

CAPITOLUL III

ARHITECTURI DE AGENȚI

3.1 Arhitecturi abstracte pentru agenți inteligenți

În acest paragraf vom prezenta câteva modele generale și proprietăți ale agenților, fără a exista vreo legătură cu modul de implementare al acestora. Se va realiza și o formalizare a prezentării *agenților inteligenți*. Vom presupune că stările ce compun mediul unui agent pot fi descrise sub forma unei mulțimi $S = \{s_1, s_2, \dots\}$, numite *stările mediului*. La orice moment, mediul se presupune a fi într-una din aceste stări. Capacitatea de acțiune a agentului se presupune a fi reprezentată sub forma unei mulțimi $A = \{a_1, a_2, \dots\}$ de *acțiuni*.

Astfel, din punct de vedere abstract un *agent* poate fi privit ca fiind o funcție

$$agent: S^* \rightarrow A,$$

funcție care asignează acțiuni unor secvențe de stări ale mediului. Un agent modelat de o astfel de funcție se va numi *agent standard*. Intuitiv, un agent va decide ce acțiuni să efectueze pe baza experienței sale anterioare. Aceste experiențe sunt reprezentate sub forma unor secvențe de stări ale mediului – acele stări pe care agentul le-a întâlnit deja.

Comportamentul (nedeterminist) al mediului poate fi modelat sub forma unei funcții

$$mediu: S \times A \rightarrow \wp(S),$$

care asignează unei perechi (s, a) o mulțime de stări $mediu(s, a)$ – stările ce pot rezulta în urma efectuării acțiunii a în starea s . Dacă toate elementele codomeniului au cardinalul unitar, atunci mediul se va numi *determinist*, iar comportamentul mediului poate fi stabilit cu exactitate.

Interacțiunea unui agent cu mediul său se poate reprezenta sub forma unui *istoric*

$$h: s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \dots,$$

s_0 fiind starea inițială a mediului, a_u fiind cea de-a u -a acțiune efectuată, iar s_u fiind cea de-a u -a stare a mediului.

Dacă $agent: S^* \rightarrow A$ este un *agent*, $mediu: S \times A \rightarrow \wp(S)$ este un *mediu*, iar s_0 este starea inițială a mediului, atunci secvența

$$h: s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \dots$$

reprezintă un posibil *istoric* al agentului în mediu, dacă și numai dacă următoarele două condiții sunt verificate:

1. $\forall u \in \mathbb{N}, a_u = agent((s_0, s_1, \dots, s_u))$
2. $\forall u \in \mathbb{N}$ astfel încât $u > 0$, $s_u \in mediu(s_{u-1}, a_{u-1})$

Comportamentul specific al unui agent $agent: S^* \rightarrow A$ într-un mediu $mediu: S \times A \rightarrow \wp(S)$ este mulțimea tuturor “istoricelor” care satisfac proprietățile anterioare. Dacă o anumită proprietate Φ este valabilă pentru fiecare istoric al agentului, atunci Φ va fi considerată ca fiind o *proprietate invariantă* a agentului în mediu.

Se va nota $ist(agent, mediu)$ mulțimea tuturor “istoricelor” agentului în mediu. În acest sens, doi agenți ag_1 și ag_2 se vor numi *comportamental echivalenți* în raport cu mediul $mediu$, dacă $ist(ag_1, mediu) = ist(ag_2, mediu)$ și *comportamental echivalenți* dacă sunt comportamental echivalenți în raport cu orice mediu.

În general, sunt de un mai mare interes agenții ale căror interacțiuni cu propriul mediu nu se termină, altfel spus al căror comportament specific este *infinit*.

3.1.1 Agenți total reactivi (purely reactive agents)

Anumite categorii de agenți decid ce acțiune să aleagă la un moment dat, fără a ține cont de *istoricul* lor, altfel spus luarea unei decizii nu are nici o legătură cu acțiunile trecute ale agentului. Un astfel de agent se va numi *total reactiv*, datorită faptului că poate răspunde *direct* mediului în care acționează.

Formal, comportamentul unui astfel de agent va fi o funcție $agent: S \rightarrow A$.

Ceea ce înseamnă că, pentru fiecare agent total reactiv există un agent standard echivalent; reciproca însă, nu este, în general, valabilă.

Un exemplu de astfel de agent total reactiv este un termostat (ce dispune de un senzor pentru a detecta temperatura camerei). Mediul unui astfel de agent este unul *fizic*, acțiunile disponibile fiind doar două, astfel încât

$$agent(s) = \begin{cases} caldura\ PORNITA \\ caldura\ OPRITA \end{cases}$$

și agentul poate la orice moment să reacționeze imediat: decizia despre acțiunea pe care o va efectua este *independentă* de acțiunile sale anterioare.

3.1.2 Percepție

După cum se poate observa, modelul *abstract* al unui agent (după cum a fost prezentat anterior) presupune două aspecte: arhitectura (partea de construcție) agentului și proiectarea (design-ul) funcției de decizie a agentului (funcția *agent*).

Este important ca modelul să fie *rafinat*, adică să împartă agentul în subsisteme, în maniera în care se procedează și în ingineria soft. O analiză simplă a celor două subsisteme ce alcătuiesc un *agent*, conduce la următoarele concluzii:

- *proiectarea (design-ul)* agentului (de fapt partea de *program* a agentului) se referă la datele și structurile de control ce vor fi prezente în descrierea agentului;
- *arhitectura* agentului se referă la partea internă – structurile de date, operațiile care vor trebui aplicate acestor structuri, precum și controlul fluxului între structurile de date.

În primul rând, funcția de decizie a agentului trebuie separată în două subsisteme: subsistemul *percepție* și subsistemul *acțiune* (Figura 3.1):

- funcția *percepție* se referă la abilitatea agentului de a-și observa (percepe) mediul;
- funcția *acțiune* se referă la procesul de aplicare a deciziilor.

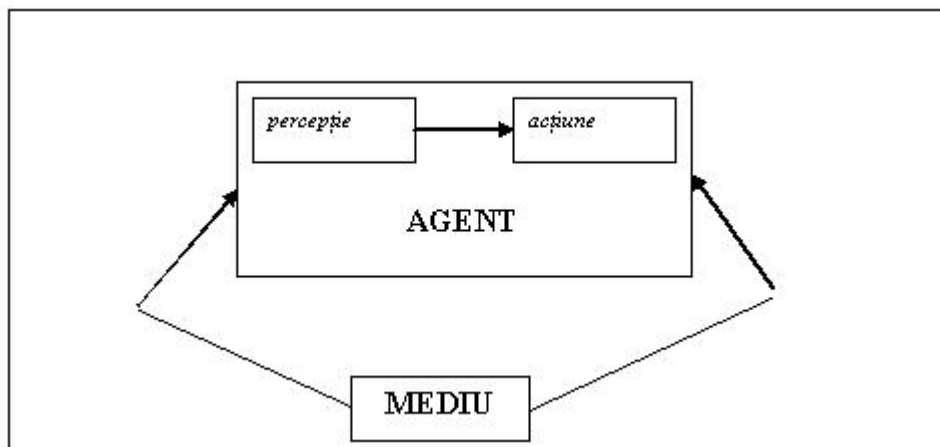


Figura 3.1. Subsistemele *percepție* și *acțiune* ale unui agent

Spre exemplu, pentru un agent soft (programul **xbiff** de sub UNIX – care monitorizează mail-urile primite de utilizator și care indică dacă acesta are sau nu mail-uri necitite), senzorii care percep mediul ar putea fi o serie de comenzi ale sistemului prin care se pot obține informații despre mediul soft (de exemplu comenzile **ls**, **finger**, sau altele).

Observăm faptul că mediul acestui agent este unul *soft*, funcțiile prin care obține informații despre mediul sunt funcții *soft*, iar acțiunile pe care le execută sunt tot acțiuni *soft* (schimbarea unei icoane pe ecran sau executarea unui program).

Rezultatul (ieșirea) funcției *percepție* este un *semnal* (percepție) – o secvență perceptuală primită la intrare. Dacă P este o mulțime nevidă de percepții, atunci funcția *percepție* va fi

$$\text{percepție}: S \rightarrow P,$$

funcție ce asignează stărilor mediului percepții, iar funcția *acțiune* va fi

$$\text{acțiune}: P^* \rightarrow A,$$

funcție ce asignează secvențelor de percepții (semnale) acțiuni.

Definițiile anterioare conduc la următoarea proprietate caracteristică agenților și percepției acestora.

Proprietate Presupunem că $s_1 \in S$ și $s_2 \in S$ sunt două stări ale mediului unui agent A astfel încât $s_1 \neq s_2$, dar $\text{percepție}(s_1) = \text{percepție}(s_2)$ (agentul va primi aceeași informație perceptuală în două stări distincte). În acest caz, agentul *nu poate deosebi* stările s_1 și s_2 (stările fiind *neidentificabile*).

De exemplu, presupunem că avem un agent care are legătură cu doar două fapte p și q (reprezentate sub formă de propoziții logice) despre mediul în care acționează. Mai mult, faptele p și q sunt independente una de alta, iar agentul percepe efectiv (prin

senzorii săi) doar faptul p . În acest caz, putem spune că mulțimea S a stărilor mediului poate fi reprezentată ca având exact 4 (2^2) elemente:

$$S = \{ \underbrace{\{\neg P, \neg Q\}}_{s_1}, \underbrace{\{\neg P, Q\}}_{s_2}, \underbrace{\{P, \neg Q\}}_{s_3}, \underbrace{\{P, Q\}}_{s_4} \}, \text{ stările } s_1 \text{ și } s_2, \text{ respectiv } s_3 \text{ și } s_4, \text{ stări în care } Q$$

sau $\neg Q$ sunt verificate fiind de fapt *neidentificabile*, deși funcția *percepție* a agentului este aceeași în ambele stări.

3.1.3 Agenți cu stări

În secțiunile anterioare s-au prezentat modele de agenți a căror funcție de decizie (*acțiune*) asigna acțiuni unor *secvențe* de stări ale mediului sau unor secvențe de percepții.

În acest paragraf, se vor prezenta agenți care își *păstrează stările* – Figura 2.4. Acești agenți au o anumită structură de date internă, reprezentând starea internă a agentului. În cele ce urmează, vom nota cu I mulțimea stărilor interne ale agentului.

Funcția *percepție* pentru un agent cu stări rămâne nemodificată, asignând stărilor mediului percepții.

$$\text{percepție}: S \rightarrow P$$

Funcția de selecție a acțiunii va fi definită astfel:

$$\text{acțiune}: I \rightarrow A,$$

ca o asignare de acțiuni *stărilor interne* ale agentului. Va fi introdusă și o funcție adițională, funcția *următor*, care asignează *stări interne* unor perechi (*stare internă*, *percepție*)

$$\text{următor}: I \times P \rightarrow I.$$

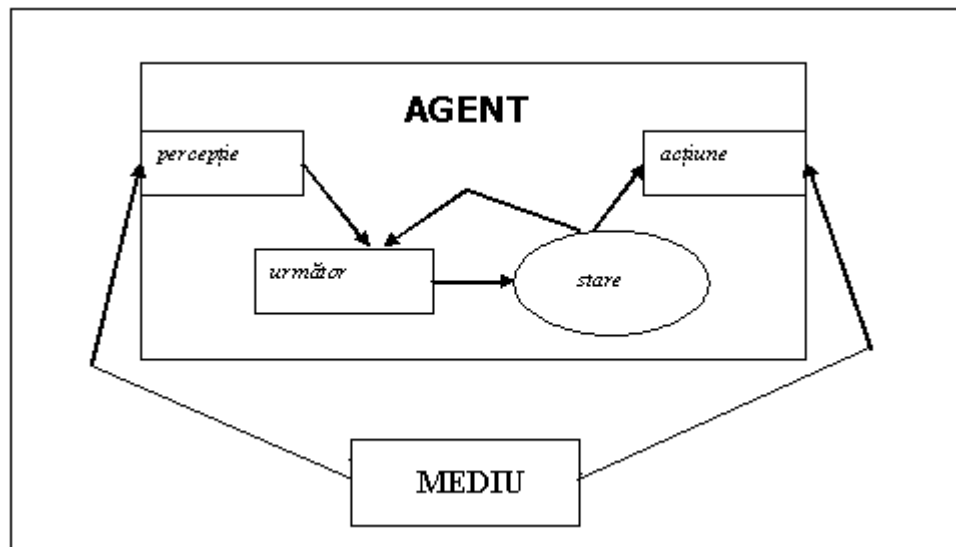


Figura 3.2. Agenți cu stări

Comportamentul unui agent bazat pe stări poate fi rezumat astfel:

- agentul pornește cu o stare internă inițială i_0 ;
- observă starea mediului s și generează un semnal (percepție) $percepție(s)$;
- starea internă a agentului va fi actualizată conform funcției *următor*, devenind $următor(i_0, percepție(s))$;
- acțiunea selectată de agent va fi $acțiune(următor(i_0, percepție(s)))$;
- acțiunea va fi executată, după care agentul reia ciclul.

De fapt, se poate observa că *agenții bazați pe stări* sunt *identici* ca putere expresivă cu *agenții standard* – orice agent bazat pe stări poate fi transformat într-un agent standard comportamental echivalent cu acesta.

3.2 Arhitecturi concrete pentru agenți inteligenți

În paragraful anterior, a fost prezentată noțiunea abstractă de *agent inteligent*. Au fost prezentate arhitecturi de agenți care își păstrează sau nu stările, dar nu s-a discutat despre cum ar trebui sau ar putea să arate aceste stări.

Deasemenea, s-a vorbit despre funcția abstractă *agent*, funcție care modelează procesul de aplicare a deciziilor unui agent, dar nu s-a prezentat modul în care o astfel de funcție ar trebui implementată.

În continuare, se vor prezenta aspecte legate de arhitectura *concretă* a unui agent. Vor fi considerate patru clase de bază:

- *agenți bazați pe logică (logic based agents)* – în care procesul de aplicare a deciziilor este realizat pe baza deducțiilor logice;
- *agenți reactivi (reactive agents)* – în care implementarea procesului de luare a deciziilor se bazează pe asignarea de acțiuni diverselor situații;
- *agenți tip opinie-cerere-intenție (belief-desire-intention)* – în care procesul de luare a deciziilor depinde de manipularea structurilor de date reprezentând opiniile, cererile, respectiv intențiile agentului;
- *arhitecturi stratificate (layered architectures)* – în care procesul de luare a deciziilor este realizat prin intermediul a diverse nivele soft, fiecare nivel ocupându-se de raționamentul despre mediu la diferite nivele de abstractizare.

3.2.1 Arhitecturi bazate pe logică

Modul “tradițional” de realizare a sistemelor inteligente artificiale (cunoscut sub denumirea de *Inteligență Artificială simbolică*) sugerează ca și comportamentul inteligent să fie generat în sistem prin furnizarea unei reprezentări *simbolice* a mediului și a comportamentului și o manipulare simbolică a acestei reprezentări.

Conform abordării tradiționale, reprezentarea simbolică se face sub formă de *formule logice*, iar manipularea simbolică corespunde *deducțiilor logice* sau *demonstrării teoremelor*. În această abordare, agentul poate fi considerat ca o *demonstrație a unei teoreme*. (dacă Φ este o teoremă care explică modul în care un agent inteligent trebuie să se comporte, sistemul va trebui să genereze de fapt o succesiune de pași - acțiuni - prin care să se *ajungă* la Φ - de fapt o “demonstrație” a lui Φ).

Va fi prezentat în continuare un model simplu de agent bazat pe logică, agent ce face parte din categoria așa numitor *agenți prudenți (deliberate agents)*. În astfel de arhitecturi, starea internă a agentului se presupune a fi o bază de date cu formule predicative aparținând logicii predicatelor de ordinul I (de exemplu: $temperatura(X,321)$

sau $presiunea(X,28)$). Baza de date reprezintă de fapt *informația* pe care o are agentul despre mediu. De precizat faptul că aceste formule predicative pot sau nu fi valide, atâta timp cât ele reprezintă *percepția* agentului.

Fie L o mulțime de propoziții din logica de ordinul I, și fie $D = \wp(L)$ mulțimea *bazelor de date* din L (o mulțime de mulțimi de formule din L).

- *Starea internă* a agentului va fi dată de un element din D . Vom nota Δ, Δ_1, \dots elementele lui D .
- Procesul de *aplicare a deciziilor* agentului se va modela sub forma unui set de *reguli de deducție*, ρ (reguli de inferență logică). Se va nota $\Delta \xrightarrow{\rho} \Phi$ dacă formula Φ poate fi demonstrată din baza de date Δ folosind doar regulile de deducție din ρ .
- Funcția de *percepție* rămâne neschimbată *percepție*: $S \rightarrow P$.
- Funcția *următor* devine *urmator*: $D \times P \rightarrow D$ - asignează unei perechi (bază de date, percepție) o nouă bază de date.
- Funcția de selectare a acțiunii va fi definită ca o *regulă de deducție*.

$$actiune: D \rightarrow A$$

Definiția pseudo-cod a funcției *actiune* este prezentată în Figura 3.3. Ideea este că programatorul agentului va reprezenta regulile de deducție ρ și baza de date Δ în așa manieră încât dacă o formulă *Executa(a)* poate fi dedusă, a reprezentând o acțiune, atunci a este cea mai bună acțiune ce ar putea fi aleasă de către agent:

- dacă formula *Executa(a)* poate fi demonstrată din baza de date, folosind regulile de deducție ρ , atunci a va fi returnată, ca fiind acțiunea ce va trebui executată;
- dacă nu, se caută o acțiune ce este *consistentă* cu regulile și baza de date, adică o acțiune $a \in A$ ce are proprietatea că $\neg Executa(a)$ nu poate fi dedusă din baza de date folosind regulile de deducție;
- dacă nu reușește nici pasul 2, atunci se returnează *null* (nici o acțiune nu a fost selectată), caz în care comportamentul agentului va fi determinat din regulile de deducție (“programul” agentului) și din baza sa de date curentă (reprezentând informația agentului despre mediul său).

funcția <i>acțiune</i> ($\Delta : D$): <i>A</i> pentru fiecare $a \in A$ execută dacă $\Delta \xrightarrow{\rho} Executa(a)$ atunci <i>acțiune</i> $\leftarrow a$ sf-dacă sf-pentru pentru fiecare $a \in A$ execută dacă $\Delta \xrightarrow{\rho} \neg Executa(a)$ atunci <i>acțiune</i> $\leftarrow a$ sf-dacă sf-pentru <i>acțiune</i> $\leftarrow null$ sf-acțiune
--

Figura 3.3. Definiția pseudo-cod a funcției *acțiune*

Spre exemplu, să considerăm problema unui *agent robotic care trebuie să facă curățenie într-o casă*. Robotul este echipat cu un senzor care îi va spune dacă este sau nu praf sau un aspirator în zona în care se află. În plus, robotul are în orice moment o orientare bine definită (*nord, sud, est* sau *vest*). Mai mult, pentru a putea aspira praful, agentul se poate deplasa înainte cu un “pas”, sau să se întoarcă spre dreapta cu 90^0 . Agentul se va deplasa prin cameră, care este împărțită într-un număr de dreptunghiuri egale (corespunzând unității de deplasare a agentului). Se va presupune că agentul nu face altceva decât să *curețe* – deci nu părăsește camera. Pentru simplificare, caroiajul (camera) se presupune că este de 3×3 , iar agentul pornește întotdeauna din poziția (0,0), fiind cu fața spre *nord* (Figura 3.4).

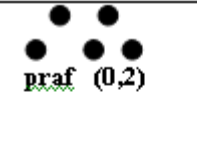
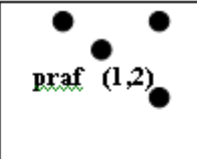
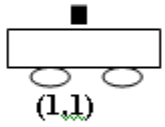
		(2,2)
(0,1)		(2,1)
(0,0)	(1,0)	(2,0)

Figura 3.4. Mediul agentului robotic

Din specificațiile problemei, rezultă că:

- agentul poate primi un semnal (percepție) *praf* (cu semnificația că este praf sub el), respectiv *null* (informație nesemnificativă);

- agentul poate executa una din cele trei acțiuni posibile: *înainte*, *aspiră* sau *întoarce*.

Se vor utiliza trei predicate în vederea rezolvării acestei probleme și anume:

- $\hat{In}(x,y)$ - agentul este în poziția (x,y)
- $Praf(x,y)$ - este praf în poziția (x,y)
- $Orientare(d)$ - agentul e orientat în direcția d

- vom nota $vechi(\Delta)$ informația “veche” din baza de date, pe care funcția *următor* ar trebui să o șteargă

$$vechi(\Delta) = \{P(t_1, t_2, \dots, t_n) \mid P \in \{In, Praf, Orientare\} \text{ si } P(t_1, t_2, \dots, t_n) \in \Delta\}$$

- e necesară o funcție *nou*, care indică mulțimea predicatelor noi ce trebuie adăugate în baza de date $nou: D \times P \rightarrow D$, definită astfel:

$$nou(\Delta, P(t_1, t_2, \dots, t_n)) = \Delta_1 \quad ; \quad \Delta_1 = \Delta \cup \{P(t_1, t_2, \dots, t_n)\},$$

de predicatul $P \in \{In, Praf, Orientare\}$, iar operația “ \cup ” aplicată bazei de date curente semnifică adăugarea unui nou fapt în baza de date.

Folosind notațiile anterioare, funcția *următor* va putea fi definită astfel:

$$urmator(\Delta, p) = (\Delta \setminus vechi(\Delta)) \cup nou(\Delta, p)$$

Urmează a fi specificate regulile ce definesc comportamentul agentului. Regulile de deducție sunt de forma $\phi(\dots) \rightarrow \psi(\dots)$, unde ϕ și ψ sunt formule care conțin predicate având ca argumente liste arbitrare de constante și variabile. Semnificația unei astfel de reguli este următoarea: dacă ϕ se “potrivește” cu baza de date a agentului, atunci ψ poate fi dedus, cu anumite variabile din ψ instanțiate.

Cea mai importantă regulă de deducție se referă la scopul agentului

$$In(x, y) \wedge Praf(x, y) \rightarrow Executa(aspira) \quad (1)$$

Celelalte reguli furnizează algoritmul de deplasare a robotului în mediu.

$$In(0,0) \wedge Orientare(nord) \wedge \neg Praf(0,0) \rightarrow Executa(inainte) \quad (2)$$

$$In(0,1) \wedge Orientare(nord) \wedge \neg Praf(0,1) \rightarrow Executa(inainte) \quad (3)$$

$$In(0,2) \wedge Orientare(nord) \wedge \neg Praf(0,2) \rightarrow Executa(intoarce) \quad (4)$$

$$In(0,2) \wedge Orientare(est) \rightarrow Executa(inainte) \quad (5)$$

De observat că regula prioritară este regula 1, iar reguli asemănătoare regulilor 2-5 trebuie descrise explicit pentru fiecare poziție a robotului. Toate aceste reguli, împreună cu funcția *următor* vor genera comportamentul specific al robotului.

Ca și concluzie, în abordarea *bazată pe logică*, procesul de aplicare a deciziilor este privit ca o *deducție*, partea de program a agentului (strategia de aplicare a deciziilor) este codificată ca o teorie logică, iar procesul de selecție a acțiunii se reduce la o problemă de demonstrație. Abordarea logică este elegantă și are o semantică bine formalizată (logică).

Totuși, câteva dezavantaje ale acestei abordări ar fi:

- complexitatea computațională ce intervine în procesul de demonstrare a teoremelor ridică problema dacă agenții reprezentați astfel pot opera efectiv în medii supuse limitărilor de timp;
- procesul de aplicare a deciziilor în astfel de arhitecturi logice se bazează pe presupunerea că mediul nu își schimbă semnificativ structura în timpul

procesului de decizie (o decizie care este corectă în momentul începerii procesului de decizie, va fi corectă și la sfârșitul acestuia);

- problema reprezentării și raționamentului în medii complexe și dinamice este deasemenea o problemă deschisă.

3.2.2 Arhitecturi reactive

Datorită problemelor care apar în abordarea logico-simbolică, spre sfârșitul anilor '80, cercetătorii au început să investigheze alternative la paradigma logico-simbolică a Inteligenței Artificiale.

Aceste abordări noi, diferite de abordarea logico-simbolică, sunt prezentate ca fiind:

- *comportamentale* – un scop comun este dezvoltarea și combinarea comportamentelor individuale;
- *integrate* – o temă comună este cea a agenților situați (integrați) în anumite medii, nu a celor neintegrați în acestea;
- *reactive* – sistemele sunt deseori percepute ca reacționând pur și simplu la mediul lor, fără raționament asupra acestuia.

Cea mai bună arhitectură reactivă cunoscută până în prezent se consideră a fi arhitectura “*subsumption*” (AS), dezvoltată de Rodney Brooks.

Cele două caracteristici definitorii ale acestei arhitecturi sunt:

- procesul de aplicare a deciziilor agentului se realizează prin intermediul unei mulțimi de *comportamente ce îndeplinesc anumite sarcini*; fiecare comportament poate fi privit ca o funcție individuală de *acțiune* (definită anterior), funcție care asociază unor percepții (recepționate la intrarea în sistem) acțiuni ce urmează a fi executate; fiecare modul “comportamental” realizează anumite sarcini, și se presupune că modulele nu includ *reprezentări simbolice* și nici *raționament simbolic*. În multe implementări, aceste comportamente sunt implementate ca reguli de forma *situație* \rightarrow *acțiune*, reguli ce asignează direct acțiuni unor secvențe perceptuale;
- modulele sunt aranjate într-o *ierarhie (subsumption hierarchy)*, comportamentele fiind dispuse pe *nivele*. Nivelele situate mai jos în ierarhie au o *prioritate* mai mare, cele situate mai sus reprezentând comportamente mai abstracte.

Funcția *percepție*, reprezentând abilitatea perceptuală a agentului, se presupune că rămâne neschimbată.

Funcția de decizie *acțiune* este realizată prin intermediul unei mulțimi de comportamente și a unei relații de *constrângere* existentă între nivele.

Un comportament va fi reprezentat sub forma unei perechi (c, a) , unde $c \subseteq P$ este o mulțime de percepții numită *condiție*, iar $a \in A$ este o acțiune.

Un comportament (c, a) se va spune că *este posibil* într-o stare $s \in S$ a mediului dacă și numai dacă $perceptie(s) \in c$.

În plus, se va nota $Comportament = \{(c, a) \mid c \subseteq P \text{ și } a \in A\}$ mulțimea tuturor regulilor de comportament.

Unei mulțimi de reguli de comportament $R \subseteq Comportament$ i se va asocia o relație binară de *constrângere*, “ \prec ”, $\prec \subseteq R \times R$. Relația se presupune a fi o relație de

ordine totală pe R (tranzitivă, antisimetrică și totală). Se va nota $b_1 \prec b_2$ dacă $(b_1, b_2) \in \prec$ și se va spune “ b_1 constrânge pe b_2 ” (b_1 se află în ierarhie mai jos decât b_2 , având deci prioritate față de b_2).

Definiția pseudo-cod a funcției *acțiune* este descrisă în Figura 3.5.

```

funcția acțiune ( $p : P$ ):  $A$ 
var posibil :  $\wp(R)$ 
    posibil  $\leftarrow \{(c, a) \mid (c, a) \in R \text{ și } p \in c\}$ 
    pentru fiecare  $(c, a) \in \textit{posibil}$  execută
        dacă  $\neg(\exists(c', a') \in \textit{posibil} \text{ a. } (c', a') \prec (c, a))$  atunci
            acțiune  $\leftarrow a$ 
        sf-dacă
    sf-pentru
        acțiune  $\leftarrow \text{null}$ 
sf-acțiune

```

Figura 3.5. Definiția pseudo-cod a funcției *acțiune*

Una din problemele pe care le ridică arhitectura bazată pe logică este complexitatea computațională, funcția *acțiune* a arhitecturii reactive având în cel mai rău caz o complexitate $O(n^2)$ (cum se poate deduce din algoritmul descris în Figura 3.5), n fiind maximul dintre numărul de comportamente și numărul de percepții.

Cu toate că arhitecturile reactive în general sunt relativ simple, elegante și cu o complexitate computațională mai redusă, există o serie de probleme nerezolvate pentru această categorie de agenți:

- dacă agenții nu utilizează modele ale mediului, atunci trebuie să aibă un număr de informații disponibile în mediul *local*, suficiente pentru a determina o soluție acceptabilă;
- e destul de dificil de văzut cum pot fi proiectate sistemele reactive pure astfel încât să *învețe* din experiență și să-și îmbunătățească performanțele în timp;
- caracteristica de bază a sistemelor pur reactive este interacțiunea dintre componentele comportamentale ale agentului, ori în anumite situații astfel de relații sunt dificile sau poate imposibil de stabilit;
- în cazul în care numărul de nivele în arhitectura agentului crește, dinamica interacțiunilor între nivele devine extrem de complexă și greu de înțeles.

3.2.3 Arhitecturi tip Opinie-Cerere-Intenție

O soluție la problema de conceptualizare a sistemelor care sunt capabile de comportament rațional este cercetarea agenților sub forma unor *sisteme intenționale*, al căror comportament poate fi prevăzut și explicat în termeni de *atitudini* ca opinii, cereri, intenții (modelul BDI propus de Rao-Georgeff). În această abordare, agenții pot fi priviți sub forma unor sisteme raționale, capabile de comportament *orientat spre scop*.

Aceste arhitecturi Opinie-Cerere-Intenție (OCI) – *Belief-Desire-Intention architectures* – își au rădăcinile în tradițiile filozofice despre înțelegerea *raționamentului practic* – procesul de a decide, în fiecare moment, ce acțiune trebuie executată în vederea realizării scopurilor propuse.

Raționamentul practic implică două procese importante: a decide *care* scopuri se doresc a fi realizate și *cum* se vor realiza aceste scopuri. Primul proces este cunoscut sub numele de *deliberare*, iar al doilea de *raționament tip medii-capete (means-ends)*.

Procesul de decizie începe în general cu stabilirea *opțiunilor* disponibile. După generarea setului de alternative, urmează *alegerea* unor alternative din cele existente. Alternativele alese devin *intenții*, care de fapt vor determina acțiunile agentului. Intențiile apoi vor genera un feed-back în procesul de raționament al agentului.

În astfel de procese, un rol important îl joacă măsura în care intențiile agentului sunt sau nu *reconsiderate* din când în când. Dar reconsiderarea are un cost – atât ca timp cât și ca resurse computaționale. Din acest motiv, există următoarea dilemă:

- un agent care nu se oprește în a-și reconsidera des intențiile, riscă să încerce să-și realizeze intențiile chiar și după ce e evident că acestea nu pot fi realizate, sau nu mai există nici un interes în a le realiza;
- un agent care își reconsideră în mod *constant* intențiile, ar putea folosi timp insuficient în vederea realizării efective a acestora, riscând astfel să nu le realizeze niciodată.

Această dilemă este de fapt problema esențială de a alege între comportamentele (arhitecturile) *pro-active* (orientate spre scop) și cele *reactive* (conduse de întâmplări, posibilități).

O soluție la această dilemă pare a fi următoarea: în mediile statice, care nu se schimbă, comportamentele pro-active, orientate spre scop sunt adecvate, pe când în mediile dinamice, abilitatea de a reacționa la schimbări prin modificarea intențiilor devine mai importantă.

Într-un model OCI, starea unui agent OCI poate fi descrisă prin:

- o mulțime de *opinii (părerii)* despre lume;
- o mulțime de *scopuri (dorințe)* pe care agentul va încerca să le realizeze;
- o colecție de *planuri* ce descriu modul în care agentul își va realiza scopurile și cum va reacționa schimbărilor în opiniile pe care le-a avut;
- o structură de *intenții*, descriind modalitatea în care agentul își realizează la un moment dat scopurile și cum reacționează la schimbările de opinii.

Procesul de raționament practic al unui agent OCI este ilustrat în Figura 3.6. După cum este ilustrat în această figură, există șapte componente de bază ale unui agent OCI:

- o mulțime de *opinii* curente, reprezentând informația pe care o are agentul despre mediul său curent;
- o *funcție de revizuire a opiniilor (fro)* - **belief revision function** - , care pe baza unei percepții și a opiniilor curente ale agentului, determină o nouă mulțime de opinii;
- o *funcție de generare a opțiunilor (opțiuni)*, care determină opțiunile disponibile ale agentului (dorințele acestuia), pe baza opiniilor (părerilor) curente despre mediu și a *intențiilor* curente;
- o mulțime de *opțiuni curente*, reprezentând posibilele acțiuni disponibile;

- o *funcție de filtrare (filtrare)*, reprezentând procesul de *deliberare* al agentului - determinarea intențiilor agentului pe baza opiniilor, cererilor și intențiilor curente ale acestuia;
- o mulțime de *intenții* curente – reprezentând scopul curent al agentului – acele stări pe care agentul s-a angajat să le atingă;
- o *funcție de selecție a acțiunii (execută)*, care determină ce acțiune trebuie executată pe baza intențiilor curente.

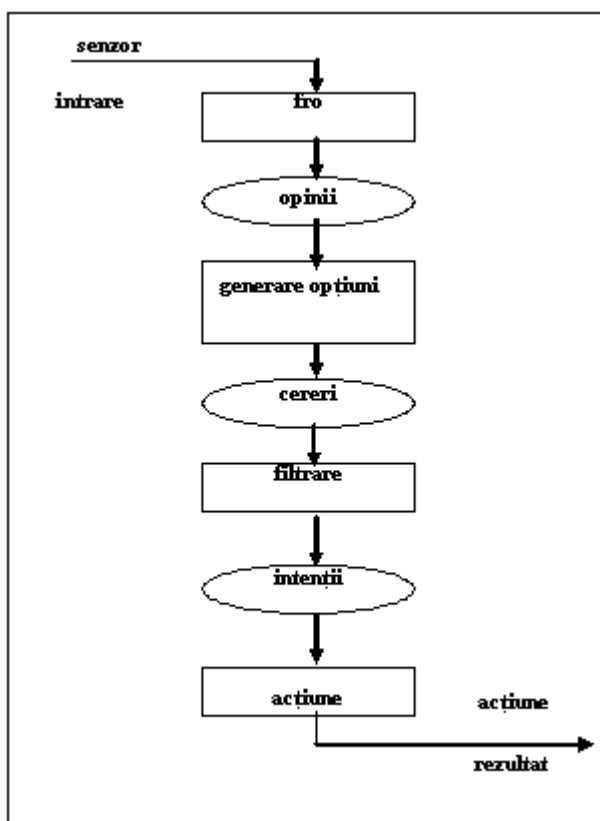


Figura 3.6. Forma generică a unei arhitecturi OCI

În continuare, vom formaliza o arhitectură OCI.

Se va nota prin Opi – mulțimea posibilelor opinii, Cer – mulțimea cererilor posibile și Int – mulțimea intențiilor posibilele ale agentului. Deseori, elementele celor trei mulțimi sunt formule logice. În plus, trebuie definită și noțiunea de *consistență* a celor trei mulțimi, adică răspunsul la o întrebare de genul: în ce măsură intenția de a realiza x este consistentă cu cererea y . În cazul reprezentării sub formă de formule logice, problema consistenței se reduce de fapt la verificarea consistenței formulelor logice.

Starea unui agent OCI la un moment dat va fi reprezentată sub forma unui triplet (O, C, I) , unde $O \subseteq Opi$, $C \subseteq Cer$, $I \subseteq Int$, iar funcția de revizuire a opiniilor va fi $fro: \wp(Opi) \times P \rightarrow \wp(Opi)$, P fiind mulțimea percepțiilor.

Funcția de generare a opțiunilor se definește astfel:
 $optiuni: \wp(Opi) \times \wp(Int) \rightarrow \wp(Cer)$.

Funcția *optiuni* are o serie de roluri:

- este responsabilă de raționamentul de tip *medii-capete* al agentului – procesul de a decide *cum* se vor realiza intențiile;
- odată ce agentul și-a format o intenție x , ulterior trebuie considerate toate opțiunile de a realiza x ; aceste opțiuni vor fi mai concrete decât x ; procesul de generare a opțiunilor unui agent OCI poate fi interpretat ca o elaborare recursivă a unui plan ierarhic, considerând în mod progresiv intenții din ce în ce mai specifice, până se va ajunge în final la acțiuni executabile imediat.

Procesul de deliberare al agentului OCI (a decide *ce* trebuie să facă) va fi reprezentat de funcția de filtrare, $filtrare: \wp(Opi) \times \wp(Cer) \times \wp(Int) \rightarrow \wp(Int)$, funcție care actualizează intențiile agentului pe baza intențiilor avute anterior, a opiniilor și cererilor actuale.

Această funcție trebuie să aibă următoarele roluri:

- trebuie să *omită* orice intenție care nu mai este realizabilă sau pentru a cărei realizare costul este mult mai mare decât cel preconizat;
- trebuie să *rețină* intențiile care nu sunt realizate, dar care sunt considerate totuși a fi avantajoase;
- trebuie să *adopte* noi intenții, fie pentru realizarea intențiilor existente, fie pentru a exploata noi oportunități.

Menționăm faptul că funcția *filtrare* trebuie să satisfacă următoarea constrângere (limitare):

$$\forall O \in \wp(Opi), \forall C \in \wp(Cer), \forall I \in \wp(Int), \quad filtrare(O, C, I) \subseteq I \cup C,$$

adică intențiile curente sunt fie intenții avute anterior, fie opțiuni nou adoptate.

Funcția *execută* se presupune că returnează orice intenție executabilă – care corespunde unei acțiuni direct executabile.

$$executa: \wp(Int) \rightarrow A$$

Ca urmare, funcția *acțiune* a unui agent OCI, $actiune: P \rightarrow A$, va fi definită cu subalgoritmul pseudo-cod prezentat în Figura 3.7.

De remarcat că în reprezentarea intențiilor agentului s-ar putea ține cont de următoarele observații:

- să se asocieze fiecărei intenții o anumită *prioritate*, indicând importanța acesteia;
- reprezentarea intențiilor să se facă sub forma unei *stive*; o intenție să fie adăugată în stivă dacă este adoptată, sau scoasă din stivă dacă a fost realizată sau dacă e nerealizabilă.

```

funcția acțiune ( $p : P$ ):  $A$ 
     $O \leftarrow fro(O, p)$ 
     $C \leftarrow optiuni(C, I)$ 
     $I \leftarrow filtrare(O, C, I)$ 
     $acțiune \leftarrow execută(I)$ 
sf-acțiune

```

Figura 3.7. Definiția pseudo-cod a funcției *acțiune*

În concluzie, arhitecturile OCI sunt arhitecturi de raționament practic, în care procesul de a decide ce trebuie făcut seamănă de fapt cu raționamentul practic pe care îl folosim în viața cotidiană.

Componentele de bază ale unei arhitecturi OCI sunt structurile de date reprezentând opinii, cereri și intenții ale agentului și funcțiile care reprezintă deliberarea și raționamentul.

Modelul OCI este atractiv deoarece:

- este *intuitiv*;
- furnizează o descompunere *funcțională* foarte clară.

În schimb, marea dificultate este maniera în care ar trebui implementate eficient funcțiile necesare.

Pentru cei mai mulți cercetători de Inteligență Artificială, ideea programării sistemelor informatice în termeni de noțiuni *mentale* ca opinii, cereri, intenții reprezintă componenta cheie a “calculului” bazat pe agenți. Conceptul a fost articulat de Yoav-Shoham, în propunerea de *programare orientată pe agenți* (secțiunea 3.3).

3.2.4 Arhitecturi stratificate

Datorită cerințelor ca un agent să fie capabil să aibă atât un comportament reactiv, cât și unul pro-activ, o descompunere evidentă ar necesita crearea de subsisteme separate care să trateze aceste două tipuri diferite de comportament.

Ideea conduce spre o clasă de arhitecturi în care diversele subsisteme sunt aranjate într-o ierarhie de *straturi* (*nivele*) ce interacționează între ele.

În general, într-o arhitectură stratificată ar trebui să existe minim două straturi, corespunzătoare celor două tipuri de comportament. Oricum, de o mare importanță este fluxul de control existent între nivele.

În principiu, există două tipuri de flux de control între nivelele unei arhitecturi stratificate (Figura 3.8):

- *Stratificare orizontală* (Figura 3.8.a) – nivelele soft sunt conectate direct la intrarea perceptuală (senzorul de intrare) și la acțiunea rezultat. De fapt, fiecare nivel acționează ca un agent, furnizând sugestii despre ce acțiune trebuie executată;
- *Stratificare verticală* (Figura 3.8.b și 3.8.c) - senzorul de intrare și acțiunea rezultat sunt fiecare tratate de cel mult un nivel.

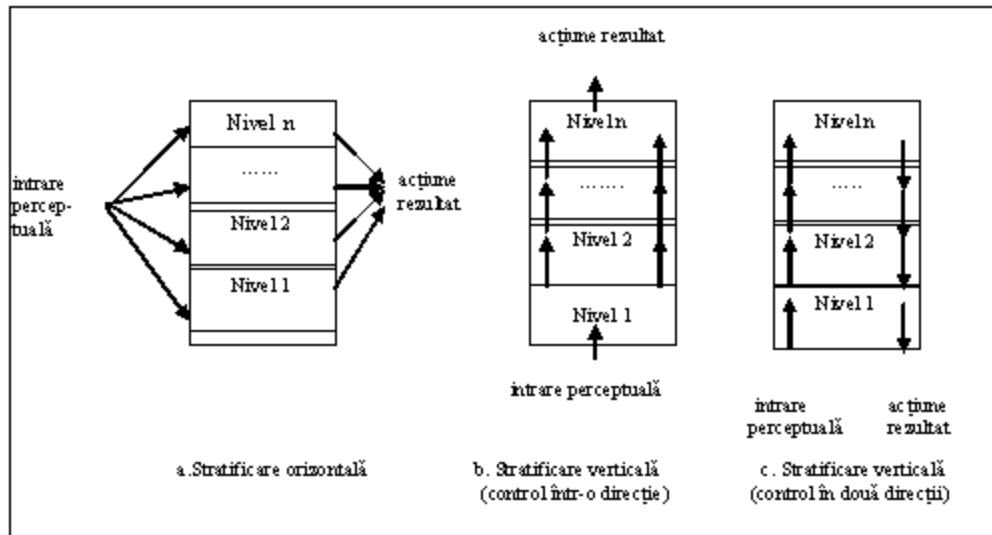


Figura 3.8. Arhitectură stratificată

3.3 Limbaje de programare pentru agenți

Pe măsură ce tehnologia agenților devine tot mai stabilă, o varietate de tehnici și tehnologii soft devin disponibile pentru proiectarea și construirea sistemelor bazate pe agenți.

În secțiunile următoare vom prezenta două dintre cele mai cunoscute limbaje de programare pentru agenți precum și un mediu pentru construirea și dezvoltarea de sisteme multi-agent.

3.3.1 Programare orientată pe agenți

Yoav Shoham propune “o nouă paradigmă de programare, bazată pe un alt punct de vedere asupra procesului de calcul”, pe care o numește *programare orientată pe agenți* - POA - (*agent-oriented programming*).

Idea de bază a POA este programarea directă a agenților în termeni de *noțiuni mentale* (opinie, cerere, intenție), termeni pe care teoreticienii în domeniu le-au dezvoltat pentru reprezentarea proprietăților agenților.

POA poate fi privită ca un fel de programare “post-declarativă”. În programarea declarativă (Prolog, spre exemplu), scopul este de a reduce importanța aspectelor de control: se stabilește un scop (*goal*) pe care sistemul își propune să-l realizeze, după care se lasă mecanismul intern să găsească ce trebuie făcut pentru realizarea scopului. Cu alte cuvinte, spre deosebire de programarea procedurală, ne focalizăm spre *ce* trebuie făcut, nu *cum* trebuie făcut.

Pe de altă parte, limbajele de programare declarativă sunt generate din principii matematice, astfel încât activități ca: analiza, proiectarea, specificarea, implementarea,

devin din ce în ce mai formale. Aceste limbaje oferă o metodă de analiză a corectitudinii programelor.

În POA, ca și în programarea declarativă, ideea este că ne stabilim scopurile și lăsăm în seama mecanismului de control al limbajului să realizeze scopul. Este, deci, destul de clar faptul că, pentru specificarea unor astfel de sisteme va trebui să folosim aserțiuni logice care pot fi formalizate printr-o logică a “cunoașterii”.

Prima implementare a paradigmei POA a fost limbajul de programare AGENT0. Într-un astfel de limbaj, un agent este specificat sub forma unei mulțimi de capabilități (lucruri pe care poate să le facă), o mulțime de *opinii* inițiale (jucând rolul opiniilor dintr-o arhitectură OCI), o mulțime de *angajamente* (similare intențiilor dintr-o arhitectură OCI) și o mulțime de *reguli de angajament*. Regulile de angajament joacă rolul cel mai important, determinând modul în care agentul acționează.

Fiecare astfel de regulă conține o *condiție de mesaj*, o *condiție mentală* și o *acțiune*. Pentru a determina care regulă va fi aplicată la un moment dat, condiția de mesaj este potrivită cu mesajele pe care agentul le primește; condiția mentală este potrivită cu intenția agentului). Dacă regula se potrivește, agentul se *angajează* să execute acțiunea.

Acțiunile pot fi *private*, corespunzătoare unor proceduri interne sau *comunicative*, sub forma unor mesaje ce vor fi transmise. Mesajele pot fi de trei tipuri: “*tip cerere*”, “*tip refuz*”, pentru a executa sau a respinge acțiuni și mesaje “*informative*”, care transmit informații.

Un exemplu de *regulă de angajament* în AGENT0 ar putea fi următorul:

COMMIT

```
(  
  (agent, REQUEST, DO(time, action)  
  ),  
  ;; condiția de mesaj  
(B,  
  [now, Friend agent] AND  
  CAN(self, action) AND  
  NOT[time, CMT(self, anyaction)]  
),  
  ;; condiția mentală  
self,  
  DO(time, action)  
)
```

Această regulă ar putea fi parafrazată în felul următor:

<<**Dacă** recepționez un mesaj de la agent prin care îmi cere să execut o acțiune la un moment dat și eu cred că:

- agentul îmi este prieten;
- pot să execut acțiunea;
- la momentul respectiv, nu sunt angajat în executarea altei acțiuni.

atunci mă angajez să execut acțiunea la momentul dat>>.

Modul de operare al unui agent poate fi descris prin următoarea buclă (Figura 3.9):

1. Citește toate mesajele curente, actualizându-ți opiniile – și în consecință angajamentele – când e necesar;

2. Execută toate angajamentele pentru ciclul curent, acolo unde condiția de capabilitate a acțiunii asociate este satisfăcută;
3. Salt la (1).

De remarcat că acest limbaj este de fapt un *prototip*, nefiind conceput pentru realizarea de sisteme de producție pe scară largă.

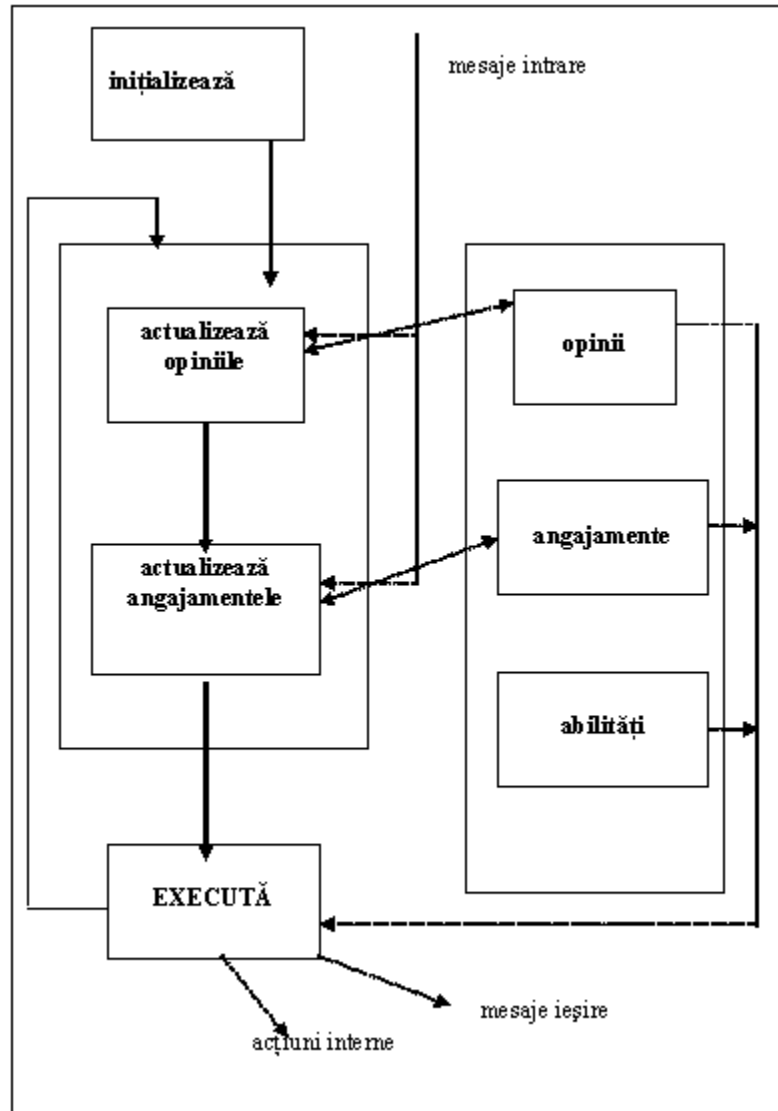


Figura 3.9. Fluxul de control în AGENT0

3.3.2 “METATEM” Concurrent

Limbajul METATEM Concurrent a fost dezvoltat de Fisher și se bazează direct pe execuția formulelor logice. Un sistem în acest limbaj conține un număr de agenți ce se execută concurrent, fiecare agent fiind capabil să comunice cu ceilalți printr-o transmisie asincronă de mesaje.

Fiecare agent este programat dându-i-se o specificație *logico-temporală* a comportamentului care se dorește ca agentul să-l aibă.

Execuția programului agent corespunde construirii iterative a unui model logic pentru specificația temporală a agentului. E posibil să se demonstreze că procedura utilizată pentru execuția specificației unui agent este corectă arătând că: dacă e posibil să se satisfacă specificațiile, atunci agentul va realiza acest lucru.

Semantica logică a limbajului METATEM Concurrent este strâns legată de semantica logicii temporale, ceea ce înseamnă, printre altele, că specificarea și verificarea sistemelor METATEM Concurrent sunt propoziții logice.

Un agent program în METATEM Concurrent are forma $\bigwedge_i P_i \Rightarrow F_i$, P_i fiind o formulă logică temporală cu referire la prezent sau trecut, iar F_i fiind o formulă logică temporală cu referire la prezent sau viitor. Formulele $P_i \Rightarrow F_i$ se numesc *reguli*. Ideea de bază a execuției unui astfel de program se exprimă astfel:

în baza trecutului execută viitorul

În consecință, pentru fiecare regulă în parte se încearcă potriviri cu o serie de structuri *interne (istoric)*, iar dacă o potrivire este găsită, atunci regula se poate accepta. Dacă o regulă este acceptată, atunci anumite variabile din partea “*viitor*” sunt instanțiate, iar viitorul devine un *angajament* pe care, în consecință, agentul va încerca să-l îndeplinească.

Îndeplinirea unui angajament înseamnă validarea unor predicate de către agent.

O definiție simplă a unui agent în limbajul METATEM Concurrent ar putea fi următoarea:

$cr(ask)[give]$:

$$\Theta ask(x) \Rightarrow \Diamond give(x)$$

$$(\neg ask(x) \Xi (give(x) \wedge \neg ask(x))) \Rightarrow \neg give(x)$$

$$give(x) \wedge give(y) \Rightarrow (x = y)$$

Agentul din exemplu este un controlor pentru o resursă care se înnoiește la infinit, dar care poate fi obținută de un singur agent la un moment dat. În exemplul anterior, operatorii Θ , \Diamond și Ξ sunt operatori ai logicii modale, a căror semnificație rezultă din interpretarea regulilor 1, 2 și 3. Agentul va trebui deci să forțeze exclusiunea mutuală asupra resursei.

- prima linie a programului definește interfața agentului: numele acestuia este *cr* și acceptă două mesaje *ask* și *give*;
- următoarele trei linii reprezintă corpul programului
 - predicatul $ask(x)$ înseamnă că agentul x cere resursa;
 - predicatul $give(x)$ înseamnă că agentului x i s-a dat resursa.

Controlorul de resurse se presupune că este singurul agent capabile să “dea” resursa. Pe de altă parte, mai mulți agenți pot lansa simultan cereri pentru resursă. Cele trei reguli care definesc comportamentul agentului pot fi descrise astfel:

Regula 1. Dacă cineva a cerut resursa, atunci eventual i se dă resursa;

Regula 2. Nu se dă resursa până când cineva a cerut de când s-a dat ultima dată;

Regula 3. Nu se dă la mai mulți agenți la un moment dat (dacă doi indivizi cer în același timp, atunci ei reprezintă același agent).

METATEM Concurrent este o bună ilustrare a modului în care funcționează o abordare bazată pe logică în programarea agenților, cu o logică destul de expresivă.

3.3.3 Limbajul “Jack Intelligent Agents”

Limbajul Jack este un limbaj de programare care extinde Java cu concepte orientate pe agenți, cum ar fi:

- agenți;
- capabilități;
- evenimente;
- planuri;
- bază de cunoștințe;
- gestiunea concurenței și a resurselor.

“**Jack Intelligent Agents**” este mai mult decât un limbaj de programare pentru agenți, este un mediu pentru construirea, execuția și integrarea sistemelor multiagent comerciale folosind o abordare bazată pe componente.

Agenții Inteligenți de tip “Jack Agent” sunt componente software autonome și flexibile care au *scopuri* precise de atins, *evenimente* prin care interacționează și *cerințe* de respectat. Pentru a descrie cum ar trebui să-și atingă aceste cerințe, agenții Jack Agent sunt programați printr-un set de *planuri*. Fiecare plan funcțional descrie cum fiecare agent, în parte, va încerca să-și atingă scopul în funcție de circumstanțe și de obiectivul final.

Agentii Jack au la baza modelul arhitectural BDI (secțiunea 3.2.3): odată ce și-a început activitatea, agentul își urmărește scopurile stabilite (**desires**), adoptând planul optim (**intentions**), în deplină concordanță cu setul curent de opinii (**beliefs**).

Jack Agent este o componentă software care prezintă un comportament logic al celor doi stimuli: cel activ (*goal directed*) și reactiv (*event driven*). Fiecare agent are:

- un *set de opinii* asupra mediului;
- un *set de evenimente* la care va răspunde;
- un *set de scopuri (cerințe)* pe care și le propune să le atingă;
- un *set de planuri* care descriu cum trebuie tratate evenimentele care ar putea apărea în desfășurător.

La *instanțierea* unui agent în sistem, acesta va aștepta până va primi un scop pe care să-l urmeze sau va testa un eveniment care se presupune că ar trebui să-i răspundă. Apariția unui astfel de scop sau eveniment îi va imprima agentului un curs al acțiunii pe care și-l va însuși. Dacă agentul crede că acel scop sau eveniment a fost tratat el nu va face nimic. Dealtfel, va cauta în planul său de acțiune pentru a afla tot ceea ce este în conformitate cu cerințele aplicabile imediat situațiilor ivite. Dacă va întâmpina alte probleme în execuția acestui plan, se va căuta un alt agent care ar putea satisface cerințele și care va continua să caute o alternativă finală sau care va epuiza toate variantele.

În sistemele orientate pe agenți, agentul este programat să execute un set de planuri la fel ca o persoană rațională. În particular, agentul este capabil să-și expună următoarele proprietăți asociate cu o funcționalitate rațională:

- *scopul* - agentul se focalizează pe obiective și nu pe metode alese a fi urmate;

- **relația cu mediul** - agentul va înmagazina opțiunile aplicabile fiecărei situații de moment, și va lua decizii despre ceea ce va încerca să finalizeze, bazându-se pe condițiile însușite în timp pentru un moment specificat;
- **validarea în timp real** – agentul se va asigura că o opțiune aleasă va fi urmată atâta timp cât anumite condiții vor fi valide;
- **concurența** – apare într-un sistem agent multi-thread. Dacă apar scopuri sau evenimente noi, agentul va fi capabil să aleagă care proces este prioritar.

După cum spuneam, limbajul Jack Agent **extinde Java** oferind suport pentru Programarea Orientată-Agent:

- definește clase de bază, interfețe și metode proprii;
- asigură extensii ale sintaxei Java pentru a dezvolta noi clase, definiții și instrucțiuni specifice agenților;
- asigură extensii semantice pentru execuțiile cerute de către un sistem orientat pe agenți.

Limbajul Jack Agent introduce cinci **clase specifice**:

- **Agent (agent)** - constructorul clasei agent este folosit pentru a defini comportamentul unui agent inteligent. La definiția acestei clase se includ și capacitățile pe care agentul le are în starea inițială și pe care le dobândește pe parcurs; mesaje eveniment la care răspunde și ce plan urmează să folosească pentru a-și îndeplini scopurile.
- **Capability (abilități)** - constructorul abilităților permite componentelor unui agent să acumuleze și să refolosească funcționalitățile deprinse în timp. O abilitate poate fi componenta unui plan, eveniment, convingeri și a altor abilități pe care le include sau de care sunt incluse și care împreună caracterizează agentul. În schimb, un Agent poate fi alcătuit dintr-un set de abilități, fiecareia atribuindu-i-se o funcție specifică.
- **BeliefSet (convingeri)** – constructorul convingerilor reprezintă baza de date de care se servește agentul într-un model relațional. Este proiectat astfel încât să asigure interogări cu date logice.
- **View (perspectiva)** – constructorul perspectivelor permite interogărilor generale să fie făcute pe baza unui model. Modelul poate fi implementat folosind multiple BeliefSet-uri sau structuri Java.
- **Event (Evenimente)** - constructorul evenimentelor descrie apariția unui răspuns cu care un agent interacționează.
- **Plan (Plan/Schema)** – planul unui agent este analog cu schema de funcționare. Conține instrucțiunile pe care un agent le urmează încercând să-și atingă scopurile și să servească evenimentele proiectate.

Limbajul Jack Agent are la bază următoarele **tipuri de construcție**:

- Clasele.
- Declarațiile (# declarations).
- Declarații de metode de raționament (Reasoning Method Statements).