database state that satisfies all logical consequences of $\bar{F}$ and no other dependency. This concept was used in [SM] to devise a practical database design tool, quite in the spirit of the Design-by-Example system.

5.2 - Experimenting with Test Data

A second possibility to experiment with a set $\bar{F}$ os XEIDs would be to use test data. That is, the user defines a database state $\bar{I}$ by filling table frames with test data and then asks the system to verify if $\bar{I}$ satisfies $\bar{F}$. The system should reply with YES of NO and, in the later case, indicate which XEIDs in $\bar{F}$ are not satisfied by $\bar{I}$.

We suggest to improve this idea as follows. Instead of just indicating which XEIDs in $\bar{F}$ are not satisfied by $\bar{I}$, the system would also indicate why they are false. The following example helps understand this suggestion. Consider again the database scheme of Section 5.1, $D = \{R_1[ABC], R_2[DE]\}$, with constraints $\bar{F} = \{R_1[AB] \subseteq R_2[DE], R_1[AC] \subseteq R_2[DE], R_2: D \rightarrow E\}$. A user would enter with test data by invoking table frame and filling in table and attribute names as usual. Then, he would fill in entries with test data; a <u>test operator</u> T. placed in the operator field of the first table indicates that it contains test data:

R_1	A	B	C
T.	a	b	c

The system would then reply that the database state $\bar{I}$ just defined violates some constraint, say, $R_1[AB] \subseteq R_2[DE]$. Prompted by the user, the system would then start to transform $\bar{I}$ into a consistent state in successive stages:

(a)

R_2	D	E
	a	b

- to satisfy $R_1[AB] \subseteq R_2[DE]$

(b)

R_2	D	E
	a	b
	a	c

- to satisfy $R_1[AC] \subseteq R_2[DE]$

(c)

CONSEQUENCE
b = c

- to satisfy R_2: D → E

The last step deserves some comment. Instead of actually renaming c to b (or b to c), the system would inform the user that b=c must hold via the consequence box.

The approach we advocate in this section becomes interesting in the following scenario. Suppose that the application being modelled already exists, automated or not. Hence, the database designer has at his disposal real data. He can then start a series of tests using a sample database state constructed from available data.

One of these tests would necessarily be to check if the sample database state is consistent. We went further and proposed a tool that would inform, when the sample state is inconsistent, which changes are necessary to restore consistency. We believe that this type of information helps locate where the database specification does not agree with the semantics of real data. (Obviously, the sample state may fail to be consistent just because the sampling process ignored certain data relationships).

Naturally, the approach described in this section also works when real data is not easily available. In this case, the database designer would have the additional burden of generating a sample database state that corresponds to a real world situation.

This concludes the necessarily sketchy presentation of the validation tools that, together with the constraint definition language, constitute Design-by-Example.

6. Conclusions

We described in this paper a constraint definition language capable of expressing, in a uniform way, most data dependencies found in the litera-

ture. The language has a straightforward syntax in the style of QBE, and requires no training in logical notation. We have also described dictionary facilities to store constraints, again trying to stay close to the standard QBE system.

Besides the constraint language, the Design-by-Example system contains tools to validate a database design. We just sketched three of these tools, using the concepts of logical inference, Armstrong database and test data.

To summarize, the Design-by-Example system consists of a language to document the semantics of a database and tools to check if the database semantics indeed corresponds to the intended semantics of the enterprise. We believe that such system will be useful in the early stages of the design of a database where emphasis is placed on semantic modelling of the enterprise.

Finally, it should be clear that this paper documents just the early stages of the design of Design-by-Example. Further work is needed to consolidate the constraint language and the dictionary interface, before an actual implementation takes place. The validation tools based on logical inference (see Section 5.1) depend on existing algorithms, if we restrict ourselves to the familiar dependencies, such as FDs and MVDs. However, if we want to include also INDs, for example, considerable work is still needed. On the other hand, the validation tool based on test data may be feasible to implement for the full class of XEIDs, depending on the degree of sophistication.

REFERENCES

[ABU] A.V.Aho, C.Beeri and J.D.Ullman, "The Theory of Joins in Relational Databases", ACM - TODS, Vol.4, Nº 3, Sep. 1979.

[Ar] W.W.Armstrong, "Dependency Structures of Data Base Relationships", Proc. IFIP 74, North Holland, 1974.

[BHF] C.Beeri, R.Fagin and J.H.Howard, "A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations", Proc. ACM - SIGMOD Int. Conf. Management of Data, Toronto, Canada, 1977.

[CB] D.D.Chamberlin and R.F.Boyce, "SEQUEL: A Structure English Query Language", Proc. 1974 ACM - SIGMOD Workshop on Data Description, Access and Control, 1974.

[CFP] M.A.Casanova, R.Fagin and C.Papadimitriou, "Inclusion Dependencies and their Interactions with Functional Dependencies", Conf.Principles of Database Systems, Los Angeles, Calif. Mar. 1982.

[CLM] A.K.Chandra, H.R.Lewis and Johann A. Makowsky, "Embedded Implicational Dependencies and their Inference Problem, Report RC8757, IBM Research Lab, Yorktown Heights, N.Y., Mar. 1981.

[Co1] E.F.Codd, "A Relational Model of Data for Large Shared Data Bases", Comm. ACM, Vol.13, Nº 6, Jun. 1970.

[Co2] E.F.Codd, "Relational Completeness of Data Base Sublanguages", Data Base Systems, Courant Computer Science Simposia Series, Vol.6,Englewood Cliffs, N.J., Prentice-Hall, 1972.

[Da] C.J.Date, "An Introduction to Database Systems", Addison Wesley, 1981.

[En] H.B.Enderton, "A Mathematical Introduction to Logic", Academic Press, 1972.

[Fa1] R.Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases", ACM - TODS, Nº2, Vol. 3, Sep. 1977.

[Fa2] R.Fagin, "A Normal Form for Relational Databases that is based on Domains and Keys, ACM - TODS, Vol. 6, Nº3, Sep. 1981.

[Fa3] R.Fagin, "Horn Clauses and Database Dependencies", Proc. ACM - SIGACT Symp. Theory of Computing, 1980.

[FMUY] R.Fagin, D.Maier, J.D.Ullman and M.Yannakakis, "Tools for Template Dependencies", IBM Report RJ3033, San Jose, Calif., May 1980.

[GJ] J.Grant and B.E.Jacobs, "On Generalized Dependencies", to appear.

[HSW] G.D.Held, M.R.Stonebraker and E.Wong, "INGRES - A Relational Data base System", Proc. NCC44, 1975.

[MMS] D.Maier, A.Mendelzon and Y.Sagiv, "Testing Implications of Data Dependencies", ACM - TODS, Vol.4, Nº 4, Dec. 1979.

[Pi] A. Pirote, "High Level Data Base Query Languages", Advances in Data Base Theory, Vol. 1, ed. H. Gallaire, J.Minker and J.M. Nicolas , Plenum Press, N.Y., 1978.

[Re] P.Reisner, "Human Factors Studies of Database Query Languages: A Survey and Assessment", Report RJ3070, IBM Research Lab., San Jose, Calif., Mar. 1981.

[Sa] Y.Sagiv, "An Algorithm for Inferring Multivalued Dependencies that works also for a Subclass of Propositional Logic", VIVCDCS-R-79-954, Dept. Computer Science, Univ. Illinois, Urbana, Jan. 1979.

[SAC] P.G.Salinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie and T.G.Price, "Access Path Selection in a Relational Database Management System", Report RJ2429, IBM Research Lab., San Jose, Calif., Aug. 1979.

[SM] A.M.Silva and M.A.Melkanoff, "A Method for Helping Discover the Dependencies of a Relation", Advances in Data Base Theory, Vol. 1, ed. H.Gallaire, J.Minker and J.M.Nicolas, Plenun Press, N.Y., 1978.

[SU] F.Sadri and J.D.Ullman, "A Complete Axiomatization for a Large Class of Dependencies in Relational Databases", 1980 ACM Symp. Theory of Computing, 1980.

[SW] Y.Sagiv and S.Walecka, "Subset Dependencies as an Alternative to Embedded Multivalued Dependencies", Tech. Rep. UIUCDCS-R-79-980 , Dept. Comp. Science, University of Illinois, 1979.

[WY] E.Wong and K.Youssefi, "Decomposition - A Strategy for Query Processing", ACM - TODS, Vol. 1, Nº 3, Sep. 1976.

[YP] M.Yannakakis and C.Papadimitriou, "Algebraic Dependencies", Proc. 21st. IEEE Symp. Found. Computer Science, 1980.

[Za] C. Zaniolo, "Analysis and Design of Relational Schemata for Database Systems", Ph.D. Dissertation, Tech. Rep. UCLA-ENG-7669, U.California, Los Angeles, Calif., Jul. 1978.

[Zl1] M.M. Zloof, "Query-by-Example", Proc. National Computer Conference, AFIPS Press, Vol. 44, 1975.

[Zl2] M.M. Zloof, "Security and Integrity whithin the Query-by-Example Data Base Management Language", Report RC6982, IBM Research Lab., Yorktown Heights, N.Y., Feb. 1978.