RELATIONAL SCHEMA DESCRIPTION LANGUAGE

INFORMATICA 1/89

Descriptors: RERLATION BASES, PROGRAMMING LANGUAGE, SCHEMA DEFINITION, INTERPRETERS, SDL LANGUAGE

Radojko Miladinović
Dušan Velasević

ABSTRACT: In this paper a schema description language for relational databases is described. The language provides a schema description on which any query language can be defined. The implemented multiuser incremental interpreter for that language is also described.

SADRZAJ: U ovom clanku opisan je jezik za definisanje seme u relacionim bazama podataka. Taj jezik omogucava definisanje seme na kojoj bilo koji upitni jezik moze biti definisan. Za taj jezik realizovan je visekorisnicki inkrementalni interpreter.

INTRODUCTION

Since 1970., when E.F. Codd had defined the relational data model [4,5,6,7], a lot of relational database management systems (RDBMS) was developed and implemented. All these systems can be classified into two groups according to the way of the schema definition. The systems from the first group, for example Relational Database Management System [9,10], have a stand alone schema definition language. The systems from the second group do not provide such a language: the schema definition is realized as a function of data sublanguage, i.e. query language. SYSTEM R [1,3,11] belongs to this group of RDBMS.
The relation between schema description language (SDL) and other languages in RDBMS (query language, data manipulation language - DML, subschema description language and physical database description language) represents a special problem in the database design. All these languages can be implemented as stand alone languages or as extensions of standard programming languages. The majority of RDBMS does not have the independent SDL, query language and DML; in fact, the data definition, data manipulation and query facility are realized as the functions of the special language called the data sublanguage. The data sublanguage can be implemented as a stand alone programming language or as an extension of the host language. If it is implemented as a stand alone language, it is usually called the query language.
Although the most of modern RDBMS have not a separate SDL, there exists a need for such a language which should be general, simple, structured and user-friendly. This language should provide the means for a schema description over which any query language can be defined. A complete functional independence of the schema description process from the data manipulation and queries is achieved in this way. Bearing this in mind, we developed a new SDL and multiuser incremental interactive interpreter for that language.
The language design was influenced by the general principles applied to the other programming languages. We especially emphasized the language reliability, precise syntax and semantics description of the language, orthogonallity and language independence. The SDL structure was designed bearing in mind that the language should be interactive. For this reason, the commands for direct communication with the users are defined, each statement must be written in one line and the language is structured to provide better readability and documentability of SDL programs.

BASIC LANGUAGE ELEMENTS

The following notation is used in the description of SDL elements:

- { } - Braces indicate that one of the elements enclosed must be specified.
- [ ] - Square brackets indicate that one of the elements enclosed may be optionally specified.
- ... - Ellipses indicate that the immediately preceding part of the format may be repeated.
- \- Upper case words are SDL reserved words.
- \- Lower case words indicate the information that should be supplied by the user.

Language alphabet

The complete SDL character set consists of 52 characters. All SDL characters are presented in table 1.

Table 1.

character	name
A,B,...,X,Y,Z	uppercase letters
0123456789	decimal digits
+	plus
-	minus
*	asterisk
/	slash
<	less than
>	greater than
=	equals
(	left parenthesis
)	right parenthesis
.	point
,	comma
:	colon
"	quotation marks
$	dollar sign
_	hyphen
	space

Identifiers

A SDL identifier (or word) is a character string that forms a user defined or a reserved word of not more than 20 characters. It can be any combination of letters, digits and hyphen sign, but hyphen cannot be the first or the last character of the identifier. A reserved word has specific meaning and may be used only in the manner presented in the statement format. Reserved words are chosen carefully so they enable writing of reliable and synoptic programs. The list of the reserved words is given in APPENDIX A.

The user defined words are SDL words that must be supplied by the user to satisfy the format of the statements. The user defined words are variable names, constants, function names, procedures and comments.

Constants and variables

The classification of constants and variables according to their format and type is given in Fig 1.

Variables in SDL are relations and attributes. Attributes are one-dimensional arrays with elements of the same type. Relations are considered as two-dimensional arrays in which all elements in the same column are of the same type; elements in the same row are not obligatory of the same type. The attribute type is explicitly defined by a particular statement in the SDL, and the relation type is implicitly defined through attributes which constitute that relation. The external representation of constants is very simple. It corresponds to the syntax representation of integer and real numbers. For example:

1000, -3.5 0.75E10, 0.32D11, -34E10

The internal representation of constants depends on the context in which constants appear. For example, if variable A.B in the relational expression

A.B > 10000

is declared as a real variable of the extended precision (binary floating point format), then the constant 1000 is presented in the same format.

Alphanumeric strings (literals) are externally represented as a character strings delimited by the quotation marks. The internal representation of literals is completely the same if the data compression is not applied.

Arithmetic expressions

Arithmetic expressions are used to compute new attribute values during updating. Arithmetic expressions are formed with arithmetic operands and arithmetic operators. An arithmetic operand may be a numeric constant and a variable (attribute). Arithmetic operators specify a computation to be performed using the values of arithmetic operands; they produce a numeric value as a result. The operators are: addition(+), subtraction(-), multiplication(*) and division(/). Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. The precedence of the operators is: first (* and /) and second (+ and -). Parentheses can be used to override the normal evaluation order. The attributes which appear in arithmetic expressions must be of type real or integer. The attribute names in arithmetic expressions have the format:

$$[[\left\{\begin{matrix}\text{OLD}\\ \text{NEW}\end{matrix}\right\}]\ \text{relation_name.}]\ \text{attribute_name}$$

The relation name must be specified in front of attribute name if an attribute appears in more than one relation. If an attribute belongs to only one relation, then the relation name is optional. The reserved words OLD and NEW can appear in front of attribute name. These words denote attribute values before and after updating. If they are omitted the immediate attribute values are considered. The examples of arithmetic expressions:

ORDER.QUANTITY - PRODUCT.QUANTITY
OLD EMPLOYEE.SALARY - 1000

Relational expressions

A relational expression may be a simple relational expression, or may be a combination of simple relational expressions, functions and logical operators. A simple relational expression consists of two operands separated by a comparison operator. The comparison operators are: less than(<), greater than(>), less than or equal to(<=), greater than or equal to(=>), equal to(=) and not equal to(<>). An operand may be a constant and a variable (attribute) of any type. An attribute name has the same format as in an arithmetic expression. The logical operators are AND and OR. The precedence of the logical operators is: first AND and second OR. In a relational expression, the simple relational expressions are evaluated first to obtain their values. Evaluation of relational expressions is performed according to an order of precedence assigned to logical operators. Logical operators of equal rank are evaluated from left to right. Parentheses may be used to alter the normal sequence of evaluation, just

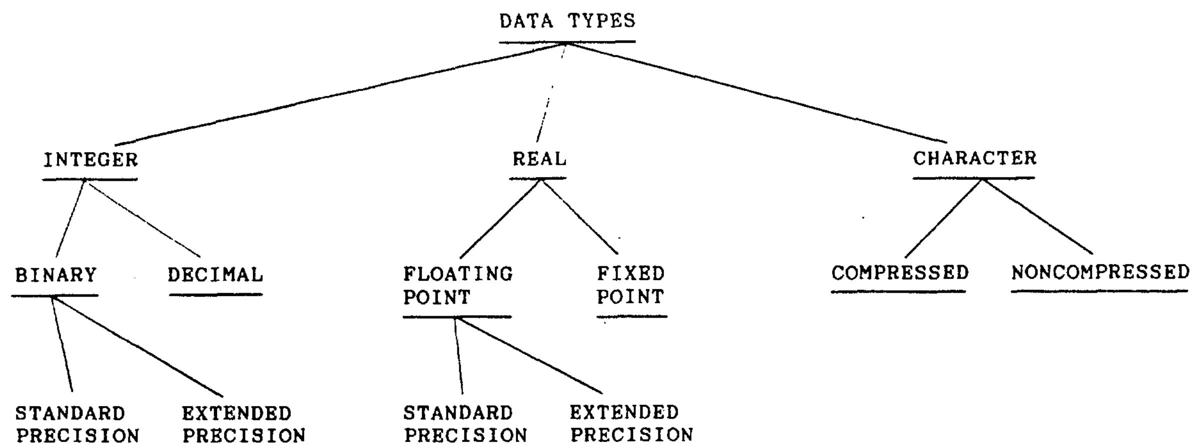

Fig 1. Data typesdefined in SDL

as in arithmetic expressions. The examples of relational expressions:

EMPLOYEE.SALARY > 2000 AND EMPLOYEE.SEX = "M"
OLD EMPLOYEE.SALARY < NEW EMPLOYEE.SALARY

Functions

The functions in SDL are used to define relational expressions that appear in more than one statements. They are defined through FUNCTION statements. The functions also enable the decomposition of long relational expressions, which cannot be written in one line, into several logical parts. Each logical part must represent the complete relational expressions defined by a separate FUNCTION statement. In that way, the long statements containing relational expressions can be spread over several lines. This must be done because a SDL statement can be written in only one line. For example:

(EMPLOYEE.JOB="PILOT" OR
EMPLOYEE.JOB="DRIVER") AND EMPLOYEE.SEX="M"
AND EMPLOYEE.AGE>60

This relational expression cannot be written in one line, so function FUN1 is defined as follows:

FUN1:=EMPLOYEE.JOB="PILOT" OR
EMPLOYEE.JOB="DRIVER"

The relational expression becomes now:

FUN1 AND EMPLOYEE.SEX="M" AND EMPLOYEE.AGE>60

The names of other functions can appear in the function definition.

LANGUAGE DESCRIPTION

Programs written in SDL have specific structure due to the fact that SDL is used only for the relational schema description. The program structure is determined by the structure of the relational database description. Each SDL program contains four type of entries: schema entry, domain entry, attribute entry and relation entry. The entries appear in the program in cited order. Each entry consists of a sequence of statements. Some statements can appear in different entries. The entry specification contains the following parts:

1. Narrative description of entry function
2. General format in which all statements of the entry are presented
3. Description of each statement

The statement specification consists of the following parts:

a. Narrative description of statement function
b. General format that defines the statement parts and their functions
c. Language rules that explain the usage of the statement

Line format

A SDL statement must be written in one line, i.e. the continuation of the statement in a few succeeding lines is not allowed. If any statement can't be written in one line, that statement is divided in several logical parts which should be defined as functions. One or more spaces can appear at the beginning of each line in other to achieve a better program structure and readability. All lexical entities in a statement may be separated by one or more spaces.

Data base example

The application of each statement described is illustrated by the examples which are composed of the following relations:

```
PRODUCT(CODE,    NAME,    PRICE,    QUANTITY)
        DOMCODE DOMNAME DOMPRICE DOMQUANTITY

EMPLOYEE(CODE,   NAME,    SEX,   SALARY)
         DOMCODE DOMNAME DOMSEX DOMSALARY

SUPPLY(CODEPRO, CODESUP, DATE,    QUANTITY)
       DOMCODE  DOMCODE  DOMDATE DOMQUANTITY

DEPT(CODE,    NAME,    ADDRESS,    EMPNO)
     DOMCODE DOMNAME DOMADDRESS DOMNUMBER
```

As we can see, an attribute can appear in more different relations. For each attribute, domain on which the attribute is defined is also represented.

SCHEMA ENTRY

1. The schema entry uniquely identifies schema by its name. Besides, the textual description of the schema contents and cripto protection method applied, if any, is given in the schema entry.
2. Entry format
   SCHEMA statement
   [DESC statement]
   [CRIPTO_PROTECTION statement]
   {domain entry}
   {attribute entry}
   {relation entry}
   ENDSCHEMA statement
3. Statements description
   The schema entry begins with SCHEMA statement and ends with ENDSCHEMA statement. The statements that define the schema are located between the SCHEMA statement and first domain entry in the schema. The order of these statements is irrelevant. In the schema entry, the entries of all domains are defined first, then the entries of all attributes, and finally the relation entries.

SCHEMA statement

a) The SCHEMA statement uniquely identifies schema by its name.
b) Format
   SCHEMA schema_name
c) Language rules
   - Schema name must be unique in the database.

DESC statement

a) The DESC statement is used to describe the content and function of schema, domain, attribute and relation. This statement can appear in any entry in the schema.
b) Format
   DESC comment
c) Language rules
   - An arbitrary number of DESC statements can appear in the schema entry
   - Comment in DESC statement can contain any character from SDL character set.
   - This statement has no meaning for interpreter.
   - If a comment can't be written in one line, several DESC statement must be used.

CRIPTO_PROTECTION statement

a) The CRIPTO_PROTECTION statement can appear in the schema and in relation entry. If it appears in schema entry, all relations in the schema are cripto protected. The cripto protection can be implemented by a special procedure defined by the user or as a standard function in RDBMS.
b) Format

$$\text{CRIPTO_PROTECTION} \begin{Bmatrix} \text{procedure_name} \\ \text{SYSTEM} \end{Bmatrix}$$

c) Language rules
   - Only one CRIPTO_PROTECTION statement can appear in the schema or relation entry.

- If the option SYSTEM is specified, the cripto protection is implemented as one of the activities of RDBMS. If the procedure_name is specified, the cripto protection is implemented by a special procedure.

ENDSCHEMA statement
a) This statements denotes the end of the schema entry, i.e. the end of the whole schema description program.
b) Format
   ENDSCHEMA

DOMAIN ENTRY

1. The domain entry uniquely identifies the domain by its name. Besides, the domain mode, the names of all attributes defined over that domain and the domain units are specified in the domain entry.
2. Entry format
   DOMAIN statement
   DEFINES statement
   [DESC statement]
   MODE statement
   [UNIT statement]
   ENDDOMAIN statement
3. Statements description
   The domain entry begins with the DOMAIN statement and ends with the ENDDOMAIN statement. The order of other statements in the entry is irrelevant.

DOMAIN statement
a) The DOMAIN statement uniquely identifies domain by its name.
b) Format
   DOMAIN domain_name
c) Language rules
   - Domain_name has to be unique in the schema.

DEFINES statement
a) The DEFINES statement specifies the names of all attributes defined over that domain.
b) Format
   DEFINES attr_1,attr_2, ... ,attr_n
c) Language rules
   - An arbitrary number of DEFINES statements can appear in the domain entry.
   - Attributes declared in this statement must be defined in the separate attribute entries.

MODE statement
a) The MODE statement defines the data types in the domain.
b) Format

```
      ( CHARACTER int_1 [COMPRESSED proc_name]               )
      (                                                      )
      (        ( FIXED_POINT     integer_2,int_3 )           )
MODE  ( REAL   (                                 )           )
      (        ( FLOATING_POINT  [EXTENDED]      )           )
      (                                                      )
      (          ( BINARY [EXTENDED] )                       )
      ( INTEGER  (                   )                       )
      (          ( DECIMAL integer_4 )                       )
```

c) Language rules
   - Only one MODE statement can appear in domain entry.
   - Integer data in the database may be represented in the binary or decimal form; the binary integer data can have the standard and extended format. If the decimal base is specified, it is necessary to give the number of decimal digits (integer_4). Default format for integer data is the binary representation in the standard format.
   - Real data can have the fixed or floating point representation in the database. Real data in floating point format is always represented in binary form with standard or extended precision. Real data in fixed point format is always represented in packed decimal form; integer_2 is the total number of decimal digits and integer_3 is the number of digits in the fractional part. The default format for real data is the binary floating point format in standard precision.
   - Character data can be represented in compressed or source form. Integer_1 specifies the number of characters. If COMPRESSED option is declared, the name of the procedure (proc_name) which performs the compression must be declared too.

UNIT statement
a) The UNIT statement defines the input unit for data in the domain. Because the domain values stored in the database can be expressed in different units, the name of the procedure which performs the conversion from input to internal units may be also specified in this statement.
b) Format
   UNIT unit [,procedure_name]
c) Language rules
   - If the procedure_name is not specified, the data in the database are measured by the same units as input data. If the procedure_name is specified, the conversion from input units to internal units must be done. For example, the input unit can be dollar but the internal unit can be million dollars, etc.
   - If this statement is not given in the domain entry, the domain values are numbers or character strings, without a specific context.

ENDDOMAIN statement
a) This statement denotes the end of the domain description.
b) Format
   ENDDOMAIN

ATTRIBUTE ENTRY

1. The attribute entry uniquely identifies the attribute by its name. Besides, the domain name over which the attribute is defined, the names of relations in which the attribute appears and the total number of different values the attribute may assume are given in the attribute entry.
2. Entry format
   ATTRIBUTE statement
   ORIGIN statement
   BELONGS statement
   [DESC statement]
   [CARDINALITY statement]
   [VALUE statement]
   ENDATTRIBUTE statement
3. Statements description
   The attribute entry begins with the ATTRIBUTE statement and ends with the ENDATTRIBUTE statement. The order of other statements is irrelevant.

ATTRIBUTE statement
a) The ATTRIBUTE statement uniquely identifies the attribute by its name.
b) Format
   ATTRIBUTE attribute_name
c) Language rules
   - Attribute_name has to be unique in the schema
   - Attribute_name must appear in DEFINES statement in the entry of the domain whose name is declared in the ORIGIN statement in this attribute entry.

- Attribute_name must appear in the CONTAINS statement in the entry of the relation whose name is declared in the BELONGS statement in this attribute entry. An attribute may appear in an arbitrary number of relations.

ORIGIN statement
a) The ORIGIN statement identifies the domain over which this attribute is defined.
b) Format
ORIGIN domain_name
c) Language rules
- Domain whose name appears in this statement must be declared in the separate domain entry.

BELONGS statement
a) This statement identifies the relations in which this
attribute appears.
b) Format
BELONGS relation_1, relation_2, ...
c) Language rules
- An arbitrary number of the BELONGS statements may appear in the attribute entry.
- Relations whose names appear in this statement must be declared in the separate relation entries.

CARDINALITY statement
a) The CARDINALITY statement specifies the total number of different attribute values. This statement specifies the total number of n-tuples in the relation, if the attribute is the primary key of the relation.
b) Format
CARDINALITY integer
c) Language rules
- Only one CARDINALITY statement may appear in the attribute entry.
- The number of different attribute values is unlimited if this statement does not appear in the attribute entry.

VALUE statement
a) This statement defines an implicit value that should be assigned to the attribute if the attribute value is not assigned during the loading of the database.
b) Format
VALUE constant
c) Language rules
- Only one VALUE statement may appear in the attribute entry.
- The constant can be of the type real, integer or character depending on the attribute type.
- If this statement is omitted, the attribute values must be assigned during the database loading.

ENDATTRIBUTE statement
a) This statement declares the end of the attribute entry.
b) Format
ENDATTRIBUTE

RELATION ENTRY

1. The relation entry uniquely identifies the relation by its name. Besides, the attributes belonging to that relation, the integrity constraints, the primary key, the cripto protection method and dependencies among this relation and other relations are specified in the relation entry.
2. Entry format
RELATION statement
CONTAINS statement
KEY statement
[DESC statement]
[CRIPTO_PROTECTION statement]
[INTEGRITY_CONSTRAINT statement]
[UNIQUE statement]
[FUNCTION statement]
[ACCESS_CONTROL statement]
[TRIGGER structure]
ENDRELATION
3. Statements description
The order of the statements in the relation entry is irrelevant.

RELATION statement
a) The RELATION statement uniquely identifies schema by its name.
b) Format
RELATION relation_name
c) Language rules
- Relation_name has to be unique in the schema.
- Relation_name must be declared in the BELONGS statement of all attributes that belongs to the relation.

CONTAINS statement
a) The CONTAINS statement specifies the names of all attributes in the relation.
b) Format
CONTAINS attribute_1, attribute_2, ...
c) Language rules
- At least one attribute name must be declared in this statement.
- An arbitrary number of the CONTAINS statements can appear in the relation entry.
- The attributes declared in this statement must be defined in the separate attributes entries.

KEY statement
a) The KEY statement defines the primary key of the relation.
b) Format
KEY attribute_1,attribute_2, ...
c) Language rules
- Only one KEY statement can appear in the relation entry.
- At least one attribute name must be declared in the KEY statement.
- The attributes declared in this statement must be defined in the separate attribute entries.
- The attributes declared in this statement must belong to this relation, i.e. they must be defined in the CONTAINS statement of this relation entry.

INTEGRITY_CONSTRAINT statement
a) This statement defines the integrity constraints in the relations. The constraints can be static and dynamic.
b) Format

INTEGRITY_CONSTRAINT $\left\{ \begin{array}{l} \text{r_exp_1} \\ \text{fun_1} \end{array} \right\}$ [IF $\left\{ \begin{array}{l} \text{r_exp_2} \\ \text{fun_2} \end{array} \right\}$]

c) Language rules
- An arbitrary number of the this statements can appear in the relation entry;
- Reserved words OLD and NEW can appear in the relational_exp_1 but not in the relational_exp_2;
- An arbitrary number of the this statements can be defined for one attribute because the attribute can have more than one integrity constraint;
- Static and dynamic integrity constraints for one attribute must be defined in different INTEGRITY_CONSTRAINT statements

- Integrity constraint is defined by relational_exp_1 or by function_1. Function is defined in the FUNCTION statement;
- Integrity constraint represents the comparison between old (OLD) and new (NEW) attribute values, if the integrity constraint is dynamic. In that case, attribute names must include reserved words OLD and NEW. If the integrity constraint is static, the attribute names in the relational expressions represents immediate values and reserved words OLD and NEW are not included in the attributes names.
- IF option specifies the attribute values on which the integrity constraint is applied. If this option is omitted, the integrity is valid for all attribute values.

Examples:

```
INTEGRITY_CONSTRAINT SUPPLY.CODEPRO =
          PRODUCT.CODE
INTEGRITY_CONSTRAINT NEW  EMPLOYEE.SAL > OLD
                         EMPLOYEE.SAL
INTEGRITY_CONSTRAINT FUN3 IF
   EMPLOYEE.DEPTCODE="B"
INTEGRITY_CONSTRAINT FUN2 IF EMPLOYEE.SEX="M"
FUNCTION FUN2:=EMPLOYEE.SAL > 5000
FUNCTION FUN3:=NEW EMPLOYEE.SAL>OLD EMP.SAL
```

The integrity constraint in the first example specifies that all values of the attribute CODEPRO in the relation SUPPLY must be equal to the values of the attribute CODE in relation PRODUCT. Second example specifies that new salaries of all employees must be greater than old salaries. The usage of the IF option is shown in the third example. This example specifies that new salary must be greater than old salary, but only for employees in department "B". The fourth example defines integrity constraint that the male employees have salary greater than 5000. In the third and fourth example functions FUN2 and FUN3 are used. These functions are defined by the FUNCTION statements.

UNIQUE statement

a) The UNIQUE statement identifies attribute or attributes group having unique values in the relation. That attribute or attributes group do not represent the primary key.

b) Format

   UNIQUE attribute_1, attribute_2, ...

c) Language rules
- An arbitrary number of the UNIQUE statements can appear in the relation entry.
- The attributes declared in this statement must be defined in the separate attributes entries;
- The attributes declared must belong to this relation, i.e. they must be declared in the CONTAINS statement of the relation entry.
- If more than one attribute is declared in the UNIQUE statement, that attribute group has unique values, not the particular attributes in that group.
- If more than one attribute or attributes group in the relation have unique values, that should be specified by separate UNIQUE statements;
- If UNIQUE statement doesn't appear in the relation entry, only attributes that constitute primary key have unique values.

FUNCTION statement

a) The FUNCTION statement defines a function.

b) Format

   FUNCTION function_name:=relational_exp

c) Language rules
- Function_name must be unique in the schema;
- The other function names can appear in the relational expression.

ACCESS_CONSTRAINT statement

a) The ACCESS_CONSTRAINT statement specifies the operations that cannot be performed on all or particular n-tuples, i.e. on all or particular attribute values.

b) Format

```
             (READ  )                   (r_exp)
ACCESS_C FOR (UPDATE) [ON attr_1] [IF (     )]
             (INSERT)                   (funct)
             (DELETE)
```

c) Language rules
- An arbitrary number of this statements can appear in the relation entry.
- One ACCESS_CONSTRAINT statement must be specified for each forbidden operation;
- The operations INSERT and DELETE denote the insertion and deletion of n-tuples. They must be applied to n-tuples, not to the attribute values. The READ operation denotes the reading of n-tuples or attribute values, and UPDATE operation denotes the updating of one or more attributes in the relation. The READ and UPDATE operations can be applied to n-tuples and attribute values. If ACCESS_CONSTRAINT specifies that UPDATE or READ are forbidden on some attributes, the ON option specifies these attributes. If ON option is omitted, UPDATE or READ are forbidden for all attributes of the relation. The ON option can appear in this statement only for READ and UPDATE operations.
- IF option specifies n-tuples or attribute values for which the given operation is forbidden. If this option is omitted the requested operation is forbidden for all n-tuples or for all attribute values;
- If the ACCESS_CONSTRAINT statement is not included in the relation entry description all operations over n-tuples or attributes of the relation are allowed

Examples:

```
ACCESS_CONSTRAINT FOR DELETE
ACCESS_CONSTRAINT FOR UPDATE ON CODE
ACCESS_CONSTRAINT FOR READ IF CODE="C"
ACCESS_CONSTRAINT FOR READ ON SAL IF CODE="C"
```

All examples are defined over the relation EMPLOYEE. First example means that the DELETE operation is forbidden for all n-tuples of relation EMPLOYEE and the second example denotes that the updating of the attribute CODE is not allowed. Third example means that n-tuples in the relation EMPLOYEE having the value of attribute CODE equal "C" cannot be read, and the fourth example means that the values of attribute SALARY of the n-tuples with attribute CODE="C" cannot be read.

TRIGGER structure

a) The TRIGGER structure defines the set of forced operations that must be performed over other relations upon the finishing of the current operation.

b) Format

```
            (INSERT)
TRIGGER FOR (UPDATE) [ON attribute_1]
            (DELETE)

(INSERT)                   (const)      (r_exp)
(UPDATE) rel_1 [SET atr_2:=(     )  [IF (     )]]
(DELETE)                   (a_exp)      (fun  )

ENDTRIGGER
```

c) Language rules
- TRIGGER contains the head line, one or

more lines in which the set of forced operations is defined and the end line;
- An arbitrary number of triggers can appear in the relation entry;
- Relation_1 is the name of the relation to which the forced operation is applied. Attribute_2 is an attribute from that relation;
- Reserved words OLD and NEW can appear in the relational expression of the IF option.
- The FOR option in the head line specifies the operation that causes the trigger;
- If the operation declared in head line is UPDATE, then the ON option must exist. The ON option specifies the attribute in the current relation over which the UPDATE operation (causing the trigger) is performed. If the operation declared in the head line is INSERT or DELETE, the ON option must not appear in the head line because these operations are performed over the n-tuples not over the attribute values;
- The SET option exists only if the forced operation declared is UPDATE. One SET option exists in the same trigger for each attribute updated. For example, if the trigger causes the updating of two attributes (attr_2 and attr_3) in relation_1, then the trigger must contain the following lines:

```
                              ⎧ constant
UPDATE rel_1 SET attr_2:=     ⎨
                              ⎩ arithmetic_exp
                              ⎧ constant
UPDATE rel_1 SET attr_3:=     ⎨
                              ⎩ arithmetic_exp
```

The arithmetic expression can be defined in the SET option only if the attr_2 and attr_3 are REAL or INTEGER.
- The IF option specifies the n-tuples, i.e. attribute values in relation_1 over which the operations defined by the trigger are performed. If this option is omitted, the forced operations are applied to all n-tuples in relation_1, i.e. to all attribute values;
- The n-tuples, i.e. attribute values to which the forced operations must be applied are determined by relational expression or by the function whose name appears in the IF option.

Examples:

```
TRIGGER FOR UPDATE ON CODE
UPDATE SUPPLY SET CODEPRO:=PRODUCT.CODE
ENDTRIGGER
TRIGGER FOR UPDATE ON DEPTCODE
UPDATE DEPT SET EMPNO:=DEPT.EMPNO+1 IF FUN1
UPDATE DEPT SET EMPNO:=DEPT.EMPNO-1 IF FUN2
ENDTRIGGER
FUNCTION FUN1:=DEPT.CODE=NEW EMPLOYEE.DEPTCODE
FUNCTION FUN2:=DEPT.CODE=OLD EMPLOYEE.DEPTCODE
```

First trigger updates the relevant CODEPRO entries in the relation SUPPLY whenever the CODE of the PRODUCT is updated. This trigger is defined in the relation PRODUCT. Second trigger updates the relevant EMPNO entries in the relation DEPT whenever the DEPTCODE of the EMPLOYEE is updated. This trigger is defined in the relation EMPLOYEE.

ENDRELATION statement

a) This statement denotes the end of the relation entry.

b) Format
ENDRELATION

The key words of SDL are given in APPENDIX A, the formal syntax description in APPENDIX B and the schema description for the example in APPENDIX C.

THE INTERPRETER REALIZATION

SDL commands

Bearing in mind that SDL is an interactive language it contains a set of commands which enables the user to list the program, run it, modify, etc. SDL commands are: LIST, INSERT, DELETE, SUBSTITUTE, RUN, PURGE, SAVE, OLD, TERMINATE, PROMPT, TEST and EXIT. The majority of these commands are the standard commands that exist in each interactive language and they have a usual meaning. For example, RUN command runs the SDL program, the OLD command brings an existing SDL program from the file to the user area in the main memory, the PURGE command deletes all program versions but the last one etc. Only TEST and TERMINATE commands have specific role in SDL. The TEST command enables the user to control and track the interpretation process, and the TERMINATE command requests from the interpreter to finish the program interpretation after its modification is done.

Interpreter structure

The interpretation is performed in two phases. In the first phase, the source program is analyzed (statement by statement) and translated into an internal form (postfix notation). The internal form of the program is interpreted and the result is generated in the second phase. Two requests are imposed to the implementation of the SDL interpreter: a) the interpreter must be incremental, b) the multi user environment must be provided.
As a result of the interpretation process, the description file which contains the description of all elements of the database and their internal dependencies is created. Each record in that file corresponds to one element of the database (schema, domain, attribute and relation).
The interpreter is implemented as a set of six internally connected program modules, Fig. 2. The interpreter modules are: command module, lexical analyzer, parser, semantic analyzer, result generator and editor. The control communications between modules are presented in full lines, and the data flow in dashed lines.
The data structures (static and dynamic) that are used during the interpretation process are: source program, internal program, dictionaries and description file. The dictionaries contain all information about the objects of the source program (operators and operands). These information are organized so they can be accessed from any part of the interpreter. There exist two static dictionaries which are parts of the interpreter (dictionary of reserved words and dictionary of delimiters) and one dynamic dictionary (symbol table). The static dictionaries are one dimensional arrays with the elements of fixed length. The dictionary of delimiters contains all arithmetic operators, comparison operators and special characters as comma, colon etc.
The symbol table is formed during the translation of the source program. All information about constants and variables from a SDL program are stored into it. The symbol table has the fixed and variable part. The fixed part contains five fields: identifier name, object type indicator (R - relation, A - attribute, D - domain, etc.), special indicator used for semantic control (if the given object is completely defined this indicator has the value one, otherwise null) and the pointer to the variable part of the symbol table. The fields of the variable part depend on the database element to whom the

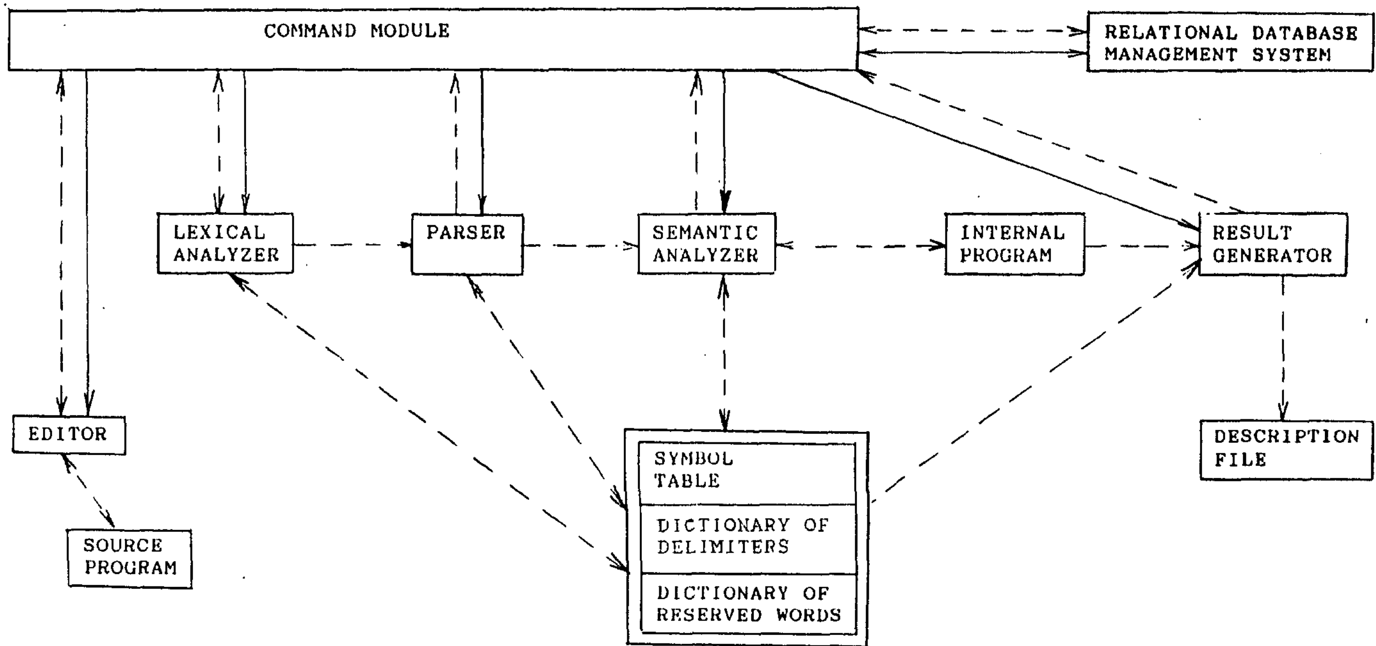

Fig. 2. The interpreter structure

given entry belongs. The number of fields in the variable part and their format for different element types (relations, domains etc.) are different, because the quantity of information that must be stored is different.
All errors detected during the program translation can be classified into five groups: lexical, syntax, semantic, editor and command errors. These errors are detected by the corresponding modules. Upon the detection of any error the interpretation process is interrupted, the command module takes over the control and sends the error message to the user. Due to the concept of the incremental interpretation, each error can be immediately corrected by the user , and the interpretation process continued.
Some diagnostic facilities are built in to enable the tracking of the interpretation process. Each phase of the interpretation process can be easily tracked independently of other phases by printing the results of that phase. For example, the lexical analysis can be tracked by printing the set of tokens which was generated during the lexical analysis. A special SDL command (TEST) specifies the interpretation phase that the user wants to track.
The communication of the SDL interpreter with other parts of the RDBMS is provided through virtual calls. In this way, the interpreter is made self-contained. In the RDBMS environment, the virtual calls must be replaced by the actual ones.

Functional analysis of interpreter parts

The lexical analyzer, parser, semantic analyzer and editor are implemented using the well known methods, because they are the standard parts of all interpreters [2,8]. The other modules are specific because their functions are dictated by the SDL characteristics.
The command module is the main interpreter module which coordinates the work of other modules and communicates with other parts of the RDBMS. This module starts and terminates the interpretation process; it accepts the statements and commands and sends messages to the user, activates other modules to do their job, performs the memory management and provides the multi-user environment.
The command module accepts the statements and commands either from terminal and or file. If the user entered a command, the command module performs the appropriate action. For example if RUN command was given, the command module activates the execution of a SDL program (in fact, activates the result generator).
If the user entered a statement, the lexical analyzer, parser and semantic analyzer are activated to perform the analysis and translation of the source statement into the internal form. If any of these modules detects an error, it informs the command module by setting the error indicator. In that case, the command module interrupts the interpretation process and sends the message to the user.
The lexical analyzer extracts tokens, forms the symbol table (fixed part) and writes into it all available information about identificators. The parser and semantic analyzer fill in the rest of the symbol table. The tokens are divided into four classes: identifiers (variable names), constants, reserved words and delimiters.
The parser performs syntax analysis of the source statements and their translation into the internal form. The internal form of the statements is postfix notation. The parser is implemented by recursive descent method [2].
The semantic analyzer performs the semantic analysis and writes corresponding information into the symbol table and internal program. The semantic analysis includes the control of the whole program and particular statements. The control of semantic correctness of the whole program includes the checking the presence of all prerequisite statements, checking the order in which statements appear and checking the uniqueness of the statements that can appear only once in the program, in the entry or in the some block structure. The consistency of the operand and operator types and the uniqueness of the identifiers (that must be unique) are also checked.
The implementation of the semantic analyzer, especially the part which controls the program form, is based on the theory of the finite

state machine. The semantic control of the single statements is done by the control routines that are called at particular places in the program structure which simulates that finite state machine [8].
The result generator, which is activated by the RUN command, creates the description file from the symbol table and internal program. The description file contains the descriptions of all database elements (schema, domains, attributes and relations). Due to the fact that SDL has no imperative statements, the result is generated by searching the symbol table and internal program and gathering all necessary information for the description file. The records in the description file have four different formats, and each record format corresponds to one data structure in the database (schema, domain, attribute and relation). The records of the same format are grouped together, so the physical description file contains four logical files. The result generator is implemented as a set of procedures and each procedure forms a description of one data type.
The editor is the module whose task is to provide the editing of the source program and support the incremental interpretation of the SDL program.
The multiuser SDL interpreter is implemented on the IBM 4331 computer using COBOL programming language. The system software for interactive work CICS [13] (which enables dynamic memory and process management, data interchange with disks and communications with the users over terminals) is used for implementation. The description file is formed as IBM VSAM KSDS data file with variable length records [12]. The interpreter volume expressed by the number of program lines is 5500.

CONCLUSION

The schema description language for relational database is a stand alone language completely independent from other languages in the DBMS (query language, data manipulation language etc.). Due to the fact that the relational database description have four different data types (schema , attribute, domain and relation), the SDL statements are grouped into four entries, so each entry describes one data type. The SDL is a structured language which provides a high readability of the SDL programs.
The interpreter developed for this language is characterized by its functional independence from the query language, the multiusesr environment and incremental interpretation.
The implementation of the interpreter provides the conditions for its easier portability to different machines. These conditions are: a) the application of standard programming language characteristics without any extensions, b) concentration of input/output activities in command module c) marking all machine dependent points in the interpreter (for example, calls of assembler routines, system service calls etc.) so they become immediately visible.

REFERENCES

1. Astrahan M. M. : "SYSTEM R: A relational database management system"; Computer, may 1979.

2. Brown P. J.: "Writing Interactive Compilers and Interpreters"; John Wiley and Sons.

3. Chamberlain D. D. etc: "A history and evolution of SYSTEM R"; Communications of ACM, no 10, october 1981.

4. Codd E.F.:"A relational model of data for large shared data banks"; Communications of the ACM, no 6, june 1970.

5. Codd E.F.:"Further Normalization of the Database Relational Model";Current Computer Science Simposia, Vol 6, Database Systems, New York city, May 1971, Prentice Hall.

6. Codd E.F.:"Normalized Database Structure: A Brief Tutorial"; Current Computer Science Simposia, Vol 6, Database Systems, New York city, May 1971, Prentice Hall.

7. Codd E.F.:"Relational Completeness of Database Sublanguages"; Current Computer Science Simposia, Vol 6, Database Systems, New York city, May 1971, Prentice Hall.

8. Gries D.: "Compiler Design for Digital Computers"; John Wiley and Sons.

9. Hutt A. T. F.:"Organizing the Description of a Relational Database";Software - Practice and Experience, vol k9, 1979.

10. Hutt A.T.F.: "A Relational Data Base Management System"; John Wiley and Sons, 1979.

11. Gray J., McJones P.:"The recovery manager of SYSTEM R database"; ACM Computing Surveys, no 2, june 1982.

12. VSE (Virtual Storage Extended) System Data Management Concepts - IBM Laboratory, Programming Publication Department, Boebling, W. Germany.

13. CICS/VS (Customer Information Control System/Virtual Storage) Introduction to Program Logic - IBM Laboratory, Technical Documentation Department, Hampshire, England.

APPENDIX A: Reserved words

ACCESS_CONSTRAINT	FUNCTION
AND	IF
ATTRIBUTE	INSERT
BELONGS	INTEGER
BINARY	INTEGRITY_CONSTRAINT
CARDINALITY	KEY
CHARACTER	MODE
COMPRESSED	NEW
CONTAINS	OLD
CRIPTO_PROTECTION	ON
DECIMAL	OR
DEFINES	ORIGIN
DELETE	READ
DESC	REAL
DOMAIN	RELATION
ENDATTRIBUTE	SCHEMA
ENDDOMAIN	SET
ENDRELATION	SYSTEM
ENDSCHEMA	TRIGGER
ENDTRIGGER	UNIT
EXTENDED	UNIQUE
FIXED_POINT	UPDATE
FLOATING_POINT	VALUE
FOR	

APPENDIX B: SDL syntax

Here we present the BNF syntax definition for SDL. In this notation square brackets [] indicate optional constructs, and braces {} indicate constructs that appear one or more times.

```
<SDL_program>::=<schema_entry>
{<domain_entry>}
                {<attribute_entry>}
                {<relation_entry>}

<schema_entry>}::=<schema_stat>
                [<description_stat>]
                [<cripto_protection_stat>]
                <endschema_stat>
<schema_stat>::=SCHEMA <schema_name>
<schema_name>::=<name>
<name>::=<letter>{<hyphen><alphanumeric>
        | <alphanumeric>}
<alphanumeric>::=<letter> | <digit>
<letter>::= A | B | C | D | ... | Z
<digit>::= 0 | 1 | 2 ... | 9
<hyphen>::= _
<description_stat>::=DESC <character_string>
<character_string>::={<character>}
<character>::=<alphanumeric>
             | <special_character>
<special_character>::= <addition_op>
                       | <multiplication_op>
                       | <comparison_op>
     | " | , | . | : | $ | ( | ) | <hyphen>
<addition_op>::= + | -
<multiplication_op>::= * | /
<comparison_op>::= < | > | = | => | <= | <>
<cripto_pr_stat>::=CRIPTO_PROTECTION <method>
<method>::=<procedure_name>
         | SYSTEM
<endschema_stat>::=ENDSCHEMA

<domain_entry>::=<domain_stat>
                 <defines_stat>
                 <mode_stat>
                 [<description_stat>]
                 [<mode_stat>]
                 <enddomain_stat>
<domain_stat>::=DOMAIN <domain_name>
<domain_name>::=<name>
<defines_stat>::=DEFINES {<attribute_name>}
<mode_stat>::=MODE <domain_type>
<domain_type>::=<character_type>
              | <integer_type>
              | <real_type>
<character_type>::=CHARACTER <integer_number>
           [COMPRESSED <procedure_name>]
<real_type>::=FIXED_POINT <integer_number>,
                          <integer_number>
           | FLOATING_POINT [EXTENDED]
<integer_type>::=INTEGER <number_type>
<number_type>::=DECIMAL <integer_number>
              | BINARY [EXTENDED]
<unit_stat>::=UNIT <unit> [,<procedure_name>]
<unit>::=<character_string>
<enddomain_stat>::=ENDDOMAIN

<attribute_entry>::=<attribute_stat>
                    [description_stat>]
                    [<value_stat>]
                    <origin_stat>
                    <belongs_stat>
                    [<cardinality_stat>]
                    <endattribute_stat>
<attribute_stat>::=ATTRIBUTE <attribute_name>
<attribute_name>::=<name>
<origin_stat>::=ORIGIN <domain_name>
<belongs_stat>::=BELONGS  {<relation_name>}
<cardinality_stat>::=CARDINALITY
                     <integer_number>
<integer_number>::={<digit>}
<value_stat>::=VALUE <constant>
<constant>::=<numeric_constant> | <literal>
<numeric_constant>::=[<sign>] <integer_number>
                   | [<sign>] <real_number>
<sign>::= + | -
<real_number>::= <integer_number>.
                 <integer_number><exp>
<exp>::=D[<sign>]<integer_number>
       |E[<sign>]<integer_number>
<literal>::="<character_string>"
<endattribute_stat>::=ENDATTRIBUTE

<relation_entry>::=<relation_stat>
                   [<description_stat>]
                   [<cripto_protection_stat>]
                   <key_stat>
                   <contains_stat>
                   [<integrity_stat>]
                   [<unique_stat>]
                   [<function_stat>]
                   [<access_constraint_stat>]
                   [<trigger_block>]
                   <endrelation_stat>
<relation_stat>::=RELATION <relation_name>
<relation_name>::=<name>   .
<key_stat>::=KEY {<attribute_name>}
<contains_stat>::=CONTAINS {<attribute_name>}
<integrity_stat>::=INTEGRITY_CONSTRAINT
           <constraint> [IF <condition>]
<constraint>::=<relational_expression>
             | <function_name>
<condition>::=<relational_expression>
            | <function_name>
<relational_expression>::=<simple_exp>
    | <simple_exp><log_op><function_name>
    | <simple_exp><log_op>(<relational_exp>)
<log_op>::= AND | OR
<simple_exp>::=<variable>
               <comparison_op><operand>
<operand>::=<variable> | <constant>
<variable>::=<prefix><e_variable>
<prefix>::=OLD | NEW
<e_variable>::=[<relation_name>].
               <attribute_name>
<function_name>::=<name>
<unique_stat>::=UNIQUE {<attribute_name>}
<function_stat>::=FUNCTION
<function_name>::<logical_exp>
<access_constraint_stat>::=ACCESS_CONSTRAINT
        FOR <operation_1>
        [ON <attrname>] [IF <condition>]
<operation_1>::= READ | <operation>
<operation>::= INSERT | UPDATE | DELETE
<trigger_block>::=<head_line> <trigger_body>
                  <end_line>
<head_line>::=TRIGGER FOR <operation>
          [ON <attribute_name>]
<trigger_body>::=<operation> <relation_name>
     [SET <attribute_name>:=<expression>
     [IF <condition>]]
<expression>::=<constant> | <arithmetic_exp>
<arithmetic_exp>::=<sign><operand>
          | <sign><operand><arit_operator>
          (<arithmetic_exp>)
<sign>::= + | -
<operand>::=<variable> | <numeric_constants>
<arit_operator>::= + | - | * | /
<end_line>::=ENDTRIGGER
```

APPENDIX C: SDL program

```
SCHEMA MARKETING
 DESC There is no protection at schema level
DOMAIN DOMADDRESS
 DEFINES ADDRESS
  MODE CHARACTER COMPRESSED PROC1
   ENDDOMAIN
DOMAIN DOMCODE
 DEFINES CODE,CODEPRO,CODESUP,DEPTCODE
  MODE INTEGER BINARY
   ENDDOMAIN
DOMAIN DOMDATE
 DEFINES DATE
  MODE CHARACTER
   ENDDOMAIN
DOMAIN DOMNO
 DEFINES EMPNO
```

```
 DEFINES EMPNO
  MODE INTEGER DECIMAL
   ENDDOMAIN
DOMAIN DOMPRICE
 DEFINES PRICE
  MODE REAL FIXED_POINT
   UNIT DOLLAR
    ENDDDOMAIN
DOMAIN DOMQUANTITY
 DEFINES QUANTITY
  MODE REAL FIXED_POINT
   ENDDOMAIN
DOMAIN DOMNAME
 DEFINES NAME
  MODE CHARACTER COMPRIMED PROC2
   ENDDOMAIN
DOMAIN DOMSALARY
 DEFINES SALARY
  MODE REAL FLOATING_POINT
   UNIT DOLLAR
    ENDDOMAIN
DOMAIN DOMSEX
 DEFINES SEX
  MODE CHARACTER
   ENDDOMAIN
ATTRIBUTE ADDRESS
 ORIGIN DOMADDRESS
  BELONGS DEPT
   ENDATTRIBUTE
ATTRIBUTE CODE
 ORIGIN DOMCODE
  BELONGS SUPPLY,PRODUCT,EMPLOYEE,DEPT
   ENDATTRIBUTE
ATTRIBUTE CODEPRO
 ORIGIN DOMCODE
  BELONGS SUPPLY
   CARDINALITY 5000
    ENDATTRIBUTE
ATTRIBUTE CODESUP
 ORIGIN DOMCODE
  BELONGS SUPPLY
   CARDINALITY 100
    ENDATTRIBUTE
ATTRIBUTE DEPTCODE
 ORIGIN DOMCODE
  BELONGS EMPLOYEE
   ENDATTRIBUTE
ATTRIBUTE DATE
 BELONGS SUPPLY
  ORIGIN DOMDATE
   VALUE "01/01/1980"
    DESC Data when the product was bought
     ENDATTRIBUTE
ATTRIBUTE EMPNO
 ORIGIN DOMNO
  BELONGS DEPT
   DESC Number of employees in the department
    VALUE 0
     ENDATTRIBUTE
ATTRIBUTE QUANT
 ORIGIN DOMQUANTITY
  BELONGS PRODUCT,SUPPLY
   VALUE 0
    ENDATTRIBUTE
ATTRIBUTE NAME
 ORIGIN DOMNAME
  BELONGS PRODUCT,EMPLOYEE,DEPT
   VALUE "XXXX"
    ENDATTRIBUTE
ATTRIBUTE PRICE
 ORIGIN DOMPRICE
  BELONGS PRODUCT
   VALUE 100
    ENDATTRIBUTE
ATTRIBUTE SALARY
 ORIGIN DOMSALARY
  BELONGS EMPLOYEE
   VALUE 1000
    ENDATTRIBUTE
ATTRIBUTE SEX
 ORIGIN DOMSEX
  BELONGS EMPLOYEE
   CARDINALITY 2
    VALUE "M"
     ENDATTRIBUTE
RELATION PRODUCT
 CONTAINS CODE,NAME,PRICE,QUANT
  KEY CODE
   DESC Products in the supply
    INTEGRITY_CONSTRAINT CODE>0 AND CODE<32767
    INTEGRITY_CONSTRAINT PRICE<999.99
     TRIGGER FOR UPDATE ON QUANT
     UPDATE SUPPLY SET QUANT:=SUPPLY.QUANT+NEW
PRODUCT.QUANT
     ENDTRIGGER
    TRIGGER FOR UPDATE ON CODE
    UPDATE      SUPPLY      SET      CODEPRO:=NEW
PRODUCT.CODE
     ENDTRIGGER
ENDRELATION
RELATION EMPLOYEE
 CONTAINS CODE,NAME SEX,SALARY,DEPTCODE
  KEY CODE
   DESC Company employees in the last 5 years
   UNIQUE DEPTCODE
    TRIGGER FOR UPDATE ON DEPTCODE
    UPDATE DEPT SET EMPNO:=EMPNO+1 IF FUN1
    UPDATE DEPT SET EMPNO:=EMPNO-1 IF FUN2
    ENDTRIGGER
   FUNCTION FUN1:=DEPT.CODE=NEW DEPTCODE
   FUNCTION FUN2:=DEPT.CODE=OLD DEPTCODE
  INTEGRITY_CONSTRAINT SEX="M" OR SEX="F"
  INTEGRITY_CONSTRAINT CODE>1 AND CODE<32767
  INTEGRITY_CONSTRAINT DEPTCODE=DEPT.CODE
 ACCESS_CONSTRAINT   FOR   READ   ON   SALARY   IF
CODE<>1
ENDRELATION
RELATION SUPPLY
 CONTAINS CODEPRO,CODESUP,DATE,QUANT
  KEY CODEPRO,CODESUP
   INTEGRITY_CONSTRAINT CODEPRO=PRODUCT.CODE
  INTEGRITY_CONSTRAINT DATE>"01/01/1980"
ENDRELATION
RELATION DEPT
 CONTAINS CODE,NAME,ADDRESS,EMPNO
  KEY CODE
   INTEGRITY_CONSTRAINT CODE>1 AND CODE<10
  UNIQUE NAME
ENDRELATION
ENDSCHEMA
```