RAFINAREA CUNOŞTINŢELOR REPREZENTATE PRIN OBIECTE STRUCTURATE (FRAME)-PARADIGMA "BLACKBOARD"

(O abordare pur obiectuală)

ing. Dragoş Sâmbotin

Universitatea "Politehnica" Bucureşti

Rezumat: În lucrare este prezentat un sistem de reprezentare a cunoştinţelor prin obiecte structurate şi modul de utilizare a lui în proiectarea unor sisteme expert, de proiectare prin rafinare concurenta . Sistemul este de tip prototip-clona ,cu unele facilităţi noi faţă de cele existente în sistemele proiectate pâna acum , cum ar fi : tratarea unitară a metodelor şi a sloturilor, posibilitatea modificării dinamice a ierarhiei de obiecte, contexte multiple pentru un obiect. Modelul de rafinare este cel bazat pe tablă (**blackboard**) folosindu-se astfel o agendă de taskuri şi un număr de surse de cunoştinţe (**demonii de rafinare**), fiecare ştiind să rafineze un anumit tip de task. Noutatea abordării de faţă constă în modalitatea de a reprezenta agenda de taskuri, folosind o abordare complet **orientată obiect** (în acest sistem **toate** entităţile cu care se lucrează sunt obiecte). Astfel, agenda de taskuri nu este explicit reprezentată, ea rezultând din ierarhia arborescentă a obiectelor construite pe parcursul rafinării. Lucrarea este structurată în trei capitole: în primul capitol (introducere) se face o prezentare generală a contextului de dezvoltare de sisteme expert bazate pe frame-uri; în capitolul al doilea, se face o prezentare a sistemului de obiecte "SELF" ; în capitolul al treilea este prezentat sistemul de rafinare (insistându-se pe particularităţile sistemului de faţă) şi un exemplu de proiectare folosind acest sistem.

Cuvinte cheie : rafinare, delegare, demon, task, frame, ierarhie OOP, moştenire, faţeta, message-passing, prototip, clonă.

1. Introducere

Sistemele expert, bazate pe obiecte structurate (frame-uri), s-au impus în ultima vreme datorită avantajelor indiscutabile pe care le oferă, cum ar fi (printre altele): posibilitatea unei uşoare reutilizări a codului, eleganţa programarii, specificarea declarativă a problemei, uşor de înţeles şi întreţinut. Pentru o abordare fundamentată a unui sistem de obiecte structurate, se impune stabilirea proprietatilor unui astfel de sistem. Aşa cum se arată şi în [2], un model convenţional al unui limbaj orientat obiect, trebuie să satisfacă următoarele două proprietăţi:

(1) obiectele structurate permit includerea specificaţiilor de moştenire precum şi descrierea elementelor constitutive tot în termeni de obiecte structurate;

(2) obiectele au asociate comportări ce pot fi activate prin transmitere de mesaje.

În viziunea general acceptată de până acum, obiectele au ca principale părţi constitutive: super-obiecte, sloturi, metode, demoni. O abatere de la acest model este, de exemplu , [1], unde, un obiect poate avea asociată şi o metadescriere.

Cu toate acestea, în lumea OOP-ului, cele mai multe discuţii se poartă, nu atât asupra modalităţii alese pentru reprezentarea internă a obiectelor, cât asupra comportării lor ca întreg şi, în special , asupra oportunităţii alegerii unui anumit model de moştenire: **prin delegare** ("delegation") sau **instanţiere** ("inheritance").

În limbajul de specialitate, această dezbatere este deseori desemnată prin paradigma:"**Classes versus Prototypes**" (clase sau prototipuri).

În sistemul la care se referă lucrarea de faţă s-a ales modelul delegării (**prototip-clona**), în care un obiect, odată definit, poate fi folosit ca atare, sau poate fi folosit ca prototip (**super-obiect**), pentru un alt obiect.

2. Consideraţii OOP

Limbajele orientate obiect, pot fi împărţite , din punctul de vedere al moştenirii, în două mari clase:

(1) limbaje bazate pe clase, cum ar fi C++, Smalltalk, CLOS, în există distincţie între obiectul generic "**clasa**" şi obiectul termenul "**instanţa**";

(2) limbaje bazate pe prototipuri, cum ar fi XRL [1] şi SELF [4], în care această distincţie dispare.

Cele două clase (1) si (2), corespund celor două modele generice ale moştenirii: **instanţiere** şi, respectiv, **delegare**. Aşa cum se arată în [5], cele două modele sunt echivalente (cu unele restricţii) adică, "delegation" include "inheritance", dar şi "inheritance" poate include "delegation", în sensul că o ierarhie de obiecte de tip "delegation" poate fi mapată pe o ierarhie de tip "inheritance" **fără instanţe**.

Astfel, se poate considera că, alegerea modelului se va face în funcţie de cerinţele problemei de rezolvat, întrucât ceea ce poate fi exprimat într-un model, poate fi transpus şi în celălalt şi, reciproc. Totuşi, la o privire subiectivă asupra modelelor, se poate observa că, deşi echivalente, modelul "delegation" oferă o flexibilitate mai mare şi o posibilitate de manipulare mai uşoară şi mai apropiată de realitatea înconjurătoare a problemelor.

Astfel, în modelul "inheritance", atributele pe care le vor avea obiectele sunt definite în clase, iar spaţiul pentru memorarea acestor atribute este alocat în instanţe (obiectele propriu-zise). Acest model permite definirea incrementală a claselor, dar o instanţă nu poate fi definită direct, în termenii unei alte instanţe. Deoarece toate instanţele folosesc definiţiile din clasă,

modificarea atributelor clasei duce la modificarea tuturor instanţelor.

Cu toate acestea, valorile memorate în instanţă sunt independente, deci, schimbând starea unei instanţe, nu vom afecta nici o altă instanţă.

Pe de altă parte, modelul "delegation", elimină toate aceste neajunsuri, un obiect fiind o entitate de sine stătătoare, putând fi definit şi utilizat în orice moment şi context se doreşte aceasta. În acest model, obiectele sunt considerate **"instances without classes"** (instanţe fără clase) [5]. De fapt, comportarea obiectelor este apropiată de cea a claselor, dar, în plus, un obiect poate fi definit în termenii altui obiect, obiectele neputând însă fi declarate fără a fi memorate. Modelul "delegation" permite partajarea metodelor şi a variabilelor. Dacă un obiect moşteneşte un atribut, metodă sau variabilă de la un prototip, orice schimbare a acestui atribut sau a valorii sale, va afecta atât prototipul cât şi obiectul. În acest fel, obiectele, într-o ierarhie de tip "delegation", pot fi făcute dependente unele faţă de altele.

Un dezavantaj al acestui model este faptul că nu există noţiunea de grupare după tip.

În plus, modelul "delegation" permite definirea dinamică a obiectelor, fapt deloc de neglijat şi mai ales conferă obiectelor structurate proprietatea de vitalitate (**"vividness"**) [2], respectând totodată axioma de compatibilitate "în sus " (**"Strict Is-A Rule"**) [5] , ceea ce asigură o bază formală solidă a modelului.

2.1 Sintaxa limbajului SELF

Suportul obiectual pe care s-a construit sistemul de rafinare este o versiune extinsă a limbajului SELF. În plus faţă de [4], varianta prezentată aici permite moştenire multiplă, oferă o tratare unitară a sloturilor şi metodelor şi permite modificarea dinamică a supers-ilor unui obiect.

În definirea sintaxei, s-a plecat de la două deziderate: simplitate şi declarativitate în definirea unui obiect, pentru a permite o uşoară transpunere a problemei de rezolvat şi pentru a facilita întreţinerea codului.

Descrierea în BNF a sintaxei unui obiect :

```
<obiect>        ::=( obj     [ <nume-obiect> ]
                        ( { <super-obiect> } * )
                        { <nume slot><valoare-
                        slot> } *
                    )

<nume-obiect>  ::= simbol COMMON LISP

<super-obiect> ::= <nume-obiect>
                   | <obiect >
                   | <expr>

<expr>          ::= forma COMMON LISP
                    evaluabilă la un
                    <obiect> sau simbol

<nume-slot>     ::= simbol COMMON LISP

<valoare-slot>  ::= forma COMMON LISP
```

În majoritatea sistemelor orientate obiect existente la ora actuală, există o diferenţă netă între noţiunile de slot şi metodă. Acest fapt, nu neaparat negativ, introduce o limitare a nivelului de abstractizare a obiectelor, creând totodată şi unele "reţete" fixe ce trebuie urmate în definirea obiectului, cum ar fi : întâi se defineşte obiectul care conţine definirea sloturilor şi declararea metodelor, apoi se definesc metodele.

În limbajul utilizat în lucrarea de faţă, nu există o diferenţă efectivă între slot şi metodă, un obiect având ca părţi constitutive doar sloturi. Aceste sloturi pot fi accesate în două moduri:

(1) se accesează **valoarea** slotului

(2) se activează **comportarea** slotului

Astfel, pentru un slot, avem definite două **"faţete"**: **faţeta valoare** şi **faţeta comportare**. Un **slot-valoare** (care nu are asociată o comportare specifică, i.e., în termeni clasici, **slot**), va răspunde în acelaşi fel la ambele tipuri de acces (i.e. va arata la fel , indiferent de faţeta prin care este privit). Un **slot-metodă** (i.e., în termeni clasici, metoda), va fi văzut prin faţeta comportare ca o funcţie, iar accesul lui în acest mod va avea ca efect lateral apelul funcţiei, rezultatul accesului fiind rezultatul întors de funcţie. Văzut prin faţeta valoare, un slot-metodă, va răspunde similar unuislot-valoare, întorcând rezultatul evaluării (în sens COMMON LISP) a entităţii asociate (în cele mai multe cazuri, o lambda expresie).

În acest mod se introduce acea "unitate în diversitate" care conferă limbajului forţa şi puterea simplităţii [4].

Exemplu

```
(obj obiect ( )
   slot   'valoare-slot
   metoda '(lambda (x) (format t "~% Eu   sunt \"
METODA \"
                        din ~A" (fname x)))
   )

   Acces pe faţeta valoare :

   (fslot 'slot 'obiect)
   ==> VALOARE-SLOT
   (fslot 'metoda 'obiect)
```

```
==> (lambda (x) (format t "~% Eu sunt \"
METODA \" din ~A"
                    (fname x)))
```

Acces pe faţeta comportare :

```
(msg 'slot 'obiect)
==> VALOARE-SLOT
(msg 'metoda 'obiect)
==> Eu sunt METODA din OBIECT
```

Interfaţa de comunicare cu reprezentarea internă a unui obiect este în mod elegant rezolvată prin mecanismul de "**message passing**" (transmitere de mesaje).

Orice sistem orientat obiect trebuie să aibă încorporat un astfel de mecanism. În sistemul de faţă, mecanismul este asemănător (ca sintaxă şi funcţii de interfaţare) cu cel prezentat în [1].

Trimiterea unui mesaj unui obiect se face cu funcţia generică "**MSG**". Aceasta este implementată intern ca un macro care va asigura apelul metodelor obiectului, care răspund la mesajul în cauză, transmiţându-le totodată lista de parametri actuali şi asigurând prezenţa obiectului pe prima poziţie a listei de parametri.

Cele spuse mai sus impun ca, la definirea unui slot-metoda, pentru un obiect, să se aibă în vedere că primul argument va fi, în mod obligatoriu, obiectul care a apelat metoda (pointerul ascuns "**this**" care, în C++, se transmite fiecarei metode a unui obiect).

Pentru a putea răspunde unui mesaj, un obiect trebuie să aibă asociat un slot-metodă fiecărui mesaj căruia doreşte să-i răspundă. Definirea unui astfel de slot-metodă se face prin specificarea la defini-rea obiectului (sau adăugarea ulterioară) a unui slot ce are pe poziţia de "valoare-slot" o formă COMMON LISP care satisface predicatul predefinit "**functionp**".

Exemplu: trei modalităţi echivalente de definire a unui slot-metodă:

```
(obj obiect ( )
   met1  '(lambda (self) (format t "~% Metoda \"
MET1 \" din
                          ~A" (fname self)))
   met2  '#(lambda (self) (format t "~% Metoda \"
MET2 \" din
                          ~A" (fname self)))
   met3  'metoda3
   )
(defun metoda3 (self) (format t "~% Metoda \"
MET3 \" din
                          ~A" (fname self)))
```

Un aspect mult controversat, ce trebuie discutat înaintea proiectării unui sistem orientat obiect este cel al schemei de moştenire folosită.

În funcţie de acest aspect, diferitele sisteme orientate obiect existente se împart în diferite clase, conform modului rezolvare al faţetelor acestei probleme.

Trebuie specificat a priori că, orice sistem orientat obiect, pentru a putea fi considerat ca atare, trebuie să asigure respectarea "**Axiomei de Compatibilitate în sus a claselor**" (numită şi "**Strict Is-A Rule**") [5].

O primă faţetă a problemei, este aceea a moştenirii multiple. Majoritatea sistemelor OOP dezvoltate permit acest tip de moştenire (C++, CLOS, XRL), aşa încât este greu de imaginat că se va mai proiecta vreodată un sistem în care relaţia de moştenire să fie binară.

O altă faţetă a moştenirii este combinarea valorilor moştenite. După acest criteriu, sistemele OOP se disociază în două grupe distincte: cele care nu permit acest lucru (C++) şi cele care oferă această facilitate (CLOS şi, în general, toate sistemele OOP, dezvoltate în LISP), considerate "**puternice**".

Un aspect nou, valid doar în cazul sistemelor orientate obiect ce fac parte din modelul "delegation", este modificarea dinamică a ierarhiei de obiecte. Astfel, merită specificat faptul că, după cunoştinţele autorului acestei lucrări, nu există (pâna la sistemul prezentat aici) un astfel de sistem.

Sistemul prezentat în această lucrare respectă axioma de compatibilitate, permite moştenire multiplă şi oferă posibilitatea combinării valorilor moştenite, atât la nivel de slot, cât şi la nivel de metodă.

În plus (o inovaţie), se oferă posibilitatea schimbării arborelui de moştenire, deci se creează astfel oportunitatea modificării dinamice a ierarhiilor de obiecte.

Într-un sistem orientat obiect, în general, şi, în particular, în cele scrise în LISP, un obiect, odată creat, rămâne activ până la momentul distrugerii lui (implicit sau explicit) sau până în momentul în care el nu mai este referit de niciunde. Se pune astfel problema: "Cum decid dacă este sau nu referit ? ".

Soluţia adoptată este aceea de a permite contexte multiple pentru un obiect . Se va memora deci o lista de perechi cu punct de forma **(<obiect> . <slot>).**

În acest mod se asigură un control strict al obiectelor din sistem, în sensul că, pentru orice obiect, putem şti în ce conexiuni apare, ceea ce conduce la menţinerea entropiei la un nivel scăzut şi deci la o mare stabilitate a sistemului.

2.2 Funcţii de interfaţare

În proiectarea interfeţei sistemului, s-a pornit de la [1], păstrându-se sintaxa şi semantica funcţiilor comune, pentru a asigura o portabilitate sporită a sistemului. În continuare, se vor prezenta (pe scurt) funcţiile oferite de sistem.

(**obj** [<nume>] ([{ <super> } *])
[{ <slot> <valoare> } *]
)

Funcţia defineşte un obiect şi apelează automat demonul **after-create** (dacă există).

Dacă <nume> nu apare, se generează un obiect anonim.

(**lpc** <obiect>) -- întoarce lista de precedentă a obiectului.

(**slots** <obiect> [<opţiuni>])-- întoarce o copie după lista sloturilor proprii.

(**fslot** <slot> <obiect> [<optiuni>]) -- intoarce valoarea slotului <slot> .

(**msg** <tip> <obiect> [<argumente &optiuni>]) -- asigură mecanismul de **message-passing** între obiecte, precum şi activarea automată a demonilor.

(**pslot** <slot> <obiect> <valoare> [<optiuni>]) -- modifică sau (dacă nu exista deja) adaugă un slot unui obiect.

(**get-supers** <obiect>) -- întoarce o copie după lista de supersi a obiectului specificat.

(**set-supers** <obiect> <super-list>) -- modifică lista de supersi a obiectului, menţinând totodată consistenţa ierarhiei de obiecte.

(**subs** <obiect>) -- întoarce lista obiectelor (clone) care-l au drept prototip pe <obiect>.

(**freeze** <obiect> [<optiuni>]) -- " îngheaţă" un obiect (obiectul va deveni "**owner**" al tuturor sloturilor la care are acces.

(**kill** <obiect> [<optiuni>]) -- "omoară" un obiect (îl elimină din ierarhia de obiecte din sistem.

(**fname** <obiect>) -- întoarce numele obiectului <obiect>

(**fobj** <nume>) -- întoarce obiectul cu numele <nume>

(**ctx** <obnume> [<opţiuni>]) --întoarce contextul complet al obiectului.

(**ctxn** <obnume> [<slot>]) -- întoarce nodul context al obiectului.

(**ctxs** <obnume> [<nod>]) -- întoarce slotul context al obiectului.

(**isa** <ob1> <ob2>) -- verifică dacă <ob1> este derivat (direct sau nu) din <ob2>.

2.3 Concluzii asupra sistemului .de obiecte SELF

Sistemul orientat obiect, prezentat mai sus, este de tip prototip -clonă (realizat după modelul de moştenire de tip "**delegation**"). El oferă, în plus faţă de alte sisteme, unele faciliăţti ce pot uşura considerabil munca de proiectare a sistemelor expert. Dintre acestea, se menţionează: simplitatea limbajului, care se traduce, mai ales în structura unui obiect (în fapt, un obiect are doar două părţi constitutive: părinţi şi sloturi). Acest deziderat, care conferă limbajului forţa inegalabilă a simplităţii, a fost realizat prin unificarea conceptelor de slot şi de metodă.

Un aspect nou constă în introducerea contextelor multiple pentru un obiect, precum şi în posibilitatea modificării dinamice a ierarhiei de obiecte.

Aplicarea acestor facilităţi în procesul de proiectare a sistemelor expert prin rafinarea cunoştinţelor va fi descrisă în continuare.

3. Consideraţii generale asupra rafinării

Procesul de rafinare poate fi descris **informal** ca un proces de reducere treptată a gradului de generalitate a unei specificaţii. Prin specificaţie se înţelege orice descriere (mai specifică sau mai generică) a unei probleme. Rafinarea concurentă adaugă problemei o nouă dimensiune: aceea de concurenţă a proceselor (**taskurilor**) de rafinare.

Pentru implementarea sistemului de rafinare, s-a folosit (ca şi în [2]) o arhitectură de rezolvare a problemelor, bazată pe tablă ("**blackboard**").

O arhitectură de tip "blackboad" are la bază trei presupuneri ([3]):

(1) toate soluţiile parţiale, generate pe parcursul procesului de rezolvare, sunt înregistrate într-o structură globală de cunoştinţe, numită "**blackboard**";

(2) soluţiile parţiale sunt generate şi înregistrate pe tabla de procese independente, numite "agenţi" sau "surse de cunoştinţe" ("**knowledge sources** ").

(3) la fiecare pas al procesului de rezolvare a problemei, un mecanism de planificare alege un singur task, pe care îl distribuie unei surse de cunoştinţe care ştie să-l trateze.

3.1. Arhitectura sistemului de rafinare

Sistemul de rafinare poate fi descris astfel:

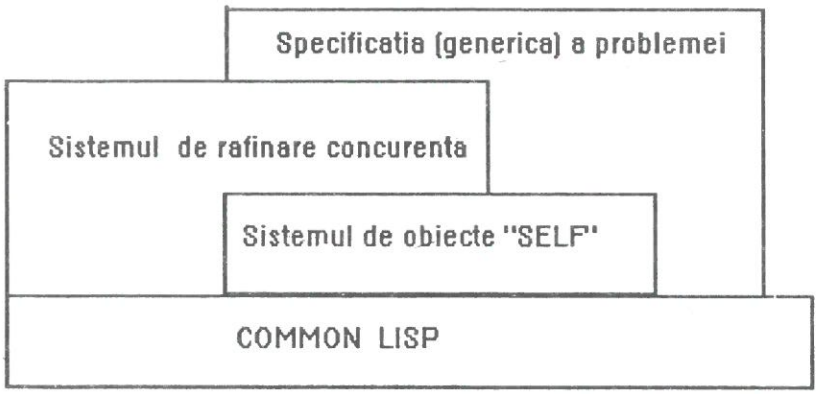

Figura 1.

Figura de mai sus reprezintă o schemă bloc a sistemului de rafinare şi a poziţiei pe care o ocupă fiecare element din sistemul de rafinare.

În general, un sistem de rafinare [2] are ca elemente constitutive:

(1) Tabla -- zona de lucru pe care sunt construite instanţierile obiectelor;

(2) Agende -- colecţii ordonate de procese de rafinare;

(3) Sisteme de rafinare -- pot avea mai multe agende şi poate lucra pe mai multe spaţii pe tablă;

(4) Interpretorul de rafinare-- planificatorul pentru execuţia proceselor din agende.

Pornind de la această descriere generală a rafinării, se prezintă caracteristicile sistemului de faţă, relativ la alte reprezentări.

Caracteristica de bază a abordării prezentate aici este obiectualizarea.

În sistem există, practic, o singură ierarhie de obiecte pe care se fac toate prelucrările necesare rafinării.

Fiecare sistem de rafinare este desemnat printr-un obiect, care ştie să prelucreze taskuri de tipul respectiv. Ierarhia de bază din sistem este cea a taskurilor, care înglobează practic toate informaţiile necesare rafinării.

Definiţie: Un task este un obiect care are la baza arborelui de moştenire obiectul generic "task" (i.e. satisface **(isa obiect 'task)**) şi, de asemenea, este derivat dintr-un obiect (demon) de rafinare (i.e. satisface **(isa obiect 'refine)**).

Un task va avea astfel doi părinţi:

(1) taskul "**tata** " care la creat, folosit în procesul de rafinare, pe ramura de revenire (colectare de rezultate);

(2) demonul (agentul) de rafinare care ştie să rafineze taskul respectiv.

În plus, în sistem mai există un obiect "**blackboard**" care are câte un slot pentru fiecare demon de rafinare.

Fiecare demon de rafinare, task şi obiectul "**blackboard**" ştiu să răspundă la mesaje de tipul:

'init -- pentru iniţializarea procesului de rafinare la nivelul propriu

'refine -- pentru a executa un pas al rafinării

'again -- pentru rafinări repetate

'done -- pentru terminarea procesului de rafinare

'save-result -- pentru salvarea (în forma obiectuală) a rezultatului rafinării (pentru o eventuală adăugare la baza de cunoştinţe).

Avantajele acestei organizări sunt:

(1) agenda de taskuri nu necesită o reprezentare explicită, ea fiind înglobată în ierarhia de taskuri. Astfel, plecând de la sloturile obiectului "blackboard", se poate construi dinamic agenda de taskuri, folosindu-se de facilitatea sistemului de obiecte (SELF) de a obţine, pentru un obiect, obiectele de la care moşteneşte (**supersii**) şi a obiectelor care moştenesc de la el (**subs**). Odată construită agenda de taskuri, la nivel de tabla tot ce rămâne de facut constă în a trimite un mesaj **'refine** fiecarui task din agendă;

(2) fiecare task este "conştient" asupra pozitiei sale in ierarhie, deci responsabilitatea creării de noi taskuri, precum şi cea a actualizării rezultatului rafinării (pe ramura de revenire) poate fi lăsată în seama taskului. La nivel de task, la primirea unui mesaj de tip **'refine**, tot ce trebuie făcut constă în a afla care demon de rafinare este responsabil de rafinarea taskului curent (folosind funcţia SELF **get-supers**) şi apoi în redirecţionarea acestui mesaj către demonul respectiv;

(3) dacă apare necesitatea schimbării modului de rafinare a unui obiect, în abordarea prezentată, lucrurile se rezolvă foarte elegant, modificând genealogia taskului răspunzător pentru rafinarea obiectului respectiv (graţie facilităţii sistemului de obiecte de a păstra consistenţa ierarhiei după modificarea părinţilor unui obiect.

Un task se poate afla în şase stări: new, active, suspended, waiting, done şi poisoned [2].

Rafinarea unui task va consta deci într-o tranziţie a taskului prin aceste stări, efectuând în acelaşi timp şi prelucrările specifice fiecărui task şi demon de rafinare, până la ajungerea în starea done sau poisoned.

3.2 Exemplu comentat

Se consideră problema proiectării unei case. Se presupune, pentru simplitate, că vor fi doar patru camere: o sufragerie, un dormitor, o bucătărie şi o baie. O condiţie suplimentară este ca baia să aibă aceeaşi orientare cu bucătăria.

Specificaţia generică a problemei ar putea arăta astfel:

```
(obj   House  ( )
      modeR            'expand
      verify           'verify-house
      after-refine     'after-refine-house
      before-refine    'before-refine-house
      sufragerie       (obj  ( living-room ) )
      dormitor         (obj  ( bed-room ) )
      bucătărie        (obj  ( kitchen) )
      baie              (obj  ( bath-room )
                  path  '(House bucătărie
                  orientation)
                  orientation '(lambda (x)
                (equal (fslot 'path x)
                  (fslot 'orientation x))
                        )
)
```

În descrierea de mai sus apar unitar sloturi-valoare şi sloturi-metode.

Slotul **modeR** este tratat special de către sistemul de rafinare, identificând modul de rafinare. Sloturile **verify, before-refine** şi **after-refine** identifică comportări ale obiectului House şi anume: metoda de verificare a rezultatului rafinării, demonul ce se va aplica înaintea începerii rafinării şi respectiv demonul ce se va aplica după terminarea rafinării .

Se presupune că se doreşte ca preţul total al casei să fie mai mic decât o anumită sumă. În acest caz, demonul before-refine va cere utilizatorului suma în care trebuie să se încadreze costul total al casei:

```
(defun  before-refine-house (self )
      (format t "~% What is the total price that
you have planed to spend for this house ? ")
      (pslot 'total-price self (read))
)
```

Metoda "verify" va verifica dacă suma costurilor camerelor casei este mai mică decât suma totală de care dispune cumpărătorul :

```
(defun verify-house (self rez)
    (< (apply '+ (slots rez)) (fslot 'price self)))
)
```

Metoda "after-refine" va adăuga un slot în rezultat, reprezentând costul total al casei:

```
(defun after-refine-house (self rez)
    (pslot 'total-price rez (apply '+ (slots rez)))
)
```

Dintre sloturile obiectului House, se exemplifică descrierea sufrageriei :

```
(obj   living-room  ( room )
  orientation  'west
)
```

Unde, obiectul generic **room** este descris astfel:

```
(obj room ( )
   modeR            'anchor
   before-refine    'before-refine-room
   after-refine     'after-refine-room
   price            '*a-particular-price*
   orientation      '*a-particular-orientation*
)
```

Slotul **'path** din specificarea generică a obiectului "baie" impune ca rafinarea slotului "baie" al casei să nu se facă decât după ce s-a terminat rafinarea slotului "bucătărie", iar slotul **'verify** impune ca baia şi bucătăria să aibă aceeaşi orientare (aceeaşi valoare pentru slotul orientation).

Presupunând ca taskurile create prin expandarea obiectului House, s-au terminat cu succes (s-au ancorat la obiecte existente în baza de cunştinţe din sistem), ierarhia de taskuri din sistem va fi conform celei din figura 2.

În pasul următor, se vor "înghiţi" sloturile **result** din taskurile derivate din taskul **self31** (răspunzator pentru rafinarea lui House) şi se va construi un nou obiect anonim (**self36**), care va fi pus pe slotul **result** al taskului **self31**. Acesta este deci rezultatul rafinării.

Pentru a se vedea dacă mai există şi alte soluţii de rafinare pentru obiectul House, se poate trimite obiectului "blackboard" mesajul **'again**.

Acest mesaj va folosi ierarhia deja construită de taskuri pentru a determina dacă se mai poate face o rafinare şi, în caz afirmativ, o va face, rerafinând doar taskurile necesare.

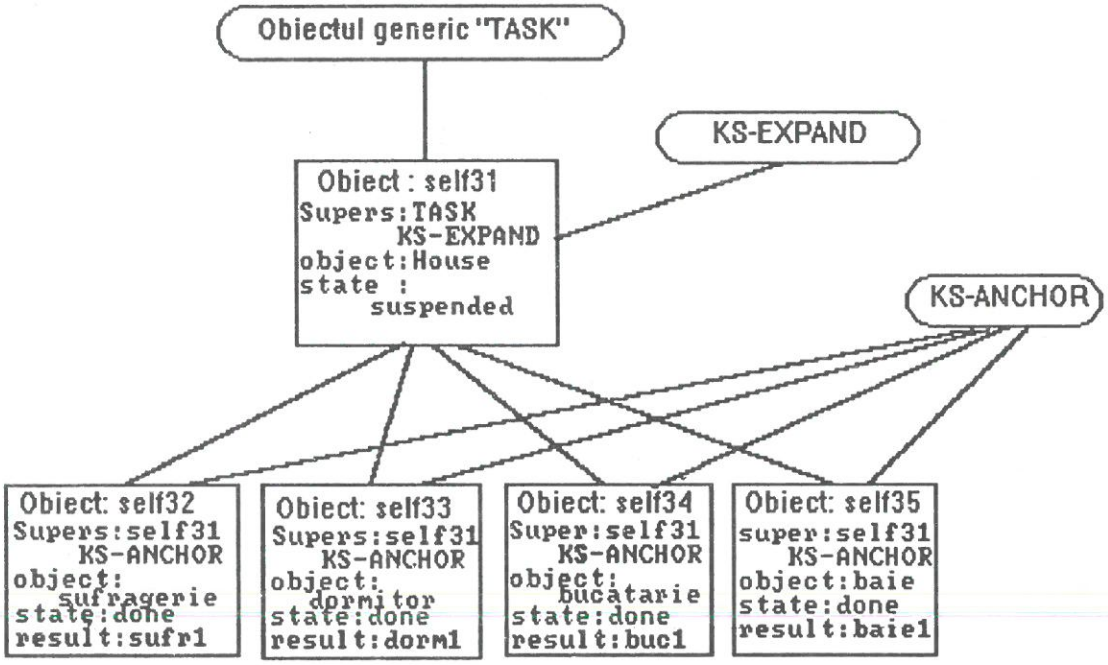

Figura 2.

3.3 Concluzii rafinare

Sistemul de rafinare propus aici păstrează specificaţia de referinţă a unui sistem de acest tip, dar încearcă o inovaţie în privinţa modului dereprezentare a structurilor necesare în procesul de rafinare (tablă, agende, agenţi de rafinare). Pentru realizarea acestui lucru, s-a implementat un sistem de reprezentare a cunoştinţelor prin obiecte structurate (frame), bazat pe modelul "delegation", cu facilităţi de schimbare dinamică a arborelui de moştenire pentru un obiect.

Acest lucru a făcut posibilă eliminarea (reprezentarea implicită) a agendei de taskuri, precum şi tratarea unitară a demonilor de rafinare (folosind avantajele unei abordări pur obiectuale).

Bibliografie

1. **BĂRBUCEANU, M., TRĂUŞAN - MATU, ŞT.** - XRL: A Layered Knowledge Processing Architecture Able to Enhance itself, Studies and Researches in Computers and Informatics, vol. 1, No. 1, Bucureşti, pp. 76-106.
2. **BĂRBUCEANU, M., TRĂUŞAN - MATU, ŞT., MOLNAR, B.**: Concurrent Refinement: A model and Shell for Hierarchical Problem Solving,. În :J.C. Rault (ed.), Proc. of 10-th Workshop on Expert Systems and Their Appications, Avignon, Franţa, 1990.
3. **HAYES - ROTH, B.**: A Blackboard Architecture for Control, Heuristic Programing Project, Stanford University.
4. **SMITH RANDALL, B., UNGAR, D.**:Self: The Power of Simplicity, OOPSLA '87 Proceedings, October 4-8, 1987.
5. **STEIN, L. A.**: Delegation Is Inheritance, OOPSLA '87 Proceedings, October 4-8, 1987.
6. **TRĂUŞAN - MATU, ŞT**: Limbaje de programare evoluată. Note de Curs, UPB, 1994.
7. **TRĂUŞAN - MATU, ŞT**: Sisteme experte. Note de Curs, UPB, 1994.

INSTITUTUL DE CERCETĂRI ÎN INFORMATICĂ
B-dul Mareşal Averescu, Sect. 1, cod 71316, Bucureşti-Romania

RESEARCH INSTITUTE FOR INFORMATICS
8-10 Averescu Avenue, 71316, Bucharest 1, ROMANIA

LEXMED

Mediu de specificare formală a sistemelor concurente

Descriere produs:

LEXMED este un mediu de specificare a sistemelor concurente, care utilizează un limbaj propriu formal, concurent, de specificare executabilă, numit LEX.

Un sistem concurent se poate descrie (specifica) prin:

a) o maşină de stări (numită doer în limbajul LEX);

b) mai mulţi doer-i care comunică între ei;

c) o combinaţie între a) şi b).

Un doer controlează accesul la resurse (date), care sunt reprezentate prin algebre (numite teorii în limbajul LEX). Doer-ii comunică prin transmitere şi recepţionare de mesaje. LEX este un limbaj formal de spectru larg, bazat pe un model algebric (pentru date) şi un model operaţional (pentru procese). De aceea corectitudinea specificaţiilor executabile concurente, scrise în LEX poate fi verificată formal. Este astfel posibilă detectaea erorilor încă din primele faze ale ciclului de viaţă a unui produs, ceea ce duce la scăderea costurilor de întreţinere. Prin nivelul lor ridicat de abstractizare, specificaţiile obţinute pot fi reutilizate pentru obţinerea unor produse similare.

Domenii de aplicabilitate:

LEXMED poate fi utilizat pentru specificarea sistemelor cu grad ridicat de criticitate: Sisteme de prelucrare a tranzacţiilor (de exemplu, sistem de rezervare a biletelor) sisteme de monitorizare şi control (de exemplu, pentru traficul aerian, rutier).

Configuraţie hardware şi software:

- Microcalculator compatibil IMB-PC, 640K RAM, 5 Mbytes spaţiu pe disc
- SO MS-DOS 4.1 sau ulterior.

Punct de contact:

lab.1.3. Ileana Rabega tel.665.60.60/int.141.

Phone: +40-1-665.26.60
Telex: 11891 icpcir
Fax: +40-1-212.07.11

INSTITUTUL DE CERCETĂRI ÎN INFORMATICĂ
B-dul Mareşal Averescu, Sect. 1, cod 71316, Bucureşti-Romania

RESEARCH INSTITUTE FOR INFORMATICS
8-10 Averescu Avenue, 71316, Bucharest 1, ROMANIA

ESYS

Sistem interactiv de elicitare a cunoştinţelor folosind teoria construcţiilor personale

Descriere produs:

ESYS este uns sistem interactiv de achiziţie a cunoştinţelor bazat pe teoria construcţiilor personale şi tehnici de învăţare inductivă.

Domenii de aplicabilitate:

Sistemul poate fi folosit pentru elicitarea cunoştinţelor şi construirea bazelor de cunoştinţe în practic orice domeniu.

În consecinţă ESYS poate fi utilizat ca modul de achiziţie a cunoştinţelor pentru sisteme experte bazate pe reguli de producţie, sau ca sistem-suport de decizie de sine stătător.

Configuraţie hardware şi software:

- microcalculator compatibil IBM-PC 386/486
- minimun 3Mb RAM
- SO MS-DOS 3.3 sau ulterior cu interfaţa Windows 3.0 sau 3.1.

Punct de contact:

lab.1.3. Doina Ţilirea tel.665.60.60/int.161

Phone: +40-1-665.26.60
Telex: 11891 icpcir
Fax: +40-1-212.07.11