LIMBAJE DE GESTIUNE A OBIECTELOR COMPLEXE
Caracteristicile fundamentale ale unui limbaj orientat pe obiecte complexe

Mat. Mircea Râureanu
Ing. Sabin Chiricescu

Institutul de Cercetări în Informatică

REZUMAT

Lucrarea constituie o introducere în teoria limbajelor orientate pe obiecte complexe. In prima parte, este definit un model de referinţă, numit **model structural**, care este un summum al caracteristicilor diferitelor modele de date, orientate pe obiecte complexe. In acest context, vor fi ulterior definite caracteristicile pe care trebuie să le aibă un limbaj de programare, asociat modelului structural. Ca urmare, se face o distincţie între **obiecte şi valori** şi se definesc constructorii utilizaţi pentru prelucrarea obiectelor şi a valorilor structurate: **vector, tuplu, set, lista**. In continuare, la nivelul schemei conceptuale a unui SDBD orientat pe obiecte complexe, se definesc noţiunile de **tip** şi **clasă**, necesare structurării bazei de obiecte complexe. In final, sînt ilustrate noţiunile anterior prezentate prin intermediul a trei limbaje dedicate: **SQL Extins, FP şi Calculus** (de fapt **calculul predicatelor de ordinul întîi**).

Cuvinte cheie: obiecte complexe, modele semantice de date, modele de date orientate pe obiecte complexe, mecanisme de abstractizare, tipuri şi clase, stări, scheme conceptuale, logici multisortate

INTRODUCERE

Lucrarea îşi propune să definească specificaţia unui limbaj de manipulare a obiectelor complexe urmînd îndeaproape studiul nostru [1].
De la început, trebuie precizat că termenul de limbaj, în sensul clasic, este desuet, aici fiind vorba de un mediu de programare, care va conţine cîteva limbaje dedicate, asociate fazelor de dezvoltare ale unei aplicaţii complexe de baze de date. Controlul va fi asigurat de diferite instrumente de programare specializate.
In continuare, vom ignora aspectele legate de implementare, conectîndu-ne atenţia asupra structurilor semantice, care vor exprima esenţa modulului de date, orientat pe obiecte complexe (OOC) prezentat în [1].

Un model structural

Modelul de date, orientat pe obiecte complexe (OOC), permite implementarea unei clase foarte largi de sisteme asociate. Din raţiuni practice, vom restricţiona modelul OOC la o viziune **structurală** asupra obiectelor (ignorînd viziunea **comportamentală**).
O asemenea viziune este similară modelelor semantice de date şi include obiecte complexe structurate, identitatea obiectelor şi o noţiune restrînsă de relaţie **isa**. In acest sens, o **bază de date** poate fi văzută ca o colecţie de elemente conectate prin diferite relaţii, cum ar fi : "face parte din clasa", "este un atribut al". Aşadar, o bază de date poate fi reprezentată ca un graf orientat, în care arcele sînt relaţii de tipul anterior.
O bază de date relaţională (sau bazată pe valori) este un caz special, în care graful asociat se supune unor restricţii.
Un important aspect al sistemelor de baze de date este **separarea netă** dintre **schemă** şi **date**.
Schema unei BD va conţine nume şi descrieri ale tipurilor de date sau **colecţii** utilizate drept containere de date, de relaţii etc.
Modelul structural extinde modelul bazat pe valoare (ex: modelul relaţional) şi, în multe cazuri, îi rezervă caracteristicile esenţiale, în particular, oferind o bază adecvată pentru limbaje declarative. In studiul [1], am prezentat unele exemple semnificative, cum ar fi: ObjectSQL, Nested-DATALOG, Algres.

1. Nivelul de date

La acest nivel, componentele de bază ale modelului structural sînt:
- obiecte,
- valori structurate.

a. Obiecte

In continuare, vom considera o Bază de Date Orientată pe Obiecte Complexe ca fiind o colecţie de obiecte complexe.
Proprietatea fundamentală a obiectelor este aceea că, orice obiect are **identitate** unică, fără a se schimba pe parcursul "vieţii" sale.
Obiectele pot fi adresate, în mod natural, prin: **nume** şi **referinţe**.
Utilizarea **numelor** implică existenţa, în **schema** BD, a **spaţiului de nume** ca asociere între identificatori şi obiecte. Sistemele relaţionale au un spaţiu de nume care conţine, în particular, numele relaţiilor şi atributelor acestora.
In unele cazuri, pot exista obiecte care nu au nume, fiind deci, necesară utilizarea referinţelor. De exemplu, dacă un obiect **salariat** (s) are o componentă **departament**, atunci expresia **s.dept** va denota departamentul. De aici, nu rezultă că este necesară utilizarea unui constructor de tipuri ca "ref", pentru că modelul conceptual nu este influenţat de detalii de implementare în care obiectele şi referinţele sînt diferite. De altfel, nu este necesar ca obiectele şi referinţele să fie tratate diferit. De aceea, în limbajul care va fi definit în capitolele următoare,de cîte ori vom utiliza un **nume** sau o **referinţă** la un obiect, vom desemna obiectul însuşi.

b.Valori structurate

Distincţia dintre valori şi obiecte poate fi explicată foarte clar, în termenii logicii matematice. Astfel, există două abordări pentru definirea **structurilor. Prima** constă în furnizarea unei colecţii de formule. O astfel de colecţie, numită **teorie**, defineşte clasa structurilor care satisfac toate formulele sale, numite **modelele** sale.

Elementele structurilor sînt definite ca neinterpretate (nu au o semnificaţie), semantica lor fiind dată de relaţia lor cu alte elemente ("comportamentul" lor) aşa cum este specificată de formule.

A doua abordare constă în denumirea explicită a structurii: de exemplu, numerele întregi împreună cu funcţiile şi predicatele asociate. O astfel de structură este referită ca **predefinită** (built-in), iar elementele sale sînt considerate **interpretate**.

In cazul general, cele două abordări sînt combinate: unele componente din domeniul de date sînt definite prin formule, în timp ce altele sînt predefinite.

In particular, aceasta trebuie să fie forma definiţiei logice a unei Baze de Date Orientate pe Obiecte Complexe, în care, atît obiectele, cît şi valorile sînt utilizate. **Valorile** sînt elemente ale domeniilor **predefinite, obiectele** sînt elemente ale domeniilor neinterpretate.

Prin **valori atomice** vom înţelege elemente de domeniu, cum ar fi:**întregi, reali şi şiruri.**

Obiectele, **care reprezintă abstracţii, se numesc** obiecte abstracte.

Multe din BDOOC utilizează pentru reprezentarea obiectelor identificatorii de **obiecte** (o_ids).

Să observăm că, atît valorile, cît şi obiectele abstracte pot fi considerate **o_ids.**

Constructorii, cum ar fi **lista, tuplu, set** sînt utilizaţi pentru a crea obiecte compuse sau structurate.

Vom utiliza conceptul de **valoare structurată** ca fiind o valoare obţinută prin intermediul constructorilor anteriori. In continuare, vom studia efectul unor mecanisme de abstractizare asupra definirii valorilor structurate.

1. **Colecţia** este un mecanism de abstractizare bine cunoscut. Mulţimile (seturi) au proprietăţile fundamentale ale obiectelor: un set este diferit de orice alt obiect şi este imutabil (o **înserare** de obiect nu schimbă un set în alt set). Mai mult, **seturile** au proprietăţile valorilor. Un set este identificat unic prin conţinutul său. Conţinutul **este** însuşi setul: prin axioma extensionalităţii, două seturi cu aceleaşi elemente sînt acelaşi set. In mod normal, seturile nu au nume: cu toate acestea un set finit poate fi considerat a avea un nume: **conţinutul său.**

In final, seturile vor fi utilizate în limbajele de BDOOC pentru descrierea altor obiecte: de exemplu, proprietatea **"copil"** a unui **salariat** asociază fiecărui salariat un set de descriptori de **"copil"**.

2. In mod similar, un alt mecanism de abstractizare, **agregarea**, impune conceptul de **tuplu (record)**. De exemplu, tuplul [A:5, B:"Ion"] este unicul obiect cu atributele A şi B, care au valorile 5 şi, respectiv, "Ion". (Aici atributele sînt pur şi simplu nume pozitionale ca în modelul relaţional). Cum un set este identificat prin conţinutul său, un tuplu este unic determinat prin structura şi conţinutul său, adică, prin atributele şi valorile lor. Să observăm că un tuplu este finit, deci, are un nume - reprezentarea sa într-o notaţie standard. Desigur, tuplele servesc pentru a descrie obiecte. Aceste obiecte pot fi obiecte abstracte din sistem sau, ca în modelul relaţional, obiecte din lumea reală, care nu sînt explicit reprezentate în sistem.

3. Mecanismele de abstractizare, anterior descrise, pot furniza şi alte concepte des utilizate, cum ar fi **lista** şi **tabloul. Lista** este o colecţie ordonată de elemente, fiecare element conţinînd o informaţie (o referinţă) la elementul următor. **Tabloul** este o colecţie ordonată de elemente, în care fiecare element are asociat un **index.** Ordinea indecşilor induce ordinea elementelor din tablou. Ambele abstracţii vor fi combinate în limbajul nostru prin noţiunea de **secvenţă.**

Astfel, **seturile, tuplele şi secvenţele** au mai curînd proprietăţile valorilor, decît ale obiectelor abstracte. In termenii logicii, ei sînt constructori standard, semnificaţia lor presupunîndu-se a fi predefinită. Acestea sînt valorile utilizate în modelele avînd relaţii imbricate [6] (relaţii care nu sînt în INF) sau cu obiecte complexe. O valoare structurală poate fi văzută ca un arbore, în care frunzele sînt **atomice,** nodurile interne sînt etichetate de constructori (este pe deplin posibilă generalizarea acestei idei, considerînd valorile ca "dag"-uri sau grafuri orientate).

In concluzie, un limbaj dedicat unui model structural trebuie să conţină mulţimea de constructori: **set, tuplu,** (record), **secvenţă.**

Din raţiuni practice, în limbajul nostru a fost introdus constructorul **multiset** (conţinutul său este format din seturi).

c. Stări

Obiectele abstracte sînt neinterpretate. Lor li se asignează semnificaţie prin definirea unei relaţii cu alte obiecte. Intr-un sistem de baze de date, un obiect abstract reprezintă o entitate despre care dorim să înregistrăm informaţii; semnificaţia i se asociază prin **legarea** cu aceste informaţii. Facilitatea prin care se realizează aceasta este asocierea unei **stări** cu un obiect. Vom utiliza notaţia **state** (o) [3], unde (o) este un obiect. Conceptul de stare, care este diferit de acela de obiect abstract, este fundamental în modelele orientate pe obiecte. In particular, acesta explică cele două feluri de egalitate care pot fi utilizate:**egalitatea obiectelor şi egalitatea stărilor** acestora.

Starea unui obiect abstract poate fi:

- o valoare atomică;
- alt obiect abstract;
- o valoare structurată (care le include pe cele de mai sus).

O altă abordare în descrierea obiectelor abstracte constă în eliminarea completă a noţiunii de stare; în locul acesteia sînt utilizate atributele definite direct pe obiect [6].

Pentru noi, utilizarea atributelor sau a stărilor **tuplu-valuate** (cu valori tuple) sînt echivalente. Dacă un obiect are o stare tuplu-valuată, putem considera atributele tuplului ca fiind definite direct pe obiect şi utilizăm notaţia **o.A**, care este o prescurtare a notaţiei **state** (o).A. Limbajul nostru va utiliza această abordare.

2.Nivelul de schemă conceptuală

In capitolul anterior am descris construcţiile semantice, necesare unui limbaj de descriere a obiectelor şi rolurilor structurate, care este, de fapt, un limbaj de definire a **schemei conceptuale** a unei BDOOC.

a.Tipuri

Conceptele primitive ale acestui nivel sînt tipurile şi clasele. Se pune întrebarea dacă sînt ambele necesare. Unele sisteme (FD [4]) au numai tipuri, altele ([8], CLOS [7]) au numai clase, iar altele dispun de ambele concepte.

In mod tradiţional, în modelul structural, tipurile au două valori:

- denotă o structură;
- denotă o extensie (domeniul elementelor care au acea structură).

De asemenea, clasele denotă extensii - fiecare clasă denotă o colecţie de obiecte. Totuşi, extensia denotată de un tip este legată şi definită de structura asociată tipului. Extensia asociată clasei este definită de utilizator. Este evident că tipurile şi clasele au roluri distincte, ambele utile. Această opţiune este justificată de caracterul oarecum exploratoriu al cercetării noastre. După implementarea prototipului versiunii preliminare, vom extinde limbajul cu **clase.**

In continuare, vom prezenta cîteva definiţii, descriind semnificaţia fiecărui concept definit în termenii, atît extensionali, cît şi structurali. Totuşi, în versiunea preliminară, limbajul nostru va avea numai tipuri.

a.Valori atomice: propunem ca tipurile **int, real,string, date** să fie predefinite în spaţiul numelor. O definiţie leagă un nou nume de extensia unuia dintre aceste tipuri. De exemplu:

type t_salariu = int;
type t_vîrsta = [0...120];
type t_greutate_mircea = {100}.

b.Obiecte abstracte

O declaraţie a unui tip de obiect abstract specifică numele său, precum şi tipul stărilor obiectelor sale. De exemplu:

o_type o_salariat : **state** = v_salariat (valoare)
(tipul v_salariat trebuie să fie definit anterior)

c.Valori structurate

O definiţie de valoare structurată arată astfel:
type t_nume = t_structura
în care structura este o expresie conţinînd constructorii **multiset, set, tuplu şi secvenţă.**
(Pentru exemplificări mai consistente, urmăriţi capitolul dedicat descrierii limbajului). De exemplu:

type v_salariat = [nume : string.salariu : t_salariu,
şef : o_salariat]

defineşte un tip record (**tuplu**) ale cărui instanţe servesc ca stări ale obiectului abstract o_salariat.

In unele lucrări se sugerează utilitatea folosirii relaţiilor. O raţiune constă în aceea că, dorim un model de date, care să conţină modelele **bazate pe valori**, ca un caz particular.

Altă raţiune este că, adeseori, răspunsurile la cereri sînt relaţii şi dorim ca limbajul să fie închis la compunere (o compunere de două coloane dă o relaţie, deci, o structură din limbaj). De asemenea, avînd relaţii, putem defini relaţiile dintre obiecte fără a crea obiecte abstracte pentru a le reprezenta. Vom defini o **relaţie** ca fiind o mulţime de elemente de acelaşi tip, notată:

rel r_nume : tip

Dacă **tip** este un tuplu, relaţia se reduce la relaţiile lineare ale modelului relaţional, cu forme normale.

Să observăm că adăugarea relaţiei complică modelul structural: de exemplu, felul în care conceptul de moştenire interacţionează cu relaţiile rămase de studiat.

Din aceste motive, limbajul nostru nu va include relaţiile în versiunea sa preliminară, rămînînd deschis la operaţia de compoziţie.

d. Schema conceptuală şi instanţe

Un set de definiţii ca acelea de mai sus constituie schema unei baze de date.

O **instanţă** este o bază de date (în prima abordare) sau o structură (în a doua abordare).

Definiţiile de tipuri fac ca, limbajul de descriere asociat să devină, din punct de vedere logic, multisortat (calculul predicatelor de ordinul 1 este netipizat). Orice element din domeniul unei structuri poate apărea în orice coloană a unui predicat. Versiunea multisortată este tipizată. Limbajul conţine o mulţime de nume de **sorturi**, iar domeniul unei structuri este divizat în subdomenii, numite **sorturi**. Fiecărei coloane a unui predicat îi este asociat un nume de sort şi, deci, numai elementele sortului corespunzător pot apărea aici.

Corespunzător, o instanţă este o structură multisortată. Pentru fiecare tip atomic de valori sau obiecte abstracte, există un domeniu corespunzător. Domeniile tipurilor valoare sînt fixate, pe cînd domeniile tipurilor abstracte nu sînt restricţionate şi nici nu trebuie să fie aceleaşi în diferite structuri.

Dîndu-se domeniile atomice, domeniile tipurilor valorilor structurate sînt definite astfel: "Un tip t1 depinde de tipul t2 dacă t2 este o frunză în expresia de definiţie a lui t1 sau dacă t1 depinde de t3 şi t3 depinde de t2" (Dependenţa este o relaţie de echivalenţă).

Dacă impunem ca relaţia de dependenţă să nu fie relaţie de echivalenţă (deci nu avem dependenţe recursive), atunci **numele** de tipuri pot fi ordonate astfel încît, fiecare tip depinde numai de tipurile precedente, iar definiţia sa poate fi transformată astfel încît, în ea să apară numai tipuri atomice. Cum extensiile tipurilor atomice sînt predefinite, iar acelea ale o-tipurilor sînt date, extensia tipurilor definite poate fi construită în mod evident. Aceasta este soluţia pe care am adaptat-o în definiţia limbajului nostru. Discuţia ce urmează are sensul de a completa definiţia extensiei.

Dacă tipurile recursive sînt permise, atunci sînt necesare construcţii mult mai complexe. Ca mai sus, vom substitui fiecare tip atomic cu domeniul său şi vom privi rezultatul ca o mulţime de ecuaţii definind domeniile tipurilor structurale. Soluţia cea mai mică

(numită și **cel mai mic punct fix**) este, prin definiție, extensia tipurilor structurate. Aceasta poate fi calculată ascendent.
De exemplu:
type v_parte = [**nume** : **string**, **subpărți** : {v_parte}]
utilizată pentru a defini o ierarhie de părți. Soluția se obține în felul următor: întîi, domeniul lui v_parte este vid; utilizînd definiția, adăugăm părți care au un nume și o mulțime vidă de subpărți.
In general, la fiecare iterație vom adăuga părți care sînt rădăcina unei ierarhii de adîncime n-1. Domeniul conține, astfel, structurile adăugate în toate iterațiile și este infinit.
In final, întrucît tuturor tipurilor li s-au asociat extensii, putem descrie alte componente ale unei structuri:

- **state** este o funcție care mapează (asociază) obiecte abstracte ale domeniului fiecărui **o type** pe obiecte, ale tipului stării date în definiția sa (a o_tipului);
- fiecare atribut este o funcție unara pe un domeniu de tuple și, posibil, pe domeniile unuia sau mai multor **o type**, cu o extensie (limite) specificată de definiția sa.

Acum sînt necesare cîteva precizări. Intîi, se poate observa că modelul structural adoptat combină caracteristicile, atît ale modelelor bazate pe valoare, cît și ale celor bazate pe obiecte. Un model general, bazat pe valoare se obține eliminînd **o_type**, (ceea ce noi am făcut în versiunea preliminară a limbajului).
De asemenea,modelele specifice, bazate pe valori se obțin restricționînd construcțiile permise în definițiile structurate de tip.
In al doilea rînd, definiția anterioară permite cicluri în date. De exemplu, pentru **un** obiect salariat **e**, valoarea lui **set** în **starea** sa poate fi **e** însuși. Alt exemplu: într-o schemă care definește persoane, dacă fiecare persoană are o soție de tip persoană, atunci fiecare pereche soț, soție formează un ciclu; observăm că, **cicluri în schemă** există atunci cînd un tip este definit prin intermediul său (definiția este **recursivă**).
Cicluri de date există cînd **un** element este conectat cu el însuși printr-o relație oarecare. Aici sînt două fenomene distincte: o schemă poate fi ciclică, dar ciclurile în date sînt interzise. De exemplu, definiția anterioară a unei ierarhii de părți este recursivă, dar, în mod evident, felul în care domeniul este definit exclude posibilitatea ca o parte să fie legată cu ea însăși printr-un lanț de relații de subparte.
Pe de altă parte, definițiile lui o_salariat, a stării sale și a lui v_salariat date anterior nu sînt recursive. Totuși, această definiție permite cicluri în date (prin funcția **state**).
In al treilea rînd, o schemă poate fi descrisă ca un dag, adică o sumarizare a grafului instanțelor sale. Nodurile sînt tipuri. Un arc-set conectează fiecare tip set cu tipurile elementelor sale. Un arc de stare (de valoare structurată) conectează un o_type cu un tip de stare, iar un arc-atribut conectează cu un tip_tuplu cu o componentă de tip corespunzător.

e. Moșteniri
Moștenirea prin relația **isa** este o caracteristică importantă a modelelor orientate pe obiecte, deci și pentru modelul OOC.
Intîi, pentru tipurile structurale există **moștenirea de tipuri** (sau relația de subtip).
Aceasta este definită astfel: "t1 este un **subtip** al lui t2

- dacă ambele sînt atomice, atunci Dom(t2) **este inclus în** DOM(t1).

sau,

- dacă sînt tipuri tuplu, t2 are toate atributele lui t1,

sau,

- dacă sînt tipuri set, ti = (Si), atunci S2 este subtip al lui S1.

Pentru tipurile tuplei se utilizează una din expresiile sintactice:

- t2 **inherits** t1, **add** atr-list

sau

- t2 **isa** t1, **add** atr-list.

In limbajul nostru, vom utiliza varianta a doua, ușor modificată, astfel:

- t2 **isa** t1 + lista de atribute.

In unele lucrări [3] se utilizează și constructori de tip V, /\ (reuniune și intersecție (least_upper_bound și greatest_upper_bound).
In mod intenționat, nu am introdus asemenea constructori în limbaj pentru a furniza informații mai precise asupra relațiilor între diferite tipuri.

3.Limbaje declarative

In studiul nostru [1] am prezentat o clasă largă de limbaje pentru modelele orientate pe obiecte complexe. Dintre acestea, vom acorda o atenție specială limbajelor declarative, care sînt formalisme de descriere a unui rezultat dorit fără a se specifica modul de calcul al acestuia. Pentru ca un asemenea limbaj să fie eficient în practică, el trebuie restricționat astfel încît, programele sale să fie executate cu performanțe acceptabile. De aceea, în locul utilizării unui calcul pur (calculul pe tuple sau cel pe domenii, în cazul modelului relațional și calculul pe obiecte complexe, în cazul modelului structural). De exemplu, limbajele relaționale se bazează pe clasa **cererilor conjunctive**, care este echivalentă cu un subset al algebrei cunoscute ca cereri **select_project_join** (SPJ). Extensii ale acestor limbaje pentru modelele orientate pe obiecte au fost prezentate în studiul nostru.
Reamintim că o BDOOC va fi văzută ca un graf orientat în care nodurile sînt obiecte, iar arcele conectează seturile de elementele lor, tuplele și obiectele de valorile atributelor lor.
In acest graf, unele căi sînt finite și se termină în valori atomice (cazul limbajului nostru), dar, dacă sînt permise cicluri, pot exista drumuri infinite.
Urmează că, aceste limbaje sînt foarte apropiate de FP (Functional Programming) - o schemă pentru programarea funcțională introdusă de BACKUS [7].

Un **sistem FP** constă dintr-un **set de obiecte**, un set de **funcții de bază** și un **set de funcționale** (sau forme de combinație), care sînt utilizate pentru a construi noi funcții.
Un program din sistemul FP este o expresie care denotă o funcție și care este construită prin aplicarea funcționalelor asupra funcțiilor de bază. Unul din avantajele acestei abordări constă în faptul că programele pot fi manipulate prin **transformări ce prezervă echivalența**. Acestea sînt și avantajele algebrei relaționale; de fapt, algebra relațională este un sistem FP restricționat și, după cum vom vedea, extensiile acesteia pentru modelul OOC constituie, de asemenea, un sistem FP.
Funcționalele unui sistem FP includ:

- **compunerea**;
- constructori de tuplu și selectori;
- **filtru**: (care se mai numește "**aplică-tuturor**"), care aplică o funcție tuturor elementelor unei liste;
- **insert**: utilizează o funcție binară pentru a calcula un agregat al valorilor dintr-o listă.

Pentru adaptarea sistemului FP la modelul structural vom adăuga un **constructor de set-finit** și vom restrînge funcția utilizată, ca argument al lui **insert**, la funcții comutative și asociative cînd funcționala este utilizată pentru a construi funcții pe seturi. Se observă că, operatorul clasic **project** este un caz special de **filtru** - functia care este aplicată este un constructor de tuplu. Similar, operatorul **select** este un caz special de filtru obținut prin utilizarea unei funcții de forma:
dacă predicat, **atunci** I,
altfel NULL,
unde I este funcția identitate.
Vom utiliza notația p <fun> (set), pentru un filtru care aplică funcția fun fiecărui element al unui set (este o proiecție generalizată) și **s<pred>(set)** pentru a denota o selecție. In modulul structural, deoarece valorile structurate pot construi mulțimi imbricate, expresiile filtru pot fi argumente funcționale ale altor filtre - deci, filtrele pot fi imbricate.
Limbajele considerate vor fi extensii ale SQL. O cerere tipică are forma:
- **select** lista-țintă
- **from** lista-delimitată (range?)
- **where** lista-de-codificări.

In stilul FP, o astfel de cerere este o **compoziție** a unui filtru **select** și a unui filtru **project** (fiecare putînd conține filtre imbricate):
p <lista-țintă> (s<lista-de codificare> (lista-delimitată))
Pentru a ilustra ideea, vom prezenta diferite cereri, fiecare dintre ele fiind exprimată în SQL extins, FP și calculul extins.
Vom utiliza schema următoare:
- **type** o-persoana : **state** = [nume: data-naștere:date];
- **type** o-salariat **isa** o-persoana + [salariu: int, copii:{s-persoana}];
- **type** o-departament: state = [d-name:string, buget:int, șef:o-salariat,salariați:{s-*s*}]
- **type** s-departament: **set of** o-departament;
- **type** s-persoana : **set of** o-persoana;
- **type** s-salariat :**set of** o-salariat.

Cerere A: să se găsească șefii departamentelor care au un buget mai mare de 1 milion.
Această cerere nu este imbricată și poate fi exprimată astfel:
SQL: **select** șef
from s-departament
where buget > 1.000.000
FP : p(<șef>(s<buget> 100000> (s-departament))
Calculus:
d.șef: s-departament(d) **and**
d.buget > 1000000
Cerere B: să se editeze setul fiecărui deportant împreună cu copiii salariaților pe care-i conduce:
SQL: **select** șef, **collapse** (select copii from salariați)
from s-departament
FP: p<șef,**collapse**(p<copii>(salariați)> (s-departament)
Calculus:
d.șef, C:s-departament(d) **and**
C = {c: e(d.salariati(e) **and** e.copii(c))}
Cerere C: pentru fiecare **departament**, să se listeze numele său, al salariaților săi, care au cel puțin trei copii, precum și numele copiilor:
SQL: **select** d_nume, (**select** nume, (select nume from copii) **from** salariați
where insert (O,+)(copii >3)
from s-departament
FP: p<d-nume, p<nume>, p<nume>(copii)> (s<insert (o,+)(copii) >3> (salariați))>
. (s-departament).
Calculus:
d.d-nume,<**e.nume,<c.nume>**;
s-departament(d) **and** d.salariați(e)
and e-copii(c) **and** count (e-copii) > 3.
Aceste cereri ilustrează în ce fel limbajele de tip SQL și FP utilizează avantajele modelului. Cu toate că joncțiunile sînt necesare, ele sînt în multe cazuri înlocuite prin traversări de drumuri (și arborele de obiecte).
Vom utiliza notația SQL extins pentru, atît performanțele ei mai bune în prezentarea vizuală, cît și datorită faptului că SQL este un limbaj standard pentru bazele de date relaționale.
Notația FP, puțin adaptată, va fi utilizată pentru reprezentarea internă, fiind considerată ca moștenirea abstractă dedicată, care face legături între front-end-ul SQL și mașina relațională abstractă.
Operatorul **collapse** este definit în [8].
In limbajul nostru vom utiliza notația **unnest** (**liniarizare**) și respectiv, **nest** (**grupare**). Operatorul **collapse** este o funcție care acceptă la intrare o colecție de seturi și furnizează reuniunea lor. Este pur și simplu o funcțională **insert**, aplicată unei reuniuni.

4. Funcții

In modelul relațional, atributele denumesc coloane ale predicatelor, iar funcțiile nu sînt folosite. Dar, vederea alternativă a atributelor ca **funcții unare** este mult mai

atractivă pentru un model orientat pe obiecte, deoarece poate fi ușor generalizată la funcții cu mai multe argumente.

Pentru început, să studiem conceptul de **funcție memorată**, care este o funcție a cărei extensie este stocată explicit în baza de date. Acesta este conceptul de **funcție de date**, introdus în limbajul COL [9]. O astfel de funcție este, în esență, o **relație** care satisface o dependentă funcțională. Din orice punct de vedere am privi, utilizator sau sistem, singura diferență între a denumi o relație sau o funcție este aceea că, în ultimul caz, sistemul garantează satisfacerea dependenței funcționale.

Să considerăm funcția binară **copii**, care asociază oricăror două persoane mulțimea copiilor lor. Cea mai simplă abordare constă în stocarea funcției ca o relație ternară imbricată, avînd schema:

[o-persoana, o-persoana,{o-persoana}]

Cea de a treia poziție conține mulțimea "copii ai celor două persoane". Alternativ, funcția poate fi stocată ca o relație binară, avînd schema:

[o-persoana, {[o-persoana, {o- persoana}]}]

Aici, prima poziție conține o persoană **p**, iar a doua poziție conține o relație binară, în care fiecare tuplu conține o persoană **q** și mulțimea copiilor lui **p** și **q**. Oricum, în ambele cazuri, funcțiile sînt memorate, iar reprezentările lor sînt relații imbricate, deci, nu ieșim din modelul structural. Faptul că, unele structuri se numesc funcții și nu relații, nu va provoca nici o dificultate. Să observăm că, a doua funcție **copii** este tratată ca o **dată**, dar acest lucru nu este diferit de tratarea ca dată a unei relații stocate ca valoarea unui atribut oarecare dintr-o relație imbricată.

Să considerăm funcția unară care mapează **p** la **copii(p)**. Să amintim că un atribut este o funcție unară. Astfel, în schemă se poate descrie această funcție ca un atribut numit, să spunem, **copii-comuni**. Pentru fiecare persoană **p**, **p.copii-comuni** este o funcție memorată, care, pentru fiecare persoană **q**, furnizează setul copiilor lui **p** și **q**. O declarare posibilă a acestui atribut ca parte a declarației tipului **o-persoana** este:

copii-comuni : o-persoana - [p-persoana].

Să observăm că, deoarece declarația atributului este parte a declarației o.persoana, domeniul **o-persoana** este omis; numai range-ul este specificat. Aceasta este standard în declararea atributelor. Valoarea lui **p.copii-comuni** pentru orice **p** este o extensie de funcție (posibil vidă), adică, mulțimea tuplelor care satisfac o restricție funcțională, deci o **valoare structurată**.

Cele trei reprezentări anterioare ale funcției **copii** sînt echivalente: factorizarea unei funcții multiargument în cîteva funcții unare este o idee banală, iar denumirea funcției multiargument ca relație imbricată, funcție imbricată sau atribut funcțional-valuat nu are importanță. Totuși, ultima reprezentare deschide căi noi de explorare plecînd de la modelul structural, (de exemplu: ideea de **funcție derivată**, care nu este memorată, ci este calculată ca un program de clauze Horn din relația **părinte** stocată în baza de date). Vom postula aplicarea constructorilor asupra funcțiilor memorate pentru a obține funcții noi. Un set de constructori poate include, de exemplu, **compoziție, aplică- tuturor** etc. Cadrul general pe care-l vom adopta este că, funcțiile date într-o bază de date, pot fi memorate sau predefinite. Acestea sînt referite, fie prin **nume**, fie prin **referință**. O colecție de valori funcționale structurate este obținută aplicînd constructorii. O abordare pentru a avea o astfel de facilitate este de a privi un limbaj de programare ca definind o mulțime de constructori, astfel că expresiile din limbaj vor denota valori funcționale construite. In acest caz, un atribut funcțional-**valuat** poate avea ca domeniu de definiție funcții care pot fi memorate sau construite. In ultimul caz, aceste funcții sînt denotate de programe.

In versiunea preliminară a limbajului nostru **nu** vom introduce conceptul de funcție memorată, urmînd să luăm o astfel de decizie după o perioadă de testare a prototipului acestei versiuni de limbaj.

BIBLIOGRAFIE:

1. RÂUREANU,M. : Studiu privind modelele semantice de date pe obiecte complexe, Lab. INTELDATA, ICI, Iunie 1991.
2. SCHECK, H.J., SCHOLL, M.H.: A Primer on Complex Object Orientation. In: Information System, nr.11, vol.2, an 1986, pp.25-35.
3. BEERI,C. : A Formal Approach to Object-oriented Databeses. In: Data & Knowledge Eng., nr.5, vol.2, anul 1988, pp 66-81.
4. DANFORTH, S. ș.a.: FAD a Database Programming Language. In: TR, MCC, nr.25, anul 1988, Austin, Texas.
5. BANERJEE, J., CHOU, H.T. ș.a.: Data Model Issue for Object- oriented Applications. In: ACM, Trans. Office Information System, nr.5, vol.4, anul 1987, pp. 19-27.
6. ABITEBOUL, S.: Towards a Deductive Object-oriented Database Language. In: Proc. DOOD conf., Washinton, 1986, pp.67-92.
7. BACKUS, J.: Can Programming Be Eliberated from the von Neumann Style? In: Commun. ACM, nr.21, 1978, pp 73-90.
8. ABITEBOUL, S., BEERI, C.: On the Power of Languages for the Manipulation of Complex Objects. In: INRIA, raport de cercetare nr. 846, Franța, 1988.
9. ABITEBOUL, S., GRUMBACH,S.: COL - a Logical Approach to the Manipulation of Complex Objects. In: Proc. EDTB, seria (Lecture Notes in C.S.) Editura Springer, New York; 1988, pp.302-330.