SQL/DDL versus VDM-SL

2000/01

J.N. Oliveira

jno@di.uminho.pt

Dept. Informática, Universidade do Minho

SQL/DDL vs VDM-SL

- **DDL** = Data Definition (Sub) Language
- Queremos tornar descrições em DDL o mais precisas possível:

DDL ⟶ **VDM-SL**

- Vamo-nos basear em exemplos como:

```
CREATE TABLE COMPONENTS (
       CompId CHAR (8) NOT NULL,
       CStock NUMERIC (10) NOT NULL
       );
```

Em VDM-SL

```
types
        COMPONENTS = set of COMPONENTS_RECORD;
        COMPONENTS_RECORD ::
                CompId: seq of char
                CStock: int;
```

onde

- `set of A` = colecções (conjuntos) de `A`
- `seq of A` = sequências (listas) de `A`
- `C::sel1:A sel2:B` = pares (*struct's*) `A`, `B`

Invariantes VDM-SL

De facto:

```
COMPONENTS = set of COMPONENTS_RECORD;

COMPONENTS_RECORD ::
            CompId: seq of char
            CStock: int
      inv r == len r.CompId  <= 8  and
            abs(r.CStock) < 10**10;
```

onde . . .

Invariantes VDM-SL (cont.)

`inv r == ...`

propriedade invariante (inalienável) do tipo respectivo.

`len : seq of A -> nat`

comprimento de sequências

`abs: int -> nat`

valor absoluto de um inteiro

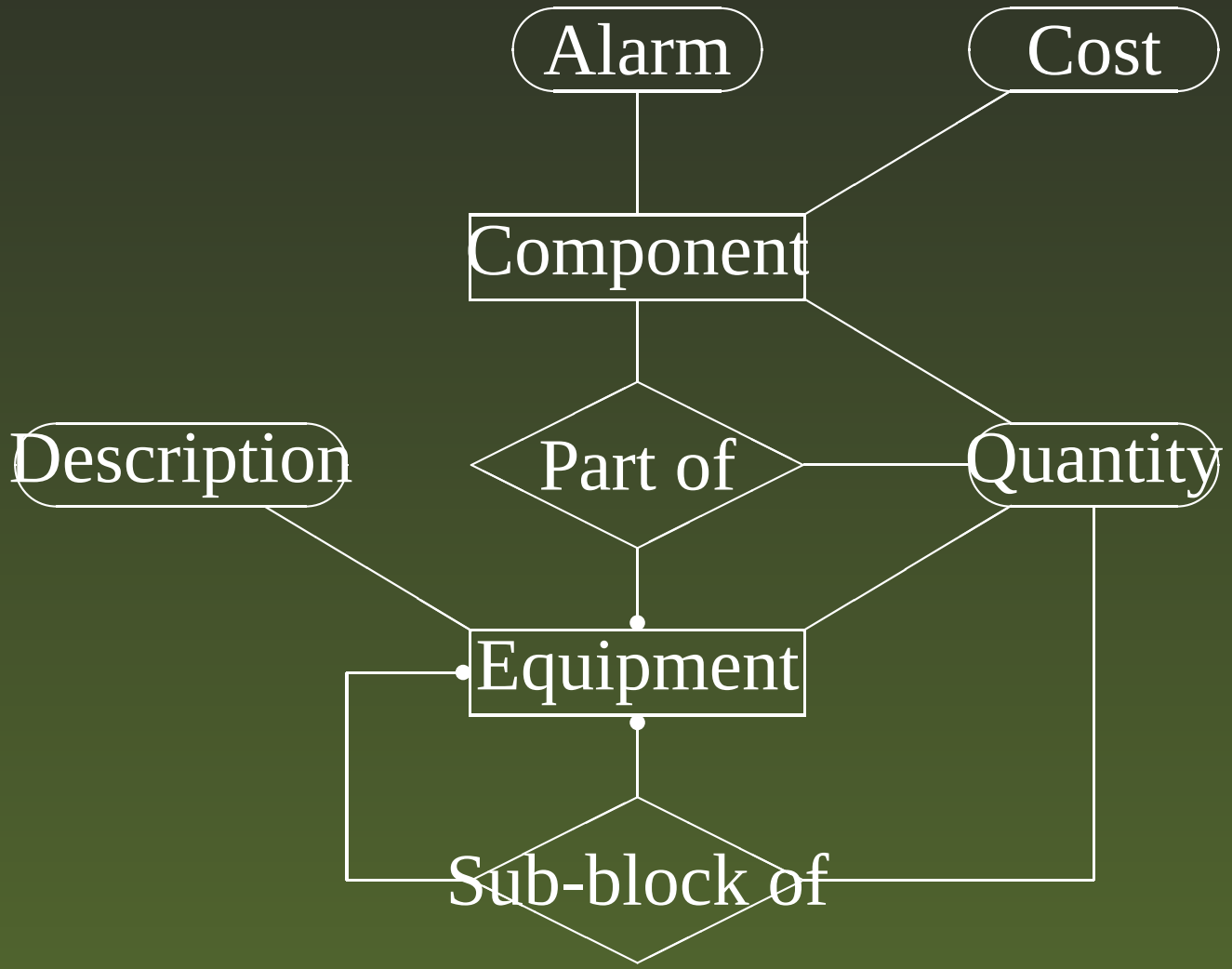
etc.

Invariantes VDM-SL (cont.)

Um **único** mecanismo para:

- Exprimir **tipificação** (‘type checking’) de forma rigorosa
- Exprimir **integridade** referencial
- Exprimir **regras de negócio** *arbitrariamente complexas*

Um exemplo clássico (ER)



Entidade “Component” (SQL)

```
CREATE TABLE COMPONENTS (
       CompId CHAR   (8)   NOT NULL,
       CStock NUMERIC (10)  NOT NULL,
       Alarm NUMERIC  (10)  NOT NULL,
       Cost NUMERIC   (6,3) NOT NULL
       );
```

Qual a semântica formal de `NUMERIC (6,3)`?

`NUMERIC` em SQL standard

`NUMERIC(p,q)` = decimal number, p digits and sign, with assumed decimal point q digits from the right ($0 \leq q \leq p \wedge p > 0$).

cf

A Guide to the SQL Standard, by C.J. Date with Hugh Darwen, Addison-Wesley, 1997.

Logo

```
Attribute NUMERIC (p,q) NOT NULL
```

significa

```
Attribute = real
inv a == abs(a * 10**q) < 10**p;
```

Entidade “Component” (VDM-SL)

```
COMPONENTS = set of COMPONENTS_RECORD;
COMPONENTS_RECORD ::
            CompId: seq of char
            CStock: int
            Alarm:  int
            Cost:   real
   inv r == len r.CompId  <= 8     and
            abs(r.CStock) < 10**10 and
            abs(r.Alarm)  < 10**10 and
            abs(r.Cost*1000) < 10**6;
```

Unicidade referencial (SQL)

Mas desenho acaba por impor `CompId` como “chave”:

```
CREATE TABLE COMPONENTS (
        CompId CHAR   (8)   NOT NULL,
        CStock NUMERIC (10)  NOT NULL,
        Alarm NUMERIC  (10)  NOT NULL,
        Cost NUMERIC   (6,3) NOT NULL
        CONSTRAINT COMPONENTS_pk
                   PRIMARY KEY(CompId)
         );
```

Unicidade referencial (VDM-SL)

```
COMPONENTS_pk :: CompId: seq of char
        inv k == len k.CompId  <= 8;

COMPONENTS_RECORD ::
          CStock: int
          Alarm:  int
          Cost:   real
      inv r == abs(r.CStock) < 10**10 and
               abs(r.Alarm)  < 10**10 and
               abs(r.Cost*1000) < 10**6;
```

Modelação com “mappings”

```
COMPONENTS = map COMPONENTS_pk
             to  COMPONENTS_RECORD;
```

O que significa `map A to B`?

Exemplo simples:

Freguesia	Eleitores
Cachadas, S. Miguel	923
Corgo, S. Maria	722
Olivais, S. Pedro	517

Significado de “map A to B”

Como “Freguesia” **determina** “Eleitores”, podemos

- definir

```
Inscritos = map Freguesia to Eleitores
```

- representar a tabela acima como um “mapping”:

```
{
      "Cachadas, S. Miguel" |-> 923,
      "Corgo, S. Maria"     |-> 722,
      "Olivais, S. Pedro"   |-> 517
}
```

Utilidade de “map A to B”

‘Mappings’ = **funções** (finitas) :

- A e B são **quaisquer** tipos de dados, inc. outros ‘mappings’, *e.g.*

```
map Freguesia to (map Ano to Eleitores)
```

- Linguagem de ‘mappings’: muito expressiva

Aplicações em modelação:

- **Identificação** única de objectos
- **Dependências** funcionais
- Integridade **referencial**

Integridade referencial (SQL)

```
CREATE TABLE EQUIPMENTS (
       EqId CHAR (8) NOT NULL,
       Description CHAR (73) NOT NULL,
       EStock NUMERIC (10) NOT NULL,
       CONSTRAINT EQUIPMENTS_pk PRIMARY KEY (EqId)
       );

CREATE TABLE PART_OF (
       Comp CHAR (8) NOT NULL,
       Equip CHAR (8) NOT NULL,
       HowManyC NUMERIC (10) NOT NULL,
       CONSTRAINT PART_OF_pk PRIMARY KEY (Comp,Equip)
       );

ALTER TABLE PART_OF ADD CONSTRAINT PART_OF_fk1
      FOREIGN KEY (Comp) REFERENCES COMPONENTS(CompId);
```

Integridade referencial (VDM-SL)

Para além de `COMPONENTS`:

```
EQUIPMENTS = map EQUIPMENTS_pk
             to EQUIPMENTS_RECORD;
```

onde

```
EQUIPMENTS_pk :: EqId: seq of char;

EQUIPMENTS_RECORD ::
          Description: seq of char
          EStock: int;
```

(omitem-se os invariantes, para simplificar)

Integridade referencial (VDM-SL)

Relacionamento:

```
PART_OF = map PART_OF_pk to PART_OF_RECORD;

PART_OF_pk ::
        Comp: seq of char
        Equip: seq of char;

PART_OF_RECORD :: HowManyC: int;
```

Integridade referencial (VDM-SL)

Quanto ao significado das restrições

```
ALTER TABLE PART_OF ADD CONSTRAINT
      PART_OF_fk1 FOREIGN KEY (Comp)
      REFERENCES COMPONENTS(CompId);

ALTER TABLE PART_OF ADD CONSTRAINT
      PART_OF_fk2 FOREIGN KEY (Equip)
      REFERENCES EQUIPMENTS(EqId);
```

vão traduzir-se sob a forma de um **invariante** relacionado as respectivas estruturas:

Integridade referencial (VDM-SL)

```
DATABASE :: Comps:  COMPONENTS
            Equips: EQUIPMENTS
            PartOf: PART_OF
  inv db == let compsTb = db.Comps,
                equipsTb = db.Equips,
                partsof = db.PartOf
            in part_ok_fk1(partsof,compsTb) and
               part_ok_fk2(partsof,equipsTb);
```

Notação: `let a = ....  in ...a...`

Predicados auxiliares

```
functions

part_ok_fk1: PART_OF * COMPONENTS -> bool
part_ok_fk1(partsof,compsTb) ==
      forall k in set dom partsof &
            k.Comp in set allCompIds(compsTb);

part_ok_fk2: PART_OF * EQUIPMENTS -> bool
part_ok_fk2(partsof,equipsTb) ==
      forall k in set dom partsof &
            k.Equip in set allEquipIds(equipsTb);
```

Operadores sobre “map A to B”

`dom : map A to B -> set of A`

conjunto de todas as chaves (vulg. **domínio**)

`rng : map A to B -> set of B`

conjunto de toda a informação dependente (vulg. **contra-domínio**)

etc.

Funções auxiliares

```
allCompIds : COMPONENTS -> set of seq of char
allCompIds(c) == { k.CompId | k in set dom c};

allEquipIds : EQUIPMENTS -> set of seq of char
allEquipIds(c) == { k.EqId | k in set dom c};
```

Invariantes arbitrários

Um caso célebre, o tipo **Data**:

```
Date :: year: nat1
        month: nat1
        day: nat1
    inv mk_Date(y,m,d) == dateOk(y,m,d);
```

Notação:

- Tipo `nat1 = { 1, 2, 3, ... }`
- Construtor `mk_Date` e ‘pattern matching’

Um invariante complicado

"**Do Ano e Sua Divisão** — (...) Júlio César instituiu o ano, de que hoje usamos, de 365 dias e 6 horas, a qual quantidade não é exacta, pois vemos claramente adiantar-se o tempo; (...) a Santa Madre Igreja usa do ano que instituiu Júlio César, tomando em cada ano as 6 horas, que formam um dia inteiro em cada quatro anos, chamando-se bissexto a esse ano, a que se acrescenta um dia (...).

Da Reforma do Calendário — Tendo-se observado, que desde a celebração do concílio de Niceia, em 325, até ao ano de 1582, se haviam antecipado os equinócios 10 dias do assento fixo em que os colocara Dionísio Romano; (...) mandou o papa Gregório XIII proceder à reforma do Calendário, em virtude da qual se determinou: 1.º que no mês de Outubro de 1582 se suprimissem 10 dias, contando 4 no dia de S.Francisco, e 15 no seguinte; 2.º que em cada 400 anos se suprimissem 3 dias, principiando de 1700, 1800, 1900, 2100, 2200, 2300, 2500, *etc.* (que por isso não são bissextos), para diminuir o excesso do ano sinodal ao civil, e os equinócios ficarem imóveis a 21 de Março e 23 de Setembro (...)."

Assim...

```
dateOk : nat1 * nat1 * nat1 -> bool
dateOk(y,m,d) ==
     if m in set {1,3,5,7,8,10,12} then
          d <= 31 and
          (not (y=1582 and m=10) or (d<5) or (14<d))
     else if m in set {4,6,9,11} then d <= 30
     else if m=2 and leapYear(y) then d <= 29
     else if m=2 and not leapYear(y) then d <= 28
     else false;

leapYear : nat1 -> bool
leapYear(y) ==
     0 = rem(y, if 1700 <= y and rem(y,100)=0
                then 400 else 4);
```

Modelação de dados relacionais (I)

```
Database = map FileName to Table;
Table    = set of Tuple ;
Tuple    = map Attribute to Value;
FileName = String;
Attribute = String;
Value = String;
```

Modelação de dados relacionais (II)

```
Database = map FileName to Table;
Table    = map Tuple  to Tuple ;
Tuple    = map Attribute to Value;
FileName = String;
Attribute = String;
Value = String;
```