FUNDP
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 Namur
Belgique

EVOLUTION D'APPLICATIONS DE BASES DE DONNÉES RELATIONNELLES
ANNEXES

Jean-Marc Hick

Thèse soumise en vue de l'obtention du grade
de Docteur en Sciences (orientation Informatique)

Jury : Professeur Jean Fichefet, Institut d'informatique (Président)
Professeur Jean-Luc Hainaut, Institut d'informatique (Promoteur)
Professeur Naji Habra, Institut d'informatique
Professeur Jacques Kouloumdjian, INSA de Lyon
Professeur Esteban Zimányi, Université Libre de Bruxelles

26 septembre 2001

TABLE DES MATIÈRES

LISTE DES TABLEAUX

ANNEXE A

NOTATIONS ET SYNTAXES PRÉLIMINAIRES

A.1 Notations

A.1.1 Introduction

Pour décrire formellement les préconditions, les postconditions et la modification des instances des transformations analysées dans les annexes B et C, des notations prédicatives sont utilisées. Cette section donne une description de toutes les notations utilisées. Elles sont classées par type d'objets. Les deux dernières rubriques reprennent les notations communes à plusieurs types d'objet et celles concernant les instances des objets.

A.1.2 Définition des notations

Type d'entités

type-entites(S)	Ensemble des types d'entités du schéma S.
sous-type(e)	Ensemble des sous-types du type d'entités e.
surtype(e)	Ensemble des surtypes du type d'entités e.
type(sous-type(e))	Type de l'ensemble des sous-types de e Les types possibles sont : partition, disjoint, total ou sans type.

Type d'associations

type-associations(S)	Ensemble des types d'associations du schéma S.

Attribut

attribut(o)	Ensemble des attributs de l'objet o (soit un type d'entités, soit un type d'associations, soit un attribut décomposable).
longueur(a)	Longueur de l'attribut a.
type(a)	Domaine de valeurs de l'attribut a. Les types possibles sont : car (caractère), num (numérique), reel, date et bool (booléen).
type-set(a)	Catégorie de collections de valeurs de l'attribut multivalué a. Les catégories possibles sont : set (ensemble), bag (amas), list (liste), ulist (liste unique), array (tableau) et uarray (tableau unique).

Rôle

role(o)	Ensemble des rôles de l'objet o (soit un type d'associations soit un type d'entités).
te-role(ro)	Ensemble des types d'entités jouant le rôle ro.

Groupe

groupe(o)	Ensemble des groupes de l'objet o (soit un type d'associations, soit un type d'entités, soit un attribut multivalué décomposable).

composant(g)	Ensemble des composants du groupe g (soit des attributs, soit des rôles).
identifiant(o)	Ensemble des groupes identifiants de l'objet o (soit un type d'associations, soit un type d'entités, soit un attribut multivalué décomposable).
pid(o)	Groupe identifiant primaire de l'objet o (soit un type d'associations, soit un type d'entités, soit un attribut multivalué décomposable).
cle-acces(e)	Ensemble des groupes définissant une clé d'accès sur le type d'entités e.
reference(e)	Ensemble des groupes contenant une contrainte référentielle du type d'entités e.
est-reference(g)	Ensemble des groupes ayant une contrainte référentielle qui référence le groupe g.
est-ref-egal(g)	Ensemble des groupes ayant une contrainte référentielle d'égalité qui référence le groupe g.
fonction(g)	Fonctions du groupe g. Aucune ou plusieurs fonctions sont autorisées. Les fonctions possibles sont : id-primaire, id-secondaire, exactement-1, cœxistence, exclusif, au-moins-1, cle-acces et egal (contrainte d'égalité).

Collection

collection(S)	Ensemble des collections du schéma S.
te-coll(c)	Ensemble des types d'entités appartenant à la collection c.

Propriétés communes

nom(o)	Nom de l'objet o (soit un type d'entités, soit un type d'associations, soit un attribut, soit un groupe, soit une collection, soit un rôle).
card(o) = [i-j]	Les cardinalités de l'objet o (soit un attribut, soit un rôle, soit un groupe). i est la cardinalité minimum et j la cardinalité maximum. Les cardinalités respectent les contraintes : $0 \leq i \leq j \leq N$ et $j > 0$ où N représente l'infini.
card-min(o)	La cardinalité minimum de l'objet o (soit un attribut, soit un rôle, soit un groupe).
card-max(o)	La cardinalité maximum de l'objet o (soit un attribut, soit un rôle, soit un groupe).
#(e)	Le nombre d'éléments d'un ensemble e.

Instances

instance(o)	Ensemble des instances de o (où o représente n'importe quel type d'objets du modèle de spécification).
supprimer(i)	Fonction de suppression d'une instance i.
valeur(i)	La valeur d'une instance i (valeur unique ou un tuple).
i[j]	Le j^{me} élément de l'instance tuple i.
i[o]	La partie de l'instance tuple i concernant l'objet o.
desagreger(i,i1,...,in)	Fonction de désagrégation de l'instance i dans les instances i1, ..., 1n.
agreger(i1,...,in,i)	Fonction d'agrégation des instances i1, ..., in dans l'instance i.
valeur-def	La valeur par défaut d'une instance d'attributs. Elle change suivant le type de l'attribut.
null	Désignation de l'absence de valeur.

A.2 Grammaire SQL-92

A.2.1 Introduction

Cette section propose une représentation de la syntaxe du langage SQL sous la forme d'une grammaire BNF (Backus-Naur Form) comme elle est présentée dans le texte de la norme SQL-92 [SQL92]. Bien qu'incomplète (la norme officielle fait plus de 600 pages), la grammaire est suffisante pour définir les instructions et requêtes SQL utilisées dans ce travail. On a adopté la présentation de Date et Darwen [Dat94] qui divise la grammaire en dix parties :

- les sessions, les connexions et les transactions (point A.2.2);
- la définition des données (point A.2.3) : les créations (`CREATE`), les modifications (`ALTER`) et les suppressions (`DROP`) de structures de données;
- les modules (point A.2.4);
- la manipulation des données (point A.2.5) : les sélections (`SELECT`), les insertions (`INSERT`), les modifications (`UPDATE`) et les destructions (`DELETE`) de données ainsi que les curseurs;
- les expressions de tables (point A.2.6) : contenu des clauses `FROM`;
- les expressions conditionnelles (point A.2.7) : contenu des clauses `WHERE` et `HAVING`;
- les contraintes (point A.2.8) : les clés primaires (`PRIMARY KEY`), les clés candidates (`UNIQUE`), les clés étrangères (`FOREIGN KEY`) et les autres contraintes (`CHECK`);
- SQL dynamique (point A.2.9);
- les expressions scalaires (point A.2.10);
- divers (point A.2.11).

A.2.2 Sessions, connexions et transactions

```
connect
    ::=   CONNECT TO { DEFAULT
                     | lit-param-or-var [ AS lit-param-or-var ]
                                        [ USER lit-param-or-var ] }

set-connection
    ::=   SET CONNECTION { DEFAULT | lit-param or var

disconnect
    ::=   DISCONNECT { DEFAULT | CURRENT | ALL | lit-param or var )

set-catalog
    ::=   SET CATALOG { lit-param or var | user-function-ref }

set-schema
    ::=   SET SCHEMA { lit-param or var | user-function-ref }

set-names
    ::=   SET NAMES { lit-param or var | user-function-ref }

set-authorization
    ::=   SET SESSION AUTHORISATION
                    { lit-param-or-var | user-function-ref }

set-time-zone
    ::=   SET TIME ZONE { interval-exp | LOCAL }
```

```
commit
    ::=   COMMIT [ WORK ]

rollback
    ::=   ROLLBACK [ WORK ]

set-transaction
    ::=   SET TRANSACTION
            [ READ ONLY | READ WRITE ]
            [ DIAGNOSTICS SIZE integer ]
            [ ISOLATION LEVEL { READ UNCOMMITTED
                              | READ COMMITTED
                              | REPEATABLE READ
                              | SERIALIZABLE } ]
```

A.2.3 Définition des données

```
schema-def
    ::=   CREATE SCHEMA [ schema ] [ AUTHORISATION user ]
               [ DEFAULT CHARACTER SET character-set ]
               [ schema-element-list ]

schema-element
    ::=   domain-def
        | base-table-def
        | view-def
        | authorization-def
        | general-constraint-def
        | character-set-def
        | collation-def
        | translation-def

domain-def
    ::=   CREATE DOMAIN domain [ AS ] data-type
               [ default-def ]
               [ domain-constraint-def ]

default-def
    ::=   DEFAULT { literal | niladic-function-ref | NULL }

base-table-def
    ::=   CREATE [ [ GLOBAL | LOCAL ] TEMPORARY ] TABLE base-table
                ( base-table-element-commalist )
                [ ON COMMIT { DELETE | PRESERVE } ROWS ]

base-table-element
    ::=   column-def | base-table-constraint-def

column-def
    ::=   column { data-type | domain }
                [ default-def ]
                [ column-constraint-def-list ]

view-def
    ::=   CREATE VIEW view [ ( column-commalist ) ]
              AS table-exp
                    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]

authorization-def
    ::=   GRANT { privilege-commalist | ALL PRIVILEGES }
                  ON accessible-object TO grantee-commalist
                                       [ WITH GRANT OPTION ]
```

```
general-constraint-def
    ::=   CREATE ASSERTION constraint CHECK ( cond-exp )
                                      [ deferrability ]

deferrability
    ::=   INITIALLY { DEFERRED | IMMEDIATE }
          [ NOT ] DEFERRABLE

privilege
    ::=   SELECT
        | INSERT [ ( column-commalist ) ]
        | UPDATE [ ( column-commalist ) ]
        | DELETE
        | REFERENCES [ ( column-commalist ) ]
        | USAGE

accessible-object
    ::=   DOMAIN domain
        | [ TABLE ] table
        | CHARACTER SET character-set
        | COLLATION collation
        | TRANSLATION translation

grantee
    ::=   user | PUBLIC

character-set-def
    ::=   CREATE CHARACTER SET character-set [ AS ]
             GET character-set
           [ COLLATE collation | COLLATION FROM collation-source ]

collation-source
    ::=   EXTERNAL ( 'collation' )
        | collation
        | DESC ( collation )
        | DEFAULT
        | TRANSLATION translation [ THEN COLLATION collation ]

collation-def
    ::=   CREATE COLLATION collation
                  FOR character-set
                  FROM collation-source

translation-def
    ::=   CREATE TRANSLATION translation
                  FOR character-set
                  TO character-set
                  FROM translation-source

translation-source
    ::=   EXTERNAL ( 'translation' )
        | IDENTITY
        | translation

domain-alteration
    ::=   ALTER DOMAIN domain domain-alteration-action

domain-alteration-action
    ::=   domain-default-alteration-action
        | domain-constraint-alteration-action

domain-default-alteration-action
    ::=   SET default-def
```

```
        | DROP DEFAULT

domain-constraint-alteration-action
    ::=   ADD domain-constraint-def
        | DROP CONSTRAINT constraint

base-table-alteration
    ::=   ALTER TABLE base-table base-table-alteration-action

base-table-alteration-action
    ::=   column-alteration-action
        | base-table-constraint-alteration-action

column-alteration-action
    ::=   ADD [ COLUMN ] column-def
        | ALTER [ COLUMN ] column
                { SET default-def | DROP DEFAULT }
        | DROP [ COLUMN ] column { RESTRICT | CASCADE }

base-table-constraint-alteration-action
    ::=   ADD base-table-constraint-def
        | DROP CONSTRAINT constraint { RESTRICT | CASCADE }

schema-drop
    ::=   DROP SCHEMA schema { RESTRICT | CASCADE }

domain-drop
    ::=   DROP DOMAIN domain { RESTRICT | CASCADE }

base-table-drop
    ::=   DROP TABLE base-table { RESTRICT | CASCADE }

view-drop
    ::=   DROP VIEW view { RESTRICT | CASCADE }

authorization-drop
    ::=   REVOKE [ GRANT OPTION FOR ] priviloge-commalist
                  ON accessible-object FROM grantee-commalist
                                       { RESTRICT | CASCADE }

general-constraint-drop
    ::=   DROP ASSERTION constraint

character-set-drop
    ::=   DROP CHARACTER SET character-set

collation-drop
    ::=   DROP COLLATION collation

translation-drop
    ::=   DROP TRANSLATION translation
```

A.2.4 Modules

```
module-def
    ::=   MODULE [ module ] [ NAMES ARE character-set ]
          LANGUAGE { ADA | C | COBOL | FORTRAN | MUMPS
                   | PASCAL | PLI }
        [ SCHEMA schema ] [ AUTHORISATION user ]
        [ temporary-table-def-list ]
          module-element-list

temporary-table-def
```

```
    ::=   DECLARE LOCAL TEMPORARY TABLE MODULE . base-table
                ( base-table-element-commalist )
                [ ON COMMIT { PRESERVE | DELETE } ROWS ]

module-element
    ::=   cursor-def
        | dynamic-cursor-def
        | procedure-def

procedure-def
    ::=   PROCEDURE procedure
           { parameter-def-list | ( parameter-def-commalist ) } ;
             SQL-statement ;

Note: "SQL-statement" est une séquence d’instructions SQL ou d’un langage spécifique.

parameter-def
    ::=   parameter data-type
        | SQLCODE
        | SQLSTATE
```

A.2.5 Manipulation des données

```
single-row-select
    ::=   SELECT [ ALL | DISTINCT ] select-item-commalist
          INTO target-commalist
          FROM table-ref-commalist
        [ WHERE cond-exp ]
        [ GROUP BY column-ref-commalist ]
        [ HAVING cond-exp ]

insert
    ::=   INSERT INTO table
          { [ ( column-commalist ) ] table-exp | DEFAULT VALUES }

searched-update
    ::=   UPDATE table
          SET update-assignment-commalist
        [ WHERE cond-exp ]

update-assignment
    ::=   column = { scalar-exp | DEFAULT | NULL }

searched-delete
    ::=   DELETE
          FROM table
        [ WHERE cond-exp ]

cursor-def
    ::=   DECLARE cursor [ INSENSITIVE ] [ SCROLL ] CURSOR FOR
                  table-exp
                [ ORDER BY order-item-commalist ]
          [ FOR { READ ONLY | UPDATE [ OF column-commalist ] } ]

order-item
    ::=   { column | integer } [ ASC | DESC ]

open
    ::=   OPEN cursor

fetch
    ::=   FETCH [ [ row-selector ] FROM ] cursor
                                  INTO target-commalist
```

```
row-selector
    ::=   NEXT | PRIOR | FIRST | LAST
                | ABSOLUTE number | RELATIVE number

positioned-update
    ::=   UPDATE table
          SET update-assignment-commalist
          WHERE CURRENT OF cursor

positioned-delete
    ::=   DELETE
          FROM table
          WHERE CURRENT OF cursor

close
    ::=   CLOSE cursor
```

A.2.6 Expressions de tables

```
table-exp
    ::=   join-table-exp | nonjoin-table-exp

join-table-exp
    ::=   table-ref [ NATURAL ] [ join-type ] JOIN table-ref
                  [ ON cond-exp | USING ( column-commalist ) ]
        | table-ref CROSS JOIN table-ref
        | ( join-table-exp )

table-ref
    ::=   table [ [ AS ] range-variable
                [ ( column-commalist ) ] ]
        | ( table-exp ) [ AS ] range-variable
                     [ ( column-commalist ) ]
        | join-table-exp

join-type
    ::=   INNER
        | LEFT [ OUTER ]
        | RIGHT [ OUTER ]
        | FULL [ OUTER ]
        | UNION

nonjoin-table-exp
    ::=   nonjoin-table-term
        | table-exp { UNION | EXCEPT } [ ALL ]
              [ CORRESPONDING [ BY ( column-commalist ) ] ]
                    table-term

nonjoin-table-term
    ::=   nonjoin-table-primary
        | table-term INTERSECT [ ALL ]
              [ CORRESPONDING [ BY ( column-commalist ) ] ]
                    table-primary

table-term
    ::=   nonjoin-table-term
        | join-table-exp

table-primary
    :=   nonjoin-table-primary
       | join-table-exp
```

```
nonjoin-table-primary
    ::=   TABLE table
        | table-constructor
        | select-exp
        | ( nonjoin-table-exp )

table-constructor
    ::=   VALUES row-constructor-commalist

row-constructor
    ::=   scalar-exp | ( scalar-exp-commalist ) | ( table-exp )

select-exp
    ::=   SELECT [ ALL | DISTINCT ] select-item-commalist
             FROM table-ref-commalist
                [ WHERE cond-exp ]
                   [ GROUP BY colunn-ref-commalist ]
                      [ HAVING cond-exp ]

select-item
    ::=   scalar-exp [ [ AS ] column ]
        | [ range-variable . ] *
```

A.2.7 Expressions conditionnelles

```
cond-exp
    ::=   cond-term
        | cond-exp OR cond-term

cond-term
    ::=   cond-factor
        | cond-term AND cond-factor

cond-factor
    ::=   [ NOT ] cond-test

cond-test
    ::=   cond-primary
             [ IS [ NOT ] ( TRUE | FALSE | UNKNOWN } ]

cond-primary
    ::=   simple-cond | ( cond-exp )

simple-cond
    ::=   comparison-cond
        | between-cond
        | like-cond
        | in-cond
        | match-cond
        | all-or-any-cond
        | exists-cond
        | unique-cond
        | overlaps-cond
        | test-for-null

comparison-cond
    ::=   row-constructor comparison-operator row-constructor

comparison-operator
    ::=   = | < | <= | > | >= | <>

between-cond
    ::=   row-constructor [ NOT ] BETWEEN row-constructor
```

```
                                    AND row-constructor

like-cond
    ::=   character-string-exp
          [ NOT ] LIKE character-string-exp
                [ ESCAPE character-string-exp ]

in-cond
    ::=   row-constructor [ NOT ] IN ( table-exp )
        | scalar-exp [ NOT ] IN ( scalar-exp-commalist )

match-cond
    ::=   row-constructor MATCH [ UNIQUE ]
                        [ PARTIAL | FULL ] ( table-exp )

all-or-any-cond
    ::=   row-constructor
              comparison-operator { ALL | ANY | SOME }
                                      ( table-exp )

exists-cond
    ::=   EXISTS ( table-exp )

unique-cond
    ::=   UNIQUE ( table-exp )

overlaps-cond
    ::=   ( scalar-exp, scalar-exp )
               OVERLAPS ( scalar-exp, scalar-exp )

test-for-null
    ::=   row-constructor IS [ NOT ] NULL
```

A.2.8 Contraintes

```
domain-constraint-def
    ::=   [ CONSTRAINT constraint ] CHECK ( cond-exp )
                                         [ deferrability ]

base-table-constraint-def
    ::=   [ CONSTRAINT constraint ]
            candidate-key-def [ deferrability ]
        | [ CONSTRAINT constraint ]
            foreign-key-def [ deferrability ]
        | [ CONSTRAINT constraint ]
            check-constraint-def [ deferrability ]

candidate-key-def
    ::=   { PRIMARY KEY | UNIQUE } ( column-commalist )

foreign-key-def
    ::=   FOREIGN KEY ( column-commalist ) references-def

references-def
    ::=   REFERENCES base-table [ ( column-commalist ) ]
             [ MATCH { FULL | PARTIAL } ;
             [ ON DELETE referential-action ]
             [ ON UPDATE referential-action ]

referential-action
    ::=   NO ACTION | CASCADE | SET DEFAULT | SET NULL

check-constraint-def
```

```
    ::=   CHECK ( cond-exp )

column-constraint-def
    ::=   [ CONSTRAINT constraint ]
            NOT NULL [ deferrability ]
        | [ CONSTRAINT constraint ]
            { PRIMARY KEY | UNIQUE } [ deferrability ]
        | [ CONSTRAINT constraint ]
            reterences-def [ deferrability ]
        | [ CONSTRAINT constraint ]
            CHECK ( cond-exp ) [ deferrability ]

set-constraints
    ::=   SET CONSTRAINTS { constraint-commalist | ALL }
                                    { DEFERRED | IMMEDIATE }
```

A.2.9 SQL dynamique

```
execute-immediate
    ::=   EXECUTE IMMEDIATE param-or-var

prepare
    ::=   PREPARE prepared FROM param-or-var

prepared
    ::=   prepped-statement-container I param-or-var

deallocate-prepare
    ::=   DEALLOCATE PREPARE prepared

execute
    ::=   EXECUTE prepared [ INTO places ] [ USING arguments ]

places
    ::=   target-commalist | SQL DESCRIPTOR descriptor

arguments
    ::=   target-commalist | SQL DESCRIPTOR descriptor

descriptor
    ::=   lit-param-or-var

allocate-descriptor
    ::=   ALLOCATE DESCRIPTOR descriptor
                  [ WITH MAX lit-param-or-var ]

deallocate-descriptor
    ::=   DEALLOCATE DESCRIPTOR descriptor

describe-input
    ::=   DESCRIBE INPUT prepared
                  USING SQL DESCRIPTOR descriptor

describe-output
    ::=   DESCRIBE [ OUTPUT ] prepared
                  USING SQL DESCRIPTOR descriptor

get-descriptor-1
    ::=   GET DESCRIPTOR descriptor target = COUNT

get-descriptor-2
    ::=   GET DESCRIPTOR descriptor
              VALUE number get-assignment-commalist
```

```
get-assignment
    ::=   target = item-descriptor-element

set-descriptor-1
    ::=   SET DESCRIPTOR descriptor COUNT = lit-param-or-var

set-descriptor 2
    ::=   SET DESCRIPTOR descriptor
              VALUE number set-assignment-commalist

set-assiqnment
    ::=   item-descriptor-element = lit-param-or-var

dynamic-cursor-def
    ::=   DECLARE cursor [ INSENSITIVE ] [ SCROLL ] CURSOR
          FOR prepared

allocate-cursor
    ::=   ALLOCATE param-or-var [ INSENSITIVE ] [ SCROLL ] CURSOR
          FOR prepared

dynamic-open
    ::=   OPEN dynamic-cursor [ USING arguments ]

dynamic-cursor
    ::=   cursor | param-or-var

dynamic-close
    ::=   CLOSE dynamic-cursor

dynamic-fetch
    ::=   FETCH [ [ row-selector ] FROM ] dynamic-cursor
                INTO places

dynamic-positioned-delete
    ::=   DELETE [ FROM table ]
          WHERE CURRENT OF dynamic-cursor

dynamic-positioned-update
    ::=   UPDATE [ table ] SET assignment-commalist
          WHERE CURRENT OF dynamic-cursor
```

A.2.10 Expressions scalaires

```
scalar-exp
    ::=   numeric-exp
        | character-string-exp
        | bit-string-exp
        | datetime-exp
        | interval-exp

numeric-exp
    ::=   numeric-term
        | numeric-exp { + | - } numeric-term

numeric-term
    ::=   numeric-factor
        | numeric-term { * | / } numeric-factor

numeric-factor
    ::=   [ + | - ] numeric-primary
```

```
numeric-primary
    ::=   column-ref
        | lit-param-or-var
        | scalar-function-ref
        | aggregate-function-ref
        | ( table-exp )
        | ( numeric-exp )

aggregate-function-ref
    ::=   COUNT(*)
        | { AVG | MAX | MIN | SUM | COUNT }
                ( [ ALL | DISTINCT ] scalar-exp )

character-string-exp
    ::=   character-string-concatenation
        | character-string-primary

character-string-concatenation
    ::=   character-string-exp || character-string-primary

Note: Le symbole "||" représente ici l'opérateur de concatenation - il ne doit
      pas être confondu avec la barre verticale "|" qui est utilisée pour séparer
      les alternatives dans la grammaire.

character-string-primary
    ::=   column-ref
        | lit-param-or-var
        | user-function-ref
        | scalar-function-ref
        | aggregate-function-ref
        | ( table-exp )
        | ( character-string-exp )

bit-string-exp
    ::=   bit-string-concatenation
        | bit-string-primary

bit-string-concatenation
    ::=   bit-string-exp || bit-string-primary

Note: Le symbole "||" représente ici l'opérateur de concatenation - il ne doit
      pas être confondu avec la barre verticale "|" qui est utilisée pour séparer
      les alternatives dans la grammaire.

bit-string-primary
    ::=   column-ref
        | lit-param-or-var
        | scalar-function-ref
        | aggregate-function-ref
        | ( table-exp )
        | ( bit-string-exp )

datetime-exp
    ::=   datetime-term
        | interval-exp + datetime-term
        | datetime-exp { + I - } interval-term

datetime-term
    ::=   datetime-primary
                [ AT { LOCAL | TIME ZONE interval-exp } ]

datetime-primary
    ::=   column-ref
        | lit-param-or-var
```

```
        | datetime-function-ref
        | scalar-function-ref
        | aggregate-function-ref
        | ( table-exp )
        | ( datetime-exp )

interval-exp
    ::=   interval-term
        | interval-exp { + | - } interval-term
        | ( datetime exp - datetime-term ) start [ TO end ]

Note: Les valeurs "start" et "end" sont ici des champs datetime.

interval-term
    ::=   interval-factor
        | interval-term { * | / } numeric-factor
        | numeric-term * interval-factor

interval-factor
    ::=   [ + | - ] interval-primary [ start [ TO end ] ]

Note: Les valeurs "start" et "end" sont ici des champs datetime, et apparaissent
      seulement dans un certain contexte SQL dynamique.

interval-primary
    ::=   column-ref
        | lit-param-or-var
        | scalar-function-ref
        | aggregate-function-ref
        | ( table-exp )
        | ( interval-exp )
```

A.2.11 Divers

```
schema
    ::=   [ catalog . ] identifier

domain
    ::=   [ schema . ] identifier

table
    ::=   base-table | view

base-table
    ::=   [ schema . ] identifier

view
    ::=   [ schema . ] identifier

constraint
    ::=   [ schema . ] identifier

character-set
    ::=   [ schema . ] identifier

collation
    ::=   [ schema . ] identifier

translation
    ::=   [ schema . ] identifier

conversion
    ::=   [ schema . ] identifier
```

```
column-ref
    ::=   [ column-qualifier . ] column

column-qualifier
    ::=   table | range-variable

param-or-var
    ::=   parameter [ [ INDICATOR ] parameter ]
        | host-variable [ [ INDICATOR ] host-variable ]

lit-param-or-var
    ::=   literal | param-or-var

target
    ::=   param-or-var

number
    ::=   lit-param-or-var

niladic-function-ref
    ::=   user-function-ref
        | datetime-function-ref

user-fanction-ref
    ::=   USER
        | CURRENT_USER
        | SESSION USER
        | SYSTEM USER

datetime-function-ref
    ::=   CURRENT DATE
        | CURRENT TIME [ ( integer ] ]
        | CURRENT TIMESTAMP [ ( integer ) ]
```

ANNEXE B

TRANSFORMATIONS À SÉMANTIQUE CONSTANTE

B.1 Introduction

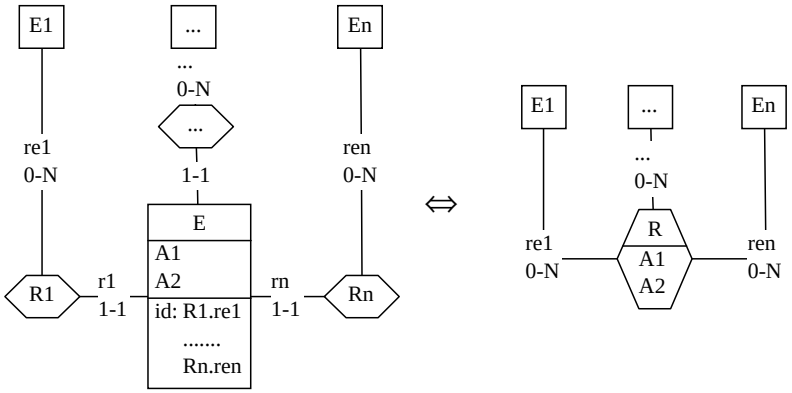
La présentation de chacune des transformations consiste en une brève description (D), une signature (S), les préconditions (P), les postconditions (Q), la transformation des instances (I) et la signature de la transformation inverse (T^{-1}). La notation prédicative (définie au point A.1) est utilisée pour la description des préconditions, des postconditions et des transformations des instances.

Bien que ces analyses ne soient pas complètes, elles sont toutefois suffisantes pour la compréhension de l'approche proposée. Dans [Hai96a] et [Hic00a], les lecteurs intéressés trouveront une analyse plus détaillée comprenant la preuve de l’équivalence de certaines transformations [Hai96a].

B.2 Transformations de types d'entités

B.2.1 Transformation d'un type d'entités en un type d'associations

D Un type d'entités E peut être transformé en un type d'associations R à condition qu'il ait un identifiant et qu'il soit au moins lié à deux autres types d'entités par des types d'associations binaires un à plusieurs.



S $R \leftarrow$ TE-en-TA(E)

P
- $E \in$ type-entites(S)
- sous-type(E) = $\emptyset \wedge$ surtype(E) = $\emptyset$
- $\forall C \in$ collection(S) : $E \notin$ te-coll(C)
- $\forall ri \in$ role(Ri), $1 \leq i \leq n$: card(ri) = [1-1] $\wedge$ te-role(ri) = {E}
- $\forall Ri \in$ rt(S), $1 \leq i \leq n$: attribut(Ri) = $\emptyset \wedge$ role(Ri) = {ri,rei} $\wedge$ te-role(ri) $\neq$ te-role(rei)
- identifiant(E) $\neq \emptyset$
- $\forall g \in$ groupe(E) : $g \notin$ reference(E) $\wedge$ est-reference(g) = $\emptyset$
- $\forall i \in$ id(E), $\forall a \in$ composant(i) : card[a] = [1-1]

Q
- $R \in$ type-associations(S)
- $\forall g \in$ groupe(E) : $g \in$ groupe(R)
- $\forall a \in$ attribut(E) : $a \in$ attribut(R)
- $\forall g \in$ groupe(Ei), $1 \leq i \leq n$: $rei \in$ composant(g) $\wedge$ $g \notin$ groupe(Ei)
- $ri \notin$ role(Ri)
- $\forall rei \in$ role(Ri) : $rei \in$ role(R)
- $Ri \notin$ type-associations(S)
- $\exists i \in$ id(R) : composant(i) = {rei,...,ren} $\wedge$ $i \notin$ identifiant(R)
- $E \notin$ type-entites(S)

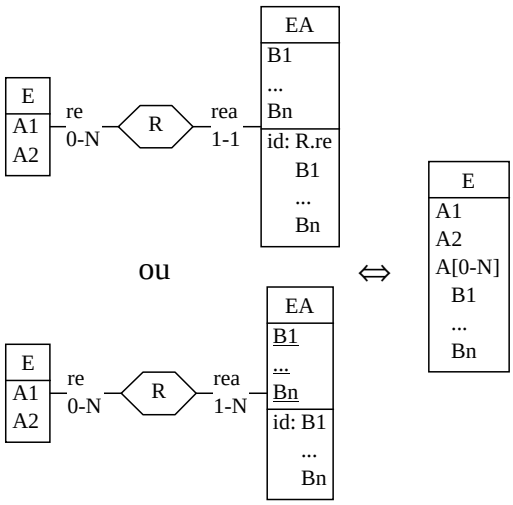
I $\forall e \in$ instance(E), $\exists! r \in$ instance(R) : ($\forall i \in$ [1..n], $\exists ri \in$ instance(Ri) : r[rei] = ri[rei]) $\wedge$ r[A1] = e[A1] $\wedge$ r[A2] = e[A2]

Pour chaque instance de E, on crée une instance de R qui est reliée aux mêmes instances de E1, ..., En et qui a les mêmes valeurs pour les attributs A1 et A2.

T^{-1} (E,{R1,...,Rn}) ← TA-en-TE(R) : transformation d'un type d'associations en un type d'entités (point B.3.1).

B.2.2 Transformation d'un type d'entités en un attribut

D Un type d'entités EA peut être transformé en un attribut A à condition qu'il joue un et un seul rôle.



S (E,A) ← TE-en-Att(EA,R)

P
- EA ∈ type-entites(S)
- R ∈ type-associations(S)
- sous-type(EA) = ∅ ∧ surtype(EA) = ∅
- ∀ C ∈ collection(S) : EA ∉ col-et(C)
- attribut(R) = ∅ ∧ role(R) = {re,rea} ∧ te-role(re) ≠ te-role(rea)
- te-role(re) = {E} ∧ te-role(rea) = {EA}
- card(rea) = [1-j] ∧ j ≥ 1
- attribut(EA) ≠ ∅
- ∀ i ∈ identifiant(EA) : est-reference(i) = ∅ ∧ ¬(card-max(re) = card-max(rea) = 1)
- card-max(r2) > 1 ∧ card(rea) = [1-1] ∧ (∀ i ∈ identifiant(EA), ∃ c ∈ composant(i) ∩ attribut(EA) : {B1,...,Bn} ⊂ composant(i))
- card-max(rea) > 1 ∧ identifiant(EA) ≠ ∅

Q *Premier cas* : #(attribut(EA)) > 1 ∨ (card-max(re) > 1 ∧ #(attribut(EA)) = 1 ∧ card-max(attribut(EA)) > 1)
- A ∈ attribut(E) ∧ attribut(A) ≠ ∅ ∧ nom(A) = nom(EA)
- {B1,...,Bn} ⊂ attribut(A)
- card(A) = card(re)
- card-max(A) > 1 ∧ card(rea) = [1-1] ∧ identifiant(EA) = ∅ ∧ type-set(A) = bag
- ∀ g ∈ E ∧ rea ∈ composant(g) : A ∈ composant(g) ∧ rea ∉ composant(g)

Deuxième cas : #(attribut(EA)) = 1 ∧ (card-max(re) = 1 ∨ card-max(attribut(EA)) = 1)
- B ∈ attribut(E)

– card(B) = [card-min(B) * card-min(re),card-max(B) * card-max(re)]
– card-max(re) > 1 ∧ card(rea) = [1-1] ∧ identifiant(EA) = ∅ ∧ type-set(B) = bag
– ∀ g ∈ E ∧ rea ∈ composant(g) : B ∈ composant(g) ∧ rea ∉ composant(g)

Post-conditions communes aux deux cas :

– ∀ i ∈ identifiant(EA) ∧ card(rea) = [1-1] ∧ card-max(re) = 1 : i ∈ identifiant(E)
– ∀ i ∈ identifiant(EA) ∧ card(rea) = [1-1] ∧ card-max(re) > 1 ∧ re ∈ composant(i) : i ∈ identifiant(A)
– ∀ i ∈ identifiant(EA) ∧ card(rea) = [1-1] ∧ card-max(re) > 1 ∧ #(composant(i)) = 1 : i ∈ identifiant(E)
– ∀ i ∈ identifiant(EA) ∧ card(rea) = [1-1] ∧ card-max(re) > 1 ∧ attribut(EA) = composant(i) : id ∈ identifiant(E) ∧ composant(id) = {A}
– ∀ g ∈ group(EA) ∧ composant(g) ⊂ attribut(EA) : g ∈ group(E)
– EA ∉ type-entites(S) ∧ R ∉ type-associations(S)

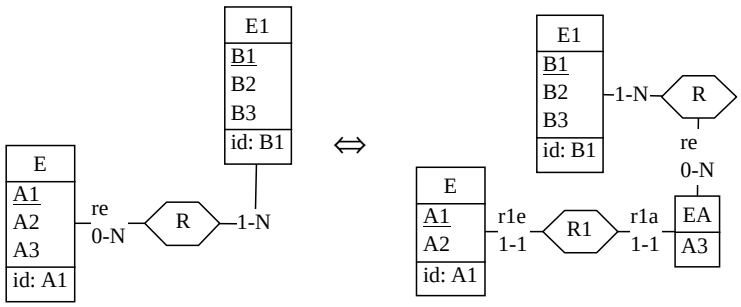
I ∀ eai ∈ instance(E), ri ∈ instance(R), i ∈ [1..N], ∃ e ∈ instance(E) : ri[re,rea] = (e,eai) ∧ e[A[i]] = eai[B1, ...,Bn]

Pour toutes les instances de EA reliées par le biais de R à une instance de E, on crée une instance de A à partir des valeur de B1, ..., Bn appartenant aux instances de EA.

T^{-1} (EA,R) ← Att-en-TE(E.Adesagreger-Att) : transformation d'un attribut en un type d'entités (point B.4.1).

B.2.3 Eclatement d'un type d'entités

D Certains composants (attributs et/ou rôles) du type d'entités E peuvent être extraits et transférés dans un nouveau type d'entités.



S (EA,R1) ← eclater-TE(E,{A3,re})

P – E ∈ type-entites(S)
– attribut(E) ≠ ∅ ∨ role(E) ≠ ∅

Q – EA ∈ type-entites(S)
– R1 ∈ type-associations(S)
– attribut(R1) = ∅ ∧ {r1e,r1a} = role(R1)
– ∀ ro ∈ role(R1) : mono-te(ro) ∧ card(ro) = [1-1]
– A3 ∈ attribut(EA) ∧ re ∈ role(EA)
– ∀ g ∈ groupe(E) ∧ composant(g) ∈ {A3,re} : g ∈ groupe(EA)

I $\forall$ e $\in$ instance(E), $\exists!$ r1 $\in$ instance(R1), ea $\in$ instance(EA) : r1[r1e] = e $\wedge$ r1[r1a] = ea $\wedge$ ea[A3] = e[A3] $\wedge$ ($\forall$ r $\in$ instance(R) : r[re] = ea)

Pour chaque instance de E, on crée une instance de EA qui a les mêmes valeurs pour ses attributs et ses rôles.

T^{-1} E $\leftarrow$ fusionner-TE(EA,R1) :fusion de deux types d'entités (point B.2.4).

B.2.4 Fusion de deux types d'entités

D Les composants (attributs et/ou rôles) du type d'entités EA sont intégrés dans le type d'entités E. Ces deux types d'entités sont reliés par un type d'associations un-à-un. Pour la représentation graphique, il faut se référer à l'éclatement d'un type d'entités au point B.2.3.

S E $\leftarrow$ fusionner-TE(EA,R1)

P
- S $\in$ type-associations(S)
- {EA,E} $\in$ type-entites(S)
- attribut(S) = $\varnothing$ $\wedge$ role(S) = {se,sea} $\wedge$ E = et-role(se) $\wedge$ EA = et-role(sea)
- $\forall$ ro $\in$ role(S) : mono-te(ro) $\wedge$ card(ro) = [1-1]
- sous-type(EA) = $\varnothing$ $\wedge$ surtype(EA) = $\varnothing$
- $\forall$ g $\in$ groupe(E) : sea $\notin$ composant(g)
- $\forall$ g $\in$ groupe(EA) : se $\notin$ composant(g)

Q
- A3 $\in$ attribut(E) $\wedge$ re $\in$ role(E)
- $\forall$ g $\in$ groupe(EA) : g $\in$ groupe(E)
- EA $\notin$ type-entites(S)
- S $\notin$ type-associations(S)

I $\forall$ ea $\in$ instance(EA), $\exists!$ r1 $\in$ instance(R1), e $\in$ instance(E) : r1[r1e] = e $\wedge$ r1[r1a] = ea $\wedge$ e[A3] = ea[A3] $\wedge$ ($\forall$ r $\in$ instance(R) : r[re] = e)

Pour chaque instance de EA reliée à une instance de E, les instances des attributs et des rôles de l'instance de EA sont transférées dans l'instance de E.

T^{-1} (EA,R1) $\leftarrow$ eclater-TE(E,{A3,re}) : éclatement d'un type d'entités (point B.2.3).

B.3 Transformations de types d'associations

B.3.1 Transformation d'un type d'associations en un type d'entités

D Un type d'associations R peut être transformé en un type d'entités et deux types d'associations qui relient E aux types d'entités jouant un rôle dans R. Pour la représentation graphique, il faut se référer à la transformation d'un type d'entités en type d'associations (point B.2.1).

S (E,{R1,...,Rn}) $\leftarrow$ TA-en-TE(R)

P
- re1 $\notin$ groupe(E1) $\wedge$ ... $\wedge$ ren $\notin$ groupe(En)
- #(role(R)) > 1

Q
- E $\in$ type-entites(S) $\wedge$ nom(E) = nom(R)
- $\forall$ a $\in$ attribut(R) : a $\in$ attribut(E)
- $\forall$ g $\in$ groupe(R) : g $\in$ groupe(E)
- R1,...,Rn $\in$ type-associations(S)
- re1 $\in$ R1 $\wedge$ ... $\wedge$ ren $\in$ Rn
- r1 $\in$ R1 $\wedge$ ... $\wedge$ rn $\in$ Rn
- te-role(r1) = {E} $\wedge$ ... $\wedge$ te-role(rn) = {E}
- card(r1) = [1-1] $\wedge$ ... $\wedge$ card(rn) = [1-1]
- pid(E) = $\varnothing$ $\wedge$ ($\forall$ rek $\in$ role(Rk), k $\in$ [1..n] : card-max(rek) > 1) $\wedge$ id $\in$ pid(E) $\wedge$ {re1,...,ren} $\subset$ composant(id)
- R $\notin$ type-associations(S)

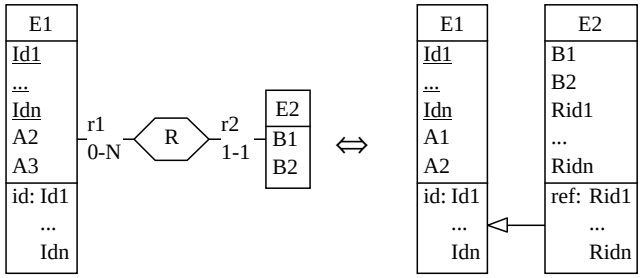
I $\forall$ r $\in$ instance(R), $\exists!$ e $\in$ instance(E) : ($\forall$ i $\in$ [1..n], $\exists$ ri $\in$ instance(Ri) : ri[rei] = r[rei]) $\wedge$ e[A1] = r[A1] $\wedge$ e[A2] = r[A2]

Pour chaque instance de R, on crée une instance de E qui est reliée aux mêmes instances de E1, ..., En (via des instances de R1, ..., Rn) et qui a les mêmes valeurs pour les attributs A1 et A2.

T^{-1} R $\leftarrow$ TE-en-TA(E) : transformation d'un type d'entités en type d'associations (point B.2.1).

B.3.2 Transformation d'un type d'associations en une clé étrangère

D Un type d'associations binaire fonctionnel R est transformé en une clé étrangère dans le type d'entités E2. Il s'agit de la transformation principale pour produire des schémas relationnels.



S (E2,{Rid1,...,Ridn}) ← TA-en-FK(R)

P
- attribut(R) = ∅
- {r1,r2} = role(R) ∧ et-role(r1) = {E1} ∧ et-role(r2) = {E2}
- ∃ i ∈ pid(E1) : composant(i) ⊂ attribut(E1)
- card-max(r2) = 1
- ¬∃ g ∈ groupe(E1) : r2 ∈ composant(g)

Q
- {Rid1,...,Ridn} ⊂ attribut(E2)
- ∀ k ∈ [1..n] :nom(Ridk) = nom(Idk) ∧ type(Ridk) = type(Idk) ∧ longueur(Ridk) = longueur(Idk) ∧ card(Ridk) = card(r2)
- g ∈ groupe(E2) ∧ composant(g) = {Rid1,...,Ridn}
- card-max(r1) = 1 ∧ ((pid(E2) = ∅ ∧ fonction(g) = id-primaire) ∨ (pid(E2) ≠ ∅ ∧ fonction(g) = id-secondaire))
- card-min(r2) = 0 ∧ fonction(g) &= coexistence
- card(g) = card(r2)
- ∀ f ∈ groupe(E2) ∧ r1 ∈ composant(f) : r1 ∉ composant(f) ∧ {Rid1,...,Ridn} ⊂ composant(f)
- card-min(r1) = 1 ∧ g ∈ est-ref-egal(i)
- card-min(r1) = 0 ∧ g ∈ est-reference(i)
- R ∉ type-associations(R)

I ∀ r ∈ instance(R), ∃! e2 ∈ instance(E2), e1 ∈ instance(E1) : r[r1,r2] = (e1,e2) ∧ (∀ i ∈ [1..n] : e2[Ridi] = e1[Idi])

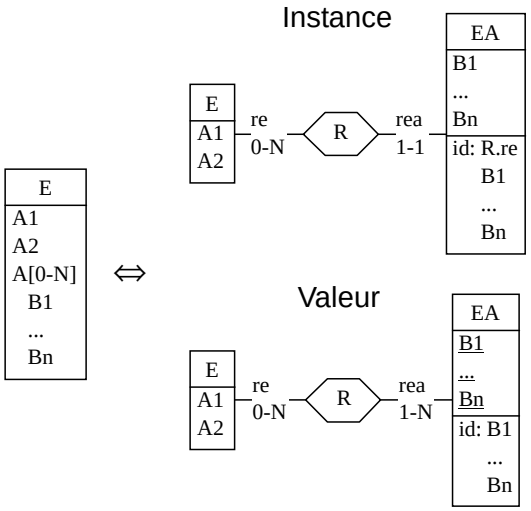
Pour chaque instance de R reliant une instance de E1 et une instance de E2, on donne aux attributs Rid1, ..., Ridn de l'instance de E2 les mêmes valeurs que celles des attributs Id1, ..., Idn de l'instance de E1.

T^{-1} R ← FK-en-TA(E2,{Rid1,...,Ridn}) : transformation d'une clé étrangère en un type d'associations (point B.4.2).

B.4 Transformations d'attributs

B.4.1 Transformation d'un attribut en un type d'entités

D Un attribut est transformé en un type d'entités EA indépendant. Il y a deux variantes selon que chaque instance de EA représente une valeur distincte de l'attribut (représentation par valeur) ou une instance de l'attribut (représentation par instance). Cette transformation permet d'éliminer des attributs multivalués ou décomposables, de promouvoir des attributs au rang de type d'entités, ...



S (EA,R) ← Att-en-TE-inst(E,A)

(EA,R) ← Att-en-TE-val(E,A)

P *Instance*

- A ∈ Attribut(E)
- ∀ i ∈ identifiant(E), ∃ c ∈ composant(i) ∩ {A,B1,...,Bn} : composant(i) ⊂ {A,B1,...,Bn}
- ∀ r ∈ reference(E), ∃ c ∈ composant(i) ∩ {A,B1,...,Bn} : composant(r) ⊂ {A,B1,...,Bn}
- ∀ g ∈ group(E), ∃ c ∈ composant(i) ∩ {A,B1,...,Bn} ∧ fonction(g) = exclusif | au-moins-un : composant(g) ⊂ {A,B1,...,Bn}
- card-max(A) > 1 ∧ type(A) = bag ∧ (¬∃ i ∈ identifiant(E) : composant(i) ⊂ {A,B1,...,Bn})
- card-max(A) > 1 ∧ (type(A) = bag ∨ type(A) = set)

Valeur

- A ∈ Attribut(E)
- ∀ r ∈ reference(E), ∃ c ∈ composant(i) ∩ {A,B1,...,Bn} : composant(r) ⊂ {A,B1,...,Bn}
- ∀ g ∈ group(E), ∃ c ∈ composant(i) ∩ {A,B1,...,Bn} ∧ fonction(g) = exclusif | au-moins-un : composant(g) ⊂ {A,B1,...,Bn}
- identifiant(A) = ∅
- card-max(A) > 1 ∧ type(A) = set

Q *Instance*

- EA ∈ type-entites(S) ∧ R ∈ type-associations(S) ∧ role(R) = {re,rea} ∧ {E} = te-role(re) ∧ {EA} = te-role(rea) ∧ card(rea) = [1-1] ∧ card(re) = card(A)

– $\forall$ g $\in$ groupe(E) : composant(g) $\in$ {A,B1,...,Bn} $\wedge$ g $\in$ groupe(EA)

– $\forall$ g $\in$ groupe(E), $\forall$ c $\in$ composant(g) $\wedge$ c $\in$ {A,B1,...,Bn} : c $\notin$ composant(g) $\wedge$ rea $\in$ composant(g)

– A $\in$ attribut(EA) $\wedge$ card(A) = [1-1]

– attribut(A) $\neq \varnothing \Rightarrow$ (EA,{B1,...,Bn}) $\leftarrow$ desagreger-Att(EA,A)

Valeur

– EA $\in$ type-entites(S) $\wedge$ R $\in$ type-associations(S) $\wedge$ role(R) = {re,rea} $\wedge$ {E} = te-role(re) $\wedge$ {EA} = te-role(rea) $\wedge$ card(rea) = [1-N] $\wedge$ card(re) = card(A)

– $\forall$ g $\in$ groupe(E) : composant(g) $\in$ {A,B1,...,Bn} $\wedge$ g $\in$ groupe(EA)

– $\forall$ g $\in$ groupe(E), $\forall$ c $\in$ composant(g) $\wedge$ c $\in$ {A,B1,...,Bn} : c $\notin$ composant(g) $\wedge$ rea $\in$ composant(g)

– pid(EA) = $\varnothing$ g $\in$ identifiant(EA) $\wedge$ fonction(g) = id-primaire $\wedge$ composant(g) = {A}

– A $\in$ attribut(EA) $\wedge$ card(A) = [1-1]

– attribut(A) $\neq \varnothing \Rightarrow$ (EA,{B1,...,Bn}) $\leftarrow$ desagreger-Att(EA,A)

I $\forall$ e $\in$ instance(E), $\exists$ ri $\in$ instance(R), eai $\in$ instance(EA), i $\in$ [1..N] : ri[re,rea] = (e,eai) $\wedge$ e[A[i]] = eai[B1, ...,Bn]

Pour chaque instance de E, on crée autant d'instances de R qu'il y a de valeurs dans l'attribut A. Dans une représentation par instance, pour chaque instance de R, on crée une instance de EA qui contient la valeur de l'attribut A. Dans une représentation par valeur, pour chaque instance de R, on crée une instance de EA qui contient la valeur de l'attribut A si aucune instance de EA n'a pas déjà la même valeur.

T^{-1} (E,A) $\leftarrow$ TE-en-Att(EA,R) : transformation d'un type d'entités en un attribut (point B.2.2).

B.4.2 Transformation d'une clé étrangère en un type d'associations

D Des attributs de référence de E2 et leur clé étrangère sont transformés en un type d'associations binaire fonctionnel R (pour la représentation graphique, consulter la transformation d'un type d'associations en une clé étrangère au point B.3.2).

S R $\leftarrow$ FK-en-TA(E2,{Rid1,...,Ridn})

P – fk $\in$ reference(E2) $\wedge$ composant(fk) = {Rid1,...,Ridn}

– i $\in$ identifiant(E1) $\wedge$ composant(i) = {Id1,...,Idn}

– fk $\in$ est-reference(i)

– longueur(Id1) = longueur(Rid1) $\wedge$ ... $\wedge$ longueur(Idn) = longueur(Ridn)

– type(Id1) = type(Rid1) $\wedge$ ... $\wedge$ type(Idn) = type(Ridn)

– card(Rid1) = ... = card(Ridn)

– card-min(Rid1) = 0 $\wedge$ ($\exists$ g1 $\in$ groupe(E2) : fonction(c) = coexistence $\wedge$ composant(c) = {Rid1,...,Ridn})

– $\neg\exists$ a $\in$ {Rid1,...,Ridn} $\wedge$ g1 $\in$ reference(E2) : g1 $\neq$ fk $\wedge$ a $\in$ composant(g1)

Q – R $\in$ type-associations(S) $\wedge$ role(R) = {r1,r2} $\wedge$ {E1} = te-role(r1) $\wedge$ {E2} = te-role(r2) $\wedge$ card(r1) = card(fk) $\wedge$ card(r2) = card(Rid1)

– $\forall$ g groupe(E2) $\wedge$ {Rid1,...,Ridn} $\subset$ composant(g) : {Rid1,...,Ridn} $\not\subset$ composant(g) $\wedge$ r1 $\in$ composant(g)

– {Rid1,...,Ridn} $\not\subset$ attribut(E2)

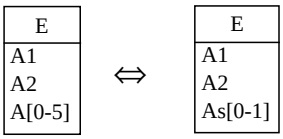
I $\forall$ e2 $\in$ instance(E2), e1 $\in$ instance(E1), $\exists!$ r $\in$ instance(R) : ($\forall$ i $\in$ [1..n] : e2[Ridi] = e1[Idi]) $\wedge$ r[r1,r2] = (e1,e2)

Pour toutes les instances de E1 et de E2 qui ont les mêmes valeurs pour les attributs Id1, ..., Idn et Rid1, ..., Ridn, on crée une instance de R reliant les instances de E1 et E2 considérées.

T^{-1} (E2,{Rid1,...,Ridn}) $\leftarrow$ TA-en-FK(R) : transformation d'un type d'associations en une clé étrangère (point B.3.2).

B.4.3 Concaténation d'un attribut multivalué

D Un attribut multivalué A est remplacé par un attribut monovalué dont chaque valeur est la concaténation des valeurs de l'attribut A d'origine.



S (E,As) $\leftarrow$ Attmult-en-Attmono(E,A)

P – A $\in$ attribut(E) $\wedge$ card-max(A) > 1 $\wedge$ type(A) = car

– $\neg\exists$ g $\in$ groupe(E) : A $\in$ composant(g)

Q – As $\in$ attribut(E) $\wedge$ type(As) = type(A)

– card-max(As) = 1

– (card-min(A) = 0 $\wedge$ card-min(As) = 0) $\vee$ (card-min(A) $\geq$ 1 $\wedge$ card-min(As) = 1)

– longueur(As) = longueur(A) * card-max(A)

Cette transformation n'est pas totalement réversible puisque l'attribut concaténé induit un ordre sur les valeurs de A qui n'existe pas dans l'attribut multivalué original.

I $\forall$ e $\in$ instance(E) : agreger(e[A[1]],e[A[2]],e[A[3]],e[A[4]],e[A[5]],e[As])

Pour toutes les instances de E, les valeurs de l'attribut A sont concaténées dans la valeur de l'attribut As.

T^{-1} (E,A) $\leftarrow$ Attmono-en-Attmult(E,As) : transformation d'un attribut monovalué en un attribut multivalué (point B.4.4).

B.4.4 Transformation d'un attribut monovalué en un attribut multivalué

D Un attribut monovalué As est remplacé par un attribut multivalué A par déconcaténation de chaque valeur de As dans les valeurs de l'attribut A (pour la représentation graphique, consulter la transformation du point B.4.3).

S (E,A) $\leftarrow$ Attmono-en-Attmult(E,As)

P – As $\in$ Attribut(E) $\wedge$ type(A) = car $\wedge$ card-max(As) = 1

– $\neg\exists$ g $\in$ groupe(E) : As $\in$ composant(g)

Q – A $\in$ attribut(E) $\wedge$ type(A) = type(As)

– card-min(A) = card-min(As)

– (longueur(As) = N $\wedge$ longueur(A) = N) $\vee$ (longueur(As) < N $\wedge$ longueur(A) = x)

– (longueur(A) = N $\wedge$ card-max(A) = N) $\vee$ (longueur(A) < N $\wedge$ card-max(A) = longueur(A) / x)

Cette transformation n'est pas totalement réversible puisque l'attribut A peut perdre l'ordre induit dans l'attribut original As.

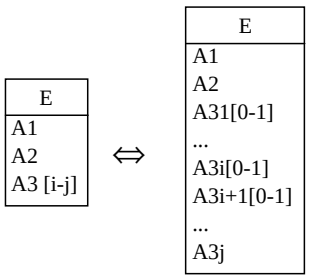
I $\forall$ e $\in$ instance(E) : desagreger(e[As],e[A[1]],e[A[2]],e[A[3]],e[A[4]],e[A[5]])

Pour toutes les instances de E, la valeur de l'attribut As est instanciée dans les valeurs de l'attribut A.

T^{-1} (E,As) $\leftarrow$ Attmult-en-Attmono(E,A) : concaténation d'un attribut multivalué (point B.4.3).

B.4.5 Transformation d'un attribut multivalué en une série d'attributs

D L'attribut multivalué A3 est remplacé par une série d'attributs monovalués. Cette transformation permet de remplacer des attributs multivalués par des attributs monovalués.



S (E,{A31,...,A3j}) $\leftarrow$ Attmult-en-Serie-Att(E,A3)

P – A3 $\in$ Attribut(E) $\wedge$ card-max(A) > 1 $\wedge$ card-max(A) < N

– $\neg\exists$ g $\in$ identifiant(E) $\cup$ reference(E) : A $\in$ composant(g) $\wedge$ #(composant(g)) > 1

Q – {A31,...,A3j} $\subset$ attribut(E) $\wedge$ type(A31) = ... = type(A3j) = type(A3) $\wedge$ longueur(A31) = ... = longueur(A3j) = longueur(A3)

– card-max(A31) = ... = card-max(A3j) = 1

– card-min(A31) = ... = card-min(A3i) = 1 $\wedge$ card-min(A3i+1) = ... = card-min(A3j) = 0

– $\forall$ g $\in$ groupe(E) $\wedge$ A3 $\in$ composant(g) $\wedge$ ($\forall$ k $\in$ [1..j] : gk $\in$ groupe(E) $\wedge$ fonction(gk) = fonction(g) $\wedge$ composant(gk) = composant(g) $\cup$ {A3k} / {A3})

– A3 $\notin$ attribut(E)

I $\forall$ e $\in$ instance(E), k $\in$ [1..j] : e[A3k] = e[A3[k]]

Pour toutes les instances de E, les valeurs de l'attribut A3 sont transférées dans les valeurs des attributs A31, ..., A3j.

T^{-1} (E,A3) $\leftarrow$ Serie-Att-en-Attmult(E,{A31,...,A35}) : transformation d'une série d'attributs en un attribut multivalué (point B.4.6).

B.4.6 Transformation d'une série d'attributs en un attribut multivalué

D La série d'attributs A31,...,A3j est remplacée par un attribut multivalué (pour la représentation graphique, consulter la transformation d'un attribut multivalué en une série d'attributs au point B.4.5).

S (E,A3) $\leftarrow$ Serie-Att-en-Attmult(E,{A31,...,A3j})

P – {A31,...,A3j} $\subset$ attribut(E)

– type(A31) = ... = type(A3j) $\wedge$ longueur(A31) = ... = longueur(A3j) $\wedge$ card-max(A31) = ... = card-max(A3j) = 1

– $\forall$ g $\in$ groupe(E) $\wedge$ fonction(g) $\neq \varnothing$, $\exists$ c $\in$ componant(g) $\cap$ {A31,...,A3j} : componant(g) = {A31,...,A3j}

Q – A3 $\in$ attribut(E) $\wedge$ type(A3) = type(A31) $\wedge$ longueur(A3) = longueur(A31)

– card-max(A3) = j

– card-min(A3) = i

– $\forall$ g $\in$ groupe(E) $\wedge$ A31 $\in$ composant(g) : A3 $\in$ composant(g)

– {A31,...,A3j} $\not\subset$ attribut(E)

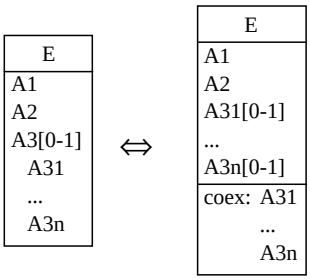
I $\forall$ e $\in$ instance(E), k $\in$ [1..j] : e[A3[k]] = e[A3k]

Pour toutes les instances de E, les valeurs des attributs A31, ..., A3j sont transférées dans les valeurs de l'attribut A3.

T^{-1} (E,{A31,...,A3j}) $\leftarrow$ Attmult-en-Serie-Att(E,A3) : transformation d'un attribut multivalué en une série d'attributs (point B.4.5).

B.4.7 Désagrégation d'un attribut décomposable

D L'attribut décomposable A3 est remplacé par ses composants.



S (E,{A31,...,A3n})$\leftarrow$ desagreger-Att(E,A3)

P – A3 $\in$ attribut(E) $\wedge$ attribut(A3) $\neq \varnothing$ $\wedge$ card-max(A3) = 1

– card(A31) = ... = card(A3n) = [1-1] $\vee$ card(A3) = [1-1]

– $\exists$ g $\in$ identifiant(E) $\wedge$ A3 $\in$ composant(g) : card-max(A31) = ... = card-max(A3n) = 1

Q – {A31,...,A3n} $\subset$ attribut(E)

– $\forall$ i $\in$ [1..n] : card-min(A3i) = card-min(A3i) * card-min(A3) $\wedge$ card-max(A3i) = card-max(A3i) * card-max(A3)

– card-min(A3) = 0 $\wedge$ n > 1 $\wedge$ g $\in$ groupe(E) $\wedge$ fonction(g) = coexistence $\wedge$ composant(g) = {A31,...,A3n}

– $\forall$ g $\in$ groupe(E) $\wedge$ A3 $\in$ composant(g) : A3 $\notin$ composant(g) $\wedge$ {A31,...,A3n} $\subset$ composant(g)

– A3 $\notin$ attribut(E)

I $\forall$ e $\in$ instance(E), k $\in$ [1..n] : e[A3k] = e[A3[A3k]]

Pour toutes les instances de E, les valeurs des attributs A3.A31, ..., A3.A3n sont transférées dans les valeurs des attributs A31, ..., A3n.

T^{-1} (E,A3) $\leftarrow$ agreger-Att(E,{A31,...,A3n}) : agrégation d'une liste d'attributs en un attribut décomposable (point B.4.8).

B.4.8 Agrégation d'une liste d'attributs en un attribut décomposable

D Une liste d'attributs est groupée dans un attribut décomposable. Les attributs doivent avoir le même objet parent (pour la représentation graphique, consulter la désagrégation d'un attribut décomposable au point B.4.7).

S (E,A3) $\leftarrow$ agreger-Att(E,{A31,...,A3n})

P – {A31,...,A3n} $\subset$ attribut(E)

Q – A3 $\in$ attribut(E) $\wedge$ card(A3) = [1-1]

– {A31,...,A3n} $\subset$ attribut(A3)

– $\exists$ g $\in$ groupe(E) $\wedge$ {A31,...,A3n} = composant(g) $\wedge$ fonction(g) = coexistence : card-min(A3) = 0 $\wedge$ card-min(A31) = ... = card-min(A3n) = 1 $\wedge$ g $\notin$ groupe(E)

– $\forall$ g $\in$ groupe(E) $\wedge$ {A31,...,A3n} $\subset$ composant(g) : {A31,...,A3n} $\not\subset$ composant(g) $\wedge$ A3 $\in$ composant(g)

I $\forall$ e $\in$ instance(E), k $\in$ [1..n] : e[A3[A3k]] = e[A3k]

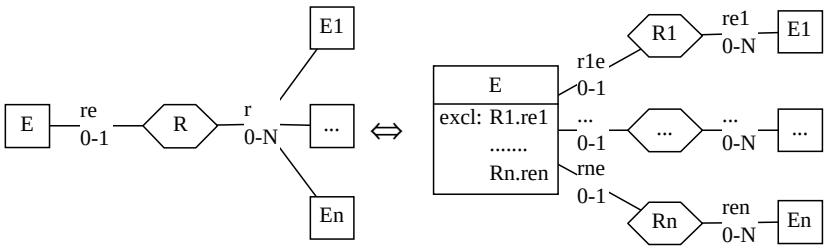
Pour toutes les instances de E, les valeurs des attributs A31, ..., A3n sont transférées dans les valeurs des attributs A3.A31, ..., A3.A3n.

T^{-1} (E,{A31,...,A3n}) $\leftarrow$ desagreger-Att(E,A3) : désagrégation d'un attribut décomposable (point B.4.7).

B.5 Transformations des rôles multi-domaines

B.5.1 Transformation d'un rôle multi-domaines en types d'associations

D Un rôle joué par plusieurs types d'entités est remplacé par des types d'associations reliés aux types d'entités jouant le rôle transformé.



S (R1,...,Rn) ← Rolemult-en-TA(R,r)

P – te-role(r) = {E1,...,En}

Q – ∀ i ∈ [1..n] : Ri ∈ type-associations(S) ∧ {rie,rei} = role(Ri) ∧ card-min(rie) = 0 ∧ card-max(rie) = card-max(re) ∧ card(rei) = card(r) ∧ te-role(re) = {E} ∧ te-role(rei) = {Ei} ∧ attribut(R) = attribut(Ri) ∧ groupe(R) = groupe(Ri)

– ∀ g ∈ groupe(E) / identifiant(E) ∧ r ∈ composant(g) : r ∉ composant(g) ∧ {re1,...,ren} ⊂ composant(g)

– ∀ g ∈ identifiant(E) ∧ r ∈ composant(g) ∧ (∀ k ∈ [1..n] : gk ∈ identifiant(E) ∧ composant(gk) = composant(g) ∪ {rei} / {r})

– (∀ i ∈ [1..n], ∀ g ∈ groupe(Ei) : re ∈ composant(g)) ∧ re ∉ composant(g) ∧ rie ∈ composant(g)

– card(re) = [0-1] ∧ g ∈ groupe(E) ∧ fonction(g) = exclusif ∧ composant(g) = {re1,...,ren}

– card(re) = [1-1] ∧ g ∈ groupe(E) ∧ fonction(g) = exactement-un ∧ composant(g) = {re1,...,ren}

– card(re) = [1-N] ∧ g ∈ groupe(E) ∧ fonction(g) = au-moins-un ∧ composant(g) = {re1,...,ren}

– R ∉ type-associations(S)

I ∀ r ∈ instance(R), ∃! ri ∈ instance(Ri), i ∈ [1..n] : ri[rie,rei] = r[re,r]

Pour chaque instance de R, il existe une seule instance de Ri (i ∈ [1..n]) reliant les mêmes instances de types d'entités.

T^{-1} (R,r) ← TA-en-Rolemult(R1,...,Rn) : transformation de types d'associations en un rôle multi-domaines (point B.5.2).

B.5.2 Transformation de types d'associations en un rôle multi-domaines

D Des types d'associations binaires liés à un type d'entités commun (E) sont remplacés par un type d'associations binaire lié à E et aux autres types d'entités impliqués dans les types d'associations transformés par le biais d'un rôle multi-domaines (pour la représentation gra-

phique, consulter la transformation d'un rôle multi-domaines en types d'associations au point B.5.1).

S (R,r) ← TA-en-Rolemult(R1,...,Rn)

P
- ∀ i ∈ [1..n] : Ri ∈ type-associations(S) ∧ {rie,rei} = role(Ri) ∧ te-role(rie) = {E} ∧ te-role(rei) = Ei
- card(r1e) = ... = card(rne) ∧ card(re1) = ... = card(ren) ∧ attribut(E1) = ... = attribut(Ei) ∧ groupe(E1) = ... = groupe(Ei)
- ∀ g ∈ groupe(E) / identifiant(E) ∧ c ∈ composant(g) ∧ c ∈ {re1,...,ren} : {re1,...,ren} ⊂ composant(g)

Q
- R ∈ type-associations(S) ∧ {re,r} = role(R) ∧ te-role(re) = {E} ∧ te-role(r) = {E1,...,En} ∧ card(re) = card(r1e) ∧ card(r) = card(re1)
- ∀ g ∈ groupe(E) ∧ {re1,...,ren} ⊂ composant(g) : {re1,...,ren} ⊄ composant(g) ∧ r ∈ composant(g)
- (∀ i ∈ [1..n], ∀ g ∈ groupe(Ei) : rie ∈ composant(g)) ∧ rie ∉ composant(g) ∧ re ∈ composant(g)
- ∃ g ∈ groupe(E) ∧ composant(g) = {re1,...,ren} ∧ fonction(g) = exclusif : card-min(re) = 0
- ∃ g ∈ groupe(E) ∧ composant(g) = {re1,...,ren} ∧ fonction(g) = exactement-un | au-moins-un : card-min(re) = 1
- {R1,...,Rn} ⊄ type-associations(S)

I ∀ ri ∈ instance(Ri), i ∈ [1..n], ∃! r ∈ instance(R) : r[re,r] = ri[rie,rei]

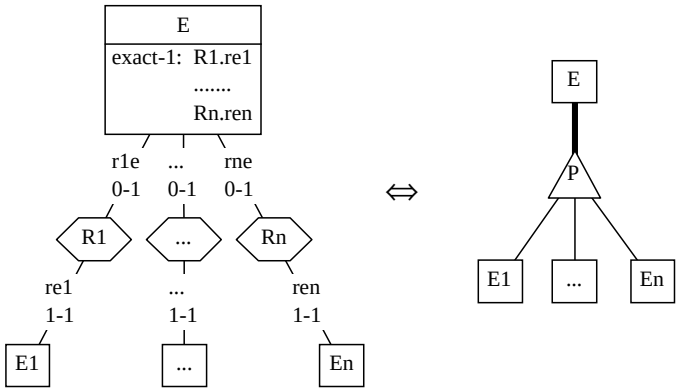
Pour chaque instance de R1, ..., Rn, il existe une seule instance de R reliant les mêmes instances de types d'entités.

T^{-1} (R1,...,Rn) ← Rolemult-en-TA(R,r) : transformation d'un rôle multi-domaines en types d'associations (point B.5.1).

B.6 Transformations de relations IS-A

B.6.1 Transformation de types d'associations en relations IS-A

D Le type d'entités A commun à un ensemble de types d'associations un-à-un est transformé en un surtype des types d'entités (E1,...,En). Les types d'associations (R1,...,Rn) sont transformés en relations IS-A.



S (E,{E1,...,En}) ← TA-en-ISA(E,{R1,...,Rn})

P
- ∀ i ∈ [1..n] : Ri ∈ type-associations(S) ∧ {rie,rei} = role(Ri) ∧ te-role(rie) = {E} ∧ te-role(rei) = {Ei} ∧ card(rie) = [0-1] ∧ card(rei) = [1-1] ∧ attribut(Ri) = ∅
- ∀ g ∈ groupe(E), ∃ c ∈ composant(g) : c ∈ {re1,...,ren} ∧ fonction(g) = exclusif | au-moins-un ∧ composant(g) = {re1,...,ren}

Q
- {E1,...,En} ⊂ sous-type(E)
- ∃ g ∈ groupe(E) : composant(g) = {R1,...,Rn} ∧ fonction(g) = exclusif ∧ type(sous-type(E)) = disjoint
- ∃ g ∈ groupe(E) : composant(g) = {R1,...,Rn} ∧ fonction(g) = au-moins-un ∧ type(sous-type(E)) = total
- ∃ g ∈ groupe(E) : composant(g) = {R1,...,Rn} ∧ fonction(g) = exclusif & au-moins-un ∧ type(sous-type(E)) = partition
- {R1,...,Rn} ⊄ type-associations(S)

I ∀ ri ∈ instance(Ri), i ∈ [1..n], ∃ e ∈ instance(E) : ri[rie] = e ∧ e[Ei] = ri[rei]

Pour chaque instance de Ri (i ∈ [1..n]), il existe une instance de E liée par une relation IS-A a la même instance de Ei.

T^{-1} (E,{R1,...,Rn}) ← ISA-en-TA(E,{E1,...,En}) : transformation de relations IS-A en types d'associations (point B.6.2).

B.6.2 Transformation de relations IS-A en types d'associations

D Les relations IS-A entre le surtype E et ses sous-types (E1,...,En) sont transformées en types d'associations un-à-un (pour la représentation graphique, consulter la transformation de types d'associations en relations IS-A au point B.6.1).

S (E,{R1,...,Rn}) ← ISA-en-TA(E,{E1,...,En})

P – {E1,...,En} ⊂ sous-type(E)

Q – ∀ i ∈ [1..n] : Ri ∈ type-associations(S) ∧ {rie,rei} = role(Ri) ∧ te-role(rie) = {E} ∧ te-role(rei) = {Ei} ∧ card(rie) = [0-1] ∧ card(rei) = [1-1]

– type(sous-type(E)) = disjoint ∧ g ∈ groupe(E) ∧ composant(g) = {R1,...,Rn} ∧ fonction(g) = exclusif

– type(sous-type(E)) = total ∧ g ∈ groupe(E) ∧ composant(g) = {R1,...,Rn} ∧ fonction(g) = au-moins-un

– type(sous-type(E)) = partition ∧ g ∈ groupe(E) ∧ composant(g) = {R1,...,Rn} ∧ fonction(g) = exclusif & au-moins-un

– sous-type(E) = ∅

I ∀ e ∈ instance(E), ∃ ri ∈ instance(Ri), i ∈ [1..n] : ri[rie,rei] = (e,e[Ei])

Pour chaque instance de E, il existe des instances de Ri (i ∈ [1..n]) liant cette instance de E avec les mêmes instances de Ei.

T^{-1} (E,{E1,...,En}) ← TA-en-ISA(E,{R1,...,Rn}) : transformation de types d'associations en relations IS-A (point B.6.1).

ANNEXE C

TYPOLOGIE DES MODIFICATIONS DES SPÉCIFICATIONS : ÉTUDE COMPLÈTE

C.1 Introduction

Dans la typologie, chaque modification est décrite pour les niveaux conceptuel, logique et physique par les caractéristiques suivantes :

- une brève description (D) avec une représentation graphique générique,
- la signature (S),
- la catégorie sémantique (C) à laquelle elle appartient en termes de préservation de l'information (voir point 3.2.2),
- les préconditions (P) à respecter avant la modification (suivant la notation prédicative définie dans l'annexe A.1),
- les postconditions (Q) après la modification (notation prédicative),
- une description des modifications des instances (I) (notation prédicative),
- une remarque éventuelle (R),
- un exemple (E).

Au niveau opérationnel (données et programmes), chaque modification est analysée en termes de dépendance/indépendance par rapport aux données ou aux programmes. Elles sont décrites par les propriétés suivantes :

- au niveau des données et structures de données :
  - la dépendance (D) des données par rapport à la modification (point 4.1.3.1),
  - le script (S) de conversion (en SQL-92) des données et structures de données (point 4.1.3.2),
  - une remarque éventuelle (R),
  - un exemple (E).
- au niveau des programmes :
  - la dépendance (D) des programmes par rapport à la modification (point 4.1.3.3),
  - la localisation (L) du code à modifier dans les programmes (point 4.1.3.4).

Dans les scripts de conversion, le respect de la syntaxe SQL-92 est la première priorité. Toutefois, la norme a certaines lacunes (comme les triggers, les espaces de stockage et les index qui sont absents) et les systèmes existants disposent de commandes non reconnues par la norme. C'est pourquoi le tableau C.1 présente, pour certains concepts, les commandes particulières définies par les SGBDR les plus présents sur le marché. Le sigle "/" signifie que la commande n'existe pas dans le SGBD considéré.

	Adaptive Server Enterprise 12.0	**DB2 5.2**	**Informix Dynamic Server.2000**	**Oracle 8**	**SQL Server 2000**
Table					
Renommage	Procédure système : `SP_RENAME old_t, new_t`	`RENAME TABLE old_t TO new_t`	`RENAME TABLE old_t TO new_t`	`RENAME old_t TO new_t`	Procédure système : `SP_RENAME 'old_t', 'new_t'`
Synonyme	/	`CREATE <ALIAS | SYNONYM> new_t FOR old_t`	`CREATE SYNONYM new_t FOR old_t`	`CREATE SYNONYM new_t FOR old_t`	/
Colonne					
Contrainte NULL par défaut	`NOT NULL`	`NULL`	`NULL`	`NULL`	Paramètre de configuration

Tableau C.1 - Commandes SQL et caractéristiques spécifiques à certains SGBDR.

	Adaptive Server Enterprise 12.0	DB2 5.2	Informix Dynamic Server.2000	Oracle 8	SQL Server 2000
Destruction	`ALTER TABLE t DROP c`	/	`ALTER TABLE t DROP c`	/	`ALTER TABLE t DROP COLUMN c`
Renommage	Procédure système : `SP_RENAME t.old_c, new_c`	/	`RENAME COLUMN t.old_c TO new_c`	/	Procédure système : `SP_RENAME 't.old_c', 'new_c', 'COLUMN'`
Ajout contrainte NOT NULL	/	/	`ALTER TABLE t MODIFY c type NOT NULL`	`ALTER TABLE t MODIFY c NOT NULL`	`ALTER TABLE t ALTER COLUMN c type NOT NULL`
Modification du type	`ALTER TABLE t MODIFY c new_type`	Type caractère uniquement : `ALTER TABLE t ALTER c SET DATA TYPE new_type`	`ALTER TABLE t MODIFY c new_type`	`ALTER TABLE t MODIFY c new_type`	`ALTER TABLE t ALTER COLUMN c new_type`
Clé étrangère					
Clé référencée	`primaire ou candidate`	`primaire ou candidate`	`primaire ou candidate`	`primaire ou candidate`	`primaire ou candidate`
Egalité traduite par un CHECK	`Possible`	/	`Possible`	/	`Possible`
Egalité traduite par une ASSERTION	/	/	/	`CREATE ASSERTION a CHECK(...);` (Oracle 7 uniquement)	/
Identifiant					
Valeur nulle	`Colonne NOT NULL et NULL`	`Colonne NOT NULL seulement`	`Colonne NOT NULL et NULL`	`Colonne NOT NULL et NULL`	`Colonne NOT NULL et NULL`
Index					
Création	`CREATE INDEX i ON t (c1,...,cn)`	`CREATE INDEX i ON t (c1,...,cn)`	`CREATE INDEX i ON t (c1,...,cn)`	`CREATE INDEX i ON t (c1,...,cn)`	`CREATE INDEX i ON t (c1,...,cn)`
Création automatique d'index	`sur contraintes PRIMARY KEY et UNIQUE`	/	`sur contrainte PRIMARY KEY`	`sur contraintes PRIMARY KEY et UNIQUE`	`sur contraintes PRIMARY KEY et UNIQUE`
Renommer	Procédure système : `SP_RENAME old_i, new_i`	/	/	`ALTER INDEX old_i RENAME TO new_i`	Procédure système : `SP_RENAME 'old_i', 'new_i', 'INDEX'`
Espace de stockage					
Création	Procédure système : `SP_ADDSEGMENT e`	`CREATE TABLESPACE e MANAGED BY DATABASE USING (FILE f nbr_page);`	Procédure système : `ONSPACES [param] e [param]`	`CREATE TABLESPACE e DATAFILE f [DEFAULT STORAGE (param)];`	`ALTER DATABASE d ADD FILEGROUP e`
Destruction	L'espace de stockage doit être vide. Procédure système : `SP_DROPSEGMENT e`	L'espace de stockage doit être vide. `DROP TABLESPACE e`	Les tables de l'espace doivent être supprimées. Procédure système : `ONSPACES [param] e [param]`	L'espace de stockage doit être vide. `DROP TABLESPACE e`	L'espace de stockage doit être vide. `ALTER DATABASE d REMOVE FILEGROUP e`
Ajout d'une table	à la création de la table : `CREATE TABLE t (...) ON e`	à la création de la table : `CREATE TABLE t (...) IN e`	à la création de la table : `CREATE TABLE t (...) IN e`	à la création de la table : `CREATE TABLE t (...) TABLESPACE e`	à la création de la table : `CREATE TABLE t (...) ON e`

Tableau C.1 - Commandes SQL et caractéristiques spécifiques à certains SGBDR.

C.2 Noyau E/A → Noyau relationnel

C.2.1 Introduction

Pour toutes informations complémentaires sur les contraintes des modèles et les correspondances entre les concepts, le lecteur est renvoyé au chapitre 4 (section 4.2 à la page 80).

C.2.2 Modifications de spécifications conceptuelles

C.2.2.1 Modifications relatives aux types d'entités

a) Création

D Créér un type d'entités.

S $E \leftarrow$ creer-TE(S,n)

C T^+

P – $\forall E' \in$ type-entites(S) : nom(E') $\neq$ n

Q – $E \in$ type-entites(S)

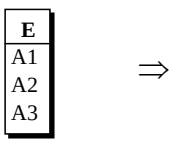
– nom(E) = n

I instance(E) = $\varnothing$

E Création du type d'entités *FOURNISSEUR* qui a un numéro (*Nfour*), un nom (*Nom*), une adresse (*Adresse*), un numéro de téléphone (*Telephone*) et un jour de livraison (*Jour*).

b) Destruction

D Détruire un type d'entités.



S $\exists E \in$ type-entites(S) : () $\leftarrow$ supprimer-TE(E)

C T^-

P – $E \in$ type-entites(S)

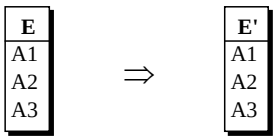
Q – $E \notin$ type-entites(S)

I ∀ i ∈ instance(E) : supprimer(i)

E On détruit le type d'entités *FOURNISSEUR* ainsi que les types d'associations auxquels il est relié.

c) Renommage

D Renommer un type d'entités.



S ∃ E ∈ type-entites(S) : E ← modifier-TE(E,n')

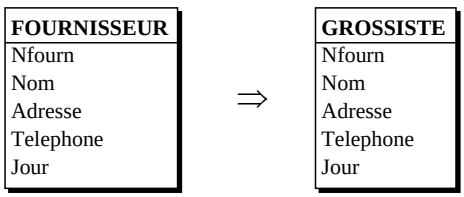
C $T^{=}$

P – E ∈ type-entites(S)
– nom(E) = n
– ∀ E' ∈ type-entites(S) : nom(E') ≠ n'

Q – nom(E) = n'

I /

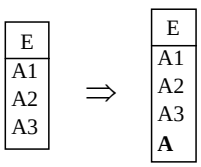
E On renomme le type d'entités *FOURNISSEUR* en *GROSSISTE*.



C.2.2.2 Modifications relatives aux attributs

a) Création

D Créer un attribut facultatif ou obligatoire pour un type d'entités.



S ∃ E ∈ type-entites(S) : A ← creer-Att(E,n)

C T^{+}

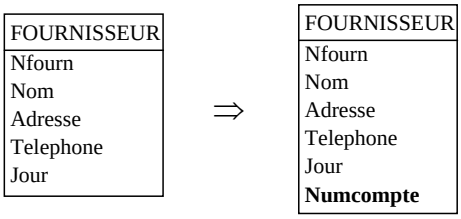
P – E ∈ type-entites(S)
– ∀ A' ∈ attribut(E) : nom(A') ≠ n

Q – A ∈ attribut(E)
– nom(A) = n

– card(A) = [0-1] ∨ card(A) = [1-1]

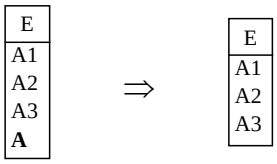
I ∀ i ∈ instance(A) : (card-min(A) = 0 ∧ valeur(i) = null) ∨ (card-min(A) > 0 ∧ valeur(i) = valeur-def)

E On décide d'ajouter un attribut *Numcompte* au type d'entités *FOUNISSEUR* qui représente le numéro de compte du fournisseur (utile pour le règlement des paiements).



b) Destruction

D Détruire un attribut facultatif ou obligatoire d'un type d'entités.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : () ← supprimer-Att(A)

C T^-

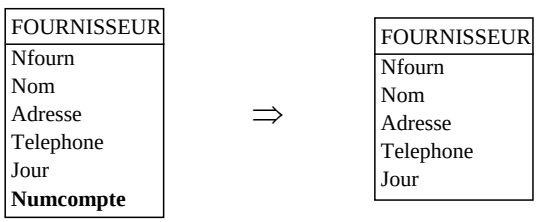
P – E ∈ type-entites(S)

– A ∈ attribut(E)

Q – A ∉ attribut(E)

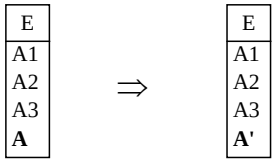
I ∀ i ∈ instance(A) : supprimer(i)

E Détruire l'attribut *Numcompte* du type d'entité *FOURNISSEUR*.



c) Renommage

D Renommer un attribut facultatif ou obligatoire.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : A ← modifier-Att(E,A,n')

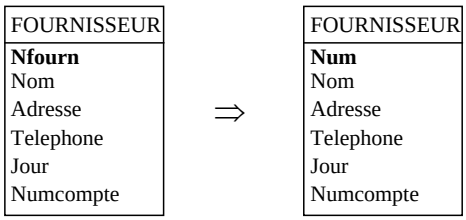
C $T^{=}$

P – E ∈ type-entites(S)
– A ∈ attribut(E)
– nom(A) = n
– ∀ A' ∈ attribut(E) : nom(A') ≠ n'

Q – nom(A) = n'

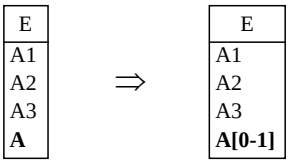
I /

E Renommer l'attribut *Nfourn* du type d'entités *FOURNISSEUR* en *Num*.



d) Changement d'une contrainte obligatoire en facultative

D Changer un attribut obligatoire en facultatif.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : A ← modifier-Att(E,A,[0-1])

C T^{+}

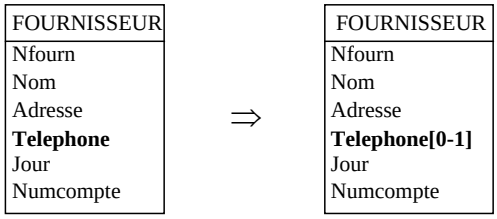
P – E ∈ type-entites(S)
– A ∈ attribut(E)
– card-min(A) = 1
– A ∉ pid(E)

R Tous les attributs d'un identifiant primaire doivent être obligatoires. Si l'on veut modifier un attribut appartenant à un identifiant primaire, il faut soit rendre l'identifiant secondaire, soit enlever l'attribut des composants de l'identifiant.

Q – card-min(A) = 0

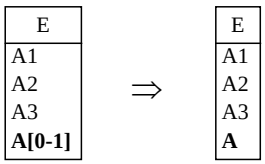
I /

E Faire de l'attribut obligatoire *Telephone* du type d'entité *FOURNISSEUR* un attribut facultatif.



e) Changement d'une contrainte facultative en obligatoire

D Changer un attribut facultatif en attribut obligatoire.



S $\exists$ E $\in$ type-entites(S), A $\in$ attribut(E) : A $\leftarrow$ modifier-Att(E,A,[1-1])

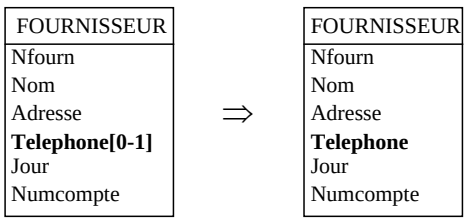
C T^-

P – E $\in$ type-entites(S)
– A $\in$ attribut(E)
– card-min(A) = 0

Q – card-min(A) = 1

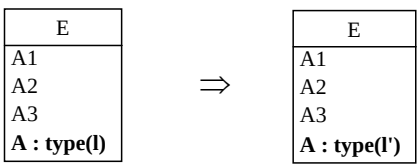
I $\forall$ i $\in$ instance(A) : valeur(i) $\neq$ null

E Rendre obligatoire l'attribut *Telephone* du type d'entités *FOURNISSEUR*.



f) Extension de domaine

D Etendre le domaine d'un attribut.



S $\exists$ E $\in$ type-entites(S), A $\in$ attribut(E) : A $\leftarrow$ modifier-Att(E,A,l')

C T^+

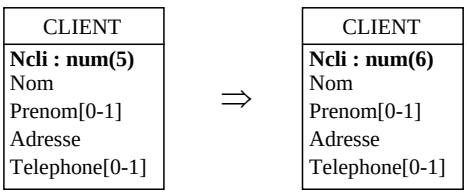
P – E $\in$ type-entites(S)

– A ∈ attribut(E)
– type(A) = car | num | reel | date | bool
– longueur(A) = l

Q – longueur(A) = l'
– l < l'

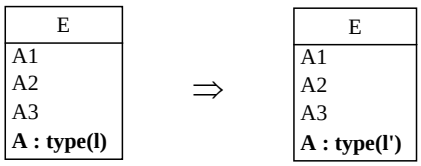
I /

E Etendre le domaine de l'attribut *Ncli* du type d'entité *CLIENT*. Passer de NUMERIC(5) à NUMERIC(6).



g) Restriction de domaine

D Restreindre le domaine du type d'un attribut.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : A ← modifier-Att(E,A,l')

C T^{-}

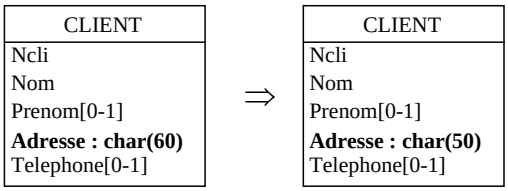
P – E ∈ type-entites(S)
– A ∈ attribut(E)
– type(A) = car | num | reel | date | bool
– longueur(A) = l

Q – longueur(A) = l'
– 0 < l' < l

I ∀ i ∈ instance(A) : longueur(valeur(i)) ≤ l'

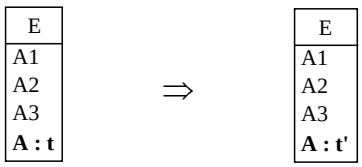
R Les valeurs des instances de A doivent être comprises dans le nouveau domaine de valeurs.

E Restreindre le domaine de l'attribut *Adresse* du type d'entité *CLIENT*.



h) Changement de type

D Changer le type d'un attribut.



S $\exists$ E $\in$ type-entites(S), A $\in$ attribut(E) : A $\leftarrow$ modifier-Att(E,A,t')

C T^+ ou T^-

En fonction du changement de type choisi, la sémantique du schéma augmente ou diminue. Par exemple, si un type numérique est remplacé par un type caractère, le domaine de valeurs de l'attribut augmente alors qu'il diminue si un type caractère est remplacé par un type numérique.

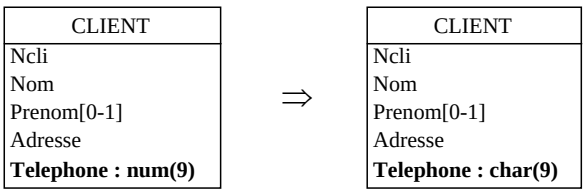
P – E $\in$ type-entites(S)
– A $\in$ attribut(E)
– type(A) = t
– t = car | num | reel | date | bool

Q – type(A) = t'
– t' = car | num | reel | date | bool
– t' $\neq$ t

I $\forall$ i $\in$ instance(A) : convert(valeur(i))

R Les instances de A doivent être converties pour vérifier le nouveau type. La conversion n'est pas toujours possible. Par exemple, la conversion d'un attribut de type caractère en type numérique est impossible si les chaînes de caractères ne représentent pas des nombres.

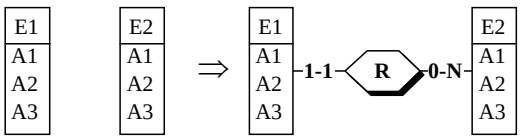
E Changer le type de l'attribut *Telephone* du type d'entité *CLIENT*. Passer du NUMERIC(9) au CHAR(9).



C.2.2.3 Modifications relatives aux types d'associations

a) Création

D Créer un type d'associations et ses rôles.



S R ← creer-TA(S,n)

C T^+

P – $\forall$ R' $\in$ type-associations(S) : nom(R') $\neq$ n

Q – R $\in$ type-associations(S)

– nom(R) = n

– ro1, ro2 $\in$ role(R)

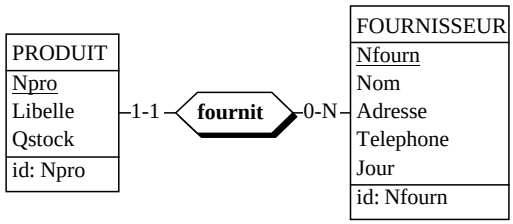
– {E1} = te-role(ro1) $\wedge$ {E2} = te-role(ro2)

– (card(ro1) = [0-1] $\wedge$ card(ro2) = [0-1]) $\vee$ (card(ro1) = [0-1] $\wedge$ card(ro2) = [1-1]) $\vee$ (card(ro1) = [0-1] $\wedge$ card(ro2) = [0-N]) $\vee$ (card(ro1) = [1-1] $\wedge$ card(ro2) = [0-N])

I $\forall$ m $\in$ [1,2] : (($\exists$ rom $\in$ role(R) : card-min(rom) > 0) $\wedge$ ($\forall$ em $\in$ instance(Em), $\exists$ r $\in$ instance(R) : r[rom] = em) $\vee$ instance(R) = $\varnothing$

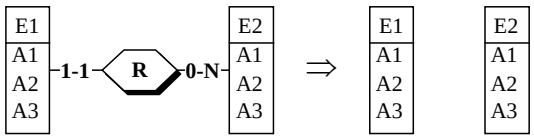
Si un des rôles de R a une cardinalité minimum supérieur à 0, il faut des instances de R pour que la contrainte de cardinalité soit vérifiée.

E On crée un type d'associations *fournit* (0-N/1-1) entre *FOURNISSEUR* et *PRODUIT*.



b) Destruction

D Supprimer un type d'associations et ses rôles.



S $\exists$ R $\in$ type-associations(S) : () ← supprimer-TA(R)

C T^-

P – R $\in$ type-associations(S)

Q – ro1,ro2 $\notin$ role(R)

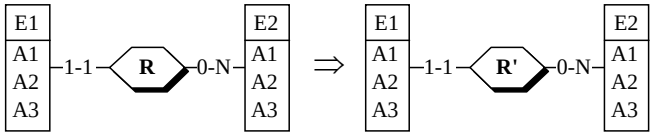
– R $\notin$ type-associations(S)

I $\forall$ i $\in$ instance(R) : supprimer(i)

E On supprime le type d'associations *fournit* entre *PRODUIT* et *FOURNISSEUR*.

c) Renommage

D Renommer un type d'associations.



S ∃ R ∈ type-associations(S) : R ← modifier-TA(R,n')

C $T^{=}$

P – R ∈ type-associations(S)

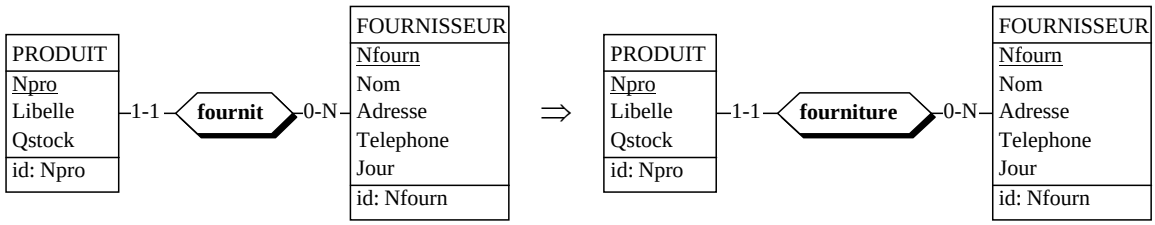
– nom(R) = n

– ∀ R' ∈ type-associations(S) : nom(R') ≠ n'

Q – nom(R) = n'

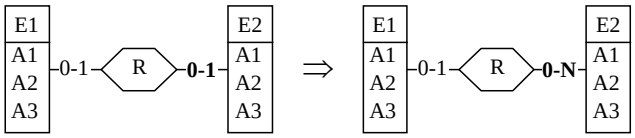
I /

E On renomme le type d'association *fournit* par *fourniture*.



d) Augmentation de la cardinalité maximum d'un rôle

D Augmenter la cardinalité maximum d'un rôle.



S ∃ R ∈ type-associations(S), ro2 ∈ role(R) : ro2 ← modifier-Role(R,ro2,[0-N])

C T^{+}

P – R ∈ type-associations(S)

– {ro1,ro2} = role(R)

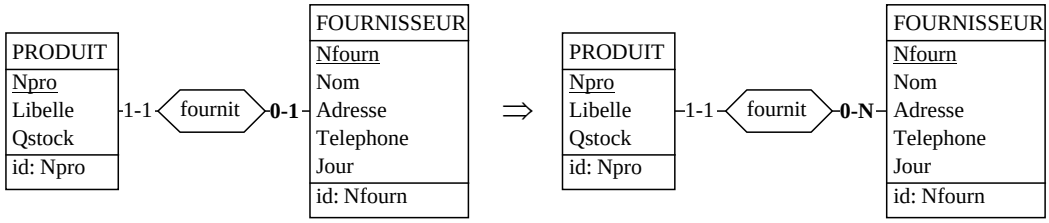
– card(ro1) = [0-1] ∨ card(ro1) = [1-1]

– card(ro2) = [0-1]

Q – card(ro2) = [0-N]

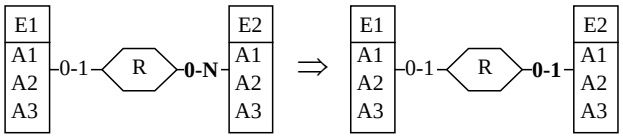
I /

E Supposons que le type d'associations *fournit* soit du type 0-1/1-1. Un fournisseur peut fournir un produit ou aucun alors qu'un produit est fourni par un et un seul fournisseur. On peut augmenter la cardinalité maximum du rôle joué par *FOURNISSEUR* dans *fournit*. Un fournisseur peut donc fournir plusieurs produits.



e) Diminution de la cardinalité maximum d'un rôle

D Diminuer la cardinalité maximum d'un rôle.



S $\exists$ R $\in$ type-associations(S), ro2 $\in$ role(R) : ro2 $\leftarrow$ modifier-Role(R,ro2,[0-1])

C T^-

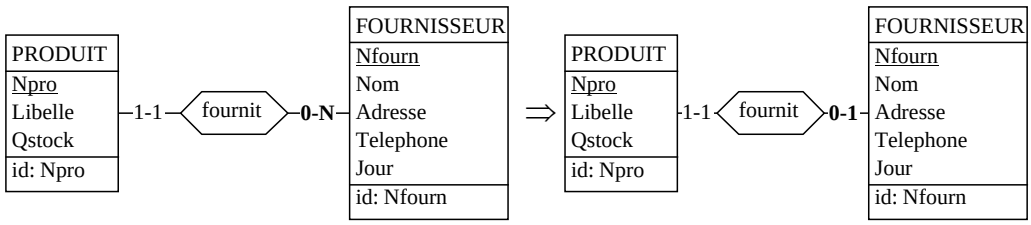
P
- R $\in$ type-associations(S)
- {ro1,ro2} = role(R)
- E1,E2 $\in$ type-entites(S)
- {E1} = te-role(ro1)
- {E2} = te-role(ro2)
- card(ro1) = [0-1] $\vee$ card(ro1) = [1-1]
- card(ro2) = [0-N]
- {E2} = te-role(ro2)
- $\forall$ id $\in$ identifiant(E2) : ro1 $\notin$ composant(id)

Q
- card-max(ro) = 1

I $\forall$ x $\in$ instance(E2), $\exists!$ y $\in$ instance(R) : y[R.E2] = x

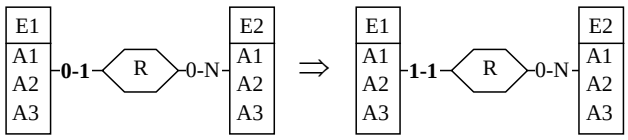
Les instances de E2 participe à une instance de R au maximum.

E On diminue la cardinalité maximum du rôle joué par *FOURNISSEUR* dans *fournit*. La cardinalité du rôle devient 0-1. Un fournisseur peut fournir un produit.



f) Augmentation de la cardinalité minimum d'un rôle

D Augmenter la cardinalité minimum d'un rôle.



S ∃ R ∈ type-associations(S), ro1 ∈ role(R) : ro1 ← modifier-Role(R,ro1,[1-1])

C T^-

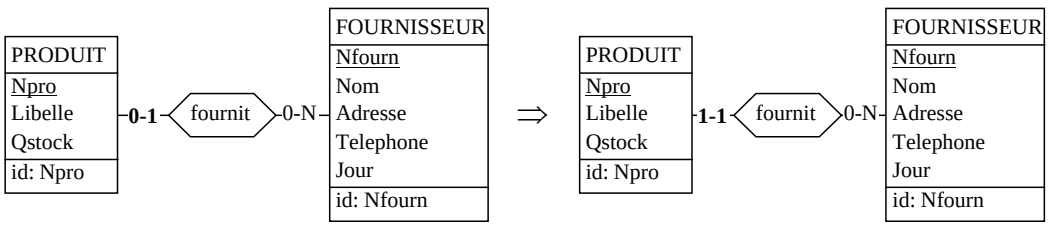
P
- R ∈ type-associations(S)
- {ro1,ro2} ∈ role(R)
- E1,E2 ∈ type-entites(S)
- {E1} = te-role(ro1)
- {E2} = te-role(ro2)
- card(ro1) = [0-1]
- card(ro2) = [0-1] ∨ card(ro2) = [0-N]

Q
- card-min(ro1) = 1

I ∀ x ∈ instance(E1), ∃! y ∈ instance(R) : y[R.E1] = x

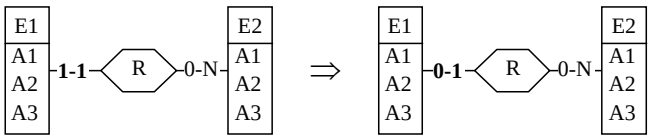
Toutes les instances de E1 participe à une et une seule instance de R.

E Si la cardinalité du rôle joué par *PRODUIT* dans *fournit* est 0-1, on la remplace par une cardinalité 1-1. Un produit est toujours fourni par un fournisseur.



g) Diminution de la cardinalité minimum d'un rôle

D Diminuer la cardinalité minimum d'un rôle.



S ∃ R ∈ type-associations(S), ro1 ∈ role(R) : ro1 ← modifier-Role(R,ro2,[0-1])

C T^+

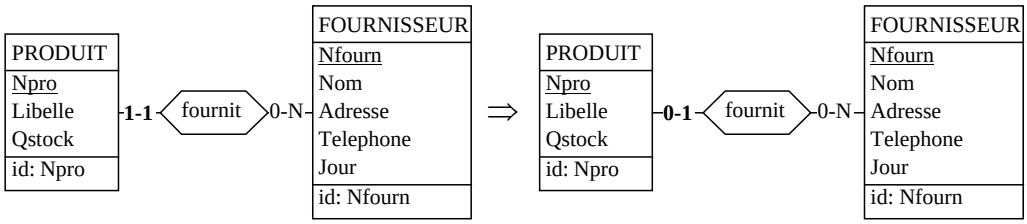
P
- R ∈ type-associations(S)
- {ro1,ro2} ∈ role(R)
- card(ro1) = [1-1]
- card(ro2) = [0-1] ∨ card(ro2) = [0-N]
- E1 ∈ te-role(ro1)

– $\exists$ P $\in$ pid(E1) : ro2 $\notin$ composant(P)

Q – card-min(ro1) = 0

I /

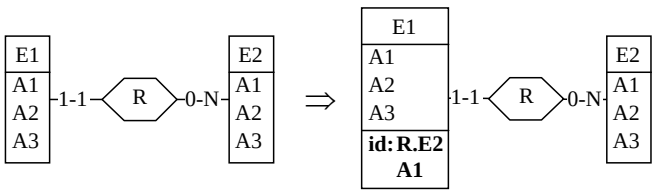
E On change la cardinalité du rôle joué par *PRODUIT* dans *fournit*. Elle devient 0-1.



C.2.2.4 Modifications relatives aux identifiants

a) Création

D Ajouter un identifiant primaire ou secondaire à un type d'entités.



S $\exists$ E1 $\in$ type-entites(S) : id $\leftarrow$ creer-Id(E1,n,id-primaire,{R.E2,A1})

C T^+

P – E1 $\in$ type-entites(S)

– $\forall$ id' $\in$ identifiant(E1) : nom(id') $\neq$ n

– fonction(id) = id-primaire $\wedge$ pid(E1) = $\varnothing$

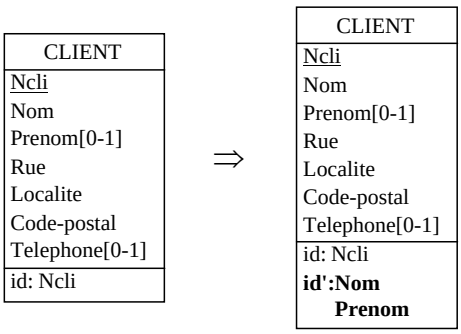
– fonction(id) = id-primaire $\wedge$ ($\forall$ A $\in$ composant(id) : card(A)=[1-1]) $\wedge$ ($\forall$ ro2 $\in$ composant(id) : {ro1,ro2} = role(R) $\wedge$ {E1} = te-role(ro1) $\wedge$ card(ro2)=[0-N] $\wedge$ card(ro1)=[1-1])

Q – id $\in$ identifiant(E1)

– nom(id) = n

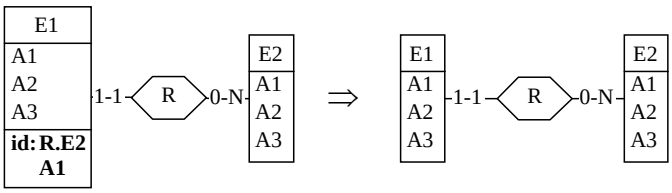
I $\forall$ x,y $\in$ instance(id) : valeur(x) = valeur(y) $\wedge$ x = y
Deux instances distinctes de E1 ne peuvent pas avoir les mêmes valeurs pour l'identifiant id.

E Ajouter un identifiant secondaire au type d'entités *CLIENT* composé des attributs *Nom* et *Prénom*.



b) Destruction

D Détruire un identifiant primaire ou secondaire d'un type d'entités.



S ∃ E1 ∈ type-entites(S), id ∈ identifiant(E1) : () ← supprimer-Id(E1,id)

C T^-

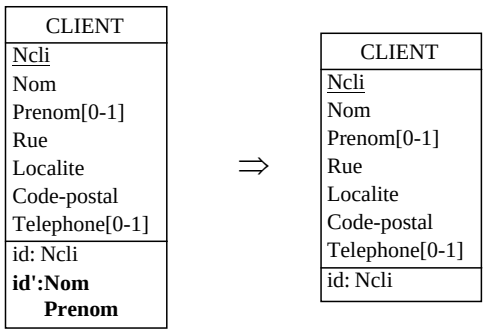
P – E1 ∈ type-entites(S)

– id ∈ identifiant(E1)

Q – id ∉ identifiant(E1)

I /

E Détruire l'identifiant secondaire composé des attributs *Nom* et *Prénom* du type d'entités *CLIENT*.



c) Renommage

D Renommer un identifiant primaire ou secondaire d'un type d'entités.

S ∃ E1 ∈ type-entites(S), id ∈ identifiant(E1) : id ← modifier-Id(E1,id,n')

C $T^=$

P – E1 ∈ type-entites(S)

- id ∈ identifiant(E1)
- nom(id) = n
- ∀ id' ∈ identifiant(E1) : nom(id') ≠ n'

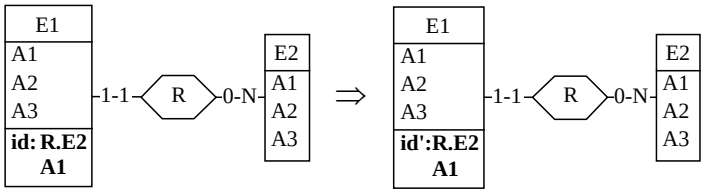
Q – nom(id) = n

I /

E Renommer dans le type d'entités *CLIENT* l'identifiant secondaire composé des attributs *Nom* et *Prénom.* Son nom *Idclient1* devient *Idsecclient*.

d) Changement d'un identifiant primaire en secondaire et inversement

D Changer un identifiant primaire en identifiant secondaire et inversement.



S ∃ E1 ∈ type-entites(S), id ∈ identifiant(E1) : id ← modifier-Id(E1,id,id-secondaire) ∨ id ← modifier-Id(E1,id,id-primaire)

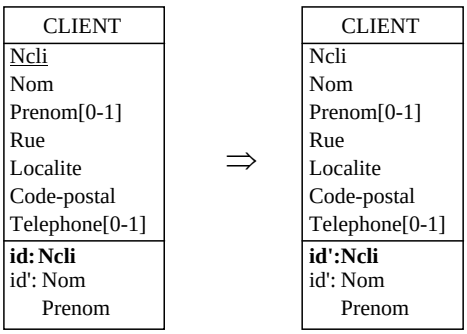
C $T^{=}$

P
- E1 ∈ type-entites(S)
- fonction(id) = id-secondaire ∧ pid(E1) = ∅
- fonction(id) = id-secondaire ∧ (∀ A ∈ composant(id) : card(A)=[1-1]) ∧ (∀ ro2 ∈ composant(id) : {ro1,ro2} = role(R) ∧ {E1} = te-role(ro1) ∧ card(ro2)=[0-N] ∧ card(ro1)=[1--1])

Q – fonction(id) = id-secondaire ∨ fonction(id) = id-primaire

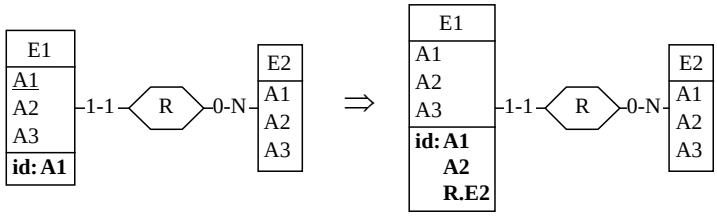
I /

E Changer l'identifiant primaire contenant *Ncli* du type d'entité *CLIENT* en identifiant secondaire.



e) Ajout d'un composant

D Ajouter un attribut ou un rôle à un identifiant d'un type d'entités.



S ∃ E1 ∈ type-entites(S), id ∈ identifiant(E1) : (id,{A1,A2,R.E2}) ← modifier-Id(E1,id,{A2,R.E2})

C T^+

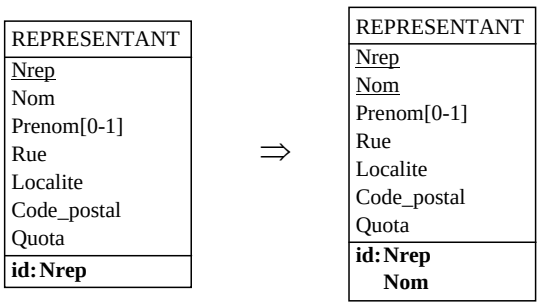
P
- E1 ∈ type-entites(S)
- C ∉ composant(id)
- C ∈ attribut(E1) ∨ ({ro1,C} = role(R) ∧ {E1} = te-role(ro1) ∧ card(C)=[0-N] ∧ card-max(ro1)=1)
- fonction(id) = id-primaire ∧ C ∈ attribut(E1) ∧ card(C)=[1-1]
- fonction(id) = id-primaire ∧ {ro1,C} = role(R) ∧ ({E1} = te-role(ro1) ∧ card(C)=[0-N] ∧ card(ro1)=[1-1]

Q – C ∈ composant(id)

I /

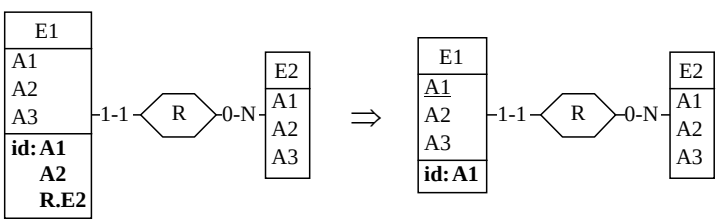
R Ajouter un composant à un identifiant ne modifie pas l'unicité des instances du groupe.

E Supposons que l'attribut *Nrep* du type d'entité *REPRESENTANT* soit relatif à une région. On lui ajoute l'attribut *Nom* pour former l'identifiant primaire. Cela suppose qu'il n'y a pas deux représentants avec le même nom dans une région.



f) Retrait d'un composant

D Enlever un attribut ou un rôle d'un identifiant.



S ∃ E1 ∈ type-entites(S), id ∈ identifiant(E1) :(id,{A1}) ← modifier-Id(E1,id,{A2,R.E2})

C T^-

P – E1 ∈ type-entites(S)

– C ∈ composant(id)

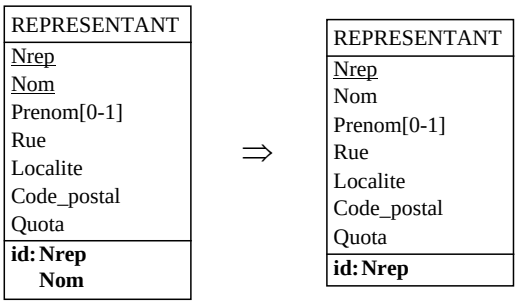
– composant(id) \ {C} ≠ ∅
id a plus d'un composant sinon il s'agit d'une destruction d'identifiant (voir C.2.2.4 b).

Q – C ∉ composant(id)

I ∀ x,y ∈ instance(id) : x = y ∧ valeur(x) = valeur(y)

R Le retrait d'un composant dans un identifiant peut altérer l'unicité des instances de l'identifiant.

E Supposons que les attributs *Nrep* et *Nom* identifie le type d'entité *REPRESENTANT*. Si *Nrep* est suffisant pour identifier le type d'entités, *Nom* est enlevé de l'identifiant primaire.



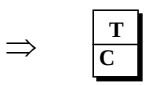
C.2.3 Modifications de spécifications logiques

Pour les tables, les colonnes, les clés primaires et candidates, les exemples sont les mêmes que pour les modifications conceptuelles relatives aux types d'entités, aux attributs et aux identifiants. Les représentations graphiques de ces exemples ne sont pas données. Les lecteurs intéressés doivent consulter les modifications correspondantes dans les sections C.2.2.1 (types d'entités), C.2.2.2 (attributs) et C.2.2.4 (identifiants).

C.2.3.1 Modifications relatives aux tables

a) Création

D Créer une table.



S ∃ T ∈ type-entites(S) : T ← creer-TE(S,n)

C T^+

P – ∀ T' ∈ type-entites(S) : nom(T') ≠ n

Q – T ∈ type-entites(S)

– nom(T) = n

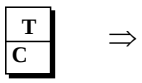
– attribut(T) ≠ ∅

I instance(T) = ∅

E On crée une table *FOURNISSEUR* avec les colonnes Nfourn, *Nom*, *Adresse*, *Telephone* et *Jour*.

b) Destruction

D Détruire une table.



S $\exists$ T $\in$ type-entites(S) : () $\leftarrow$ supprimer-TE(T)

C T^-

P – T $\in$ type-entites(S)

– $\forall$ id $\in$ identifiant(T) : est-reference(id) = $\varnothing$

R La table T n'est référencée par aucune clé étrangère. Les clés étrangères qui référencent la table doivent d'abord être détruites (voir point C.2.3.3 b).

Q – T $\notin$ type-entites(S)

I $\forall$ i $\in$ instance(T) : supprimer(i)

E On détruit la table *FOURNISSEUR*.

c) Renommage

D Renommer une table.

S $\exists$ T $\in$ type-entites(S) : T $\leftarrow$ modifier-TE(T,n')

C $T^=$

P – T $\in$ type-entites(S)

– nom(T) = n

– $\forall$ T' $\in$ type-entites(S) : nom(T') $\neq$ n'

Q – nom(T) = n'

I /

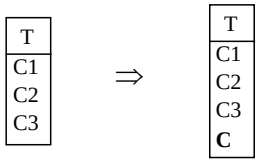
E On renomme la table *FOURNISSEUR* en *GROSSISTE*.

C.2.3.2 Modifications relatives aux colonnes

Dans ce point, seules les colonnes ne participant pas à des clés étrangères sont prises en compte. Les colonnes de référence (participant à des clés étrangères) sont des colonnes dont le but est de créer un lien vers la table référencée. Avec les clés étrangères, elles sont la matérialisation des types d'associations et des rôles dans un modèle logique relationnel. Les préconditions étant plus fortes sur ces colonnes, les modifications les concernant font l'objet du point C.2.3.3.

a) Création

D Créer une colonne facultative ou obligatoire dans une table.



S ∃ T ∈ type-entites(S) : C ← creer-Att(T,n)

C T^+

P – T ∈ type-entites(S)

– ∀ C' ∈ attribut(T) : nom(C') ≠ n

Q – C ∈ attribut(E)

– nom(C) = n

– card(C) = [0-1] ∨ card(C) = [1-1]

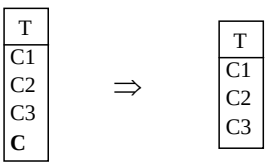
R n est conforme à la norme SQL-92.

I ∀ i ∈ instance(C) : (card-min(C) = 0 ∧ valeur(i) = null) ∨ (card-min(C) = 1 ∧ valeur(i) = valeur-def)

E On décide d'ajouter une colonne *Numcompte* à la table *FOUNISSEUR*.

b) Destruction

D Détruire une colonne facultative ou obligatoire d'une table.



S ∃ T ∈ type-entites(S), C ∈ attribut(T) : () ← supprimer-Att(T,C)

C T^-

P – T ∈ type-entites(S)

– C ∈ attribut(T)

– ∀ fk ∈ reference(T) : C ∉ composant(fk)

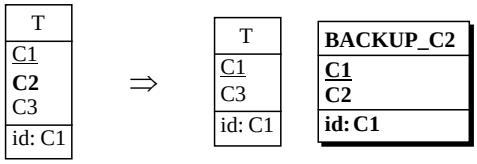
– ∀ id ∈ identifiant(T) : C ∉ composant(id)

R Il faut d'abord enlever la colonne des clés primaires et candidates auxquelles elle participe ou détruire ces clés avant de détruire la colonne (voir les points C.2.3.4 b) et f).

Q – C ∉ attribut(T)

I ∀ i ∈ instance(A) : supprimer(i)

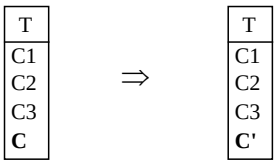
Les instances de la colonne détruite sont perdues. Elles peuvent être conservées dans une table de sauvegarde. Pour assurer une éventuelle exploitation de ces sauvegardes, les colonnes identifiantes de la table d'origine sont ajoutées à la table de sauvegarde. Si la table originale n'a pas de clé primaire ou candidate, la table de sauvegarde est une copie de la table d'origine.



E Détruire la colonne *Numcompte* de la table *FOURNISSEUR*.

c) Renommage

D Renommer une colonne facultative ou obligatoire d'une table.



S $\exists$ T $\in$ type-entites(S), C $\in$ attribut(T) : C $\leftarrow$ modifier-Att(T,C,n')

C $T^{=}$

P – T $\in$ type-entites(S)
– C $\in$ attribut(T)
– nom(C) = n
– $\forall$ C' $\in$ attribut(T) : nom(C') $\neq$ n'

Q – nom(A) = n'

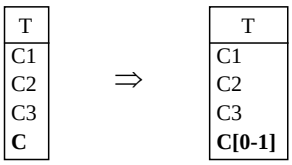
R n' est conforme à la norme SQL-92.

I /

E Renommer la colonne *Nfourn* de la table *FOURNISSEUR* en *Num*.

d) Changement d'une contrainte NOT NULL en NULL

D Enlever la contrainte obligatoire d'une colonne.



S $\exists$ T $\in$ type-entites(S), C $\in$ attribut(T) : C $\leftarrow$ modifier-Att(T,C,n',0-1)

C T^{+}

P – T $\in$ type-entites(S)
– C $\in$ attribut(T)
– card-min(C) = 1
– $\forall$ P $\in$ pid(E) : C $\notin$ composant(P)
– $\forall$ fk $\in$ reference(T) : C $\notin$ composant(fk)

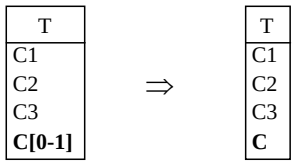
Q – card-min(C) = 0

I /

E La colonne *Telephone* de la table *FOURNISSEUR* devient facultative.

e) Changement d'une contrainte NULL en NOT NULL

D Ajouter la contrainte obligatoire à une colonne facultative.



S ∃ T ∈ type-entites(S), C ∈ attribut(T) : C ← modifier-Att(T,C,1-1)

C T^-

P – T ∈ type-entites(S)
– C ∈ attribut(T)
– card-min(C) = 0
– ∀ fk ∈ reference(T) : C ∉ composant(fk)

Q – card-min(C) = 1

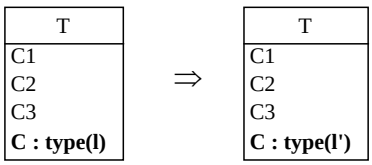
I ∀ i ∈ instance(C), valeur(i) ≠ null

Les instances de la colonne obligatoire doivent avoir une valeur. Dans le cas contraire, trois solutions sont envisageables. Premièrement, les instances illicites sont enlevées de la table d'origine et stockées dans une table annexe avant de modifier la colonne. Deuxièmement, les instances ayant une valeur nulle sont remplacées par une valeur par défaut (cette solution est possible si la colonne n'appartient pas à une contrainte identifiante). Troisièmement, la modification est reportée tant que le problème n'est pas résolu.

E *Telephone* de la table *FOURNISSEUR* devient une colonne obligatoire.

f) Extension de domaine

D Etendre le domaine de valeurs d'une colonne.



S ∃ T ∈ type-entites(S), C ∈ attribut(T) : C ← modifier-Att(T,C,l')

C T^+

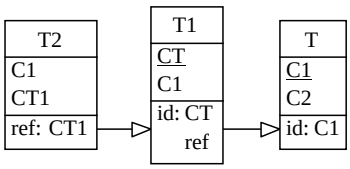
P – T ∈ type-entites(S)
– C ∈ attribut(T)
– type(C) = car | num | reel
– longueur(C) = l
– ∀ fk ∈ reference(T) : C ∉ composant(fk)

Q – longueur(C) = l'

– l < l'
  l doit respecter la limite maximum autorisée par la norme SQL-92.

I /

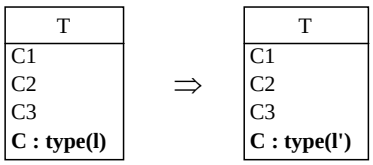
R Si C appartient à une clé primaire ou candidate référencée par des clés étrangères, alors la modification doit être propagée sur les colonnes qui référencent cette clé primaire ou candidate. Cette propagation peut être en cascade si, à leur tour, les colonnes de référence sont référencées. Dans l'exemple ci-dessus, l'extension du domaine de C1 entraîne l'extension des domaines de CT et CT1.



E Etendre le domaine de la colonne *Ncli* de la table *CLIENT*. Passer de NUMERIC(5) à NUMERIC(6).

g) Restriction de domaine

D Restreindre le domaine de valeurs d'une colonne.



S ∃ T ∈ type-entites(S), C ∈ attribut(T) : C ← modifier-Att(T,C,l')

C T^{-}

P – E ∈ type-entites(S)
  – A ∈ attribut(E)
  – type(A) = car | num | reel
  – longueur(A) = l
  – ∀ fk ∈ reference(T) : C ∉ composant(fk)

Q – longueur(C) = l'
  – 0 < l' < l

I ∀ i ∈ instance(C), longueur(valeur(i)) ≤ l'

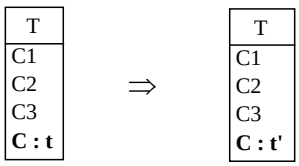
  Les valeurs des instances de C doivent être comprises dans le nouveau domaine de valeurs. Dans le cas contraire, trois solutions sont possibles. Premièrement, la modification est quand même appliquée. Dans ce cas, les valeurs trop grandes seront tronquées. Deuxièmement, les instances illicites sont transférées dans une table annexe avant la modification de la colonne. Troisièmement, la modification est rejetée (c'est le cas lorsque la colonne appartient à une clé étrangère, primaire ou candidate et que les valeurs tronquées entraînent une violation de ces contraintes).

R Comme pour l'extension du domaine, la restriction du domaine d'une colonne peut impliquer la propagation en cascade de la modification vers les colonnes qui référencent la colonne modifiée.

E Restreindre le domaine de la colonne *Adresse* de la table *CLIENT*. Passer de CHAR(60) à CHAR(50).

h) Changement de type

D Changer le type d'un colonne.



S ∃ T ∈ type-entites(S), C ∈ attribut(T) : C ← modifier-Att(T,C,t')

C T^+ ou T^-

P
- T ∈ type-entites(S)
- C ∈ attribut(T)
- type(C) = t
- t = car | num | reel | date | bool
- ∀ fk ∈ reference(T) : C ∉ composant(fk)

Q
- type(C) = t'
- t' = car | num | reel | date | bool
- t' ≠ t

I ∀ i ∈ instance(C), convert(valeur(i))

Les instances de C doivent être converties pour vérifier le nouveau type. La conversion n'est pas toujours possible. Par exemple, la conversion d'une colonne de type caractère en type numérique est impossible si les chaînes de caractères ne représentent pas des nombres.

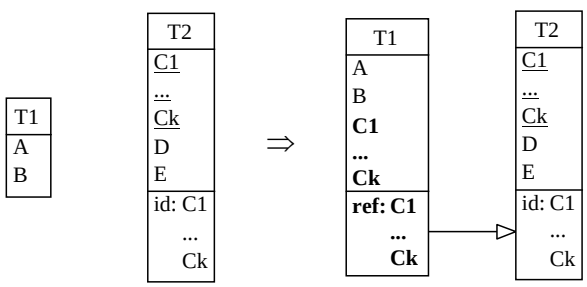
R Comme pour l'extension et la restriction du domaine, le changement de type d'une colonne référencée implique la propagation en cascade de la modification vers les colonnes qui la référencent.

E Changer le type de la colonne *Telephone* de la table *CLIENT*. Passer du NUMERIC(9) au CHAR(9).

C.2.3.3 Modifications relatives aux clés étrangères

a) Création

D Créer un ou plusieurs colonnes de référence dans la table de référence ainsi qu'une clé étrangère entre ces colonnes et la clé primaire ou candidate référencée.



S $\exists$ T1,T2 $\in$ type-entites(S) : C1 $\leftarrow$ creer-Att(T1,n1) $\wedge$ ... $\wedge$ Ck $\leftarrow$ creer-Att(T1,nk) $\wedge$ fk $\leftarrow$ creer-FK(T1,n,{C1,...,Ck},{T2.C1,...,T2.Ck})

C T^+

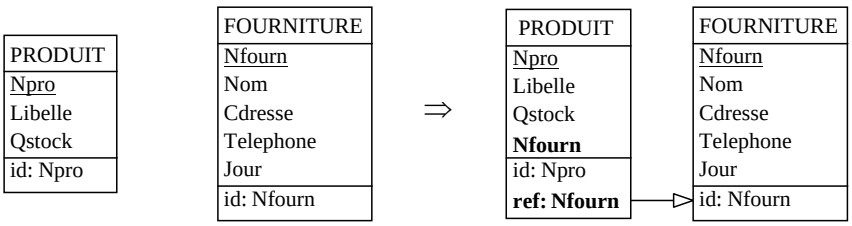
P
- T1,T2 $\in$ type-entites(S)
- identifiant(T2) $\neq \varnothing$
- $\forall$ fk' $\in$ reference(T1) : nom(fk') $\neq$ n
- $\forall$ C $\in$ attribut(T1) : nom(C) $\neq$ n1 $\wedge$ ... $\wedge$ nom(C) $\neq$ nk

Q
- fk $\in$ reference(T1)
- C1,...,Ck $\in$ attribut(T1)
- nom(fk) = n
- nom(C1) = n1 $\wedge$ ... $\wedge$ nom(Ck) = nk

I $\forall$ f $\in$ instance(fk), $\exists$ i $\in$ instance(id) : valeur(f) = valeur(i)

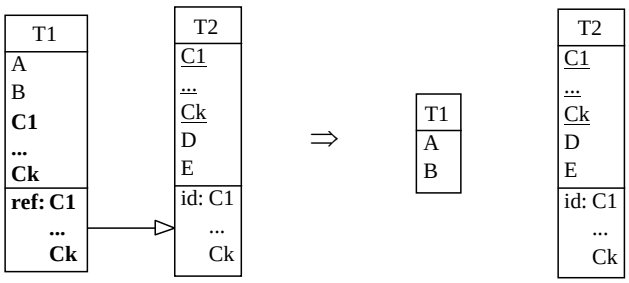
Les instances des colonnes de références doivent avoir les mêmes valeurs qu'une instance des colonnes identifiantes. L'ingénieur doit mettre les données pour que cette contrainte soit vérifiée.

E On crée dans la table *PRODUIT* une colonne obligatoire *Nfourn* qui référence la clé primaire *Nfourn* de la table *FOURNISSEUR*.



b) Destruction

D Supprimer une clé étrangère ainsi que les colonnes de référence qui lui appartiennent.



S $\exists$ C1,...,Ck $\in$ attribut(T1), fk $\in$ reference(T1) : () $\leftarrow$ supprimer-Att(T1,C1) $\wedge$ ... $\wedge$ () $\leftarrow$ supprimer-Att(T1,Ck) $\wedge$ () $\leftarrow$ supprimer-FK(T1,fk)

C T^-

P
- T1 $\in$ type-entites(S)
- C1,...,Ck $\in$ attribut(T1)
- fk $\in$ reference(T1)

– $\forall$ id $\in$ identifiant(T1) : est-reference(id) $\neq \emptyset \wedge$ C1 $\notin$ composant(id) $\wedge$ ... $\wedge$ Ck $\notin$ composant(id)
C1, ..., Ck ne peuvent faire partie d'une clé primaire ou candidate référencée par des clés étrangères. Si c'est le cas, il faut d'abord supprimer ces clés.

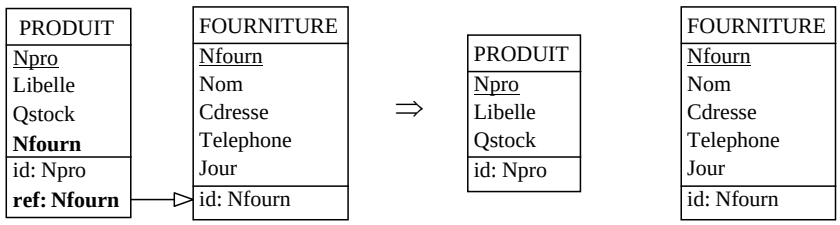
Q – C1,...,Ck $\notin$ attribut(T1)

– fk $\notin$ reference(T)

I $\forall$ i $\in$ instance(C1) : supprimer(i), ..., $\forall$ i $\in$ instance(Ck) : supprimer(i)

Les instances de C1, ..., Ck sont perdues. Elles peuvent être sauvegardées dans une table temporaire pour une exploitation ultérieure (voir point C.2.5.2 b).

E On détruit la clé étrangère et sa colonne de référence *Nfourn* dans la table *PRODUIT*.



c) Renommage

D Renommer une clé étrangère.

S $\exists$ fk $\in$ reference(T) : fk $\leftarrow$ modifier-FK(T1,fk,n')

C $T^{=}$

P – T1 $\in$ type-entites(S)

– fk $\in$ reference(T1)

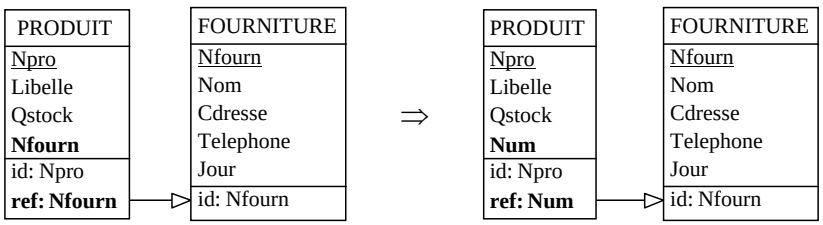
– nom(fk) = n

– $\forall$ fk' $\in$ reference(T1) : nom(fk') $\neq$ n'

Q – nom(fk) = n'

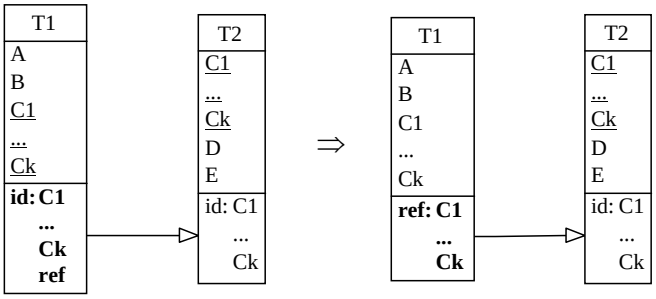
I /

E Dans *PRODUIT*, on renomme la colonne *Nfourn* en *Num*.



d) Destruction de la clé primaire définie sur les colonnes de référence

D Détruire la clé primaire contenant les colonnes de référence.



S ∃ fk∈ reference(T1) : fk ← modifier-FK(T1,fk,∧id-primaire)

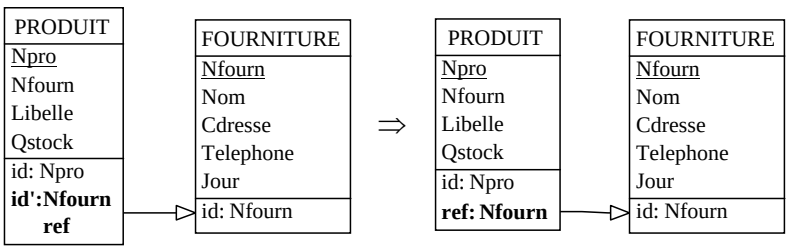
C T^+

P
- T1 ∈ type-entites(S)
- C1,...,Ck ∈ attribut(T1)
- fk ∈ reference(T1)
- {C1,...,Ck} = composant(fk)
- fk ∈ identifiant(T1)
- est-reference(fk) = ∅
  fk ne peut être une clé primaire référencée par des clés étrangères. Si c'est le cas, la suppression de la clé primaire rend le schéma incohérent puisqu'une clé étrangère référence toujours un identifiant.

Q – fk ∉ identifiant(T1)

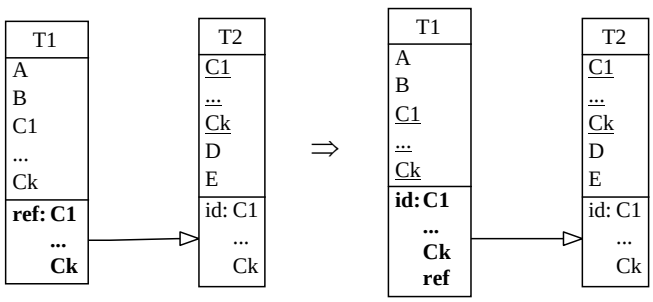
I /

E On détruit la clé candidate contenant *Nfourn* dans *PRODUIT*.



e) Création d'une clé primaire sur les colonnes de référence

D Ajouter un identifant formé des colonnes de référence.



S ∃ fk ∈ reference(T1) : fk ← modifier-FK(T1,fk,id-primaire)

C T-

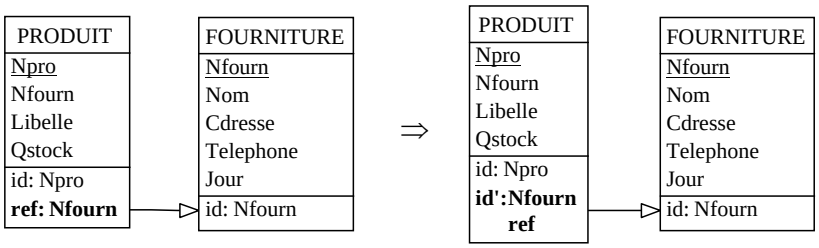
P – T1 ∈ type-entites(S)
– C1,...,Ck ∈ attribut(T1)
– fk ∈ reference(T1)
– {C1,...,Ck} = composant(fk)
– fk ∉ identifiant(T1)

Q – fk ∈ identifiant(T1)

I ∀ x,y ∈ instance(fk) : valeur(x) = valeur(y) ∧ x = y

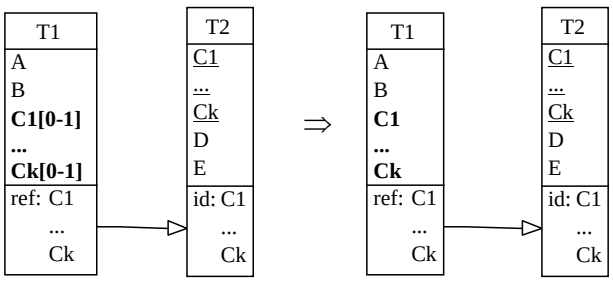
Deux instances de T1 ne peuvent pas avoir les mêmes valeurs pour l'identifiant fk. Les solutions pour résoudre ce problème sont décrites dans la création d'une clé primaire ou candidate au point C.2.3.4 a).

E Ajouter au type d'entités *PRODUIT* une clé candidate composée de la colonne de référence *Nfourn*.



f) Changement des contraintes NULL définies sur les colonnes de référence en NOT NULL

D Les colonnes de référence deviennent obligatoires.



S ∃ T1 ∈ type-entites(S), C1,...,Ck ∈ attribut(T1) : C1 ← modifier-Att(T1,C1,1-1) ∧ ... ∧ Ck ← modifier-Att(T1,Ck,1-1)

C T-

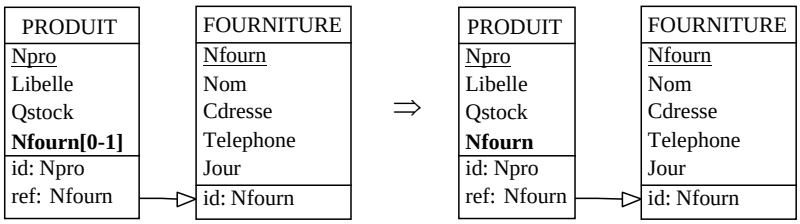
P – T1 ∈ type-entites(S)
– C1,...,Ck ∈ attribut(T1)
– fk ∈ reference(T1)
– {C1,...,Ck} = composant(fk)
– card(C1) = [0-1] ∧ ... ∧ card(Ck) = [0-1]

Q – card(C1) = [1-1] ∧ ... ∧ card(Ck) = [1-1]

I $\forall$ i1 $\in$ instance(C1), ..., $\forall$ ik $\in$ instance(Ck) : valeur(i1) $\neq$ null $\wedge$ ... $\wedge$ valeur(ik) $\neq$ null

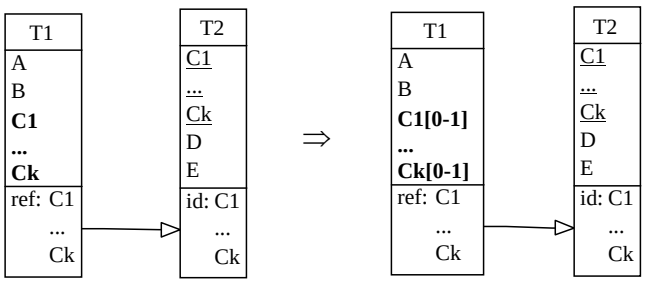
Les instances de C1, ..., Ck doivent avoir une valeur. Ce problème est résolu dans le changement d'une colonne NULL en colonne NOT NULL au point C.2.3.2 e).

E La colonne *Nfourn* de la table *PRODUIT* devient obligatoire.



g) Changement des contraintes NOT NULL définies sur les colonnes de référence en NULL

D Les colonnes de référence deviennent facultatives.



S $\exists$ T1 $\in$ type-entites(S), C1,...,Ck $\in$ attribut(T1) : C1 $\leftarrow$ modifier-Att(T1,C1,0-1) $\wedge$ ... $\wedge$ Ck $\leftarrow$ modifier-Att(T1,Ck,0-1)

C T^+

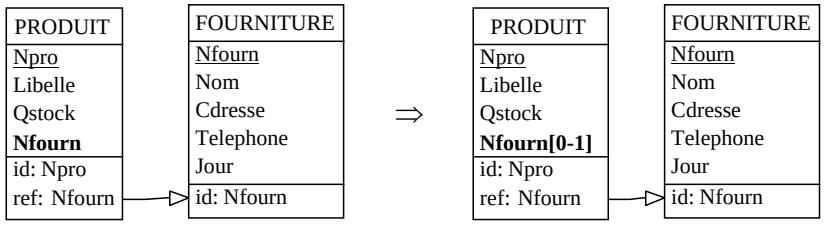
P
- T1 $\in$ type-entites(S)
- C1,...,Ck $\in$ attribut(T1)
- fk $\in$ reference(T1)
- {C1,...,Ck} = composant(fk)
- card(C1) = [1-1] $\wedge$ ... $\wedge$ card(Ck) = [1-1]

Q
- card(C1) = [0-1] $\wedge$ ... $\wedge$ card(Ck) = [0-1]
- fk $\in$ identifiant(T1) $\wedge$ type(fk) = id-secondaire

R Certains SGBDR exigent que les colonnes référencées par une clé étrangère appartiennent à une clé primaire. Dans ce cas, les colonnes de référence ne peuvent pas être référencées par des clés étrangères.

I /

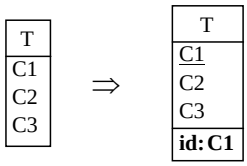
E La colonne *Nfourn* de la table *PRODUIT* devient facultative.



C.2.3.4 Modifications relatives aux clés primaires et candidates

a) Création

D Ajouter une clé primaire ou candidate dans une table.



S $\exists$ T $\in$ type-entites(S), C1 $\in$ attribut(T) : id $\leftarrow$ creer-Id(T1,n,{C1})

C T^+

P
- T $\in$ type-entites(S)
- $\forall$ id' $\in$ identifiant(T) : nom(id') $\neq$ n
- fonction(id) = id-primaire $\wedge$ pid(E1) = $\varnothing$
- fonction(id) = id-primaire $\wedge$ $\forall$ C $\in$ composant(id) : card(C)=[1-1]

Q
- id $\in$ identifiant(T)
- nom(id) = n

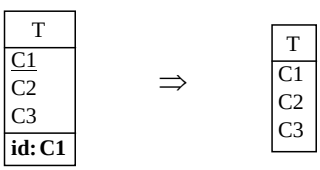
I $\forall$ x,y $\in$ instance(id) : valeur(x) = valeur(y) $\wedge$ x = y

Deux instances de T ne peuvent pas avoir les mêmes valeurs pour l'identifiant id. Si c'est le cas, l'ingénieur a le choix entre trois solutions : soit il refuse d'appliquer la modification, soit il supprime les instances qui ne vérifient pas l'identifiant (avec une sauvegarde éventuelle des instances illicites dans une table temporaire), soit il corrige les instances pour qu'elles vérifient la contrainte d'unicité.

E Ajouter une clé candidate à la table *CLIENT* qui contient les colonnes *Nom* et *Prenom*.

b) Destruction

D Enlever une clé primaire ou candidate d'une table.



S $\exists$ T $\in$ type-entites(S), id $\in$ identifiant(T) : () $\leftarrow$ supprimer-Id(T1,id)

C T^-

P – $T \in$ type-entites(S)
– id $\in$ identifiant(E1)
– est-reference(id) = $\varnothing$

R id n'est référencé par aucune clé étrangère.

Q – id $\notin$ identifiant(E1)

I /

E Détruire la clé candidate contenant les colonnes *Nom* et *Prenom* de la table *CLIENT*.

c) Renommage

D Renommer une clé primaire ou candidate d'une table.

S $\exists$ T $\in$ type-entites(S), id $\in$ identifiant(T) : id $\leftarrow$ modifier-Id(T,id,n')

C $T^{=}$

P – $T \in$ type-entites(S)
– id $\in$ identifiant(T)
– nom(id) = n
– $\forall$ id' $\in$ identifiant(T) : nom(id') $\neq$ n'

Q – nom(id) = n

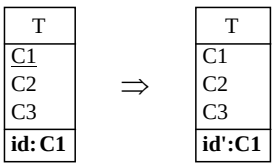
R n est conforme à la norme SQL-92.

I /

E Renommer la clé primaire *Idclient1* concernant les colonnes *Nom* et *Prenom* de la table *CLIENT* en *Idsecclient*.

d) Changement d'une clé primaire en candidate et inversement

D Changer une clé primaire en candidate et inversement.



S $\exists$ T $\in$ type-entites(S), id $\in$ identifiant(T) : id $\leftarrow$ modifier-Id(T,id,id-secondaire) $\vee$ id $\leftarrow$ modifier-Id(T,id,id-primaire)

C $T^{=}$

P – $T \in$ type-entites(S)
– fonction(id) = id-secondaire $\wedge$ pid(T) = $\varnothing$
– fonction(id) = id-secondaire $\wedge$ ($\forall$ C $\in$ composant(id) : card(C) = [1-1])

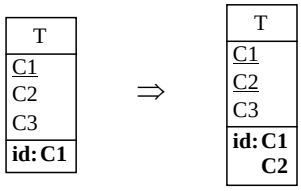
Q – fonction(id) = id-secondaire $\vee$ fonction(id) = id-primaire

I /

E Changer la clé primaire sur *Ncli* de la table *CLIENT* en une clé candidate.

e) Ajout d'un composant

D Ajouter une colonne à une clé primaire ou candidate d'une table.



S ∃ T ∈ type-entites(S), id ∈ identifiant(T), C2 ∈ attribut(T) : (id,{C1,C2}) ← modifier-Id(T,id,{C2})

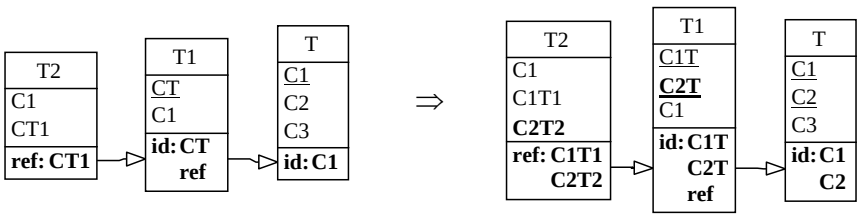
C T^+

P
- T ∈ type-entites(S)
- id ∈ identifiant(T)
- C2 ∈ attribut(T)
- fonction(id) = id-primaire ∧ card(C2)=[1-1]
- id ∉ reference(T)

Q
- C2 ∈ composant(id)

I / L'ajout d'un composant à id ne viole pas l'unicité des instances de T.

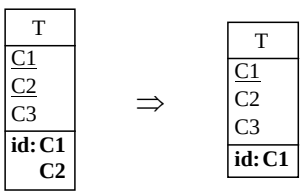
R Si id est référencée par des clés étrangères, alors la modification doit être propagée dans les colonnes de référence de ces clés. Cette propagation peut être en cascade si, à leur tour, les colonnes de référence sont référencées. Dans l'exemple ci-dessus, l'ajout de C2 à la clé primaire de T entraîne des modifications dans T1 et T2.



E Ajouter la colonne *Nom* à la clé primaire de la table *REPRESENTANT*.

f) Retrait d'un composant

D Enlever une colonne d'une clé primaire ou candidate d'une table.



S ∃ T ∈ type-entites(S), id ∈ identifiant(T), C2 ∈ attribut(T) : (id,{C1}) ← modifier-Id(T,id,{C2})

C T^-

P
- T ∈ type-entites(S)
- C2 ∈ attribut(T)

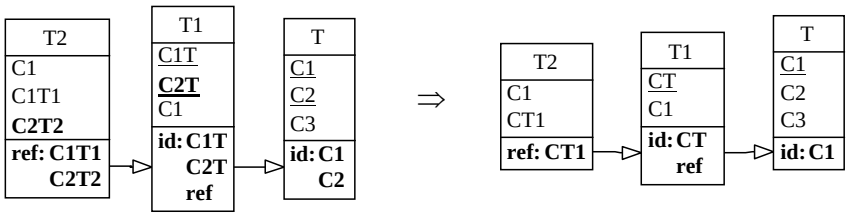
– composant(id) \ {C2} ≠ ∅
  id a plus d'un composant sinon il s'agit d'une destruction de clé primaire ou candidate (voir point C.2.3.4 b).
– id ∉ reference(T)

Q – C2 ∉ composant(id)

I ∀ x,y ∈ instance(id) : valeur(x) = valeur(y) ∧ x = y

Le retrait de C2 de id peut altérer l'unicité des instances de la table T.

R Si id est référencée par des clés étrangères, alors la modification doit être propagée dans les colonnes de référence de ces clés. Cette propagation peut être en cascade si, à leur tour, les colonnes de référence sont référencées. Dans l'exemple ci-dessus, le retrait de C2 de la clé primaire de T entraîne des modifications dans T1 et T2.



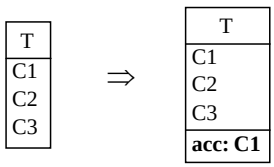
E Enlever la colonne *Nom* de la clé primaire de la table *REPRESENTANT*.

C.2.4 Modifications de spécifications physiques

C.2.4.1 Modifications relatives aux index

a) Création

D Ajouter un index.



S ∃ T ∈ type-entites(S) : acc ← creer-Index(T,n,{C1})

C T^+

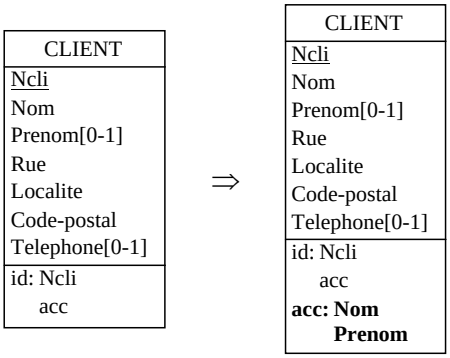
P – T ∈ type-entites(S)
– ∀ acc' ∈ cle-acces(T) : nom(acc') ≠ n

Q – acc ∈ cle-acces(T)
– nom(acc) = n
– ∃ C ∈ attribut(T) : C ∈ composant(acc)

R Il est conseillé que l'index créé ne soit pas préfixe d'un index existant ou inversement. Dans ce cas, l'index préfixe est inutile.

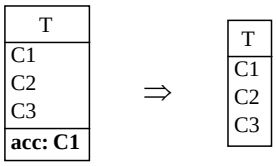
I /

E Ajouter un index sur les colonnes *Nom* et *Prenom* de la table *CLIENT*.



b) Destruction

D Détruire un index.



S ∃ T ∈ type-entites(S), acc ∈ cle-acces(T) : () ← supprimer-Index(T,acc)

C T^-

P – T ∈ type-entites(S)

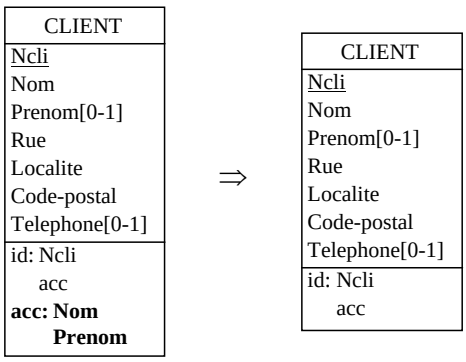
– acc ∈ cle-acces(T)

Q – acc ∉ cle-acces(T)

I /

R La destruction de acc peut détériorer les performances des accès à la base de données.

E Détruire l'index défini sur les colonnes *Nom* et *Prenom* de la table *CLIENT*.



c) Renommage

D Renommer un index.

S ∃ T ∈ type-entites(S), acc ∈ cle-acces(T) : acc ← modifier-Index(T,acc,n')

C T^+

P – T ∈ type-entites(S)
– acc ∈ cle-acces(T)
– nom(acc) = n
– ∀ acc' ∈ cle-acces(T) : nom(acc') ≠ n'

Q – nom(acc) = n'

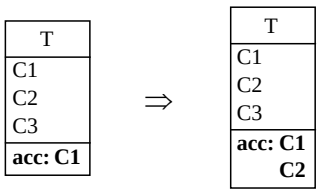
R n' est conforme à la norme SQL-92.

I /

E Renommer l'index *Idx_client* sur les colonnes *Nom* et *Prenom* de la table *CLIENT* en *Idx_nomcli*.

d) Ajout d'un composant

D Ajouter un composant à un index.



S ∃ T ∈ type-entites(S), acc ∈ cle-acces(T), C2 ∈ attribut(T) : (acc,{C1,C2}) ← modifier-Index(T,acc,{C2})

C $T^{=}$

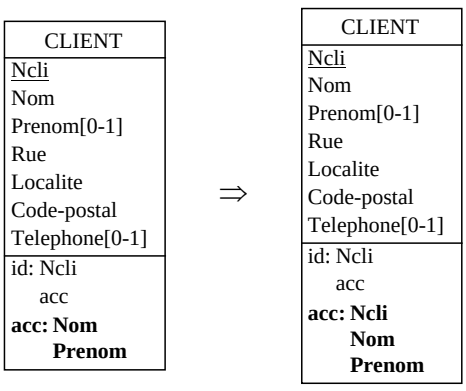
P – T ∈ type-entites(S)
– acc ∈ cle-acces(T)
– C2 ∈ attribut(T)

Q – C2 ∈ composant(acc)

I /

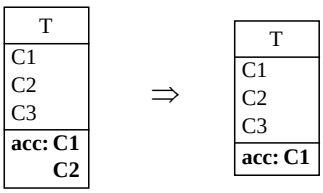
R Il est conseillé que l'index modifié ne soit pas préfixe d'un index existant ou inversement. Par exemple, les index (B,C) et (A,B) deviennent après ajout de A à (B,C) : (A,B,C) et (A,B). (A,B) est préfixe de (A,B,C). Dans ce cas, l'ajout d'un composant s'accompagnera de la destruction d'un index devenu inutile.

E Ajouter la colonne *Ncli* à l'index composé des colonnes *Nom* et *Prenom* de la table *CLIENT*.



e) Retrait d'un composant

D Enlever un composant d'un index.



S ∃ T ∈ type-entites(S), acc ∈ cle-acces(T), C2 ∈ attribut(T) : (acc,{C1}) ← modifier-Index(T,acc,{C2})

C T^-

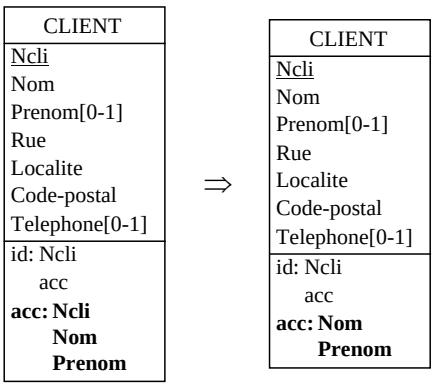
P
- T ∈ type-entites(S)
- C2 ∈ attribut(T)
- composant(acc) \ {C2} ≠ ∅
  acc a plus d'un composant sinon il s'agit d'une destruction d'index (voir C.2.4.1 b).

Q
- C2 ∉ composant(acc)

I /

R Il est conseillé que l'index modifié ne soit pas préfixe d'un index existant ou inversement. Par exemple, les index (A,B,C) et (B) deviennent après retrait de A à (A,B,C) : (B,C) et (B). (B) est préfixe de (B,C). Dans ces cas, le retrait s'accompagnera d'une destruction d'un index devenu inutile.

E Enlever la colonne *Ncli* de l'index contenant les colonnes *Ncli*, *Nom* et *Prenom* de la table *CLIENT*.



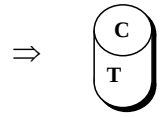
f) Remarques importantes

Les modifications relatives aux index sont des opérations qui peuvent être très lourdes (en temps, accès disques, ...) pour le système dans le cas de grosses bases de données. Il faut donc les réaliser avec beaucoup de prudence.

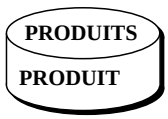
C.2.4.2 Modifications relatives aux espace de stockage

a) Création

D Créer un espace de stockage.

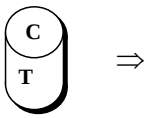


S ∃ C ∈ collection(S) : C ← creer-Coll(S,n)
C T^+
P – ∀ C' ∈ collection(S) : nom(C') ≠ n
Q – C ∈ collection(S)
– nom(C) = n
– te-coll(C) ≠ ∅
I instance(C) ≠ ∅
E Créer un espace de stockage *PRODUITS* contenant la table *PRODUIT*.



b) Destruction

D Détruire un espace de stockage.



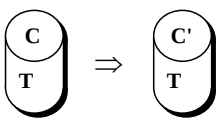
S ∃ C ∈ collection(S) : () ← supprimer-Coll(S,C)
C T^-
P – C ∈ collection(S)
Q – C ∉ collection(S)
I ∀ i ∈ instance(C) : supprimer(i)

Les instances des tables contenues dans l'espace de stockage sont perdues. Il faut les extraire avant de détruire l'espace de stockage.

E Détruire l'espace de stockage *PRODUITS* contenant la table *PRODUIT*.

c) Renommage

D Renommer un espace de stockage.



S ∃ C ∈ collection(S) : C ← modifier-Coll(S,C,n')
C $T^=$

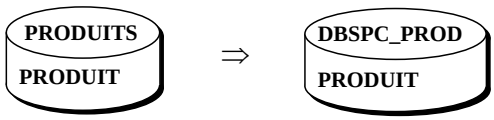
P – $C \in$ collection(S)
– nom(C) = n
– $\forall$ C' $\in$ collection(S) : nom(C') $\neq$ n'

Q – nom(C) = n'

R n' est conforme à la norme SQL-92.

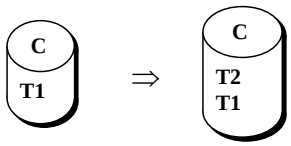
I /

E Renommer l'espace de stockage *PRODUITS* contenant la table *PRODUIT* en *DBSPC_PROD*.



d) Ajout d'une table

D Ajouter une table à un espace de stockage.



S $\exists$ C $\in$ collection(S), T2 $\in$ type-entites(S) : (C,{T1,T2}) $\leftarrow$ modifier-Coll(S,C,{T2})

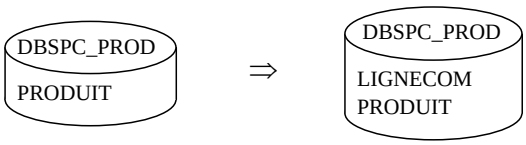
C T^+

P – $C \in$ collection(S)
– $T \in$ type-entites(S)
– $\forall$ C' $\in$ collection(S) : $T \notin$ te-coll(C')

Q – $T \in$ te-coll(C)

I /

E Ajouter la table *LIGNECOM* dans l'espace de stockage *DBSPC_PROD*.



e) Retrait d'une table

D Enlever une table d'un espace de stockage.

S $\exists$ C $\in$ collection(S), T $\in$ type-entites(S) : (C,{T1}) $\leftarrow$ modifier-Coll(S,C,{T2})

C T^-

P – C $\in$ collection(S)

– T $\in$ type-entites(S)

– T $\in$ te-coll(C)

– te-coll(C) \ {T} $\neq \varnothing$
C contient plus d'une table sinon il s'agit d'une destruction d'espace de stockage (voir C.2.4.2 b).

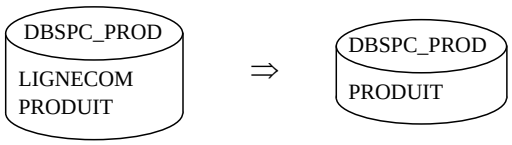
Q – T $\notin$ te-coll(C)

I $\forall$ T $\in$ te-coll(C), $\forall$ i $\in$ instance(T) : supprimer(i)

$\forall$ i $\in$ instance(C) : supprimer(i)

Les instances de T sont perdues. Il faut les extraire avant de détruire C.

E Enlever la table *LIGNECOM* de l'espace de stockage *DBSPC_PROD*.



f) Remarques importantes

Les modifications relatives aux espaces de stockage sont des opérations qui peuvent être très lourdes (en temps, accès disques, ...) pour le système dans le cas de grandes bases de données. Il faut donc les réaliser avec beaucoup de prudence.

Lors de la génération d'un script de génération de conversion de données sur la base d'un journal, il faut factoriser les modifications. Par exemple, si l'on enlève une table d'un espace de stockage pour l'ajouter à une autre, il y a des opérations redondantes dans les deux scripts. On ne doit créer qu'une seule fois la table temporaire et recréer une seule fois la table initiale en l'ajoutant à son nouvel espace de stockage.

C.2.5 Modifications des structures, des données et des programmes

Les exemples sont basés sur ceux présentés dans les points C.2.3 et C.2.4.

C.2.5.1 Modifications relatives aux tables

a) Création

Données

D Indépendance : la création d'une table n'influence pas les instances existantes de la base de données tant qu'elle n'est pas accompagnée de création de clés étrangères et de colonnes de référence.

S `CREATE TABLE t (c1 datatype NOT NULL,...,constraint ...);`

R – Certains SGBDR exigent l'utilisation de la contrainte NULL pour les colonnes facultatives (par défaut une colonne est NOT NULL). Toutefois, en règle générale, une colonne est NULL par défaut (voir le tableau de la page 36).

– Ce type de modifications s'accompagne d'autres créations comme la création de colonnes, de clés étrangères, d'identifiants, ...

E On crée une table *FOURNISSEUR* avec les colonnes Nfourn, *Nom*, *Adresse*, *Telephone* et *Jour*.

```
CREATE TABLE fournisseur (nfourn NUMERIC(5) NOT NULL,
  nom CHAR(30) NOT NULL,adresse CHAR(100) NOT NULL,telephone CHAR(10),
  jour CHAR(8) NOT NULL);
```

Programmes

D Indépendance totale : la création de la table n'affecte pas les programmes.

Toutefois, pour prendre en compte la nouvelle table, il faut créer de nouvelles fonctions de gestion (création, modification, suppression, affichage, ...).

Si la table est liée à d'autres tables par des clés étrangères, la création des contraintes affectera les programmes (voir C.2.5.3 a).

L /

b) Destruction

Données

D Indépendance : les instances de la table détruite sont perdues. Si la table est impliquée dans des clés étrangères, elle sont détruites avant (voir C.2.5.3 b).

S `DROP TABLE t CASCADE;`

Le mot clé CASCADE autorise la destruction d'une table sur laquelle des clés étrangères ou des vues ont été définies. Les index sont généralement détruits automatiquement. La destruction enlève aussi la table de l'espace de stockage dans lequel elle était stockée.

I La table peut être conservée sans les clés étrangères vers d'autres tables. Elle est renommée pour signaler sa disparition en préfixant son nom par "remove_" .

```
ALTER TABLE t DROP CONSTRAINT fk_tref; ...;
-- + Renommer t en remove_t.
```

E On détruit la table *FOURNISSEUR*.

```
DROP TABLE fournisseur CASCADE;
```

Programmes

D Dépendance structurelle : toutes les fonctions de gestion concernant la table détruite (création, modification, suppression, ...) sont obsolètes ainsi que les fonctions sur les tables ayant des clés étrangères vers la table supprimée.

L Recherche sur le nom de la table détruite, ses vues, ses colonnes ainsi que toutes les variables qui en dépendent.

c) Renommage

Données

D Indépendance : Les instances de la table sont toujours valables.

S Dans la norme SQL-92, il n'existe pas de commande pour renommer une table. Il faut créer une nouvelle table avec les mêmes colonnes et contraintes que la table à renommer.

```
-- Créer la nouvelle table identique à la table originale
CREATE TABLE t' (c1 datatype [NOT NULL],...,constraint ...,...) [IN spc];
-- Recréer les index sur t'
[CREATE INDEX idx_table ON t' (c1,...);
```

```
 ...;]
-- Transférer les instances de t dans t'.
INSERT INTO t' SELECT * FROM t;
-- Créer les clés étrangères référençant t'
[ALTER TABLE tref ADD CONSTRAINT fk_t'
   FOREIGN KEY (colref,...) REFERENCES t' (colid,...);
 ...;]
-- Détruire t
DROP TABLE t CASCADE;
```

R Il existe trois autres techniques pour renommer une table. Le point 4.2.3.3 les évalue de manière précise.

Premièrement, certains SGBDR disposent d'une commande pour renommer une table, toutes les contraintes sur la table étant conservées :

```
RENAME t to t';
```

Deuxièmement, on peut aussi utiliser la technique des vues pour créer une vue modifiable (nommée comme le nouveau nom de la table) qui contient toutes les colonnes de la table renommée. L'instruction de création d'une telle vue est :

```
CREATE VIEW t' AS SELECT * FROM t.
```

Il est possible également de donner un synonyme à une table qui n'est pas réellement renommée :

```
CREATE SYNONYME t' FOR t;
```

E On renomme la table *FOURNISSEUR* en *GROSSISTE*.

```
CREATE TABLE grossiste (nfourn NUMERIC(5) NOT NULL,nom CHAR(30) NOT NULL,
  adresse CHAR(100) NOT NULL,telephone CHAR(10) NOT NULL,
  jour CHAR(8) NOT NULL, CONSTRAINT id_gross PRIMARY KEY (nfourn));
INSERT INTO grossiste SELECT * FROM fournisseur;
ALTER TABLE produit ADD CONSTRAINT fk_grossiste
  FOREIGN KEY (nfourn) REFERENCES grossiste (nfourn);
DROP TABLE fournisseur CASCADE;
```

Programmes

D Indépendance structurelle : la recherche du nom original de la table et son remplacement par son nouveau nom dans l'application est la seule modification nécessaire pour assurer le bon fonctionnement du programme. Toutefois, pour assurer la cohésion de l'application, il faudrait peut-être également changer certains noms de champs, de titres, ... dans l'interface.

L Recherche sur l'ancien nom de la table renommée.

C.2.5.2 Modifications relatives aux colonnes

a) Création

Données

D Indépendance : L'ajout d'une colonne ne perturbe pas les instances de la table.

S Si la colonne est obligatoire, une valeur par défaut doit être ajoutée.

```
ALTER TABLE t ADD c datatype [DEFAULT value NOT NULL];
```

E On décide d'ajouter une colonne *Numcompte* à la table *FOUNISSEUR*.

```
ALTER TABLE fournisseur ADD numcompte num(12) DEFAULT 0 NOT NULL;
```

Programmes

D Dépendance structurelle : dans certaines instructions de manipulation de données (destruction, sélection, insertion ou modification), il faut tenir compte de la nouvelle colonne. Par exemple, une instruction du type `INSERT INTO T VALUES (...)` pose des problèmes puis-

que le nombre de paramètres dans VALUES ne sera plus identique au nombre de colonnes de la table. De même, la déclaration d'un curseur du type `DECLARE x CURSOR FOR SELECT * FROM ...` pose des problèmes lorsqu'on analyse le curseur. Dans l'interface, il faut ajouter le traitement de la colonne (affichage, modification, création, suppression, ...).

L Recherche sur les instructions de manipulation de données portant sur la table contenant la nouvelle colonne.

b) Destruction

Données

D Indépendance : les instances de la colonne détruite sont perdues.

S
```
ALTER TABLE table DROP column;
```

Certains SGBDR (par exemple Oracle 8) n'ont pas de commande de destruction de colonnes dans une table (voir le tableau de la page 36). Pour supprimer une colonne, il faut passer par une table intermédiaire :

```
-- Créer tmp identique a t
CREATE TABLE tmp(c1 datatype [NOT NULL],...);
-- Insérer les données de t dans tmp.
INSERT INTO tmp SELECT * FROM t;
-- Supprimer t
DROP TABLE t CASCADE;
-- Recréer t sans la colonne détruite (c)
CREATE TABLE t(c1 datatype [NOT NULL],...,constraint,...) [IN spc];
-- Recréer les index de t.
[CREATE INDEX idx ON t (c1, ...);
 ...;]
-- Recréer les clés étrangères référencant t
[ALTER TABLE tref ADD CONSTRAINT fk_t
   FOREIGN KEY(colref,...) REFERENCES t(colid,...);
 ...;]
-- Insérer les instances de tmp dans t
INSERT INTO t SELECT c1,... FROM tmp;
-- Détruire tmp
DROP TABLE tmp;
```

I Les instances de la colonne détruite peuvent être stockées dans une table de sauvegarde. Si la table d'origine a un identifiant, les colonnes identifiantes font aussi partie de cette table, sinon toutes les colonnes sont ajoutées à la table, ceci pour rendre la table exploitable en cas de besoin.

```
CREATE TABLE backup (colid datatype [NOT NULL],...,c datatype [NOT NULL]);
INSERT INTO backup SELECT colid,...,c FROM t;
```

E Détruire la colonne *Numcompte* de la table *FOURNISSEUR*.

```
CREATE TABLE backup_fournisseur (nfourn NUMERIC(5) NOT NULL,
  numcompte NUMERIC(12) NOT NULL);
INSERT INTO backup_fournisseur SELECT nfourn,numcompte FROM fournisseur;
ALTER TABLE fournisseur DROP numcompte;
```

Programmes

D Dépendance structurelle : dans les instructions de manipulation de données, il faut enlever les variables et la colonne correspondante. Dans l'interface, il faut enlever le traitement de la colonne (affichage, modification, ...).

L Recherche des requêtes sur la table contenant la colonne détruite.

c) Renommage

Données

D Indépendance : aucune perte de données.

S Le renommage d'une colonne se fait par la création d'une nouvelle colonne.

```
-- Créer la nouvelle colonne
ALTER TABLE t ADD c datatype [DEFAULT value NOT NULL];
-- Transférer les instances de c vers c'
UPDATE t SET c' = c;
-- Détruire c
ALTER TABLE t DROP c CASCADE;
-- Recréer les contraintes sur c'
[ALTER TABLE t ADD CONSTRAINT
   <id_t PRIMARY KEY | uniq_t UNIQUE> (c', ...);
 ALTER TABLE t ADD CONSTRAINT fk_t_id
   FOREIGN KEY (c', ...) REFERENCES tid (colid, ...);
 ...;]
-- Recréer les contraintes qui référençent c'
[ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (colref, ...) REFERENCES t (c', ...);
 ...;]
-- Recréer les index sur c'
[CREATE [UNIQUE] INDEX idx_t ON t (c',...);
 ...;]
```

La clause CASCADE dans le DROP d'une colonne assure que les contraintes sur celle-ci ainsi que les clés étrangères qui la référencent sont enlevées. Pour les SGBDR qui ne disposent pas de la commande DROP sur une colonne, le script est pratiquement le même que celui de la destruction d'une colonne, excepté la recréation de la table dans laquelle on crée la nouvelle colonne et le transfert des données qui se fait sur l'ensemble des colonnes.

E Renommer la colonne *Nfourn* de la table *FOURNISSEUR* en *Num*.

```
ALTER TABLE fournisseur ADD num NUMERIC(5) default 0 NOT NULL;
UPDATE fournisseur SET num = nfourn;
ALTER TABLE fournisseur DROP nfourn CASCADE;
ALTER TABLE fournisseur ADD PRIMARY KEY (num);
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (num) REFERENCES fournisseur (num);
CREATE INDEX idx_fourn ON fournisseur(num);
```

Programmes

D Indépendance structurelle : dans les instructions où la colonne apparaît, il faut la renommer. Une mise à jour de l'interface (écran de saisie, boîte de dialogue, ...) est peut-être nécessaire pour assurer la cohésion avec les données.

L Rechercher le nom original de la colonne modifiée ainsi que les variables qui en dépendent (pour modifier l'interface).

d) Destruction d'une contrainte NOT NULL

Données

D Indépendance

S
```
-- Détruire la contrainte NOT NULL définie sur c
ALTER TABLE t DROP CONSTRAINT not_null_c;
```

E Enlever la contrainte NOT NULL sur la colonne *Telephone* de la table *FOURNISSEUR*.

```
ALTER TABLE fournisseur DROP CONSTRAINT not_null_telephone;
```

Programmes

D Dépendance structurelle : puisque des valeurs nulles peuvent être introduites pour la colonne modifiée, il faut ajouter des tests pour traiter ces valeurs dans les programmes sinon il y a des risques d'exécuter des opérations non conformes (comme par exemple la division par une valeur nulle, l'affichage de valeur nulle dans un écran de saisie, ...). L'interface doit être également revue pour accepter ces valeurs nulles.

L Recherche sur le nom de la colonne modifiée ainsi que les variables qui en dépendent.

e) Création d'une contrainte NOT NULL

Données

D Dépendance : si la colonne modifiée contient des valeurs nulles, la nouvelle contrainte NOT NULL sur cette colonne n'est pas vérifiée.

I Pour voir si la modification est permise, il faut d'abord tester la validité des instances de la colonne modifiée :

```
SELECT c FROM t WHERE c IS NULL;
```

Si des instances ne vérifient pas la contrainte, deux solutions sont possibles :

Premièrement, la présence de valeurs nulles indiquent qu'il y a des problèmes avec certaines instances, la modification est reportée momentanément.

Deuxièmement, la modification est réalisée et les instances de la table violant la contrainte NOT NULL pour la colonne modifiée sont :

– soit enlevées de la table d'origine et stockées dans une table annexe ayant les mêmes colonnes que la table d'origine,

```
-- Créer la table contenant les lignes qui violent la contrainte NOT NULL
CREATE TABLE violation_t (c datatype [NOT NULL], ...);
-- Transférer les lignes concernées dans violation_t
INSERT INTO TABLE violation_t SELECT * FROM t WHERE c IS NULL;
-- Enlever les lignes concernées de t
DELETE FROM t WHERE c IS NULL;
```

– soit forcées à une valeur par défaut.

```
UPDATE t SET c = default_value WHERE c IS NULL;
```

S

```
-- Créer une colonne temporaire ctmp.
ALTER TABLE t ADD ctmp datatype DEFAULT value NOT NULL;
-- Insérer les instances de c dans ctmp.
UPDATE t SET ctmp = c;
-- Détruire c
ALTER TABLE t DROP c CASCADE;
-- Recréer c NOT NULL.
ALTER TABLE t ADD c datatype DEFAULT value NOT NULL;
-- Insérer les données de ctmp dans c
UPDATE t SET c = ctmp;
-- Détruire ctmp.
ALTER TABLE t DROP ctmp;
-- Recréer les contraintes sur c
[ALTER TABLE t ADD CONSTRAINT uniq_t UNIQUE (c,...);
 ...;]
-- Recréer les contraintes qui référencaient c
[ALTER TABLE tref ADD CONSTRAINT fk_t
   FOREIGN KEY (cref,...) REFERENCES table (c, ...);
 ...;]
-- Recréer les index sur c
[CREATE INDEX idx ON t (c, ...);
 ...;]
```

Dans certains SGBDR, il existe une commande pour ajouter une contrainte NOT NULL à une colonne (voir le tableau de la page 36). Dans Oracle 8, on peut exécuter :

```
ALTER TABLE t MODIFY (c NOT NULL);
```

E Ajouter la contrainte NOT NULL à la colonne *Telephone* de la table *FOURNISSEUR*. Pour les lignes où *Telephone* était NULL, la valeur ' ' a été mise par défaut.

```
ALTER TABLE fournisseur ADD telephone_tmp CHAR(10) DEFAULT '' NOT NULL;
UPDATE fournisseur SET telephone_tmp = telephone;
ALTER TABLE fournisseur DROP telephone CASCADE;
ALTER TABLE fournisseur ADD telephone CHAR(10) NOT NULL;
UPDATE fournisseur SET telephone = telephone_tmp;
ALTER TABLE fournisseur DROP telephone_tmp;
```

Programmes

D Dépendance structurelle : il faut vérifier le caractère obligatoire de la colonne notamment au niveau des interfaces sinon on risque d'introduire des valeurs nulles non autorisées. Les tests sur la colonne concernant des valeurs nulles deviennent superflus.

L Recherche sur le nom de la colonne modifiée ainsi que les variables qui en dépendent.

f) Extension de domaine

Données

D Indépendance : il n'y a pas de problème au niveau des instances.

S
```
-- Créer une colonne temporaire ctmp
ALTER TABLE t ADD ctmp new_datatype [DEFAULT value NOT NULL];
-- Transférer les données de c vers ctmp
UPDATE t SET ctmp = c;
-- Détruire c
ALTER TABLE t DROP c CASCADE;
-- Recréer c avec son nouveau type
ALTER TABLE t ADD c new_datatype [DEFAULT value NOT NULL];
-- Transférer les données de ctmp vers c
UPDATE t SET c = ctmp;
-- Détruire ctmp
ALTER TABLE DROP ctmp;
-- Recréer les identifiants sur c
[ALTER TABLE t ADD CONSTRAINT id_t <PRIMARY KEY | UNIQUE> (c,...);]
-- Recréer les index sur c
[CREATE INDEX idx ON t (c,...);
 ...;]
-- Il faut propager la modification aux colonnes qui referencent l'identifiant
-- dont c fait partie. Un script equivalent à celui ci-dessus est appliqué autant
-- de fois qu'il y a de colonnes de référence à modifier.
-- Recréer les contraintes référencant c
[ALTER TABLE tref ADD CONSTRAINT fk_t
   FOREIGN KEY (cref,...) REFERENCES t (c,...);
 ...;]
```

Dans certains SGBDR, il existe une commande qui permet de modifier le type d'une colonne (voir le tableau de la page 36). Cette commande ne demande pas que la table soit vide pour effectuer la modification. Dans Oracle 8, on a :

```
ALTER TABLE t MODIFY c new_datatype;
```

E Etendre le domaine de la colonne *Ncli* de la table *CLIENT.* Passer de NUMERIC(5) à NUMERIC(6).

```
ALTER TABLE client ADD ncli_tmp NUMERIC(6) DEFAULT 0 NOT NULL;
UPDATE client SET ncli_tmp = ncli;
ALTER TABLE client DROP ncli CASCADE;
ALTER TABLE client ADD ncli NUMERIC(6) DEFAULT 0 NOT NULL;
UPDATE client SET ncli = ncli_tmp;
```

```
ALTER TABLE DROP ncli_tmp;
ALTER TABLE client ADD CONSTRAINT idclient PRIMARY KEY (ncli);
CREATE INDEX idclient ON client(ncli);
+ propagation de la modification a ncli de commande.
ALTER TABLE commande ADD CONSTRAINT fk_client
  FOREIGN KEY (ncli) REFERENCES client(ncli);
```

Programmes

D Indépendance structurelle : les programmes sont corrects. Toutefois, pour permettre l'introduction de valeurs dans le nouveau domaine, il faut changer le format des variables qui vont contenir des données des colonnes modifiées ainsi que la taille des champs de saisie et d'affichage dans l'interface.

L Recherche sur le nom des colonnes dont on a étendu le domaine ainsi que les variables qui en dépendent.

g) Restriction de domaine

Données

D Dépendance : si la colonne contient des valeurs dont la longueur est supérieure à celle du domaine, cela pose des problèmes.

I Pour voir si la modification est permise, il faut d'abord lancer la commande :

```
SELECT * FROM t WHERE LENGTH(c) > maxlength;
```

Si des instances ne vérifient pas la contraintes, deux solutions sont envisageables :

Premièrement, la présence de valeurs dont la longueur est supérieure à la longueur maximum indiquent qu'il y a des problèmes avec certaines instances, la modification est reportée momentanément.

Deuxièmement, la modification est réalisée et les instances de la table violant la contrainte de domaine sont :

– soit enlevées de la table d'origine et stockées dans une table annexe ayant les mêmes colonnes que la table d'origine. Cette solution doit être également appliquée aux colonnes qui référencent la colonne modifiée.

```
-- Créer la table contenant les lignes illicites
CREATE TABLE violation_t (c datatype [NOT NULL], ...);
-- Transférer les lignes concernées dans violation_t
INSERT INTO TABLE violation_t SELECT * FROM t WHERE LENGTH(c) > maxlength;
-- Enlever les lignes illicites de t
DELETE FROM t WHERE LENGTH(c) > maxlength;
```

– soit tronquées pour respecter le domaine. Pour cela, il faut que des opérateurs pour le tronquage soient disponibles et qu'on effectue la même opération sur les colonnes qui référencent la colonne modifiée.

```
UPDATE t SET c = TRUNC(c) WHERE LENGTH(c) > maxlength;
```

S La commande MODIFY (Oracle 8) présentée dans le point précédent ne peut être utilisée pour restreindre un domaine que si la colonne est vide. Le script passant par une colonne temporaire (voir le point f) est la seule solution possible pour cette modification.

E Restreindre le domaine de la colonne *Adresse* de la table *CLIENT*. Passer de CHAR(60) à CHAR(50).

```
ALTER TABLE client ADD adresse_tmp CHAR(50) DEFAULT ' ' NOT NULL;
UPDATE client SET adresse_tmp = adresse;
ALTER TABLE client DROP adresse CASCADE;
ALTER TABLE client ADD adresse CHAR(50) DEFAULT ' ' NOT NULL;
UPDATE client SET adresse = adresse_tmp;
ALTER TABLE DROP adresse_tmp;
```

Programmes

D Indépendance structurelle : les programmes fonctionnent correctement mais pour les nouvelles instances, il faut changer le format des variables qui vont contenir des instances de la colonne modifiée ainsi que les interfaces.

L Recherche sur le nom de la colonne dont on a restreint le domaine ainsi que les variables qui en dépendent.

h) Changement de type

Données

D Dépendance : il se peut que des instances de la colonne modifiée ne puissent être converties. Par exemple, une colonne de type caractère contenant des noms de personnes ne peut être convertie en nombres.

I Pour voir si la modification est permise, il faut vérifier que les instances peuvent être converties. Il n'est pas possible de fournir une requête standard pour vérifier la conversion car chaque SGBDR a ses propres fonctions de conversion et tous les types de conversion ne sont pas possibles. C'est donc au concepteur d'évaluer la faisabilité de la conversion.

Comme dans le point g), le concepteur a le choix entre deux solutions pour traiter le problème des instances non convertibles.

S Le script est identique au script pour l'extension du domaine. Il faut simplement utiliser une fonction de conversion lorsque les données de la colonne temporaire sont transférées dans la colonne modifiée. En règle générale, il y a autant de fonctions de conversion différentes que de SGBDR.

L'utilisation de la commande MODIFY (Oracle 8) n'est pas possible lorsque la colonne n'est pas vide.

E Changer le type de la colonne *Telephone* de la table *CLIENT*. Passer du NUMERIC(9) au CHAR(9).

```
ALTER TABLE client ADD telephone_tmp CHAR(9) DEFAULT ' ' NOT NULL;
UPDATE client SET telephone_tmp = telephone;
ALTER TABLE client DROP telephone CASCADE;
ALTER TABLE client ADD telephone CHAR(9) DEFAULT ' ' NOT NULL;
UPDATE client SET telephone = CAST(telephone_tmp AS CHAR);
ALTER TABLE DROP telephone_tmp;
```

Programmes

D Indépendance structurelle : les variables chargées de contenir les données de la colonne modifiée vont devoir changer de type. Les traitements sur ces variables vont également devoir être modifiés dans certains cas.

L Recherche sur le nom de la colonne dont on a changé le type ainsi que les variables qui en dépendent.

C.2.5.3 Modifications relatives aux clés étrangères

a) Création

Données

D Dépendance : si la table contient déjà des instances, les colonnes de référence doivent avoir des valeurs correctes (identiques aux valeurs de la clé référencée si elles sont obligatoires). Si, en plus elles forment une clé primaire ou candidate, les valeurs doivent être distinctes.

S
```
ALTER TABLE t ADD c1 datatype [DEFAULT value NOT NULL];
...;
ALTER TABLE t ADD ck datatype [DEFAULT value NOT NULL];
+ Remplissage de valeurs pour les colonnes créées
[ALTER TABLE t ADD CONSTRAINT
   id_t <PRIMARY KEY | uniq_t UNIQUE> (c1,...,ck);]
ALTER TABLE t ADD CONSTRAINT fk_t_id
  FOREIGN KEY (c1,...,ck) REFERENCES t_id (cid1,...,cidk);
[CREATE INDEX idx ON table (c1,...,ck);]
```

E On crée dans la table *PRODUIT* une colonne obligatoire *Nfourn* qui référence la clé primaire *Nfourn* de la *FOURNISSEUR*.
```
ALTER TABLE produit ADD num NUMERIC(5) DEFAULT 0 NOT NULL;
-- + Remplissage de valeurs
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (num) REFERENCES fournisseur (nfourn);
```

Programmes

D Dépendance structurelle : dans certaines instructions de manipulation de données (destruction, sélection, insertion ou modification), il faut tenir compte des colonnes de référence (voir la création de colonnes au point C.2.5.2). Les interfaces (écrans, boîtes de saisies, ...) doivent être revues pour insérer des fonctions de gestion de la nouvelle contrainte.

L Recherche sur le nom de la table de référence dans laquelle on a créé la clé étrangère.

b) Destruction

Données

D Indépendance : la destruction de la clé étrangère et des colonnes de référence n'altère pas la base de données.

S
```
ALTER TABLE t DROP c1 CASCADE;
...;
ALTER TABLE t DROP ck CASCADE;
```

Le mot clé CASCADE assure que toutes les contraintes sur les colonnes sont détruites.

I Les instances des colonnes supprimées sont stockées dans une table annexe avec les identifiants éventuels de la table (ce qui permettra aux données d'être éventuellement exploitées) ou l'ensemble des autres colonnes de la table.
```
-- Créer la table contenant les colonnes détruites et identifiantes
CREATE TABLE backup_t (cid datatype [NOT NULL],...,
                       c1 datatype [NOT NULL],...,ck datatype [NOT NULL]);
-- Transférer les colonnes concernées dans backup_t
INSERT INTO TABLE backup_t SELECT cid,...,c1,...,ck FROM t;
```

E On détruit la clé étrangère et la colonne *Nfourn* dans la table *PRODUIT*.
```
ALTER TABLE produit DROP num CASCADE;
```

Programmes

D Dépendance structurelle : dans les instructions de manipulation de données, il faut enlever les colonnes détruites ainsi que les variables correspondantes. Dans l'interface, il faut enlever la gestion de la clé étrangère détruite (affichage, modification, ...).

L Recherche sur le nom des colonnes de référence supprimées ainsi que les variables qui en dépendent.

c) Renommage

Données

D Indépendance : Les instances des colonnes renommées restent inchangées.

S Voir renommer une colonne au point C.2.5.2.

E Dans *PRODUIT*, on renomme la colonne *Nfourn* par *Num*.

```
ALTER TABLE produit ADD num NUMERIC(5) DEFAULT 0 NOT NULL;
UPDATE produit SET num = nfourn;
ALTER TABLE produit DROP nfourn CASCADE;
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (num) REFERENCES fournisseur (nfourn);
```

Programmes

D Indépendance structurelle : dans les instructions où les colonnes apparaissent, il faut les renommer. Une mise à jour de l'interface (écran de saisie, boîte de dialogue, ...) est peut-être nécessaire pour assurer la cohésion avec les schémas de la base.

L Recherche sur les noms des anciennes colonnes de référence ainsi que les variables qui en dépendent.

d) Destruction de la clé primaire définie sur les colonnes de référence

Données

D Indépendance : les instances de la table sont correctes vis-à-vis de la modification.

S Voir la destruction d'une clé primaire ou candidate au point C.2.5.4 .

```
ALTER TABLE t DROP CONSTRAINT id_t;
```

E On détruit la clé candidate contenant *Nfourn* dans *PRODUIT*.

```
ALTER TABLE PRODUIT DROP CONSTRAINT id_produit;
```

Programmes

D Dépendance structurelle : le résultat de certaines requêtes va devoir être stocké dans un curseur plutôt que dans une variable. Cela nécessite la gestion de curseurs. Dans l'interface, on pourrait devoir changer des champs en listes puisque la sélection sur les colonnes de référence peut donner plusieurs instances.

L Recherche sur les colonnes de référence ainsi que les variables qui en dépendent.

e) Création d'une clé primaire sur les colonnes de référence

Données

D Dépendance : un problème d'intégrité concernant les instances de la table peut apparaître si elles ont les mêmes valeurs que les colonnes de référence.

I Il faut vérifier par une requête si la modification peut se faire (Voir création d'une clé primaire ou candidate au point C.2.5.4).

S `ALTER TABLE t ADD CONSTRAINT id_t <PRIMARY KEY | UNIQUE> (c1,...,ck);`

E Ajouter au type d'entités *PRODUIT* une clé candidate contenant *Nfourn*.

```
ALTER TABLE produit ADD UNIQUE (nfourn);
```

Programmes

D Dépendance structurelle : il faut ajouter des contraintes au niveau de la saisie et des modifications des colonnes de référence pour éviter de violer la contrainte d'unicité. Les programmes pourraient être simplifiés au niveau des instructions de sélection (l'utilisation de curseur n'est plus nécessaire), des interfaces (certaines listes peuvent être remplacées par des champs, ...).

L Recherche sur les colonnes de référence ainsi que les variables qui en dépendent.

f) Création de contraintes NOT NULL sur les colonnes de référence

Données

D Dépendance : les valeurs nulles dans les colonnes de référence posent problème.

I Voir la création d'une contrainte NOT NULL au point C.2.5.2.

S Le script est identique à celui du point C.2.5.2 excepté qu'il peut s'appliquer à plusieurs colonnes et qu'il faut recréer la clé étrangère portant sur ces colonnes.

```
-- Créer des colonnes temporaires
ALTER TABLE t ADD c1tmp datatype DEFAULT value NOT NULL;
...;
ALTER TABLE t ADD cktmp datatype DEFAULT value NOT NULL;
-- Insérer les données de c1,...,ck dans c1tmp,...,cktmp
UPDATE t SET c1tmp = c1,...,cktmp = ck;
-- Détruire c et de ses contraintes
ALTER TABLE t DROP c1 CASCADE;
...;
ALTER TABLE t DROP ck CASCADE;
-- Recréer c1,...,ck NOT NULL
ALTER TABLE t ADD c1 datatype DEFAULT value NOT NULL;
...;
ALTER TABLE t ADD ck datatype DEFAULT value NOT NULL;
-- Insérer les données de c1tmp,...,cktmp dans c1,...,ck
UPDATE t SET c1 = c1tmp,...,ck = cktmp;
-- Détruire c1tmp,...,cktmp
ALTER TABLE t DROP c1tmp;
...;
ALTER TABLE t DROP cktmp;
-- Recréer les contraintes sur c1,...,ck
[ALTER TABLE t ADD CONSTRAINT uniq_t UNIQUE (c1,...,ck);]
ALTER TABLE t ADD CONSTRAINT fk_t_id
  FOREIGN KEY (c1,...,ck) REFERENCES t_id (c1id,...,ckid);
...;
-- Recréer les contraintes qui référencaient c1,...,ck
[ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (c1ref,...,ckref) REFERENCES table (c1,...,ck);
 ...;]
-- Recréer les index sur c1,...,ck
[CREATE INDEX idx ON t (c1,...,ck);
 ...;]
```

Dans certains SGBDR, il existe une commande pour ajouter une contrainte NOT NULL à une colonne (voir le tableau de la page 36). Dans Oracle 8, on peut exécuter :

```
ALTER TABLE t MODIFY (c NOT NULL);
```

E La colonne *NFOURN* de la table *PRODUIT* devient NOT NULL.

```
ALTER TABLE produit ADD nfourn1 NUMERIC(5) DEFAULT 0 NOT NULL;
UPDATE produit SET nfourn1 = nfourn;
ALTER TABLE produit DROP nfourn CASCADE;
ALTER TABLE produit ADD nfourn NUMERIC(5) DEFAULT 0 NOT NULL;
UPDATE produit SET nfourn = nfourn1;
ALTER TABLE produit DROP nfourn1;
```

```
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (nfourn) REFERENCES fournisseur (nfourn);
```

Programmes

D Dépendance structurelle : il faut vérifier le caractère NOT NULL de la colonne notamment au niveau des interfaces sinon on risque d'introduire des valeurs nulles non autorisées. Les tests sur la colonne concernant des valeurs nulles deviennent superflus.

L Recherche sur les colonnes de référence ainsi que sur les variables qui en dépendent.

g) Destruction des contraintes NOT NULL définies sur les colonnes de référence

Données

D Indépendance

S
```
-- Si c1,...,ck appartenaient à une clé primaire, détruire la contrainte PRIMARY
-- KEY ainsi que les contraintes qui les référençaient.
-- Recréer une contrainte UNIQUE sur c1,...,ck ainsi que les contraintes
-- qui les référençaient.
[ALTER TABLE t DROP CONSTRAINT id_t CASCADE;
 ALTER TABLE t ADD CONSTRAINT UNIQUE(c1,...,ck);
 ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (c1ref,...,ckref) REFERENCES t (c1,...,ck);
 ...;]
-- Détruire les contraintes NOT NULL.
ALTER TABLE t DROP CONSTRAINT not_null_c1;
...;
ALTER TABLE t DROP CONSTRAINT not_null_ck;
```

R Certains SGBDR obligent que les colonnes référencées appartiennent à une clé primaire (voir le tableau de la page 36). Dans ce cas, les colonnes de référence ne peuvent être référencées à leur tour.

E La colonne *NFOURN* du type d'entités *PRODUIT* devient NULL.
```
ALTER TABLE produit DROP CONSTRAINT not_null_nfourn;
```

Programmes

D Dépendance structurelle : voir la destruction d'une contrainte NOT NULL au point C.2.5.2.

L Recherche sur le nom de la colonne modifiée ainsi que sur les variables qui en dépendent.

C.2.5.4 Modifications relatives aux clés primaires et candidates

a) Création

Données

D Dépendance : certaines instances de la table peuvent ne pas vérifier la contrainte d'unicité.

I Pour vérifier si les instances vérifient la nouvelle contrainte, l'instruction suivante peut être utilisée :
```
SELECT column,... FROM table GROUP BY column,... HAVING COUNT(*) > 1;
```
Si la requête donne un résultat, la contrainte n'est pas vérifiée et le concepteur a deux choix possibles :

Premièrement, la modification est refusée tant que le problème n'est pas règlé.

Deuxièmement, les instances illicites sont transférées dans une table annexe ayant la même structure que la table originale.

```
-- Créer une table temporaire avec les colonnes du nouvel identifiant
CREATE TABLE t_tmp (c1 datatype [DEFAULT value NOT NULL]);
-- Insérer dans t_tmp les valeurs des colonnes en double dans t
INSERT INTO violation_t (c1)
  SELECT c1 FROM t GROUP BY c1 HAVING COUNT(*) > 1;
-- Création et stockage des données redondantes dans violation_t
CREATE TABLE t_violation (c1 datatype [DEFAULT value NOT NULL], ...);
INSERT INTO t_violation
  SELECT * FROM t WHERE (c1) IN (SELECT c1 FROM t_tmp);
-- Destruction des lignes de t violants la contrainte
DELETE FROM t WHERE (c1) IN (SELECT c1 FROM t_tmp);
-- Detruire t_tmp
DROP TABLE t_tmp;
```

S `ALTER TABLE t ADD CONSTRAINT <id_t PRIMARY KEY | uniq_t UNIQUE> (c1);`

R Dans certains SGBDR, une clé candidate (contrainte UNIQUE) définie sur une seule colonne n'autorise que la présence d'une seule valeur NULL pour cette colonne (voir le tableau de la page 36). Cela veut dire que la valeur NULL est considérée comme tout autre valeur. Il en va de même pour une contrainte UNIQUE portant sur plusieurs colonnes. Oracle 8 autorise plusieurs NULL pour une contrainte UNIQUE mono-colonne mais pas multi-colonnes.

E Ajouter une clé candidate à la table *CLIENT* qui contient les colonnes *Nom* et *Prenom*.

```
CREATE TABLE tmp (nom CHAR(50) NOT NULL,prenom CHAR(30));
INSERT INTO tmp (nom,prenom)
  SELECT nom,prenom FROM client GROUP BY nom,prenom HAVING COUNT(*) > 1;
CREATE TABLE client_tmp(ncli NUMERIC(5) NOT NULL,nom CHAR(50) NOT NULL,
  prenom CHAR(30),adr_rue CHAR(40) NOT NULL,adr_localite CHAR(30) NOT NULL,
  adr_code_postal NUMERIC(4) NOT NULL,telephone CHAR(12),
  nrep NUMERIC(5) NOT NULL,cr_nom CHAR(30) NOT NULL);
INSERT INTO client_tmp
  SELECT * FROM client WHERE (nom,prenom) IN (SELECT nom,prenom FROM tmp);
DELETE FROM client WHERE (nom,prenom) IN (SELECT nom,prenom FROM tmp);
DROP TABLE tmp;
ALTER TABLE client ADD CONSTRAINT idclient1 UNIQUE (nom,prenom);
```

Programmes

D Dépendance structurelle : il faut ajouter des contraintes au niveau de la saisie et des modifications des colonnes identifiantes pour éviter de violer la contrainte d'unicité. Les programmes pourraient être simplifiés au niveau des instructions de sélection (l'utilisation de curseur n'est plus nécessaire).

L Recherche sur le nom de la table à laquelle on ajoute une contrainte ainsi que les colonnes participant à la contrainte et les variables qui en dépendent.

b) Destruction

Données

D Indépendance : Les instances sont conformes aux contraintes du schéma.

S `ALTER TABLE t DROP CONSTRAINT <id_t | uniq_t>;`

E Détruire la clé candidate concernant les colonnes *Nom* et *Prenom* de la table *CLIENT*.

```
ALTER TABLE client DROP CONSTRAINT idsecclient
```

Programmes

D Dépendance structurelle : le résultat de certaines requêtes va devoir être stocké dans un curseur plutôt que dans une variable. Cela nécessite la gestion de curseurs. Dans l'interface, la vérification du caractère unique doit être enlevée.

L Recherche sur le nom des colonnes dont on enlève la contrainte ainsi que les variables qui en dépendent.

c) Renommage

Données

D Indépendance : les instances sont toujours conformes au schéma.

S Il faut enlever la contrainte concernée et la recréer ainsi que les clés étrangères qui référençaient la clé primaire ou candidate.

```
ALTER TABLE t DROP CONSTRAINT <id_t | uniq_t> CASCADE;
ALTER TABLE t ADD CONSTRAINT
  <new_id_t PRIMARY KEY | new_uniq_t UNIQUE> (c1);
[ALTER TABLE t_ref ADD CONSTRAINT fk_t FOREIGN KEY (cref) REFERENCES t (c1);
 ...;]
```

La clause CASCADE dans le DROP CONSTRAINT entraîne que les contraintes qui référencaient la clé primaire ou candidate sont également supprimées. Si la clause CASCADE n'est pas autorisée, il faut explicitement détruire ces clés étrangères. Ensuite, la recréer avec son nouveau nom ainsi que les clés étrangères qui la référencaient.

E Renommer la clé candidate IDSECCLIENT contenant les colonnes NOM et PRENOM de la table CLIENT en UNIQCLIENT.

```
ALTER TABLE client DROP CONSTRAINT idsecclient;
ALTER TABLE client ADD CONSTRAINT uniqclient;
```

Programmes

D Indépendance totale : normalement, le nom de la contrainte n'est pas utilisé dans les programmes.

L /

d) Changement d'une clé primaire en candidate et inversement

Données

D Indépendance

S Il faut d'abord enlever les clés étrangères qui référencent la clé primaire ou candidate modifiée. Ensuite, on enlève la contrainte identifiante. Finalement, on crée la nouvelle contrainte et les clés étrangères qui ont été supprimées.

```
-- Détruire l'identifiant.
ALTER TABLE t DROP CONSTRAINT <id_t | uniq_t> CASCADE;
-- Recréer le nouvel identifiant.
ALTER TABLE t ADD CONSTRAINT
  <new_id_t PRIMARY KEY | new_uniq_t UNIQUE> (c1);
-- Recréer les contraintes qui référençaient l'identifiant.
[ALTER TABLE t_ref ADD CONSTRAINT fk_t FOREIGN KEY (cref) REFERENCES t (c1);
 ...;]
```

Certains SGBDR n'autorisent pas qu'une clé étrangère référence une clé candidate (contrainte UNIQUE).

E Changer la clé primaire définie sur *Ncli* de la table *CLIENT* en clé candidate.

```
ALTER TABLE client DROP CONSTRAINT idclient CASCADE;
```

```
ALTER TABLE client ADD CONSTRAINT uniqclient (ncli);
ALTER TABLE commande ADD CONSTRAINT fk_client FOREIGN KEY (ncli)
                         REFERENCES client (ncli);
```

Programmes

D Indépendance totale : le programmeur ne connait pas la contrainte. Il sait juste qu'il y a une contrainte d'unicité sur des colonnes.

L /

e) Ajout d'un composant

Données

D Indépendance : Ajouter une colonne à une clé primaire ou candidate assure que les instances de la table vérifie toujours la contrainte.

S Il faut supprimer la contrainte PRIMARY KEY ou UNIQUE et la reconstruire en y ajoutant une colonne.

Si la clé modifiée est référencée par d'autres tables, il faut ajouter dans les clés étrangères correspondantes une colonne correspondant à la colonne ajoutée à la clé identifiante. Les données de la colonne ajoutée dans les tables de référence sont celles de la colonne ajoutée à la clé identifiante en fonction des valeurs des autres colonnes de la clé étrangère. Cette modification des tables de référence peut se produire en cascade.

```
-- Détruire l'identifiant.
ALTER TABLE t DROP CONSTRAINT <id_t | uniq_t> CASCADE;
-- Recréer le nouvel identifiant.
ALTER TABLE t ADD CONSTRAINT
  <id_t PRIMAY KEY | uniq_t UNIQUE> (c1,c2);
-- Créer une nouvelle colonne dans les tables référencants la clé primaire ou
-- candidate, remplir cette colonne avec les valeurs correspondantes de table et
-- recréer les clé étrangères qui référençaient la clé primaire ou candidate.
[ALTER TABLE t_ref ADD c2ref datatype [DEFAULT value NOT NULL];
 UPDATE t_ref SET
   c2ref = (SELECT c2 FROM t WHERE t_ref.cref = t.cid AND ...);
 ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (...,c2ref) REFERENCES t (...,c2);
 ...;]
```

E Ajouter la colonne *Nom* à la clé primaire de la table *REPRESENTANT*.

```
ALTER TABLE representant DROP CONSTRAINT id_representant CASCADE;
ALTER TABLE representant ADD CONSTRAINT id_representant
  PRIMAY KEY (nrep, nom);
ALTER TABLE client ADD nomrep CHAR(50) DEFAULT '' NOT NULL;
UPDATE client SET nom_rep = (SELECT nom FROM representant
                             WHERE client.nrep = representant.nrep);
ALTER TABLE client ADD CONSTRAINT fk_representant
  FOREIGN KEY (nrep, nom_rep) REFERENCES representant(nrep, nom);
```

Programmes

D Dépendance structurelle : dans les requêtes, les jointures et les sélections imbriquées impliquant les colonnes de la clé primaire ou candidate doivent changer.

Il faut revoir les interfaces. L'ajout d'un composant entraîne une modification de la vérification de l'unicité d'un objet.

L Recherche sur le nom des colonnes de la clé primaire ou candidate ainsi que des variables qui en dépendent.

f) Retrait d'un composant

Données

D Dépendance : le retrait d'une colonne d'une clé primaire ou candidate peut entraîner la violation de la contrainte d'unicité par certaines instances de la table.

I Le résolution du problème est identique au point a).

S Il faut supprimer la contrainte PRIMARY KEY ou UNIQUE et la reconstruire en y enlevant un élément.

```
-- Supprimer la contrainte primary key
ALTER TABLE t DROP CONSTRAINT <id_t | uniq_t> CASCADE;
-- Recréer la contrainte primary key
ALTER TABLE t ADD CONSTRAINT <id_t PRIMAY KEY | uniq_t UNIQUE> (c1);
-- Détruire la colonne correspondant a la colonne enlevée de l'identifiant
-- dans les tables référençant l'identifiant et recréer les contraintes
-- qui référençaient l'identifiant.
[ALTER TABLE t_ref DROP c2ref;
 ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (c1ref) REFERENCES t (c1);
 ...;]
```

E Enlever la colonne *Nom* de la clé primaire de la table *REPRESENTANT*.

```
ALTER TABLE representant DROP CONSTRAINT id_representant;
ALTER TABLE representant ADD CONSTRAINT id_representant PRIMAY KEY(nrep);
```

Programmes

D Dépendance structurelle : dans les requêtes, les jointures et les sélections imbriquées doivent changer.

Il faut revoir les interfaces. Le retrait d'un composant entraîne une modification de la vérification de l'unicité d'un objet.

L Recherche sur le nom des colonnes de la clé primaire ou candidate ainsi que les variables qui en dépendent.

C.2.5.5 Modifications relatives aux index

a) Création

Données

D Indépendance

S Il faut ajouter un index à la base de données.

```
CREATE [UNIQUE] INDEX idx ON t (c1);
```

La clause UNIQUE n'est pas nécessaire si on a déjà créé une contrainte PRIMARY KEY ou UNIQUE sur les colonnes de la table. Cette clause vérifie simplement si les entrées dans l'index sont uniques.

Certains SGBD (comme Oracle 8) ajoute systématiquement un index sur toutes les colonnes d'une table pour lesquelles on a défini une contrainte PRIMARY KEY ou UNIQUE (voir le tableau de la page 36).

E Ajouter un index sur les colonnes *NOM* et *PRENOM* de la table *CLIENT*.

```
CREATE INDEX idx_client ON client (nom, prenom);
```

Programmes

D Indépendance totale : les index accélèrent certaines requêtes mais ils ne nécessitent pas de modifications au niveau des programmes. C'est le moteur SQL qui va les utiliser pour améliorer ces performances. Toutefois, certaines requêtes peuvent être revues suite à la création d'un nouvel index.

L /

b) Destruction

Données

D Indépendance

S Il faut détruire un index.

```
DROP INDEX idx;
```

En Oracle 8, il est interdit de détruire un index créé automatiquement avec les contraintes PRIMARY KEY ou UNIQUE.

E Détruire l'index sur les colonnes *Nom* et *Prenom* de la table *CLIENT*.

```
DROP INDEX idx_client;
```

Programmes

D Indépendance totale : la perte d'un index détériore les performances mais n'affecte pas les programmes. Toutefois, certaines requêtes peuvent être revues pour améliorer les performances.

L /

c) Renommage

Données

D Indépendance

S Il faut détruire l'index et le recréer avec le nouveau nom. Ce genre d'opérations est relativement lourd et doit être utilisé avec précaution.

```
DROP INDEX idx;
CREATE [UNIQUE] INDEX idx_new ON t (c1);
```

En Oracle 8, un index qui renforce une clé primaire (PRIMARY KEY) ou candidate (UNIQUE) ne peut être détruit.

E Renommer l'index *Idx_client* sur les colonnes *Nom* et *Prenom* de la table *CLIENT* en *Idx_nomcli*.

```
DROP INDEX idx_client;
CREATE INDEX idx_nomcli ON client (nom, prenom);
```

Programmes

D Indépendance totale : le nom d'un index n'apparaît pas dans les programmes.

L /

d) Ajout d'un composant

Données

D Indépendance

S Il faut détruire l'index et le recréer avec le nouveau composant. Ce genre d'opérations peut être lourd.

```
DROP INDEX idx;
CREATE [UNIQUE] INDEX idx ON table (c1,c2);
```

E Ajouter la colonne *Ncli* à l'index *Idx_client* de la table *CLIENT*.

```
DROP INDEX idx_client;
CREATE INDEX idx_client ON client (ncli, nom, prenom);
```

Programmes

D Indépendance totale : ajouter un composant à un index n'affecte pas les programmes.

Toutefois, on peut revoir certaines instructions de manipulation de données dans un souci d'optimisation.

L Recherche sur les noms de la colonne ainsi que sur les variables qui en dépendent.

e) Retrait d'un composant

Données

D Indépendance

S Il faut détruire l'index et le recréer en enlevant un composant. Ce genre d'opérations peut être lourd.

```
DROP INDEX idx;
CREATE [UNIQUE] INDEX idx ON t (c1);
```

E Enlever la colonne *Ncli* de l'index *Idx_client* de la table *CLIENT*.

```
DROP INDEX idx_client;
CREATE INDEX idx_client ON client (nom, prenom);
```

Programmes

D Indépendance totale : enlever un composant d'un index n'affecte pas les programmes.

Toutefois, certaines requêtes peuvent être revues dans un souci d'optimisation.

L Recherche sur les noms de la colonne ainsi que sur les variables qui en dépendent.

C.2.5.6 Modifications relatives aux espaces de stockage

a) Création

Données

D Indépendance

S La norme SQL-92 ne donnant aucune information sur la notion d'espace de stockage, nous présenterons les syntaxes de quelques SGBDR. On parle de *FILEGROUP* en SQL/Server ou de *TABLESPACE* en Oracle et DB2.

```
SQL/Server: ALTER DATABASE database ADD FILEGROUP filegroup;
Oracle:     CREATE TABLESPACE tablespace DATAFILE datafile
              [DEFAULT STORAGE (parameters)];
DB2:        CREATE TABLESPACE tablespace MANAGED BY DATABASE
              USING (FILE filename nbr-page);
```

Il faut également ajouter les tables dans le nouvel espace de stockage. Cela se fait lors de la création des tables. Cela pose des problèmes si les tables sont déjà créées et non vides. La seule solution possible est d'utiliser une table temporaire pour stocker les instances et de recréer la table dans son espace de stockage.

```
-- Créer t_tmp avec la structure de t
```

```
CREATE TABLE t_tmp (c1 datatype [NOT NULL], ...);
-- Insérer les données de t dans t_tmp
INSERT INTO t_tmp SELECT * FROM t;
-- Supprimer t
DROP TABLE t CASCADE;
-- Recréer t avec sa structure et ses contraintes en l'ajoutant à son
-- espace de stockage
SQL/Server: CREATE TABLE t (c1 datatype [NOT NULL],...) ON filegroup;
Oracle:     CREATE TABLE t (c1 datatype [NOT NULL],...) TABLESPACE tablespace;
DB2:        CREATE TABLE t (c1 datatype [NOT NULL],...) IN tablespace;
+ création des contraintes de la table
-- Insérer les données de t_tmp dans t
INSERT INTO t SELECT * FROM t_tmp;
-- Recréer les clés étrangères qui référencaient t
[ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (cref,...) REFERENCES table (cid,...);]
-- Recréer les index de t
[CREATE INDEX idx ON t (c1,...);
 ...;]
-- Détruire t_tmp
DROP TABLE t_tmp;
```

E Ajouter un espace de stockage PRODUITS contenant la table *PRODUIT*.

```
CREATE TABLESPACE produits DATAFILE 'produits.dat' SIZE 100K REUSE;
CREATE TABLE produit_tmp (NPRO char(5) not null,
  LIBELLE char(50) not null,QSTOCK num(10) not null);
INSERT INTO produit_tmp SELECT * FROM produit;
DROP TABLE produit CASCADE CONSTRAINT;
CREATE TABLE produit (NPRO char(5) not null,LIBELLE char(50) not null,
  QSTOCK num(10) not null,PRIMARY KEY (NPRO)) TABLESPACE produits;
INSERT INTO produit SELECT * FROM produit_tmp;
ALTER TABLE lignecom ADD CONSTRAINT fkpl
  FOREIGN KEY (npro) REFERENCES produit;
ALTER TABLE vente ADD CONSTRAINT fkvp
  FOREIGN KEY (npro) REFERENCES produit;
CREATE UNIQUE INDEX produit ON produit(npro);
DROP TABLE produit_tmp;
```

Programmes

D Indépendance totale : la façon et l'endroit où est stockée une table peut influencer les performances de certaines requêtes ou mises à jour dans les programmes. Toutefois, aucune modification n'est utile pour le bon fonctionnement des programmes.

L /

b) Destruction

Données

D Indépendance

S Comme pour la création d'un espace de stockage, les opérations de destruction nécessitent des espaces de stockage vides. Il faut donc utiliser des tables temporaires qui vont contenir les instances des tables à enlever de l'espace de stockage.

Premièrement, on crée les tables temporaires qui vont contenir les instances des tables appartenant à l'espace de stockage et on détruit les tables initiales (voir point a) création d'un espace de stokage).

Deuxièmement, on détruit les espaces de stockage.

```
SQL/Server: ALTER DATABASE database REMOVE FILEGROUP filegroup;
Oracle:     DROP TABLESPACE tablespace;
DB2:        DROP TABLESPACE tablespace;
```

Troisièmement, les tables originales sont recréées avec leurs instances, leurs contraintes et leurs index. Les tables temporaires sont détruites (voir point a).

E Détruire la collection *PRODUITS* contenant la table *PRODUIT*.

```
CREATE TABLE produit_tmp (NPRO char(5) not null,
  LIBELLE char(50) not null,QSTOCK num(10) not null);
INSERT INTO produit_tmp SELECT * FROM produit;
DROP TABLE produit CASCADE CONSTRAINT;
DROP TABLESPACE produits;
CREATE TABLE produit (NPRO char(5) not null,LIBELLE char(50) not null,
  QSTOCK num(10) not null,primary key (NPRO));
INSERT INTO produit SELECT * FROM produit_tmp;
ALTER TABLE lignecom ADD CONSTRAINT fkpl
  FOREIGN KEY (npro) REFERENCES produit;
ALTER TABLE vente ADD CONSTRAINT fkvp
  FOREIGN KEY (npro) REFERENCES produit;
CREATE UNIQUE INDEX produit ON produit(npro);
DROP TABLE produit_tmp;
```

Programmes

D Indépendance totale

L /

c) Renommage

Données

D Indépendance

S Pour renommer un espace de stockage, il faut le détruire et ensuite le reconstruire. Comme pour la création et la destruction, des tables temporaires sont nécessaires.

Premièrement, on crée les tables temporaires qui vont contenir les instances des tables appartenant à l'espace de stockage et on détruit les tables initiales (voir point a).

Deuxièmement, l’espace de stockage est détruit (voir point b).

Troisièmement, l’espace de stockage est reconstruit avec le nouveau nom (voir point a).

Finalement, les tables de l’espace sont recréées (colonnes, contraintes identifiantes et clés étrangères) dans le nouvel espace de stockage. Les données des tables temporaires sont transférées vers les tables originales correspondantes, on recrée les contraintes qui référencent les tables originales ainsi que leurs index. Les tables temporaires sont détruites.

E Renommer la collection *PRODUITS* contenant la table *PRODUIT* en *DBSPC_PROD*.

```
CREATE TABLE produit_tmp (NPRO char(5) not null,
  LIBELLE char(50) not null,QSTOCK num(10) not null);
INSERT INTO produit_tmp SELECT * FROM produit;
DROP TABLE produit CASCADE CONSTRAINT;
DROP TABLESPACE produits;
CREATE TABLESPACE dbspc_prod DATAFILE 'dbspc_prod.dat' SIZE 100K REUSE;
CREATE TABLE produit (NPRO char(5) not null,LIBELLE char(50) not null,
  QSTOCK num(10) not null,primary key (NPRO)) TABLESPACE dbspc_prod;
INSERT INTO produit SELECT * FROM produit_tmp;
ALTER TABLE lignecom ADD CONSTRAINT fkpl
  FOREIGN KEY (npro) REFERENCES prod;
ALTER TABLE vente ADD CONSTRAINT fkvp
  FOREIGN KEY (npro) REFERENCES produit;
CREATE UNIQUE INDEX produit ON produit(npro);
DROP TABLE produit_tmp;
```

Programmes

D Indépendance totale

L /

d) Ajout d'une table

Données

D Indépendance

S Le script d'ajout d'une table dans un espace est identique à celui présenté dans le point a) (excepté les instructions pour la création de l'espace lui-même).

E Ajouter la table *LIGNECOM* à la collection *PRODUITS*.

```
CREATE TABLE lignecom_tmp (QUANTITE num(6,2) not null,
  NPRO char(5) not null, NCOM num(5) not null);
INSERT INTO lignecom_tmp SELECT * FROM lignecom;
DROP TABLE lignecom CASCADE CONSTRAINT;
CREATE TABLE lignecom (NPRO char(5) not null,LIBELLE char(50) not null,
  QSTOCK num(10) not null,primary key (NPRO),
  CONSTRAINT fkpl FOREIGN KEY (npro) REFERENCES produit,
  CONSTRAINT fklc FOREIGN KEY (ncom) REFERENCES commande)
  TABLESPACE produits;
INSERT INTO lignecom SELECT * FROM lignecom_tmp;
CREATE INDEX fkpl ON lignecom(npro);
CREATE INDEX fklc ON lignecom(ncom);
DROP TABLE lignecom_tmp;
```

Programmes

D Indépendance totale

L /

e) Retrait d'une table

Données

D Indépendance

S Le script d'enlèvement d'une table d'un espace de stockage est identique à celui présenté dans la destruction d'un espace (sans les instructions de destruction de l'espace lui-même).

E Enlever la table *LIGNECOM* de l'espace de stockage *PRODUITS*.

```
CREATE TABLE lignecom_tmp (QUANTITE num(6,2) not null,
  NPRO char(5) not null, NCOM num(5) not null);
INSERT INTO lignecom_tmp SELECT * FROM lignecom;
DROP TABLE lignecom CASCADE CONSTRAINT;
CREATE TABLE lignecom (NPRO char(5) not null,LIBELLE char(50) not null,
  QSTOCK num(10) not null,primary key (NPRO),
  CONSTRAINT fkpl FOREIGN KEY (npro) REFERENCES produit,
  CONSTRAINT fklc FOREIGN KEY (ncom) REFERENCES commande);
INSERT INTO lignecom SELECT * FROM lignecom_tmp;
CREATE INDEX fkpl ON lignecom(npro);
CREATE INDEX fklc ON lignecom(ncom);
DROP TABLE lignecom_tmp CASCADE;
```

Programmes

D Indépendance totale

L /

C.3 Modèle E/A de base → Modèle relationnel riche

C.3.1 Introduction

Pour toutes informations complémentaires sur les contraintes des modèles et les correspondances entre les concepts, le lecteur est renvoyé au chapitre 4 (section 4.3 à la page 97).

C.3.2 Modifications de spécifications conceptuelles

C.3.2.1 Modifications relatives aux attributs

Nous allons simplement ajouter une précondition à la modification qui consiste à transformer un attribut facultatif en obligatoire. Pour la représentation graphique de cette modification, il faut consulter le point C.2.2.2 e). Les autres modifications relatives aux attributs restent inchangées dans le modèle E/A de base.

a) Changement d'une contrainte facultative en obligatoire

D Changer un attribut facultatif en attribut obligatoire.

S $\exists$ E $\in$ type-entites(S), A $\in$ attribut(E) : A $\leftarrow$ modifier-Att(S,A,1-1)

C T^-

P
- E $\in$ type-entites(S)
- A $\in$ attribut(E)
- card-min(A) = 0
- $\neg\exists$ g $\in$ groupe(E) : A $\in$ composant(g) $\wedge$ fonction(g) = coexistence | exclusif | au-moins-un | exactement-un

R L'attribut n'appartient pas à un groupe de coexistence, exclusif, au-moins-un ou exactement-un. Si c'est le cas, il faut d'abord retirer l'attribut du groupe avant de le modifier.

Q
- card-min(A) = 1

I $\forall$ i $\in$ instance(A), valeur(i) $\neq$ null

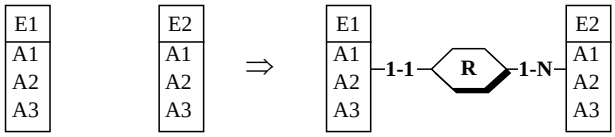
Les instances d'un attribut obligatoire doivent avoir une valeur. Sinon il faut leur en donner une.

C.3.2.2 Modifications relatives aux types d'associations

Pour les modifications relatives aux cardinalités des rôles, seules celles qui impliquent la création ou la suppression d'une contrainte d'égalité au niveau logique sont analysés. Il s'agit des modifications qui augmentent et diminuent la cardinalité minimum d'un rôle.

a) Création

D Créer un type d'associations.



S R ← creer-TA(S,n)

C T^+

P – ∀ R' ∈ type-associations(S) : nom(R') ≠ n

Q – R ∈ type-associations(S)

– nom(R) = n

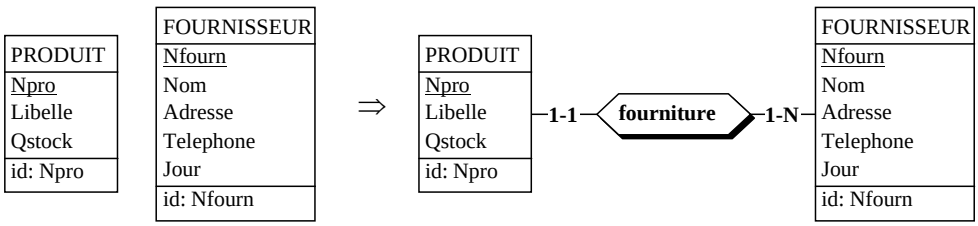
– ro1, ro2 ∈ role(R)

– (card(ro1) = [0-1] ∧ card(ro2) = [0-1]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [1-1]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [0-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [0-N]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [1-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [1-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [1-1])

I ∀ m ∈ [1,2] : ((∃ rom ∈ role(R) : card-min(rom) > 0) ∧ (∀ em ∈ instance(Em), ∃ r ∈ instance(R) : r[rom] = em) ∨ instance(R) = ∅)

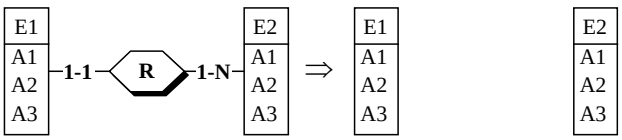
Si un des rôles de R a une cardinalité minimum supérieure à 0, il faut des instances de R pour que la contrainte de cardinalité soit vérifiée.

E On crée un type d'associations *fourniture* (1-N/1-1) entre *FOURNISSEUR* et *PRODUIT*.



b) Destruction

D Supprimer un type d'associations.



S ∃ R ∈ type-associations(S) : () ← supprimer-TA(R)

C T^-

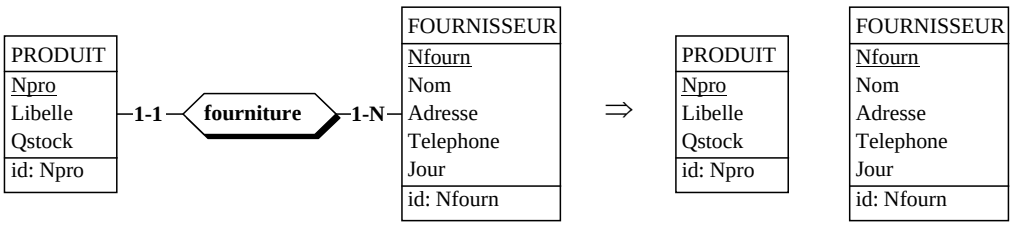
P – R ∈ type-associations(S)

Q – ro1,ro2 ∉ role(R)

– R ∉ type-associations(S)

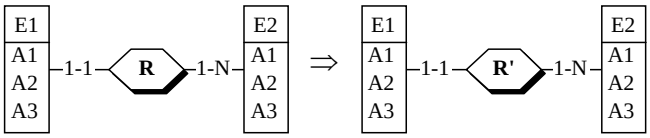
I ∀ i ∈ instance(R) : supprimer(i)

E On supprime le type d'associations *fourniture* entre *PRODUIT* et *FOURNISSEUR*.



c) Renommage

D Renommer un type d'associations.



S ∃ R ∈ type-associations(S) : R ← modifier-TA(R,n')

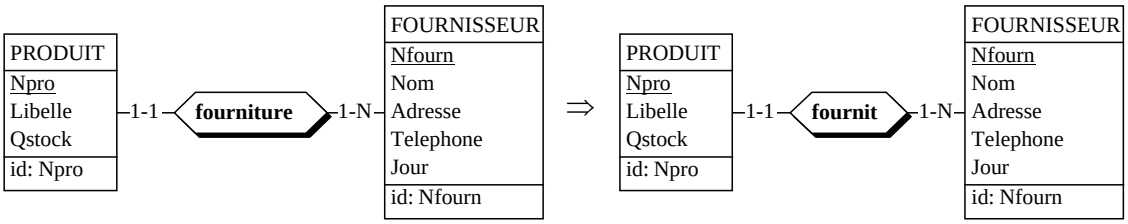
C $T^{=}$

P
- R ∈ type-associations(S)
- nom(R) = n
- ∀ R' ∈ type-associations(S) : nom(R') ≠ n'

Q
- nom(R) = n'

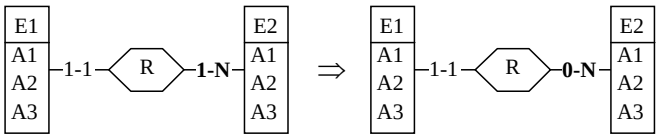
I /

E On renomme le type d'association *fourniture* en *fournit*.



d) Diminution de la cardinalité minimum d'un rôle

D Diminuer la cardinalité minimum du rôle ro2.



S ∃ R ∈ type-associations(S), ro2 ∈ role(R) : ro2 ← modifier-Role(R,ro2,[0-N])

C T^{+}

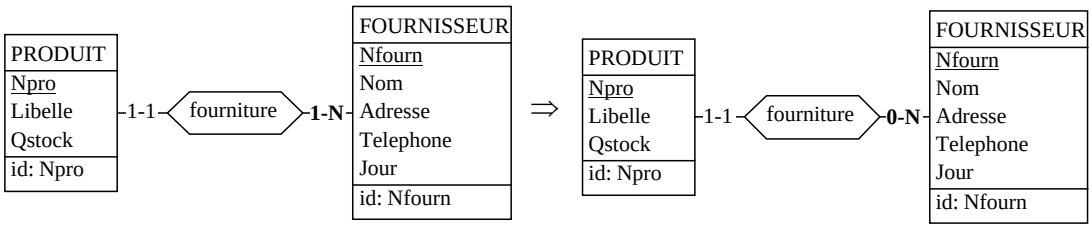
P
- R ∈ type-associations(S)
- {ro1,ro2} ∈ role(R)

- E1,E2 ∈ type-entites(S)
- {E1} = te-role(ro1)
- {E2} = te-role(ro2)
- (card(ro1) = [0-1] ∧ card(ro2) = [1-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [1-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [1-N])

Q – card-min(ro2) = 0

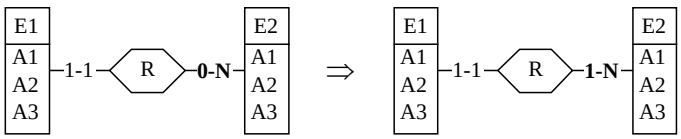
I /

E Supposons que le type d'associations *fourniture* soit du type 1-1/1-N. Un fournisseur peut fournir un ou plusieurs produits alors qu'un produit est fournit par un et un seul fournisseur. On peut diminuer la cardinalité minimum du rôle joué par *FOURNISSEUR* dans *fourniture*. Un fournisseur peut ne fournir aucun produit.



e) Augmentation de la cardinalité minimum d'un rôle

D Augmenter la cardinalité minimum du rôle ro2.



S ∃ R ∈ type-associations(S), ro2 ∈ role(R) : ro2 ← modifier-Role(R,ro2,[1-N])

C T^-

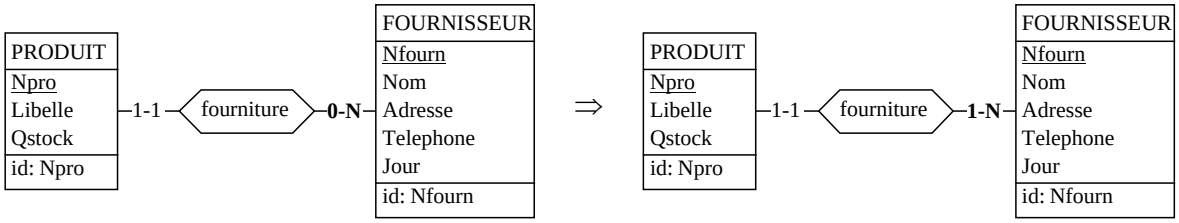
P
- R ∈ type-associations(S)
- {ro1,ro2} ∈ role(R)
- E1,E2 ∈ type-entites(S)
- {E1} = te-role(ro1)
- {E2} = te-role(ro2)
- (card(ro1) = [1-1] ∧ card(ro2) = [0-1]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [0-N]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [0-N])

Q – card-min(ro2) = 1

I ∀ x ∈ instance(E2), ∃ y ∈ instance(ro2) : {x} = te-role(y)

Toutes les instances de E2 doivent au moins jouer un rôle dans une instance de R.

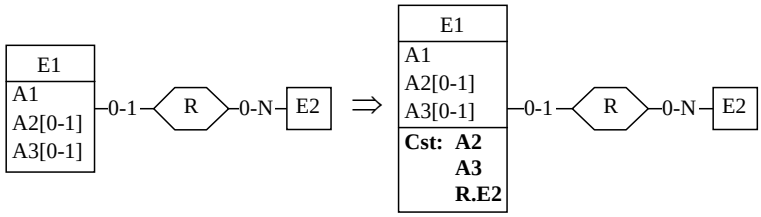
E Supposons que le type d'associations *fourniture* soit du type 1-1/0-N. Un fournisseur peut fournir aucun, un ou plusieurs produits alors qu'un produit est fournit par un et un seul fournisseur. On peut augmenter la cardinalité minimum du rôle joué par FOURNISSEUR dans *fourniture*. Un fournisseur doit fournir au moins un produit.



C.3.2.3 Modifications relatives aux contraintes de coexistence, d'exclusivité, au-moins-un et exactement-un

a) Création

D Ajouter une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S $\exists$ E1 $\in$ type-entites(S) : g $\leftarrow$ creer-Groupe(E1,n,cst,{A2,A3,R.E2})

C T^+

P
- E1 $\in$ type-entites(S)
- $\forall$ g' $\in$ groupe(E1) : nom(g') $\neq$ n
- $\forall$ A $\in$ composant(g) : card(A) = [0-1]
- $\forall$ ro2 $\in$ composant(g) : {ro1,ro2} = role(R) $\wedge$ {E1} = te-role(ro1) $\wedge$ card(ro1) = [0-1]
- cst = coexistence | exclusif | au-moins-un | exactement-un

Q
- g $\in$ groupe(E1)
- fonction(g) = coexistence | exclusif | au-moins-un | exactement-un
- nom(g) = n

I $\forall$ x $\in$ instance(A1), y $\in$ instance(A2), z $\in$ instance(ro2) :

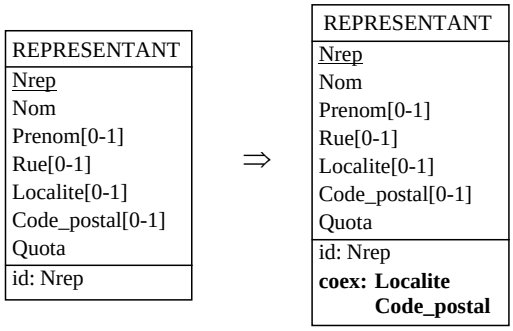
(fonction(g) = coexistence $\wedge$ ((valeur(x) $\neq$ null $\wedge$ valeur(y) $\neq$ null $\wedge$ valeur(z) $\neq$ null) $\vee$ (valeur(x) = null $\wedge$ valeur(y) = null $\wedge$ valeur(z) = null))) $\vee$

(fonction(g) = exclusif $\wedge$ ((valeur(x) = null $\wedge$ valeur(y) = null) $\vee$ (valeur(y) = null $\wedge$ valeur(z) = null) $\vee$ (valeur(x) = null $\wedge$ valeur(z) = null))) $\vee$

(fonction(g) = au-moins-un $\wedge$ (valeur(x) $\neq$ null $\vee$ valeur(y) $\neq$ null $\vee$ valeur(z) $\neq$ null)) $\vee$

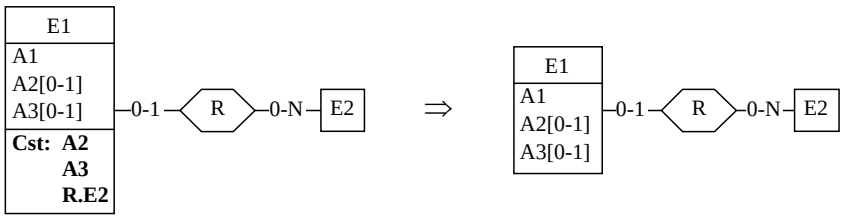
(fonction(g) = exactement-un $\wedge$ ((valeur(x) $\neq$ null $\wedge$ valeur(y) = null $\wedge$ valeur(z) = null) $\vee$ (valeur(x) = null $\wedge$ valeur(y) $\neq$ null) $\wedge$ valeur(z) = null) $\vee$ (valeur(x) = null $\wedge$ valeur(y) = null) $\wedge$ valeur(z) $\neq$ null))

E Ajouter une contrainte de coexistence sur les attributs *Code_postal* et *Localite* du type d'entité *REPRESENTANT*.



b) Destruction

D Détruire une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ E1 ∈ type-entites(S), g ∈ groupe(E1) : g ← supprimer-Groupe(E1,g)

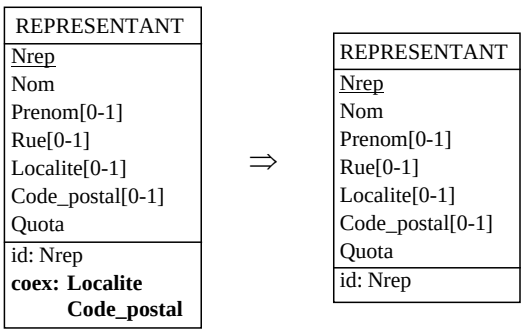
C T^-

P – E1 ∈ type-entites(S)
– g ∈ groupe(E1)
– fonction(g) = cst = coexistence | exclusif | au-moins-un | exactement-un

Q – g ∉ groupe(E1)

I /

E Détruire la contrainte de coexistence sur les attributs *Localite* et *Code_postal* du type d'entité *REPRESENTANT*.



c) Renommage

D Renommer une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.

S ∃ E1 ∈ type-entites(S), g ∈ groupe(E1) : g ← modifier-Groupe(E1,g,n')

C $T^=$

P – E1 ∈ type-entites(S)
– g ∈ groupe(E1)
– fonction(g) = coexistence | exclusif | au-moins-un | exactement-un
– nom(g) = n
– ∀ g' ∈ groupe(E1) : nom(g') ≠ n'

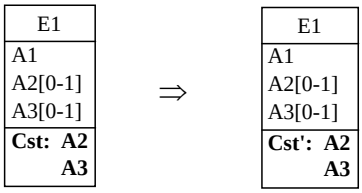
Q – nom(g) = n'

I /

E La contrainte de coexistence *Grrepresentant* contenant les colonnes *Localite* et *Code_postal* du type d'entités *REPRESENTANT* est renommée en *coex_rep*.

d) Changement de type

D Changer le type d'une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ E1 ∈ type-entites(S), g ∈ groupe(E1) : g ← modifier-Groupe(E1,g,cst')

C Indéterminé

P – E1 ∈ type-entites(S)
– g ∈ groupe(E1)
– fonction(g) = cst = coexistence | exclusif | au-moins-un | exactement-un

Q – cst' ≠ cst
– fonction(g) = cst' = coexistence | exclusif | au-moins-un | exactement-un

I ∀ x ∈ instance(A1), y ∈ instance(A2) :

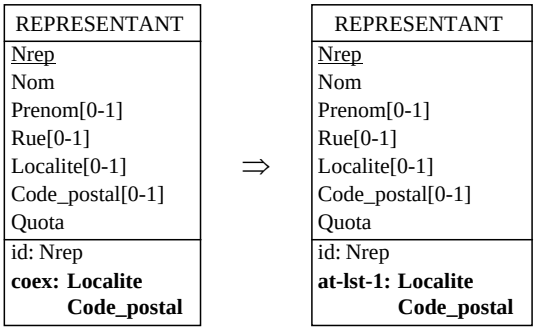
(fonction(g) = coexistence ∧ (valeur(x) ≠ null ∧ valeur(y) ≠ null) ∨ (valeur(x) = null ∧ valeur(y) = null)) ∨

(fonction(g) = exclusif ∧ (valeur(x) = null ∨ valeur(y) = null)) ∨

(fonction(g) = au-moins-un ∧ (valeur(x) ≠ null ∨ valeur(y) ≠ null)) ∨

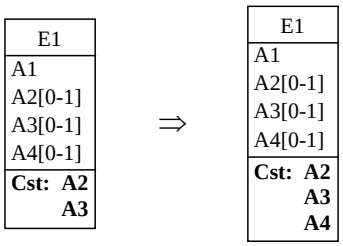
(fonction(g) = exactement-un ∧ ((valeur(x) ≠ null ∧ valeur(y) = null) ∨ (valeur(x) = null ∧ valeur(y) ≠ null)))

E Changer la contrainte de coexistence sur les attributs *Localite* et *Code_postal* du type d'entité *REPRESENTANT* en contrainte au-moins-un.



e) Ajout d'un composant

D Ajouter un composant à une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S $\exists$ E1 $\in$ type-entites(S), g $\in$ groupe(E1), C $\in$ attribut(E1) $\vee$ C $\in$ role(R) : (g,{A2,A3,C}) $\leftarrow$ modifier-Groupe(E1,g,{C})

C T^+

P
- E1 $\in$ type-entites(S)
- fonction(g) = cst = coexistence | exclusif | au-moins-un | exactement-un
- C $\in$ attribut(E1) : card(C) = [0-1]
- C $\in$ role(R) : {ro1,C} = role(R) $\wedge$ {E1} = te-role(ro1) $\wedge$ card(ro1)=[0-1]
- C $\notin$ composant(g)

Q
- C $\in$ composant(g)

I $\forall$ x $\in$ instance(A2), y $\in$ instance(A3), z $\in$ instance(C) :

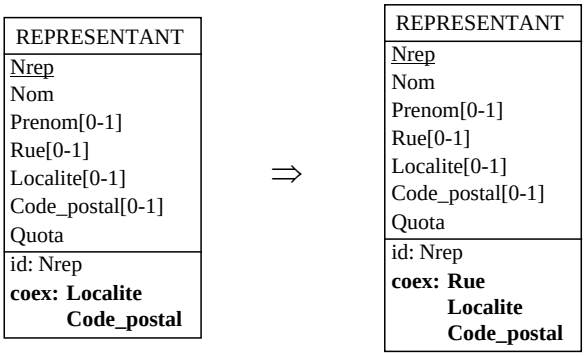
(fonction(g) = coexistence $\wedge$ ((valeur(x) $\neq$ null $\wedge$ valeur(z) $\neq$ null) $\vee$ (valeur(x) = null $\wedge$ valeur(z) = null))) $\vee$

(fonction(g) = exclusif $\wedge$ ((valeur(x) = null $\wedge$ valeur(y) = null) $\vee$ (valeur(x) = null $\wedge$ valeur(z) = null) $\vee$ (valeur(y) = null $\wedge$ valeur(z) = null))) $\vee$

(fonction(g) = exactement-un $\wedge$ valeur(z) = null)

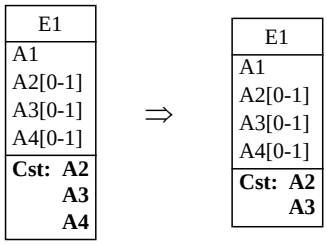
R Ajouter un composant à un groupe nécessite la vérification de la contrainte si son type est coexistence, exclusif ou exactement-un.

E Ajouter l'attribut *Rue* à la contrainte de coexistence sur les attributs *Localite* et *Code_postal* du type d'entité *REPRESENTANT*.



f) Retrait d'un composant

D Enlever un composant à une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S $\exists$ E1 $\in$ type-entites(S), g $\in$ groupe(E1), C $\in$ attribut(E1) $\vee$ C $\in$ role(R) : (g,{A2,A3}) $\leftarrow$ modifier-Groupe(E1,g,{C})

C T^-

P
- E1 $\in$ type-entites(S)
- C $\in$ attribut(E1) $\vee$ C $\in$ role(R)
- C $\in$ composant(g)
- #(composant(g)) > 2

R g a au moins trois composants sinon la contrainte n'aura plus de sens après la modification et elle devra être détruite (voir C.3.2.3 b).

Q
- C $\notin$ composant(g)

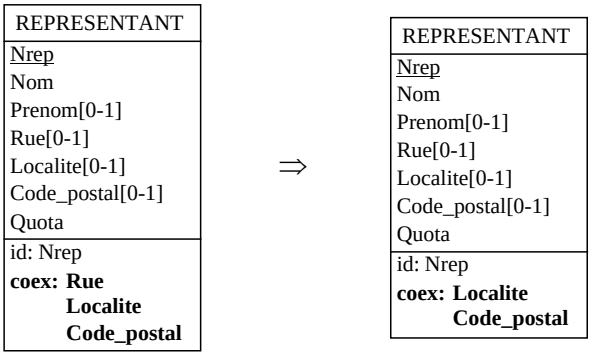
I $\forall$ x $\in$ instance(A2), y $\in$ instance(A3) :

(fonction(g) = au-moins-un $\wedge$ (valeur(x) $\neq$ null $\vee$ valeur(y) $\neq$ null)) $\vee$

(fonction(g) = exactement-un $\wedge$ ((valeur(x) $\neq$ null $\wedge$ valeur(y) = null) $\vee$ (valeur(x) = null $\wedge$ valeur(y) $\neq$ null)))

Enlever un composant d'une contrainte nécessite la vérification de la contrainte si son type est au-moins-un ou exactement-un.

E Enlever l'attribut *Rue* de la contrainte de coexistence contenant les attribut *Localite* et *Code_postal* du type d'entité *REPRESENTANT*.



C.3.3 Modifications de spécifications logiques

Pour les modifications relatives aux contraintes de coexistence, exclusive, au-moins-un et exactement-un, les exemples sont les mêmes que pour les modifications conceptuelles relatives aux contraintes du même type. Les représentations graphiques de ces exemples ne sont pas données. Les lecteurs intéressés doivent consulter les modifications correspondantes dans la section C.3.2.3.

C.3.3.1 Modifications relatives aux colonnes

a) Changement d'une contrainte NULL en NOT NULL

D Changer une colonne facultative en colonne obligatoire.

S $\exists$ T $\in$ type-entites(S), C $\in$ attribut(T) : C $\leftarrow$ modifier-Att(T,C,1-1)

C T^-

P
- T $\in$ type-entites(S)
- C $\in$ attribut(E)
- card-min(C) = 0
- $\neg\exists$ g $\in$ groupe(T) : C $\in$ composant(g) $\wedge$ fonction(g) = coexistence | exclusif | au-moins-un | exactement-un
- $\neg\exists$ g $\in$ reference(T) : C $\in$ composant(g)

R L'attribut n'appartient pas à un groupe de coexistence, exclusif, au-moins-un ou exactement-un. Si c'est le cas, il faut d'abord détruire la contrainte.

C ne peut pas référencer une clé primaire ou candidate. Dans le cas contraire, toutes les colonnes de référence doivent être modifiées (voir point C.3.3.2).

Q
- card-min(C) = 1

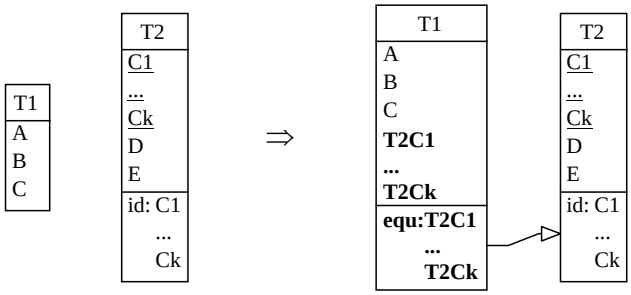
I $\forall$ i $\in$ instance(C), valeur(i) $\neq$ null

Les instances d'une colonne obligatoire doivent avoir une valeur. Sinon il faut leur en donner une.

C.3.3.2 Modifications relatives aux clés étrangères

a) Création

D Créer une ou plusieurs colonnes de référence dans la table de référence ainsi qu'une clé étrangère et une contrainte d'égalité.



S ∃ T1,T2 ∈ type-entites(S) : C1 ← creer-Att(T1,n1) ∧ ... ∧ Ck ← creer-Att(T1,nk) ∧ fk ← creer-FK(T1,n,{C1,...,Ck},{T2.C1,...,T2.Ck},egal)

C T^+

P
- T1,T2 ∈ type-entites(S)
- ∃ id ∈ groupe(T2) : fonction(id) = id-primaire | id-secondaire
- ∀ g' ∈ groupe(T1) : nom(g') ≠ n
- ∀ C ∈ attribut(T1) : nom(C) ≠ n1 ∧ ... ∧ nom(C) ≠ nk

Q
- fk ∈ groupe(T1)
- fonction(fk) = egal
- nom(fk) = n
- C1,...,Ck ∈ attribut(T1)
- nom(C1) = n1 ∧ ... ∧ nom(Ck) = nk
- card(C1) = [0-1] ∧ ... ∧ card(Ck) = [0-1] ∧ fonction(fk) &= coexistence

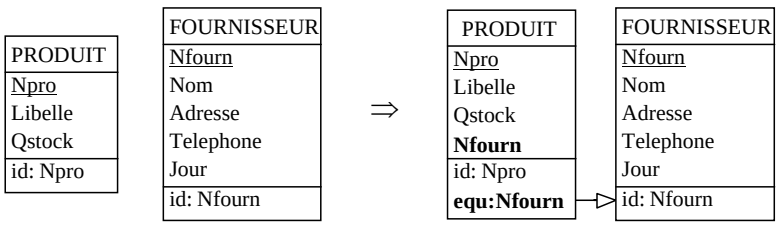
R Si les colonnes de référence sont NULL, on crée une contrainte de coexistence sur le groupe fk.

I ∀ f ∈ instance(fk), ∃ i ∈ instance(id) : valeur(f) = valeur(i)

∀ i ∈ instance(id), ∃ f ∈ instance(fk) : valeur(i) = valeur(f)

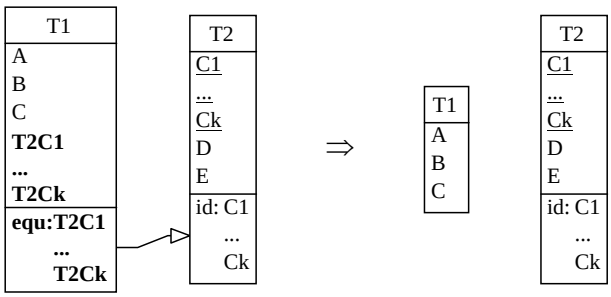
Les instances des colonnes de références doivent avoir les mêmes valeurs qu'une instance des colonnes identifiantes et toutes les instances des colonnes identifiantes doivent être référencées au moins une fois.

E On crée dans la table *PRODUIT* une colonne obligatoire *Nfourn* qui référence l'identifiant primaire *Nfourn* de la table *FOURNISSEUR* (contrainte d'égalité).



b) Destruction

D Supprimer une clé étrangère, sa contrainte d'égalité et les colonnes de référence correspondantes.



S ∃ T1 ∈ type-entités(S), C1,...,Ck ∈ attribut(T1), fk ∈ reference(T1) : () ← supprimer-Att(T1,C1) ∧ ... ∧ () ← supprimer-Att(T1,Ck) ∧ () ← supprimer-FK(T1,fk)

C T^-

P
- T1 ∈ type-entites(S)
- C1,...,Ck ∈ attribut(T1)
- fk ∈ reference(T1)
- ∀ id ∈ groupe(T1), ∃ f ∈ est-reference(id) : {C1,...,Ck} ⊄ composant(id)

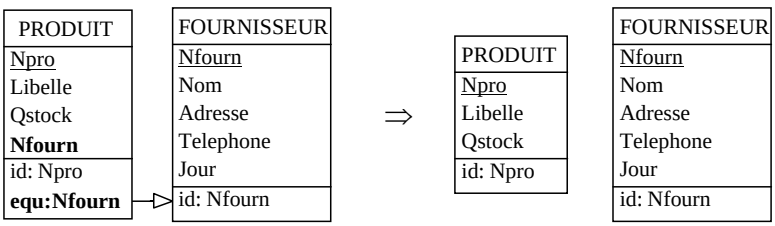
R C1,...,Ck ne peuvent faire partie d'un identifiant référencé par des clés étrangères. Si c'est le cas, il faut d'abord supprimer ces clés.

Q
- C1,...,Ck ∉ attribut(T1)
- fk ∉ groupe(T1)

I ∀ i ∈ instance(C1) : supprimer(i), ..., ∀ i ∈ instance(Ck) : supprimer(i)

Les instances de C1,...,Ck sont perdues. Elles peuvent être sauvegardées pour une exploitation ultérieure (voir point C.3.4.2 b).

E On détruit la contrainte référentielle d'égalité référençant la table *FOURNISSEUR* et la colonne Nfourn dans la table *PRODUIT*.



c) Renommage

D Renommer une clé étrangère avec une contrainte d'égalité.

S ∃ T1 ∈ type-entités(S), fk ∈ reference(T1) : fk ← modifier-FK(T1,fk,n')

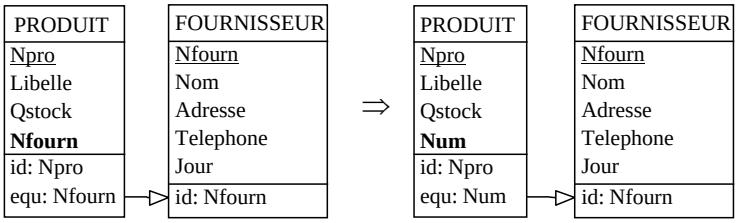
C $T^=$

P
- T1 ∈ type-entites(S)
- fk ∈ reference(T1)
- nom(fk) = n
- ∀ fk' ∈ groupe(T1) : nom(fk') ≠ n'

Q
- nom(fk) = n'

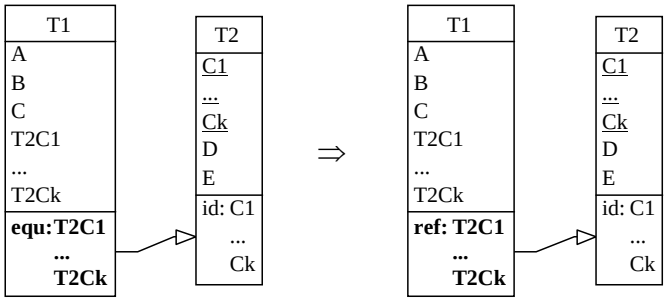
I /

E Dans *PRODUIT*, on renomme la colonne *NFOURN* en *NUM*.



d) Suppression d'une contrainte d'égalité sur une clé étrangère

D Retirer la contrainte d'égalité définie sur une clé étrangère.



S ∃ t1∈ type-entites(S), fk ∈ reference(T1) : fk ← modifier-FK(T1,fk,∧egal)

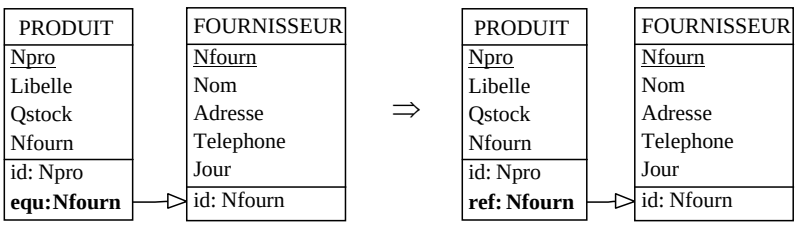
C T^+

P
- T1 ∈ type-entites(S)
- C1,...,Ck ∈ attribut(T1)
- fk ∈ reference(T1)
- fonction(fk) = egal
- {C1,...,Ck} = composant(fk)

Q
- fonction(fk) = ∧egal

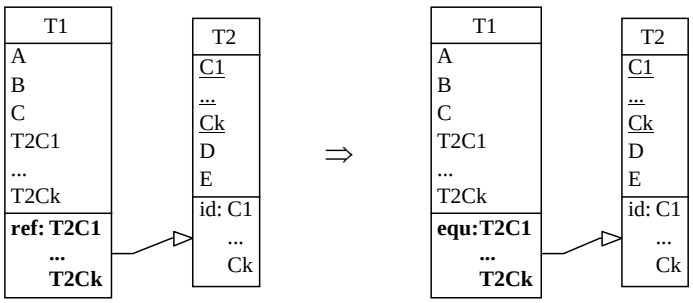
I /

E On supprime la contrainte d'égalité sur la clé étrangère de *PRODUIT* vers *FOURNISSEUR*.



e) Ajout d'une contrainte d'égalité sur une clé étrangère

D Ajouter une contrainte d'égalité sur une clé étrangère.



S ∃ t1∈ type-entites(S), fk ∈ reference(T1) : fk ← modifier-FK(T1,fk,egal)

C T^-

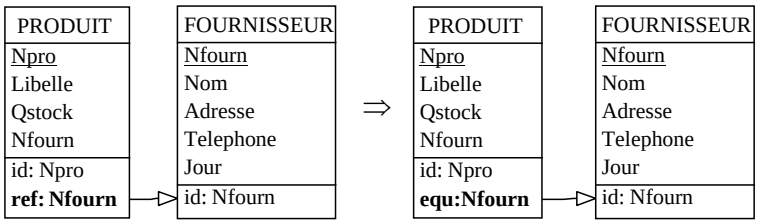
P
- T1 ∈ type-entites(S)
- C1,...,Ck ∈ attribut(T1)
- fk ∈ reference(T1)
- {C1,...,Ck} = composant(fk)

Q
- fonction(fk) = egal

I ∀ i ∈ instance(id), ∃ f ∈ instance(fk) : valeur(i) = valeur(f)

Toutes les instances de id sont référencées par au moins une instance de fk.

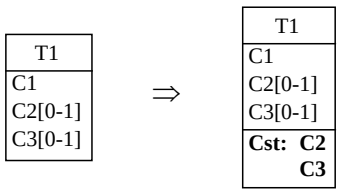
E On ajoute une contrainte d'égalité sur la clé étrangère de *PRODUIT* vers *FOURNISSEUR*.



C.3.3.3 Modifications relatives aux contraintes de coexistence, exclusive, au-moins-un et exactement-un

a) Création

D Ajouter une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ T1 ∈ type-entites(S) : g ← creer-Groupe(T1,n,cst,{C2,C3})

C T^+

P
- T1 ∈ type-entites(S)
- ∀ g' ∈ groupe(E1) : nom(g') ≠ n

– ∀ C ∈ composant(g) : card(C) = [0-1]

Q – g ∈ groupe(T1)

– fonction(g) = coexistence | exclusif | au-moins-un | exactement-un

– nom(g) = n

I ∀ x ∈ instance(C1), y ∈ instance(C2) :

(fonction(g) = coexistence ∧ (valeur(x) ≠ null ∧ valeur(y) ≠ null) ∨ (valeur(x) = null ∧ valeur(y) = null)) ∨

(fonction(g) = exclusif ∧ (valeur(x) = null ∨ valeur(y) = null)) ∨

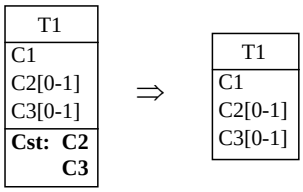
(fonction(g) = au-moins-un ∧ (valeur(x) ≠ null ∨ valeur(y) ≠ null)) ∨

(fonction(g) = exactement-un ∧ ((valeur(x) ≠ null ∧ valeur(y) = null) ∨ (valeur(x) = null ∧ valeur(y) ≠ null)))

E Ajouter une contrainte de coexistence à la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal*.

b) Destruction

D Détruire une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ T1 ∈ type-entites(S), g ∈ groupe(T1) : () ← supprimer-Groupe(T1,g)

C T^-

P – T1 ∈ type-entites(S)

– g ∈ groupe(T1)

– fonction(g) = coexistence | exclusif | au-moins-un | exactement-un

– g ∉ reference(T1)

R On ne peut détruire une contrainte de coexistence dont toutes les colonnes appartiennent également à une clé étrangère (sinon cette clé devient incohérente par rapport au modèle).

Q – g ∉ groupe(T1)

I /

E Détruire la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal*.

c) Renommage

D Renommer une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.

S ∃ T1 ∈ type-entites(S), g ∈ groupe(T1) : g ← modifier-Groupe(T1,g,n')

C $T^=$

P – T1 ∈ type-entites(S)

– g ∈ groupe(T1)

– fonction(g) = coexistence | exclusif | au-moins-un | exactement-un

– nom(g) = n

– ∀ g' ∈ groupe(T1) : nom(g') ≠ n'

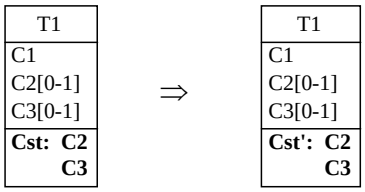
Q – nom(g) = n'

I /

E La contrainte de coexistence *Grrepresentant* contenant les colonnes *Localite* et *Code_postal* du type d'entités *REPRESENTANT* est renommée en *coex_rep*.

d) Changement de type

D Changer le type d'une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ T1 ∈ type-entites(S), g ∈ groupe(T1) : g ← modifier-Groupe(T1,g,cst')

C Indéterminé

P – T1 ∈ type-entites(S)

– g ∈ groupe(T1)

– cst = coexistence | exclusif | au-moins-un | exactement-un

– g ∉ reference(T1)

R On ne peut changer le type d'une contrainte de coexistence dont toutes les colonnes appartiennent également à une clé étrangère (contrainte du modèle).

Q – cst' ≠ cst

– cst' = coexistence | exclusif | au-moins-un | exactement-un

I ∀ x ∈ instance(C1), y ∈ instance(C2) :

(fonction(g) = coexistence ∧ (valeur(x) ≠ null ∧ valeur(y) ≠ null) ∨ (valeur(x) = null ∧ valeur(y) = null)) ∨

(fonction(g) = exclusif ∧ (valeur(x) = null ∨ valeur(y) = null)) ∨

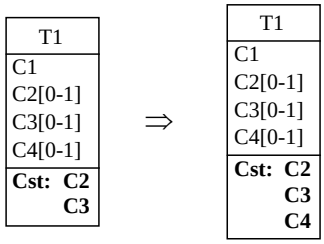
(fonction(g) = au-moins-un ∧ (valeur(x) ≠ null ∨ valeur(y) ≠ null)) ∨

(fonction(g) = exactement-un ∧ ((valeur(x) ≠ null ∧ valeur(y) = null) ∨ (valeur(x) = null ∧ valeur(y) ≠ null)))

E Changer la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal* en contrainte au-moins-un.

e) Ajout d'un composant

D Ajouter une colonne à une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ T1 ∈ type-entites(S), g ∈ groupe(T1), C ∈ attribut(T1) : (g,{C2,C3,C}) ← modifierGroupe(T1,g,{C})

C T^+

P
- T1 ∈ type-entites(S)
- C ∈ attribut(T1) : card(C) = [0-1]
- fonction(g) = coexistence | exclusif | au-moins-un | exactement-un
- C ∉ composant(g)

Q
- C ∈ composant(g)

I ∀ x ∈ instance(C2), y ∈ instance(C3), z ∈ instance(C)

(fonction(g) = coexistence ∧ ((valeur(x) ≠ null ∧ valeur(z) ≠ null) ∨ (valeur(x) = null ∧ valeur(z) = null))) ∨

(fonction(g) = exclusif ∧ ((valeur(x) = null ∧ valeur(y) = null) ∨ (valeur(x) = null ∧ valeur(z) = null) ∨ (valeur(y) = null ∧ valeur(z) = null))) ∨

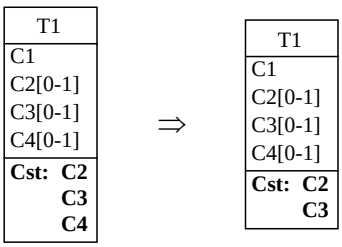
(fonction(g) = exactement-un ∧ valeur(z) = null)

Ajouter un composant à un groupe nécessite la vérification de la contrainte si son type est coexistence, exclusif ou exactement-un.

E Ajouter la colonne *Rue* à la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal*.

f) Retrait d'un composant

D Enlever une colonne à une contrainte de coexistence, d'exclusivité, au-moins-un ou exactement-un.



S ∃ T1 ∈ type-entites(S), g ∈ groupe(T1), C ∈ attribut(T1) : (g,{C2,C3}) ← modifierGroupe(T1,g,{C})

C T^-

P
- T1 ∈ type-entites(S)
- C ∈ attribut(T1)
- C ∈ composant(g)

– #(composant(g)) > 2

– g ∉ reference(T1)

R g a au moins trois composants sinon la contrainte n'aura plus de sens après la modification. Dans ce cas, elle doit être détruite (voir C.3.3.3 b).

On ne peut enlever une colonne d'une contrainte de coexistence s'il existe une clé étrangère contenant les mêmes composants dans le même ordre (contrainte du modèle).

Q – C ∉ composant(g)

I ∀ x ∈ instance(C2), y ∈ instance(C3) :

(fonction(g) = au-moins-un ∧ (valeur(x) ≠ null ∨ valeur(y) ≠ null)) ∨

(fonction(g) = exactement-un ∧ ((valeur(x) ≠ null ∧ valeur(y) = null) ∨ (valeur(x) = null ∧ valeur(y) ≠ null)))

Enlever un composant d'une contrainte nécessite la vérification de la contrainte si son type est au-moins-un ou exactement-un.

E Enlever la colonne *Rue* à la contrainte de coexistence de la table *REPRESENTANT*.

C.3.4 Modifications des structures, des données et des programmes

C.3.4.1 Modifications relatives aux colonnes

a) Création d'une contrainte NOT NULL

Voir point C.2.5.2 e) : Création d'une contrainte NOT NULL.

C.3.4.2 Modifications relatives aux clés étrangères

a) Création

Données

D Dépendance : si les tables (de référence et référencée) contiennent déjà des instances, les colonnes de référence doivent contenir des instances qui correspondent à celles de l'identifiant référencé et chaque instance de l'identifiant doit être référencée par au moins une instance des colonnes de référence.

S Voir la création d'une clé étrangère avec les colonnes de références (point C.2.5.3 a). La vérification de la contrainte d'égalité peut être traduite par :

– une clause CHECK sur la table référencée :

```
ALTER TABLE t2 ADD CONSTRAINT chk_equ
  CHECK(EXISTS(SELECT * FROM t1 WHERE (t2.c1 = t2c1 AND ... AND t2.ck = t2ck)));
```

– un TRIGGER (syntaxe Oracle 8) :

```
CREATE TRIGGER trig_equ
  BEFORE INSERT OR UPDATE OF c1,...,ck ON t2
  REFERENCING NEW AS NEW OLD AS OLD
  FOR EACH ROW
  DECLARE
    x NUMBER;
    violated_constraint EXCEPTION;
  BEGIN
    SELECT COUNT(*) INTO x FROM t1
    WHERE (t2c1 = :new.c1 AND ... AND t2ck = :new.ck);
    IF x = 0 THEN RAISE violated_constraint; END IF;
  END;
```

– une ASSERTION (Oracle 8)

```
CREATE ASSERTION ass_equ
  CHECK(NOT EXISTS(SELECT * FROM t2 WHERE (c1,...,ck) NOT IN
                   (SELECT t2c1,...,t2ck FROM t1)));
```

– une STORED PROCEDURE (syntaxe Oracle 8)

```
CREATE OR REPLACE PROCEDURE verify_constraint_equ IS
    x NUMBER;
    violated_constraint EXCEPTION;
  BEGIN
    SELECT COUNT(*) FROM t2 INTO x
      WHERE (c1,...,ck) NOT IN (SELECT t2c1,...,t2ck FROM t1);
    IF x = 0 THEN RAISE violated_constraint; END IF;
  END;
```

Il faut peut-être créer aussi une contrainte de coexistence sur les colonnes de référence (voir point C.3.4.3 a) :

```
[ALTER TABLE t1 ADD CONSTRAINT chk_coex
   CHECK((t2c1 IS NOT NULL AND ...) OR (t2c1 IS NULL AND ...));]
```

R – Certains SGBDR n'acceptent pas de référence à des colonnes d'autres tables dans un clause CHECK. Le premier script est donc interdit dans certains cas (voir le tableau de la page 36).

– Dans [Dat94], une ASSERTION est définie comme une contrainte générale, d'une complexité arbitraire, impliquant une collection arbitraire de colonnes d'une collection arbitraire de tables. Peu de SGBDR fournissent ce mécanisme. La deuxième solution est donc rarement utilisable.

– Certains SGBDR n'offrent pas la possibilité de reporter la vérification des contraintes à un moment déterminé (au COMMIT par exemple). Dans ce cas, les trois premières solutions (check, assertion et trigger) posent problème. La vérification de la contrainte d'égalité doit se faire lorsque les données concernant les deux tables ont été modifiées, créées ou détruites. Au moment de la création d'une instance de la table référencée, il faudrait qu'elle soit référencée par une instance de la table de référence. C'est impossible puisqu'une instance de la table de référence ne peut référencer qu'une instance existante de la table référencée. Il s'agit d'une situation d'interblocage. La solution idéale consiste à traduire de telles contraintes par du code dans l'application qui vérifiera les contraintes lorsque toutes les données seront créées ou modifiées. C'est le concepteur qui décide de l'emploi de ces fonctions. La quatrième solution semble donc la meilleure solution.

E On crée dans la table *PRODUIT* une colonne obligatoire *Nfourn* qui référence l'identifiant primaire *Nfourn* de la table *FOURNISSEUR* (contrainte d'égalité).

```
ALTER TABLE produit ADD nfourn NUMERIC(5) NOT NULL;
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (nfourn) REFERENCES fournisseur (nfourn);
ALTER TABLE fournisseur ADD CONSTRAINT chk_produit
  CHECK(EXISTS(SELECT * FROM produit WHERE produit.nfourn = nfourn));
```

Programmes

D Dépendance structurelle : voir le point C.2.5.3 a) à la page 82.

L Recherche sur le nom de la table de référence sur laquelle on a crée la clé étrangère.

b) Destruction

Données

D Indépendance : les instances de la base de données vérifient toujours le schéma.

S

```
ALTER TABLE t1 DROP CONSTRAINT fk_t2;
<ALTER TABLE t2 DROP CONSTRAINT chk_equ; |
```

```
 DROP TRIGGER trig_equ; | DROP ASSERTION ass_equ;>
ALTER TABLE t1 DROP t2c1;
...;
ALTER TABLE t1 DROP t2ck;
```

I Les instances supprimées peuvent être stockées dans une table annexe (voir le point C.2.5.3 b) à la page 83).

E On détruit la clé étrangère et la colonne *Nfourn* dans la table *PRODUIT*.

```
ALTER TABLE produit DROP CONSTRAINT fk_fournisseur;
ALTER TABLE fournisseur DROP CONSTRAINT chk_produit;
ALTER TABLE produit DROP nfourn;
```

Programmes

D Dépendance structurelle : dans les instructions de manipulation de données, il faut enlever la gestion des colonnes de référence dans des variables ou des curseurs. Dans l'interface, il faut supprimer la gestion du lien entre les deux tables.

L Recherche sur le nom des colonnes de références ainsi que les variables qui en dépendent.

c) Renommage

Données

D Indépendance

S Voir le point C.2.5.3 c) à la page 84. Les contraintes de coexistence sur les colonnes de référence ainsi que la gestion de la contrainte d'égalité (CHECK, TRIGGER, ASSERTION ou STORED PROCEDURE) doivent être détruites puis recréées pour tenir compte des colonnes de référence renommées.

E Dans PRODUIT, on renomme la colonne NUM en NFOURN.

```
ALTER TABLE produit ADD nfourn NUMERIC(5) DEFAULT 0 NOT NULL;
UPDATE produit SET nfourn = num;
ALTER TABLE fournisseur DROP CONSTRAINT chk_produit;
ALTER TABLE produit DROP num CASCADE;
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (nfourn) REFERENCES fournisseur (nfourn);
ALTER TABLE fournisseur ADD CONSTRAINT chk_produit
  CHECK(EXISTS(SELECT * FROM produit WHERE produit.nfourn = nfourn));
```

Programmes

D Indépendance structurelle : il faut renommer les colonnes de référence dans les requêtes où elles apparaissent et, dans l'interface, on pourrait éventuellement modifier les intitulés des champs où sont gérés la clé étrangère. La structure des programmes reste inchangée.

L Recherche sur le nom des anciennes colonnes de référence pour les remplacer par les nouveaux noms.

d) Destruction d'une contrainte d'égalité

Données

D Indépendance : la suppression d'une contrainte n'influence pas l'intégrité des instances de la base.

S On va détruire la contrainte qui permettait d'exprimer l'égalité de la clé étrangère par une des instructions suivantes :

```
ALTER TABLE t1 DROP CONSTRAINT chk_equ; |
DROP ASSERTION ass_equ; |
```

```
DROP TRIGGER trig_equ;
DROP PROCEDURE proc_equ;
```

E On supprime la contrainte d'égalité de la clé étrangère de *PRODUIT* vers *FOURNISSEUR*.

```
ALTER TABLE fournisseur DROP CONSTRAINT chk_produit;
```

Programmes

D Dépendance structurelle : les requêtes sont inchangées et, dans les interfaces, il faut supprimer la gestion de la contrainte d'égalité.

L Recherche sur le nom des colonnes de référence et des variables qui en dépendent.

e) Création d'une contrainte d'égalité

Données

D Dépendance

I Il faut vérifier que la contrainte d'égalité est respectée. Toutes les instances de la table référencée doivent être référencées par au moins une instance de la table de référence.

```
SELECT * FROM t2 WHERE c1,...,ck NOT IN (SELECT t2c1,...,t2ck FROM t1);
```

Si la requête donne un résultat, la contrainte n'est pas vérifiée et les instances de la table référencée qui ne la vérifie pas doivent être transférées dans une table annexe.

R Avant l'ajout de la contrainte, la base pouvait contenir des fournisseurs qui ne fournissaient aucun produit. Maintenant, ce n'est plus autorisé. Il faut enlever de la base les fournisseurs non autorisés. On pourrait envisager de les stocker dans une table temporaire ou de les laisser dans la table d'origine moyennant quelques aménagements (par exemple, ajouter une colonne de status qui indique que le fournisseur doit être complété en lui ajoutant un produit). La deuxième solution est intéressante dans le cas d'une base de données en cours de remplissage (où toutes les données ne seraient pas encore enregistrées).

S On va ajouter la contrainte qui permettait d'exprimer l'égalité de la clé étrangère.

Soit on ajoute une contrainte CHECK à la table référencée.

```
ALTER TABLE t2 ADD CONSTRAINT chk_equ
  CHECK(EXISTS(SELECT * FROM t1 WHERE (t1.t2c1 = t2.c1 AND ...
                                       AND t1.t2ck = t2.ck)));
```

Soit on définit la contrainte par un TRIGGER.

```
 CREATE TRIGGER trig_equ
  BEFORE INSERT OR UPDATE OF c1,...,ck ON t2
  REFERENCING NEW AS NEW OLD AS OLD
  FOR EACH ROW
  DECLARE
    x NUMBER;
    violated_constraint EXCEPTION;
  BEGIN
    SELECT COUNT(*) INTO x FROM t1
      WHERE (t2c1 = :new.c1 AND ... AND t2ck = :new.ck);
    IF x = 0 THEN RAISE violated_constraint; END IF;
  END;
```

Soit on définit la contrainte par une ASSERTION.

```
CREATE ASSERTION ass_equ
  CHECK(NOT EXISTS(SELECT * FROM t2 WHERE (c1,...,ck) NOT IN
                                          (SELECT t2c1,...,t2ck FROM t1)));
```

Soit on ajoute le code qui permettait de vérifier cette contrainte dans une procedure stockée.

E On ajoute une contrainte d'égalité à la clé étrangère de *PRODUIT* vers *FOURNISSEUR*.

```
ALTER TABLE fournisseur ADD CONSTRAINT chk_produit
  CHECK(EXISTS(SELECT * FROM produit WHERE produit.num = nfourn));
```

Programmes

D Dépendance structurelle : il faut ajouter la gestion de la contrainte d'égalité dans les interfaces.

L Recherche sur le nom des colonnes de référence et des variables qui en dépendent.

C.3.4.3 Modifications relatives aux contraintes de coexistence, exclusive, au-moins-un et exactement-un

a) Création

Données

D Dépendance : les instances de la base doivent vérifier la nouvelle contrainte.

I Premièrement, il faut vérifier que les instances de la base vérifient la contrainte (de coexistence).

```
SELECT * FROM t1 WHERE (c2 IS NULL OR c3 IS NULL) AND
                       (c2 IS NOT NULL OR c3 IS NOT NULL);
```

Si la requête donne un résultat, la contrainte n'est pas respectée. On peut éventuellement transférer les instances qui ne la vérifient pas dans une table temporaire

```
-- Créer une table temporaire de même structure que la table d'origine.
CREATE TABLE t1_tmp (c1 datatype [DEFAULT value NOT NULL], ...);
-- Insérer dans t1_tmp les instances de t1 qui ne respecte pas
-- la contrainte de coexistence.
INSERT INTO t1_tmp
  SELECT * FROM t1 WHERE (c2 IS NULL OR c3 IS NULL) AND
                         (c2 IS NOT NULL OR c3 IS NOT NULL);
-- Supprimer dans t1 les instances qui ne respecte pas la contrainte
DELETE FROM t1 WHERE (c2 IS NULL OR c3 IS NULL) AND
                     (c2 IS NOT NULL OR c3 IS NOT NULL);
```

S Il faut créer la contrainte soit par un CHECK, un TRIGGER ou une ASSERTION. Pour la contrainte de coexistence, on va donner les trois syntaxes et pour les autres contraintes, on se contentera du CHECK.

Contrainte de coexistence :

```
<ALTER TABLE t1 ADD CONSTRAINT chk_coex
 CHECK((c2 IS NOT NULL AND c3 IS NOT NULL) OR (c2 IS NULL AND c3 IS NULL)); |
 CREATE TRIGGER trig_coex
   BEFORE INSERT OR UPDATE OF c2,c3 ON table
   REFERENCING NEW AS NEW OLD AS OLD
   FOR EACH ROW
   WHEN ((NEW.c2 IS NULL OR NEW.c3 IS NULL) AND
         (NEW.c2 IS NOT NULL OR NEW.c3 IS NOT NULL))
   DECLARE
     violated_constraint EXCEPTION;
   BEGIN
     RAISE violated_constraint;
   END; |
 CREATE ASSERTION ass_coex
   CHECK(NOT EXISTS(SELECT * FROM t1 WHERE (c2 IS NULL OR c3 IS NULL)
                                    AND (c2 IS NOT NULL OR c3 IS NOT NULL)));>
```

Contrainte d'exclusivité :

```
ALTER TABLE t1 ADD CONSTRAINT chk_excl
  CHECK((c2 IS NOT NULL AND column3 IS NULL)
        OR (c3 IS NOT NULL AND c2 IS NULL) OR (c2 IS NULL AND c3 IS NULL));
```

Contrainte au-moins-un :

```
ALTER TABLE t1 ADD CONSTRAINT chk_atlst1
  CHECK(c2 IS NOT NULL OR c3 IS NOT NULL);
```

Contrainte exactement-un :

```
ALTER TABLE t1 ADD CONSTRAINT chk_exact1
  CHECK((c2 IS NOT NULL AND c3 IS NULL) OR (c3 IS NOT NULL AND c2 IS NULL));
```

E Ajouter une contrainte de coexistence à la table *REPRESENTANT* qui implique les colonnes *Localite* et *Code_postal*.

```
CREATE TABLE representant_tmp
  (nrep NUMERIC(5) NOT NULL,nom CHAR(50) not NULL,prenom CHAR(30),
   rue CHAR(50),localite CHAR(30),code_postal NUMERIC(4),
   quota NUMERIC(8,2) NOT NULL);
INSERT INTO representant_tmp
  SELECT * FROM representant WHERE
    (localite IS NULL OR code_postal IS NULL)
    AND (localite IS NOT NULL OR code_postal IS NOT NULL);
DELETE FROM representant WHERE (localite IS NULL OR code_postal IS NULL)
                  AND (localite IS NOT NULL OR code_postal IS NOT NULL);
ALTER TABLE representant ADD CONSTRAINT chk_representant
  CHECK((localite IS NOT NULL AND code_postal IS NOT NULL)
        OR (localite IS NULL AND code_postal IS NULL));
```

Programmes

D Dépendance structurelle : dans les interfaces, il faut vérifier la contrainte au moment de la saisie des données.

L Recherche sur le nom de la table, des colonnes participant à la contrainte ainsi que les variables qui en dépendent.

b) Destruction

Données

D Indépendance

S Il faut détruire une contrainte CHECK, TRIGGER ou ASSERTION.

```
<ALTER TABLE t1 DROP CONSTRAINT chk_cst; |
 DROP TRIGGER trig_cst; |
 DROP ASSERTION ass_cst;>
```

E Détruire la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal*.

```
ALTER TABLE representant DROP CONSTRAINT chk_representant;
```

Programmes

D Dépendance structurelle : dans les interfaces, il ne faut plus vérifier la contrainte au moment de la saisie des données.

L Recherche sur le nom de la table, des colonnes participant à la contrainte ainsi que les variables qui en dépendent.

c) Renommage

Données

D Indépendance

S Il faut détruire la contrainte CHECK, TRIGGER ou ASSERTION (voir point b) et la reconstruire avec le nouveau nom (voir point a).

E Renommer la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal*.

```
ALTER TABLE representant DROP CONSTRAINT chk_representant;
```

```
ALTER TABLE representant ADD CONSTRAINT chk_rep_adr
  CHECK((localite IS NOT NULL AND code_postal IS NOT NULL)
        OR (localite IS NULL AND code_postal IS NULL));
```

Programmes

D Indépendance totale

L /

d) Changement de type

Données

D Dépendance

I Seul le changement d'une contrainte exactement-un en au-moins-un n'entraîne aucun problème au niveau des instances. Il faut vérifier si les instances sont conformes à la nouvelle contrainte (voir point a).

S Il faut détruire une contrainte CHECK, TRIGGER, ASSERTION (voir point b) et recréer une contrainte de type différent (voir point a).

E Changer la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Localite* et *Code_postal* en contrainte au-moins-un.

```
CREATE TABLE representant_tmp
  (nrep NUMERIC(5) NOT NULL,nom CHAR(50) not NULL,prenom CHAR(30),
   rue CHAR(50),localite CHAR(30),code_postal NUMERIC(4),
  quota NUMERIC(8,2) NOT NULL);
INSERT INTO representant_tmp
  SELECT * FROM representant WHERE (localite IS NULL AND code_postal IS NULL);
DELETE FROM representant WHERE (localite IS NULL AND code_postal IS NULL);
ALTER TABLE representant ADD CONSTRAINT chk_representant
  CHECK(localite IS NOT NULL OR code_postal IS NOT NULL);
```

Programmes

D Dépendance structurelle : dans les interfaces, il faut changer la vérification de la contrainte au moment de la saisie des données.

L Recherche sur le nom de la table, les colonnes participant à la contrainte ainsi que les variables qui en dépendent.

e) Ajout d'un composant

Données

D Dépendance

I Il faut vérifier si les instances sont conformes à la contrainte contenant une colonne supplémentaire (voir point a). L'ajout d'une colonne à une contrainte peut entraîner que cette contrainte ne soit plus vérifiée pour toutes les instances d'une table. Seule l'ajout d'une colonne à une contrainte au-moins-un ne pose aucun problème.

S Il faut détruire la contrainte (voir point b) et ensuite la recréer en ajoutant la colonne (voir point a).

E Ajouter la colonne *Prenom* à la contrainte de coexistence de la table *REPRESENTANT* qui concerne les colonnes *Code_postal* et *Localite*.

```
CREATE TABLE representant_tmp
  (nrep NUMERIC(5) NOT NULL,nom CHAR(50) not NULL,prenom CHAR(30),
   rue CHAR(50),localite CHAR(30),code_postal NUMERIC(4),
   quota NUMERIC(8,2) NOT NULL);
```

```
INSERT INTO representant_tmp
  SELECT * FROM representant
  WHERE (localite IS NULL OR code_postal IS NULL OR prenom IS NULL)
         AND (localite IS NOT NULL OR code_postal IS NOT NULL OR
              prenom IS NOT NULL);
DELETE FROM representant
  WHERE (localite IS NULL OR code_postal IS NULL OR prenom IS NULL)
        AND (localite IS NOT NULL OR code_postal IS NOT NULL OR
             prenom IS NOT NULL);
ALTER TABLE representant DROP CONSTRAINT chk_representant;
ALTER TABLE representant ADD CONSTRAINT chk_representant
  CHECK((localite IS NOT NULL AND code_postal IS NOT NULL AND
         prenom IS NOT NULL)
       OR (localite IS NULL AND code_postal IS NULL AND prenom IS NULL));
```

Programmes

D Dépendance structurelle : dans les interfaces, il faut changer la vérification de la contrainte au moment de la saisie des données.

L Recherche sur le nom de la table et des colonnes participant à la contrainte ainsi que les variables qui en dépendent.

f) Retrait d'un composant

Données

D Dépendance

I Il faut vérifier si les instances sont conformes à la contrainte avec une colonne en moins (voir point a). Le retrait d'une colonne d'une contrainte peut entraîner que cette contrainte ne soit plus vérifiée pour toutes les instances d'une table. Seul le retrait d'une colonne de contraintes de coexistence et d'exclusion ne pose pas de problème.

S Il faut détruire la contrainte (voir point b) et ensuite la recréer sans la colonne (voir point a).

E Enlever la colonne *Prenom* à la contrainte de coexistence de la table *REPRESENTANT*.

```
ALTER TABLE representant DROP CONSTRAINT chk_representant;
ALTER TABLE representant ADD CONSTRAINT chk_representant
  CHECK((localite IS NOT NULL AND code_postal IS NOT NULL)
       OR (localite IS NULL AND code_postal IS NULL));
```

Programmes

D Dépendance structurelle : dans les interfaces, il faut changer la vérification de la contrainte au moment de la saisie des données.

L Recherche sur le nom de la table et des colonnes participant à la contrainte ainsi que les variables qui en dépendent.

C.4 Modèle E/A riche → Modèle E/A de base

C.4.1 Introduction

Pour toutes informations complémentaires sur les contraintes des modèles et les correspondances entre les concepts, le lecteur est renvoyé au chapitre 4 (section 4.4 à la page 105).

C.4.2 Modifications de spécifications dans un schéma E/A riche

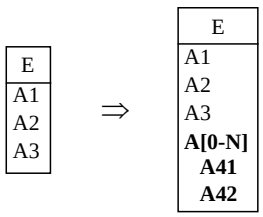
C.4.2.1 Modifications relatives aux attributs décomposables et/ou multivalués

Ce point étudie la création, la destruction ou la modification d'attributs décomposables et/ou multivalués. Ces structures conceptuelles ne sont pas directement exprimables dans un schéma logique relationnel. Elles doivent être transformées pour obtenir des structures conformes au modèle logique. Les transformations possibles sont :

- la transformation d'un attribut en un type d'entités (par instance ou par valeur),
- la désagrégation d'un attribut décomposable,
- la transformation d'un attribut multivalué en une série d'attributs monovalués (ou instanciation),
- la transformation d'un attribut multivalué en un attribut monovalué (ou concaténation).

a) Création

D Créer un attribut décomposable et/ou multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S) : A $\leftarrow$ creer-Att(P,n)

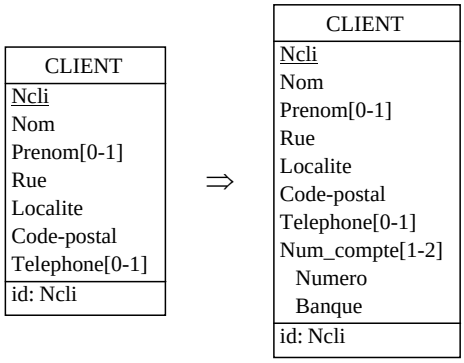
C T^+

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- $\forall$ A' $\in$ attribut(P) : nom(A') $\neq$ n

Q
- A $\in$ attribut(P)
- nom(A) = n

I $\forall$ i $\in$ instance(A) : (card-min(A) = 0 $\wedge$ valeur(i) = null) $\vee$ (card-min(A) = 1 $\wedge$ valeur(i) = valeur-def)

E Ajouter l'attribut décomposable multivalué *Num_compte* au type d'entités *CLIENT*. Il a comme sous-attributs *Numero* et *Banque*. Sa cardinalité est [1-2].



b) Destruction

D Détruire un attribut décomposable et/ou multivalué.

S – $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : () $\leftarrow$ supprimer-Att(P,A)

C T^-

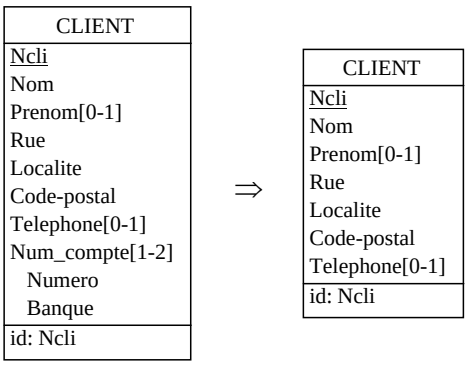
P – P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)

– A $\in$ attribut(P)

Q – A $\notin$ attribut(P)

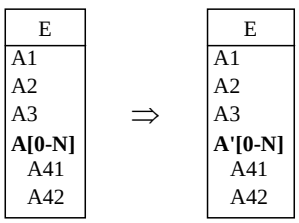
I $\forall$ i $\in$ instance(A) : supprimer(i)

E Détruire l'attribut *Num_compte* du type d'entité *CLIENT*.



c) Renommage

D Renommer un attribut décomposable et/ou multivalué.



S ∃ P ∈ type-entites(S) ∪ type-associations(S) ∪ attribut(S), A ∈ attribut(P) : A ← modifier-Att(P,A,n')

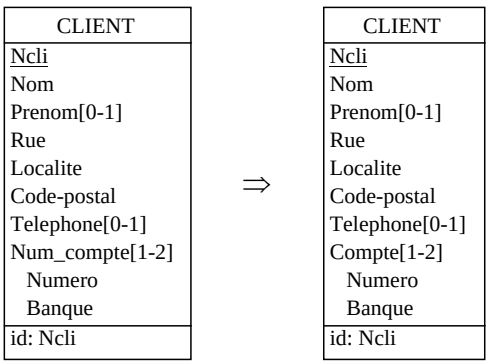
C $T^{=}$

P
- P ∈ type-entites(S) ∨ P ∈ type-associations(S) ∨ P ∈ attribut(S)
- A ∈ attribut(P)
- nom(A) = n
- ∀ A' ∈ attribut(P) : nom(A') ≠ n'

Q
- nom(A) = n'

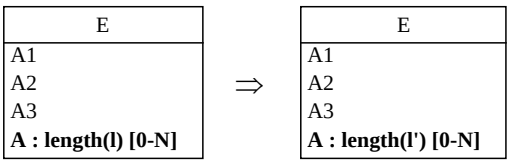
I /

E Renommer l'attribut décomposable *Num_compte* du type d'entités *CLIENT* en *Compte*.



d) Extension de domaine

D Etendre le domaine d'un attribut multivalué atomique.



S ∃ P ∈ type-entites(S) ∪ type-associations(S) ∪ attribut(S), A ∈ attribut(P) : A ← modifier-Att(P,A,l')

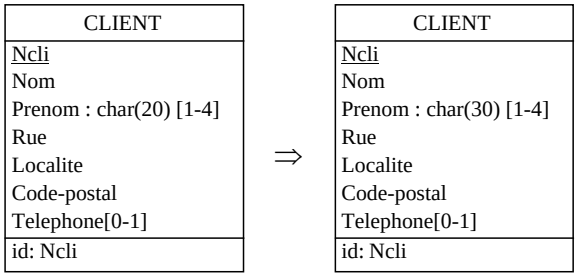
C T^{+}

P
- P ∈ type-entites(S) ∨ P ∈ type-associations(S) ∨ P ∈ attribut(S)
- A ∈ attribut(P)
- type(A) = car | num | reel | date | bool
- longueur(A) = l

Q
- longueur(A) = l'
- l < l'

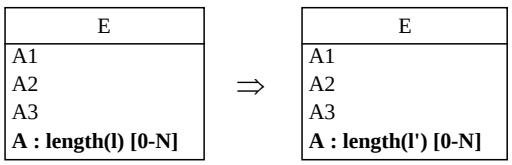
I /

E Etendre le domaine de l'attribut *Prenom* du type d'entité *CLIENT*. Passer de CHAR(20) à CHAR(30).



e) Restriction de domaine

D Restreindre le domaine d'un attribut multivalué atomique.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ modifier-Att(P,A,l')

C T^{-}

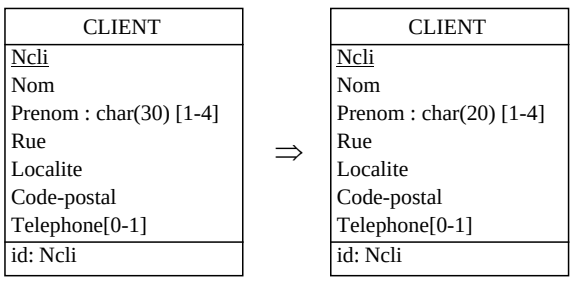
P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- type(A) = car | num | reel | date | bool
- longueur(A) = l

Q
- longueur(A) = l'
- 0 < l' < l

I $\forall$ i $\in$ instance(A), longueur(valeur(i)) $\leq$ l'

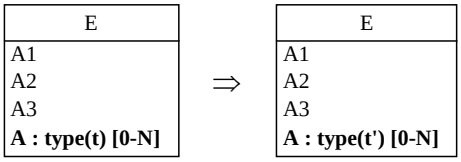
Les valeurs des instances de A doivent être comprises dans le nouveau domaine de valeurs.

E Restreindre le domaine de l'attribut *Prenom* du type d'entité *CLIENT*.



f) Changement de type

D Changer le type d'un attribut multivalué.



S ∃ P ∈ type-entites(S) ∪ type-associations(S) ∪ attribut(S), A ∈ attribut(P) : A ← modifier-Att(P,A,t')

C T^+ ou T^-

Si un type numérique est remplacé par un type caractère, le domaine de valeurs de l'attribut augmente et, inversement, si un type caractère est remplacé par un type numérique.

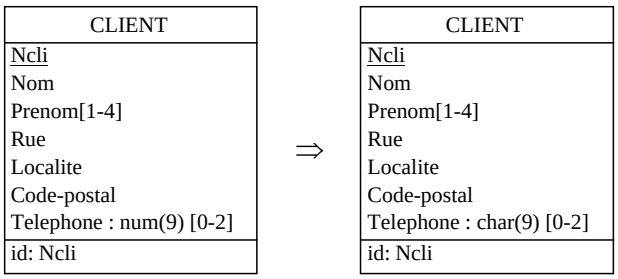
P
- P ∈ type-entites(S) ∨ P ∈ type-associations(S) ∨ P ∈ attribut(S)
- A ∈ attribut(P)
- type(A) = t
- t = car | num | reel | date | bool

Q
- type(A) = t'
- t' = car | num | reel | date | bool
- t' ≠ t

I ∀ i ∈ instance(A) : convert(valeur(i))

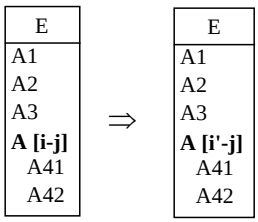
R Les instances de A doivent être converties pour vérifier le nouveau type. La conversion n'est pas toujours possible. Par exemple, la conversion d'un attribut de type caractère en type numérique est impossible si les chaînes de caractères ne représentent pas des nombres.

E Changer le type NUMERIC(9) de l'attribut multivalué *Telephone*[0-2] du type d'entités *CLIENT* en CHAR(9).



g) Augmentation de la cardinalité minimum

D Augmenter la cardinalité minimum d'un attribut multivalué et/ou décomposable.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ modifier-Att(P,A,i'-j)

C T^-

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card-min(A) = i
- i $\geq$ 0
- $\neg\exists$ g $\in$ groupe(P) : A $\in$ composant(g) $\wedge$ fonction(g) = coexistence | exclusif | au-moins-un | exactement-un

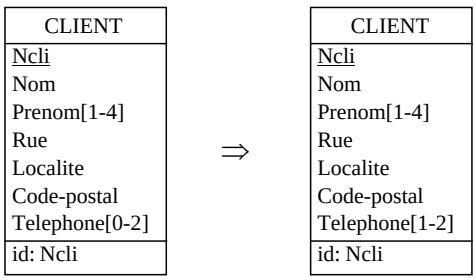
R L'attribut n'appartient pas à un groupe de coexistence, d'exclusivité, au-moins-un ou exactement-un. Si c'est le cas, il faut d'abord détruire la contrainte.

Q
- card-min(A) = i'
- j $\geq$ i' > i

I $\forall$ x $\in$ instance(A), k $\in$ [1..i'] : valeur(x[k]) $\neq$ null

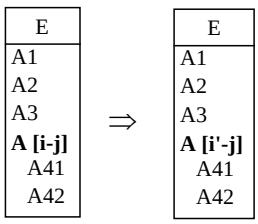
Les instances de A doivent avoir i' valeurs non nulles.

E L'attribut multivalué *Telephone* du type d'entités *CLIENT* devient obligatoire.



h) Diminution de la cardinalité minimum

D Diminuer la cardinalité minimum d'un attribut multivalué et/ou décomposable.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ modifier-Att(P,A,i'-j)

C T^+

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card-min(A) = i
- i > 0
- i' = 0 $\wedge$ ($\neg\exists$ g $\in$ groupe(P) : A $\in$ composant(g) $\wedge$ fonction(g) = id-primaire)

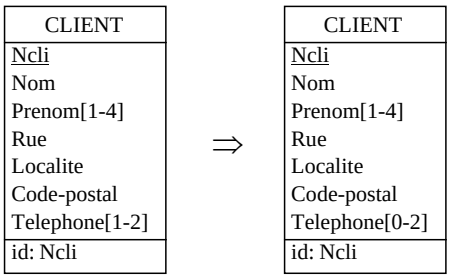
R Tous les attributs d'un identifiant primaire doivent être obligatoires. Si on veut rendre facultatif un attribut appartenant à un identifiant primaire, il faut soit changer cet identifiant en identifiant secondaire, soit enlever l'attribut des composants de l'identifiant.

Q – card-min(A) = i'

– $0 \leq i' < i$

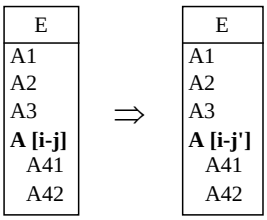
I /

E L'attribut multivalué *Telephone* du type d'entités *CLIENT* devient facultatif.



i) Augmentation de la cardinalité maximum

D Augmenter la cardinalité maximum d'un attribut multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ modifier-Att(P,A,i-j')

C T^+

P – P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)

– A $\in$ attribut(P)

– card-max(A) = j

– j < N

– $\neg\exists$ g $\in$ groupe(P) : A $\in$ composant(g) $\wedge$ fonction(g) = id-primaire | id-secondaire $\wedge$ #(composant(g)) > 1

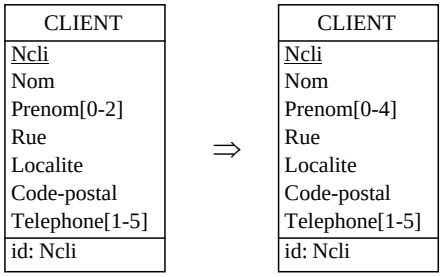
R L'attribut ne fait pas partie d'un identifiant multi-composants (contrainte du modèle).

Q – card-max(A) = j'

– j < j'

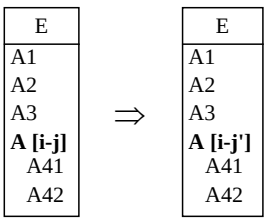
I /

E L'attribut monovalué *Prenom* du type d'entités *CLIENT* devient multivalué.



j) Diminution de la cardinalité maximum

D Diminuer la cardinalité maximum d'un attribut multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ modifier-Att(P,A,i-j')

C T^-

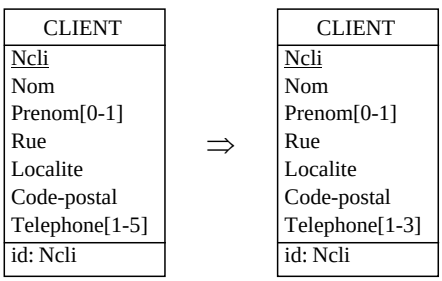
P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card-max(A) = j
- j > 1

Q
- card-max(A) = j'
- i $\leq$ j' < j

I $\forall$ x $\in$ instance(A), k $\in$ [j'+1..j] : valeur(x[k]) = null

R Les instances de A doivent avoir au maximum j' valeurs, sinon les j-j' dernières valeurs sont perdues.

E L'attribut multivalué *Telephone* du type d'entités *CLIENT* devient monovalué.



k) Changement de la nature d'un attribut

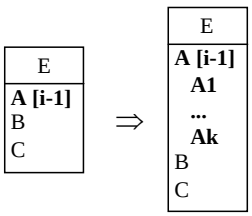
Les modifications relatives à un attribut peuvent aussi transformer sa nature. On entend par *nature* les caractères atomique, décomposable, monovalué ou multivalué d'un attribut. Ces modifications posent des problèmes au niveau d'un schéma E/A de base car elles nécessitent des transformations de conception différentes avant et après la modification. Par exemple, un attribut atomique monovalué qui ne nécessitait aucune transformation dans le schéma E/A de base doit être transformé s'il devient multivalué. Dans ce contexte, il est important d'analyser les changements de nature qui peuvent survenir. Douze situations ont été répertoriées :

- transformation d'un attribut atomique monovalué en un attribut décomposable monovalué et inversement,
- transformation d'un attribut atomique monovalué en un attribut décomposable multivalué et inversement,
- transformation d'un attribut atomique monovalué en un attribut atomique multivalué et inversement,
- transformation d'un attribut atomique multivalué en un attribut décomposable monovalué et inversement,
- transformation d'un attribut atomique multivalué en un attribut décomposable multivalué et inversement,
- transformation d'un attribut décomposable monovalué en un attribut décomposable multivalué et inversement.

Chaque cas est étudié comme les autres modifications. Cette étude est surtout très intéressante en ce qui concerne le transfert des instances. Il est clair qu'il existe, pour chaque cas, plusieurs façons d'envisager la transformation des instances. Toutefois, seule la solution qui semblait la plus logique a été retenue.

Transformation d'un attribut atomique monovalué en un attribut décomposable monovalué

D Transformer un attribut atomique monovalué en un attribut décomposable monovalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-ato-mono-en-Att-dec-mono(P,A,{A1,...,Ak})

C T^+

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- a$\in$ attribut(P)
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

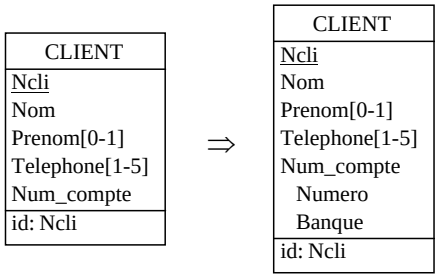
Q
- A1, ..., Ak $\in$ attribut(A)
- nom(A1) = n1 $\wedge$ ... $\wedge$ nom(Ak) = nk

I $\forall$ i $\in$ instance(A), i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak) : desagreger(i,i1,...,ik)

Pour chaque instance de P, il faut répartir l'instance de A dans les instances des sous-attributs A1, ..., Ak.

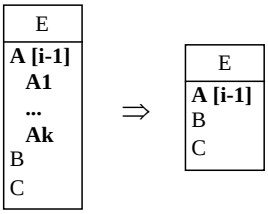
Le domaine de valeurs de A doit être compatible avec ceux des sous-attributs A1, ..., Ak. *Compatible* sous-entend qu'il est possible de transférer les instances de l'attribut original vers le nouvel attribut sans perte d'information. Dans notre cas, le type de A est le même que ceux de A1, ..., Ak et la longueur de A est inférieure ou égale à la somme des longueurs de A1, ..., Ak. Il en sera de même pour les autres modifications.

E L'attribut atomique *Num_compte* de *CLIENT* devient un attribut décomposable possédant les sous-attributs *Numero* et *Banque*.



Transformation d'un attribut décomposable monovalué en un attribut atomique monovalué

D Transformer un attribut décomposable monovalué en un attribut atomique monovalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-dec-mono-en-Att-ato-mono(A,t,l)

C T^-

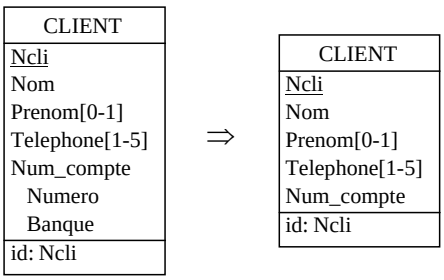
P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-1]
- A1, ..., Ak $\in$ attribut(A)

Q
- A1, ..., Ak $\notin$ attribut(A)
- type(A) = t = car | num | reel | date | bool
- longueur(A) = l

I $\forall$ i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak), i $\in$ instance(A) : agreger(i1,...,ik,i)
Pour chaque instance de P, il faut agréger les instances de A1, ..., Ak dans l'instance de A.

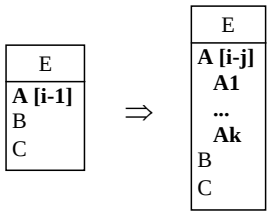
Les domaines de valeurs des sous-attributs A1, ..., Ak sont compatibles avec celui de A.

E L'attribut décomposable *Num_compte* de *CLIENT* devient un attribut atomique.



Transformation d'un attribut atomique monovalué en un attribut décomposable multivalué

D Transformer un attribut atomique monovalué en un attribut décomposable multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-ato-mono-en-Att-dec-multi(A,i-j,{A1,...,Ak})

C T^+

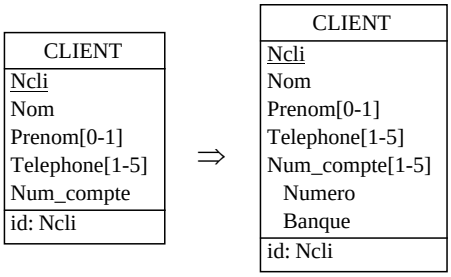
P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

Q
- A1, ..., Ak $\in$ attribut(A)
- nom(A1) = n1 $\wedge$ ... $\wedge$ nom(Ak) = nk
- card(A) = [i-j] $\wedge$ j >1

I $\forall$ i $\in$ instance(A), i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak) : desagreger(i,i1[1],...,ik[1])

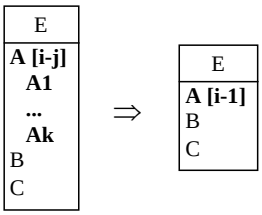
Pour chaque instance de P, il faut désagréger l'instance de A dans les premiers éléments des instances des sous-attributs A1, ..., Ak. Le domaine de valeurs de A doit être compatible avec ceux des sous-attributs A1, ..., Ak.

E L'attribut atomique *Num_compte* de *CLIENT* devient un attribut décomposable multivalué possédant les sous-attributs *Numero* et *Banque*.



Transformation d'un attribut décomposable multivalué en un attribut atomique monovalué

D Transformer un attribut décomposable multivalué en un attribut atomique monovalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-dec-multi-en-Att-ato-mono(A,t,l)

C T^-

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-j] $\wedge$ j > 1 $\wedge$ 0 $\leq$ i $\leq$ 1
- A1, ..., Ak $\in$ attribut(A)

Q
- A1,...,Ak $\notin$ attribut(A)
- type(A) = t = car | num | reel | date | bool
- longueur(A) = l
- card(A) = [i-1]

I $\forall$ i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak), i $\in$ instance(A) : agreger(i1[1],...,ik[1],i)

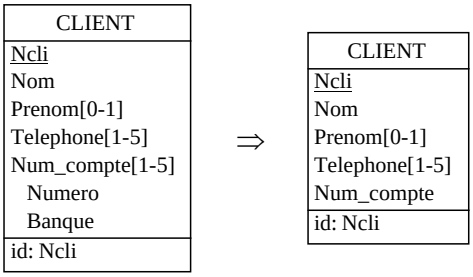
Pour chaque instance de P, il faut agréger les premières instances de A1,...,Ak dans l'instance de A.

$\forall$ i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak), 2 $\leq$ m $\leq$ j : valeur(i1[m]) = null $\wedge$ ... $\wedge$ valeur(ik[m]) = null

Pour chaque instance de P, toutes les instances de A1,...,Ak exceptés éventuellement les premières sont nulles sinon elles sont perdues lors de l'agrégation.

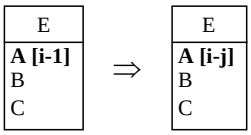
Les domaines de valeurs des sous-attributs A1, ..., Ak sont compatibles avec celui de A.

E L'attribut décomposable multivalué *Num_compte* de *CLIENT* devient un attribut atomique.



Transformation d'un attribut atomique monovalué en un attribut atomique multivalué

D Transformer un attribut atomique monovalué en un attribut atomique multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-ato-mono-en-Att-ato-multi(A,i-j)

C T^+

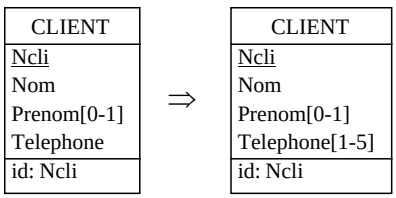
P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

Q
- card(A) = [i-j] $\wedge$ j >1

I $\forall$ i $\in$ instance(A) : valeur(i[1]) = valeur(i)

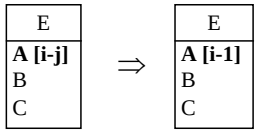
Chaque instance de A est insérée dans le premier élément de l'instance du nouveau A.

E L'attribut atomique monovalué *Telephone* de *CLIENT* devient un attribut atomique multivalué.



Transformation d'un attribut atomique multivalué en un attribut atomique monovalué

D Transformer un attribut atomique multivalué en attribut atomique monovalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-ato-multi-en-Att-ato-mono(A,i-1)

C T^-

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-j] $\wedge$ j > 1 $\wedge$ 0 $\leq$ i $\leq$ 1
- type(A) = car | num | reel | date | bool

Q
- card(A) = [i-1]

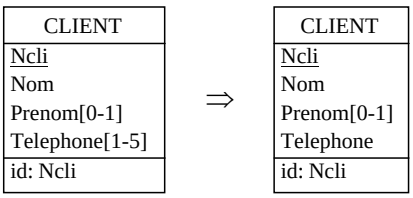
I $\forall$ i $\in$ instance(A) : valeur(i) = valeur(i[1])

Pour chaque instance de P, la première instance de A est insérée dans la nouvelle instance de A.

$\forall$ i $\in$ instance(A), 2 $\leq$ m $\leq$ j : valeur(i[m]) = null

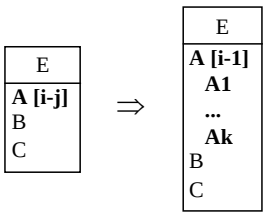
Tous les éléments des instances de A, excepté éventuellement le premier, sont nuls sinon leurs valeurs sont perdues lors du transfert.

E L'attribut atomique multivalué *Telephone* de *CLIENT* devient un attribut atomique monovalué.



Transformation d'un attribut atomique multivalué en un attribut décomposable monovalué

D Transformer un attribut atomique multivalué en attribut décomposable monovalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-ato-multi-en-Att-dec-mono(A,i-1,{A1,...,Ak})

C T^+ ou T^- : indéterminé c'est en fonction du domaine d'application.

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)

– card(A) = [i-j] ∧ j > 1 ∧ 0 ≤ i ≤ 1
– type(A) = car | num | reel | date | bool

Q – A1, ..., Ak ∈ attribut(A)
– nom(A1) = n1 ∧ ... ∧ nom(Ak) = nk
– card(A) = [i-1]

I ∀ i ∈ instance(A), i1 ∈ instance(A1), ..., ik ∈ instance(Ak) : desagreger(i[1],i1,...,ik)

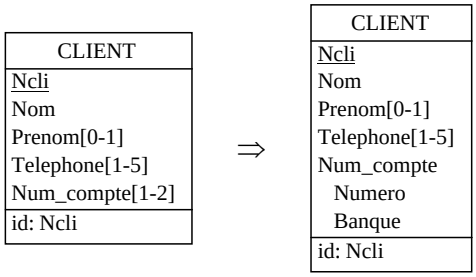
Il faut désagréger le premier élément de chaque instance de A dans les instances de A1,...,Ak.

∀ i ∈ instance(A), m ∈ [2..j] : valeur(i[m]) = null

Tous les éléments de chaque instance de A, excepté le premier, sont nuls sinon ils sont perdus lors de l'agrégation des valeurs.

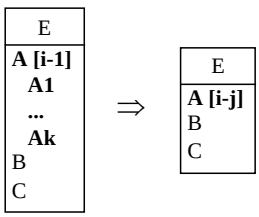
Les domaines de valeurs des sous-attributs A1, ..., Ak sont compatibles avec celui de A.

E L'attribut atomique multivalué *Num_compte* de *CLIENT* devient un attribut décomposable monovalué possédant les sous-attributs *Numero* et *Banque*.



Transformation d'un attribut décomposable monovalué en un attribut atomique multivalué

D Transformer un attribut décomposable monovalué en un attribut atomique multivalué.



S ∃ P ∈ type-entites(S) ∪ type-associations(S) ∪ attribut(S), A ∈ attribut(P) : A ← Att-dec-mono-en-Att-ato-multi(A,i-j,t,l)

C T^- ou T^+

P – P ∈ type-entites(S) ∨ P ∈ type-associations(S) ∨ P ∈ attribut(S)
– A ∈ attribut(P)
– card(A) = [i-1]
– A1, ..., Ak ∈ attribut(A)

Q – A1,...,Ak ∉ attribut(A)
– type(A) = t = car | num | reel | date | bool

– longueur(A) = l

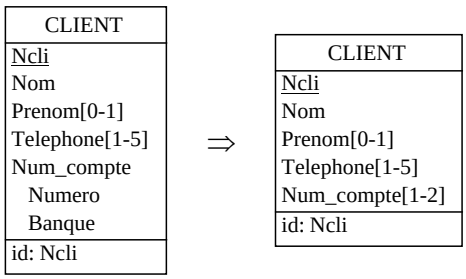
– card(A) = [i-j] ∧ j > 1

I ∀ i1 ∈ instance(A1), ..., ik ∈ instance(Ak), i ∈ instance(A) : agreger(i1,...,ik,i[1])

Il faut agréger les instances de A1,...,Ak dans le premier élément des instances de A.

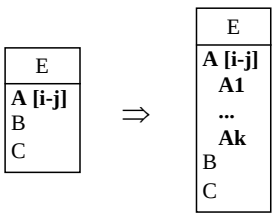
Le domaine de valeurs de A est compatible avec ceux des sous-attributs A1, ..., Ak.

E L'attribut décomposable monovalué *Num_compte* de *CLIENT* devient un attribut atomique multivalué.



Transformation d'un attribut atomique multivalué en un attribut décomposable multivalué

D Transformer un attribut atomique monovalué en un attribut décomposable multivalué.



S ∃ P ∈ type-entites(S) ∪ type-associations(S) ∪ attribut(S), A ∈ attribut(P) : A ← Att-ato-multi-en-Att-dec-multi(A,{A1,...,Ak})

C T^+

P – P ∈ type-entites(S) ∨ P ∈ type-associations(S) ∨ P ∈ attribut(S)

– A ∈ attribut(P)

– type(A) = car | num | reel | date | bool

– card(A) = [i-j] ∧ j > 1 ∧ 0 ≤ i ≤ 1

Q – A1, ..., Ak ∈ attribut(A)

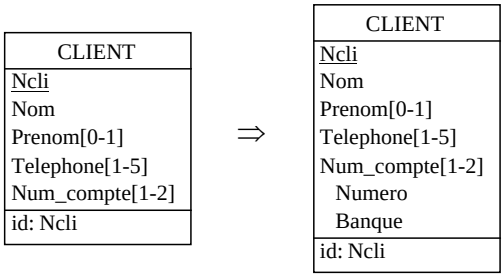
– nom(A1) = n1 ∧ ... ∧ nom(Ak) = nk

I ∀ i ∈ instance(A), i1 ∈ instance(A1), ..., ik ∈ instance(Ak), m ∈ [1..j] : desagreger(i[m],i1[m],...,ik[m])

Il faut désagréger les instances de A dans les instances des sous-attributs A1,...,Ak.

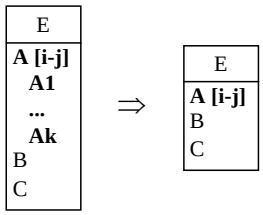
Le domaine de valeurs de A doit être compatible avec ceux des sous-attributs A1, ..., Ak.

E L'attribut atomique multivalué *Num_compte* de *CLIENT* devient un attribut décomposable multivalué possédant les sous-attributs *Numero* et *Banque*.



Transformation d'un attribut décomposable multivalué en un attribut atomique multivalué

D Transformer un attribut décomposable multivalué en attribut atomique multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-dec-multi-en-Att-ato-multi(A,t,l)

C T^-

P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-j] $\wedge$ j > 1 $\wedge$ 0 $\leq$ i $\leq$ 1
- A1, ..., Ak $\in$ attribut(A)

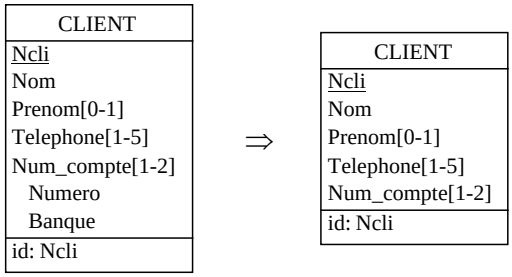
Q
- A1,...,Ak $\notin$ attribut(A)
- type(A) = t = car | num | reel | date | bool
- longueur(A) = l

I $\forall$ i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak), i $\in$ instance(A), m $\in$ [1..j] : agreger(i1[m],...,ik[m],i[m])

Il faut agréger chaque instance de A1, ..., Ak dans les instances de A.

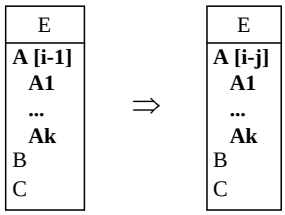
Les domaines de valeurs des sous-attributs A1, ..., Ak sont compatibles avec celui de A.

E L'attribut décomposable multivalué *Num_compte* de *CLIENT* devient un attribut atomique multivalué.



Transformation d'un attribut décomposable monovalué en un attribut décomposable multivalué

D Transformer un attribut décomposable monovalué en attribut décomposable multivalué.



S $\exists$ P $\in$ type-entites(S) $\cup$ type-associations(S) $\cup$ attribut(S), A $\in$ attribut(P) : A $\leftarrow$ Att-dec-mono-en-Att-dec-multi(A,i-j)

C T^+

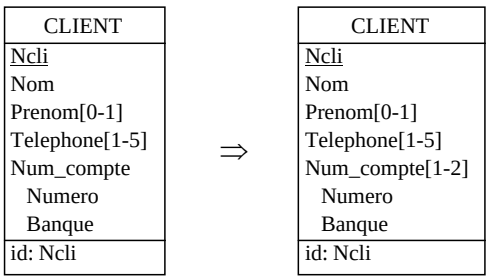
P
- P $\in$ type-entites(S) $\vee$ P $\in$ type-associations(S) $\vee$ P $\in$ attribut(S)
- A $\in$ attribut(P)
- card(A) = [i-1]
- A1, ..., Ak $\in$ attribut(A)

Q
- card(A) = [i-j] $\wedge$ j >1

I $\forall$ i1 $\in$ instance(A1), ..., ik $\in$ instance(Ak) : valeur(i1[1]) = valeur(i1) $\wedge$ ... $\wedge$ valeur(ik[1]) = valeur(ik)

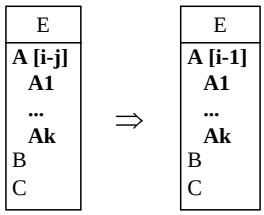
Les instances de A1, ..., Ak sont mises dans les premiers éléments des nouvelles instances de A1, ..., Ak.

E L'attribut décomposable monovalué *Num_compte* de *CLIENT* devient un attribut décomposable multivalué.



Transformation d'un attribut décomposable multivalué en attribut décomposable monovalué

D Transformer un attribut décomposable multivalué en un attribut décomposable monovalué.



S ∃ P ∈ type-entites(S) ∪ type-associations(S) ∪ attribut(S), A ∈ attribut(P) : A ← Att-dec-multi-en-Att-dec-mono(A,i-1)

C T-

P
- P ∈ type-entites(S) ∨ P ∈ type-associations(S) ∨ P ∈ attribut(S)
- A ∈ attribut(P)
- card(A) = [i-j] ∧ j > 1 ∧ 0 ≤ i ≤ 1
- A1, ..., Ak ∈ attribut(A)

Q
- card(A) = [i-1]

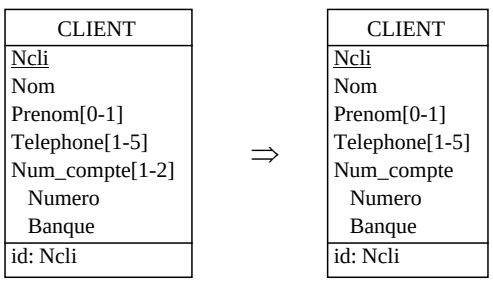
I ∀ i1 ∈ instance(A1), ..., ik ∈ instance(Ak) : valeur(i1) = valeur(i1[1]) ∧ ... ∧ valeur(ik) = valeur(ik[1])

Les premiers éléments des instances de A1, ..., Ak sont mis dans les instances de A1,...,Ak.

∀ i1 ∈ instance(i1), ..., ik ∈ instance(ik), m ∈ [2..j] : valeur(i1[m]) = null,...,valeur(ik[m]) = null

Tous les éléments des instances de A1,...,Ak, exceptés les premiers, sont nuls sinon les valeurs sont perdues lors du transfert dans l'attribut monovalué.

E L'attribut décomposable multivalué *Num_compte* de *CLIENT* devient un attribut décomposable monovalué.

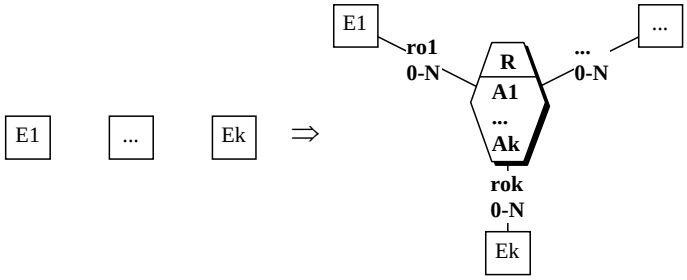


C.4.2.2 Modifications relatives aux types d'associations complexes

Ce point étudie les modifications relatives aux types d'associations non fonctionnelles ou complexes (jouant plus que deux rôles et/ou possédant des attributs). Ces structures conceptuelles ne sont pas exprimables en tant que telles dans un schéma logique relationnel. Il faudra les transformer en un type d'entités pour obtenir des structures conformes au modèle relationnel.

a) Création

D Créer un type d'associations complexe.



S $R \leftarrow$ creer-TA(S,n)

C T^+

P – $\forall$ R' $\in$ type-associations(S) : nom(R') $\neq$ n

Q – R $\in$ type-associations(S)

– nom(R) = n

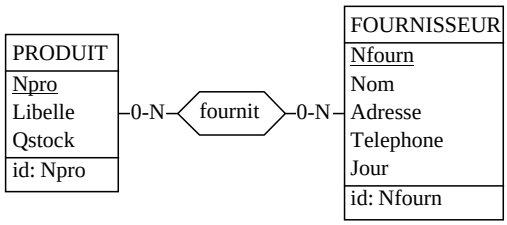
– ro1, ...,rok $\in$ role(R)

– A1,...,Ak $\in$ attribut(R)

I $\forall$ m $\in$ [1..k] : (($\exists$ rom $\in$ role(R) : card-min(rom) > 0) $\wedge$ ($\forall$ em $\in$ instance(Em), $\exists$ r $\in$ instance(R) : r[rom] = em) $\vee$ instance(R) = $\varnothing$

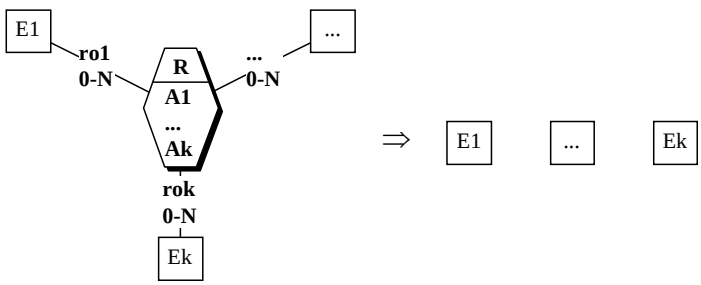
Si un des rôles de R a une cardinalité minimum supérieure à 0, il faut des instances de R pour que la contrainte de cardinalité soit vérifiée.

E On crée un type d'associations *fournit* (0-N/0-N) entre *FOURNISSEUR* et *PRODUIT*.



b) Destruction

D Supprimer un type d'associations complexe.



S $\exists$ R $\in$ type-associations(S) : R $\leftarrow$ supprimer-TA(R)

C T^-

P – R $\in$ type-associations(S)

Q – R $\notin$ type-associations(S)

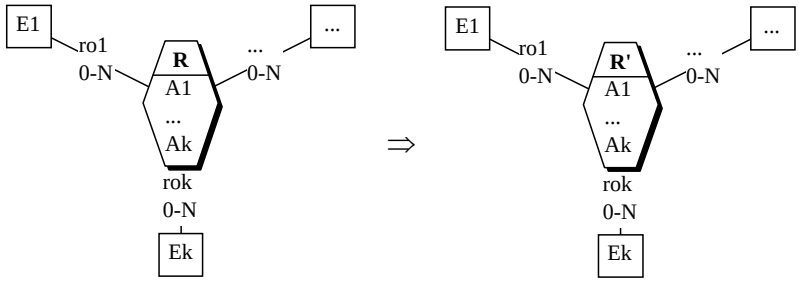
– ro1,...,rok ∉ role(R)
– A1,...,Ak ∉ attribut(R)

I ∀ i ∈ instance(R) : supprimer(i)

E On supprime le type d'associations *fournit* entre *PRODUIT* et *FOURNISSEUR*.

c) Renommage

D Renommer un type d'associations complexe.



S ∃ R ∈ type-associations(S) : R ← modifier-TA(R,n')

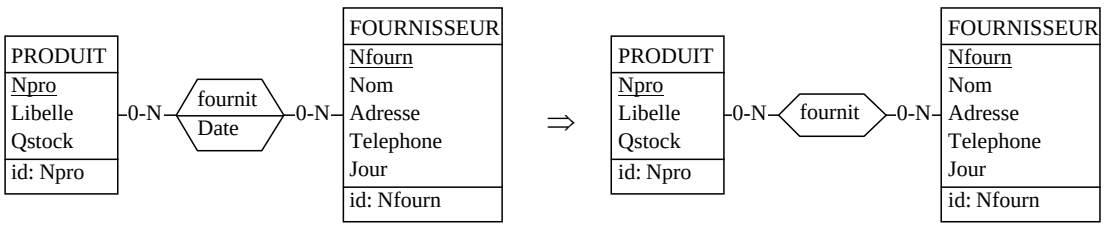
C T=

P – R ∈ type-associations(S)
– nom(R) = n
– ∀ R' ∈ type-associations(S) : nom(R') ≠ n'

Q – nom(R) = n'

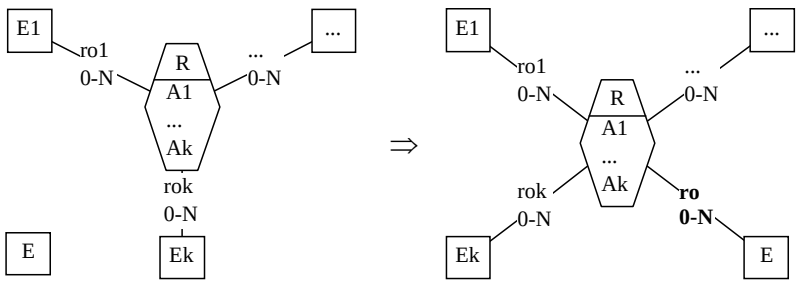
I /

E On renomme le type d'associations *fournit* entre *PRODUIT* et *FOURNISSEUR* en *fourniture*.



d) Création d'un rôle

D Ajouter un rôle à un type d'associations complexe.



S ∃ R ∈ type-associations(S), E ∈ type-entites(S) : ro ← creer-Role(R,nro,{E},[i-j])

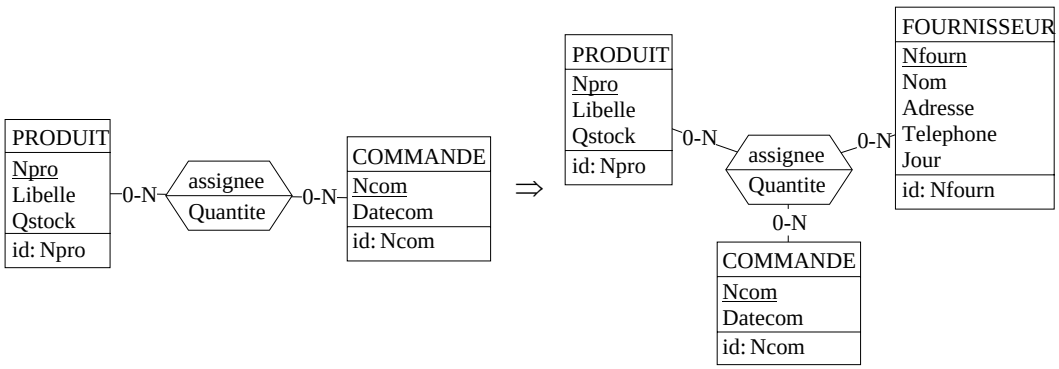
C T^+

P – R ∈ type-associations(S)
– E ∈ type-entites(S)
– ro1,...,rok ∈ role(R)
– A1,...,Am ∈ attribut(R)
– ∀ ro' ∈ type-associations(S) : nom(ro') ≠ n

Q – ro ∈ role(R)
– te-role(ro) = {E}
– nom(ro) = n

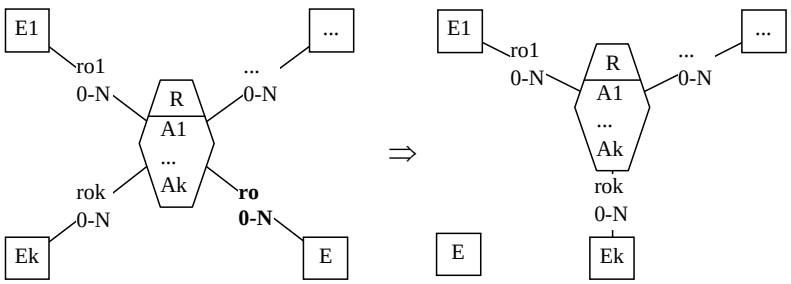
I ∀ r ∈ instance(R), ∃! e ∈ instance(E) : r[ro] = e
Il faut ajouter à chaque instance de R des instances de ro.

E On ajoute un rôle entre le type d'associations *assignee* et le type d'entités *FOURNISSEUR*.



e) Destruction d'un rôle

D Détruire un rôle d'un type d'associations complexe.



S ∃ R ∈ type-associations(S), ro ∈ role(R) : () ← supprimer(R,ro)

C T^-

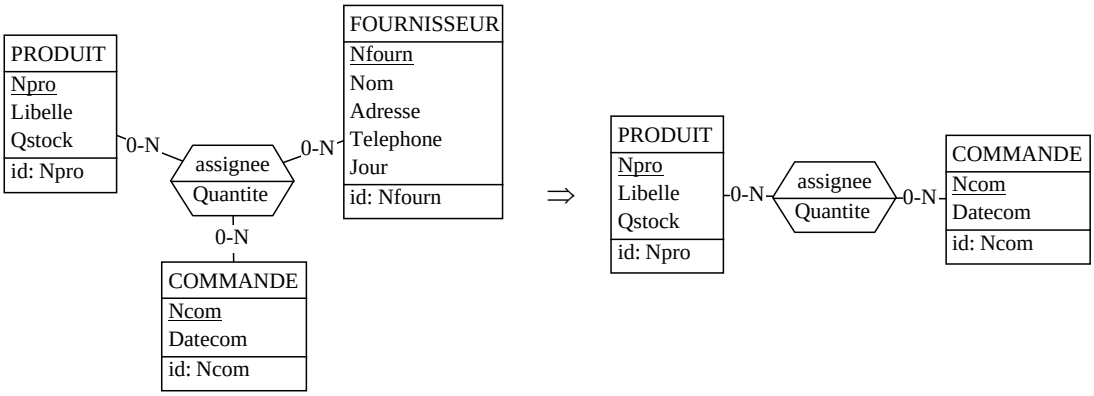
P – R ∈ type-associations(S)
– ro1,...,rok,ro ∈ role(R)
– A1,...,Am ∈ attribut(R)

Q – ro ∉ role(R)

I ∀ r ∈ instance(R) : r[ro] = null

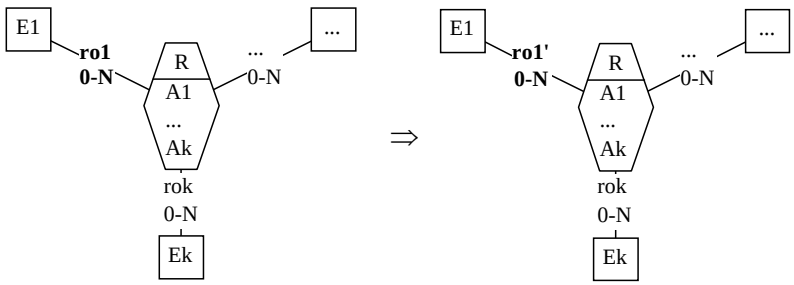
Il faut supprimer de chaque instance de R les instances de ro.

E On détruit le rôle entre le type d'associations *assignee* et le type d'entités *FOURNISSEUR*.



f) Renommage d'un rôle

D Renommer un rôle d'un type d'associations complexe.



S ∃ R ∈ type-associations(S), ro1 ∈ role(R) : ro1 ← modifier-Role(R,ro1,n')

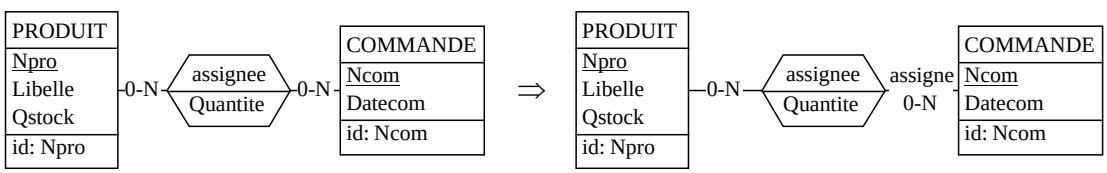
C $T^{=}$

P
- R ∈ type-associations(S)
- ro1,...,rok ∈ role(R)
- A1,...,Am ∈ attribut(R)
- nom(ro1) = n
- ∀ ro' ∈ role(R) : nom(ro') ≠ n'

Q
- nom(ro1) = n'

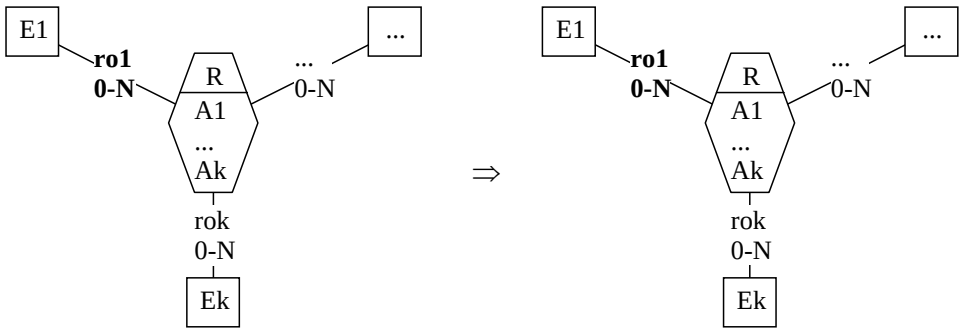
I /

E Le rôle *assignee.COMMANDE* entre le type d'associations *assignee* et le type d'entités *COMMANDE* est renommé *assigne*.



g) Modification des cardinalités d'un rôle

D Modifier les cardinalités d'un rôle d'un type d'associations complexe.



S ∃ R ∈ type-associations(S), ro1 ∈ role(R) : ro1 ← modifier-Role(R,ro1,i'-j')

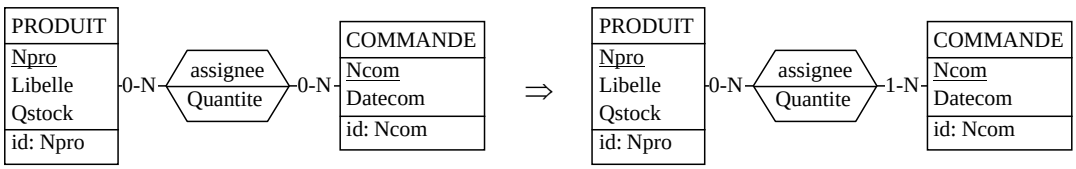
C Augmentation de la cardinalité minimum et diminution de la cardinalité maximum : T^-
Diminution de la cardinalité minimum et augmentation de la cardinalité maximum : T^+

P
- R ∈ type-associations(S)
- ro1,...,rok ∈ role(R)
- A1,...,Am ∈ attribut(R)
- card(ro1) = [i-j]

Q
- card(ro1) = [i'-j']
- 0 ≤ i' ≤ j'
- i' ≤ j' ≤ N
- i' ≠ i ∨ j' ≠ j

I ∀ e1 ∈ instance(E1), ∃ rik ∈ instance(R), k ∈ [i'..j'] : rik[ro1] = e1
Chaque instance de E1 doit participer à au minimum i' et au maximum j' instances de R.

E Les cardinalités du rôle joué par *COMMANDE* dans *assignee* deviennent 1-N.



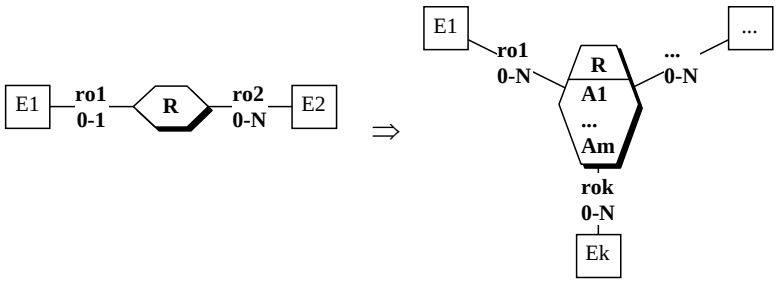
h) Changement de la nature d'un type d'associations

Les modifications relatives à un type d'associations peuvent également changer sa nature. On sous-entend par nature le fait qu'un type d'associations peut être soit complexe soit fonctionnel. Au niveau d'un schéma E/A de base, un type d'associations complexe doit être transformé. Par exemple, un type d'association fonctionnel qui ne nécessitait pas de transformation dans le schéma E/A de base doit être transformé en un type d'entités si on lui ajoute un attribut.

Ce point analyse la transformation d'un type d'associations fonctionnel en un type d'associations complexe et son inverse.

Transformation d'un type d'associations fonctionnel en un type d'associations complexe

D Modifier un type d'associations fonctionnel (binaire, un-à-un ou un-à-plusieurs, sans attribut) pour en faire un type d'associations complexe grâce à l'ajout d'attributs, de rôles ou la modification des cardinalités des rôles.



S ∃ R ∈ type-associations(S) : ro3 ← creer-Role(R,nro3,{E3},i3-j3) ∨ ... ∨ rok ← creer-Role(R,nrok,{Ek},ik-jk) ∨ A1 ← creer-Att(R,na1) ∨ ... ∨ Am ← creer-Att(R,nam) ∨ ro1 ← modifier-Role(R,ro1,i'-j')

C T^+

P
- R ∈ type-associations(S)
- ro1,ro2 ∈ role(R)
- (card(ro1) = [0-1] ∧ card(ro2) = [0-1]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [1-1]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [0-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [0-N]) ∨ (card(ro1) = [0-1] ∧ card(ro2) = [1-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [1-N]) ∨ (card(ro1) = [1-1] ∧ card(ro2) = [1-1])

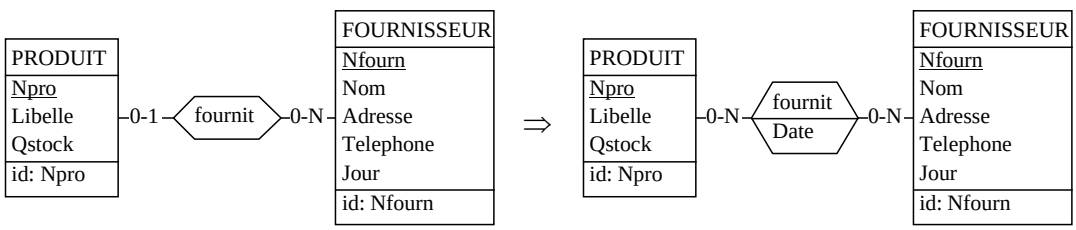
Q
- ro1,...,rok ∈ role(R)
- A1,...,Am ∈ attribut(R)
- nom(ro1) = n1,..., nom(rok) = nk
- nom(A1) = s1,..., nom(Am) = sl
- card-max(ro1) = N

I ∀ r ∈ instance(R), m ∈ [3..k], ∃ em ∈ instance(Em) : r[rom] = em

∀ r ∈ instance(R), m ∈ [1..k] : (card-max(Am) > 0 ∧ r[Am] = valeur-def) ∨ r[Am] = null

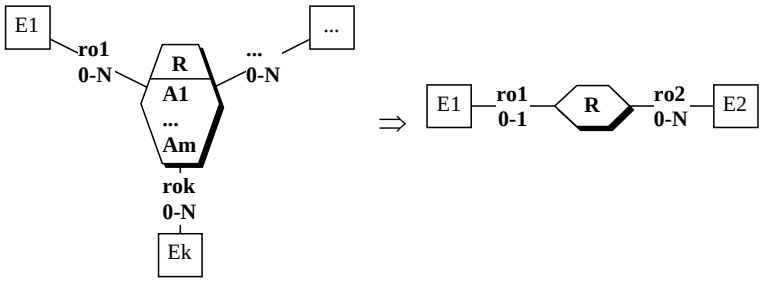
Il faut créer les instances pour les nouveaux rôles et attributs dans les instances existantes de R.

E Le type d'associations *fournit* reçoit un attribut *Date* et les cardinalités du rôle joué par *PRODUIT* dans *fournit* deviennent 0-N.



Transformation d'un type d'associations complexe en un type d'associations fonctionnel

D Modifier un type d'associations complexe pour en faire un type d'associations fonctionnel (binaire, un-à-plusieurs ou un-à-un, sans attribut).



S $\exists$ R $\in$ type-associations(S), ro1,ro3,...,rok $\in$ role(R), A1,...,Am $\in$ attribut(R) : () $\leftarrow$ supprimer-Role(R,ro3) $\vee$ ... $\vee$ () $\leftarrow$ supprimer-Role(R,rok) $\vee$ () $\leftarrow$ supprimer-Att(R,A1) $\vee$ ... $\vee$ () $\leftarrow$ supprimer-Att(R,Am) $\vee$ ro1 $\leftarrow$ modifier-Role(R,ro1,i'-j')

C T^-

P
- R $\in$ type-associations(S)
- ro1,...,rok $\in$ role(R)
- A1,...,Am $\in$ attribut(R)

Q
- ro1,ro2 $\in$ role(R)
- ro3,...,rok $\notin$ role(R)
- A1,...,Am $\notin$ attribut(R)
- (card(ro1) = [0-1] $\wedge$ card(ro2) = [0-1]) $\vee$ (card(ro1) = [0-1] $\wedge$ card(ro2) = [1-1]) $\vee$ (card(ro1) = [0-1] $\wedge$ card(ro2) = [0-N]) $\vee$ (card(ro1) = [1-1] $\wedge$ card(ro2) = [0-N]) $\vee$ (card(ro1) = [0-1] $\wedge$ card(ro2) = [1-N]) $\vee$ (card(ro1) = [1-1] $\wedge$ card(ro2) = [1-N]) $\vee$ (card(ro1) = [1-1] $\wedge$ card(ro2) = [1-1])
- card-max(ro1) = 1

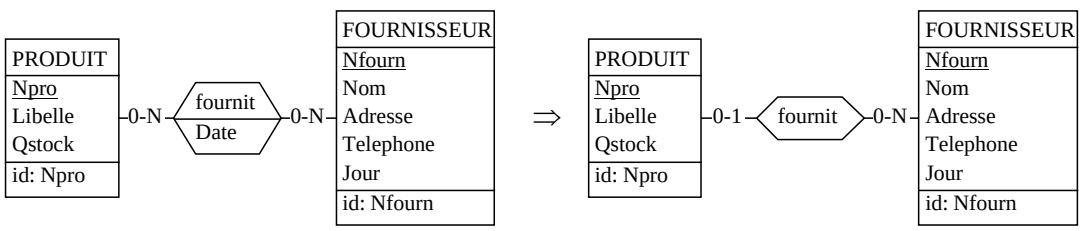
I $\forall$ m $\in$ [1..k] : instance(rom) = $\varnothing$

$\forall$ k $\in$ [1..m] : instance(Ak) = $\varnothing$

$\forall$ r $\in$ instance(R), $\exists!$ e1 $\in$ instance(E1) : r(E1) = e1

Les instances des rôles et des attributs supprimés sont perdues, seules les instances des deux rôles restant sont conservés. En plus, une instance de E1 ne peut jouer qu'un seul rôle dans une R.

E Le type d'associations *fournit* perd son attribut *Date* et le rôle joué par *PRODUIT* dans *fournit* est 0-1.

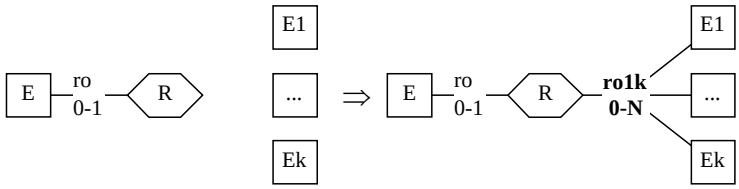


C.4.2.3 Modifications relatives aux rôles multi-TE

Ce point analyse les modifications relatives aux rôles multi-TE. Il s'agit de structures conceptuelles qui ne sont pas exprimables en tant que telles dans un schéma logique relationnel. Il faut transformer ces structures pour obtenir des structures conformes au modèle relationnel. C'est la transformation d'un rôle multi-TE en types d'associations qui est utilisée.

a) Création

D Créer un rôle multi-TE.



S ∃ R ∈ type-associations(S) : ro1k ← creer-Role(R,nro1k,{E1,...,Ek},0-N)

C T^+

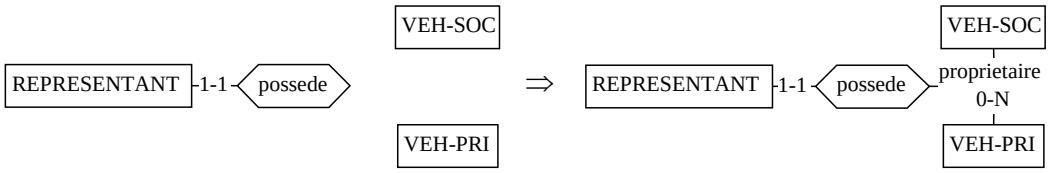
P
- R ∈ type-associations(S)
- E1,...Ek ∈ type-entites(S)
- ∀ ro' ∈ role(R) : nom(ro') ≠ n

Q
- ro1k ∈ role(R)
- te-role(ro) = {E1,...,Ek}
- nom(ro1k) = n

I ∀ r ∈ instance(R), ∃! e ∈ instance(E1) ∪ ... ∪ instance(Ek) : r[ro1k] = e

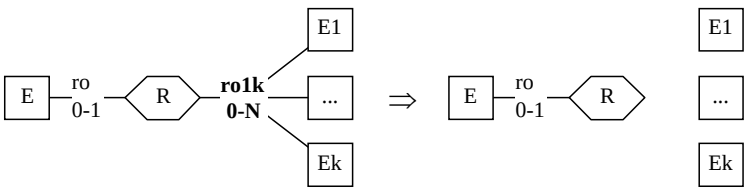
Il faut ajouter à chaque instance de R une instance E1, ..., Ek-1 ou Ek.

E Créer un rôle multi-TE entre les types d'entités *VEH-SOC* et *VEH-PRI* et le type d'associations *possede*.



b) Destruction

D Supprimer un rôle multi-TE.



S ∃ R ∈ type-associations(S), ∃ ro1k ∈ role(R) : () ← supprimer-Role(R,ro1k)

C T^-

P – $R \in$ type-associations(S)

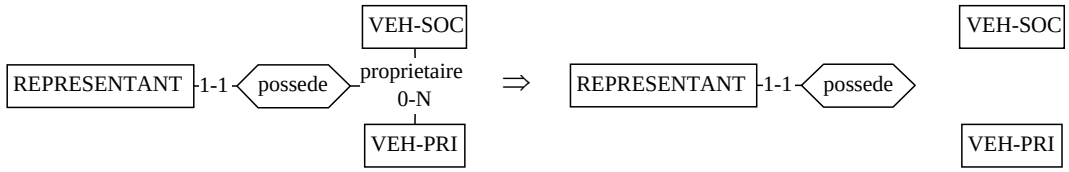
– ro,ro1k $\in$ role(R)

Q – ro1k $\notin$ role(R)

I $\forall$ r $\in$ instance(R) : r[ro1k] = null

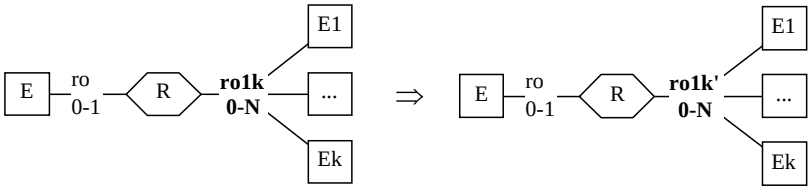
Il faut enlever à chaque instance de R une instance de E1, ... ou Ek.

E Détruire le rôle multi-TE *proprietaire* entre les types d'entités *VEH-SOC* et *VEH-PRI* et le type d'associations *possede*.



c) Renommage

D Renommer un rôle multi-TE.



S $\exists$ R $\in$ type-associations(S), $\exists$ ro1k $\in$ role(R) : ro1k $\leftarrow$ modifier-Role(R,ro1k,n')

C T^-

P – $R \in$ type-associations(S)

– ro1k $\in$ role(R)

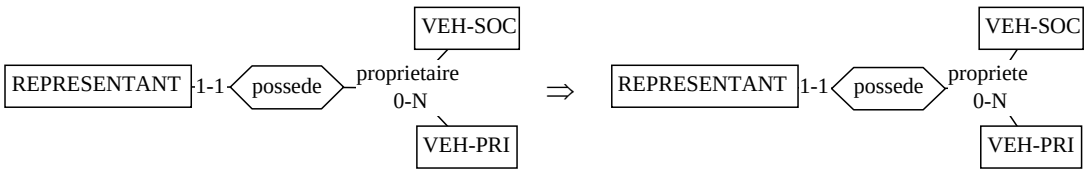
– nom(ro1k) = n

– $\forall$ ro' $\in$ role(R) : nom(ro') $\neq$ n'

Q – nom(ro1k) = n'

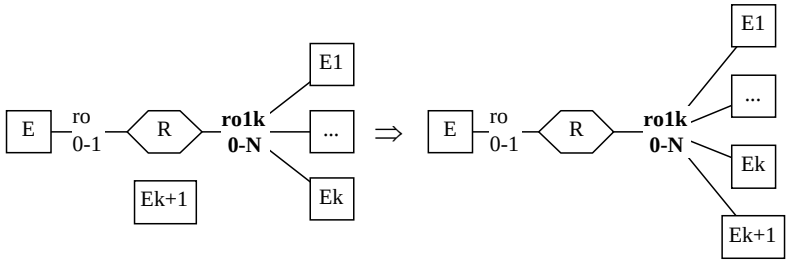
I /

E Renommer le rôle multi-TE *proprietaire* en *propriete*.



d) Ajout d'un type d'entités

D Ajouter un type d'entités à un rôle multi-TE.



S ∃ R ∈ type-associations(S), ∃ ro1k ∈ role(R) : (ro1k,{E1,...,Ek+1}) ← modifier-Role(R,ro1k,{Ek+1})

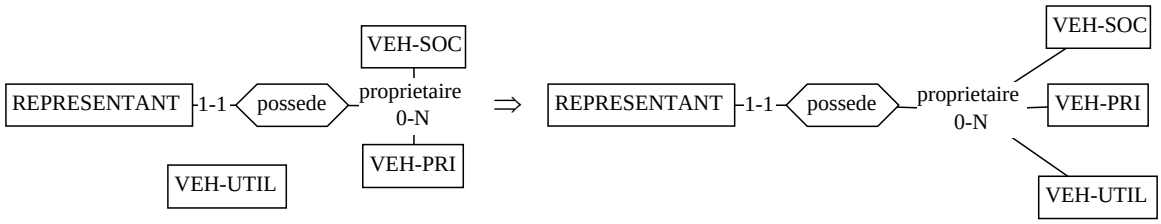
C T^+

P – R ∈ type-associations(S)
– ro, ro1k ∈ role(R)
– {E1,...,Ek} = te-role(ro1k)

Q – {E1,...,Ek,Ek+1} = te-role(ro1k)

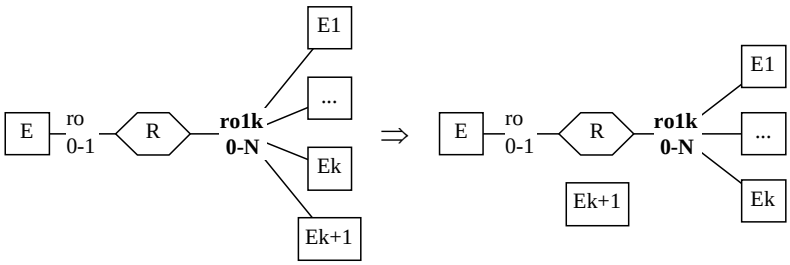
I /

E Le type d'entités *VEH-UTIL* participe au rôle multi-TE *proprietaire*.



e) Retrait d'un type d'entités

D Enlever un type d'entités d'un rôle multi-TE.



S ∃ R ∈ type-associations(S), ∃ ro1k ∈ role(R) : (ro1k,{E1,...,Ek}) ← modifier-Role(R,ro1k,{Ek+1})

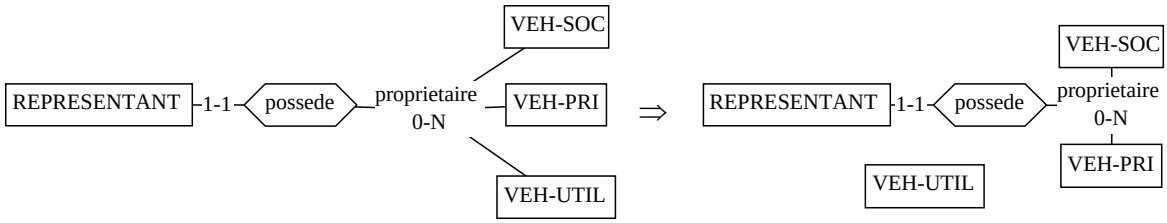
C T^-

P – R ∈ type-associations(S)
– ro, ro1k ∈ role(R)
– {E1,...,Ek,Ek+1} = te-role(ro1k) avec k > 1

Q – {E1,...,Ek} = te-role(ro1k)

I $\forall$ r $\in$ instance(R), $\neg\exists$ e $\in$ instance(Ek+1) : r[ro1k] = e
Aucune instance de Ek+1 ne peut participer à des instances de R.

E Le type d'entités *VEH-UTIL* ne participe plus au rôle multi-TE *proprietaire*.

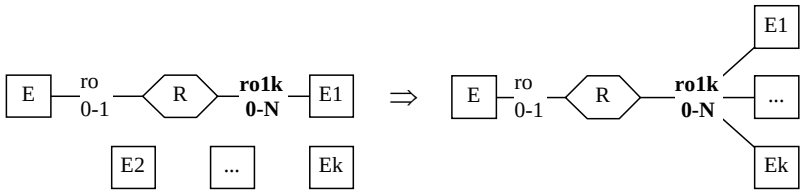


f) Changement de la nature d'un rôle

Les modifications relatives à un rôle peuvent également changer sa nature. On sous-entend par nature le fait qu'un rôle soit mono-TE ou multi-TE. Une modification portant sur ces caractéristiques pose des problèmes au niveau d'un schéma E/A de base. Par exemple, un rôle mono-TE qui ne nécessitait pas de transformation dans le schéma E/A de base doit être transformé en plusieurs types d'associations s'il devient multi-TE. Les deux cas possibles sont la transformation d'un rôle mono-TE en un rôle multi-TE et inversement.

Transformation d'un rôle mono-TE en un rôle multi-TE

D Ajouter un ou plusieurs types d'entités à un rôle mono-TE pour en faire un rôle multi-TE.



S $\exists$ R $\in$ type-associations(S), $\exists$ ro1k $\in$ role(R) : (ro1k,{E1,...,Ek}) $\leftarrow$ modifierRole(R,ro1k,{E2,...,Ek})

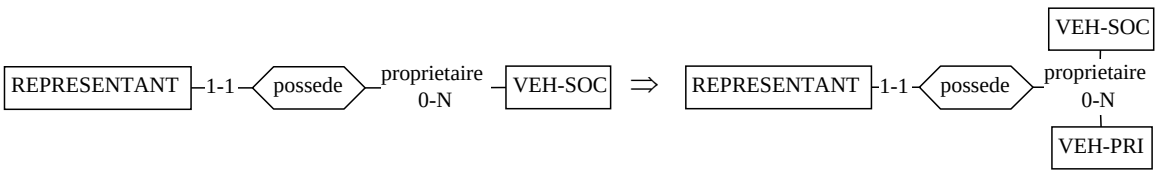
C T^+

P
- R $\in$ type-associations(S)
- E1,...,Ek $\in$ type-entites(S)
- ro, ro1k $\in$ role(R)
- {E1} = te-role(ro1k)

Q
- {E1,...,Ek} = te-role(ro1k)

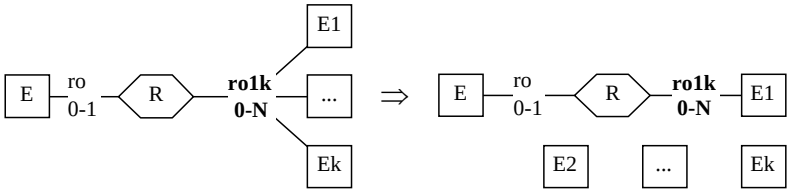
I /

E Le rôle *proprietaire* devient multi-TE.



Transformation d'un rôle multi-TE en un rôle mono-TE

D Enlever un ou plusieurs types d'entités d'un rôle multi-TE pour en faire un rôle mono-TE.



S ∃ R ∈ type-associations(S), ∃ ro1k ∈ role(R) : (ro1k,{E1}) ← modifier-Role(R,ro1k,{E2,...,Ek})

C T^-

P – R ∈ type-associations(S)

– ro, ro1k ∈ role(R)

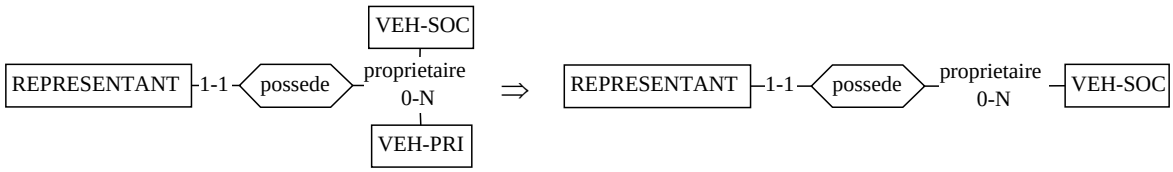
– {E1,...,Ek} = te-role(ro1k)

Q – {E1} = te-role(ro1k)

I ∀ r ∈ instance(R), ¬∃ e ∈ instance(E2) ∪ ... ∪ instance(Ek) : r[ro1k] = e

Aucune instance de E1, ..., Ek ne peut participer à des instances de R.

E Le rôle *proprietaire* devient mono-TE.

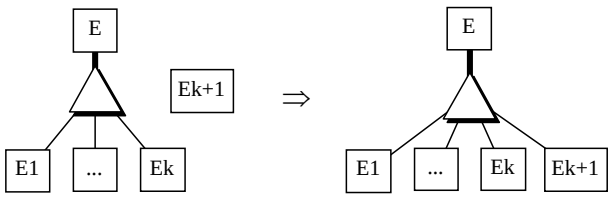


C.4.2.4 Modifications relatives aux relations IS-A

Ce point étudie les modifications relatives aux relations IS-A. Ces relations d'héritage ne sont pas exprimables en tant que telles dans un schéma relationnel. Elles doivent être transformées en types d'associations pour obtenir des structures conformes au modèle relationnel.

a) Création

D Créer une relation d'héritage (IS-A).



S ∃ E,E1,...,Ek,Ek+1 ∈ type-entité(S) : (E,{E1,...,Ek,Ek+1}) ← creer-Isa(E,{Ek+1})

C T^+

P – E,E1,...,Ek,Ek+1 ∈ type-entites(S)

– {E1,...,Ek} = sous-types(E)

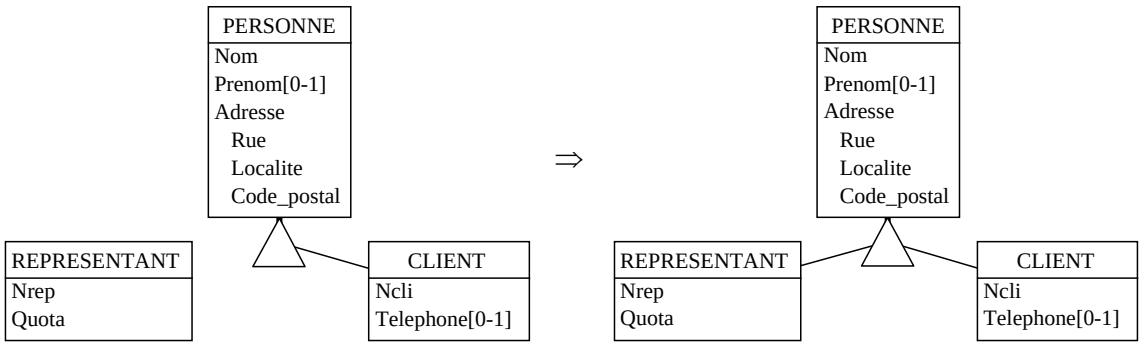
– E ∉ sous-types(Ek+1)

R Il faut empêcher les circuits au niveau de l'héritage.

Q – {E1,...,Ek,Ek+1} = sous-types(E)

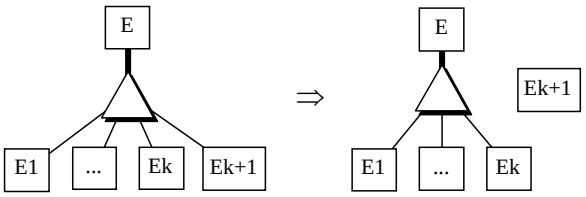
I /

E Créer une relation IS-A entre *PERSONNE* et *REPRESENTANT*.



b) Destruction

D Enlever une relation d'héritage (IS-A).



S ∃ E,E1,...,Ek,Ek+1 ∈ type-entité(S) : (E,{E1,...,Ek}) ← supprimer-Isa(E,{Ek+1})

C T^-

P – E,E1,...,Ek,Ek+1 ∈ type-entites(S)

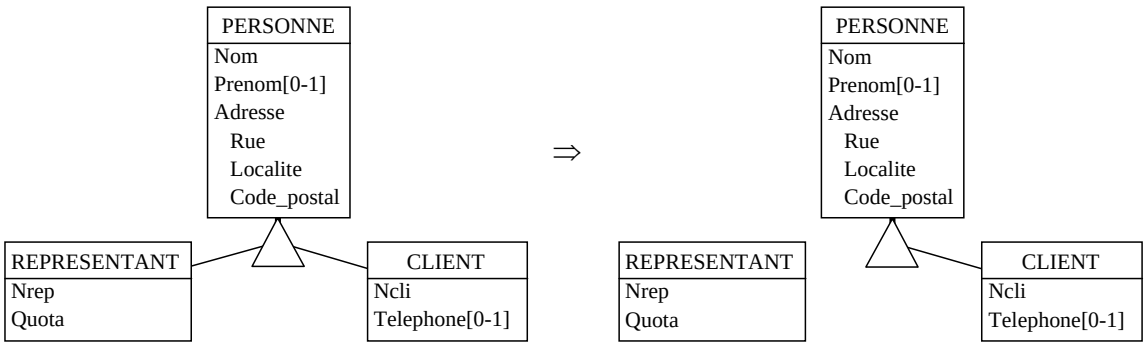
– {E1,...,Ek,Ek+1} = sous-types(E)

Q – {E1,...,Ek} = sous-types(E)

I ∀ e ∈ instance(E), ¬∃ e1 ∈ instance(Ek+1) : e[Ek+1] = e1

Aucune instance de Ek+1 n'est sous-type d'une instance de E. Dans le cas contraire, les contraintes (partition et totale) définies sur les sous-types risquent d'être violées.

E Détruire la relation IS-A entre *PERSONNE* et *REPRESENTANT*.



C.4.3 Modifications de spécifications dans un schéma E/A de base

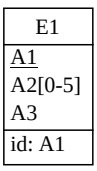
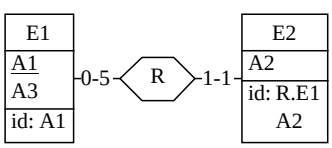
Chaque modification du point C.4.2 est analysée en termes de structures autorisées dans un schéma E/A de base. Il faut déterminer pour chaque structure modifiée dans le modèle E/A riche les modifications correspondantes sur les structures du modèle E/A de base en fonction des transformations qui peuvent être appliquées.

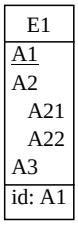
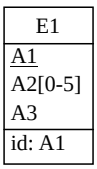
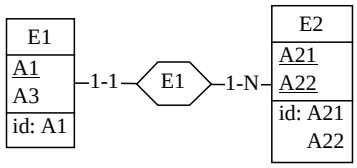
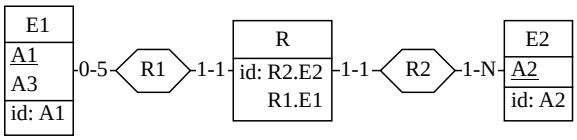
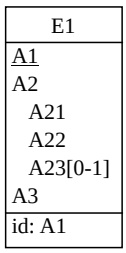
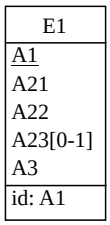
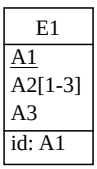
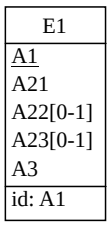
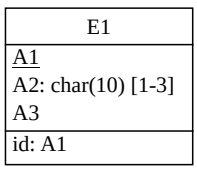
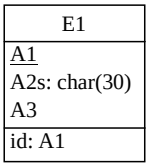
Pour chaque type de modification, une table de transformation est proposée. Elle répertorie les transformations applicables à la structure modifiée afin de la transformer en structures conformes au modèle E/A de base ainsi que la traduction des modifications du modèle riche en modifications du modèle de base. Les modifications du modèle de base, étudiées en détail dans les sections C.2 et C.3, sont seulement référencées. Seules les modifications qui découlent de changement de nature des objets dans le schéma E/A riche et qui nécessitent des changements de transformation sont approfondies.

C.4.3.1 Modifications relatives aux attributs

Un attribut multivalué et/ou décomposable peut être transformé en type d'entités (par instance ou par valeur), un attribut décomposable peut être désagrégé, un attribut multivalué peut être instancié (transformé en une série d'attributs monovalué) ou concaténé (transformé en un attribut monovalué). Pour chaque modification, un tableau contenant le type de la transformation appliquée donne les modifications au niveau des spécifications du modèle de base. La création est illustrée par des exemples qui seront utilisés par les autres modifications.

a) Création

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités E2 par instance. 	Création du type d'entités E2 (page 38). Création du type d'associations R (page 44). Création de l'attribut E2.A2 (page 39). Création de l'identifiant de E2 (page 49). 
Transformation de l'attribut A2 en un type d'entités par valeur.	Création d'un type d'entités E2 (page 38). Création d'un attribut E2.A2 (page 39). Création d'un identifiant de E2 (page 49).

Si A2 est monovalué :  Si A2 est multivalué, on doit aussi transformer le type d'associations plusieurs-à-plusieurs R en un type d'entités.  	Création d'un type d'associations R[a] (page 44).  Création d'un type d'entités R (page 38). Création de deux types d'associations R1 et R2 (page 44). Création d'un identifiant de R[b] (page 49). 
Désagrégation de l'attribut décomposable A2.  	Création des attributs A21, A22 et A23 (page 39). 
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.  	Création des attributs A21,A22 et A23 (page 39). 
Transformation de l'attribut multivalué A2 en un attribut monovalué  	Création de l'attribut A2s (page 39) 

a. Si A2 est monovalué, le type d'associations R est un-à-plusieurs (structure autorisée dans le modèle E/A de base).

b. Si A2 est multivalué, le type d'associations R est plusieurs-à-plusieurs (structure non autorisée dans le modèle E/A de base). R est donc transformé en type d'entités.

b) Destruction

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Destruction du type d'associations R (page 45). Destruction du type d'entités E2 (page 38).
Transformation de l'attribut A2 en un type d'entités par valeur.	Destruction d'un type d'entités E2 (page 38). Si A2 était monovalué, destruction du type d'associations R (page 45). Si A2 était multivalué, destruction du type d'entités R. (page 38) et des types d'associations R1 et R2 (page 45).
Désagrégation de l'attribut décomposable A2.	Destruction des attributs A21, A22 et A23 (page 40).
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Destruction des attributs A21, A22 et A23 (page 40).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	Destruction de l'attribut A2s (page 40).

c) Renommage

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Renommage du type d'entités E2 (page 39) Renommage du type d'associations R (page 45)
Transformation de l'attribut A2 en un type d'entités par valeur.	Renommage du type d'entités E2 (page 39). Si A2 monovalué, renommage du type d'associations R (page 45). Si A2 était multivalué, renommage du type d'entités R (page 39) et des types d'associations R1 et R2 (page 45).
Désagrégation de l'attribut décomposable A2.	Renommage des attributs A21, A22 et A23[a] (page 40).
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Renommage des attributs A21, A22 et A23 (page 40).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	Renommage de l'attribut A2s (page 40).

a. Pour la désagrégation, il n'est peut-être pas nécessaire de renommer les attributs si on ne leur a pas donné un préfixe rappelant le nom de l'attribut désagrégé.

De manière générale, le renommage d'un attribut dans le modèle E/A riche n'entraîne pas nécessairement des renommages au niveau E/A de base.

d) Extension de domaine

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Extension du domaine de l'attribut E2.A2 (page 42).
Transformation de l'attribut A2 en un type d'entités par valeur.	Extension du domaine de l'attribut E2.A2 (page 42).
Désagrégation de l'attribut décomposable A2.	[a]
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Extension du domaine des attributs A21, A22 et A23 (page 42).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	Extension du domaine de l'attribut A2s (page 42).

a. Un attribut décomposable n'a pas de domaine.

e) Restriction de domaine

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Restriction du domaine de l'attribut E2.A2 (page 43).
Transformation de l'attribut A2 en un type d'entités par valeur.	Restriction du domaine de l'attribut E2A2 (page 43).
Désagrégation de l'attribut décomposable A2.	[a]
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Restriction du domaine des attributs A21, A22 et A23 (page 43).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	Restriction du domaine de l'attribut A2s (page 43).

a. Un attribut décomposable n'a pas de domaine.

f) Changement de type

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Changement du type de l'attribut E2.A2 (page 44).
Transformation de l'attribut A2 en un type d'entités par valeur.	Changement du type de l'attribut E2.A2 (page 44).
Désagrégation de l'attribut décomposable A2.	[a]
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Changement du type des attributs A21, A22 et A23 (page 44).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	Changement du type de l'attribut A2s (page 44).

a. Un attribut décomposable n'a pas de type.

g) Augmentation de la cardinalité minimum

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Augmentation de la cardinalité minimum du rôle joué par E1 dans R (page 47).
Transformation de l'attribut A2 en un type d'entités par valeur.	Si A2 était monovalué, augmentation de la cardinalité minimum du rôle joué par E1 dans R (page 47). Si A2 était multivalué, augmentation de la cardinalité minimum du rôle joué par E1 dans R1 (page 47).
Désagrégation de l'attribut décomposable A2.	Les attributs A21, A22 et A23 deviennent obligatoires (page 42).
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Les attributs A21, A22 et A23 deviennent obligatoires (page 42).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	L'attribut A2s devient obligatoire (page 42).

h) Diminution de la cardinalité minimum

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Diminution de la cardinalité minimum du rôle joué par E1 dans R (page 48).
Transformation de l'attribut A2 en un type d'entités par valeur.	Si A2 était monovalué, diminution de la cardinalité minimum du rôle joué par E1 dans R (page 48). Si A2 était multivalué, diminution de la cardinalité minimum du rôle joué par E1 dans R1 (page 48).
Désagrégation de l'attribut décomposable A2.	Les attributs A21, A22 et A23 deviennent facultatifs (page 41).
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Les attributs A21, A22 et A23 deviennent facultatifs (page 41).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	L'attribut A2s devient facultatif (page 41).

i) Augmentation de la cardinalité maximum

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Augmentation de la cardinalité maximum du rôle joué par E1 dans R (page 46).
Transformation de l'attribut A2 en un type d'entités par valeur.	Si A2 était monovalué, augmentation de la cardinalité maximum du rôle joué par E1 dans R (page 46). Si A2 était multivalué, augmentation de la cardinalité maximum du rôle joué par E1 dans R1 (page 46).
Désagrégation de l'attribut décomposable A2.	[a]
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Création de nouveaux attributs (page 39).
Transformation de l'attribut multivalué A2 en un attribut monovalué.	Extension du domaine de l'attribut A2s (page 57).

a. Cette transformation n'est pas autorisée car l'attribut décomposable doit être monovalué.

j) Diminution de la cardinalité maximum

Schéma E/A riche	Schéma E/A base
Transformation de l'attribut A2 en un type d'entités par instance.	Diminution de la cardinalité maximum du rôle joué par E1 dans R (page 47).
Transformation de l'attribut A2 en un type d'entités par valeur.	Si A2 était monovalué, diminution de la cardinalité maximum du rôle joué par E1 dans R (page 47). Si A2 était multivalué, diminution de la cardinalité maximum du rôle joué par E1 dans R1 (page 47).
Désagrégation de l'attribut décomposable A2.	[a]
Transformation de l'attribut multivalué A2 en une série d'attributs monovalués.	Destruction d'attributs (page 40).
Transformation de l'attribut multivalué A2 en un attribut monovalué	Restriction du domaine de l'attribut A2s (page 43).

a. Cette transformation n'est pas autorisée car l'attribut décomposable est obligatoirement monovalué.

k) Changement de transformation

Au niveau du modèle E/A de base, la transformation appliquée sur un attribut peut changer d'un schéma à l'autre. Il y a deux raisons possibles à ce changement :

1. Au niveau du schéma E/A riche, les modifications apportées à un attribut change sa nature (voir le point C.4.2.1 k). La transformation originale n'est plus applicable et doit être remplacée par une autre.
   Par exemple, un attribut décomposable monovalué devient multivalué. Dans ce cas, la transformation de désagrégation n'est plus applicable. Il faut utiliser la transformation d'un attribut en un type d'entités.
2. Le concepteur décide de changer de transformation, pour des raisons tout à fait légitimes, sans qu'aucune modification n'intervienne.
   Par exemple, il décide de ne plus représenter un attribut multivalué par un type d'entités mais bien par une série d'attributs.

Le tableau C.2 reprend les six transformations possibles sur un attribut [1]. Les lignes présentent les transformations appliquées sur les spécifications originales et les colonnes, celles appliquées sur les nouvelles spécifications. Pour chaque cellule du tableau, les raisons qui ont amené le changement de transformation sont précisées. L'abréviation *chgt. trsf.* signifie un changement de transformation sans modification des spécifications. L'abréviation *multi. en mono. déc* indique qu'un attribut multivalué (décomposable ou atomique) a été transformé en un attribut décomposable monovalué.

		Nouvelles spécifications					
		Attribut ato. mono.	**Att. en TE par instance**	**Att. en TE par valeur**	**Instanciation**	**Désagrégation**	**Concaténation**
Spécifications originales	**Att. ato. mono.**		- mono. ato. en multi.	- mono. ato. en multi.	- mono. ato. en multi. ato.	- mono. ato. en déc. ato.	- mono. ato. en multi. ato.
	Att. en TE par instance	- multi. en mono. ato.		- chgt. trsf. - multi. ato. en multi. déc. - multi. déc. en multi. ato.	- chgt. trsf. - multi. déc. en multi. ato.	- multi. en mono. déc.	- chgt. trsf. - multi. déc. en multi. ato.
	Att. en TE par valeur	- multi. en mono. ato.	- chgt. trsf. - multi. ato. en multi. déc. - multi. déc. en multi. ato.		- chgt. trsf. - multi. déc. en multi. ato.	- multi. en mono. déc.	- chgt. trsf. - multi. déc. en multi. ato.
	Instanciation	- multi. ato. en mono. ato.	- chgt. trsf. - multi. ato. en multi. déc.	- chgt. trsf. - multi. ato. en multi. déc.		- multi. ato. en mono. déc.	- chgt. trsf.
	Désagrégation	- déc. ato. en mono. ato.	- mono. déc. en multi.	- mono. déc. en multi.	- mono. déc. en multi. ato.		- mono. déc. en multi. ato.
	Concaténation	- multi. ato. en mono. ato.	- chgt. trsf. - multi. ato. en multi. déc.	- chgt. trsf. - multi. ato. en multi. déc.	- chgt. trsf.	- multi. ato. en mono. déc.	

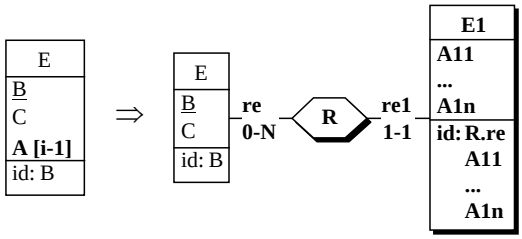
Tableau C.2 - Tableau des modifications relatives à un attribut entraînant des changements de transformations.

Toutes les changements de transformations possibles sont maintenant analysés. Trente cas différents ont été répertoriés.

1. Il faut tenir compte du cas où aucune transformation n'est appliquée lorsqu'on est en présence d'un attribut monovalué atomique. Cette absence de transformation est à ajouter aux cinq transformations possibles sur un attribut décomposable et/ou multivalué.

Transformation d'un attribut atomique monovalué en un type d'entités (représentation des instances)

D Passer d'un attribut monovalué atomique à un type d'entités (représentation par instance) d'un attribut multivalué. Si c'est une représentation par instance d'un attribut multivalué atomique, n est égal à 1.



S $\exists$ E $\in$ type-entites(S), A $\in$ attribut(E) : (E1,R) $\leftarrow$ Att-mono-ato-en-TE-inst(E,A)

C T^+

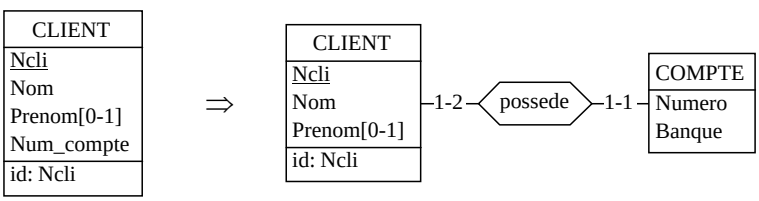
P
- E $\in$ type-entites(S)
- A $\in$ attribut(E)
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

Q
- A $\notin$ attribut(E)
- E1 $\in$ type-entites(S) $\wedge$ nom(E1) = ne1
- A11,...,A1n $\in$ attribut(E1) $\wedge$ nom(A11) = na11 $\wedge$ ... $\wedge$ nom(A1n) = na1n
- $\forall$ k $\in$ [1..n] : card(A1k) = [1-1] $\wedge$ type(A1k) = car | num | reel | date | bool
- R $\in$ type-associations(S) $\wedge$ nom(R) = nr
- re, re1 $\in$ role(R) $\wedge$ nom(re) = nre $\wedge$ nom(re1) = nre1
- card(re) = [i-j] $\wedge$ j > 1 $\wedge$ card(re1) = [1-1]
- id $\in$ identifiant(E1) $\wedge$ nom(id) = nid
- composant(id) = {re,A11,...,A1n}

I $\forall$ e $\in$ instance(E), $\exists!$ r $\in$ instance(R), $\exists!$ e1 $\in$ instance(E1) : r[E,E1] = (e,e1) $\wedge$ desagreger(e[A],e1[A11],...,e1[A1n])

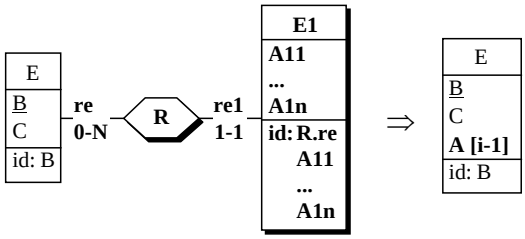
Chaque instance de A est désagrégée dans A11, ..., A1n d'une instance de E1. Si n = 1, e1[A11] = e[A]. Les domaines de valeurs de A, A11, ..., A1n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut atomique monovalué *Num_compte* de *CLIENT* est transformé en un type d'entités *COMPTE* contenant deux attributs (*Numero* et *Compte*).



Transformation d'un type d'entités (représentation des instances) en un attribut monovalué atomique

D Passer d'un type d'entités (représentation par instance d'un attribut multivalué) à un attribut monovalué atomique. Si c'est la représentation par instance d'un attribut multivalué atomique, n est égal à 1.



S ∃ E ∈ type-entites(S), E1 ∈ type-entites(S), R ∈ type-associations(S) : (E,A) ← TE-inst-en-Att-mono-ato(E1,R)

C T^-

P
- E ∈ type-entites(S)
- E1 ∈ type-entites(S)
- A11, ..., A1n ∈ attribut(E1)
- ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car | num | reel | date | bool
- R ∈ type-associations(S)
- re, re1 ∈ role(R)
- card(re) = [i-j] ∧ j > 1 ≥ i ∧ card(re1) = [1-1]
- id ∈ identifiant(E1)
- composant(id) = {re,A11,...,A1n}

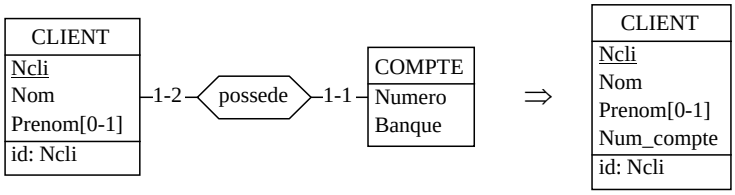
Q
- E1 ∉ type-entites(S)
- A11, ..., A1n ∉ attribut(E1)
- R ∉ type-associations(S)
- re, re1 ∉ role(R)
- id ∉ identifiant(E1)
- A ∈ attribut(E) ∧ nom(A) = na
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

I ∀ e1 ∈ instance(E1), r ∈ instance(R), ∃! e ∈ instance(E) : r[re,re1] = (e,e1) ∧ agreger(e1[A11],...,e1[A1n],e[A])

∀ e ∈ instance(E), ∃! r ∈ instance(r) : r[re] = e

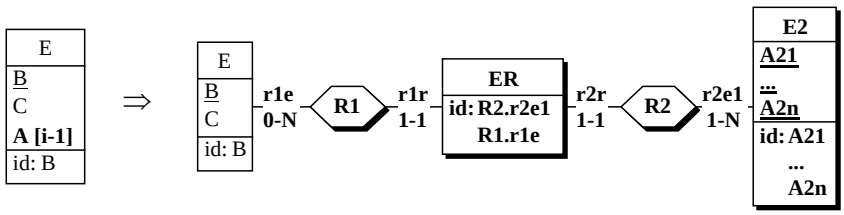
Pour chaque instance de E1, les valeurs de A11,...,A1n sont agrégés dans A. Si n = 1, e[A] = e1[A11]. Si, pour une instance de E, il existe plusieurs instances de R et donc plusieurs instances de E1, on prend les valeurs de la première (les autres sont perdues). Les domaines de valeurs de A, A11, ..., A1n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *COMPTE* est transformé en un attribut atomique monovalué *Num_compte* de *CLIENT*. Les attributs *Numero* et *Compte* de *COMPTE* sont agrégés dans l'attribut *Num_compte*.



Transformation d'un attribut monovalué atomique en un type d'entités (représentation des valeurs)

D Passer d'un attribut monovalué atomique à un type d'entités (représentation par valeur d'un attribut multivalué). Dans le cas d'une représentation par valeur d'un attribut multivalué atomique, n est égal à 1.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : ({E2,ER},{R1,R2}) ← Att-mono-ato-en-TE-val(E,A)

C T^+

P
- E ∈ type-entites(S)
- A ∈ attribut(E)
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

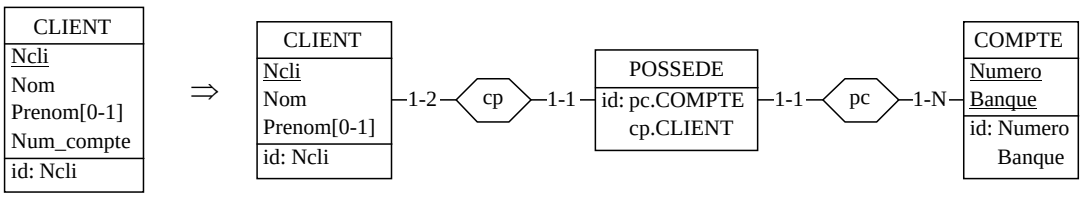
Q
- A ∉ attribut(E)
- E2,ER ∈ type-entites(S) ∧ nom(E2) = ne2 ∧ nom(ER) = ner
- A21,...,A2n ∈ attribut(E1) ∧ nom(A21) = na21 ∧ ... ∧ nom(A2n) = na2n
- ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car | num | reel | date | bool
- R1,R2 ∈ type-associations(S) ∧ nom(R1) = nr1 ∧ nom(R2) = nr2
- r1e, r1r ∈ role(R1) ∧ nom(r1e) = nr1e ∧ nom(r1r) = nr1r
- card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2) ∧ nom(r2r) = nr2r ∧ nom(r2e1) = nr2e1
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2) ∧ nom(ide2) = nide2
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER) ∧ nom(ider) = nider
- composant(ider) = {r1e,r2e1}

I ∀ e ∈ instance(E), ∃! e2 ∈ instance(E2), er ∈ instance(ER), r1 ∈ instance(R1), r2 ∈ instance(R2) : r1[r1e,r1r] = (e,er) ∧ r2[r2r,r2e1] = (er,e2) ∧ desagreger(e[A],e2[A21],...,e2[A2n])

Chaque instance de A est désagrégée dans une instance de E2 à condition qu'il n'existe pas déjà une instance de E2 avec les mêmes valeurs (à cause de l'identifiant). Si n = 1, e2[A21]

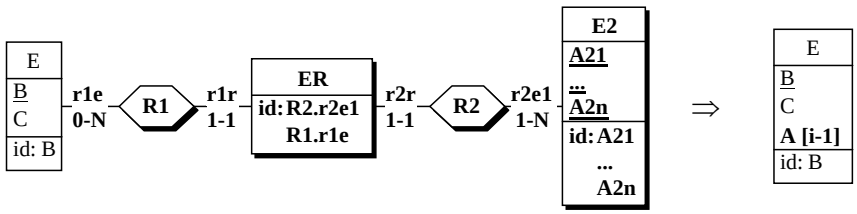
= e[A]. Les domaines de valeurs de A, A21, ..., A2n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut atomique monovalué *Num_compte* de *CLIENT* est transformé en un type d'entités *COMPTE* contenant deux attributs (*Numero* et *Compte*). Le type d'associations *possede* entre *CLIENT* et *COMPTE* est transformé en un type d'entités *POSSEDE*.



Transformation d'un type d'entités (représentation des valeurs) en un attribut monovalué atomique

D Passer d'un type d'entités (représentation par valeur d'un attribut multivalué) à un attribut monovalué atomique. Dans la représentation par valeur d'un attribut multivalué atomique, n est égal à 1.



S ∃ E ∈ type-entites(S), E2,ER ∈ type-entites(S), R1,R2 ∈ type-associations(S) : (E,A) ← Att-mono-ato-en-TE-val(E2,ER,R1,R2)

C T^-

P
- E,E2,ER ∈ type-entites(S)
- A21, ..., A2n ∈ attribut(E1)
- ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car | num | reel | date | bool
- R1,R2 ∈ type-associations(S)
- r1e, r1r ∈ role(R1)
- card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2)
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2)
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER)
- composant(ider) = {r1e,r2e1}

Q
- E2,ER ∉ type-entites(S)
- R1,R2 ∉ type-associations(S)
- A ∈ attribut(E) ∧ nom(A) = na

– card(A) = [i-1]

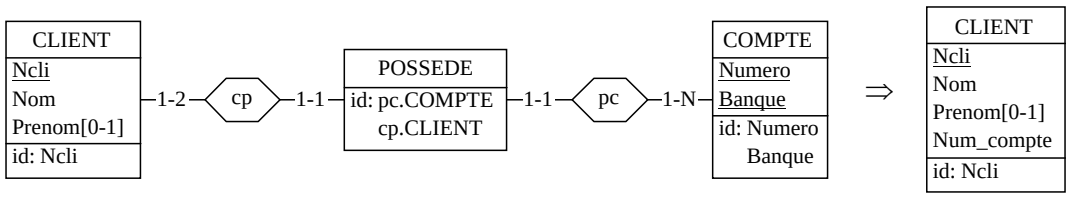
– type(A) = car | num | reel | date | bool

I ∀ e2 ∈ instance(E2), er ∈ instance(ER), r1 ∈ instance(R1), r2 ∈ instance(R2), ∃! e ∈ instance(E) : r1[r1e,r1r] = (e,er) ∧ r1[r2r,r2e2] = (er,e2) ∧ agreger(e2[A21],...,e2[A2n],e[A])

∀ e ∈ instance(E), ∃! r1 ∈ instance(r1) : r1[r1e] = e

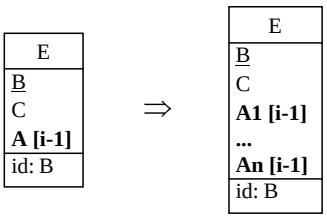
Pour chaque instance de E2, les instances de A21,...,A2n sont agrégées dans une instance de A. Si n = 1, l'instance de A21 est stockée dans l'instance de A. Si, pour chaque instance de E, il existe plusieurs instances de ER et de E2, on ne prend que les valeurs de la première instance, les autres étant perdues. Les domaines de valeurs de A, A21, ..., A2n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *POSSEDE* est transformé en un type d'associations *possede* et le type d'entités *COMPTE* est ensuite transformé en un attribut atomique monovalué *Num_compte* de *CLIENT*. Les attributs *Numero* et *Compte* de *COMPTE* sont agrégés dans l'attribut *Num_compte*.



Transformation (par désagrégation) d'un attribut monovalué atomique en une série attributs

D Passer d'un attribut monovalué atomique à la désagrégation d'un attribut décomposable.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : (E,{A1,...,An}) ← Att-mono-ato-en-desagreger-Att(E,A)

C T^+

P – E ∈ type-entites(S)

– A ∈ attribut(E)

– card(A) = [i-1]

– type(A) = car | num | reel | date | bool

Q – A ∉ attribut(E)

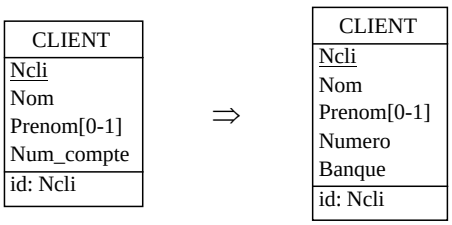
– A1,...,An ∈ attribut(E) ∧ nom(A1) = na1 ∧ ... ∧ nom(An) = nan

– ∀ k ∈ [1..n] : card(Aj) = [i-1] ∧ type(Aj) = car | num | reel | date | bool

I ∀ e ∈ instance(E) : desagreger(e[A],e[A1],...,e[An])

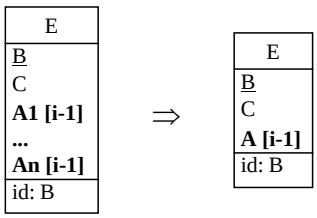
Pour chaque instance de E, l'instance de A est désagrégée dans celles de A1, ..., An. Les domaines de valeurs de A, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut *Num_compte* du type d'entités *CLIENT* est transformé en deux attributs *Numero* et *Compte*.



Transformation d'une série d'attributs (désagrégation) en un attribut monovalué atomique

D Passer de la désagrégation d'un attribut décomposable à un attribut monovalué atomique.



S $\exists$ E $\in$ type-entites(S), A1,...,An $\in$ attribut(E) : (E,A) $\leftarrow$ desagreger-Att-en-Att-mono-ato(E,{A1,...,An})

C T^{-}

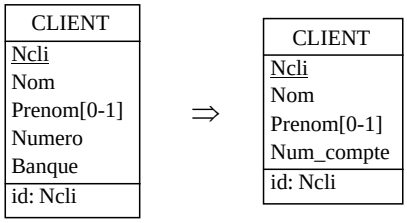
P
- E $\in$ type-entites(S)
- A1,...,An $\in$ attribut(E)
- $\forall$ k $\in$ [1..n] : card(Aj) = [i-1] $\wedge$ type(Aj) = car | num | reel | date | bool

Q
- A1 $\notin$ attribut(E) $\wedge$ ... $\wedge$ An $\notin$ attribut(E)
- A $\in$ attribut(E) $\wedge$ nom(A) = na
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

I $\forall$ e $\in$ instance(E) : agreger(e[A1],...,e[An],e[A])

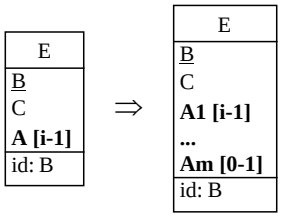
Pour chaque instance de E, les instances de A1,...,An sont agrégées dans une instance de A. Les domaines de valeurs de A, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Numero* et *Compte* du type d'entités *CLIENT* sont transformés en un attribut *Num_compte*.



Transformation (par instanciation) d'un attribut atomique monovalué en une série d'attributs

D Passer d'un attribut monovalué atomique à l'instanciation d'un attribut atomique multivalué.



S $\exists$ E $\in$ type-entites(S), A $\in$ attribut(E) : (E,{A1,...,Am}) $\leftarrow$ Att-mono-ato-en-Serie-Att(E,A)

C T^+

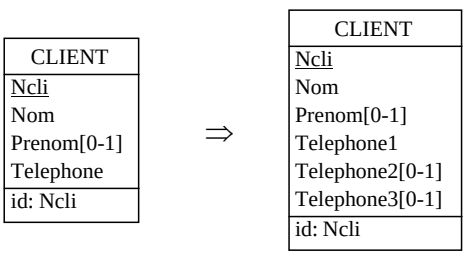
P
- E $\in$ type-entites(S)
- A $\in$ attribut(E)
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

Q
- A $\notin$ attribut(E)
- A1,...,Am $\in$ attribut(E) $\wedge$ nom(A1) = na1 $\wedge$ ... $\wedge$ nom(Am) = nam
- card(A1) = [i-1] $\wedge$ ($\forall$ k $\in$ [2..m] : card(Ak) = [0-1])
- $\forall$ k $\in$ [1..m] : type(Aj) = car | num | reel | date | bool

I $\forall$ e $\in$ instance(E) : e[A1] = e[A]

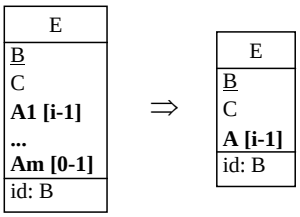
Pour chaque instance de E, l'instance de A est mise dans A1. Les domaines de valeurs de A et A1 sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut *Telephone* du type d'entités *CLIENT* est instancié en trois attributs *Telephone1*, *Telephone2* et *Telephone3*.



Transformation d'une série d'attributs (instanciation) en un attribut monovalué atomique

D Passer de l'instanciation d'un attribut atomique multivalué à un attribut monovalué atomique.



S ∃ E ∈ type-entites(S), A1,...,Am ∈ attribut(E) : (E,A) ← Serie-Att-en-Att-mono-ato(E,{A1,...,Am})

C T-

P
- E ∈ type-entites(S)
- A1,...,Am ∈ attribut(E)
- (∀ j ∈ [2..k] : card(Aj) = [i-1]) ∧ (∀ j ∈ [k..m] : card(Aj) = [0-1])
- ∀ k ∈ [1..m] : type(Aj) = car | num | reel | date | bool

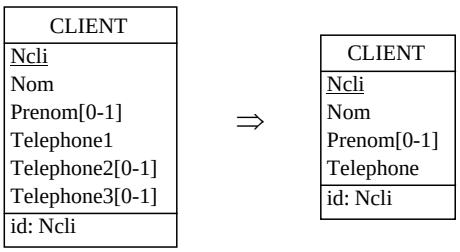
Q
- A1,...,Am ∉ attribut(E)
- A ∈ attribut(E) ∧ nom(A) = na
- card(A) = [i-1]
- type(A) = car | num | reel | date | bool

I ∀ e ∈ instance(E) : e[A] = e[A1]

∀ e ∈ instance(E) : e[A2] = null ∧ ... ∧ e[Am] = null

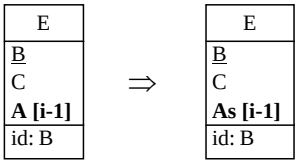
Pour chaque instance de E, l'instance de A1 est stockée dans A et les instances des A2, ..., Am sont perdues. Les domaines de valeurs de A et A1 sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Telephone1*, *Telephone2* et *Telephone3* du type d'entités *CLIENT* sont concaténés dans l'attribut *Telephone*.



Transformation d'un attribut monovalué atomique en un attribut concaténé

D Passer d'un attribut monovalué atomique à la concaténation d'un attribut atomique multivalué.



S ∃ E ∈ type-entites(S), A ∈ attribut(E) : (E,As) ← Att-mono-ato-en-Att-concat(E,A)

C T^+ car le domaine de valeurs de As est normalement plus grand que celui de A.

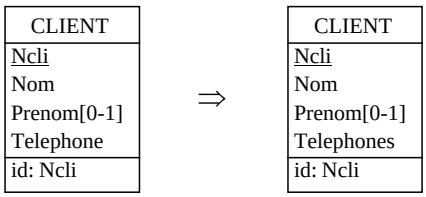
P
- E ∈ type-entites(S)
- A ∈ attribut(E)
- card(A) = [i-1]
- type(A) = car

Q
- A ∉ attribut(E)
- As ∈ attribut(E) ∧ nom(As) = nas
- card(As) = [i-1]
- type(As) = car

I ∀ e ∈ instance(E) : e[As] = e[A]

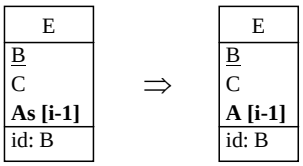
Pour chaque instance de E, l'instance de A est mise dans As. Les domaines de valeurs de A et As sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut *Telephone* du type d'entités *CLIENT* est concaténé dans l'attribut *Telephone*s.



Transformation d'un attribut concaténé en un attribut monovalué atomique

D Passer de la concaténation d'un attribut atomique multivalué à un attribut monovalué atomique.



S ∃ E ∈ type-entites(S), As ∈ attribut(E) : (E,A) ← Att-concat-en-Att-mono-ato(E,As)

C T^- car le domaine de valeurs de As est normalement plus grand que celui de A.

P
- E ∈ type-entites(S)
- As ∈ attribut(E)
- card(As) = [i-1]

– type(As) = car

Q – As ∉ attribut(E)

– A ∈ attribut(E) ∧ nom(A) = na

– card(A) = [i-1]

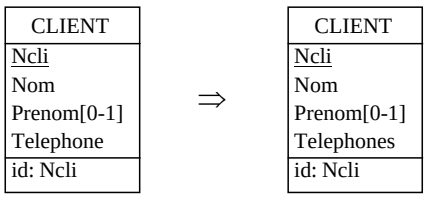
– type(A) = car

I ∀ e ∈ instance(E) : desagreger(e[As],e[a])

∀ e ∈ instance(E) : longueur(e[As]) ≤ longueur(e[a])

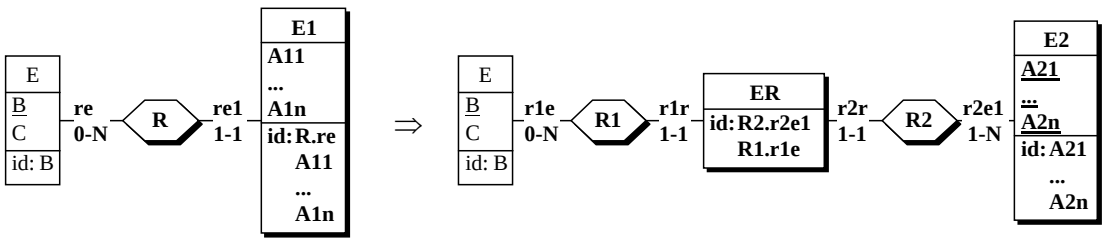
Pour chaque instance de E, une partie de l'instance de As est mise dans A. Comme la longueur de A est normalement inférieure à celle de As, on ne met qu'une partie de l'instance de As dans A, le reste étant perdu. Les domaines de valeurs de A et As sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut *Telephones* du type d'entités *CLIENT* est désagrégé dans l'attribut *Telephone*.



Transformation d'un type d'entités (représentation des instances) en un type d'entités (représentation des valeurs)

D Passer d'une représentation par instance à une représentation par valeur d'un attribut multivalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1 (cela vaut aussi pour une représentation par valeur).



S ∃ E,E1 ∈ type-entites(S), R ∈ type-associations(S) : ({E2,ER},{R1,R2}) ← TE-inst-en-TE-val(E1,R)

C T^+ si un attribut atomique multivalué devient décomposable.

T^- si un attribut décomposable multivalué devient atomique.

$T^=$ s'il s'agit d'un changement de transformation sans modification.

P – E ∈ type-entites(S)

– E1 ∈ type-entites(S)

– A11,...,A1n ∈ attribut(E1)

– ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car | num | reel | date | bool

– R ∈ type-associations(S)

– re, re1 ∈ role(R)

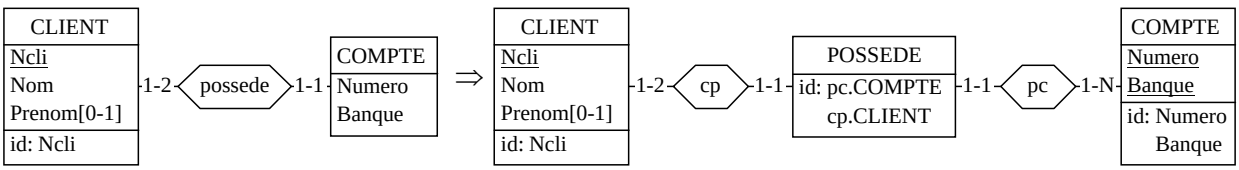
– card(re) = [i-j] ∧ j > 1 ≥ i ∧ card(re1) = [1-1]
– id ∈ identifiant(E1)
– composant(id) = {re,A11,...,A1n}

Q
– E1 ∉ type-entites(S)
– R ∉ type-associations(S)
– E2,ER ∈ type-entites(S) ∧ nom(E2) = ne1 ∧ nom(ER) = nr
– A21,...,A2n ∈ attribut(E2) ∧ nom(A21) = na1 ∧ ... ∧ nom(A2n) = nan
– ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car | num | reel | date | bool
– R1,R2 ∈ type-associations(S) ∧ nom(R1) = nr1 ∧ nom(R2) = nr2
– r1e, r1r ∈ role(R1) ∧ nom(r1e) = nr1e ∧ nom(r1r) = nr1r
– card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
– r2r, r2e1 ∈ role(R2) ∧ nom(r2r) = nr2r ∧ nom(r2e1) = nr2e1
– card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
– ide1 ∈ identifiant(E2) ∧ nom(ide1) = nide1
– composant(ide1) = {r2r,A21,...,A2n}
– idr ∈ identifiant(ER) ∧ nom(idr) = nidr
– composant(idr) = {r1e,r2e1}

I ∀ e ∈ instance(E), e1 ∈ instance(E1), r ∈ instance(R), ∃ e2 ∈ instance(E2), er ∈ instance(ER), r1 ∈ instance(R1), r2 ∈ instance(R2) : r[re,re1] = (e,e1) ∧ r1[r1e,r1r] = (e,er) ∧ r2[r2r,r2e1] = (er,e2) ∧ e2[A21] = e1[A11] ∧ ... ∧ e2[A2n] = e1[A1n]

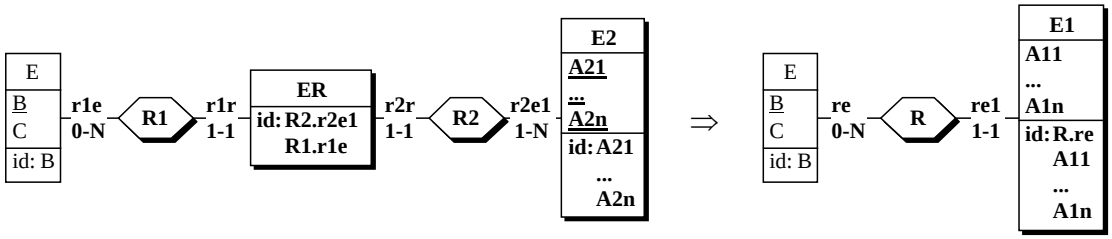
Pour chaque instance de E, les instances de E1 sont mises dans les instances de E2 à condition qu'il n'existe pas déjà une instance de E2 avec les mêmes valeurs. Si n = 1 dans la représentation par instance, e2[A21] = desagreger(e1[A11],e2[A21],...,e2[A2n]). Si n = 1 dans la représentation par valeur, agreger(e1[A11],...,e1[A1n],e2[A21]). Les domaines de valeurs de A11, ..., A1n, A21, ..., A2n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *COMPTE* est transformé en un type d'entités représentant les valeurs distinctes de *COMPTE*. Le type d'associations *possede* est transformé en un type d'entités *POSSEDE*.



Transformation d'un type d'entités (représentation des valeurs) en un type d'entités (représentation des instances)

D Passer d'une représentation par valeur à une représentation par instance d'un attribut multivalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1 (cela vaut aussi pour une représentation par instance).



S ∃ E,ER,E2 ∈ type-entites(S), R1,R2 ∈ type-associations(S) : (E1,R) ← TE-val-en-TE-inst({E2,ER},{R1,R2})

C T^+ si un attribut atomique multivalué devient décomposable.

T^- si un attribut décomposable multivalué devient atomique.

$T^=$ s'il s'agit d'un changement de transformation sans modification.

P
- E ∈ type-entites(S)
- E2,ER ∈ type-entites(S)
- A21,...,A2n ∈ attribut(E2)
- ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car | num | reel | date | bool
- R1,R2 ∈ type-associations(S)
- r1e, r1r ∈ role(R1)
- card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2)
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2)
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER)
- composant(ider) = {r1e,r2e1}

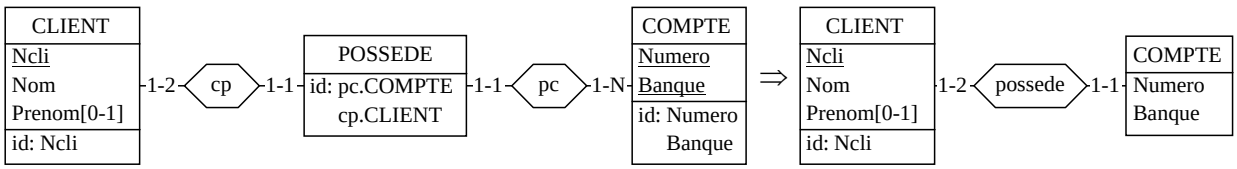
Q
- E2,ER ∉ type-entites(S)
- R1,R2 ∉ type-associations(S)
- E1 ∈ type-entites(S) ∧ nom(E1) = ne1
- A11,...,A1n ∈ attribut(E1) ∧ nom(A11) = na11 ∧ ... ∧ nom(A1n) = na1n
- ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car | num | reel | date | bool
- R ∈ type-associations(S) ∧ nom(R) = nr
- re, re1 ∈ role(R) ∧ nom(re) = nre ∧ nom(re1) = ne1
- card(re) = [i-j] ∧ j > 1 ≥ i
- card(re1) = [1-1]
- ide1 ∈ identifiant(E1) ∧ nom(ide1) = nide1
- composant(id) = {re,A11,...,A1n}

I ∀ e ∈ instance(E), e2 ∈ instance(E2), er ∈ instance(ER), r1 ∈ instance(R1), r2 ∈ instance(R2), ∃ e1 ∈ instance(E1), r ∈ instance(R) : r1[r1e,r1r] = (e,er) ∧ r2[r2r,r2e1] = (er,e2) ∧ r[re,re1] = (e,e1) ∧ e1[A11] = e2[A21] ∧ ... ∧ e1[A1n] = e2[A2n]

Pour chaque instance de E, les instances de E2 sont mises dans les instances de E1. Si n = 1 dans la représentation par valeur, on a desagreger(e2[A21],e1[A11],...,e1[A1n]). Si n = 1

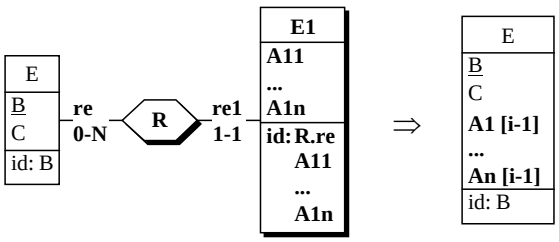
dans la représentation par instance, on a agreger(e2[A21],...,e2[A2n],e1[A11]). Les domaines de valeurs de A11, ..., A1n, A21, ..., A2n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *COMPTE* est transformé en un type d'entités représentant les instances de *COMPTE*. Le type d'entités *POSSEDE* est transformé en un type d'associations *possede*.



Transformation d'un type d'entités (représentation des instances) en une série d'attributs (désagrégation)

D Passer d'une représentation par instance d'un attribut multivalué à la désagrégation d'un attribut décomposable monovalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1.



S ∃ E,E1 ∈ type-entites(S), R ∈ type-associations(S), A11,...,A1n ∈ attribut(E1) : (E,{A1,...,An}) ← TE-inst-en-desagreger-Att(E1,R)

C T^-

P
- E ∈ type-entites(S)
- E1 ∈ type-entites(S)
- A11,...,A1n ∈ attribut(E1)
- ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car | num | reel | date | bool
- R ∈ type-associations(S)
- re, re1 ∈ role(R)
- card(re) = [i-j] ∧ j > 1 ≥ i
- card(re1) = [1-1]
- ide1 ∈ identifiant(E1)
- composant(ide1) = {re,A11,...,A1n}

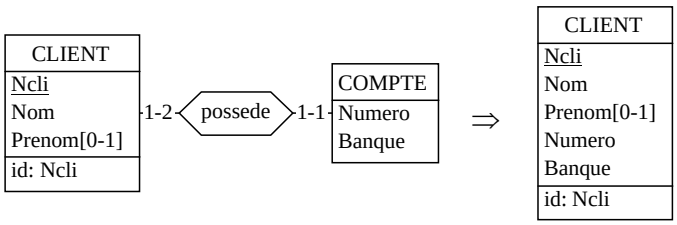
Q
- E1 ∉ type-entites(S)
- A11,...,A1n ∉ attribut(E1)
- R ∉ type-associations(S)
- re, re1 ∉ role(R)
- id ∉ identifiant(E1)
- A1,...,An ∈ attribut(E) ∧ nom(A1) = na1 ∧ ... ∧ nom(An) = nan

– $\forall$ k $\in$ [1..n] : card(Ak) = [i-1] $\wedge$ type(Ak) = car | num | reel | date | bool

I $\forall$ e $\in$ instance(E), $\exists$! e1 $\in$ instance(E1), r $\in$ instance(R) : r[re,re1] = (e,e1) $\wedge$ e[A1] = e1[A11] $\wedge$ ... $\wedge$ e[An] = e1[A1n]

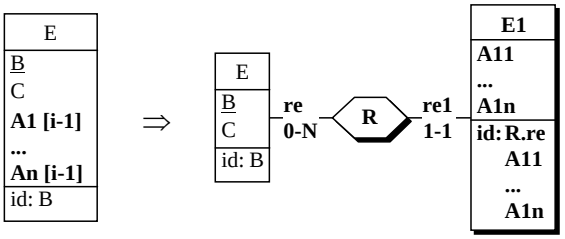
Pour chaque instance de E, la première instance de E1 est mise dans les instances de A1,...,An. Les autres instances de E1 sont perdues. Si n = 1, desagreger(e1[A11],e[A1],...,e[An]). Les domaines de valeurs de A11, ..., A1n, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *COMPTE* est transformé en deux attributs *Numéro* et *Banque* dans le type d'entités *CLIENT*.



Transformation d'une série d'attributs (désagrégation) en un type d'entités (représentation des instances)

D Passer de la désagrégation d'un attribut atomique décomposable à une représentation par instance d'un attribut multivalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1.



S $\exists$ E $\in$ type-entites(S), A1,...,An $\in$ attribut(E) : (E1,R) $\leftarrow$ desagreger-Att-en-TE-inst(E,{A1,...,An})

C T^+

P – E $\in$ type-entites(S)

– A1,...,An $\in$ attribut(E)

– $\forall$ k $\in$ [1..n] : card(Ak) = [i-1] $\wedge$ type(Ak) = car | num | reel | date | bool

Q – A1,...,An $\notin$ attribut(E1)

– E1 $\in$ type-entites(S) $\wedge$ nom(E1) = ne1

– A11,...,A1n $\in$ attribut(E1) $\wedge$ nom(A11) = na11 $\wedge$ ... $\wedge$ nom(A1k) = na1k

– $\forall$ k $\in$ [1..n] : card(A1k) = [1-1] $\wedge$ type(A1k) = car | num | reel | date | bool

– R $\in$ type-associations(S) $\wedge$ nom(R) = nr

– re, re1 $\in$ role(R) $\wedge$ nom(re) = nre $\wedge$ nom(re1) = nre1

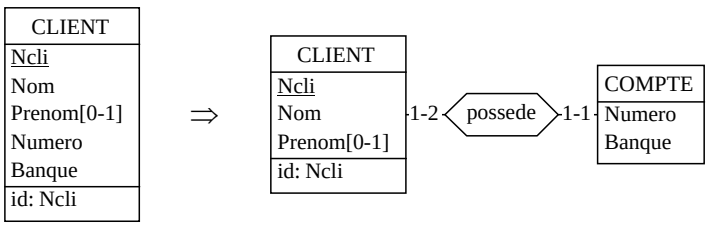
– card(re) = [i-j] $\wedge$ j > 1 $\geq$ i

- card(re1) = [1-1]
- ide1 ∈ identifiant(E1) ∧ nom(ide1) = nide1
- composant(ide1) = {re,A11,...,A1n}

I ∀ e ∈ instance(E), ∃! e1 ∈ instance(E1), r ∈ instance(R) : r[re,re1] = (e,e1) ∧ e1[A11] = e[A1] ∧ ... ∧ e1[A1n] = e[An]

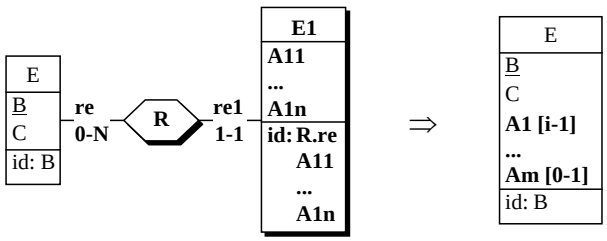
Pour chaque instance de E, les instances de A1,...,An sont mises dans une instance de E1. Si n = 1, agreger(e[A1],...,e[An],e1[A11]). Les domaines de valeurs de A11, ..., A1n, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Numéro* et *Banque* de *CLIENT* sont transformés en un type d'entités *COMPTE*.



Transformation d'un type d'entités (représentation des instances) à une série d'attributs (instanciation)

D Passer d'une représentation par instance d'un attribut multivalué à une instanciation d'un attribut atomique multivalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1.



S ∃ E,E1 ∈ type-entites(S), R ∈ type-associations(S) : (E,{A1,...,Am}) ← TE-inst-en-Serie-Att(E1,R)

C T^-

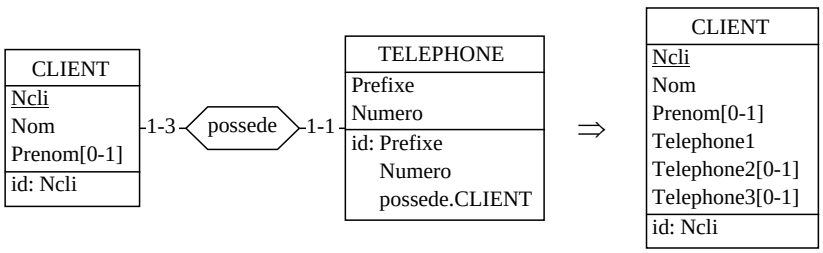
P
- E ∈ type-entites(S)
- E1 ∈ type-entites(S)
- A11,...,A1n ∈ attribut(E1)
- ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car | num | reel | date | bool
- R ∈ type-associations(S)
- re, re1 ∈ role(R)
- card(re) = [i-j] ∧ j > 1 ≥ i
- card(re1) = [1-1]
- id ∈ identifiant(E1)
- composant(id) = {re,A11,...,A1n}

Q – E1 ∉ type-entites(S)
– A11,...,A1n ∉ attribut(E1)
– R ∉ type-associations(S)
– re, re1 ∉ role(R)
– id ∉ identifiant(E1)
– A1,...,Am ∈ attribut(E) ∧ nom(A1) = na1 ∧ ... ∧ nom(Am) = nam
– card(A1) = [i-1] ∧ (∀ k ∈ [2..m] : card(Ak) = [0-1])
– ∀ k ∈ [1..m] : type(Ak) = car | num | reel | date | bool

I ∀ e ∈ instance(E), ei ∈ instance(E1), ri ∈ instance(R), i ∈ [1..m] : ri[re,re1] = (e,ei) ∧ agreger(ei[A11],...,ei[A1n],e[Ai])

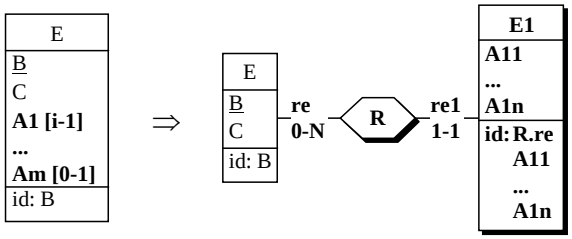
Les m premières instances de E1 reliées à une instance de E sont agrégées dans A1,...,Am (les autres instances de E1 sont perdues). Si n = 1, e[A1] = e1[A11] ∧ ... ∧ e[Am] = em[A11]. Les domaines de valeurs de A11, ..., A1n, A1, ..., Am sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *TELEPHONE* est transformé en trois attributs *Telephone1*, *Telephone2* et *Telephone3*.



Transformation d'une série d'attributs (instanciation) en un type d'entités (représentation des instances)

D Passer de l'instanciation d'un attribut atomique multivalué à une représentation par instance d'un attribut multivalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1.



S ∃ E ∈ type-entites(S), A1,...,Am ∈ attribut(E) : (E1,R) ← Serie-Att-en-TE-inst(E,{A1,...,Am})

C T^+

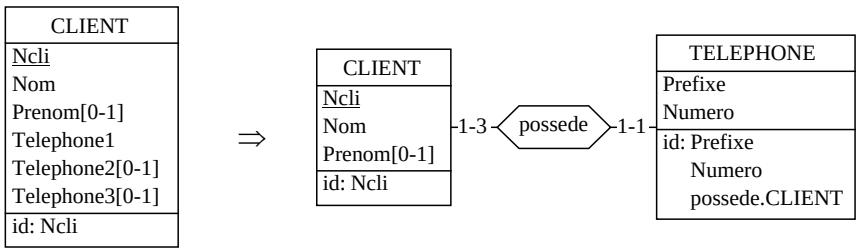
P – E ∈ type-entites(S)
– A1,...,Am ∈ attribut(E)
– card(A1) = [i-1] ∧ (∀ k ∈ [2..m] : card(Ak) = [0-1])
– ∀ k ∈ [1..m] : type(Ak) = car | num | reel | date | bool

Q – A1,...,Am ∉ attribut(E1)
– E1 ∈ type-entites(S) ∧ nom(E1) = ne1
– A11,...,A1n ∈ attribut(E1) ∧ nom(A11) = na11 ∧ ... ∧ nom(A1k) = na1k
– ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car | num | reel | date | bool
– R ∈ type-associations(S) ∧ nom(R) = nr
– re, re1 ∈ role(R) ∧ nom(re) = nre ∧ nom(re1) = nre1
– card(re) = [i-j] ∧ j > 1 ≥ i
– card(re1) = [1-1]
– ide1 ∈ identifiant(E1)
– composant(ide1) = {re,A11,...,A1n}

I ∀ e ∈ instance(E), ∃ ei ∈ instance(E1), ri ∈ instance(R), i ∈ [1..m] : ri[re,re1] = (e,ei) ∧ desagreger(e[Ai],ei[A11],...,ei[A1n])

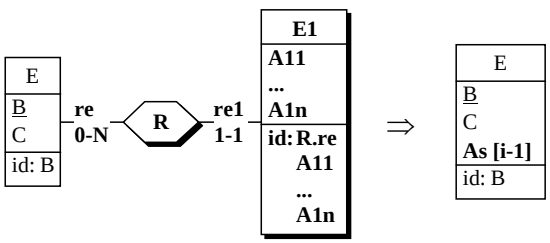
Pour chaque instance de E, les instances de A1,...,Am sont désagrégées dans m instances de E1. Si n = 1, ei[A11] = e[Ai]. Les domaines de valeurs de A11, ..., A1n, A1, ..., Am sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Telephone1*, *Telephone2* et *Telephone3* sont transformés en un type d'entités *TELEPHONE*.



Transformation d'un type d'entités (représentation des instances) en un attribut concaténé

D Passer d'une représentation par instance d'un attribut multivalué à la concaténation d'un attribut atomique multivalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1.



S ∃ E,E1 ∈ type-entites(S), R ∈ type-associations(S) : (E,As) ← TE-inst-en-Att-concat(E1,R)

C T^-

P – E ∈ type-entites(S)
– E1 ∈ type-entites(S)

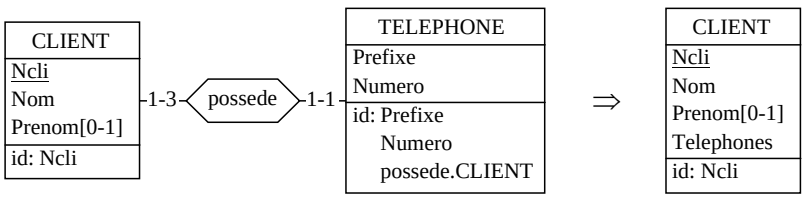
– A11,...,A1n ∈ attribut(E1)
– ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car
– R ∈ type-associations(S)
– re, re1 ∈ role(R)
– card(re) = [i-j] ∧ j > 1 ≥ i ∧ card(re1) = [1-1]
– id ∈ identifiant(E1)
– composant(id) = {re,A11,...,A1n}

Q – E1 ∉ type-entites(S)
– A11,...,A1n ∉ attribut(E1)
– R ∉ type-associations(S)
– re, re1 ∉ role(R)
– ide1 ∉ identifiant(E1)
– As ∈ attribut(E) ∧ nom(As) = nas
– card(As) = [i-1]
– type(As) = car

I ∀ e ∈ instance(E), ei ∈ instance(E1), ri ∈ instance(R), i ∈ [1..m] : ri[re,re1] = (e,ei) ∧ agreger(ei[A11],...,ei[A1n],e[As])

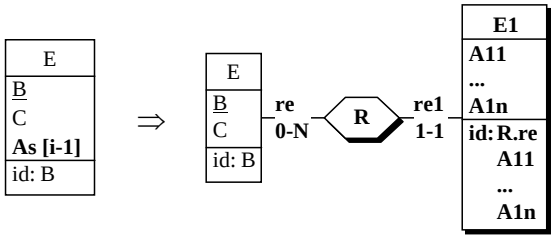
Chaque instance de E1 reliée à une instance de E est agrégée dans As. Si n = 1, e[As] = agreger(e1[A11],...,em[A11]). Les domaines de valeurs de A11, ..., A1n, As sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *TELEPHONE* est transformé en un attribut *Telephones* de *CLIENT*.



Transformation d'un attribut concaténé en un type d'entités (représentation des instances)

D Passer de la concaténation d'un attribut multivalué atomique à une représentation par instance d'un attribut multivalué. Dans le cas d'une représentation par instance d'un attribut atomique multivalué, n est égal à 1.



S ∃ E ∈ type-entites(S), As ∈ attribut(E) : (E1,R) ← Att-concat-en-TE-inst(E,As)

C T^+

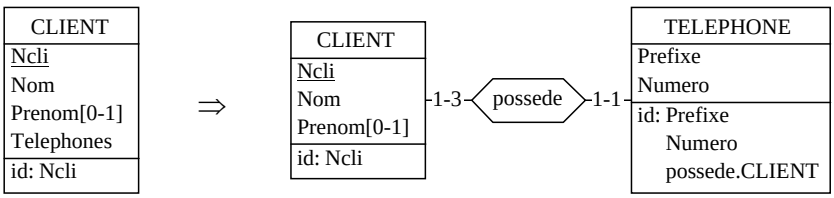
P – E ∈ type-entites(S)
– As ∈ attribut(E)
– card(As) = [i-1]
– type(As) = car

Q – As ∉ attribut(E)
– E1 ∈ type-entites(S) ∧ nom(E1) = ne1
– A11,...,A1n ∈ attribut(E1) ∧ nom(A11) = na11 ∧ ... ∧ nom(A1n) = na1n
– ∀ k ∈ [1..n] : card(A1k) = [1-1] ∧ type(A1k) = car
– R ∈ type-associations(S) ∧ nom(R) = nr
– re, re1 ∈ role(R) ∧ nom(re) = nre ∧ nom(re1) = nre1
– card(re) = [i-j] ∧ j > 1 ≥ i ∧ card(re1) = [1-1]
– ide1 ∈ identifiant(E1) ∧ nom(ide1) = nide1
– composant(ide1) = {re,A11,...,A1n}

I ∀ e ∈ instance(E), ∃ ei ∈ instance(E1), ri ∈ instance(R), i ∈ [1..m] : ri[re,re1] = (e,ei) ∧ desagreger(e[As],ei[A11],...,ei[A1n])

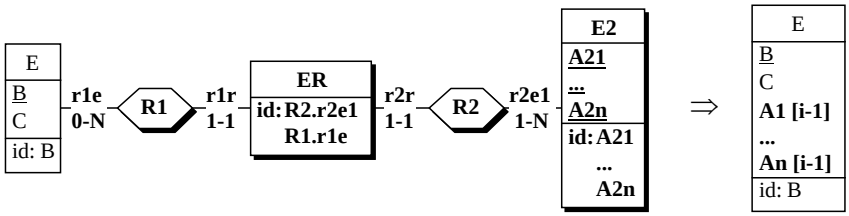
Pour chaque instance de E, une instance de As est désagrégée et répartie dans m instances de E1. Si n = 1, desagreger(e[As],e1[A11],...,em[A11]). Les domaines de valeurs de As, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut *Telephones* du type d'entités *CLIENT* est transformé en un type d'entités *TELEPHONE*.



Transformation d'un type d'entités (représentation des valeurs) à une série d'attributs (désagrégation)

D Passer de la représentation par valeur d'un attribut multivalué à la désagrégation d'un attribut atomique monovalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1.



S ∃ E,ER,E2 ∈ type-entites(S), R1,R2 ∈ type-associations(S) : (E,{A1,...,An}) ← TE-val-en-desagreger-Att({E2,ER},{R1,R2})

C T^-

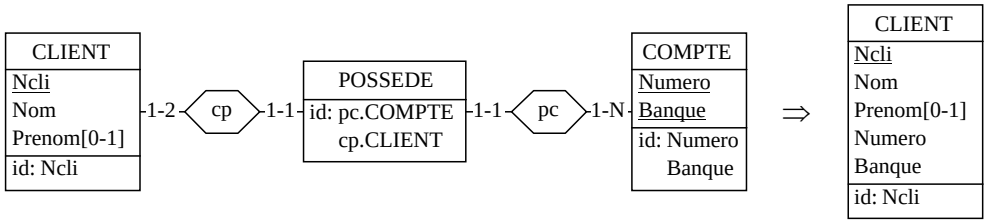
P
- E,E2,ER ∈ type-entites(S)
- A21,...,A2n ∈ attribut(E2)
- ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car | num | reel | date | bool
- R1,R2 ∈ type-associations(S)
- r1e, r1r ∈ role(R1)
- card(r1e) = [i-j] ∧ j > 1 ≥ i
- card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2)
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2)
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER)
- composant(ider) = {r1e,r2e1}

Q
- E2,ER ∉ type-entites(S)
- A21,...,A2n ∉ attribut(E2)
- R1,R2 ∉ type-associations(S)
- r1e, r1r ∉ role(R1)
- r2r, r2e1 ∉ role(R2)
- ider ∉ identifiant(R)
- ide2 ∉ identifiant(E1)
- A1,...,An ∈ attribut(E) ∧ nom(A1) = na1 ∧ ... ∧ nom(An) = nan
- ∀ k ∈ [1..n] : card(Ak) = [i-1] ∧ type(Ak) = car | num | reel | date | bool

I ∀ e ∈ instance(E), ∃! e2 ∈ instance(E2), er ∈ instance(ER), r1 ∈ instance(R1), r2 ∈ instance(R2) : r1[r1e,r1r] = (e,er) ∧ r1[r2r,r2e1] = (er,e2) ∧ e[A1] = e2[A21] ∧ ... ∧ e[An] = e2[A2n]

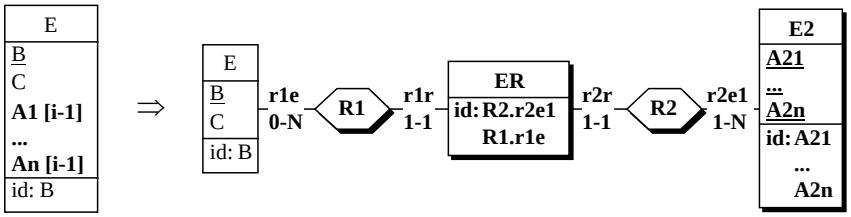
La première instance de E2 est mise dans A1,...,An. Les autres instances de E2 sont perdues. Si n = 1, desagreger(e2[A21],e[A1],...,e[An]). Les domaines de valeurs de A21, ..., A2n, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *POSSEDE* est transformé en un type d'associations *possede* et le type d'entités *COMPTE* est transformé en deux attributs *Numero* et *Banque* du type d'entités *CLIENT*.



Transformation d'une série d'attributs (désagrégation) à un type d'entités (représentation des valeurs)

D Passer de la désagrégation d'un attribut décomposable monovalué à une représentation par valeur d'un attribut multivalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1.



S ∃ E ∈ type-entites(S), A1,...,An ∈ attribut(E) : ({E2,ER},{R1,R2}) ← desagreger-Att-en-TE-val(E,{A1,...,An})

C T^+

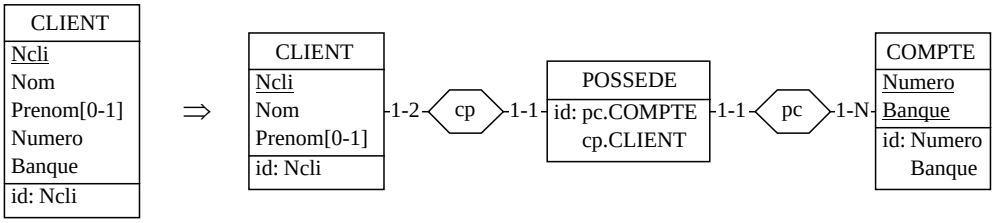
P
- E ∈ type-entites(S)
- A1,...,An ∈ attribut(E)
- ∀ k ∈ [1..n] : card(Ak) = [i-1] ∧ type(Ak) = car | num | reel | date | bool

Q
- A1,...,An ∉ attribut(E)
- E2,ER ∈ type-entites(S) ∧ nom(E2) = ne2 ∧ nom(ER) = ner
- A21,...,A2n ∈ attribut(E2) ∧ nom(A21) = na21 ∧ ... ∧ nom(A2n) = na2n
- ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car | num | reel | date | bool
- R1,R2 ∈ type-associations(S) ∧ nom(R1) = nr1 ∧ nom(R2) = nr2
- r1e, r1r ∈ role(R1) ∧ nom(r1e) = nr1e ∧ nom(r1r) = nr1r
- card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2) ∧ nom(r2r) = nr2r ∧ nom(r2e1) = nr2e1
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2) ∧ nom(ide2) = nide2
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER) ∧ nom(ider) = nider
- composant(ider) = {r1e,r2e1}

I ∀ e ∈ instance(E), ∃! e2 ∈ instance(e2), er ∈ instance(ER), r1 ∈ instance(R1), r2 ∈ instance(R2) : r1[r1e,r1r] = (e,er) ∧ r2[r2r,r2e1] = (er,e2) ∧ e2[A21] = e[A1] ∧ ... ∧ e2[A21] = e[A1]

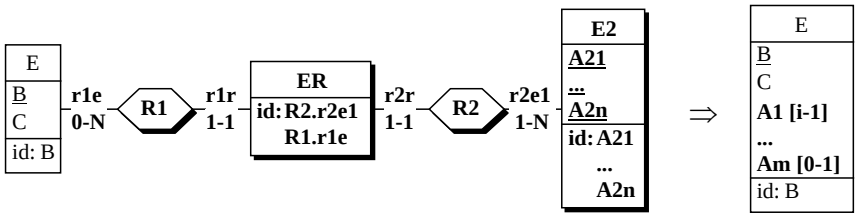
Pour chaque instance de E, les instances de A1,...,An sont mises dans l'instance de E2 à condition qu'il n'existe pas encore d'instances de E2 ayant les mêmes valeurs. Si n = 1, agreger(e[A1],...,e[An],e2[A21]). Les domaines de valeurs de A21, ..., A2n, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Numero* et *Banque* du type d'entités *CLIENT* sont transformés en un type d'entités *COMPTE* et le type d'associations *possede* est transformé en un type d'entités *POSSEDE*.



Transformation d'un type d'entités (représentation des valeurs) en une série d'attributs (instanciation)

D Passer de la représentation par valeur d'un attribut multivalué à l'instanciation d'un attribut atomique multivalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1.



S $\exists$ E,ER,E2 $\in$ type-entites(S), R1,R2 $\in$ type-associations(S) : (E,{A1,...,Am}) $\leftarrow$ TE-val-en-Serie-Att({E2,ER},{R1,R2})

C T^{-}

P
- E,E2,ER $\in$ type-entites(S)
- A21,...,A2n $\in$ attribut(E2)
- $\forall$ k $\in$ [1..n] : card(A2k) = [1-1] $\wedge$ type(A2k) = car | num | reel | date | bool
- R1,R2 $\in$ type-associations(S)
- r1e, r1r $\in$ role(R1)
- card(r1e) = [i-j] $\wedge$ j > 1 $\geq$ i $\wedge$ card(r1r) = [1-1]
- r2r, r2e1 $\in$ role(R2)
- card(r2r) = [1-1] $\wedge$ card(r2e1) = [1-N]
- ide2 $\in$ identifiant(E2)
- composant(ide2) = {r2r,A21,...,A2n}
- ider $\in$ identifiant(ER)
- composant(ider) = {r1e,r2e1}

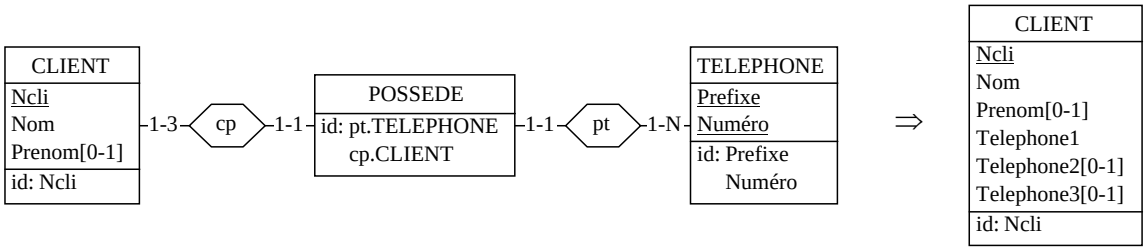
Q
- E2,ER $\notin$ type-entites(S)
- A21,...,A2n $\notin$ attribut(E2)
- R1,R2 $\notin$ type-associations(S)
- r1e, r1r $\notin$ role(R1) $\wedge$ r2r, r2e1 $\notin$ role(R2)
- ider $\notin$ identifiant(R) $\wedge$ ide2 $\notin$ identifiant(E1)
- A1,...,Am $\in$ attribut(E) $\wedge$ nom(A1) = na1 $\wedge$ ... $\wedge$ nom(Am) = nam
- card(A1) = [i-1] $\wedge$ ($\forall$ k $\in$ [2..m] : card(Ak) = [0-1])

– $\forall$ k $\in$ [1..n] : type(Ak) = car | num | reel | date | bool

I $\forall$ e $\in$ instance(E), e2i $\in$ instance(E2), eri $\in$ instance(ER), r1i $\in$ instance(R1), r2i $\in$ instance(R2), i $\in$ [1..m] : r1i[r1e,r1r] = (e,eri) $\wedge$ r2i[r2r,r2e1] = (eri,e2i) $\wedge$ agreger(e2i[A21],...,e2i[A2n],e[Ai])

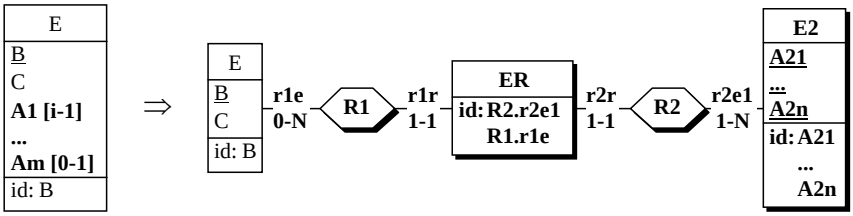
Les instances de E2 reliées à une instance de E via ER sont agrégées dans les instances de A1, ..., Am. Si n = 1, on a e[Ai] = e2i[A21]. Les domaines de valeurs de A21, ..., A2n, A1, ..., Am sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *POSSEDE* est transformé en un type d'associations *possede* et le type d'entités *TELEPHONE* est transformé en trois attributs *Telephone1*, *Telephone2* et *Telephone3* du type d'entités *CLIENT*.



Transformation d'une série d'attributs (instanciation) en un type d'entités (représentation des valeurs)

D Passer de l'instanciation d'un attribut atomique multivalué à une représentation par valeur d'un attribut multivalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1.



S $\exists$ E $\in$ type-entites(S), A1,...,Am $\in$ attribut(E) : ({E2,ER},{R1,R2}) $\leftarrow$ Serie-Att-en-TE-val(E,{A1,...,Am})

C T^+

P – E $\in$ type-entites(S)

– A1,...,Am $\in$ attribut(E)

– card(A1) = [i-1] $\wedge$ ($\forall$ k $\in$ [2..m] : card(Ak) = [0-1])

– $\forall$ k $\in$ [1..m] : type(Ak) = car | num | reel | date | bool

Q – A1,...,Am $\notin$ attribut(E)

– E2,ER $\in$ type-entites(S) $\wedge$ nom(E2) = ne2 $\wedge$ nom(ER) = ner

– A21,...,A2n $\in$ attribut(E2) $\wedge$ nom(A21) = na21 $\wedge$ ... $\wedge$ nom(A2n) = na2n

– $\forall$ k $\in$ [1..n] : card(A2k) = [1-1] $\wedge$ type(A2k) = car | num | reel | date | bool

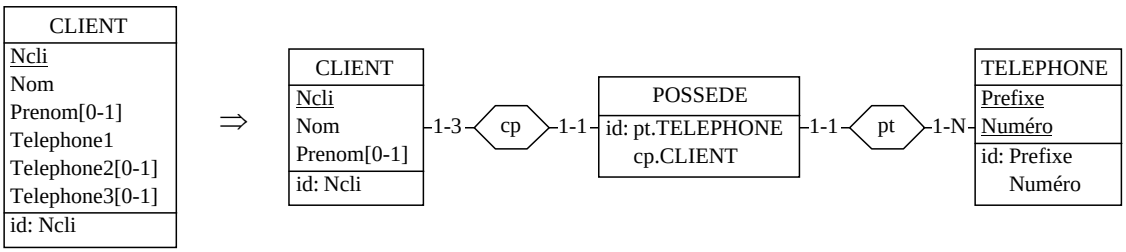
– R1,R2 $\in$ type-associations(S) $\wedge$ nom(R1) = nr1 $\wedge$ nom(R2) = nr2

- r1e, r1r ∈ role(R1) ∧ nom(r1e) = nr1e ∧ nom(r1r) = nr1r
- card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2) ∧ nom(r2r) = nr2r ∧ nom(r2e1) = nr2e1
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2) ∧ nom(ide2) = nide2
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER) ∧ nom(ider) = nider
- composant(ider) = {r1e,r2e1}

I ∀ e ∈ instance(E), e2i ∈ instance(E2), eri ∈ instance(ER), r1i ∈ instance(R1), r2i ∈ instance(R2), i ∈ [1..m] : r1i[r1e,r1r] = (e,eri) ∧ r2i[r2r,r2e1] = (eri,e2i) ∧ desagreger(e[Ai],e2i[A21],...,e2i[A2n])

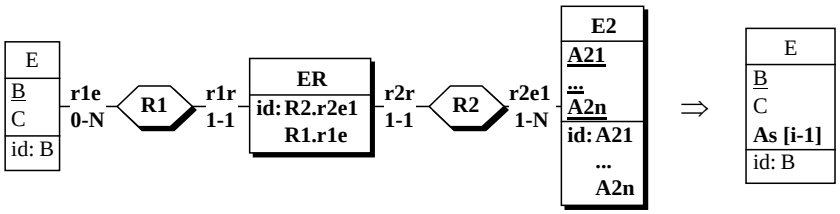
Chaque instance de A1,...,Am est désagrégée dans une instance de E2 à condition qu'il n'existe pas encore une instance de E2 ayant les mêmes valeurs. Si n = 1, on a e2i[A21] = e[Ai]. Les domaines de valeurs de A21, ..., A2n, A1, ..., Am sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Telephone1*, *Telephone2* et *Telephone3* du type d'entités *CLIENT* sont transformés en un type d'entités *TELEPHONE* et le type d'associations *possede* est transformé en un type d'entités *POSSEDE*.



Transformation d'un type d'entités (représentation des valeurs) en un attribut concaténé

D Passer de la représentation par valeur d'un attribut multivalué à la concaténation d'un attribut atomique multivalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1.



S ∃ E,ER,E2 ∈ type-entites(S), R1,R2 ∈ type-associations(S) : (E,As) ← TE-val-en-Att-concat({E2,ER},{R1,R2})

C T^-

P
- E,E2,ER ∈ type-entites(S)
- A21,...,A2n ∈ attribut(E2)

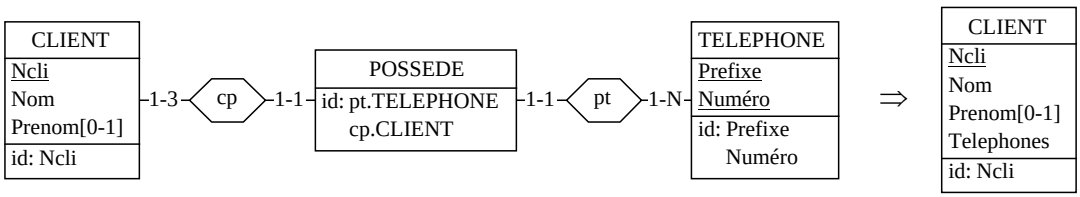
– $\forall$ k $\in$ [1..n] : card(A2k) = [1-1] $\wedge$ type(A2k) = car
– R1,R2 $\in$ type-associations(S)
– r1e, r1r $\in$ role(R1)
– card(r1e) = [i-j] $\wedge$ j > 1 $\geq$ i $\wedge$ card(r1r) = [1-1]
– r2r, r2e1 $\in$ role(R2)
– card(r2r) = [1-1] $\wedge$ card(r2e1) = [1-N]
– ide2 $\in$ identifiant(E2)
– composant(ide2) = {r2r,A21,...,A2n}
– ider $\in$ identifiant(ER)
– composant(ider) = {r1e,r2e1}

Q
– E2,ER $\notin$ type-entites(S)
– A21,...,A2n $\notin$ attribut(E2)
– R1,R2 $\notin$ type-associations(S)
– r1e, r1r $\notin$ role(R1) $\wedge$ r2r, r2e1 $\notin$ role(R2)
– ider $\notin$ identifiant(R) $\wedge$ ide2 $\notin$ identifiant(E1)
– As $\in$ attribut(E) $\wedge$ nom(As) = nas
– card(As) = [i-1]
– type(As) = car

I $\forall$ e $\in$ instance(E), e2i $\in$ instance(E2), eri $\in$ instance(ER), r1i $\in$ instance(R1), r2i $\in$ instance(R2), i $\in$ [1..m] : agreger(e2i[A21],...,e2i[A2n],e[As])

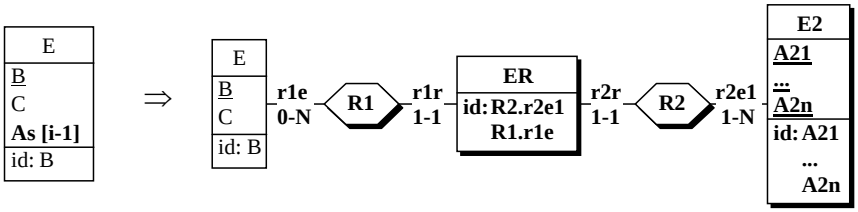
Les instances de E2 reliées à une instance de E via ER sont agrégées dans l'instance de As. Si n = 1, on a : agreger(e21[A21],...,e2m[A21],e[As]). Les domaines de valeurs de As, A21, ..., A2n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *POSSEDE* est transformé en un type d'associations *possede* et le type d'entités *TELEPHONE* est transformé en un attribut *Telephones* du type d'entités *CLIENT*.



Transformation d'un attribut concaténé en un type d'entités (représentation des valeurs)

D Passer de la concaténation d'un attribut atomique multivalué à une représentation par valeur d'un attribut multivalué. Dans le cas d'une représentation par valeur d'un attribut atomique multivalué, n est égal à 1.



S ∃ E ∈ type-entites(S), As ∈ attribut(E) : ({E2,ER},{R1,R2}) ← Att-concat-en-TE-val(E,As)

C T^+

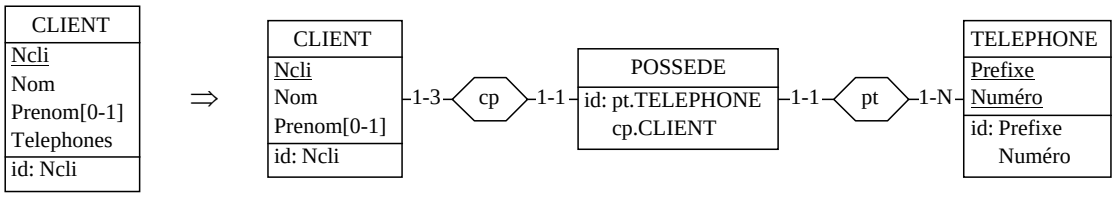
P
- E ∈ type-entites(S)
- As ∈ attribut(E)
- card(As) = [i-1]
- type(As) = car

Q
- As ∉ attribut(E)
- E2,ER ∈ type-entites(S) ∧ nom(E2) = ne2 ∧ nom(ER) = ner
- A21,...,A2n ∈ attribut(E2) ∧ nom(A21) = na21 ∧ ... ∧ nom(A2n) = na2n
- ∀ k ∈ [1..n] : card(A2k) = [1-1] ∧ type(A2k) = car
- R1,R2 ∈ type-associations(S) ∧ nom(R1) = nr1 ∧ nom(R2) = nr2
- r1e, r1r ∈ role(R1) ∧ nom(r1e) = nr1e ∧ nom(r1r) = nr1r
- card(r1e) = [i-j] ∧ j > 1 ≥ i ∧ card(r1r) = [1-1]
- r2r, r2e1 ∈ role(R2) ∧ nom(r2r) = nr2r ∧ nom(r2e1) = nr2e1
- card(r2r) = [1-1] ∧ card(r2e1) = [1-N]
- ide2 ∈ identifiant(E2) ∧ nom(ide2) = nide2
- composant(ide2) = {r2r,A21,...,A2n}
- ider ∈ identifiant(ER) ∧ nom(ider) = nider
- composant(ider) = {r1e,r2e1}

I ∀ e ∈ instance(E), e2i ∈ instance(e2), eri ∈ instance(ER), r1i ∈ instance(R1), r2i ∈ instance(R2), i ∈ [1..m] : r1i[r1e,r1r] = (e,eri) ∧ r2i[r2r,r2e1] = (eri,e2i) ∧ desagreger(e[As],e2i[A21],...,e2i[A2n])

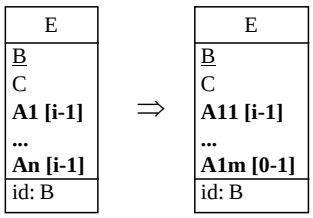
Chaque instance de As est désagrégée dans plusieurs instances de E2 à condition qu'il n'existe pas encore une instance de E2 ayant les mêmes valeurs. Si n = 1, on a : desagreger(e[As],e21[A21],...,e2m[A21]). Les domaines de valeurs de As, A21, ..., A2n sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Le type d'entités *POSSEDE* est transformé en un type d'associations *possede* et le type d'entités *TELEPHONE* est transformé en un attribut *Telephones* du type d'entités *CLIENT*.



Transformation d'une série d'attributs (désagrégation) en une série d'attributs (instanciation)

D Passer de la désagrégation d'un attribut décomposable monovalué à l'instanciation d'un attribut atomique multivalué.



S ∃ E ∈ type-entites(S), A1,...,An ∈ attribut(E) : (E,{A11,...,A1m}) ← desagreger-Att-en-Serie-Att(E,{A1,...,An})

C Indéterminé

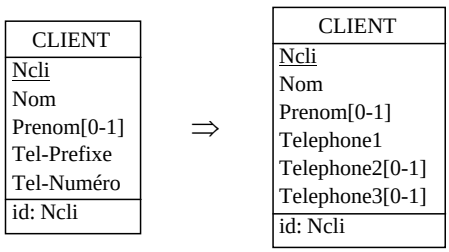
P
- E ∈ type-entites(S)
- A1,...,An ∈ attribut(E)
- ∀ k ∈ [1..n] : card(Ak) = [i-1] ∧ type(Ak) = car | num | reel | date | bool

Q
- A1,...,An ∉ attribut(E)
- A11,...,A1m ∈ attribut(E) ∧ nom(A11) = na11 ∧ ... ∧ nom(A1m) = na1m
- card(A11) = [i-1] ∧ (∀ k ∈ [2..m] : card(A1k) = [0-1])
- ∀ k ∈ [1..m] : type(A1k) = car | num | reel | date | bool

I ∀ e ∈ instance(E) : agreger(e[A1],...,e[An],e[A11])

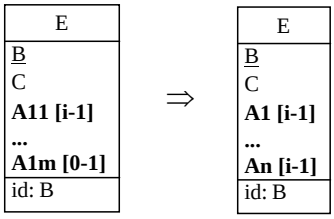
Pour chaque instance de E, les instances de A1,...,An sont agrégées dans A11. Les domaines de valeurs de A11, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Tel-Prefixe* et *Tel-Numero* du type d'entités *CLIENT* sont transformés en trois attributs *Telephone1*, *Telephone2* et *Telephone3*.



Transformation d'une série d'attributs (instanciation) en une série d'attributs (désagrégation)

D Passer de l'instanciation d'un attribut atomique multivalué à la désagrégation d'un attribut décomposable monovalué.



S ∃ E ∈ type-entites(S), A11,...,A1m ∈ attribut(E) : (E,{A1,...,An}) ← Serie-Att-en-desagreger-Att(E,{A11,...,A1m})

C Indéterminé

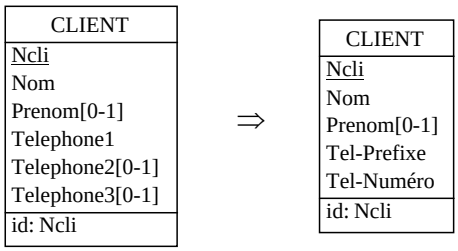
P
- E ∈ type-entites(S)
- A11,...,A1m ∈ attribut(E)
- card(A11) = [i-1] ∧ (∀ k ∈ [2..m] : card(A1k) = [0-1])
- ∀ k ∈ [1..m] : type(A1k) = car | num | reel | date | bool

Q
- A11,...,A1m ∉ attribut(E)
- A1,...,An ∈ attribut(E) ∧ nom(A1) = na1 ∧ ... ∧ nom(An) = nan
- ∀ k ∈ [1..n] : card(Ak) = [i-1] ∧ type(Ak) = car | num | reel | date | bool

I ∀ e ∈ instance(E) : desagreger(e[A11],e[A1],...,e[An])

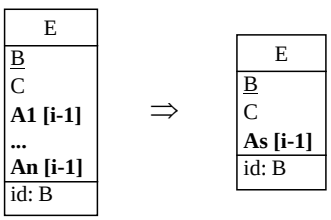
Pour chaque instance de E, l'instance de A11 est désagrégée dans A1,...,An. Les instances de A12, ..., A1m sont perdues. Les domaines de valeurs de A11, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Telephone1*, *Telephone2* et *Telephone3* du type d'entités *CLIENT* sont transformés en deux attributs *Tel-Prefixe* et *Tel-Numero*.



Transformation d'une série d'attributs (désagrégation) en un attribut concaténé

D Passer de la désagrégation d'un attribut décomposable monovalué à la concaténation d'un attribut atomique multivalué.



S ∃ E ∈ type-entites(S), A1,...,An ∈ attribut(E) : (E,As) ← desagreger-Att-en-Att-concat(E,{A1,...,An})

C Indéterminé

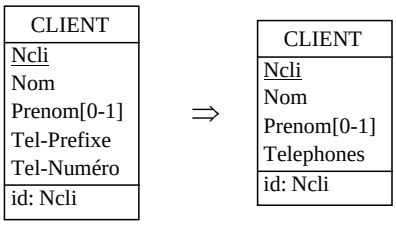
P
- E ∈ type-entites(S)
- A1,...,An ∈ attribut(E)
- ∀ k ∈ [1..n] : card(Ak) = [i-1] ∧ type(Ak) = car

Q
- A1,...,An ∉ attribut(E)
- As ∈ attribut(E) ∧ nom(As) = nas
- card(As) = [i-1]
- type(As) = car

I ∀ e ∈ instance(E) : agreger(A1,...,An,As)

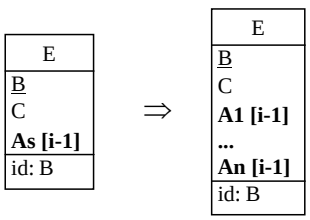
Pour chaque instance de E, les instances de A1,...,An sont agrégées dans As. Les domaines de valeurs de As, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Tel-Prefixe* et *Tel-Numero* du type d'entités *CLIENT* sont transformés en un attribut *Telephones*.



Transformation d'un attribut concaténé à une série d'attributs (désagrégation)

D Passer de la concaténation d'un attribut atomique multivalué à la désagrégation d'un attribut décomposable monovalué.



S ∃ E ∈ type-entites(S), As ∈ attribut(E) : (E,{A1,...,An}) ← Att-concat-en-desagreger-Att(E,As)

C Indéterminé

P
- E ∈ type-entites(S)
- As ∈ attribut(E)
- card(As) = [i-1]
- type(As) = car

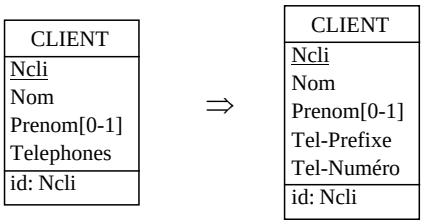
Q
- As ∉ attribut(E)
- A1,...,An ∈ attribut(E) ∧ nom(A1) = na1 ∧ ... ∧ nom(An) = nan

– $\forall$ k $\in$ [1..n] : card(Ak) = [i-1] $\wedge$ type(Ak) = car

I $\forall$ e $\in$ instance(E) : desagreger(e[As],e[A1],...,e[An])

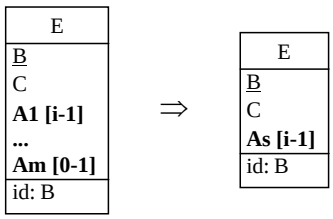
Pour chaque instance de E, l'instance de As est répartie dans A1,...,An. Une partie de la fin de la valeur de l'instance de As peut être perdue. Les domaines de valeurs de As, A1, ..., An sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E L'attribut *Telephones* du type d'entités *CLIENT* est transformé en deux attributs *Tel-Prefixe* et *Tel-Numero*.



Transformation d'une série d'attributs (instanciation) en un attribut concaténé

D Passer de l'instanciation à la concaténation d'un attribut atomique multivalué.



S $\exists$ E $\in$ type-entites(S), A1, ..., Am $\in$ attribut(E) : (E,As) $\leftarrow$ Serie-Att-en-Att-concat(E,{A1,...,Am})

C $T^{=}$

P – E $\in$ type-entites(S)

– A11,...,A1m $\in$ attribut(E)

– card(A11) = [i-1] $\wedge$ ($\forall$ k $\in$ [2..m] : card(A1k) = [0-1])

– $\forall$ k $\in$ [1..m] : type(A1k) = car

Q – A11,...,A1m $\notin$ attribut(E)

– As $\in$ attribut(E) $\wedge$ nom(As) = nas

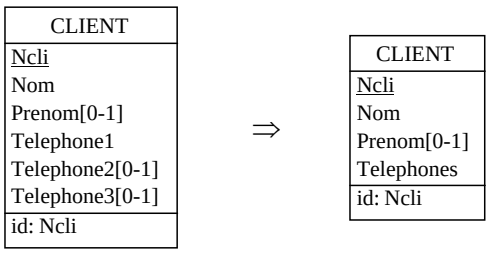
– card(As) = [i-1]

– type(As) = car

I $\forall$ e $\in$ instance(E) : agreger(e[A1],...,e[Am],e[As])

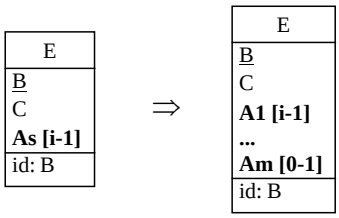
Pour chaque instance de E, les instances de A1,...,An sont agrégées dans As. Les domaines de valeurs de As, A1, ..., Am sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

E Les attributs *Telephone1*, *Telephone2* et *Telphone3* du type d'entités *CLIENT* sont transformés en un attribut *Telephones*.



Transformation d'un attribut concaténé en une série d'attributs (instanciation)

D Passer de la concaténation à l'instanciation d'un attribut atomique multivalué.



S $\exists$ E $\in$ type-entites(S), As $\in$ attribut(E) : (E,{A1,...,Am}) $\leftarrow$ Att-concat-en-Serie-Att(E,As)

C $T^{=}$

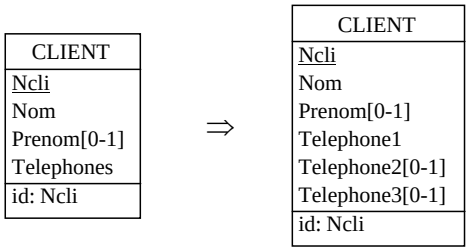
P
- E $\in$ type-entites(S)
- As $\in$ attribut(E)
- card(As) = [i-1]
- type(As) = car

Q
- As $\notin$ attribut(E)
- A1,...,Am $\in$ attribut(E) $\wedge$ nom(A1) = na1 $\wedge$ ... $\wedge$ nom(Am) = nam
- card(A1) = [i-1] $\wedge$ ($\forall$ k $\in$ [2..m] : card(Ak) = [0-1])
- $\forall$ k $\in$ [1..m] : type(Ak) = car

I $\forall$ e $\in$ instance(E) : desagreger(e[As],e[A1],...,e[Am])

Pour chaque instance de E, l'instance de As est répartie dans les instances de A1,...,Am. Les domaines de valeurs de As, A1, ..., Am sont compatibles de manière à transférer les instances d'une structure à l'autre sans perte de données.

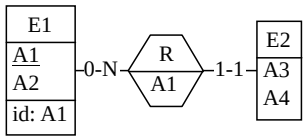
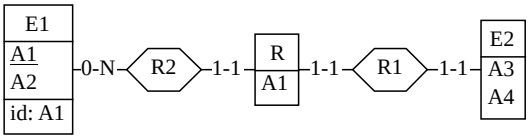
E L'attribut *Telephones* du type d'entités *CLIENT* est transformé en trois attributs *Telephone1*, *Telephone2* et *Telphone3*.



C.4.3.2 Modifications relatives aux types d'associations

La seule transformation possible sur un type d'associations complexe est la transformation en un type d'entités. Les tableaux de ce point regroupe le passage du modèle E/A riche au modèle E/A de base pour toutes les modifications étudiées au point C.4.2.2. La création est illustrée par des exemples qui seront utilisés par les autres modifications.

a) Création

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Création du type d'associations R. 	Création du type d'entités R (page 38). Création des types d'associations R1 et R2 (page 44). Création de l'attribut A1 (page 39). 

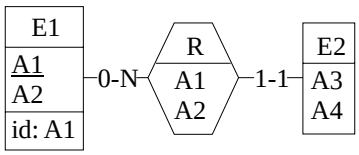
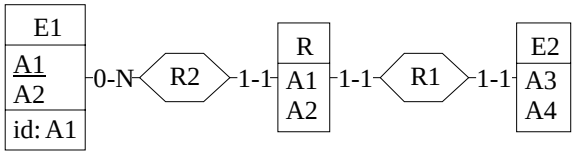
b) Destruction

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Destruction du type d'associations R.	Destruction du type d'entités R (page 38). Destruction des type d'associations R1 et R2 (page 45).

c) Renommage

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Renommage du type d'associations R.	Renommage du type d'entités R (page 39). Renommage des types d'associations R1 et R2 (page 45).

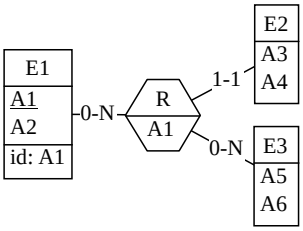
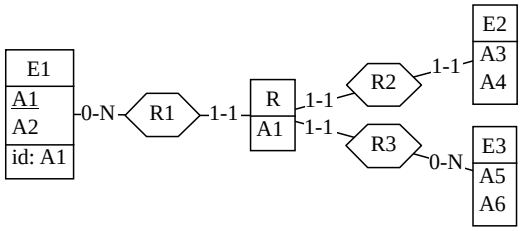
d) Ajout d'un attribut

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Ajout de l'attribut A2 au type d'associations R. 	Création de l'attribut A2 dans le type d'entités R (page 39). 

e) Retrait d'un attribut

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Destruction de l'attribut A2 du type d'associations R.	Destruction de l'attribut A2 dans le type d'entités R (page 40).

f) Création d'un rôle

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Création du rôle R.E3 au type d'associations R. 	Création du type d'associations R3 (page 44). 

g) Destruction d'un rôle

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Destruction du rôle R.E2 du type d'associations R.	Destruction du type d'associations R2 (page 45).

h) Renommage d'un rôle

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Renommage du rôle R.E2 du type d'associations R.	Renommage du type d'associations R2 (page 45).

i) Modification des cardinalités d'un rôle

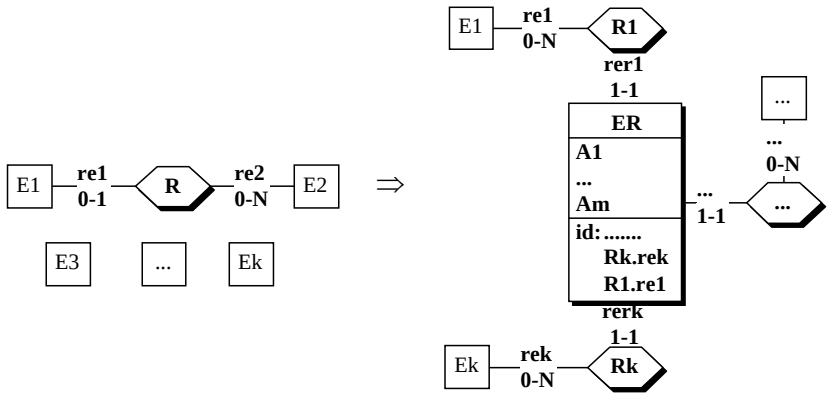
Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Modifications des cardinalités du rôle R.E2 du type d'associations R.	Modifications des cardinalités du rôle R2.E2 (page 46).

j) Changement de transformation

Il faut encore traiter le changement de structure lorsqu'on passe d'un type d'associations fonctionnel en un type d'associations complexe et inversement. Comme un seul type de transformation existe pour les types d'associations complexes, deux cas sont possibles.

Transformation d'un type d'associations fonctionnel en un type d'entités

D Passer d'un type d'associations fonctionnel à la transformation en type d'entités. Cette transformation intervient suite à certaines modifications au niveau du schéma E/A riche dont l'ajout de rôles, l'ajout d'attributs ou la modification d'une cardinalité maximum d'un rôle.



S ∃ R ∈ type-associations(S) : (ER,{R1,...,Rk}) ← TA-en-TE(R)

C T^+

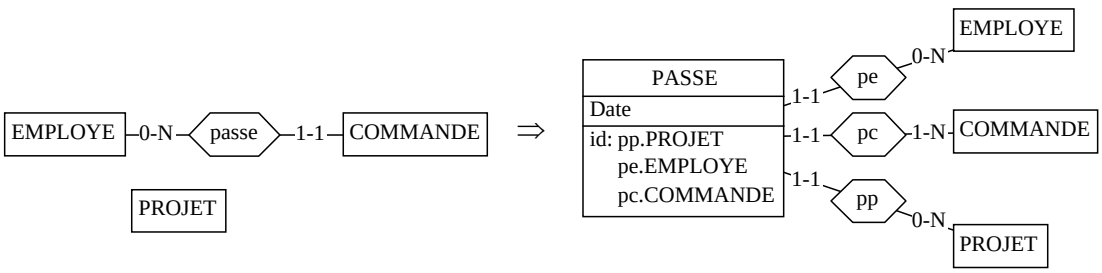
P
- E1, ..., Ek ∈ type-entites(S)
- R ∈ type-association(S)
- re1, re2 ∈ role(R)
- (card(re1) = [0-1] ∧ card(re2) = [0-1]) ∨ (card(re1) = [0-1] ∧ card(re2) = [1-1]) ∨ (card(re1) = [1-1] ∧ card(re2) = [1-1]) ∨ (card(re1) = [0-1] ∧ card(re2) = [0-N]) ∨ (card(re1) = [1-1] ∧ card(re2) = [0-N]) ∨ (card(re1) = [0-1] ∧ card(re2) = [1-N]) ∨ (card(re1) = [1-1] ∧ card(re2) = [1-N])

Q
- R ∉ type-associations(S)
- ER ∈ type-entites(S) ∧ nom(ER) = ner
- A1, ..., Am ∈ attribut(ER) ∧ nom(A1) = na1 ∧ ... ∧ nom(Am) = nam
- R1, ..., Rk ∈ type-associations(S) ∧ nom(R1) = nr1 ∧ ... ∧ nom(Rk) = nrk
- ∀ i ∈ [1..k] : rei, reri ∈ role(Ri) ∧ nom(rei) = nrei ∧ nom(reri) = nreri ∧ card(rei) = [mini-maxi] ∧ card(reri) = 1-1
- ider ∈ identifiant(ER) ∧ nom(ider) = nider
- composant(ider) = {re1, ..., rek}

I ∀ r ∈ instance(r), ∃! er ∈ instance(ER), ri ∈ instance(Ri), ei ∈ instance(Ei), i ∈ [1..k] : ri[rei,reri] = (ei,er)

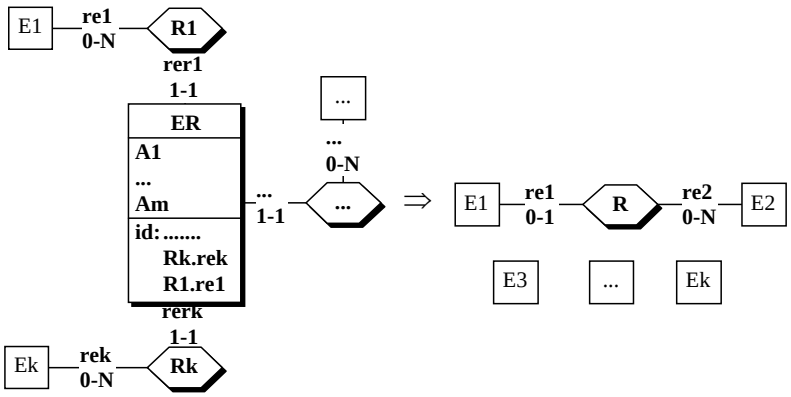
Pour chaque instance de R, on crée une instance de ER ainsi que les instances R1, ..., Rk. Il faut donner des valeurs aux attributs de A1,...,An de ER s'ils sont obligatoires.

E Un type d'associations *passe* est transformé en un type d'entités *PASSE*.



Transformation d'un type d'entités en un type d'associations fonctionnel

D Passer de la transformation d'un type d'associations complexe en un type d'associations fonctionnel. Cette transformation intervient suite à certaines modifications au niveau du schéma E/A riche comme la destruction de rôles, d'attributs ou la modification d'une cardinalité maximum d'un rôle.



S $\exists$ ER $\in$ type-entites(S) : R $\leftarrow$ TE-en-TA(ER)

C T^-

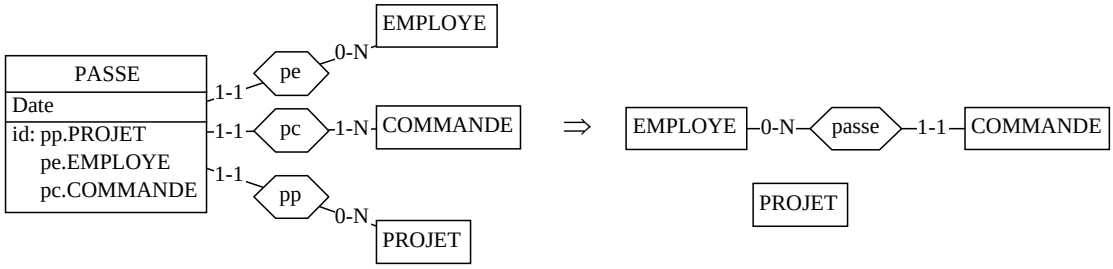
P
- E1, ..., Ek $\in$ type-entites(S)
- ER $\in$ type-entites(S)
- A1, ..., Am $\in$ attribut(ER)
- R1, ..., Rk $\in$ type-associations(S)
- $\forall$ i $\in$ [1..k] : rei, reri $\in$ role(Ri) $\wedge$ card(rei) = [mini-maxi] $\wedge$ card(reri) = 1-1
- ider $\in$ identifiant(ER)
- composant(ider) = {re1, ..., rek}

Q
- ER $\notin$ type-entites(S)
- A1, ..., Am $\notin$ attribut(ER)
- R1, ..., Rk $\notin$ type-associations(S)
- $\forall$ i $\in$ [1..k] : rei, reri $\notin$ role(Ri)
- ider $\notin$ identifiant(ER)
- R $\in$ type-association(S) $\wedge$ nom(R) = nr
- re1, re2 $\in$ role(R) $\wedge$ nom(re1) = nre1 $\wedge$ nom(re2) = nre2
- (card(re1) = [0-1] $\wedge$ card(re2) = [0-1]) $\vee$ (card(re1) = [0-1] $\wedge$ card(re2) = [1-1]) $\vee$ (card(re1) = [1-1] $\wedge$ card(re2) = [1-1]) $\vee$ (card(re1) = [0-1] $\wedge$ card(re2) = [0-N]) $\vee$ (card(re1) = [1-1] $\wedge$ card(re2) = [0-N]) $\vee$ (card(re1) = [0-1] $\wedge$ card(re2) = [1-N]) $\vee$ (card(re1) = [1-1] $\wedge$ card(re2) = [1-N])

I $\forall$ er $\in$ instance(E1), r1 $\in$ instance(R1), r2 $\in$ instance(R2), $\exists!$ r $\in$ instance(R) : r[re1,re2] = (r1[re1],r2[re2])

Pour chaque instance de ER, on crée une instance de R à partir de la première instance de E1 reliée à ER par R1 et de la première instance de E2 reliée à ER par R2. Les autres instances (ou parties d'instances) de ER sont perdues.

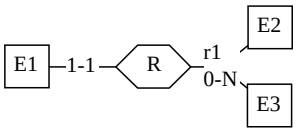
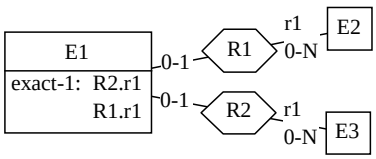
E Un type d'entités *PASSE* est transformé en un type d'associations *passe*.



C.4.3.3 Modifications relatives aux rôles

La seule transformation possible sur un rôle multi-TE est la transformation en types d'associations. Les tableaux de ce point regroupe le passage du modèle E/A riche au modèle E/A de base pour toutes les modifications étudiées au point C.4.2.3. La création est illustrée par des exemples qui seront utilisés par les autres modifications.

a) Création

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Création du rôle multi-TE r1. 	Création des types d'associations R1 et R2 (page 44). Création d'une contrainte exactement-un dans E1 (page 100). 

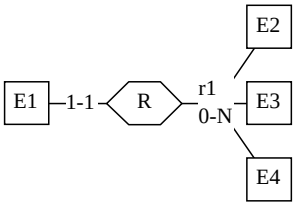
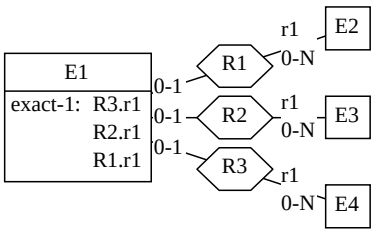
b) Destruction

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Destruction du rôle multi-TE r1.	Destruction des types d'associations R1 et R2 (page 45). Destruction d'une contrainte exactement-un dans E1 (page 101).

c) Renommage

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Renommage du rôle multi-TE r1.	Renommage des types d'associations R1 et R2 (page 45).

d) Ajout d'un type d'entités

Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Ajout du type d'entités E4 au rôle multi-TE r1. 	Création du type d'associations R3 (page 44). Ajout du rôle R3.r1 à la contrainte exactement-un de E1 (page 103). 

e) Retrait d'un type d'entités

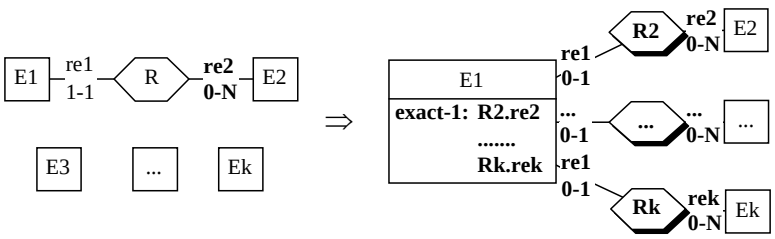
Modifications sur schéma E/A riche	Modifications sur schéma E/A de base
Retrait du type d'entités E4 du rôle multi-TE r1.	Destruction du type d'associations R3 (page 45). Retrait du rôle R3.r1 de la contrainte exactement-un de E1 (page 104).

f) Changement de transformation

En modifiant un rôle mono-TE en rôle multi-TE, on ajoute un nouveau type d'associations (plus éventuellement une contrainte exclusive ou exactement-un) dans le schéma E/A de base. Inversement, on détruit un type d'associations (et éventuellement une contrainte).

Transformation d'un rôle en plusieurs types d'associations

D Passer d'un rôle mono-TE à la transformation en types d'associations. Cette transformation intervient suite à l'ajout de types d'entités à un rôle au niveau du schéma E/A riche.



S $\exists$ R $\in$ type-associations(S), re2 $\in$ role(R) : (R2, ..., Rk) $\leftarrow$ Role-en-TA(R,re2)

C T^+

P
- E1, ..., Ek $\in$ type-entites(S)
- R $\in$ type-association(S)
- re1, re2 $\in$ role(R)
- card(re2) = [l-m]
- te-role(re1) = {E1} $\wedge$ te-role(re2) = {E2}

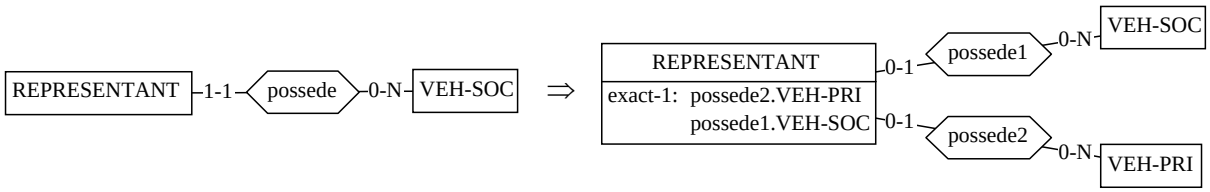
Q
- R $\notin$ type-associations(S)

- R2, ..., Rk ∈ type-associations(S) ∧ nom(R2) = nr2 ∧ ... ∧ nom(Rk) = nrk
- ∀ i ∈ [2..k] : re1, rei ∈ role(Ri) ∧ nom(re1) = nre1 ∧ nom(rei) = nrei
- card(re1) = [0-j] ∧ (∀ i ∈ [2..k] : card(rei) = [l-m])
- te-role(re1) = {E1} ∧ (∀ i ∈ [2..k] : te-role(rei) = {Ei})
- gr ∈ groupe(E1) ∧ nom(gr) = ngr
- fonction(gr) = {exactement-un,au-moins-un}
- composant(gr) = {re2, ..., rek}

I ∀ r ∈ instance(R), ∃! r2 ∈ instance(R2) : r[re1,re2] = r2[re1,re2]

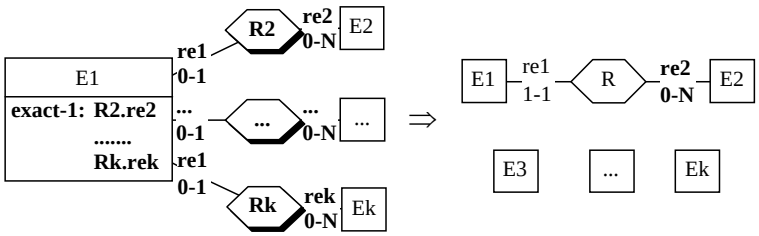
Les instances de R2 sont celles de R.

E Le rôle *possede.VEH-SOC* est transformé en deux types d'associations *possede1* et *possede2*.



Transformation de plusieurs types d'associations en un rôle

D Passer de la transformation en type d'associations d'un rôle multi-TE à un rôle mono-TE. Cette transformation intervient suite aux retraits de types d'entités dans un rôle au niveau du schéma E/A riche.



S ∃ R2, ..., Rk ∈ type-associations(S) : (R,re2) ← TA-en-Role(R2,...,Rk)

C T^-

P
- E1, ..., Ek ∈ type-entites(S)
- R2, ..., Rk ∈ type-associations(S)
- ∀ i ∈ [2..k] : re1, rei ∈ role(Ri)
- card(re1) = [0-j] ∧ (∀ i ∈ [2..k] : card(rei) = [l-m])
- te-role(re1) = {E1} ∧ (∀ i ∈ [2..k] : te-role(rei) = {Ei})
- gr ∈ groupe(E1)
- fonction(gr) = {exactement-un,au-moins-un}
- composant(gr) = {re2, ..., rek}

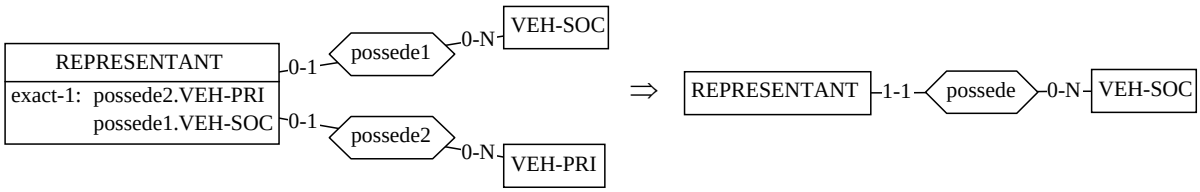
Q
- R2, ..., Rk ∉ type-associations(S)
- re1, rei ∉ role(Ri) ∧ 2 ≤ i ≤ k

- gr ∉ groupe(E1)
- R ∈ type-association(S) ∧ nom(R) = nr
- re1, re2 ∈ role(R) ∧ nom(re1) = nre1 ∧ nom(re2) = nre2
- card(re1) = [i-j] ∧ card(re2) = [l-m] ∧ 0 ≤ i ≤ 1
- te-role(re1) = {E1} ∧ te-role(re2) = {E2}

I ∀ r2 ∈ instance(R2), ∃! r ∈ instance(r) : r2[re1,re2] = r[re1,re2]

Les instances de R sont celles de R2. Les instances de R3, ..., RK sont perdues.

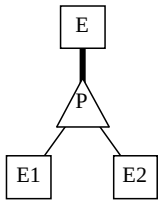
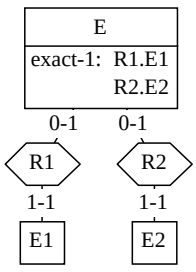
E Les deux types d'associations *possede1* et *possede2* sont transformés en un rôle *possede.VEH-SOC*.



C.4.3.4 Modifications relatives aux relations IS-A

La seule transformation possible sur une relation is-a est la transformation en un type d'associations. Les tableaux de ce point regroupe le passage du modèle E/A riche au modèle E/A de base pour toutes les modifications étudiées au point C.4.2.4. La création est illustrée par des exemples qui seront utilisés par les autres modifications.

a) Création

Modifications sur schémas E/A riche	Modifications sur schéma E/A de base
Création d'une relation is-a entre E et E2. 	Création du type d'associations R2 (page 44). Création de la contrainte exactement-un de E (page 100) ou ajout du rôle R2.E2 dans la contrainte exactement-un de E (page 103). 

b) Destruction

Modifications sur schémas E/A riche	Modifications sur schéma E/A de base
Destruction de la relation is-a entre E et E2.	Destruction du type d'associations R2 (page 45). Destruction de la contrainte exactement-un de E (page 101) ou retrait du rôle R2.E2 de la contrainte exactement-un de E (page 104).

C.4.4 Modifications des structures, des données et des programmes

Les scripts de conversion des données et structures de données ainsi que la modification des programmes sont étudiés pour les modifications engendrées par des changements de transformation dans le modèle E/A de base (point C.4.3). Les autres modifications ne sont pas étudiées car elles peuvent être ramenées à une suite de modifications simples déjà analysées dans les sections précédentes. Les scripts présentés sont décomposés en trois parties :

- la création des nouvelles structures,
- le transfert des instances des anciennes structures vers les nouvelles,
- la destruction des anciennes structures.

La première et la troisième parties font référence aux modifications déjà analysées précédemment. Seul le transfert des instances est détaillé.

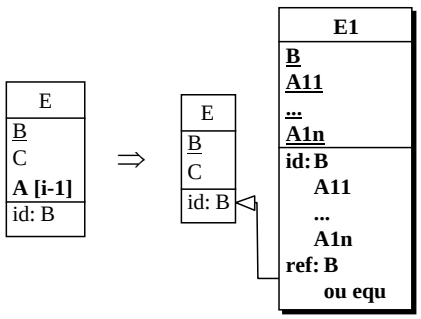
C.4.4.1 Changement de transformation sur les colonnes

Tous les changements de transformations du point C.4.3.1 k) sont analysés en termes de conversion de structures et de modifications de programmes.

a) Transformation d'une colonne en une table (représentation des instances)

Données

D Dépendance : les instances de A sont intégrées dans la table E1, cela peut poser des problèmes avec les types des colonnes A11,...,A1n.



I Si A est transféré dans plusieurs colonnes, ces instances doivent être compatibles avec les types des nouvelles colonnes (A11,...,A1n). Le concepteur doit évaluer cela en analysant les instances de A.

S Création de E1 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert dans une seule colonne (n = 1) :
INSERT INTO E1 SELECT B,A from E WHERE A IS NOT NULL;
-- Si transfert dans plusieurs colonnes (n > 1) :
INSERT INTO E1 SELECT B,fct1(A),...,fctn(A) from E WHERE A IS NOT NULL;
```

Destruction de E.A : voir destruction d'une colonne (page 77).

Programmes

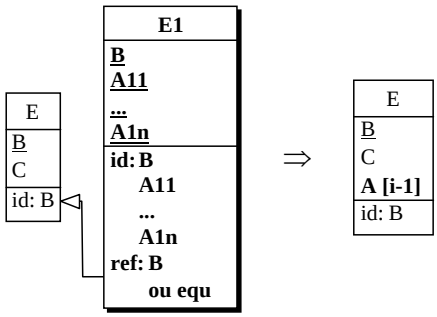
D Dépendance structurelle : pour accèder aux colonnes, il faut une jointure ou une requête imbriquée supplémentaire. Dans les interfaces, le champ de la colonne doit être remplacé par une liste de valeurs.

L Recherche sur la colonne E.A ainsi que toutes les variables qui en dépendent.

b) Transformation d'une table (représentation des instances) en une colonne

Données

D Dépendance : les instances de E1 sont transférées dans la colonnes A. Cela peut poser des problèmes de compatibilité entre les types de colonnes A11,...,A1n et A ainsi que des problème si une instance de E est référencée par plusieurs instances de E1.



I Il faut vérifier qu'une instance de E est référencée par au maximum une instance de E1.

```
SELECT B FROM E1 GROUP BY B HAVING COUNT(B) > 1;
```

Si cette requête donne un résultat, il faut une intervention du concepteur.

Les instances de A11,...,A1n doivent être compatibles avec le type de la colonne A. Le concepteur doit évaluer cela en analysant chaque instances de A11,...,A1n.

S Création de E.A : voir création d'une colonne (page 76).

Transfert des instances :

```
-- Si transfert dans une seule colonne (n = 1) :
UPDATE TABLE E SET A = (SELECT A1 FROM E1 WHERE E.B = E1.B);
-- Si transfert de plusieurs colonnes (n > 1) :
UPDATE TABLE E SET A = (SELECT fct(A11,...,A1n) FROM E1 WHERE E.B = E1.B);
```

Destruction de E1 : voir la destruction d'une table (page 75).

Programmes

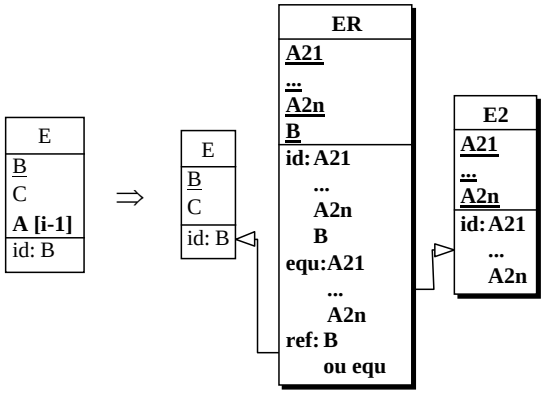
D Dépendance structurelle : les jointures ou les requêtes imbriquées ne sont plus nécessaires pour accéder aux informations. Dans les interfaces, la liste ou la fenêtre d'accès à la table E1 est remplacée par un champ.

L Recherche sur la table E1, ses colonnes ainsi que les variables qui en dépendent.

c) Transformation d'une colonne en plusieurs tables (représentation des valeurs)

Données

D Dépendance : Les instances de A sont intégrées dans la table E2, cela peut poser des problèmes au niveau des types des colonnes A21,...,A2n.



I Si A est transféré dans plusieurs colonnes (A21,...,A2n), ces instances doivent être compatibles avec les types des nouvelles colonnes. Le concepteur doit évaluer cela en analysant chaque instance de A.

S Création de ER et E2 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO E2 SELECT DISTINCT(A) from E;
INSERT INTO ER SELECT A,B from E WHERE A IS NOT NULL;
-- Si transfert vers plusiers colonnes (n > 1) :
INSERT INTO E2 SELECT DISTINCT fct1(A),...,fctn(A) from E WHERE A IS NOT NULL;
INSERT INTO ER SELECT fct1(A),...,fctn(A),B from E WHERE A IS NOT NULL;
```

Destruction de E.A : voir destruction d'une colonne (page 77).

Programmes

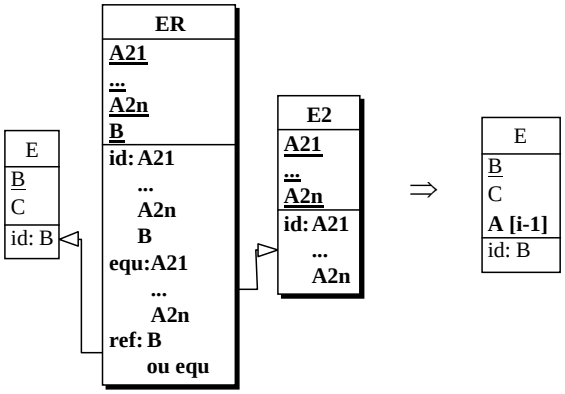
D Dépendance structurelle : dans les instructions de manipulation de données, il faut faire une jointure ou deux requêtes imbriquées supplémentaires pour accèder aux nouvelles colonnes. Dans les interfaces, le champ de la colonne va être remplacé par des listes ou des écrans de gestion supplémentaires.

L Recherche sur la colonne E.A ainsi que sur les variables qui en dépendent.

d) Transformation de plusieurs tables (représentation des valeurs) à une colonne

Données

D Dépendance : Les instances de ER sont transférées dans la colonne A de E. Cela pose des problèmes si une instance de E est référencée par plus qu'une instance de ER.



I Une instance de E ne peut être référencée que par une instance de ER au maximum.
```
SELECT B FROM ER GROUP BY B HAVING COUNT(B) > 1;
```
Si la requête donne un résultat, cela pose des problèmes. Il faut une intervention du concepteur.

S Création de E.A : voir création d'une colonne (page 76).

Transfert des instances de ER dans E.A:
```
-- Si transfert dans une seule colonne (n = 1) :
UPDATE TABLE E SET A = (SELECT A FROM ER WHERE E.B = ER.B);
-- Si transfert dans plusieurs colonnes (n > 1) :
UPDATE TABLE E SET A = (SELECT CONCAT(A1,...,An) FROM ER WHERE E.B = ER.B);
```
Destruction de E2 et ER: voir la destruction d'une table (page 75).

Programmes

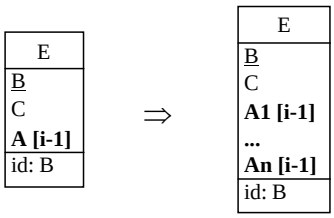
D Dépendance structurelle : les jointures ou les requêtes imbriquées ne sont plus nécessaires pour accèder aux données. Dans les interfaces, les fenêtres de gestion des tables ER et E2 peuvent être remplacées par un champ pour la colonne A.

L Recherche sur les tables E2 et ER ainsi que sur toutes leurs colonnes et les variables qui en dépendent.

e) Transformation d'une colonne en plusieurs colonnes (désagrégation)

Données

D Dépendance : les instances de A sont transférées dans les colonnes A1,...,An. Cela peut poser des problèmes avec les types et les longueurs des nouvelles colonnes.



I Les instances de A doivent être compatibles avec les types des nouvelles colonnes (A1,...,An). La longueur du domaine de A doit être inférieure ou égale à la somme des longueurs des domaines de A1,...,An. Le concepteur doit évaluer cela en analysant chaque instance de A.

S Création de E.A1, ..., E.An: voir la création d'une colonne (page 77).

Transfert des instances :

```
UPDATE TABLE E SET (A1,...,An) = fct1(A),...,fctn(A) WHERE A IS NOT NULL;
UPDATE TABLE E SET (A1,...,An) = NULL,...,NULL WHERE A IS NULL;
```

Destruction de E.A: voir la destruction d'une colonne (page 77).

Programmes

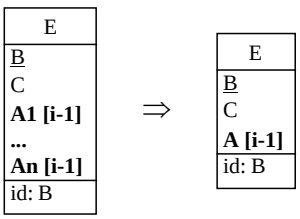
D Dépendance structurelle : dans les requêtes, il faut remplacer la colonne A par les colonnes A1,...,An. Dans les interfaces, il faut remplacer le champ A par les champs A1,...,An.

L Recherche sur la colonne E.A ainsi que sur toutes les variables qui en dépendent.

f) Transformation de plusieurs colonnes (désagrégation) à une colonne

Données

D Dépendance : les instances de A1,...,An sont transférées dans la colonne A de E. Cela pose des problèmes avec les longueurs et les types des instances de A1,...,An.



I Les instances de A1,...,An doivent être compatibles avec le type de la nouvelle colonne A. La somme des longueurs des domaines de A1,...,An doit être inférieure ou égale à la la longueur du domaine de A. Le concepteur doit évaluer cela en analysant chaque instances de A1,...,An.

S Création de E.A : voir la création d'une colonne (page 77).

Transfert des instances :

```
UPDATE TABLE E SET A = fct(A1,...,An)
               WHERE A1 IS NOT NULL AND ... AND An IS NOT NULL;
UPDATE TABLE E1 SET A2 = NULL WHERE A21 IS NULL AND ... AND A2n IS NULL;
```

Destruction de E.A1, ..., E.An : voir la destruction d'une colonne (page 77).

Programmes

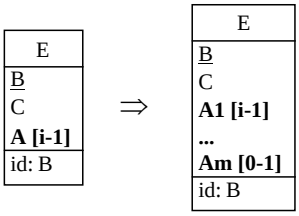
D Dépendance structurelle : dans les requêtes, il faut remplacer les colonnes A1,...,An par A. Dans les interfaces, les champs pour les colonnes A1, ..., An doivent être remplacés par un seul champ.

L Recherche sur les colonnes E.A1, ..., E.An et sur toutes les variables qui en dépendent.

g) Transformation d'une colonne en plusieurs colonnes (instanciation)

Données

D Dépendance : les instances de A vont être transférées dans la colonne A1. il faut que les domaines de valeurs soient compatibles.



I Les instances de A doivent être compatibles avec le domaine de valeurs de A1. Le concepteur doit évaluer cela en analysant chaque instance de A.

S Création de E.A1, ..., E.An : voir la création d'une colonne (page 77).

Transfert des instances :
```
UPDATE TABLE E SET A1 = A;
```

Destruction de E.A: voir la destruction d'une colonne (page 77).

Programmes

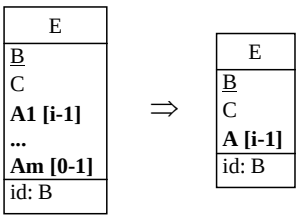
D Dépendance structurelle : dans les requêtes, il faut remplacer la colonne A par A1,...,Am. Dans les interfaces, il faut remplacer le champ A par les champs A1,...,Am.

L Recherche sur la colonne E.A ainsi que toutes les variables qui en dépendent.

h) Transformation de plusieurs colonnes (instanciation) en une colonne

Données

D Dépendance : les instances de A1,...,Am vont être transférées dans la colonne A. Cela pose des problèmes avec le type et la longueur de la colonne A. Si A2, ..., Am contiennent des valeurs, elles sont perdues.



I Les instances de A1 doivent être compatibles avec le type de la nouvelle colonne A. Le concepteur doit évaluer cela en analysant chaque instance de A1.

Les colonnes A2, ..., Am doivent être vides sinon les informations sont perdues. La requête ci-dessous ne doit donner aucun résultat.
```
SELECT A2,...,Am FROM E WHERE A2 IS NOT NULL OR ... OR Am IS NOT NULL;
```

S Création de E.A : voir la création d'une colonne (page 77).

Transfert des instances :
```
UPDATE TABLE E SET A = A1;
```

Destruction de E.A1, ..., E.An : voir la destruction d'une colonne (page 77).

Programmes

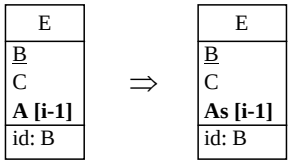
D Dépendance structurelle : dans les requêtes, il faut remplacer les colonnes A1,...,Am par A. Dans les interfaces, les champs A1, ..., Am sont remplacés par un seul champ.

L Recherche sur les colonnes E.A1, ..., E.Am et sur toutes les variables qui en dépendent.

i) Transformation d'une colonne en une colonne (concaténation)

Données

D Dépendance : les instances de A sont transférées dans As. Les types et longueurs des colonnes sont compatibles.



I Les instances de A1 doivent être compatibles avec le type de la nouvelle colonne As. Le concepteur doit évaluer cela en analysant chaque instance de A.

S Création de E.As : voir la création d'une colonne (page 77).

Transfert des instances :

```
UPDATE TABLE E SET As = A;
```

Destruction de E.A: voir la destruction d'une colonne (page 77).

Programmes

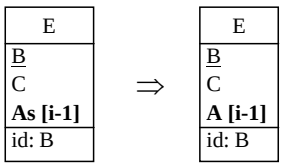
D Dépendance structurelle : dans les requêtes, il faut remplacer la colonne A par As. Dans les interfaces, comme As représente une concaténation de valeurs, il faut peut-être plusieurs champs dans lesquels les instances de As vont être décomposées et stockées.

L Recherche sur la colonne E.A ainsi que toutes les variables qui en dépendent.

j) Transformation d'une colonne (concaténation) en une colonne

Données

D Dépendance : les instances de As sont transférées dans A. Compatibilité de type et de longueur.



I Les instances de As doivent être compatibles avec le type de la nouvelle colonne As. Le concepteur doit évaluer cela en analysant chaque instance de As.

S Création de E.A : voir la création d'une colonne (page 77).

Transfert des instances :

```
UPDATE TABLE E SET A = As;
```

Destruction de E.As: voir la destruction d'une colonne (page 77).

Programmes

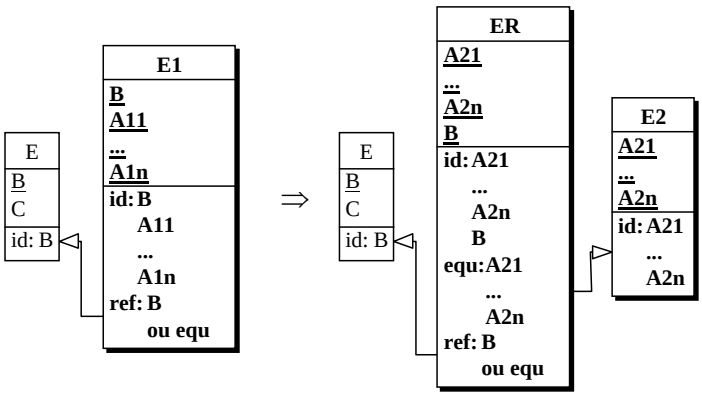
D Dépendance structurelle : dans les requêtes, il faut remplacer la colonne As par A. Dans les interfaces, les instances de A ne doivent plus être décomposées et stockées dans plusieurs champs.

L Recherche sur la colonne E.As ainsi que toutes les variables qui en dépendent.

k) Transformation d'une table (représentation des instances) en plusieurs tables (représentation des valeurs)

Données

D Dépendance : les instances de E1 vont être transférées dans les tables E2 et ER. Les types et longueurs des colonnes doivent être compatibles.



I Les instances de A11, ..., A1n doivent être compatibles avec les types et longueurs des nouvelles colonnes A21, ..., A2n. Le concepteur doit évaluer cela en analysant chaque instance de A11, ..., A1n.

S Création de ER et E2 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances de E1 dans ER :

```
-- Si transfert de plusieurs colonnes vers une seule colonne :
INSERT INTO TABLE E2 SELECT DISTINCT fct(A11,...,A1n) FROM E1;
INSERT INTO TABLE ER SELECT fct(A11,...,A1n),B FROM E1;
-- Si transfert d'une seule colonne vers plusieurs colonnes :
INSERT INTO TABLE E2 SELECT DISTINCT fct1(A11),...,fctn(A11) FROM E1;
INSERT INTO TABLE ER SELECT fct1(A11),...,fctn(A11),B FROM E1;
-- Si transfert de plusieurs colonnes vers plusieurs colonnes :
INSERT INTO TABLE E2 SELECT DISTINCT A11,...,A1n FROM E1;
INSERT INTO TABLE ER SELECT A11,...,A1n,B FROM E1;
```

Destruction de E1 : voir la destruction d'une table (page 75).

Programmes

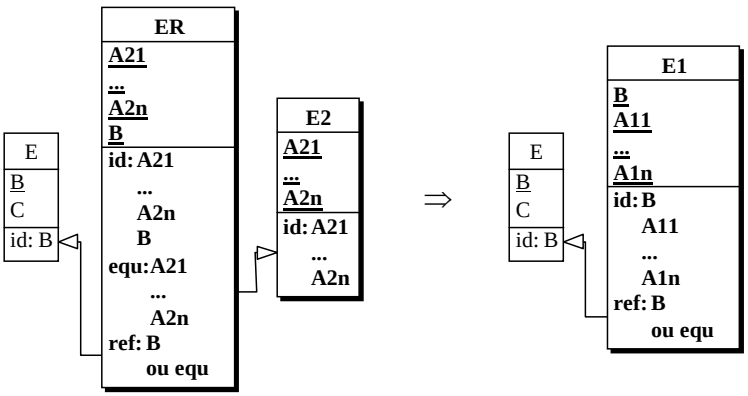
D Dépendance structurelle : dans les requêtes, il y a une jointure ou une requête imbriquée supplémentaire. Dans les interfaces, il faut gérer deux tables pour obtenir les informations en ajoutant, par exemple, une fenêtre de gestion spécifique de la table E2.

L Recherche sur la table E1, ses colonnes ainsi que les variables qui en dépendent.

l) Transformation de plusieurs tables (représentation des valeurs) en une table (représentation des instances)

Données

D Dépendance : les instances de ER vont être transférées dans la table E1. Les types et longueurs des colonnes doivent être compatibles.



I Les instances de A21, ..., A2n doivent être compatibles avec les types et longueurs des nouvelles colonnes A11, ..., A1n. Le concepteur doit évaluer cela en analysant chaque instance de A21, ..., A2n.

S Création de E1 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances de ER dans E1 :

```
-- Si transfert de plusieurs colonnes vers une seule colonne :
INSERT INTO E1 SELECT B,fct(A21,...,A2n) FROM ER;
-- Si transfert d'une seule colonne vers plusieurs colonnes :
INSERT INTO E1 SELECT B,fct1(A21),...,fctn(A21) FROM ER;
-- Si transfert de plusieurs colonnes vers plusieurs colonnes :
INSERT INTO E1 SELECT B,A21,...,A2n FROM ER;
```

Destruction de ER et E2 : voir la destruction d'une table (page 75).

Programmes

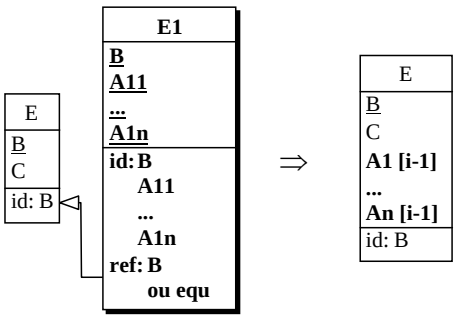
D Dépendance structurelle : dans les requêtes, il y a une jointure ou une requête imbriquée en moins. Dans les interfaces, la gestion des tables E2 et ER est remplacée par la gestion de la seule table E1.

L Recherche sur les tables E2 et ER ainsi que sur toutes leurs colonnes et les variables qui en dépendent.

m) Transformation d'une table (représentation des instances) en plusieurs colonnes (désagrégation)

Données

D Dépendance : Les instances de E1 vont être transférées dans les colonnes A1,...,An. Cela peut poser des problèmes si une instance de E est référencée par plusieurs instances de E1. Les types des colonnes A11, ..., A1n et A1, ..., An doivent être compatibles.



I Il faut vérifier qu'une instance de E est référencée par une instance de E1 au maximum. Si la requête ci-dessous donne un résultat, cela nécessite une intervention du concepteur.

```
SELECT B FROM E1 GROUP BY B HAVING COUNT(B) > 1;
```

Les instances de A11, ..., A1n doivent être compatibles avec les types et les longueurs des nouvelles colonnes A1, ..., An. Le concepteur doit évaluer cela en analysant chaque instance de A11, ..., A1n.

S Création de E.A1, ..., E.An : voir la création d'une colonne (page 76).

Transfert des instances :

```
-- Si transfert d'une seule colonne (n = 1) :
INSERT INTO E(A1,...,An) SELECT fct1(A11),...,fctn(A11) FROM E1
                         WHERE E.B = E1.B;
-- Si transfert de plusieurs colonnes (n > 1) :
INSERT INTO E(A1,...,An) SELECT A11,...,A1n FROM E1 WHERE E.B = E1.B;
```

Destruction de E1 : voir la destruction d'une table (page 75).

Programmes

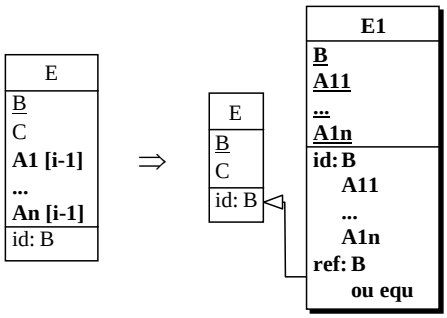
D Dépendance structurelle dans les requêtes, il y a une jointure ou une requête imbriquée qui n'est plus nécessaire. Dans les interfaces, les listes ou les boîtes de dialogue qui accèdent à la table E1 sont remplacées pas la gestion de plusieurs champs.

L Recherche sur la table E1, ses colonnes ainsi que les variables qui en dépendent.

n) Transformation de plusieurs colonnes (désagrégation) en une table (représentation des instances)

Données

D Dépendance : les types et longueurs des colonnes doivent être compatibles.



I Le concepteur doit vérifier si les instances de A1,...,An sont compatibles avec les types et longueurs des colonnes de E1.

S Création de E1 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO E1 SELECT B,fct(A1,...,An) FROM E
               WHERE A1 IS NOT NULL AND ... AND An IS NOT NULL;
-- Si transfert vers plusieurs colonnes (n > 1) :
INSERT INTO E1 SELECT B,A11,...,A1n from E
               WHERE A1 IS NOT NULL AND ... AND An IS NOT NULL;
```

Destruction de E.A1, ..., E.An : voir la destruction d'une colonne (page 77).

Programmes

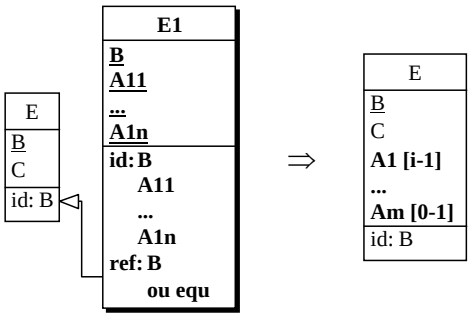
D Dépendance structurelle : dans les requêtes, il faut faire une jointure ou une requête imbriquée supplémentaire pour accèder aux colonnes. Dans les interfaces, les champs A1, ..., An doivent être remplacés par la gestion d'une liste ou d'une nouvelle fenêtre pour la table E1.

L Recherche sur les colonnes E.A1, ..., E.An et les variables qui en dépendent.

o) Transformation d'une table (représentation des instances) en plusieurs colonnes (instanciation)

Données

D Dépendance : les instances de E1 vont être transférées dans les colonnes A1,...,Am. Cela peut poser des problèmes de compatibilité entre les types et les longueurs des colonnes A11,...,A1n et A1,...,Am. En plus, E ne peut être référencé que par m instances de E1 au maximum.



I Il faut vérifier qu'une instance de E est référencée par au maximum m instances de E1. Si la requête ci-dessous donne un résultat, il faut une intervention du concepteur.

```
SELECT B FROM E1 GROUP BY B HAVING COUNT(B) > m;
```

Les instances de A11, ..., A1n doivent être compatibles avec le type des colonnes A1, ..., Am. Le concepteur doit évaluer cela en analysant chaque instance de A11, ..., A1n.

S Création de E.A1, ..., E.Am : voir la création d'une colonne (page 76).

Transfert des instances :

```
CREATE OR REPLACE PROCEDURE Trf_data IS
    CURSOR c1 IS SELECT * FROM E WHERE E.B IN (SELECT E1.B FROM E1);
    CURSOR c2 IS SELECT * FROM E1;
    tE c1%ROWTYPE;
    tE1 c2%ROWTYPE;
    comp NUMBER;
  BEGIN
    FOR tE IN c1 LOOP
      comp := 1;
```

```
      FOR tE1 IN c2 LOOP
        IF tE1.B=tE.B THEN
          IF comp=1 THEN
            -- si transfert de plusieurs colonnes (n > 1) :
            UPDATE E SET A1=fct(tE1.A11,...,tE1.A1n) WHERE B=tE.B; END IF;
            -- si transfert d'une seule colonne (n = 1) :
            -- UPDATE E SET A1=tE1.A11 WHERE B=tE.B; END IF;
          ...;
          IF comp=m THEN
            -- si transfert de plusieurs colonnes (n > 1) :
            UPDATE E SET Am=fct(tE1.A11,...,tE1.A1n) WHERE B=tE.B; END IF;
            -- si transfert d'une seule colonne (n = 1) :
            -- UPDATE E SET Am=tE1.A11 WHERE B=tE.B; END IF;
          comp := comp + 1;
        END IF;
      END LOOP;
    END LOOP;
  END;
```

Destruction de E1 : voir la destruction d'une table (page 75).

Programmes

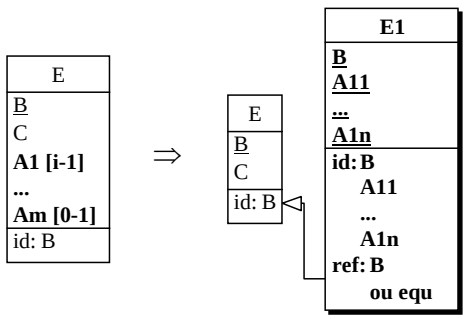
D Dépendance structurelle : dans les requêtes, il y a une jointure ou une requête imbriquées supplémentaires. Dans les interfaces, la gestion d'une liste ou d'une fenêtre est remplacée par m champs.

L Recherche sur la table E1, ses colonnes ainsi que les variables qui en dépendent.

p) Transformation de plusieurs colonnes (instanciation) en une table (représentation des instances)

Données

D Dépendance : les instances de A1, ... Am vont être transférées dans la table E1. Cela peut poser des problèmes de compatibilité entre les types et les longueurs des colonnes A11,...,A1n et A1,...,Am.



I Les instances de A1,...,Am doivent être compatibles avec les types et longueurs des colonnes de E1. Le concepteur doit évaluer cela en analysant chaque instance de A1, ..., Am.

S Création de E1 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO E1 SELECT B,A1 from E WHERE A1 IS NOT NULL;
...;
INSERT INTO E1 SELECT B,Am from E WHERE Am IS NOT NULL;
-- Si transfert vers plusieurs colonnes (n > 1) :
```

```
INSERT INTO E1 SELECT B,fct1(A1),...,fctn(A1) from E WHERE A1 IS NOT NULL;
...;
INSERT INTO E1 SELECT B,fct1(Am),...,fctn(Am) from E WHERE Am IS NOT NULL;
```

Destruction de E.A1,...,E.Am : voir la destruction d'une colonne (page 77).

Programmes

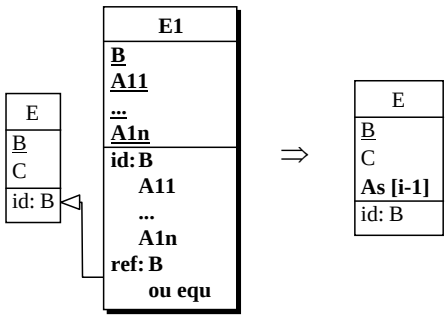
D Dépendance structurelle : dans les requêtes, il faut faire une jointure ou une requête imbriquée supplémentaire pour accèder aux colonnes de E1. Dans les interfaces, la gestion d'une liste ou de m champs doit être remplacée par la gestion de la table E1.

L Recherche sur les colonnes E.A1, ..., E.Am et les variables qui en dépendent.

q) Transformation d'une table (représentation des instances) en une colonne (concaténation)

Données

D Dépendance : les instances de E1 vont être transférées dans la colonne As. Cela peut poser des problèmes de compatibilité entre les types et longueurs des colonnes A1,...,An et As.



I Les instances de A1,...,An doivent être compatibles avec le type de la colonne As. Il faut aussi vérifier que la longueur de la concaténation des valeurs des instances de A1, ..., An pour une instance de E est inférieur à la longueur de As. Le concepteur doit évaluer cela en analysant chaque instance de A1, ..., An.

S Création de E.As : voir la création d'une colonne (page 76).

Transfert des instances :

```
-- Transférer les données de E1 dans (As).
CREATE OR REPLACE PROCEDURE Trf_data IS
    CURSOR c1 IS SELECT * FROM E WHERE E.B IN (SELECT E1.B FROM E1);
    CURSOR c2 IS SELECT * FROM E1;
    tE c1%ROWTYPE;
    tE1 c2%ROWTYPE;
    tmp CHAR(1000);
    i integer;
  BEGIN
    FOR tE IN c1 LOOP
      tmp := '';
      i := 0;
      FOR tE1 IN c2 LOOP
        IF tE1.B = tE.B THEN
          -- si transfert de plusieurs colonnes (n > 1) :
          --   fct est une fonction d'agrégation des valeurs de A11, ..., A1n.
          --   Dans le cas de valeurs de type caractère, CONCAT est utilisé.
          tmp := SUBSTR(CONCAT(SUBSTR(tmp,1,i),fct(tE1.A11,...,tE1.A1n)),
                        1,LENGTH(tmp));
          i := i + 1 + LENGTH(tE1.A11) + ... + LENGTH(tE1.A1n);
```

```
          -- si transfert d'une seule colonne (n = 1) :
          --   fct transforme les valeurs de A11 en valeurs de type caractère.
          -- tmp := SUBSTR(CONCAT(SUBSTR(tmp,1,i),fct(tE1.A11)),1,LENGTH(tmp));
          -- i := i + 1 + LENGTH(tE1.A11);
        END IF;
      END LOOP;
      UPDATE E SET As=SUBSTR(tmp,1,LENGTH(tmp)) WHERE B=tE.B;
    END LOOP;
  END;
```

Destruction de E1 : voir la destruction d'une table (page 75).

Programmes

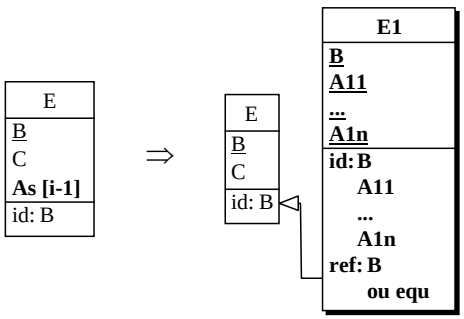
D Dépendance structurelle : dans les requêtes, il y a une jointure ou une requête imbriquée qui n'est plus nécessaire. Dans les interfaces, il faut remplacer une liste par un champ ou éventuellement plusieurs champs (As décomposé).

L Recherche sur la table E1, ses colonnes et les variables qui en dépendent.

r) Transformation d'une colonne (concaténation) en une table (représentation des instances)

Données

D Dépendance : les instances de As vont être intégrées dans la table E1, cela peut poser des problèmes avec les types et les longueurs des colonnes de E1.



I Si As est transféré dans plusieurs colonnes, ces instances doivent être compatibles avec les types et longueurs des nouvelles colonnes A11, ..., A1n. Le concepteur doit évaluer cela en analysant les instances de As une par une.

S Création de E1 : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert dans une seule colonne (n = 1) :
INSERT INTO E1 SELECT B,fct1(As) FROM E WHERE fct1(As) IS NOT NULL;
...;
INSERT INTO E1 SELECT B,fctm(As) FROM E WHERE fctm(As) IS NOT NULL;
-- Si transfert dans plusieurs colonnes (n > 1) :
INSERT INTO E1 SELECT B,fct11(As),...,fctn1(As) FROM E
               WHERE fct11(As) IS NOT NULL AND ... AND fctn1(As) IS NOT NULL;
...;
INSERT INTO E1 SELECT B,fct1m(As),...,fctnm(As) FROM E
               WHERE fct1m(As) IS NOT NULL AND ... AND fctnm(As) IS NOT NULL;
```

Destruction de E.As : voir la destruction d'une colonne (page 77).

Programmes

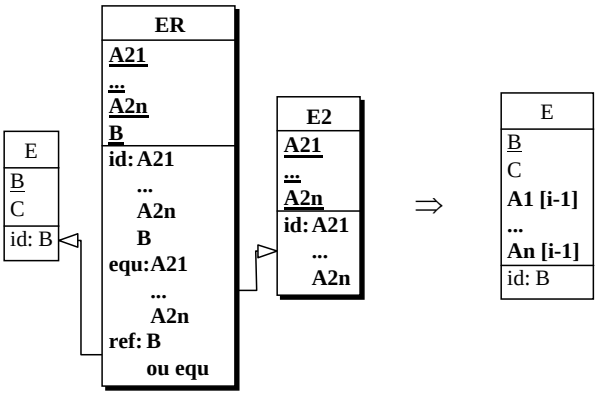
D Dépendance structurelle : dans les requêtes, pour accèder aux colonnes de E1, il faut une jointure ou une requête imbriquée supplémentaire. Dans les interfaces, le ou les champs de la colonne As doivent être remplacés par une liste de valeurs ou la gestion d'une nouvelle fenêtre pour E1.

L Recherche sur la colonne E.As ainsi que toutes les variables qui en dépendent.

s) Transformation de plusieurs tables (représentation des valeurs) en plusieurs colonnes (désagrégation)

Données

D Dépendance : les instances de ER vont être transférées dans les colonnes A1,...,An de E. Cela pose des problèmes si une instance de E est référencée par plus d'une instance de ER. Les types et longueurs des colonnes A1, ..., An et A21, ..., A2n doivent être compatibles.



I Il faut vérifier qu'une instance de E est référencée par une instance de ER au maximum. Si la requête ci-dessous donne un résultat, cela nécessite une intervention du concepteur pour règler le problème.

```
SELECT B FROM ER GROUP BY B HAVING COUNT(B) > 1;
```

Les instances de A21, ..., A2n doivent être compatibles avec les types et longueurs des nouvelles colonnes A1, ..., An. Le concepteur doit évaluer cela en analysant chaque instance de A21, ..., A2n.

S Création de E.A1, ..., E.An : voir la création d'une colonne (page 76).

Transfert des instances de ER dans A1,...,An :

```
-- Si transfert d'une seule colonne (n = 1) :
UPDATE INTO E SET A1 = (SELECT fct1(A21) FROM ER WHERE E.B = ER.B),...,
                  An = (SELECT fctn(A21) FROM ER WHERE E.B = ER.B);
-- Si transfert de plusieurs colonnes (n > 1) :
UPDATE INTO E SET A1 = (SELECT A21 FROM ER WHERE E.B = ER.B),...,
                  An = (SELECT A2n FROM ER WHERE E.B = ER.B);
```

Destruction de E2 et ER: voir la destruction d'une table (page 75).

Programmes

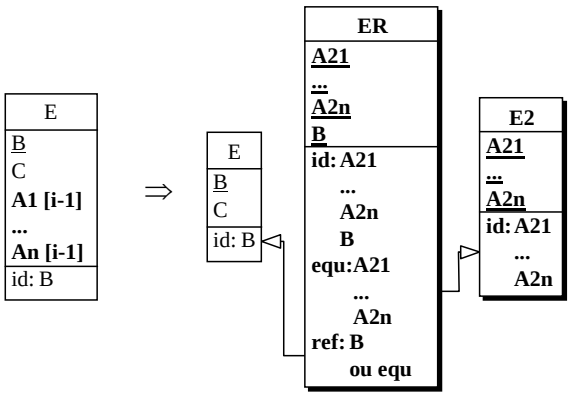
D Dépendance structurelle : dans les requêtes, il y a une jointure ou des requêtes imbriquées qui ne sont plus nécessaires. Dans les interfaces, la gestion des fenêtres ou des listes pour ER et E2 doit être remplacée par la gestion de n champs.

L Recherche sur les tables E2 et ER ainsi que sur toutes leurs colonnes et les variables qui en dépendent.

t) Transformation de plusieurs colonnes (désagrégation) en plusieurs tables (représentation des valeurs)

Données

D Dépendance : Les colonnes A1, ..., An doivent avoir des instances compatibles avec les types et longueurs des colonnes A21, ..., A2n.



I Le concepteur doit vérifier si les instances de A1,...,An sont compatibles avec les types et longueurs des colonnes de E2.

S Création de E2 et ER : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO E2 SELECT DISTINCT fct(A1,...,An) from E
               WHERE A1 IS NOT NULL OR ... OR An IS NOT NULL;
INSERT INTO ER SELECT fct(A1,...,An),B from E
               WHERE A1 IS NOT NULL OR ... OR An IS NOT NULL;
-- Si transfert vers plusieurs colonnes (n > 1) :
INSERT INTO E2 SELECT DISTINCT A1,...,An from E
               WHERE A1 IS NOT NULL AND ... AND An IS NOT NULL;
INSERT INTO ER SELECT A1,...,An,B from E
               WHERE A1 IS NOT NULL AND ... AND An IS NOT NULL;
```

Destruction de E.A1, ..., E.An : voir la destruction d'une colonne (page 77)

Programmes

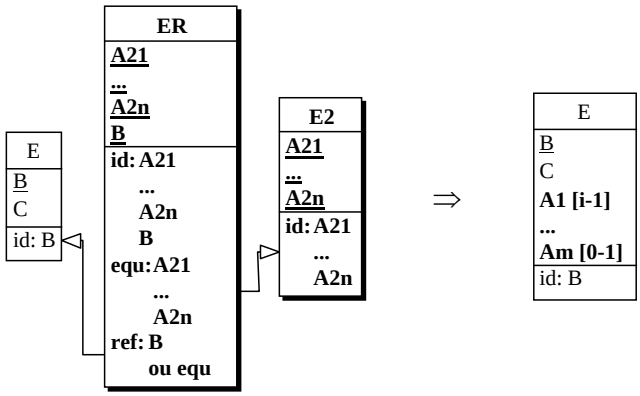
D Dépendance structurelle : dans les requêtes, il faut faire une jointures ou deux requêtes imbriquées supplémentaires pour accèder à la colonne. Dans les interfaces, la gestion de n champs est remplacée par la gestion de listes ou de fenêtres pour E2 et ER.

L Recherche sur les colonnes E.A1, ..., E.An et sur toutes les variables qui en dépendent.

u) Transformation de plusieurs tables (représentation des valeurs) en plusieurs colonnes (instanciation)

Données

D Dépendance : les instances de ER vont être transférées dans les colonnes A1,...,Am de E. Cela pose des problèmes si une instance de E est référencée par plus de m instances de ER. il faut aussi vérifier la compatibilité des types et longueurs de colonnes A1, ..., Am et A21, ..., A2n.



I Il faut vérifier qu'une instance de E est référencée par au maximum m instances de ER. Si la requête ci-dessous donne un résultat, il faut une intervention du concepteur.

```
SELECT B FROM ER GROUP BY B HAVING COUNT(B) > m;
```

Les instances de A21,...,A2n doivent être compatibles avec le type des colonnes A1, ..., Am. Le concepteur doit évaluer cela en analysant chaque instances de A21, ..., A2n.

S Création de E.A1, ..., E.Am : voir la création d'une colonne (page 76).

Transfert des instances de ER dans A1, ..., Am :

```
CREATE OR REPLACE PROCEDURE Trf_data IS
    CURSOR c1 IS SELECT * FROM E WHERE B IN (SELECT B FROM ER);
    CURSOR c2 IS SELECT * FROM ER;
    tE c1%ROWTYPE;
    tER c2%ROWTYPE;
    comp NUMBER;
  BEGIN
    FOR tE IN c1 LOOP
      comp := 1;
      FOR tER IN c2 LOOP
        IF tER.B = tE.B THEN
          IF comp=1 THEN
            -- si transfert de plusieurs colonnes (n > 1) :
            UPDATE E SET A1=fct(tER.A21,...,tER.A2n) WHERE B=tE.B; END IF;
            -- si transfert d'une seule colonne (n = 1) :
            -- UPDATE E SET A1=tER.A21 WHERE B=tE.B; END IF;
          ...;
          IF comp=m THEN
            -- si transfert de plusieurs colonnes (n > 1) :
            UPDATE E SET Am=fct(tER.A21,...,tER.A2n) WHERE B=tE.B; END IF;
            -- si transfert d'une seule colonne (n = 1) :
            -- UPDATE E SET Am=tER.A21 WHERE B=tE.B; END IF;
          comp := comp + 1;
        END IF;
      END LOOP;
    END LOOP;
  END;
```

Destruction de E2 et ER: voir la destruction d'une table (page 75).

Programmes

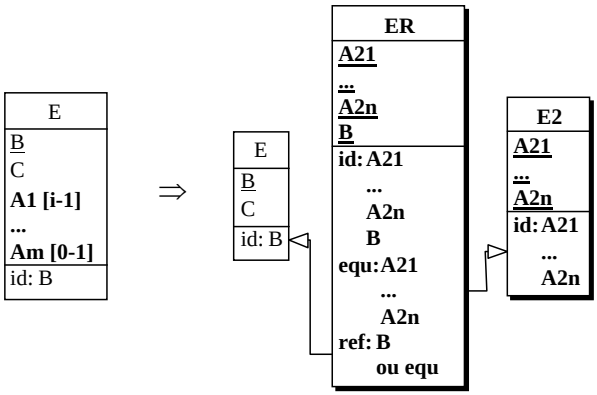
D Dépendance structurelle : dans les requêtes, il y a des jointures ou des requêtes imbriquées à enlever. Dans les interfaces, la gestion des listes ou fenêtres pour E2 et ER doit être remplacée par la gestion de m champs.

L Recherche sur les tables E2 et ER ainsi que sur toutes leurs colonnes et les variables qui en dépendent.

v) Transformation de plusieurs colonnes (instanciation) en plusieurs tables (représentation des valeurs)

Données

D Dépendance : les types et les longueurs des colonnes A1, ..., Am et A21, ..., A2n doivent être compatibles.



I Les instances de A1,...,Am doivent être compatibles avec les types et longueurs des colonnes de ER. Le concepteur doit évaluer cela en analysant chaque instance de A1, ..., Am.

S Création de E2 et ER : voir la création d'une table (page 74) et la création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO E2 SELECT DISTINCT A1 from E WHERE A1 IS NOT NULL AND
                                               A1 NOT IN (SELECT A21 FROM E2);
...;
INSERT INTO E2 SELECT DISTINCT Am from E WHERE Am IS NOT NULL AND
                                               Am NOT IN (SELECT A21 FROM E2);
INSERT INTO ER SELECT A1,B from E WHERE A1 IS NOT NULL;
...;
INSERT INTO ER SELECT Am,B from E WHERE Am IS NOT NULL;
-- Si transfert vers plusieurs colonnes (n > 1) :
INSERT INTO E2 SELECT DISTINCT fct1(A1),...,fctn(A1) from E WHERE A1 IS NOT NULL AND
  fct1(A1),...,fctn(A1) NOT IN (SELECT A21,...,A2n FROM E2);
...;
INSERT INTO E2 SELECT DISTINCT fct1(Am),...,fctn(Am) from E WHERE Am IS NOT NULL AND
  fct1(Am),...,fctn(Am) NOT IN (SELECT A21,...,A2n FROM E2);
INSERT INTO ER SELECT fct1(A1),...,fctn(A1),B from E WHERE A1 IS NOT NULL;
...;
INSERT INTO ER SELECT fct1(Am),...,fctn(Am),B from E WHERE Am IS NOT NULL;
```

Destruction de E.A1,...,E.Am : voir la destruction d'une colonne (page 77).

R Le script de transfert des instances peut être simplifié si on déplace la création des contraintes référentielles après le transfert. Cela permet de remplir la table ER avant la table E2. On obtient le script :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO ER SELECT A1,B from E WHERE A1 IS NOT NULL;
...;
INSERT INTO ER SELECT Am,B from E WHERE Am IS NOT NULL;
INSERT INTO E2 SELECT DISTINCT A21 from ER;
-- Si transfert vers plusieurs colonnes (n > 1) :
INSERT INTO ER SELECT fct1(A1),...,fctn(A1),B from E WHERE A1 IS NOT NULL;
...;
INSERT INTO ER SELECT fct1(Am),...,fctn(Am),B from E WHERE Am IS NOT NULL;
INSERT INTO E2 SELECT DISTINCT A21,...,A2n from ER;
```

Programmes

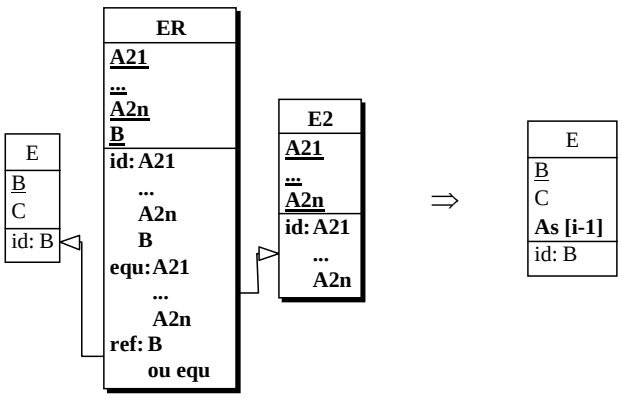
D Dépendance structurelle : dans les requêtes, il faut faire une jointure ou deux requêtes imbriquées supplémentaires pour accèder aux colonnes de E2. Dans les interfaces, la gestion des champs A1, ..., Am doit être remplacée par la gestion de listes ou de fenêtres pour ER et E2.

L Recherche sur les colonnes E.A1,...,E.Am et sur toutes les variables qui en dépendent.

w) Transformation de plusieurs tables (représentation des valeurs) en une colonne (concaténation)

Données

D Dépendance : les instances de ER vont être transférées dans la colonne As de E. Cela pose des problèmes si la longueur de la concaténation des instances référencant une instance de E est supérieure à la longueur de As.



I Les instances de A21, ..., A2n doivent être compatibles avec le type et la longueur de As. Il faut aussi vérifier que la longueur de la concaténation des valeurs des instances de A21, ..., A2n pour une instance de E est inférieure ou égale à la longueur de As. Le concepteur doit évaluer cela en analysant chaque instance de A21, ..., A2n.

S Création de E.As : voir la création d'une colonne (page 76).

Transfert des instances de ER dans As :

```
CREATE OR REPLACE PROCEDURE Trf_data IS
    CURSOR c1 IS SELECT * FROM E WHERE E.B IN (SELECT B FROM ER);
    CURSOR c2 IS SELECT * FROM ER;
    tE c1%ROWTYPE;
    tR c2%ROWTYPE;
    tmp CHAR(1000);
```

```
  i integer;
BEGIN
  FOR tE IN c1 LOOP
    tmp := '';
    i := 0;
    FOR tER IN c2 LOOP
      IF tER.ncli=tE.ncli THEN
        -- Si transfert de plusieurs colonnes (n > 1) :
        --   fct est une fonction d'agrégation des valeurs de A21, ..., A2n.
        --   Dans le cas de valeurs de type caractère, CONCAT est utilisé.
        tmp := SUBSTR(CONCAT(SUBSTR(tmp,1,i),fct(tER.A21,...,tER.A2n)),
                     1,LENGTH(tmp));
        i := i + 1 + LENGTH(tER.A21) + ... + LENGTH(tER.A2n);
        -- Si transfert d'une seule colonne (n = 1) :
        --   fct transforme les valeurs de A21 en valeurs de type caractère.
        -- tmp := SUBSTR(CONCAT(SUBSTR(tmp,1,i),fct(tER.A21)),1,LENGTH(tmp));
        -- i := i + 1 + LENGTH(tER.A21);
      END IF;
    END LOOP;
    UPDATE E SET As = SUBSTR(tmp,1,LENGTH(tmp)) WHERE B=tE.B;
  END LOOP;
END;
```

Destruction de E2 et ER: voir la destruction d'une table (page 75).

Programmes

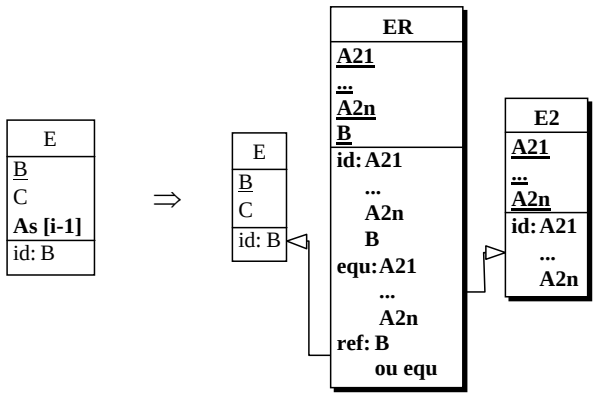
D Dépendance structurelle : dans les requêtes, il y a une jointure ou des requêtes imbriquées qui ne sont plus nécessaires. Dans les interfaces, la gestion des fenêtres ou des listes pour E2 et ER est remplacée par la gestion d'un champ (ou plusieurs si As est décomposé).

L Recherche sur les tables E2 et ER ainsi que sur toutes leurs colonnes et les variables qui en dépendent.

x) Transformation d'une colonne (concaténation) en plusieurs tables (représentation des valeurs)

Données

D Dépendance : les instances de As vont être intégrées dans la table E1, cela peut poser des problèmes au niveau des types et des longueurs des colonnes A1,...,An.



I Les instances de As doivent être compatibles avec les types et longueurs des nouvelles colonnes. Le concepteur doit évaluer cela en analysant chaque instance de As.

S Création de E2 et ER : voir la création d'une table (page 74) et création d'une clé étrangère (page 82).

Transfert des instances :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO E2 SELECT DISTINCT fct1(As) from E WHERE fct1(As) IS NOT NULL AND
  fct1(As) NOT IN (SELECT A21 FROM E2);
...;
INSERT INTO E2 SELECT DISTINCT fctm(As) from E WHERE fctm(As) IS NOT NULL AND
  fctm(As) NOT IN (SELECT A21 FROM E2);
INSERT INTO ER SELECT fct1(As),B from E WHERE fct1(As) IS NOT NULL;
...;
INSERT INTO ER SELECT fctm(As),B from E WHERE fctm(As) IS NOT NULL;
-- Si transfert vers plusiers colonnes (n > 1) :
INSERT INTO E2 SELECT DISTINCT fct11(As),...,fctn1(As) from E WHERE fct11(As) IS
  NOT NULL AND ... AND fctn1(As) AND fct11(As),...,fctn1(As) NOT IN
  (SELECT A21,...,A2n FRON E2);
...;
INSERT INTO E2 SELECT DISTINCT fct1m(As),...,fctnm(As) from E WHERE fct1m(As) IS
  NOT NULL AND ... AND fctnm(As) AND fct1m(As),...,fctnm(As) NOT IN
  (SELECT A21,...,A2n FRON E2);
INSERT INTO ER SELECT fct11(As),...,fctn1(As),B from E WHERE fct11(As) IS NOT NULL
  AND ... AND fctn1(As) IS NOT NULL;
...;
INSERT INTO ER SELECT fct1m(As),...,fctnm(As),B from E WHERE fct1m(As) IS NOT NULL
  AND ... AND fctnm(As) IS NOT NULL;
```

Destruction de E.As : voir la destruction d'une colonne (page 77).

R Le script de transfert des instances peut être simplifié si on déplace la création des contraintes référentielles après le transfert. Cela permet de remplir la table ER avant la table E2. On obtient le script :

```
-- Si transfert vers une seule colonne (n = 1) :
INSERT INTO ER SELECT fct1(As),B from E WHERE As IS NOT NULL;
...;
INSERT INTO ER SELECT fctm(As),B from E WHERE As IS NOT NULL;
INSERT INTO E2 SELECT DISTINCT(A21) from ER;
-- Si transfert vers plusiers colonnes (n > 1) :
INSERT INTO ER SELECT fct11(As),...,fctn1(As),B from E WHERE As IS NOT NULL;
...;
INSERT INTO ER SELECT fct1m(As),...,fctnm(As),B from E WHERE As IS NOT NULL;
INSERT INTO E2 SELECT DISTINCT A21,...,A2n from ER;
```

Programmes

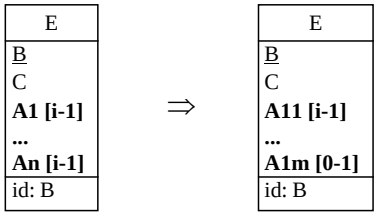
D Dépendance structurelle : dans les requêtes, il faut faire une jointure ou deux requêtes imbriquées supplémentaires pour accèder aux colonnes de E2. Dans les interfaces, le champ de la colonne va être remplacé par des listes ou des écrans de gestion supplémentaires.

L Recherche sur la colonne E.As ainsi que sur les variables qui en dépendent.

y) Transformation de plusieurs colonnes (désagrégation) en plusieurs colonnes (instanciation)

Données

D Dépendance : les instances de A1, ..., An sont transférées dans A11. Les longueurs et les types de A1, ..., An et A11 sont compatibles.



I La somme des instances de A1, ..., An est inférieure ou égale aux longueurs des A11, ..., A1m. Les types de A1,...,An sont compatibles avec chaque type de A11, ..., A1m.

S Création de E.A11, ..., E.A1m : voir la création d'une colonne (page 76).

Transfert des instances :

```
UPDATE TABLE E SET A11 = fct(A1,...,An)
               WHERE A1 IS NOT NULL OR ... OR An NOT IS NULL;
UPDATE TABLE E SET A11 = NULL WHERE A1 IS NULL AND ... AND An IS NULL;
```

Destruction de E.A1, ..., E.An : voir la destruction d'une colonne (page 77).

Programmes

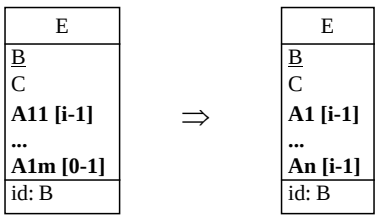
D Dépendance structurelle : il faut modifier les requêtes pour accèder aux colonnes A11, ..., A1m au lieu de A1, ..., An. Dans les interfaces, il faut afficher les colonnes A11, ..., A1m différemment (sous la forme de listes par exemple) que ne l'étaient les colonnes A1, ..., An.

L Recherche sur les colonnes E.A1, ..., E.An et sur toutes les variables qui en dépendent.

z) Transformation de plusieurs colonnes (instanciation) en plusieurs colonnes (désagrégation)

Données

D Dépendance : les types et longueurs de A11, ..., A1m et A1, ..., An doivent être compatibles. En plus, les colonnes A12, ..., A1m ne peuvent contenir aucune valeur sinon elles sont perdues lors du transfert.



I Il faut vérifier que les instances de A12, ..., A1m sont vides sinon ces valeurs sont perdues. La requête ci-dessous ne doit donner aucun résultat sinon le concepteur doit intervenir.

```
SELECT A12,...,A1m FROM E WHERE A12 IS NOT NULL OR ... OR A1m IS NOT NULL;
```

Les longueurs des instances de A11,...,A1m sont inférieures ou égales à la somme des longueurs des A1, ..., An. Les types de A11, ..., A1m sont compatibles avec les types de A1, ..., An.

S Création de E.A1, ..., E.An : voir la création d'une colonne (page 76).

Transfert des instances :

```
UPDATE TABLE E SET A1 = fct1(A11),...,An = fctn(A11) WHERE A11 IS NOT NULL;
UPDATE TABLE E SET A1 = NULL,...,An = NULL WHERE A11 IS NULL;
```

Destruction de E.A11, ..., E.A1m : voir la destruction d'une colonne (page 77).

Programmes

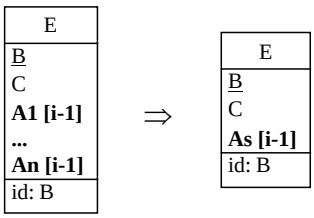
D Dépendance structurelle : il faut modifier les requêtes pour accèder aux colonnes A1, ..., An. Dans les interfaces, il faut afficher les colonnes A1, ..., An différemment (comme plusieurs champs ou un champ agrégé) que ne l'étaient les colonnes A11, ..., A1m.

L Recherche sur E.A11, ..., E.A1m et sur toutes les variables qui en dépendent.

aa)Transformation de plusieurs colonnes (désagrégation) en une colonne (concaténation)

Données

D Dépendance : la somme des longueurs de A1,...,An est inférieure ou égale à la longueur de As. Les types des attributs A1,...,An sont compatibles avec le type de As.



I La longueur des instances de A1, ..., An doit être inférieure ou égale à la longueur de As et leur type doit être compatible. La vérification de la compatibilité des longueurs et des types est à la charge du concepteur.

S Création de E.As : voir la création d'une colonne (page 76).

Transfert des instances :

```
UPDATE TABLE E SET As = fct(A1,...,An)
               WHERE A1 IS NOT NULL OR ... OR An IS NOT NULL;
UPDATE TABLE E SET As = NULL WHERE A1 IS NULL AND ... AND An IS NULL;
```

Destruction de E.A1, ..., E.An : voir la destruction d'une colonne (page 77).

Programmes

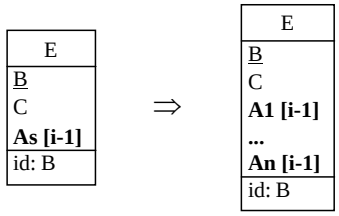
D Dépendance structurelle : il faut modifier les requêtes pour accéder à la colonne As au lieu de A1, ..., An. Dans les interfaces, il faut afficher la colonne As différemment (une liste ou un champ) que ne l'étaient les colonnes A1, ..., An.

L Recherche sur les colonnes E.A1,...,E.An et sur toutes les variables qui en dépendent.

ab)Transformation d'une colonne (concaténation) en plusieurs colonnes (désagrégation)

Données

D Dépendance : les instances de As vont être transférées dans les colonnes A1,...,An. Cela peut poser des problèmes avec les types et les longueurs des nouvelles colonnes.



I Les instances de As doivent être compatibles avec les types des nouvelles colonnes A1, ..., An. La longueur du domaine de As doit être inférieure ou égale à la somme des longueurs des domaines de A1,...,An. Le concepteur doit évaluer cela en analysant chaque instance de As.

S Création de E.A1, ..., E.An: voir la création d'une colonne (page 76).

Transfert des instances :

```
UPDATE TABLE E SET A1 = fct1(As),...,An = fctn(As) WHERE As IS NOT NULL;
UPDATE TABLE E SET A1 = NULL,...,An = NULL WHERE As IS NULL;
```

Destruction de E.As : voir la destruction d'une colonne (page 77).

Programmes

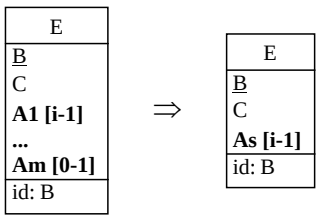
D Dépendance structurelle : dans les requêtes, il faut remplacer la colonne As par A1,...,An. Dans les interfaces, remplacer la gestion du champ As par la gestion des champs A1,...,An.

L Recherche sur la colonne E.As ainsi que sur toutes les variables qui en dépendent.

ac) Transformation de plusieurs colonnes (instanciation) en une colonne (concaténation)

Données

D Dépendance : la somme des longueurs de A1, ..., Am sont inférieures ou égales à la longueur de As. Le type des attributs A1, ..., Am est compatible avec le type de As.



I La longueur des instances de A1, ..., Am est inférieure ou égale à la longueur de As. Les types de As et A1, ..., Am sont compatibles. La vérification de ces contraintes est à la charge du concepteur.

S Création de E.As : voir la création d'une colonne (page 76).

Transfert des instances :

```
UPDATE E SET As = fct(A1,...,Am) WHERE A1 IS NOT NULL OR ... OR Am IS NOT NULL;
UPDATE E SET As = NULL WHERE A1 IS NULL AND ... AND Am IS NULL;
```

Destruction de E.A1, ..., E.Am : voir la destruction d'une colonne (page 77).

Programmes

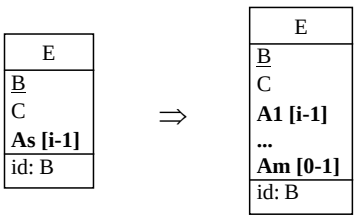
D Dépendance structurelle : il faut modifier les requêtes pour accèder à la colonne As au lieu des colonnes A1, ..., Am. Les valeurs de cette colonne peuvent devoir être décomposées. Dans les interfaces, il faut gérer la colonne As différemment (champ ou une liste) que ne l'étaient les colonnes A1, ..., Am.

L Recherche sur les colonnes E.A1, ..., E.Am et sur toutes les variables qui en dépendent.

ad)Transformation d'une colonne (concaténation) en plusieurs colonnes (instanciation)

Données

D Indépendance : les instances de As vont être transférées dans la colonne A1,...,Am. Les colonnes As, A1, ..., Am doivent être compatibles au niveau des types et des longueurs.



I La longueur des instances de As est inférieure ou égale à la somme des longueurs de A1, ..., Am. Les types de As et A1, ..., Am sont compatibles. La vérification de ces contraintes est à la charge du concepteur.

S Création de E.A1, ..., E.Am: voir la création d'une colonne (page 76).

Transfert des instances :

```
UPDATE TABLE E SET A1 = fct1(As),...,Am = fctm(As) WHERE As IS NOT NULL;
UPDATE TABLE E SET A1 = NULL,...,Am = NULL WHERE As IS NULL;
```

Destruction de E.As : voir la destruction d'une colonne (page 77).

Programmes

D Dépendance structurelle : dans les requêtes, il faut remplacer la colonne As par A1, ..., Am. Dans les interfaces, il faut remplacer la gestion du champ As par celle des champs A1, ..., Am.

L Recherche sur la colonne E.As ainsi que toutes les variables qui en dépendent.

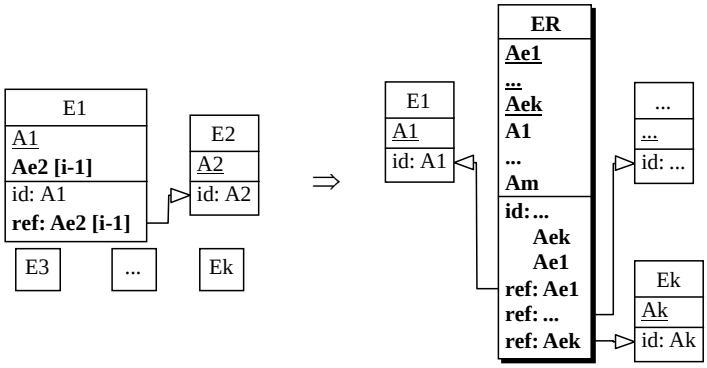
C.4.4.2 Changement de transformation sur les clés étrangères

On va reprendre les changements de transformations du point C.4.3.2 j) pour les analyser en termes de conversion de structures et de modifications de programmes.

a) Transformation d'une clé étrangère en une table

Données

D Dépendance : les instances de certaines colonnes de ER sont sans valeurs.



I Puisqu'on a introduit de nouvelles structures qui n'étaient pas présentes dans le schéma original (les colonnes Ae3, ..., Aek ainsi que A1, ..., Am de ER), elles sont sans valeur ce qui provoquera des violations de contraintes. Le concepteur doit pallier à ces manquements en donnant des valeurs par défaut ou en complétant sa base avec de nouvelles informations. Ce problème ne se pose pas si les colonnes mises en cause sont facultatives.

S Création de ER : voir la création d'une table (page 76) et la création d'une clé étrangère (page 82).

Transfert des instances de E1.Ae2 dans ER :

```
INSERT INTO ER (Ae1,Ae2) SELECT A1,Ae2 from E1 WHERE Ae2 IS NOT NULL;
-- Ae3,...,Aek,A1,...,Am doivent recevoir une valeur si elles sont obligatoires
```

Destruction de Ae2 : voir la destruction d'une clé étrangère et de ses colonnes de référence (page 83).

Programmes

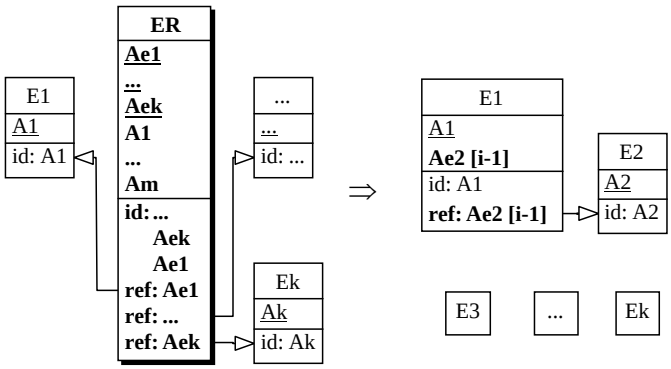
D Dépendance structurelle : dans les requêtes, les jointures et les requêtes imbriquées (entre E1 et E2) doivent être revues pour intégrer la table ER. Dans les interfaces, la gestion du lien entre E1 et E2 doit être revue en ajoutant la gestion de la table ER. En général, il faut également gérer les nouvelles clés étrangères et colonnes de référence dans tous les programmes.

L Recherche sur la colonne E1.Ae2 ainsi que toutes les variables qui en dépendent.

b) Transformation d'une table en une clé étrangère

Données

D Dépendance : les instances de ER vont être partiellement perdues lors du transfert



I La suppression de la table ER va provoquer la perte de données (notamment celles contenues dans les colonnes Ae3, ..., Aek, A1, ..., Am). Le concepteur doit évaluer les pertes et décider s'il applique la modification.

Remarquons également que, dans le schéma original, une instance de E1 peut être liée à plusieurs instances de E2 (via ER). Il faut donc vérifier qu'une instance de E1 est référencée par une instance de ER au maximum. Si la requête ci-dessous donne un résultat, cela pose des problèmes. Il faut une intervention du concepteur.

```
SELECT Ae1 FROM ER GROUP BY Ae1 HAVING COUNT(Ae1) > 1;
```

S Création de E.Ae2 : création d'une clé étrangère et de ces colonnes de référence (page 82).

Transfert des instances de ER vers E1.Ae2 :

```
UPDATE E1 SET Ae2 = (SELECT Ae2 FROM ER WHERE E1.A1 = ER.Ae1);
```

Destruction de ER : voir la destruction d'une table (page 75).

Programmes

D Dépendance structurelle : il faut revoir les jointures et requêtes imbriquées sur la table ER pour en faire des jointures entre E1 et E2. Dans les interfaces, il faut enlever la gestion de la table ER. En général, il faut enlever toute la gestion concernant la table ER et revoir la gestion du lien entre E1 et E2.

L Recherche sur la table ER ainsi que sur toutes ses colonnes et les variables qui en dépendent.

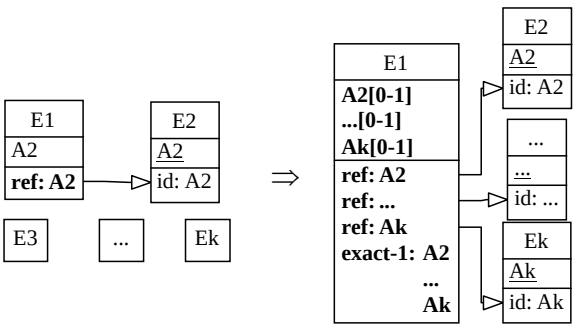
C.4.4.3 Changement de transformation sur les colonnes de référence

On va reprendre les changements de transformations du point C.4.3.3 f) pour les analyser en termes de conversion de structures et de modifications de programmes.

a) Transformation d'une clé étrangère en plusieurs clés étrangères

Données

D Indépendance : les instances du schéma original sont valables pour le schéma final.



I /

S Création de E1.A3, ..., E1.Ak : création d'une clé étrangère et de ses colonnes de référence (page 82).

Pas de transfert des instances.

Modification de E1.A2 : voir le changement d'une contrainte NOT NULL en NULL (page 78).

Programmes

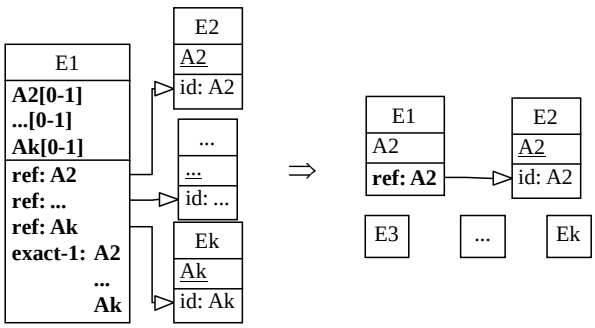
D Dépendance structurelle : puisque des valeurs nulles peuvent être introduites pour E1.A2, il faut ajouter des tests pour traiter ces valeurs dans les programmes sinon il y a des risques d'exécuter des opérations non conformes (comme par exemple la division par une valeur nulle, l'affichage de valeur nulle dans un écran de saisie, ...). L'interface doit être également revue pour accepter ces valeurs nulles. Il faut également ajouter la gestion des nouvelles clés étrangères définies sur A3, ..., Ak dans tous les programmes.

L Recherche sur la colonne E1.A2 ainsi que toutes les variables qui en dépendent.

b) Transformation de plusieurs clés étrangères en une clé étrangère

Données

D Dépendance : certaines instances sont perdues et la colonne A2 de E1 devient obligatoire ce qui peut poser des problèmes.



I Les instances de A3, ..., Ak vont être perdues. Le concepteur doit prendre en charge ce problème pour décider d'apliquer la modification.

La colonne A2 devient obligatoire. Pour chaque instance de E1, il faut une instance de A2. Ce problème a déjà été traité à la page 79 (création d'une contrainte NOT NULL sur une colonne).

S Modification de E1.A2 : voir le changement d'une contrainte NULL en NOT NULL (page 79).

Pas de transfert d'instances.

Destruction de E1.A3, ..., E1.Ak : voir la destruction d'une clé étrangère et de ses colonnes de référence (page 83).

Programmes

D Dépendance structurelle : il faut enlever la gestion des clés étrangères sur A3, ..., Ak dans tous les programmes.

L Recherche sur la colonne E1.A3, ..., E1.Ak ainsi que toutes les variables qui en dépendent.

ANNEXE D

TABLES DE COMPARAISONS ET DE TRADUCTIONS DES MODIFICATIONS

D.1 Tables de comparaisons des modifications

D.1.1 Introduction

Pour créer un historique simplifié HS, chaque modification devant être ajoutée à HS est comparée à toutes les modifications de HS. Cette section présente les tables de comparaisons des modifications. Il existe trois catégories de comparaisons pertinentes :

- les comparaisons de modifications relatives à un même objet,
- les comparaisons de modifications relatives à un objet et ses éléments,
- les comparaisons de modifications relatives à un objet et ses composants.

D.1.2 Table de comparaisons des modifications relatives à un même objet

Au niveau conceptuel, un type d'objets peut représenter un type d'entités, un type d'associations, un attribut, un groupe, un rôle ou une relation IS-A. Au niveau logique, il peut représenter une table, une collection, une colonne ou un groupe (clé étrangère, clé primaire ou candidate, index ou autres). La lettre O est utilisée pour représenter un type d'objets. Quatre grands types de modifications sont répertoriées : la suppression, la création, la modification et la transformation.

La table D.1 compare les modifications relatives à un même objet. Chaque cellule de la table est divisée en deux parties : la partie supérieure contient les deux modifications à comparer et la partie inférieure contient les modifications conservées dans l'historique simplifié. La première ligne de chaque partie concerne la modification présente dans HS, la deuxième ligne concerne la modification comparée à celle de HS pour être éventuellement ajoutées. Précisons encore que O' et O" désignent l'objet O qui a été modifié.

	Ajout HS	**TYPE DE MODIFICATION**			
		Supprimer	**Créer**	**Modifier**	**Transformer**
TYPE DE MODIFICATION	**Supprimer**	()←supprimer(O1) ()←supprimer(O2)	()←supprimer(O1) O2←creer()	()←supprimer(O1) O2'←modifier(O2)	()←supprimer(O1) O3←transfo(O2)
		()←supprimer(O1) ()←supprimer(O2)	()←supprimer(O1) O2←creer() **(1)**	()←supprimer(O1) O2'←modifier(O2)**(1)**	()←supprimer(O1) O3←transfo(O2)
	Créer	O←creer() ()←supprimer(O)	O1←creer() O2←creer()	O←creer() O'←modifier(O)	O←creer() O1←transfo(O)
		(2)	O1←creer() O2←creer()	O'←creer() **(3)**	O1←creer() **(4)**
	Modifier	O'←modifier(O) ()←supprimer(O')	O1'←modifier(O1) O2←creer()	O'←modifier(O) O"←modifier(O')	O'←modifier(O) O1←transfo(O')
		()←supprimer(O)**(5)**	O1'←modifier(O1) O2←creer()	O"←modifier(O) **(6)**	O1←transfo(O)**(7)**
	Transformer	O2←transfo(O1) ()←supprimer(O2)	O2←transfo(O1) O3←creer()	O2←transfo(O1) O2'←modifier(O2)	O2←transfo(O1) O3←transfo(O2)
		()←supprimer(O1)**(8)**	O2←transfo(O1) O3←creer()	O2'←transfo(O1) **(9)**	O2←transfo(O1) O3←transfo(O2)

Tableau D.1 - Table de comparaisons des modifications relatives à un même objet.

(1) Problème si nom(O2) = nom(O1) ou nom(O2') = nom(O1).

(2) La création et la suppression d'un même objet se neutralise dans HS.

(3) La création d'un objet, qui est modifié par la suite, est remplacé par la création de l'objet modifié.

(4) La création d'un objet qui est transformé par la suite est remplacée par la création des nouveaux objets créés par la transformation.

(5) Toutes les modifications, survenant à un objet qui est finalement supprimé, ne sont pas pertinentes, on supprime l'objet avant ses modifications.

(6) On ne garde dans HS que la dernière modification survenue à un objet.

(7) Si un objet est modifié puis ensuite transformé, la modification est enlevée de HS. On ajoute la transformation de l'ancien objet.

(8) Si l'objet résultat d'une transformation est supprimé, on enlève la transformation de HS et on ajoute la suppression de l'objet transformé.

(9) Si l'objet résultat d'une transformation est modifié, la modification n'est pas enregistrée.

Les autres comparaisons sont sans relation. La modification est ajoutée à HS.

D.1.3 Table de comparaisons des modifications relatives à un objet et ses éléments

La notion d'éléments englobe tous les composants d'un type d'objets dont l'existence dépend de l'existence du père. Par exemple, un attribut est un élément d'un type d'entités. Si le type d'entités est supprimé, l'attribut disparaît également.

Au niveau conceptuel, les types d'entités, les types d'associations et les attributs décomposables ont des éléments. Un type d'entités peut avoir des attributs, des groupes et des relations IS-A. Un type d'associations peut avoir des rôles, des attributs et des groupes. Un attribut décomposable peut avoir des sous-attributs et des groupes. Aux niveaux logique et physique, les tables ont des éléments. Une table peut contenir des colonnes et des groupes (clé étrangère, clé primaire ou candidate, index ou autres). On utilise la lettre O pour représenter l'objet parent et E pour ses éléments.

La table D.2 compare les modifications de HS relatives aux éléments d'un objet parent avec les modifications qui doivent être ajoutées à HS et sont relatives à cet objet parent. La table D.3 fait l'inverse.

Ajout HS		OBJET			
		Supprimer	**Créer**	**Modifier**	**Transformer**
ELEMENTS	**Supprimer**	()←supprimer(O,E) ()←supprimer(O)	()←supprimer(O1,E) O2←creer()	()←supprimer(O,E) O'←modifier(O)	()←supprimer(O1,E) O2←transfo(O1)
		()←supprimer(O)**(1)**	()←supprimer(O1,E) O2←creer()	()←supprimer(O,E) O'←modifier(O)**(2)**	O2←transfo(O1)**(3)**
	Créer	(O,E)←creer() ()←supprimer(O)	(O1,E)←creer() O2←creer()	(O,E)←creer() O'←modifier(O)	(O1,E)←creer() O2←transfo(O1)
		()←supprimer(O)**(1)**	(O1,E)←creer() O2←creer()	(O,E)←creer() O'←modifier(O) **(2)**	O2←transfo(O1)**(3)**
	Modifier	E'←modifier(O,E) ()←supprimer(O)	E'←modifier(O1,E) O2←creer()	E'←modifier(O,E) O'←modifier(O)	E'←modifier(O1,E) O2←transfo(O1)
		()←supprimer(O)**(1)**	E'←modifier(O1,E) O2←creer()	E'←modifier(O,E) O'←modifier(O) **(2)**	O2←transfo(O1)**(3)**
	Transformer	E2←transfo(O,E1) ()←supprimer(O)	E2←transfo(O1,E1) O2←creer()	E2←transfo(O,E1) O'←modifier(O)	E2←transfo(O1,E1) O2←transfo(O1)
		E2←transfo(O,E1) ()←supprimer(O)**(4)**	E2←transfo(O1,E1) O2←creer()	E2←transfo(O,E1) O'←modifier(O)**(2)**	E2←transfo(O1,E1) O2←transfo(O1)

Tableau D.2 - Table de comparaisons des modifications relatives à un élément et son objet parent.

(1) La suppression d'un parent rend inutile la suppression, la création ou la modification de ses éléments.

(2) Dans la deuxième phase de simplification, les modifications relatives aux attributs, aux groupes et aux relations IS-A disparaissent lorsque le type d'entités parent est modifié.

(3) Les opérations de suppression, création et modification d'un élément dont le parent est ensuite transformé sont enlevées de HS car l'enregistrement de la transformation dans HS permet de les retrouver en comparant les objets avant et après transformation. Nous verrons cela dans la traduction des modifications en modifications physiques (section D.2).

(4) Les opérations de transformation d'un élément disparaissent ou pas (en grisé dans la table) de HS si le parent est supprimé. Par exemple, si on transforme un attribut d'un type d'associations en un type d'entités, la suppression du type d'associations n'enlève pas la transformation de son attribut de HS. Par contre, si on désagrège un attribut décomposable dans un type d'associations, la suppression du type d'associations enlève la transformation de son attribut de HS.

Les autres comparaisons sont sans relation. La modification est ajoutée à HS.

Ajout HS		ELEMENTS			
		Supprimer	**Créer**	**Modifier**	**Transformer**
OBJET	**Supprimer**	()←supprimer(O1) ()←supprimer(O2,E)	()←supprimer(O1) (O2,E)←creer()	()←supprimer(O1) E'←modifier(O2,E)	()←supprimer(O1) E2←transfo(O2,E1)
		()←supprimer(O1) ()←supprimer(O2,E)	()←supprimer(O1) (O2,E)←creer()	()←supprimer(O1) E'←modifier(O2,E)	()←supprimer(O1) E2←transfo(O2,E1)
	Créer	O←creer() ()←supprimer(O,E)	O←creer() (O,E)←creer()	O←creer() E'←modifier(O,E)	O←creer() E2←transfo(O,E1)
		O←creer() **(1)**	O←creer() **(1)**	O←creer() **(1)**	O←creer() (O,E2)←creer()**(2)**
	Modifier	O'←modifier(O) ()←supprimer(O',E)	O'←modifier(O) (O',E)←creer()	O'←modifier(O) E'←modifier(O',E)	O'←modifier(O) E2←transfo(O',E1)
		O'←modifier(O) ()←supprimer(O',E)	O'←modifier(O) (O',E)←creer() **(3)**	O'←modifier(O) E'←modifier(O',E)**(3)**	O'←modifier(O) E2←transfo(O',E1)
	Transformer	O2←transfo(O1) ()←supprimer(O2,E)	O2←transfo(O1) (O2,E)←creer()	O2←transfo(O1) E'←modifier(O2,E)	O2←transfo(O1) E2←transfo(O2,E1)
		O2←transfo(O1) **(4)**	O2←transfo(O1) **(4)**	O2←transfo(O1) **(4)**	O2←transfo(O1) E2←transfo(O2,E1)

Tableau D.3 - Table de comparaisons des modifications relatives à un objet parent et ses éléments.

(1) La création d'un objet rend inutile la suppression, la création ou la modification d'un de ses éléments puisqu'on créera l'objet dans son état final.

(2) Si la transformation enlève l'appartenance de l'élément à son parent, la création de la nouvelle structure de la transformation est ajoutée à HS sinon on n'ajoute pas la transformation à HS.

(3) Dans la deuxième phase de simplification, les modifications relatives aux attributs, aux groupes et aux relations IS-A disparaissent lorsque le type d'entités parent est modifié.

(4) Les opérations de suppression, création et modification d'un élément dont le parent a déjà été transformé ne sont pas mises dans HS car l'enregistrement de la transformation dans HS permet de les retrouver en comparant les objets avant et après la transformation. Nous verrons cela dans la traduction des modifications en modifications physiques (section D.2).

Les autres comparaisons sont sans relation. La modification est ajoutée à HS.

D.1.4 Table de comparaisons des modifications relatives à un objet et ses composants

La notion de composants englobe tous les composants d'un type d'objets dont l'existence ne dépend pas de l'existence du type d'objets composé. Par exemple, un attribut est un composant d'un groupe. Si le groupe est supprimé, l'attribut ne subit aucune modification.

Au niveau conceptuel, les groupes ont des composants. Un groupe peut contenir des attributs et des rôles. Au niveau physique, les collections et les groupes ont des composants. Une collection contient des tables et un groupe contient des colonnes. On utilise la lettre O pour représenter le type d'objets composé et C pour ses composants.

La table D.4 compare les modifications relatives à un objet et ses composants. La table D.5 fait l'inverse. On constate que toutes les comparaisons sont sans relation. Les modifications sont donc ajoutées à HS.

HS \ Ajout		COMPOSANTS			
		Supprimer	**Créer**	**Modifier**	**Transformer**
OBJET	**Supprimer**	()←supprimer(O) ()←supprimer(C)	()←supprimer(O) C←creer()	()←supprimer(O) C'←modifier(C)	()←supprimer(O) C2←transfo(C1)
		()←supprimer(O) ()←supprimer(C)	()←supprimer(O) C←creer()	()←supprimer(O) C'←modifier(C)	()←supprimer(O) C2←transfo(C1)
	Créer	O←creer() ()←supprimer(C)	O←creer() C←creer()	O←creer() C'←modifier(C)	O←creer() C2←transfo(C1)
		O←creer() ()←supprimer(C)	O←creer() C←creer()	O←creer() C'←modifier(C)	O←creer() C2←transfo(C1)
	Modifier	O'←modifier(O) ()←supprimer(C)	O'←modifier(O) C←creer()	O'←modifier(O) C'←modifier(C)	O'←modifier(O) C2←transfo(C1)
		O'←modifier(O) ()←supprimer(C)	O'←modifier(O) C←creer()	O'←modifier(O) C'←modifier(C)	O'←modifier(O) C2←transfo(C1)
	Transformer	O2←transfo(O1) ()←supprimer(C)	O2←transfo(O1) C←creer()	O2←transfo(O1) C'←modifier(C)	O2←transfo(O1) C2←transfo(C1)
		O2←transfo(O1) ()←supprimer(C)	O2←transfo(O1) C←creer()	O2←transfo(O1) C'←modifier(C)	O2←transfo(O1) C2←transfo(C1)

Tableau D.4 - Table de comparaisons des modifications relatives à un objet et ses composants.

HS \ Ajout		OBJET			
		Supprimer	**Créer**	**Modifier**	**Transformer**
COMPOSANTS	**Supprimer**	()←supprimer(C) ()←supprimer(O)	()←supprimer(C) O←creer()	()←supprimer(C) O'←modifier(O)	()←supprimer(C) O2←transfo(O1)
		()←supprimer(C) ()←supprimer(O)	()←supprimer(C) O←creer()	()←supprimer(C) O'←modifier(O)	()←supprimer(C) O2←transfo(O1)
	Créer	C←creer() ()←supprimer(O)	C←creer() O←creer()	C←creer() O'←modifier(O)	C←creer() O2←transfo(O1)
		C←creer() ()←supprimer(O)	C←creer() O←creer()	C←creer() O'←modifier(O)	C←creer() O2←transfo(O1)
	modifier	C'←modifier(C) ()←supprimer(O)	C'←modifier(C) O←creer()	C'←modifier(C) O'←modifier(O)	C'←modifier(C) O2←transfo(O1)
		C'←modifier(C) ()←supprimer(O)	C'←modifier(C) O←creer()	C'←modifier(C) O'←modifier(O)	C'←modifier(C) O2←transfo(O1)
	Transformer	C2←transfo(C1) ()←supprimer(O)	C2←transfo(C1) O←creer()	C2←transfo(C1) O'←modifier(O)	C2←transfo(C1) O2←transfo(O1)
		C2←transfo(C1) ()←supprimer(O)	C2←transfo(C1) O←creer()	C2←transfo(C1) O'←modifier(O)	C2←transfo(C1) O2←transfo(O1)

Tableau D.5 - Table de comparaisons des modifications relatives à un composant et son objet composé.

D.2 Tables de traductions des modifications

D.2.1 Introduction

Les modifications conceptuelles ou logiques de l'historique d'évolution doivent être traduites en modifications physiques. Cette section présente les tables de traductions des modifications. Les transformations de création, de suppression et de modification sont étudiées séparément.

D.2.2 Création

Il faut regarder dans H_{C1} les transformations appliquées sur l'objet créé dans H_{E1}. Si elles existent, on remplace la modification conceptuelle de création par des modifications de création des nouveaux objets dans lesquelles l'objet conceptuel a été transformé. Si elles n'existent pas, on remplace l'objet conceptuel par l'objet physique correspondant. La correspondance entre les objets des différents niveaux est assurée par le nom des objets. Par exemple, si on a la création d'un type d'associations dans H_{E1}, on ajoute la création d'une colonne de référence et d'une clé étrangère dans H_{EP1}.

La table D.6 donne les traductions des créations conceptuelles de H_{E1} en créations physiques de H_{EP1}.

Transformations d'évolution (H_{E1})	**Transformations de conception (H_{C1})**	**Transformations physiques (H_{EP1})**
E←creer(S,n)	/	E←creer(S,n)
	E'←modifier(E,n')	E←creer(S,n')
	(E,E1)←eclater-TE(E,{...})	E←creer(S,n) E1←creer(S,n1)
I←creer(E,n)	(E,{R})←ISA-en-TA(E,{E1}) (E1,{A1,...,An})←TA-en-FK(R,E1)	A1←creer(E1,n1), ..., An←creer(E1,nn) FK←creer(E1,nfk)
R←creer(S,n)	(E1,{A1,...,An})←TA-en-FK(R,E1)	A1←creer(E1,n1), ..., An←creer(E1,nn) FK←creer(E1,nfk)
	ER←TA-en-TE(R) (ER,{A11,...})←TA-en-FK(R1,ER) ... (ER,{An1,...})←TA-en-FK(Rn,ER)	ER←creer(S,ner)
Ro←creer(R,n)	ER←TA-en-TE(R) (ER,{A11,...})←TA-en-FK(R1,ER) ... (ER,{An1,...})←TA-en-FK(Rn,ER) (ER,{Aro1,...})←TA-en-FK(Ro,ER)	Aro1←creer(ER,naro1), ..., Arom←creer(ER,narom) FK←creer(ER,nfkro)

Tableau D.6 - Table de traductions des modifications de création de H_{E1}.

Transformations d'évolution (H_{E1})	**Transformations de conception (H_{C1})**	**Transformations physiques (H_{EP1})**
A←creer(O,n) (1)	/	A←creer(E,na)
	A'←modifier(A)	A'←creer(E,na)
	(EA,R)←Att-en-TE-inst(E,A) (EA,{A1,...,An})←TA-en-FK(R,EA)	EA←creer(S,nea)
	(EA,R)←Att-en-ET-val(E,A) (ER)←TA-en-TE(R) (ER,{A11,...})←TA-en-FK(R1,ER) ... (ER,{An1,...})←TA-en-FK(Rn,ER)	EA←creer(S,nea) ER←creer(S,ner)
	(E,{A1,...,An})←desagreger-Att(E,A)	A1←creer(E,na1), ..., An←creer(E,nan)
	(E,{A1,...,An})←Attmult-en-Serie-Att(E,A)	A1←creer(E,na1), ..., An←creer(E,nan)
	(E,As)←Attmult-en-Attmono(E,A)	As←creer(E,nas)
G←creer(O,n) (2)	/	G←creer(E,ng)
	G'←modifier(G)	G'←creer(E,ng)

Tableau D.6 - Table de traductions des modifications de création de H_{E1}.

(1) Le sigle O représente soit un type d'entités, soit un type d'associations, soit un attribut décomposable. Dans le cas d'un type d'entités, O est identique à E. Pour un type d'associations, E est le type d'entités dans lequel le type d'associations a été transformé. Pour un attribut décomposable, E représente soit le type d'entités dans lequel il a été transformé (transformation d'un attribut en un type d'entités par instance ou par valeur), soit le parent de l'attribut désagrégé. Remarquons que l'ajout de rôles ou d'attributs à un type d'associations binaire fonctionnel et que l'ajout d'un attribut à un attribut atomique nécessitent la transformation du type d'associations ou de l'attribut. Dans ce cas, l'ajout de composant a modifié la structure du composant existant. La modification d'un type d'associations ou d'un attribut est traitée au point D.2.4.

(2) Le sigle O représente soit un type d'entités, soit un type d'associations. Dans le cas d'un type d'entités, O est identique à E. Pour un type d'associations, E est soit le type d'entités dans lequel le type d'associations O a été transformé, soit le type d'entités contenant les attribut de référence (transformation d'un type d'associations en une clé étrangère).

D.2.3 Suppression

Il faut regarder dans H_{C0} les transformations qui avaient été appliquées sur l'objet supprimé. Si elles existent, on remplace la modification conceptuelle de suppression par des modifications de suppression des anciens objets qui représentaient au niveau physique l'objet conceptuel supprimé. Si elles n'existent pas, on remplace l'objet par l'objet du niveau physique (correspondance par nom). Par exemple, si on a la suppression d'un type d'associations dans H_{E1}, on le remplace par la suppression d'une colonne de référence et d'une clé étrangère dans H_{EP1}.

Le tableau D.7 donne les traductions des suppressions conceptuelles de H_{E1} en suppressions physiques de H_{EP1}.

Transformations d'évolution E1	Transformations de conception C0	Transformations physiques EP1
()←supprimer(E)	/	()←supprimer(E)
	E'←modifier(E,n')	()←supprimer(E)
	(E,E1)←eclater-TE(E,{...})	()←supprimer(E) ()←supprimer(E1)
()←supprimer(I)	(E,{R})←ISA-en-TA(E,{E1}) (E1,{A1,...,An})←TA-en-FK(R,E1)	()←supprimer(A1), ..., ()←supprimer(An) ()←supprimer(FK)
()←supprimer(R)	(E1,{A1,...,An})←TA-en-FK(R,E1)	()←supprimer(A1), ..., ()←supprimer(An) ()←supprimer(FK)
	ER←TA-en-TE(R) (ER,{A11,...})←TA-en-FK(R1,ER) ... (ER,{An1,...})←TA-en-FK(Rn,ER)	()←supprimer(ER)
()←supprimer(Ro)	ER←TA-en-TE(R) (ER,{A11,...})←TA-en-FK(R1,ER) ... (ER,{An1,...})←TA-en-FK(Rn,ER) (ER,{Aro1,...})←TA-en-FK(Ro,ER)	()←supprimer(Aro1), ..., ()←supprimer(Arom) ()←supprimer(FK)
()←supprimer(A) (1)	/	()←supprimer(A)
	A'←modifier(A)	()←supprimer(A')
	(EA,R)←Att-en-TE-inst(E,A) (EA,{A1,...,An})←TA-en-FK(R,EA)	()←supprimer(EA)
	(EA,R)←Att-en-ET-val(E,A) (ER)←TA-en-TE(R) (ER,{A11,...})←TA-en-FK(R1,ER) ... (ER,{An1,...})←TA-en-FK(Rn,ER)	()←supprimer(EA) ()←supprimer(ER)
	(E,{A1,...,An})←desagreger-Att(E,A)	()←supprimer(A1), ..., ()←supprimer(An)
	(E,{A1,...,An})←Attmult-en-Serie-Att(E,A)	()←supprimer(A1), ..., ()←supprimer(An)
	(E,As)←Attmult-en-Attmono(E,A)	()←supprimer(As)
()←supprimer(G)	/	()←supprimer(G)
	G'←modifier(G)	()←supprimer(G')

Tableau D.7 - Table de traductions des modifications de suppression de H_{E1}.

(1) L'attribut supprimé appartient à un type d'entités, à un type d'associations ou à un attribut décomposable. Pour un type d'associations, E est le type d'entités dans lequel le type d'associations a été transformé. Pour un attribut décomposable, E représente soit le type d'entités dans lequel il a été transformé (transformation d'un attribut en un type d'entités par instance ou par valeur), soit le parent de l'attribut désagrégé. Remarquons que la suppression de rôles ou d'attributs d'un type d'associations complexe (non binaire, non fonctionnel) ou la suppression d'attribut d'un attribut décomposable peut rendre inutile la transformation en un type d'entités du type d'associations ou la désagrégation ou la transformation en un type d'entités (par instance ou par valeur) dans les modification de conception. Dans ce cas, le retrait d'un composant a modifié la structure du composant existant. La modification du type d'associations ou d'un attribut décomposable est traitée dans le point D.2.4.

D.2.4 Modification

Il faut regarder dans H_{C0} les transformations qui avaient été appliquées sur l'objet (on les nomme T_{C0}). Ensuite, on regarde les transformations appliquées sur l'objet modifié dans H_{C1} (on les nomme T_{C1}). En fonction de T_{C0} et T_{C1}, on remplace la modification dans la table soit par une

transformation de structure physique soit par une modification physique. Dans le cas où on doit transformer des structures physiques, on commence par créer les nouvelles structures, ensuite on convertit les données de l'ancienne structure dans la nouvelle et, finalement, les anciennes structures sont supprimées. Si ($T_{C0} \neq \emptyset$ et $T_{C1} = \emptyset$) ou ($T_{C0} = \emptyset$ et $T_{C1} \neq \emptyset$) ou ($T_{C0} \neq \emptyset$ et $T_{C1} \neq \emptyset$ et $T_{C0} \neq T_{C1}$), alors il s'agit d'une transformation de structure physique. Sinon, on a une modification physique.

D.2.4.1 Modification d'un type d'entités

Les seuls modifications possibles sur un type d'entités sont le renommage. Si on change de transformation lors de la conception sans modifier le type d'entités, le résultat est le même avec E' = E.

Le tableau D.8 donne les traductions des modifications relatives à un type d'entités en modifications physiques de H_{EP1}.

Transformations de conception H_{C0}	**Transformations de conception H_{C1}**	**Transformations physiques H_{EP1}**
/	/	E'←modifier(E)
	E"←modifier(E')	E"←modifier(E)
	(E',E1)←eclater-TE(E')	E1←Créer() Trf_donnees(E,E1,A1,...,An) E'←modifier(E) ()←supprimer(E',A1),...,()←supprimer(E',An)
E"←modifier(E)	/	E'←modifier(E")
	E"'←modifier(E')	E"'←modifier(E")
	(E',E1)←eclater-TE(E')	E1←Créer() Trf_donnees(E",E1,A1,...,An) E'←modifier(E") ()←supprimer(E',A1),...,()←supprimer(E',An)
(E,E1)←eclater-TE(E)	/	Trf_donnees(E1,E,A1,...,An) ()←supprimer(E1)
	E"←modifier(E')	E"←modifier(E) Trf_donnees(E1,E",A1,...,An) ()←supprimer(E1)
	(E',E1)←eclater-TE(E') **(1)**	E'←modifier(E) Trf_donnees(E1,E',A1,...,An) Trf_donnees(E',E1,An+1,...,Am)

Tableau D.8 - Traduction des modifications relatives à un type d'entités en modifications physiques.

(1) Si les transformations d'éclatement ne sont pas les mêmes, il faut rapatrier des attributs de E1 vers E' et transférer des attributs de E' vers E1.

D.2.4.2 Modification d'un type d'associations

Les modifications possibles sur un type d'associations sont le renommage (1 et 2) ou le changement de transformations dû à un choix du concepteur, à l'ajout ou à la suppression d'un attribut ou d'un rôle (3 et 4).

Le tableau D.9 donne les traductions des modifications relatives à un type d'associations en modifications physiques de H_{EP1}.

	Transformations de conception H_{C0}	Transformations de conception H_{C1}	Transformations physiques H_{EP1}
1.	(A1,...,An)←TA-en-FK(R)	(A1',...,An')←TA-en-FK(R')	A1'←modifier(A1),...,An'←modifier(An)
2.	(E,R1,...,Rn)←TA-en-ET(R) (E,A11,...)←TA-en-FK(R1) ... (E,An1,...)←TA-en-FK(Rn)	(E',R1',...,Rn')←TA-en-ET(R') (E',A11',...)←TA-en-FK(R1') ... (E',An1',...)←TA-en-FK(R2')	E'←modifier(E) A11'←modifier(E',A11),... ... An1'←modifier(E',An1)
3.	(A1,...,An)←TA-en-FK(R)	(E,R1,...,Rn)←TA-en-ET(R) (E,A11,...)←TA-en-FK(R1) ... (E,A1n,...)←TA-en-FK(Rn)	E←creer() Trf_donnees(A1,...,An,E) ()←supprimer(E1,A1),...,()←supprimer(E1,An)
4.	(E,R1,...,Rn)←TA-en-ET(R) (E,A11,...)←TA-en-FK(R1) ... (E,An1,...)←TA-en-FK(Rn)	(A1,...,An)←TA-en-FK(R)	(E1,A1)←creer(),...,(E1,An)←creer() Trf_donnees(E,E1,A1,...,An) ()←supprimer(E)

Tableau D.9 - Traduction des modifications relatives à un type d'associations en modifications physiques.

D.2.4.3 Modification d'un rôle

Si on choisit de changer de transformation, on se retrouve dans le point D.2.4.2.

Le tableau D.10 donne les traductions des modifications relatives à un rôle en modifications physiques de H_{EP1}.

Transformations de conception H_{C0}	Transformations de conception H_{C1}	Transformations physiques H_{EP1}
(A1,...,An)←TA-en-FK(R)	(A1,...,An)←TA-en-FK(R)	FK'←modifier(E,FK)
(E,R1,...,Rn)←TA-en-ET(R) (E,A11,...)←TA-en-FK(R1) ... (E,An1,...)←TA-en-FK(Rn)	(E,R1,...,Rn)←TA-en-ET(R) (E,A11,...)←TA-en-FK(R1) ... (E,An1,...)←TA-en-FK(Rn)	FK'←modifier(E,FK)

Tableau D.10 - Traduction des modifications relatives à un rôle en modifications physiques.

D.2.4.4 Modification d'un attribut

L'objet E représente soit un type d'entités, soit un type d'associations, soit un attribut décomposable. La modification d'un attribut peut porter sur son nom, son type, sa longueur, ses cardinalités, ... Certaines modifications peuvent entraîner le changement de transformation au niveau physique. Comme pour les types d'associations, le concepteur peut envisager de changer de représentation physique pour un attribut.

Le tableau D.11 donne les traductions des modifications relatives à un attribut en modifications physiques de H_{EP1}.

Transformations de conception H_{C0}	**Transformations de conception H_{C1}**	**Transformations physiques H_{EP1}**
/	/	A'←modifier(E,A)
	A"←modifier(E,A')	A"←modifier(E,A)
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←creer() Trf_donnees(A,EA') ()←supprimer(E,A)
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	EA'←creer() E1←creer() Trf_donnees(A,EA') ()←supprimer(E,A)
	(A1,...,An)←desagreger-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(A,A1,...,An) ()←supprimer(E,A)
	(A1,...,An)←Attmult-en-Serie-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(A,A1,...,An) ()←supprimer(E,A)
	As←Attmult-en-Attmono(E,A')	(E,As)←creer() Trf_donnees(A,As) ()←supprimer(E,A)
A"←modifier(E,A)	/	A'←modifier(E,A")
	A'"←modifier(E,A')	A'"←modifier(E,A")
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←creer() Trf_donnees(A",EA') ()←supprimer(E,A")
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	EA'←creer() E1←creer() Trf_donnees(A",EA') ()←supprimer(E,A")
	(A1,...,An)←desagreger-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(A",A1,...,An) ()←supprimer(E,A")
	(A1,...,An)←Attmult-en-Serie-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(A",A1,...,An) ()←supprimer(E,A")
	As←Attmult-en-Attmono(E,A')	(E,As)←creer() Trf_donnees(A",As) ()←supprimer(E,A")

Tableau D.11 - Traduction des modifications relatives à un attribut en modifications physiques.

Transformations de conception H_{C0}	**Transformations de conception H_{C1}**	**Transformations physiques H_{EP1}**
(EA,R)←Att-en-TE-inst(E,A) (EA,A1,...,An)←TA-en-FK(R)	/	(E,A')←creer() Trf_donnees(EA,A') ()←supprimer(EA)
	A"←modifier(E,A')	(E,A")←creer() Trf_donnees(EA,A") ()←supprimer(EA)
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←modifier(EA)
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	EA'←creer() E1←creer() Trf_donnees(EA,EA') ()←supprimer(EA)
	(A1,...,An)←desagreger-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(EA,A1,...,An) ()←supprimer(EA)
	(A1,...,An)←Attmult-en-Serie-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(EA,A1,...,An) ()←supprimer(EA)
	As←Attmult-en-Attmono(E,A')	(E,As)←creer() Trf_donnees(EA,As) ()←supprimer(EA)
(EA,R)←Att-en-ET-val(E,A) (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	/	(E,A')←creer() Trf_donnees(EA,A') ()←supprimer(EA) ()←supprimer(E1)
	A"←modifier(E,A')	(E,A")←creer() Trf_donnees(EA,A") ()←supprimer(EA) ()←supprimer(E1)
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←creer() Trf_donnees(EA,EA') ()←supprimer(EA) ()←supprimer(E1)
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	EA'←modifier(EA)
	(A1,...,An)←desagreger-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(EA,A1,...,An) ()←supprimer(EA) ()←supprimer(E1)
	(A1,...,An)←Attmult-en-Serie-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(EA,A1,...,An) ()←supprimer(EA) ()←supprimer(E1)
	As←Attmult-en-Attmono(E,A')	(E,As)←creer() Trf_donnees(EA,As) ()←supprimer(EA) ()←supprimer(E1)

Tableau D.11 - Traduction des modifications relatives à un attribut en modifications physiques.

Transformations de conception H_{C0}	**Transformations de conception H_{C1}**	**Transformations physiques H_{EP1}**
(A1,...,An)←desagreger-Att(E,A)	/	(E,A')←creer() Trf_donnees(A1,...,An,A') ()←supprimer(A1),...,()←supprimer(An)
	A"←modifier(E,A')	(E,A")←creer() Trf_donnees(A1,...,An,A") ()←supprimer(A1),...,()←supprimer(An)
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←creer() Trf_donnees(A1,...,An,EA') ()←supprimer(A1),...,()←supprimer(An)
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	EA'←creer() E1←creer() Trf_donnees(A1,...,An,EA') ()←supprimer(A1),...,()←supprimer(An)
	(A1',...,A2')←desagreger-Att(E,A')	A1'←modifier(E,A1),...,An'←modifier(E,An)
	(A1',...,An')←Attmult-en-Serie-Att(E,A')	(E,A1')←creer(),...,(E,An')←creer() Trf_don(A1,...,An,A1',...,An') ()←supprimer(A1),...,()←supprimer(An)
	As←Attmult-en-Attmono(E,A')	(E,As)←creer() Trf_donnees(A1,...,An,As) ()←supprimer(A1),...,()←supprimer(An)
(A1,...,An)←Attmult-en-Serie-Att(E,A)	/	(E,A')←creer() Trf_donnees(A1,...,An,A') ()←supprimer(A1),...,()←supprimer(An)
	A"←modifier(E,A')	(E,A")←creer() Trf_donnees(A1,...,An,A") ()←supprimer(A1),..,()←supprimer(An)
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←creer() Trf_donnees(A1,...,An,EA') ()←supprimer(A1),...,()←supprimer(An)
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A22,...)←TA-en-FK(R2)	EA'←creer() E1←creer() Trf_donnees(A1,...,An,EA') ()←supprimer(A1),...,()←supprimer(An)
	(A1',...,An')←desagreger-Att(E,A')	(E,A1')←creer(),...,(E,An')←creer() Trf_don(A1,...,An,A1',...,An') ()←supprimer(A1),...,()←supprimer(An)
	(A1',...,An')←Attmult-en-Serie-Att(E,A')	A1'←modifier(E,A1),...,An'←modifier(E,An)
	As←Attmult-en-Attmono(E,A')	(E,As)←creer() Trf_donnees(A1,...,An,As) ()←supprimer(A1),...,()←supprimer(An)

Tableau D.11 - Traduction des modifications relatives à un attribut en modifications physiques.

Transformations de conception H_{C0}	Transformations de conception H_{C1}	Transformations physiques H_{EP1}
As←Attmult-en-Att-mono(E,A)	/	(E,A')←creer() Trf_donnees(As,A') ()←supprimer(As)
	A"←modifier(E,A')	(E,A")←creer() Trf_donnees(As,A') ()←supprimer(As)
	(EA',R)←Att-enTE-inst(E,A') (EA',A1,...,An)←TA-en-FK(R)	EA'←creer() Trf_donnees(As,EA') ()←supprimer(As)
	(EA',R)←Att-en-ET-val(E,A') (E1,R1,R2)←TA-en-ET(R) (E1,A11,...)←TA-en-FK(R1) (E1,A21,...)←TA-en-FK(R2)	EA'←creer() E1←creer() Trf_donnees(As,EA') ()←supprimer(As)
	(A1,...,An)←desagreger-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(As,A1,...,An) ()←supprimer(As)
	(A1,...,An)←Attmult-en-Serie-Att(E,A')	(E,A1)←creer(),...,(E,An)←creer() Trf_donnees(As,A1,...,An) ()←supprimer(As)
	As'←Attmult-en-Attmono(E,A')	As'←modifier(As)

Tableau D.11 - Traduction des modifications relatives à un attribut en modifications physiques.

D.2.4.5 Modification d'un groupe

Le groupe détruit appartient à un type d'entités ou à un type d'associations. Pour un type d'associations, E est soit le type d'entités dans lequel le type d'associations a été transformé soit le type d'entités contenant les attribut de référence (transformation d'un type d'associations en une clé étrangère).

Le tableau D.12 donne les traductions des modifications relatives à un groupe en modifications physiques de H_{EP1}.

Transformations de conception H_{C0}	Transformations de conception H_{C1}	Transformations physiques H_{EP1}
/	/	G'←modifier(E,G)
G"←modifier(E,G)	G"'←modifier(E,G')	G"'←modifier(E,G")
/	G"←modifier(E,G')	G"←modifier(E,G)
G"←modifier(E,G)	/	G'←modifier(G")

Tableau D.12 - Traduction des modifications relatives à un groupe en modifications physiques.

ANNEXE E

EXEMPLES DE MÉTHODE ET DE JOURNAL

E.1 Exemple de méthode définie en MDL

Le code ci-dessous décrit, dans le langage MDL, la méthode définie pour la première stratégie (modification des spécifications conceptuelles) au point 7.2.2.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Définitions des modèles de produit %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

text-model FICHIER_TEXTE
  title "Fichier texte"
  description
    Un fichier texte contient du texte libre. Dans cette méthode, il est utilisé pour
    stocker des rapports écrits en langage naturel.
  end-description
  extensions "TXT"
end-model

text-model FICHIER_SQL
  title "Fichier SQL"
  description
    Un scriptSQL contenant des instructions SQL pour la création ou la modification des
    structures d'une base de données relationnelle.
  end-description
  extensions "SQL", "DDL"
end-model

text-model FICHIER_LOG_EVOL
  title "Journal pour l'évolution"
  description
    Un journal pour l'évolution contient l'historique des opérations pour
    l'évolution conceptuelle, logique ou physique d'un schéma conceptuel, logique ou
    physique.
  end-description
  extensions "LOG"
end-model

text-model FICHIER_LOG_LOG
  title "Journal pour la conception logique"
  description
    Un journal pour la conception logique contient l'historique des opérations
    pour la conception logique d'un schéma conceptuel.
  end-description
  extensions "LOG"
end-model

text-model FICHIER_LOG_PHYS
  title "Journal pour la conception physique"
  description
    Un journal pour la conception physique contient l'historique des opérations
    pour la conception physique d'un schéma logique.
  end-description
  extensions "LOG"
end-model

schema-model SCHEMA_PHYS_SQL
  title "Schéma physique relationnel"
  description
    Le modèle schéma physique relationnel fait correspondre le modèle Entité/association
    générique de DB-Main au modèle relationnel SQL, incluant des caractéristiques
    physiques telles que les index et les espaces de stockage. C'est de ce modèle que
    les scripts de création ou de conversion de la base de données sont dérivés.
```

```
  end-description
  concepts
    collection         "table space"
    schema             "view"
    entity_type        "table"
    atomic_attribute   "column"
    user_constraint    "constraint"
    identifier         "unique constraint"
    primary_identifier "primary key"
    access_key         "index"
  constraints
    ET_per_SCHEMA (1 N)     % Au moins une table requise
      diagnosis "Le schéma &NAME doit avoir au moins une table"
    RT_per_SCHEMA (0 0)     % Type d'associations non autorisé
      diagnosis "Le type d'associations &NAME ne peut pas exister"
    ATT_per_ET (1 N)        % Au moins une colonne par table
      diagnosis "La table &NAME doit avoir au moins une colonne"
    PID_per_ET (0 1)        % Au plus un identifiant primaire par table
      diagnosis "La table &NAME a trop de clés primaires"
    SUB_TYPES_per_ISA (0 0) % Relation Is-a non autorisées
      diagnosis "Les relations Is-a ne sont pas autorisées et &NAME a un sous-type"
    ID_NOT_KEY_per_ET (0 0) % Chaque contraint unique est un index
      diagnosis "La contrainte unique &NAME doit être un index"
    OPT_ATT_per_EPID (0 0)
         % Colonnes facultatives non autorisées dans les identifiants primaires.
      diagnosis "Il ne peut y avoir de colonnes facultative dans la clé primaire &NAME."
    DEPTH_of_ATT (1 1) and MAX_CARD_of_ATT (1 1) % Colonnes atomique et monovaluées
      diagnosis "La colonne &NAME doit être atomique et monovaluée."
    ALL_CHARS_in_LIST_NAMES
      (ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789$_)
      and NONE_in_LIST_NAMES (_$,$$)
      and LENGTH_of_NAMES (0 31)
      and NONE_in_FILE_CI_NAMES (PHYSRDB.NAM)
      diagnosis "Le nom &NAME est interdit"
end-model

schema-model SCHEMA_LOG_SQL
  title "Schéma logique relationnel"
  description
    Le modèle du schéma logique relationnel fait correspondre le modèle E/A générique de
    DB-Main au modèle générique relationnel.
  end-description
  concepts
    schema             "view"
    entity_type        "table"
    atomic_attribute   "column"
    user_constraint    "constraint"
    identifier         "unique constraint"
    primary_identifier "primary key"
  constraints
    ET_per_SCHEMA (1 N)     % Au moins une table requise
      diagnosis "Le schéma &NAME doit avoir au moins une table."
    RT_per_SCHEMA (0 0)     % Aucun type d'associations autorisé
      diagnosis "Le type d'associations &NAME ne peut pas exister."
    COLL_per_SCHEMA (0 0)   % Collections non autoriséez
      diagnosis "Le schéma ne peut pas avoir des collections."
    ATT_per_ET (1 N)        % Au moins une colonne par table
      diagnosis "La table &NAME doit avoir au moins une colonne."
    PID_per_ET (0 1)        % Au plus un identifiant primaire par table
      diagnosis "La table &NAME a plus d'un identifiant primaire."
    KEY_per_ET (0 0)        % Index ou clés d'accès non autorisés
      diagnosis "La table &NAME ne peut pas avoir d'index."
    SUB_TYPES_per_ISA (0 0) % Relations Is-a non autorisées
      diagnosis "Les relations Is-a ne sont pas autorisées and la table &NAME a un
```

```
                sous-type."
    OPT_ATT_per_EPID (0 0)
      % Colonnes facultatives non autorisées dans les identifiants primaires
      diagnosis "Il ne peut y avoir aucune colonne facultative dans l'identifiant
                primaire &NAME."
    DEPTH_of_ATT (1 1) and MAX_CARD_of_ATT (1 1) % Colonnes atomiques et monovaluées
      diagnosis "La colonne &NAME doit être atomique et monovaluée."
end-model

schema-model SCHEMA_CONCEPTUEL
  title "Schéma conceptuel"
  description
    Le modèle du schéma conceptuel permet de représenter le monde réel. Un schéma
    conforme à ce modèle montre la sémantique précise de la base de données. Il ne peut
    être directement implémenté et constitue surtout une documentation de la base de
    données et un support pour le dialogue avec les utilisateurs du système
    d'information.
  end-description
  concepts
    schema             "schema"
    entity_type        "entity type"
    rel_type           "relationship type"
    atomic_attribute   "attribute"
    compound_attribute "compound attribute"
    role               "role"
    group              "group"
    user_constraint    "constraint"
  constraints
    ET_per_SCHEMA (1 N)        % Au moins un type d'entités
      diagnosis "Le schéma &NAME doit avoir au moins un type d'entités."
    COLL_per_SCHEMA (0 0)      % Collections non autorisées
      diagnosis "Le schéma ne peut contenir des collections."
    ATT_per_ET (1 N)           % Au moins un attribut par type d'entités
      diagnosis "Le type d'entités &NAME doit avoir au moins un attribut."
    KEY_per_ET (0 0)           % Pas de clé d'accès
      diagnosis "Le type d'entités &NAME ne peut avoir des clés d'accès."
    REF_per_ET (0 0)           % Pas de contrainte référentielle
      diagnosis "Le type d'entités &NAME ne peut pas avoir de contrainte référentielle"
    ID_per_ET (1 N)            % S'il y a des identifiants, au moins un est primaire
        and PID_per_ET (1 1)
      or ID_per_ET (0 0)
      diagnosis "Un des identifiants du type d'entités &NAME doit être primaire."
    EMBEDDED_ID_per_ET (0 0)   % Identifiants emboîtés non autorisés
      diagnosis "Les identifiants emboîtés doivent être enlevés dans le type d'entités
                &NAME."
    ID_DIFF_in_ET (components) % Tous les identifiants doivent avoir des composants
                               % différents
      diagnosis "Les identifiants construits à partir de composants identiques doivent
être évités dans &NAME"
    TOTAL_in_ISA (no)          % Les relations IS-A totales impliquent au moins
      or TOTAL_in_ISA (yes)    % deux sous-types
        and SUB_TYPES_per_ISA (2 N)
      diagnosis "Les relations IS-A totales avec un seul sous-type sont interdites"
    DISJOINT_in_ISA (no)       % Les relations IS-A disjointes impliquent au moins
      or TOTAL_in_ISA (yes)    % deux sous-types
        and SUB_TYPES_per_ISA (2 N)
      diagnosis "Les relations IS-A disjointes avec un seul sous-type sont interdites"
    ROLE_per_RT (2 2)          % 2 <=  degré d'un type d'associations  <= 4
      or ROLE_per_RT (3 4)     % si 3 ou 4, le type d'associations ne peut avoir
        and ATT_per_RT (1 N)   % un rôle avec une cardinalité maximum de un
      or ROLE_per_RT (3 4)     % ou alors il a également des attributs
        and ATT_per_RT (0 0)
        and ONE_ROLE_per_RT (0 0)
      diagnosis "Le type d'associations &NAME a trop de rôles ou pas assez d'attributs"
```

```
    ID_per_RT (1 N)            % Si un type d'associations a des identifiants,
        and PID_per_RT (1 1)   % au moins est un identifiant primaire
      or ID_per_RT (0 0)
      diagnosis "Un des identifiants du type d'associations &NAME doit être primaire"
    EMBEDDED_ID_per_RT (0 0)   % Les identifiants imbriqués sont interdits"
      diagnosis "Les identifiants imbriqués doivent être enlevés dans le type
                 d'associations &NAME"
    ID_DIFF_in_RT (components)
      % Tous les identifiants doivent avoir des composants différents
      diagnosis "Les identifiants faits de composants identiques doivent être évités
                 dans le type d'associations &NAME"
    not SUB_ATT_per_ATT (1 1)
      % Les attributs décomposables doivent avoir au moins deux composants
      diagnosis "L'attribut décomposable &NAME a trop peu de sous-attributs"
    ID_per_ATT (0 0)          % Un attribut décomposable ne peut pas avoir d'identifiant
      diagnosis "L'attribut décomposable multivalué &NAME ne devrait pas avoir
                 d'identifiant"
    COMP_per_GROUP (1 N)       % Chaque groupe doit avoir au moins un composant
      diagnosis "Le groupe &NAME doit avoir des composants"
    ROLE_per_EID (0 0)         % Un identifiant d'un TE ne peut pas être composé que
        and COMP_per_EID (1 N) % d'un rôle
      or ROLE_per_EID (1 N)
        and COMP_per_EID (2 N)
      diagnosis "L'identifiant du type d'entités &NAME devrait contenir d'autres
                 composants"
    MULT_ATT_per_EID (1 1)
      % Si un identifiant de TE contient un attribut multivalué
        and COMP_per_EID (1 1) % il s'agit du seul composant de l'identifiant
      or MULT_ATT_per_EID (0 0)
      diagnosis "L'identifiant du type d'entités &NAME ne devrait pas contenir
                 d'attribut multivalué ou d'autres composants"
    ONE_ROLE_per_EID (0 0)
      % Un identifiant de TE ne devrait pas contenir un rôle avec une cardinalité
      % maximum de un
      diagnosis "Les rôles avec une cardinalité maximum de un devraient être enlevés de
                 l'identifiant du type d'entités &NAME"
    OPT_ATT_per_EPID (0 0)
      % Les attributs facultatifs ne sont pas autorisés dans les identifiants primaires
      diagnosis "Il ne peut y avoir d'attribut facultatif dans l'identifiant primaire
                 &NAME."
    COMP_per_RID (1 1)         % Si un identifiant d'un type d'associations a un
        and ROLE_per_RID (0 0) % seul composant, il s'agit d'un attribut
      or COMP_per_RID (2 N)
      diagnosis "L'identifiant du type d'associations &NAME devrait avoir plusieurs
                 composants"
    MULT_ATT_per_RID (1 1)     % Si un identifiant d'un TA contient un attribut
        and COMP_per_RID (1 1) % multivalué, il s'agit du seul composant de
      or MULT_ATT_per_RID (0 0)% l'identifiant
      diagnosis "L'identifiant du type d'associations &NAME ne devrait pas contenir
                 d'attribut multivalué ou d'autres composants"
    ONE_ROLE_per_RID (0 0)    % Un identifiant d'un type d'associations ne devrait pas
                              % contenir de rôle avec une cardinalité maximum de un
      diagnosis "Les rôles de cardinalité maximum un devraient être enlevés de
                 l'identifiant du type d'associations &NAME"
    OPT_ATT_per_RPID (0 0)
      % Pas d'attribut facultatif dans un identifiant d'un type d'associations
      diagnosis "Les attributs facultatifs devraient être enlevés de l'identifiant du
                 type d'associations &NAME"
end-model

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Signatures des procédures Voyager 2 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
extern SauverFichierLog "modifpm.oxo".SauverFichierLog(list,list,integer)
extern RejouerHistorique "modifpm.oxo".RejouerHistorique(list,list)
extern InverserHistorique "modifpm.oxo".InverserHistorique(list,list)
extern GenererScriptConversion "modifpm.oxo".GenererScriptConversion
                      (list,list,list,list,list,list,list,list,list,list,list,integer)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Définitions des boîtes à outils %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

toolbox TB_EVOLUTION_CONCEPTUELLE
  title "Evolution conceptuelle"
  description
    Cette boîte à outils permet de modifier un schéma conceptuel. On peut créer,
    modifier, détruire et transformer des types d'entités, des types d'associations,
    des attributs, des rôles et des groupes.
  end-description
  add create-entity-type
  add create-rel-type
  add create-attribute
  add create-group
  add create-role
  add modify-entity-type
  add modify-rel-type
  add modify-attribute
  add modify-group
  add modify-role
  add delete-entity-type
  add delete-rel-type
  add delete-attribute
  add delete-group
  add delete-role
  add tf-ET-into-att
  add tf-ET-into-RT
  add tf-split-merge
  add tf-RT-into-ET
  add tf-att-into-ET
end-toolbox

toolbox TB_CONCEPTION_LOGIQUE
  title "Conception logique"
  description
    Cette boîte à outils permet de transformer un schéma conceptuel en un schéma logique
    relationnel. On peut modifier et transformer des types d'entités, des types
    d'associations, des attributs et des groupes.
  end-description
  add modify-entity-type
  add modify-attribute
  add modify-group
  add tf-split-merge
  add tf-RT-into-ET
  add tf-att-into-ET
  add tf-disaggregate
  add tf-multi-att-into-list
  add tf-multi-att-into-single
  add tf-RT-into-att
  add tf-isa-into-RT
end-toolbox

toolbox TB_CONCEPTION_PHYSIQUE
  title "Conception physique"
  description
    Cette boîte à outils permet de transformer un schéma logique relationnel en un
    schéma physique relationnel. On peut créer, modifier et détruire des collections et
```

```
    des groupes.
  end-description
  add create-collection
  add create-group
  add modify-collection
  add modify-group
  add delete-collection
  add delete-group
end-toolbox

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Définition des types de processus %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

process EVOLUTION_CONCEPTUELLE
  title "Evolution conceptuelle"
  description
    L'évolution conceptuelle est le processus qui permet de modifier un schéma
    conceptuel de spécifications en vue de propager les modifications en aval de manière
    à modifier la base de données et les programmes.
  end-description
  input  sc0 "Schéma conceptuel"               : SCHEMA_CONCEPTUEL
  output sc1 "Nouveau schéma conceptuel"       : SCHEMA_CONCEPTUEL,
         ec1 "Historique évolution conceptuel" : FICHIER_LOG_EVOL
  strategy
    copy(sc0,sc1);
    toolbox TB_EVOLUTION_CONCEPTUELLE [log all] (sc1);
    external SauverFichierLog(sc1,ec1,1);
end-process

process CONCEPTION_LOGIQUE
  title "Conception logique"
  description
    La conception logique est le processus qui transforme un schéma conceptuel en un
    schéma logique relationnel.
  end-description
  input  sc1 "Nouveau schéma conceptuel"       : SCHEMA_CONCEPTUEL,
         cl0 "Historique de conception log."   : FICHIER_LOG_LOG
  output sl1 "Nouveau schéma logique"          : SCHEMA_LOG_SQL,
         cl1 "Nouvel hist. de conception log." : FICHIER_LOG_LOG
  strategy
    copy(sc1,sl1);
    external RejouerHistorique(sl1,cl0);
    toolbox TB_CONCEPTION_LOGIQUE [log all] (sl1);
    external SauverFichierLog(sl1,cl1,4);
end-process

process CONCEPTION_PHYSIQUE
  title "Conception physique"
  description
    La conception physique est le processus qui transforme un schéma conceptuel en un
    schéma physique relationnel.
  end-description
  input  sl1 "Nouveau schéma logique"          : SCHEMA_LOG_SQL,
         cp0 "Historique de conception phys."  : FICHIER_LOG_PHYS
  output sp1 "Nouveau schéma physique"         : SCHEMA_PHYS_SQL,
         cp1 "Nouvel hist. de conception phys." : FICHIER_LOG_PHYS
  strategy
    copy(sl1,sp1);
    external RejouerHistorique(sp1,cp0);
    toolbox TB_CONCEPTION_PHYSIQUE [log all] (sp1);
    external SauverFichierLog(sp1,cp1,5);
end-process
```

```
process EVOLUTION
  title "Modifications des spécifications conceptuelles"
  description
    Le processus d'évolution permet de construire une base de données relationnelle à
    partir d'un schéma conceptuel ainsi que de faire évoluer les schémas conceptuel,
   logique et physique.
  end-description
  intern rap_int         "Rapport d'interview"        : FICHIER_TEXTE,
         sc0             "Schéma conceptuel"          : SCHEMA_CONCEPTUEL,
         sl0             "Schéma logique"             : SCHEMA_LOG_SQL,
         sp0             "Schéma physique"            : SCHEMA_PHYS_SQL,
         cl0             "Hist. de conc. logique"     : FICHIER_LOG_LOG,
         cp0             "Hist. de conc. physique"    : FICHIER_LOG_PHYS,
         sc1             "Nouveau schéma conceptuel"  : SCHEMA_CONCEPTUEL,
         sl1             "Nouveau schéma logique"     : SCHEMA_LOG_SQL,
         sp1             "Nouveau schéma physique"    : SCHEMA_PHYS_SQL,
         ec1             "Hist. d'évol. conceptuel"   : FICHIER_LOG_EVOL,
         el1             "Hist. d'évol. logique"      : FICHIER_LOG_EVOL,
         ep1             "Hist. d'évol. physique"     : FICHIER_LOG_EVOL,
         cl1             "Nouv. hist. de conc. log."  : FICHIER_LOG_LOG,
         cp1             "Nouv. hist. de conc. phys." : FICHIER_LOG_PHYS,
         script_sql      "Script SQL"                 : FICHIER_SQL,
         script_conv_sql "Script de conversion SQL"   : FICHIER_SQL
  strategy
    do EVOLUTION_CONCEPTUELLE(sc0,sc1,ec1);
    do CONCEPTION_LOGIQUE(sc1,cl0,sl1,cl1);
    do CONCEPTION_PHYSIQUE(sl1,cp0,sp1,cp1);
    external GenererScriptConversion(sc0,sc1,sl0,sl1,sp0,sp1,ec1,cl0,cl1,cp0,cp1,101);
end-process

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Définition de la méthode %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

method
  title "Evolution"
  version "1.0"
  author "Jean-Marc HICK"
  date "21-05-2001"
  perform EVOLUTION
end-method
```

E.2 Exemple de journal

Le journal ci-dessous (*ec1.log*) matérialise l'historique d'évolution conceptuelle h_{EC1} de l'étude de cas du chapitre 8.

```
*POT "begin-file"
*MOD SIA
%BEG
  *OLD SIA
  %BEG
    %OID 5173
    %NAM "Telephone"
    %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."CLIENT" 5241
    %SNA "Tel"
    %CAR 0-1
    %SET S
    %TYP A
    %FLA "r"
    %LEN 12
    %DEC 0
  %END
  %OID 5173
  %NAM "Telephone"
  %SNA "Tel"
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."CLIENT" 5241
  %CAR 0-5
  %TYP A
  %SET S
  %LEN 12
  %DEC 0
  %FLA "r"
%END
*MOD ENT
%BEG
  *OLD ENT
  %BEG
    %OID 5183
    %NAM "REGION"
    %OWN 1 "QUOTA VENTES"/"Conceptuel-1" 5156
    %SNA "REG"
  %END
  %OID 5183
  %NAM "ZONE"
  %SNA "ZON"
  %OWN 1 "QUOTA VENTES"/"Conceptuel-1" 5156
%END
*CRE SIA
%BEG
  %OID 6290
  %NAM "Nomregion"
  %POX 0
  %POY 0
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."ZONE" 5183
  %PRV 6 "QUOTA VENTES"/"Conceptuel-1"."ZONE"."Nom" 5213
  %CAR 1-1
  %TYP A
  %SET S
  %LEN 30
%END
*MOD ROL
%BEG
  *OLD ROL
```

```
  %BEG
    %OID 5255
    %OWN 2 "QUOTA VENTES"/"Conceptuel-1"."visite" 5167
    %ETR 3 "QUOTA VENTES"/"Conceptuel-1"."CLIENT" 5241
    %CAR 1-1
  %END
  %OID 5255
  %OWN 2 "QUOTA VENTES"/"Conceptuel-1"."visite" 5167
  %ETR 3 "QUOTA VENTES"/"Conceptuel-1"."CLIENT" 5241
  %CAR 0-N
%END
*MOD SIA
%BEG
  *OLD SIA
  %BEG
    %OID 5199
    %NAM "Prix"
    %OWN 2 "QUOTA VENTES"/"Conceptuel-1"."vente" 5169
    %SNA "Prix"
    %CAR 1-1
    %SET S
    %TYP N
    %FLA "r"
    %LEN 8
    %DEC 0
  %END
  %OID 5199
  %NAM "Prix"
  %SNA "Prix"
  %OWN 2 "QUOTA VENTES"/"Conceptuel-1"."vente" 5169
  %CAR 1-1
  %TYP N
  %SET S
  %LEN 8
  %DEC 2
  %FLA "r"
%END
*CRE ENT
%BEG
  %OID 5286
  %NAM "FOURNISSEUR"
  %POX 185522
  %POY 114022
  %OWN 1 "QUOTA VENTES"/"Conceptuel-1" 5156
%END
*CRE SIA
%BEG
  %OID 5288
  %NAM "Nfour"
  %POX 0
  %POY 0
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %CAR 1-1
  %TYP N
  %SET S
  %LEN 5
  %DEC 0
%END
*CRE SIA
%BEG
  %OID 5290
  %NAM "Nom"
  %POX 0
  %POY 0
```

```
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %PRV 6 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR"."Nfour" 5288
  %CAR 1-1
  %TYP A
  %SET S
  %LEN 50
%END
*CRE SIA
%BEG
  %OID 5292
  %NAM "Adresse"
  %POX 0
  %POY 0
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %PRV 6 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR"."Nom" 5290
  %CAR 1-1
  %TYP A
  %SET S
  %LEN 70
%END
*CRE SIA
%BEG
  %OID 5294
  %NAM "Telephone"
  %POX 0
  %POY 0
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %PRV 6 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR"."Adresse" 5292
  %CAR 1-1
  %TYP A
  %SET S
  %LEN 12
%END
*CRE SIA
%BEG
  %OID 5296
  %NAM "Jour livraison"
  %POX 0
  %POY 0
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %PRV 6 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR"."Telephone" 5294
  %CAR 0-1
  %TYP A
  %SET S
  %LEN 10
%END
*CRE GRP
%BEG
  %OID 5298
  %NAM "IDFOURNISSEUR"
  %OWN 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %COM 6 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR"."Nfour" 5288
  %TYP A
  %FLA "P"
%END
*CRE REL
%BEG
  %OID 5300
  %NAM "fournit"
  %POX 157570
  %POY 112008
  %OWN 1 "QUOTA VENTES"/"Conceptuel-1" 5156
%END
*CRE ROL
```

```
%BEG
  %OID 5301
  %POX 167224
  %POY 112703
  %OWN 2 "QUOTA VENTES"/"Conceptuel-1"."fournit" 5300
  %ETR 3 "QUOTA VENTES"/"Conceptuel-1"."FOURNISSEUR" 5286
  %CAR 0-N
%END
*CRE ROL
%BEG
  %OID 5303
  %POX 147916
  %POY 111313
  %OWN 2 "QUOTA VENTES"/"Conceptuel-1"."fournit" 5300
  %ETR 3 "QUOTA VENTES"/"Conceptuel-1"."PRODUIT" 5195
  %CAR 1-1
%END
*POT "end-file"
```