

Sisteme bazate pe reguli, o implementare
modernă.
Algoritmul Rete

Victor Hurdugaci
Facultatea de Matematică și Informatică
Universitatea Transilvania Brașov

Iulie 2009
Brașov



“Sisteme bazate pe reguli, o implementare modernă. Algoritmul Rete” by Victor Hurdugaci is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported License.

Părinților mei

Cuprins

1	Introducere	7
1.1	Sisteme bazate pe reguli	7
1.1.1	DENDRAL	8
1.1.2	MYCIN	8
1.1.3	XCON	9
1.1.4	Radar	9
1.2	Sisteme de producții	10
1.3	Lumea cuburilor	12
2	Algoritmul Rete	19
2.1	O privire de ansamblu	19
2.2	Implementarea rețelei alfa	23
2.2.1	Rețea de date (dataflow network)	24
2.2.2	Rețea de date cu indexare	27
2.2.3	Rețea de date cu indexare exhaustivă	27
2.3	Implementarea nodurilor de memorie	28
2.3.1	Implementarea memoriilor alfa	31
2.3.2	Implementarea memoriilor beta	32
2.3.3	Implementarea nodurilor de producții	33
2.4	Implementarea nodurilor de joncțiune	34
2.5	Eliminarea elementelor ML	36
2.6	Condiții negate	40
2.7	Conjunții negate	47
2.8	Optimizări ale algoritmului Rete	52
3	O Implementare Modernă	55
3.1	De ce cloud computing?	55
3.2	Windows Azure	58
3.3	CloudRuleBasedSystem (CRBS)	60

3.3.1	De ce serviciu (scalabil)?	60
3.3.2	Arhitectura	61
3.3.3	Securitate	63
3.3.4	Ce aduce nou?	65
4	Pseudocodul final	67
	Bibliografie	77

Capitolul 1

Introducere

1.1 Sisteme bazate pe reguli

Deși rețelele neuronale și algoritmi genetici oferă multe tehnici utile pentru rezolvarea eficientă și eficientă a problemelor, sistemele bazate pe reguli și dezvoltarea și cercetarea în acest domeniu au făcut posibilă abordarea unor probleme practice și realiste. Sistemele bazate pe reguli au fost primele aplicații comerciale ale cercetării realizate în domeniul Inteligenței Artificiale.

Primul sistem de succes bazat pe reguli a fost DENDRAL (vezi secțiunea 1.1.1), realizat de Feigenbaum la începutul anilor '60. Acesta a demonstrat o tehnică de rezolvare a problemelor care nu era caracteristică Inteligenței Artificiale. Programul simula capacitățile de analiză și decizie ale unui chimist expert. O serie întreagă de sisteme expert pentru domenii variate (geologie, diagnostică medicală, explorare, etc.) au fost create folosind conceptele descrise de Feigenbaum în DENDRAL.

Comunitățile de Inteligență Artificială au acceptat sistemele bazate pe reguli ca programe inteligente deoarece foloseau cunoștințe specifice unui domeniu pentru a rezolva un set (restrâns) de probleme. Dezvoltarea unor sisteme expert practice s-a accelerat o dată cu introducerea conceptelor de script-uri și cadre (eng. Frames). În anul 1972 Roger Schank¹ a introdus conceptul de "script" care reprezenta un set de evenimente similare ce erau așteptate în urma unei configurații des întâlnite. În 1975 Minsky² a introdus

¹Roger Schank (n. 1946) fost profesor la Stanford, Yale și Northwestern, este unul din cei care au influențat major inteligența artificială și psihologia cognitivă între anii '70 și '80.

²Marvin Lee Minsky (n. 1927) este un cercetător american în domeniul inteligenței

conceptul de cadru care ajută la reprezentarea structurată a obiectelor și scenariilor. Aceste două concepte împreună cu o combinație de euristici au permis îmbunătățirea considerabilă a capacităților de reprezentare a cunoștințelor și realizare a proceselor de inferență.

1.1.1 DENDRAL

Dendral a fost un pionier cu mare influență al sistemelor expert, realizat la începutul anilor '60. Scopul principal a fost ajutarea chimiștilor organici în identificarea moleculelor organice necunoscute prin analiza masei spectrale³ și utilizarea unor cunoștințe preexistente de chimie. Sistemul a fost realizat la Universitatea Stanford de Edward Feigenbaum, Bruce Buchanan, Joshua Lederberg și Carl Djerassi.

Produsul software Dendral este considerat primul sistem expert datorită procesului său de luare a decizilor și a comportamentului axat pe rezolvarea problemelor de chimie organică. Este format din două sub-aplicații: "Heuristic Dendral" și "Meta-Dendral". Întreg sistemul a fost scris în Lisp.

Mai multe sisteme au derivat din Dendral dintre care amintim MYCIN, MOLGEN, MACSYMA, PROSPECTOR, XCON și STREAMER.

1.1.2 MYCIN

MYCIN ocupă un loc tot între pionierii sistemelor expert. A fost dezvoltat în cinci sau șase ani la începutul anilor '70 la Universitatea Stanford. A fost scris în Lisp ca parte a lucrării de doctorat a lui Edward Shortliffe⁴ sub îndrumarea lui Bruce Buchanan și Stanley N. Cohen. Sistemul a fost realizat cu scopul de a identifica bacteriile ce cauzau infecții severe (precum meningita) și de a recomanda antibiotice cu doze corespunzătoare greutății pacientului.

MYCIN folosea un motor de inferență simplu și o bază de cunoștințe de aproximativ 600 de reguli. Pentru a primi o soluție, cel care rula aplicația trebuia să răspundă la o serie lungă de întrebări cu răspuns binar (da sau nu). La final era oferită o listă de posibile bacterii vinovate de starea pacientului, fiecare bacterie având un factor de certitudine și o listă de medicații.

artificiale, co-fondator al laboratorului de IA din cadrul MIT și autor al unor lucrări despre IA și filosofie.

³Spectrometria masei: http://en.wikipedia.org/wiki/Mass_spectrometry

⁴Edward Shortliffe (n. 1947) este un informatician biomedical, fizician și cercetător, pionier în cercetarea sistemelor expert și a informaticii medicale.

În ciuda succesului, MYCIN a pornit o serie de dezbateri legate de implementarea modului de incertitudine. Dezvoltatorii au susținut că sistemul lor este minim influențat de perturbările din măsurătorile asociate incertitudinii regulilor individuale afirmând că puterea sistemul este în cunoștințe și în schema de inferență, nu în valorile numerice ale incertitudinii. Mulți au susținut că era posibilă utilizarea statisticii Bayesiane clasice.

1.1.3 XCON

Numit inițial R1, programul XCON (acronim de la eXpert CONfigurer) a fost un sistem bazat pe reguli scris în OPS5 de către John P. McDermott la Carnegie Mellon University în 1978. Scopul aplicației era asistarea la cumpărarea computerelor VAX selectând componente pe baza cerințelor clientului. Dezvoltarea lui XCON a fost marcată inițial de două încercări eșuate de a realiza sistemul în FORTRAN și BASIC.

Înainte de apariția lui XCON, când era comandat un nou calculator VAX de la DEC⁵ fiecare componentă era livrată separat. Deoarece vânzătorii nu erau întotdeauna niște persoane tehnice se întâmpla ca unii clienți să primească componente incompatibile. Aceste evenimente duceau la scăderea renumelui firmei, satisfacție redusă pentru clienți și profit mai mic.

XCON a fost folosit prima dată în 1980 de către DEC în Salem, New Hamshire. În final a ajuns la aproximativ 2500 de reguli, iar în 1986 avea peste 80000 de comenzi procesate cu o acuratețe de 95-98%. Se estimează că DEC a câștigat peste 25 milioane de dolari americani prin folosirea sistemului expert deoarece a fost redus numărul greșelilor umane. Succesul XCON a determinat DEC să rescrie XCON ca XSEL.

Lucrarea lui McDermott din 1980 despre R1 i-a adus acestuia premiul Academiei Americane de Inteligență Artificială în 1999. Numele R1 provine de la McDermott care zis “Three years ago I wanted to be a knowledge engineer, and today I *are one*. (n.a. se citește R1)”.

1.1.4 Radar

Radar este un model cognitiv dezvoltat ca parte a proiectului de cercetare în domeniul procedurilor de antrenare a agenților bazați pe luarea deciziilor. Sistemul examinează imagini radar și învață să le clasifice în

⁵Digital Equipment Corporation - companie fondată de Ken Olsen în 1957. A fost cumpărată în 1998 de Compaq care ulterior, în 2002, a fuzionat cu HP

aliat, neutre sau inamice. Imaginile sunt specificate cu ajutorul a două caracteristici precum viteză, altitudine, mărime, etc.; fiecare caracteristică are două valori posibile. Sistemul învață să clasifice corect imaginile prin construirea unei memorii episodice cu imaginile precedente și clasificarea lor corectă. Memoria este indexată folosind o reprezentare mai abstractă decât cele două caracteristici, astfel încât un set individual de test se poate extrage. Atunci când sistemului îi este prezentată o nouă imagine, el va căuta cele două caracteristici și le va transforma în reprezentarea abstractă urmând ca apoi să folosească imaginile anterioare similare din memorie pentru clasificare. "Similar" înseamnă "care au aceiași reprezentare abstractă". Dacă nici o imagine nu este prezentă în memoria episodică atunci sistemul va clasifica aleator imaginea. Pe baza feedback-ului primit de la utilizator sistemul își actualizează periodic memoria episodică.

Radar folosește șapte spații de probleme și a pornit inițial cu 341 de reguli, unele scrise de mână, altele compilate din cod sursă scris în TAQL. A învățat peste 105207 de reguli în decursul a 7500 de exemple de test generate aleator. Exemplele au fost selectate, prin înlocuire, cu o distribuție uniformă peste 3^9 specificații de imagini.

1.2 Sisteme de producții

Un sistem de producții este un limbaj de programare cu o caracteristică nefericită: programele mari (cu număr mare de instrucțiuni) se execută mult mai încet decât cele mici. Instrucțiunile extra din programul mare nu trebuie să fie executate pentru a încetini sistemul ci este suficientă prezența lor. Se poate ca cineva să creeze un sistem de producții cu un număr mare de reguli, nefolosite, care să dea o falsă impresie de ineficiență.

Totuși aceste sisteme nu sunt marcate de insucces. Există o caracteristică care le face atractive pentru construcția sistemelor de producții de dimensiune mare, caracteristică care nu este întâlnită în nici un alt tip de sistem: programatorul nu trebuie să specifice în detaliu modul cum diverse părți ale programului vor interacționa.

Sistemele de producții sunt compuse dintr-o unitate de procesare (UP) plus două memorii disjuncte numite memorie de producții (MP) respectiv memorie de lucru (ML). ML stochează programul executat de UP pe când MP reține datele cu care operează programul. Elementele ML se numesc date și, de obicei, sunt reprezentate sub forma unei structuri de simboluri. Cu alte cuvinte aceste elemente sunt descrise de un limbaj formal precum

limbajul listelor sau un limbaj format din tuple.

Elementele unei astfel de memorii care descrie o cameră sunt exemplificate în rândurile următoare. Fiecare element al memorie este un tuplu de forma (Obiect, Atribut, Valoare):

```
(Cameră Suprafață 12)
(Cameră NumărDePereți 4)
(Cameră CuloarePereți Alb)
(Cameră NumărDeFerestre 1)
(Cameră Mobilă Dulap)
(Dulap Înălțime 2)
(Dulap NumărUși 2)
```

Elementele MP reprezintă o listă de condiții și acțiuni. În general ele pot fi traduse prin propoziții condiționale “dacă ... atunci ...”. Fiecare producție este compusă din două părți: ipoteză și concluzie. Pentru a realiza concluzia, ipoteza trebuie să fie satisfăcută. Atât ipotezele cât și concluziile pot fi formate din mai multe condiții, respectiv acțiuni. Pentru a putea realiza sisteme complexe, asupra impotezelor se pot aplica operatori logici unari (NOT) iar condițiile pot fi combinate prin operatori logici binari (Și, SAU, XOR). În continuare este prezentat un exemplu de MP pentru o ML care conține descrierea unei camere. Fiecare element al ipotezei este de forma (Obiect Atribut Operator Valoare) (câmpurile dintre croșete se numesc variabile și se substituie la interpretare cu instanțe ale obiectelor din ML) iar elementele concluziei au forma (Obiect Atribut = Valoare):

1. DACĂ (<cameră> Suprafață > 20) ȘI
 (<cameră> Aspect = Îngrijit) SAU
 (<cameră> CantitateLuminăNaturală = Multă)
 ATUNCI
 (<cameră> Impresie = Plăcută)
 (<cumpărător> Satisfacție = Mare)

2. DACĂ (<cameră> Suprafață > 20) ȘI
 (<cameră> Aspect = Îngrijit) ȘI
 NOT (<cameră> CantitateLuminăNaturală = Multă)
 ATUNCI
 (<cameră> Impresie = Satisfăcătoare)
 (<cumpărător> Satisfacție = Medie)

Condițiile din MP sunt condiții pentru elementele MP iar concluziile ML sunt principalul mijloc de a modifica elementele ML. Cele mai frecvente operații asupra ML sunt adăugarea și ștergerea de elemente.

Spre deosebire de programele convenționale, sistemele de producții nu au un flux de execuție. Totuși există un astfel de flux, dar el nu poate fi precizat deoarece depinde de ordinea, tipul și valorile datelor de intrare. Interpretorul datelor de intrare realizează următorii pași pentru a determina o soluție a problemei:

1. Selectează un element al ML
2. Determină care elemente ale MP sunt adevărate pentru elementul ales
3. Dacă nici o condiție nu e adevărată mergi la 5; altfel alege una din producțiile care se potrivesc
4. Execută acțiunile corespunzătoare producției
5. Dacă mai există elemente mergi la 1; altfel IEȘI

Secvența 2-4 se numește “ciclul recunoaștere-acțiune” (en. recognize-act cycle). Pasul 2 se numește “potrivire”, pasul 3 “rezolvarea conflictelor” iar pasul 4 “acțiune” [3].

Din secvența de pași de mai sus cititorul atent poate deduce deja o mare problemă a sistemelor de producții: pentru fiecare element la ML trebuie încercată potrivirea cu fiecare element al MP. Dacă s-ar aplica o metodă de genul forței brute atunci performanțele sistemelor de dimensiune mare ar fi sub pragul minim de timp acceptat pentru execuție. În continuare va fi prezentată o abordare inedită a algoritmului Rete folosind tehnologii disponibile în prezent - Internetul și sisteme distribuite.

1.3 Lumea cuburilor

Definiție 1. *Model*

Numim model toate atributele care descriu un obiect. Un model poate fi format la rândul său din alte modele.

Definiție 2. *Obiect*

Se numește obiect o instanță a unui model. Obiectele reprezintă elementele ML.

Definiție 3. Fapt

Numim **fapt** o informație despre o un anumit model (ex. (Casă Înălțime = 5)).

Definiție 4. Regulă

Numim **regulă** un element al MP de forma (DACĂ ... ATUNCI ...).

Definiție 5. Lume

Se numește **lume** o colecție de modele, fapte și reguli.

În continuare vom folosi, acolo unde este posibil, o lume a cuburilor⁶, simplă dar destul de complexă pentru a exemplifica algoritmi și procedee. Această lume este specifică unei probleme în care, pe baza unor informații inițiale despre niște cuburi se dorește aflarea a cât mai multe despre așezarea lor într-un plan 2D.

Deoarece este vorba de cuburi, primul și de altfel singurul model din lume este “Cub”. Modelul Cub este format următoarele atribute:

- Latură: atribut de tip numeric cu valori reale pozitive. Reprezintă lungimea laturii cubului.
- Înălțime: atribut de tip numeric cu valori reale pozitive. Reprezintă înălțimea la care se află cubul față de nivelul solului.
- VecinStânga: atribut de tip Cub care poate fi nedefinit. Reprezintă cubul care se află în stânga.
- VecinDreapta: atribut de tip Cub care poate fi nedefinit. Reprezintă cubul care se află în dreapta.
- VecinJos: atribut de tip Cub care poate fi nedefinit. Reprezintă cubul care se află dedesubt.
- VecinSus: atribut de tip Cub care poate fi nedefinit. Reprezintă cubul care se află deasupra.

O reprezentare schematică a modelului ar putea fi:

```
Cub
- Latură [numeric]
- Înălțime [numeric]
```

⁶Inspirată de Doorenbos[2]

- VecinStânga [Cub]
- VecinDreapta [Cub]
- VecinJos [Cub]
- VecinSus [Cub]

Regulile care vor compune MP sunt în număr de opt și sunt suficiente pentru a determina majoritatea informațiilor despre un set de cuburi.

REGULA 1:

DACĂ (<cub1> VecinStânga = <cub2>) ȘI
 (<cub1> VecinSus = <cub3>) ȘI
 (<cub3> VecinStânga = <cub4>)

ATUNCI

(<cub2> VecinSus = <cub4>)

Deși se poate observa că din ipoteză se pot deduce mai multe concluzii am decis ca, pentru simplitate, concluziile să fie formate dintr-o singură acțiune. Regulile au fost de fapt simplificate și au fost eliminate, a priori, acțiunile redundante deoarece simplificarea regulilor nu face obiectul acestei lucrări (o întregă altă lucrare ar putea fi scrisă despre acest subiect fără riscul de a-l epuiza).

Regula 1 se traduce în felul următor: “Dacă există un cub care are în partea stângă un alt cub iar pe cubul inițial se află un alt cub care are și el un vecin stâng atunci cei doi vecini din stânga fiecărui cub se află unul deasupra celuilalt”.

În continuare pentru simplitate vom utiliza doar operatorul logic ȘI care va fi omis din scriere; dacă va fi nevoie de alți operatori aceștia vor fi specificați explicit.

REGULA 2:

DACĂ (<cub1> VecinJos = NULL)
 (<cub1> VecinSus = <cub2>)
 (<cub2> VecinSus = <cub3>)

ATUNCI

(<cub3> Înălțime = Cub1.Latură+Cub2.Latură)

Se poate observa că rezultatul unei acțiuni poate fi o combinație a elementelor din ipoteză. În acest caz, înălțimea la care se află cubul de mai de sus dintr-o stivă de trei cuburi este suma laturilor cuburilor pe care stă.

REGULA 3:

```
DACĂ (<cub1> VecinJos = NULL)
    (<cub1> VecinSus = <cub2>)
ATUNCI
    (<cub2> Înălțime = <cub1>.Latură)
```

REGULA 4:

```
DACĂ (<cub1> VecinJos = NULL)
ATUNCI
    (<cub1> Înălțime = 0)
```

REGULA 5:

```
DACĂ (<cub1> VecinStânga = <cub2>)
ATUNCI
    (<cub2> VecinDreapta = <cub1>)
```

REGULA 6:

```
DACĂ (<cub1> VecinDreapta = <cub2>)
ATUNCI
    (<cub2> VecinStânga = <cub1>)
```

REGULA 7:

```
DACĂ (<cub1> VecinSus = <cub2>)
ATUNCI
    (<cub2> VecinJos = <cub1>)
```

REGULA 8:

```
DACĂ (<cub1> VecinJos = <cub2>)
ATUNCI
    (<cub2> VecinSus = <cub1>)
```

Ultimele patru reguli sunt extrem de logice pentru cititorul uman, dar calculatorul nu poate ști că dacă un cub se află în stânga altuia atunci cel din urmă este în dreapta primului (regula 5). Traducerea regulilor 3 și 4 va fi lăsată ca un exercițiu pentru cititor.

Au mai rămas nedefinite faptele care fac lumea să existe. Aceste fapte, într-o situație reală sunt datele de intrare și ieșire ale programului. Faptele reprezintă cinci cuburi așezate într-o anumită configurație:

CUB1

- Latură: 5
- Înălțime: NULL
- VecinStânga: NULL
- VecinDreapta: CUB3
- VecinJos: NULL
- VecinSus: NULL

CUB2:

- Latură: 5
- Înălțime: NULL
- VecinStânga: NULL
- VecinDreapta: NULL
- VecinJos: NULL
- VecinSus: NULL

CUB3:

- Latură: 5
- Înălțime: NULL
- VecinStânga: CUB1
- VecinDreapta: NULL
- VecinJos: NULL
- VecinSus: CUB4

CUB4:

- Latură: 5
- Înălțime: NULL
- VecinStânga: CUB2
- VecinDreapta: NULL
- VecinJos: NULL
- VecinSus: NULL

CUB5:

- Latură: 5
- Înălțime: NULL
- VecinStânga: NULL
- VecinDreapta: NULL
- VecinJos: CUB4
- VecinSus: NULL

Faptele vor fi introduse în sistem în următoarea formă (dacă pentru

un atribut nu se specifică nici un fapt atunci acel atribut este nedefinit - NULL):

F1: (CUB1 Latură = 5)
F2: (CUB1 VecinDreapta = CUB3)
F3: (CUB2 Latură = 5)
F4: (CUB3 Latură = 5)
F5: (CUB3 VecinStânga = CUB1)
F6: (CUB3 VecinSus = CUB4)
F7: (CUB4 Latură = 5)
F8: (CUB4 VecinStânga = CUB2)
F9: (CUB5 Latură = 5)
F10: (CUB5 VecinJos = CUB4)

Citiorul uman poate deduce foarte ușor așezarea cuburilor și vom vedea în continuare că cele 8 reguli sunt suficiente pentru a afla toate informațiile despre cuburi pe baza faptelor de mai sus. Pentru exemplificare, vom executa doar una dintre reguli.

Faptul (CUB4 VecinStânga CUB2) verifică ipoteza regulii 5 ceea ce duce la executarea acțiunii (CUB2 VecinStânga = CUB4). Procedând analog și executând diverse reguli se pot afla toate informațiile despre cuburi.

Încercarea de potrivire pentru fiecare fapt în fiecare condiție din fiecare regulă duce la o complexitate de timp dezastruoasă și o dată cu creșterea numărului de reguli se va realiza că este nevoie de o abordare diferită. O astfel de abordare este algoritmul lui Rete care reduce considerabil timpul necesar procesării faptelor și duce mult mai repede la aflarea soluției.

Capitolul 2

Algoritmul Rete

Deoarece algoritmul de inferență Rete reprezintă baza sistemului expert descris de această teză acest capitol descrie Rete. Din păcate cele mai multe explicații ale algoritmului în literatură nu sunt foarte clare și probabil acesta este motivul pentru care Rete a căpătat “o reputație de dificultate extremă” [7].

Înainte de a începe discuția efectivă despre algoritm trebuie să definim terminologia și notațiile. Rete (se pronunță “Rit” sau “Ri-ti” și provine din cuvântul de origine latină care înseamnă “rețea”) lucrează cu o Memorie de Producție (MP) și o Memorie de Lucru (ML). Fiecare dintre cele două menționate anterior se schimbă în timp. Pentru exemplele care urmează vom folosi Lumea Cuburilor prezentată în secțiunea 1.3.

2.1 O privire de ansamblu

Algoritmul Rete îmbunătățește doar o parte a algoritmului de inferență și anume partea de potrivire a condițiilor. Execuția efectivă a unei reguli rămâne în sarcina dezvoltatorului.

După cum se poate observa în figura 2.1, Rete poate fi văzut ca o cutie neagră. Ca date de intrare primește informații despre MP și ML iar la fiecare schimbare a acestora se modifică rezultatele.

În figura 2.2 este prezentată o schemă generică a rețelei create de Rete. Rețeaua are două părți. Partea “alfa” realizează teste cu constantele din elementele memoriei de lucru precum “VecinStânga” sau “20”. Rezultatul testelor este stocat în memoriile alfa (MA), fiecare MA reținând elementele ML care au trecut testele constante pentru o condiție individuală (ex: pentru regula DACĂ (Cub1 VecinSus = Cub2) ATUNCI ... și faptul

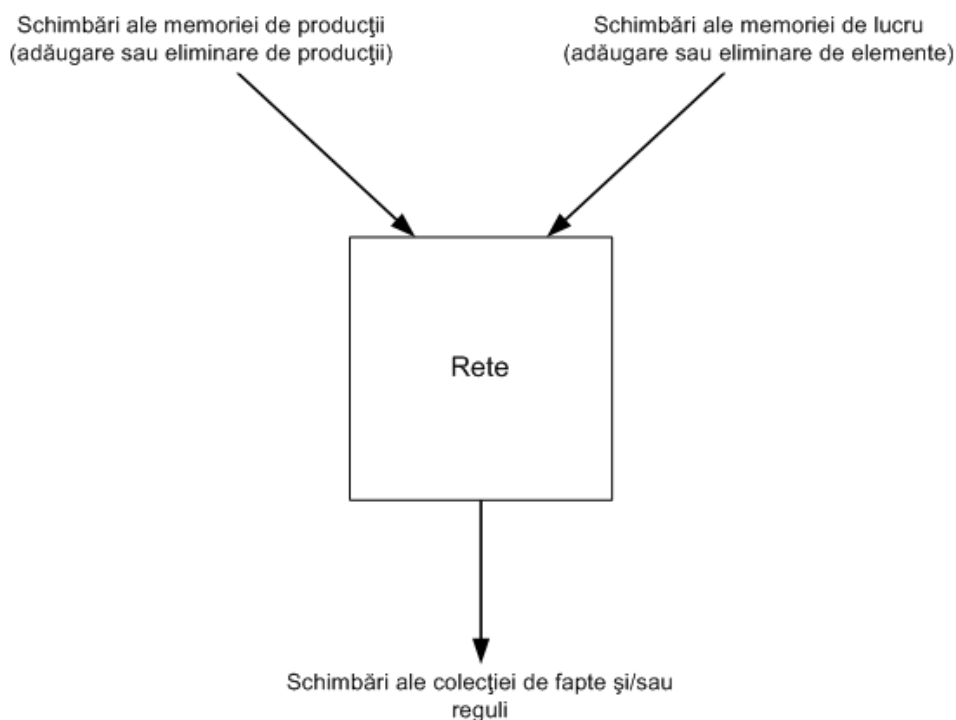


Figura 2.1: Rete văzut ca o cutie neagră

(`Cub1 VecinSus Cub2`) memoria alfa corespunzătoare condiției va reține faptul deoarece a trecut testul). Memoria beta este formată din noduri de joncțiune (eng. join nodes) și memorii beta. Nodurile de joncțiune realizează teste de consistență între elementele unei condiții iar memoriile beta stochează condiții parțial validate, adică combinații de fapte (elemente ale memoriei de lucru) care se potrivesc cu o parte din condițiile unei reguli.

În funcție de implementare, memoriile alfa ar putea realiza și teste de consistență între variabilele aceleiași condiții, de exemplu presupunem că trebuie să fie același cub în condiția (`<cub1> Înălțime > <cub1>.Lățime`) pentru a fi validată. Desigur, acest tip de reguli sunt extrem de rare, chiar unele implementări le ignoră complet.

Pentru a sumariza ce am zis până acum: memoriile alfa realizează orice test care implică un singur element al ML pe când memoriile beta testează consistența între mai multe elemente.

O analogie cu bazele de date relaționale este foarte utilă. Putem să ne imaginăm memoria curentă de lucru ca o relație și fiecare producție ca

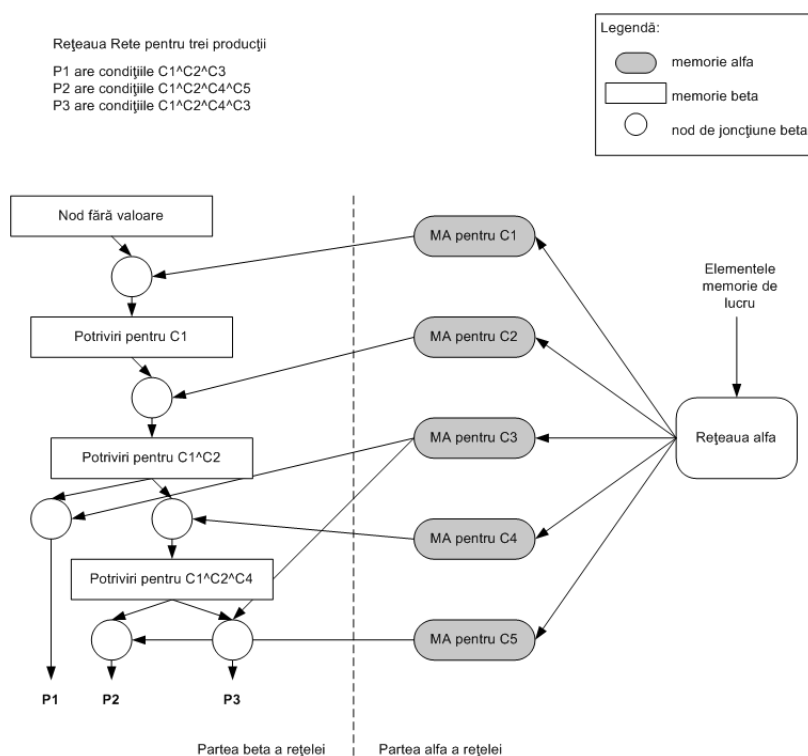


Figura 2.2: Rețeaua Rete pentru trei reguli

o interogare. Testele constante dintr-o condiție reprezintă o operație de proiecție (SELECT) peste relație. Pentru fiecare condiție c_i a sistemului există o memorie alfa care stochează rezultatul $r(c_i)$ al proiecției. Acum, considerând P o producție cu condițiile c_1, c_2, \dots, c_k , potrivirile pentru P cu ML (dacă există potriviri) sunt date de $r(c_1) \bowtie r(c_2) \bowtie \dots \bowtie r(c_k)$ unde operația de joncțiune reprezintă testele de consistență. Nodurile de joncțiune din partea beta a rețelei Rete realizează testele menționate anterior și fiecare memorie beta reține rezultatul intermediar al unei joncțiuni $r(c_1) \bowtie \dots \bowtie r(c_i)$ unde $i < k$ [2].

Atunci când o schimbare este realizată în memoria de lucru, schimbarea este trimisă prin rețeaua alfa și memoriile care sunt validate se modifică. Aceste modificări sunt apoi propagate la nodurile de joncțiune asociate memoriilor și spunem că aceste noduri au fost activate. Dacă nodurile de joncțiune sunt validate de noile fapte atunci și acestea propagă la rândul lor schimbările prin rețeaua de care se leagă, activând alte noduri. Atunci când

schimbările ajung la baza rețelei, condițiile unei producții sunt satisfăcute în totalitate și putem executa partea de concluzie corespunzătoare regulii. Se obișnuiește ca pe ultimele nivele ale rețelei să existe un nod special numit numit “nod de producție” care atunci când este activat execută o concluzie.

În funcție de originea activării, pentru fiecare nod avem:

- Activare din dreapta: Spunem că un nod este activat din dreapta atunci când el este notificat despre o schimbare de către o memorie alfa (de nod/memorie din partea alfa a rețelei).
- Activare din stânga: Spunem că un nod este activat din stânga atunci când el este notificat despre o schimbare de către un nod din partea beta a rețelei Rete.

Așadar, putem spune că un nod este activat din dreapta atunci când un element al Memoriei de Lucru este adăugat în memoria alfa corespunzătoare și este activat la stânga atunci când nodul său părinte validează o condiție parțială. În general, cele două tipuri de activări sunt tratate de proceduri diferite, dar indiferent de caz sunt interogate memoriile părinți alfa și beta ale nodului pentru a face validările de consistență. Nodurile rețelei din partea alfa sunt de un singur fel dar în partea beta putem întâlni mai multe tipuri de noduri. În funcție de operatorii logici cu care sunt legate condițiile unei reguli putem avea noduri ȘI, SAU, NOT, XOR etc. Fiecare dintre acestea tratează într-un mod diferit elementele ML de care sunt activate. Din această cauză o mare parte a algoritmului Rete este dedicat implementării acestor tipuri de joncțiuni.

Există două caracteristici importante care fac algoritmul Rete mult mai rapid decât implementările naive de potrivire. Prima caracteristică este salvarea stării: după fiecare schimbare, starea (rezultatele) este salvată în memoriile alfa și beta; la schimbările ulterioare o mare parte a acestor memorii rămân neschimbate și în acest mod se evită recalcularea unor valori. Totuși Rete funcționează rapid în acele sisteme unde elementele ML nu se schimbă foarte des; dacă acest fapt nu este îndeplinit atunci se recomandă o altă abordare.

Cea de a doua caracteristică este partajarea nodurilor între producțiile care au condiții similare. Diverse tipuri de partajări au loc în diverse părți ale rețelei. Se face partajare la nivel de memorii alfa unde se reține o condiție o singură dată indiferent de numărul ei de apariții în reguli (în exemplul din figura 2.2 se poate vedea că este partajată condiția C3 de producțiile P1 și P3). În partea beta a rețelei, dacă două sau mai multe producții au aceleași

condiții de început, atunci nodurile folosite pentru potrivire sunt partajate pentru a se evita duplicarea și pentru a reduce efortul necesar potrivirii. Tot în figura 2.2 toate cele trei producții au primele două condiții identice, iar două din producții au și a treia condiție în comun. Din cauza acestei partajări, rețeaua beta formează un arbore.

2.2 Implementarea rețelei alfa

Atunci când un fapt este adăugat în memoria de lucru, rețeaua alfa realizează validările constante necesare și stochează faptul în memoria alfa corespunzătoare. Există mai multe moduri în care se poate determina memoria alfa corespunzătoare.

Înainte de a trece la modele de implementare vom folosi lumea cuburilor prezentată în secțiunea 1.3 (pagina 12) pentru a termina condițiile pe care le vom aborda în exemple.

Definiție 6. *Condiții echivalente*

Două condiții sunt echivalente dacă fiecare din cei patru termeni (Obiect, Atribut, Operator, Valoare) ai fiecăreia sunt echivalenți, adică au aceiași valoare pentru constante sau referă același tip de model pentru variabile.

Definiție 7. *Condiții distincte*

Două condiții sunt distincte dacă nu sunt echivalente.

Următoarele două condiții sunt echivalente deoarece primul termen din fiecare se referă la un cub, atributul este același (VecinStânga), avem același operator, iar ultimul termen este tot o referință la un model de tip cub.

```
(<cub1> VecinStânga = <cub2>)
(<cub3> VecinStânga = <cub4>)
```

Următoarele două condiții nu sunt echivalente deoarece ultimul membru, în prima condiție, este o referință la un cub, iar în cea de-a doua este valoarea NULL.

```
(<cub1> VecinStânga = <cub2>)
(<cub1> VecinStânga = NULL)
```

Pentru fiecare condiție din Lumea Cuburilor vom alocă un index $C_i; i = 1..n; n = \text{numărul de condiții distincte}$. Dacă două condiții sunt echivalente

atunci vor avea același index. Din fiecare regulă este prezentată doar partea de ipoteză deoarece Rete, după cum am menționat anterior, nu optimizează și nici nu oferă soluții pentru executarea concluziei. Regulile le vom numi producții și le vom nota cu $P_j; j = 1..m; m = \text{numărul de reguli}$:

P_1 :DACĂ (<cub1> VecinStânga = <cub2>) [C_1]
 (<cub1> VecinSus = <cub3>) [C_2]
 (<cub3> VecinStânga = <cub4>) [C_1]
 P_2 :DACĂ (<cub1> VecinJos = NULL) [C_3]
 (<cub1> VecinSus = <cub2>) [C_2]
 (<cub2> VecinSus = <cub3>) [C_2]
 P_3 :DACĂ (<cub1> VecinJos = NULL) [C_3]
 (<cub1> VecinSus = <cub2>) [C_2]
 P_4 :DACĂ (<cub1> VecinJos = NULL) [C_3]
 P_5 :DACĂ (<cub1> VecinStânga = <cub2>) [C_1]
 P_6 :DACĂ (<cub1> VecinDreapta = <cub2>) [C_4]
 P_7 :DACĂ (<cub1> VecinSus = <cub2>) [C_2]
 P_8 :DACĂ (<cub1> VecinJos = <cub2>) [C_5]

Așadar, cele opt producții conțin cinci condiții distincte (pentru variabilele vom elimina indicii și vom păstra doar tipul modelului referențiat):

[C_1]: (<cub> VecinStânga = <cub>)
 [C_2]: (<cub> VecinSus = <cub>)
 [C_3]: (<cub> VecinJos = NULL)
 [C_4]: (<cub> VecinDreapta = <cub>)
 [C_5]: (<cub> VecinJos = <cub>)

2.2.1 Rețea de date (dataflow network)

Abordarea originală și probabil cea mai simplă este utilizarea unei rețele de date. Figura 2.3 ilustrează o astfel de rețea pentru Lumea Cuburilor care se construiește după cum urmează. Pentru fiecare condiție $C_i; i < n$ se construiesc, în arbore, două noduri corespunzătoare atributului și valorii. Dacă două condiții distincte au atributul comun atunci nodul corespunzător va fi partajat (după cum se vede la C_3 și C_5). Nodurile, exceptând nodul părinte și frunzele se numesc “noduri de teste constante”. Fiecare frunză este o memorie alfa.

Se observă că am construit arborele din condițiile distincte și nu am ținut cont de indicele variabilelor. Există cazuri (rare) când unele condiții nu conțin nici un test și atunci nodurile frunză ale lor se leagă direct de nodul părinte.

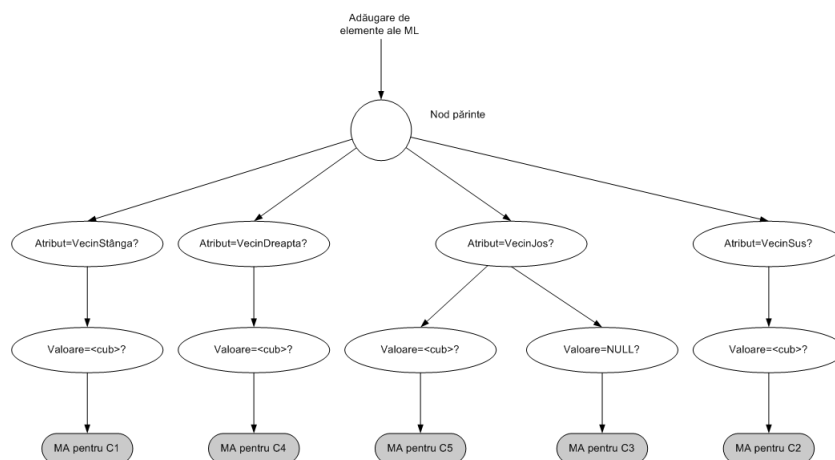


Figura 2.3: Rețea de date pentru lumea cuburilor

Nodurile rețelei și elementele ML pot fi realizate sub forma unor structuri de date. Structura `NOD_TC` (`NOD_TestConsistență`) conține referințe către nodurile copii sau către memoria alfa copil și modul în care se testează elementul (câmpul elementului). Testează “Nimic” se folosește la nodul părinte care va permite oricărui element al ML să treacă de el. Structura `ELEMENT_ML` conține, după cum s-a menționat în primul capitol, câmpurile (Model Atribut Valoare).

structura `NOD_TC`

TESTEAZĂ: ‘‘Atribut’’, ‘‘Model’’, ‘‘Valoare’’ sau ‘‘Nimic’’

REZULTAT_AȘTEPTAT: <atribut>, <model> sau <valoare>

COPII: listă de noduri sau NULL

MEMORIE_ALFA: referință la MA sau NULL

sfârșit

structura `ELEMENT_ML`

MODEL: referință spre un model

ATRIBUT: referință către un atribut al modelului

VALOARE: constantă, referință sper un model sau NULL

sfârșit

Pentru a adăuga un element al ML în rețeaua de date trebuie să verificăm dacă elementul este testat cu succes de nodurile rețelei. Pentru aceasta folosim procedura `ADAUGĂ_ELEMENT_ML` care activează nodul părinte al rețelei.

Acesta activează fiecare din copii care dacă sunt validați vor propaga activarea. Atunci când se ajunge la un nod ce nu are copii vom activa memoria alfa corespunzătoare. Procedura ACTIVEAZĂ_MA este prezentată în secțiunea 2.3.1.

```
procedura ADAUGĂ_ELEMENT_ML(elem:ELEMENT_ML)
    ACTIVEZĂ_NODUL(nod_părinte, elem)
sfârșit
```

```
procedura ACTIVEZĂ_NODUL(nod:NOD_TC,
                        elem:ELEMENT_ML)
    rezultat = nod.REZULTAT_AȘTEPTAT
    câmp = nod.TESTEZĂ
    dacă (CÂMP(câmp, elem) == rezultat) sau
        (nod.TESTEAZĂ == 'Nimic') atunci
        dacă număr(nod.COPII) <> 0 atunci
            pentru fiecare nod_copil din nod.COPII
                ACTIVEZĂ_NODUL(nod_copil, elem)
            sfârșit pentru fiecare
        altfel
            ACTIVEAZĂ_MA(elem)
        sfârșit dacă
    altfel
        returnează NEPOTRIVIRE
    sfârșit dacă
sfârșit
```

```
procedura CÂMP(câmp:NOD_TC.TESTEZĂ,
              elem_al_ml:ELEMENT_ML)
    selectează (câmp)
    cazul 'Atribut':
        returnează elem_al_ml.ATRIBUT
    cazul 'Model':
        returnează elem_al_ml.MODEL
    cazul 'Valoare':
        returnează elem_al_ml.VALOARE
    altfel:
        returnează NICI_O_VALOARE
    sfârșit selectează
sfârșit
```

Procedurile prezentate în pseudocod reprezintă doar o schemă a implementării ce s-a realizat folosind limbajul C# și este posibil să difere substanțial. De asemenea, procedurile nu realizează nici un fel de test de consistență.

2.2.2 Rețea de date cu indexare

Implementarea anterioară funcționează însă pentru un număr mare de condiții merge foarte greu. Se poate observa că un nod poate avea mai mulți copii. Dacă acest nod este activat, atunci va trebui realizată validarea fiecărui copil, lucru care durează mult și duce la o complexitate exponențială de timp. Pentru a îmbunătăți viteza de procesare fiecare nod ar putea să își indexeze copii (valorile constante, referința la modele, etc.) pentru ca validarea să se facă doar cu acele noduri care ar putea permite trecerea de ele.

Se poate folosi un tabel de dispersie sau un arbore binar balansat pentru a determina care este calea pe care o poate urma un element al ML pentru a ajunge la următorul copil al nodului curent[3].

2.2.3 Rețea de date cu indexare exhaustivă

Atât timp cât tuplele care formează elemente ML sunt de lungime fixă și finită (considerând că avem doar operația de egalitate atunci tuplele din Lumea Cuburilor au lungimea 3) există o metodă simplă și elegantă de a implementa rețeaua alfa cu doar câteva interogări. Vom presupune pentru simplitate că un tuplu al ML este (*Obiect Atribut = Valoare*) sau ignorând egalitatea avem (*Obiect Atribut Valoare*), pe scurt (*O A V*).

Putem face următoarea observație: “Pentru oricare element al ML sunt cel mult opt alfa memorii în care se poate potrivi”[2]. Acest lucru este adevărat deoarece memoria are forma (*O A V*), ceea ce înseamnă că din cele trei câmpuri fiecare poate fi testat. Notăm testul pentru câmpul *O* cu *T1*, pentru *A* cu *T2* și cu *T3* testul pentru *V*. Așadar, tuplul pentru test este (*T1 T2 T3*). Dar nu toate memoriile alfa realizează verificările tuturor câmpurilor și pentru aceasta vom folosi simbolul *** pentru a marca testele nerealizate. O memorie alfa poate avea atunci una din următoarele forme:

(<i>*</i> <i>*</i> <i>*</i>)	(<i>*</i> <i>T2 T3</i>)
(<i>*</i> <i>*</i> <i>T3</i>)	(<i>T1</i> <i>*</i> <i>T3</i>)
(<i>*</i> <i>T2</i> <i>*</i>)	(<i>T1</i> <i>T2</i> <i>*</i>)
(<i>T1</i> <i>*</i> <i>*</i>)	(<i>T1</i> <i>T2</i> <i>T3</i>)

Sunt opt posibilități de a scrie o condiție cu care un element la ML să se potrivească. Deci, fiind dat un element al ML trebuie să facem doar opt căutări într-o tabelă de dispersie pentru a determina dacă elementul se poate potrivi într-o memorie alfa.

Generalizând, putem avea tuplele de orice dimensiune r și atunci vom avea nevoie de 2^r căutări în tabela de dispersie. Acest lucru este eficient pentru valori mici ale lui r . Pentru a relaxa această ultimă constrângere putem folosi una din următoarele două strategii[2]:

- Rezultatul tabelii de dispersie, în loc să fie o memorie alfa poate fi o mică rețea de date (vezi secțiunea 2.2.1). Toate testele de egalitate cu valori constante sunt indexate în tabelă pe când restul testelor sunt “mutate” în rețeaua de date.
- Toate testele de neegalitate (mai mic, mai mare, diferit, etc.) sunt mutate în partea beta a rețelei Rete. Această strategie transformă partea alfa în ceva trivial și este dovedit ca nu crește foarte tare complexitatea părții beta. Totuși o astfel de abordare reduce potențialul de partajare a nodurile, din rețeaua beta, care au condiții comune.

Indiferent de abordare (rețea de date cu indexare sau rețea de date cu indexare exhaustivă), rețeaua alfa rămâne foarte eficientă rulând în timp aproximativ constant pentru fiecare schimbare a ML. Dacă nu avem teste de neegalitate atunci avem o complexitate $O(1)$ /fiecare schimbare a ML folosind o tabelă de dispersie exhaustivă (considerând că funcția de dispersie produce o distribuție rezonabilă a elementelor). Dacă sunt folosite și teste de neegalitate abuziv atunci nu poate fi dată o valoare limitată a complexității pentru că oricare element al ML se poate potrivi în multe memorii alfa.[2]

2.3 Implementarea nodurilor de memorie

Vom discuta implementarea nodurilor alfa și beta. Memoriile alfa stochează elemente ale ML, iar memoriile beta rețin secvențe de k elemente ale ML, secvențe care validează primele k condiții ale unei reguli (aceste secvențe sunt valide și din punct de vedere al constitenței).

Încă de la prima încercare de implementare apar două întrebări:

- Cum sunt colecțiile de (secvențe de) elemente ale ML structurate?
- Cum este o secvență de elemente ale ML reprezentată?

Pentru a răspunde la prima întrebare, cea mai simplă implementare este cea cu liste - elementele nu au nici o ordine particulară. Totuși, putem eficientiza operațiile de joncțiune prin indexarea structurilor. Considerăm producția P_1 prezentată anterior:

$$P_1 : \text{DACĂ } (\langle \text{cub1} \rangle \text{ VecinStânga} = \langle \text{cub2} \rangle) \quad [C_1]$$

$$(\langle \text{cub1} \rangle \text{ VecinSus} = \langle \text{cub3} \rangle) \quad [C_2]$$

$$(\langle \text{cub3} \rangle \text{ VecinStânga} = \langle \text{cub4} \rangle) \quad [C_1]$$

Pentru aceasta se va construi rețeaua Rete din figura 2.4. Atunci când faptul $F_5: (\text{CUB3 VecinStânga} = \text{CUB1})$ este adăugat în memoria de lucru, va fi adăugat în MA pentru C_1 și cele două noduri beta, copii ai lui MA C_1 , vor fi activate din dreapta. Pentru nodul J1 nu se va face nici o validare deoarece este un nod gol, legate de un părinte gol. În schimb pentru J3 se va încerca potrivirea cu secvențele de potriviri din memoria corespunzătoare J2 (dacă există vreo secvență care are valoarea din ultimul termen $\langle \text{cub3} \rangle$ identică cu o valoare din memoria alfa C_1).

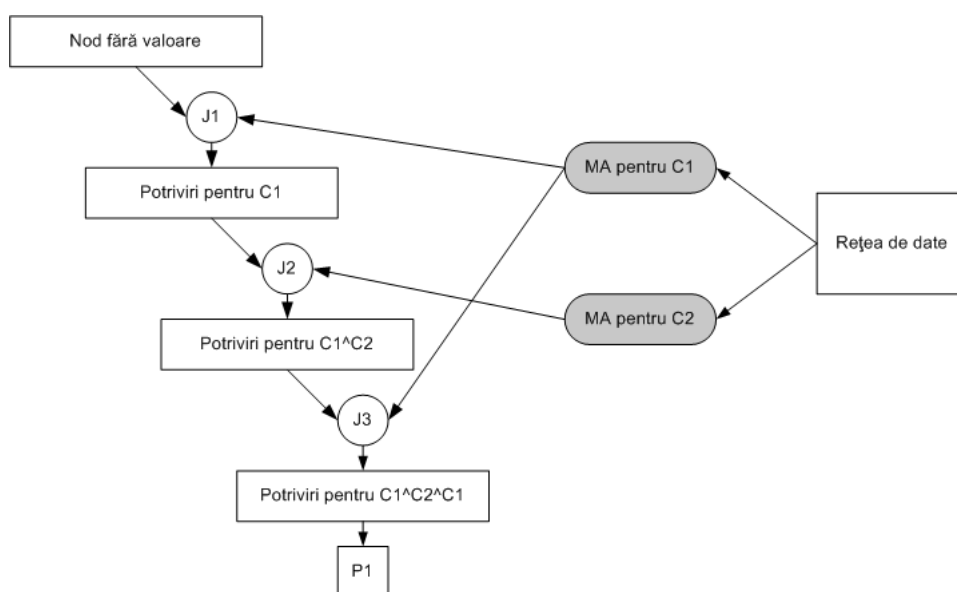


Figura 2.4: Parte din rețeaua Rete pentru producția P_1

Fără nici un fel de indexare, procesul de căutare ar necesita iterarea prin toate potrivirile din memoria beta părinte. Similar, când o potrivire este adăugată în memoria beta, se activează din stânga nodurile copii și este necesară căutarea secvențială printre elementele din memoria alfa pentru potriviri.

Cea mai frecventă metodă de indexare este tabela de dispersie. Arborii sunt o altă posibilitate de indexare dar nu s-au văzut prea multe implementări Rete folosind această metodă iar Barachini (1991) a arătat că tabela de dispersie, în general, subclasează arborii binari nebalansați. Totuși, alegerea unei tabele de dispersie duce la o mare dilemă: cât de mare trebuie să fie tabela având în vedere că va fi reținută în fiecare nod. Dacă tabela e prea mică vom avea foarte multe coliziuni și performanțe slabe, iar dacă tabela e prea mare vom face risipă de memorie. Am putea redimensiona dinamic tabela pe măsură ce elementele sunt adăugate/eliminate dar aceasta va transforma un nod extrem simplu într-o secvență de cod foarte complicată.

Soluția cea mai des întâlnită la această problemă este indexarea tuturor potrivirilor într-o singură, mare tabelă de dispersie și toate elemente ML (cele ținute în memoriile alfa) într-o altă mare tabelă. Dimensiunile acestor tabele se stabilesc a priori și sunt numere mari de ordinul sutelor. Funcția de dispersie este o funcție de potrivire, dar și o funcție de identificare. Valoarea acesteia ar trebui să identifice, în mod unic elementele dar și să ajute la potrivirea din nodurile de joncțiune.

După cum am menționat mai sus, indexarea poate accelera procesul de testare al potrivirii dar în același timp are și dezavantaje. În primul rând, timpul necesar adăugării/eliminării elementelor crește deoarece este nevoie ca indexul de dispersie să se actualizeze. În al doilea rând, capacitatea de a partaja noduri scade și e nevoie ca uneori să se creeze memorii duplicate, conținând aceiași informație, dar indexate în alt mod.

În ciuda acestor dezavantaje, rezultatele empirice indică faptul că, în practică memoriile indexate sunt multe mai rapide decât cele stocate ca liste neordonate. Gupta et al.[4] au descoperit că folosirea indexării a făcut algoritmul mai rapid cu un factor de 1.2-3.5 în câteva sisteme OPS5, iar Scales[8] a raportat un factor de îmbunătățire de 1.2-1.3 în sistemele Soar.

Revenind la cea de a doua întrebare - cum este o secvență de elemente ale ML reprezentată? - există două posibilități. O secvență poate fi reprezentată ca un șir sau ca o listă. Alegerea pare a fi trivială și naturală în favoarea șirului care ne permite să obținem direct, pe baza unui index i , elementul al i -lea în timp constant pe când o listă ar necesita iterarea prin primele $i - 1$ elemente pentru a-l obține pe al i -lea.

Totuși, stocarea sub formă de șir poate duce la memorarea de informații redundante și implicit la utilizarea a mai mult spațiu. Pentru fiecare memorie beta, se rețin primele $i > 1$ condiții dintr-o producție, există o altă memorie beta - din punct de vedere al arborilor fiind nodul bunic (se sare

peste nodul de joncțiune de care se leagă direct) - conținând primele $i - 1$ condiții. Dacă (e_1, e_2, \dots, e_i) este o succesiune de elemente potrivite pentru primele i condiții, atunci $(e_1, e_2, \dots, e_{i-1})$ este obligatoriu o potrivire pentru primele $i - 1$ condiții. Acest lucru înseamnă că orice succesiune dintr-un nod aflat într-un nivel mai jos în arborele beta poate fi reprezentat succint ca o pereche (părinte, i), unde părinte este un pointer către succesiunea de elemente potrivite din prima memorie beta părinte. Apelând la această metodă, fiecare succesiune devine efectiv o listă înlănțuită, legată de părinte și reprezentând succesiunea de potriviri în ordine inversă. Pentru uniformitate putem lega nodul memorie beta cel mai de sus din arbore de un nod fals conținând succesiunea vidă de potriviri.

Folosind un șir de elemente pentru primele i condiții, complexitatea de spațiu a nodului care le reține este $O(i)$ pe când folosind metoda listelor înlănțuite fiecare nod are complexitatea $O(1)$. Acest lucru este extrem de benefic pentru sistemele care au un număr foarte mare de condiții. Dacă avem o producție cu C condiții, care este potrivită complet, atunci utilizând șiruri avem o complexitate minimă de spațiu de $1 + 2 + \dots + C = O(C^2)$, pe când pentru utilizarea listelor înlănțuite duce la o complexitate, mai bună, $O(C)$.

Sumarizând, fiecare metodă are dezavantaje: șirul necesită mai mult spațiu de stocare, iar listele înlănțuite necesită iterarea prin ele la fiecare modificare. Totuși, uneori este nevoie, de cât mai puțin spațiu de stocare folosit, dar și acces rapid la elemente arbitrare atunci când au loc activări de noduri. Este nevoie de un compromis și nu se poate da un răspuns exact. În funcție de sistem, o variantă poate fi mai potrivită decât cealaltă.

2.3.1 Implementarea memoriilor alfa

Un element al ML conține doar cele patru câmpuri (simboluri) ale sale (Obiect Atribut Operator Valoare):

```
structura ELEMENT_ML
  câmpuri: șir de 4 simboluri
sfârșit
```

O memorie alfa reține lista de elemente ML, plus o listă cu nodurile de joncțiune succesori (copii):

```
structura MEMORIE_ALFA
  elemente: listă de ELEMENT_ML
```

```

    succesori: listă de NODURI_JOIN
sfârșit

```

Atunci când un element este filtrat prin rețeaua alfa și ajunge la o memorie alfa, îl adăugăm în lista de elemente din memorie și informăm (activăm) fiecare nod de joncțiune atașat:

```

procedura ACTIVARE_MA(memorie: MEMORIE_ALFA,
                      e:ELEMENT_ML)
    inserează e în memorie.elemente
    pentru fiecare copil din memorie.succesori
        ACTIVARE_DREAPTA_NOD_JONCȚIUNE(copil, e)
    sfârșit pentru fiecare
sfârșit

```

2.3.2 Implementarea memoriilor beta

Folosind metoda listelor înlănțuite prezentată anterior, o potrivire dintr-un nod al memoriei beta conține legătura către părinte și un element al ML.

```

structura POTRIVIRE
    părinte: legătură către o POTRIVIRE de mai sus în arbore
    elemente: ELEMENT_ML
sfârșit

```

O memorie beta reține o listă de potriviri și legături către nodurile copii (alte noduri beta din rețea). Înainte de a prezenta structura de date reamintim că vom avea nevoie de proceduri pentru activare din stânga și din dreapta și fiecare trebuie să poată decide ce fel de nod este activat, așadar vom avea nevoie și de structură de branching sau un indexator conform cu tipul nodului activat. Vom folosi “...” pentru a specifica locația unde se pot introduce date specifice aceluși tip de nod. În implementarea efectivă acest lucru se poate realiza prin moștenire. Fiecare nod al rețelei beta va fi reprezentat ca o structură NOD_RETE:

```

structura NOD_RETE
    tip: ‘‘memorie-beta’’, ‘‘nod-joncțiune’’, ‘‘nod-producție’’
    copii: listă de NOD_RETE
    părinte: NOD_RETE
    ...
sfârșit

```

Pe măsură ce vom prezenta noi noduri, vom arăta doar noile informații pentru ele. Rețineți că fiecare nod conține `tip` (cu valoarea corespunzătoare), copii și părinte.

Revenind la nodurile beta, singura informație suplimentară față de un nod Rete este o listă de potriviri:

```
structura MEMORIE_BETA
  potriviri: listă de POTRIVIRE
sfârșit
```

Atunci când o memorie beta (MB) este informată despre o nouă potrivire (informația conținând o potrivire existentă și niște elemente ale ML), construim o nouă potrivire, o adăugăm în lista din memoria beta și informăm (activăm) fiecare nod copil:

```
procedure MB_ACTIVARE_STÂNGA(nod:MEMORIE_BETA,
                             p:POTRIVIRE,
                             e:ELEMENT_ML)
  potrivire_nouă=alocă_memorie(POTRIVIRE)
  potrivire_nouă.părinte=p
  potrivire_nouă.element=e
  inserează potrivire_nouă în nod.potriviri
  pentru fiecare copil din nod.copii
    ACTIVARE_STÂNGA_NOD_JONCTIUNE(copil, potrivire_nouă)
  sfârșit pentru fiecare
sfârșit
```

2.3.3 Implementarea nodurilor de producții

Menționăm implementarea nodurilor de producție tot în această secțiune deoarece sunt foarte asemănătoare cu nodurile de memorie (din anumite puncte de vedere). Implementarea acestora diferă de la sistem la sistem și nu vom da nici un fel de pseudocod. Un nod de producție poate stoca potriviri precum un nod beta cu deosebirea că nodurile de producții stochează potriviri integrale, pe când nodurile beta potriviri parțiale. Nodul de producție, atunci când este activat informează într-un fel sau altul, sistemul despre o nouă potrivire completă.

În general, un nod de producție conține și specificații despre producția căreia îi corespunde și acțiunile care ar trebui executate la o potrivire completă. Tot în funcție de implementare putem avea și metadate despre variabilele aflate în acea producție.

2.4 Implementarea nodurilor de joncțiune

După cum am menționat la începutul acestui capitol, un nod de joncțiune poate fi activat din dreapta atunci când un element al ML este adăugat în memoria alfa corespunzătoare, sau poate fi activat din stânga atunci când o potrivire este adăugată în memoria beta părinte. În oricare din cele două cazuri, cealaltă memorie va fi interogată pentru elemente care au potriviri de variabile cu noile elemente. Dacă astfel de potriviri sunt găsite, atunci ele sunt propagate nodurilor copii.

Structura de date pentru un nod de joncțiune trebuie, așadar, să conțină referințe către cele două memorii (alfa și beta) părinte pentru a putea fi căutate, metode pentru a realiza teste de consistență, către potrivire și o listă cu noduri copii. Folosind modelul structurii NOD_RETE de la pagina 32 avem deja lista nodurilor copii iar câmpul `părinte` ne oferă una din referințele către părinți, mai exact spre memoria beta părinte. Vom adăuga două câmpuri suplimentare:

```
structura NOD_JONCȚIUNE
  mem_alfa: MEMORIE_ALFA
  teste: listă de TEST_JONCȚIUNE
sfârșit
```

TEST_JONCȚIUNE este o structură reprezentând locația a două câmpuri a căror valoare trebuie să fie egală pentru ca joncțiunea să fie consistentă:

```
structura TEST_JONCȚIUNE
  câmp1: ‘obiect’, ‘atribut’ sau ‘valoare’
  câmp2: ‘obiect’, ‘atribut’ sau ‘valoare’
  locație_câmp_2: întreg
sfârșit
```

Trebuie menționat că `locație_câmp_2` reprezintă numărul condiției cu care trebuie făcută potrivirea. Pentru producția P_1 :

$$P_1 : \text{DACĂ } (\langle \text{cub1} \rangle \text{ VecinStânga} = \langle \text{cub2} \rangle) \quad [C_1]$$

$$(\langle \text{cub1} \rangle \text{ VecinSus} = \langle \text{cub3} \rangle) \quad [C_2]$$

$$(\langle \text{cub3} \rangle \text{ VecinStânga} = \langle \text{cub4} \rangle) \quad [C_1]$$

nodul de joncțiune corespunzător ultimei condiții care va verifica dacă există consistență între $\langle \text{cub3} \rangle$ din ultima condiție și $\langle \text{cub3} \rangle$ din cea de a doua arată în felul următor:

```

structura TEST_JONCTIUNE_ULTIMA_CONDIȚIE
  câmp1: ‘‘obiect’’
  câmp2: ‘‘valoare’’
  locație_câmp_2: 2
sfârșit

```

În momentul activării din dreapta căutăm în memoria beta pentru a găsi una sau mai multe potriviri pentru care teste de consistență sunt trecute. Orice combinație bună este propagată nodurilor copii. Similar, la activarea din stânga, căutăm în memoria alfa pentru a găsi orice element al ML care trece testul; din nou, potrivirile sunt propagate:

```

procedura ACTIVARE_DREAPTA_NOD_JONCTIUNE(nod:NOD_JONCTIUNE,
                                           e:ELEMENT_ML)
  pentru fiecare potrivire din nod.părinte.potriviri
    dacă E_POTRIVIRE(nod.teste,potrivire,e) atunci
      pentru fiecare copil din nod.copii
        MB_ACTIVARE_STÂNGA(copil, potrivire,e)
      sfârșit pentru fiecare
    sfârșit dacă
  sfârșit pentru fiecare
sfârșit

```

```

procedura ACTIVARE_STÂNGA_NOD_JONCTIUNE(nod:NOD_JONCTIUNE,
                                           p:POTRIVIRE)
  pentru fiecare elem_ml din nod.mem_alfa.elemente
    dacă E_POTRIVIRE(nod.teste, p, elem_ml) atunci
      pentru fiecare copil din nod.copii
        MB_ACTIVARE_STÂNGA(copil, p, elem_ml)
      sfârșit pentru fiecare
    sfârșit dacă
  sfârșit pentru fiecare
sfârșit

```

```

procedura E_POTRIVIRE(teste:listă de TEST_JONCTIUNE,
                      p: POTRIVIRE,
                      e: ELEMENT_ML)
  pentru fiecare test din teste
    arg1=e[test.câmp1]
    condiție=selectează condiția a [test.locație_câmp_2]-ia

```

```

    arg2=condiție[test.câmp2]
    dacă arg1<>arg2 atunci
        returnează FALS;
    sfârșit dacă
sfârșit pentru fiecare
returnează ADEVĂRAT
sfârșit

```

Remarcăm câteva lucruri despre procedurile anterioare. Pentru a putea folosi procedurile de activare pe nodurile din vârful rețelei, trebuie să avem ca părinte niște noduri goale (eng. dummy) care să acționeze ca niște memorii fără elemente. De asemenea, procedurile prezentate sub formă de pseudocod presupun că se realizează doar teste de egalitate între două câmpuri; este evident și banal modul cum acestea se pot extinde și pentru alte tipuri de teste. Ultima observație se referă la faptul că se presupune că memoriile nu sunt indexate de nici un fel și de aceea este nevoie să iterăm prin toate elementele/faptele din memorii; indexarea ar accelera procesul de găsimă al potrivirilor.

2.5 Eliminarea elementelor ML

Nu toate implementările Rete permit eliminarea elementelor ML. Dacă totuși acest lucru este permis, eliminarea unui element va duce la scoaterea lui din ML și notificarea/actualizarea nodurilor alfa, beta, joncțiune și producție corespunzătoare. Există mai multe modalități de a face acest lucru.

În implementarea originală a algoritmului, eliminarea era tratată în același mod ca și adăugarea. Vom numi, în continuare, această metodă “eliminare bazată pe potrivire”. Ideea de bază este că fiecare procedură primește un argument care specifică dacă este vorba de adăugare sau eliminare; acest ultim argument se numește **adăugare** și poate lua valorile **ADEVĂRAT** pentru cazul când adăugăm un element, respectiv **FALS** pentru eliminare. Valoarea acestui argument se propagă în toată rețeaua. Deoarece metoda eliminării bazată pe potrivire tratează în mod similar adăugarea și eliminarea de elemente ale ML, ea este considerată simplă și elegantă.

Din păcate, este înceată cel puțin comparată cu alte metode posibile de eliminare. Folosind metoda eliminării bazată pe potrivire costul eliminării unui element este același cu cel al adăugării pentru că este apelată aceeași procedură care face aceeași cantitate de muncă. Problema este că nici o

informație obținută în timpul adăugării elementului nu este folosită la eliminarea lui. Există cel puțin trei metode de a trata eliminarea care folosesc aceste informații[2].

“Eliminarea bazată pe scanare” presupune renunțarea la revocarea schimbărilor prin analiza testelor de consistență și pur și simplu scanează memoriile căutând elemente/potriviri care conțin elementul ce trebuie eliminat. Atunci când un nod de joncțiune este activat din dreapta pentru a elimina elementul e al ML, trimitem acest element la memoria sau memoriile lui, ștergem potrivirile și trimitem comanda de ștergere a lui e și la nodurile copil. Similar, când un nod este activat din stânga pentru a elimina potrivirea p , trimitem p -ul prin memorii și dacă se întâmplă ca părintele vreunei memorii să conțină p îl ștergem și trimitem comanda copiilor. Trebuie remarcat că partea de căutare a procedurii de ștergere din stânga poate fi făcută eficient doar dacă folosim implementarea bazată pe liste înlănțuite și nu pe șiruri. Scales[8] a obținut un spor de performanță de 28% înlocuind eliminare bazată pe potrivire cu eliminarea bazată pe scanare în Soar; Barachini[1] a raportat o îmbunătățire de 10% folosind eliminare bazată pe potrivire cu o variantă a acestei metode.

Probabil cea mai rapidă metodă de a elimina elemente este notarea precisă, a priori, a elementelor care trebuie eliminate. Ideea stă la baza celor două metode, “eliminare bazată pe liste” și “eliminare bazată pe arbori” și presupune păstrarea unor referințe suplimentare spre elementele ML și/sau potriviri astfel încât, atunci când un element al ML este eliminat să putem găsi toate potrivirile care îl conțin prin simpla urmărire a referinței.

Eliminarea bazată pe liste, propusă de Scales[8] în 1986, presupune stocarea pentru fiecare element e al ML a unei liste cu toate potrivirile ce implică e . Atunci când e este șters, iterăm prin acea listă și ștergem fiecare potrivire din ea. Dezavantajul este necesitatea de a folosi mai mult spațiu de stocare și durată mai mare de timp atunci când se crează o potrivire. O potrivire (e_1, e_2, \dots, e_i) trebuie adăugată în fiecare din cele i liste corespunzătoare elementelor e_i . Nu există nici un rezultat empiric, al folosirii eliminării bazate pe liste în literatură, deci este neclar dacă funcționează bine sau nu în practică (sau chiar dacă a fost vreodată implementat)[2].

Metoda eliminării bazată pe arbori presupune că pentru fiecare elemente e al ML să păstrăm o listă a tuturor potrivirilor pentru care e este ultimul element. În fiecare potrivire p , reținem o listă cu copiii lui p . Aceste referințe la copii ne permit să găsim toți descendenții lui p din memoriile beta aflate mai jos în arbore (în memorii beta sau noduri de producție). Atunci când e este eliminat, iterăm prin subarbori și ștergem totul din ei. Desigur, aceste

referințe suplimentare înseamnă mai multă memorie utilizată și timp mai mult pierdut pentru a crea aceste legături. Rezultatele empirice au arătat că timpul câștigat la eliminarea elementelor este mai mare decât timpul necesar pentru a crea referințele și, deci, aceste modificări sunt justificate. Atunci când autorul a înlocuit eliminarea bazată pe potriviri cu eliminarea bazată pe arbori, în Soar, s-a obținut o îmbunătățire a vitezei cu un factor de 1.3; Barachini[1] a estimat un factor de îmbunătățire de 1.25 pentru un sistem asemănător cu OPS5.

Pentru a putea implementa eliminarea bazată pe arbori, vom modifica structura de date ELEMENT_ML pentru a include o listă a tuturor alfa memoriilor conținând acel element și o listă a tuturor potrivilor ce au elementul curent ca ultim element:

```
structura ELEMENT_ML
    câmpuri: șir de simboluri
    memorii_alfa: listă de MEMORIE_ALFA
    potriviri: listă de POTRIVIRE
sfârșit
```

Structura pentru o potrivire este îmbogățită pentru a conține referințe către nodul de memorie care o reține și o listă de copii:

```
structura POTRIVIRE
    părinte: legătură către o POTRIVIRE de mai sus în arbore
    elemente: ELEMENT_ML
    nod: NOD_RETE
    copii: listă de POTRIVIRE
sfârșit
```

Vom modifica procedurile ACTIVARE_MA și MB_ACTIVARE_STÂNGA pentru a inițializa lista. Atunci când un element e al ML este adăugat în memoria alfa a, vom adăuga a la sfârșitul listei e.memorii_alfa.

```
procedura ACTIVARE_MA(a: MEMORIE_ALFA,
                    e: ELEMENT_ML)
    inserează e în a.elemente
    inserează a în e.memorii_alfa
    pentru fiecare copil din a.sucesori
        ACTIVARE_DREAPTA_NOD_JONCȚIUNE(copil, e)
    sfârșit pentru fiecare
sfârșit
```

Similar, atunci o potrivire nouă $pot = (p, e)$ este adăugată în memoria beta, adăugăm pot în $p.copii$ și în $e.potriviri$. De asemenea, completăm noul câmp nod din potrivire. Pentru simplificarea pseudocodului vom folosi o procedură ajutătoare `CREAZĂ_POTRIVIRE` care va construi o nouă potrivire și va inițializa corespunzător câmpurile[2].

```

procedura CREAZĂ_POTRIVIRE(nod:NOD_RETE,
                           părinte:POTRIVIRE,
                           e:ELEMENT_ML)
    pot=alocă_memorie(POTRIVIRE)
    pot.părinte=părinte
    pot.element=e
    pot.nod=nod
    pot.copii=NULL
    inserează pot în părinte.copii
    inserează pot în e.potriviri
    returnează pot
sfârșit

procedură MB_ACTIVARE_STÂNGA(nod:MEMORIE_BETA,
                              p:POTRIVIRE,
                              e:ELEMENT_ML)
    potrivire_nouă=CREAZĂ_POTRIVIRE(nod,p,e)
    inserează potrivire_nouă în nod.potriviri
    pentru fiecare copil din nod.copii
        ACTIVARE_STÂNGA_NOD_JONCTIUNE(copil, potrivire_nouă)
    sfârșit pentru fiecare
sfârșit

```

Acum, pentru a elimina un element al ML, trebuie doar să-l eliminăm din fiecare alfa memorie care îl conține și să apelăm procedura ajutătoare `ȘTERGE_POTRIVIRE` pentru a șterge toate potrivirile ce îl conțin:

```

procedură ȘTERGE_ELEMENT_ML(e:ELEMENT_ML)
    pentru fiecare alfa_mem din e.alfa_memorii
        șterge e din lista alfa_mem.elemente
    sfârșit pentru fiecare
    cât timp e.potriviri conține elemente
        ȘTERGE_POTRIVIRE(prima potrivire din e.potriviri)
    sfârșit cât timp
sfârșit

```

Trebuie remarcat faptul că folosim o structură repetitivă cu număr necunoscut de pași în locul uneia cu număr fix de pași. De asemenea, extragem primul element din `e.potriviri` deoarece ar fi nesigură folosirea unui iterator pentru că procedura `ȘTERGE_POTRIVIRE` duce la modificarea colecției de potriviri și iteratorul ar deveni invalid referențind o zonă de memorie nealocată.

Procedura ajutătoare `ȘTERGE_POTRIVIRE` șterge o potrivire și toate aparițiile ei din arborele de descendenți. Pentru simplitate, pseudocodul este recursiv; implementarea efectivă s-ar putea dovedi mult mai rapidă folosind o metoda iterativă de parcurgere a arborelui[2].

```

procedura ȘTERGE_POTRIVIRE(p:POTRIVIRE)
    cât timp p.copii conține elemente
        ȘTERGE_POTRIVIRE(prima potrivire din p.copii)
    sfârșit cât timp
    șterge p din lista p.nod.elemente
    șterge p din lista p.element.potriviri
    șterge p din lista p.părinte.copii
    alocă_memorie(p)
sfârșit

```

2.6 Condiții negate

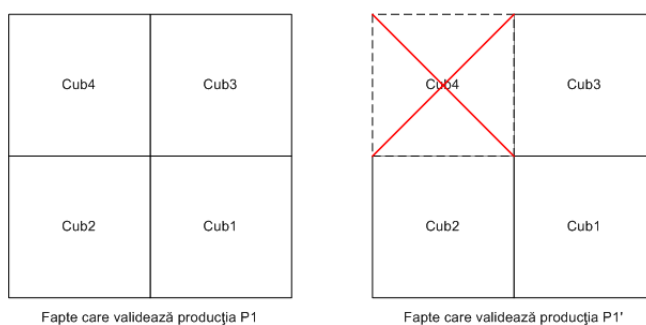
Până acum am discutat despre condiții care implică *prezența* unui element în ML. Acum ne vom axa pe condiții care testează *absența* unor elemente din ML.

Vom considera arhicunoscuta producție P_1 doar că vom nega ultima condiție (o condiție negată este indicată prin semnul “-” care o precede); această nouă producție va fi numită P'_1 :

$$\begin{aligned}
 P'_1 : \text{DACĂ } (<\text{cub1}> \text{ VecinStânga} = <\text{cub2}>) & [C_1] \\
 & (<\text{cub1}> \text{ VecinSus} = <\text{cub3}>) & [C_2] \\
 & -(<\text{cub3}> \text{ VecinStânga} = <\text{cub4}>) & [C_3]
 \end{aligned}$$

Această producție va fi complet validată dacă există un `Cub1`, aflat la dreapta unui `Cub2`, având deasupra un `Cub3`, despre care nu se știe că are la stânga un alt cub. Această configurație de cuburi este prezentată în figura 2.5.

Pentru a putea realiza testele de consistență ale acestei producții avem nevoie de un nod care va primi primele două elemente ale ML (e_1, e_2) care validează C_1 și C_2 și va le va propaga mai departe prin rețea dacă și numai

Figura 2.5: Diferența dintre P_1 și P'_1

dacă nu există nici un element e_3 care ar realiza testul de consistență dintre C_2 și C_3 nenegat.

Metoda standard de a face aceste lucru este folosirea unui nod Rete diferit, numit “nod negativ”, pentru condiții negate, nod ilustrat în figura 2.6. Nodul negativ pentru condiția C_i reține toate potrivirile pentru condițiile anterioare, exact ca o memorie beta; este legat în rețeaua beta ca un copil al nodului de joncțiune C_{i-1} , exact ca o memorie beta. De asemenea, nodul negativ este legat și de o memorie alfa ca și când ar fi un nod pozitiv. Până acum pare că un nod negativ este doar o combinație de nod de joncțiune cu memorie beta. Nodul negativ realizează joncțiunea între elementele ML din memoria alfa și fiecare potrivire din condițiile anterioare, la fel ca un nod (pozitiv) de joncțiune; totuși, în loc să propage rezultatele joncțiunii mai departe în rețea, le reține într-o memorie locală pentru fiecare potrivire. O potrivire este propagată dacă și numai dacă memoria asociată ei este goală, acest lucru indicând absența unui element al ML.

Beta memoriile și nodurile de joncțiune pozitive sunt noduri separate, pe când memoria și nodul negativ sunt combinate într-un singur nod. Acest lucru este făcut doar pentru a salva spațiu. În cazul joncțiunii pozitive avem această separare pentru a permite diferitelor tipuri de noduri de joncțiune să partajeze aceeași memorie beta. Nu putem partaja memorii ale nodurilor negative, deoarece trebuie să stocăm local rezultate pentru fiecare potrivire din memorie, iar aceste rezultate pot fi diferite pentru diverse condiții (negative).

Pentru a implementa toate cele despre care am vorbit anterior, avem nevoie de un câmp `rezultate_joncțiune` în structura de date pentru o potrivire; acest câmp reține o referință către o listă de structuri de tip `REZULTAT_JONCȚIUNE_NEGATIVĂ`. Nevoia pentru o astfel de listă în locul

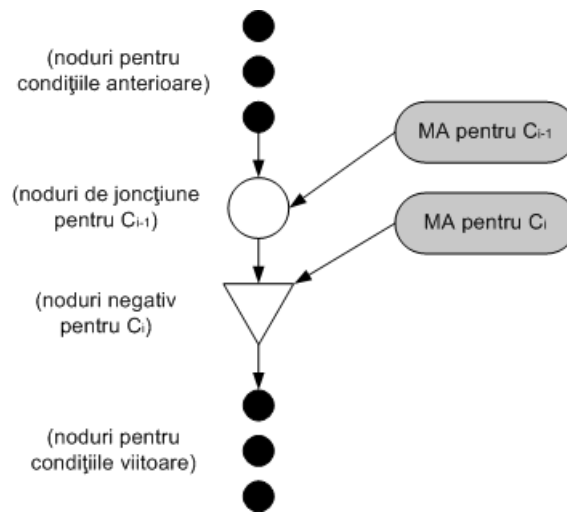


Figura 2.6: Nod negativ

unei simple liste de `ELEMENT_ML` va deveni evidentă atunci când vom discuta despre eliminarea elementelor `ML`[2]. Câmpul `rezultate_joncțiune` nu va fi folosit pentru potrivirile din memoriile beta.

structura `POTRIVIRE`

părinte: legătură către o `POTRIVIRE` de mai sus în arbore

elemente: `ELEMENT_ML`

nod: `NOD_RETE`

copii: listă de `POTRIVIRE`

rezultate_negate: listă de `REZULTAT_JONCȚIUNE_NEGATIVĂ`

sfârșit

Un `REZULTAT_JONCȚIUNE_NEGATIVĂ` specifică perechea (p, e) care rezultă în urma joncțiunii:

structura `REZULTAT_JONCȚIUNE_NEGATIVĂ`

pot: `POTRIVIRE`

element: `ELEMENT_ML`

sfârșit

Trebuie să adăugăm și un câmp `rezultate_negate` în structura de date `ELEMENT_ML`, pentru a reține o referință către o listă a tuturor structurilor `REZULTAT_JONCȚIUNE_NEGATIVĂ` ce implică elementul.

```

structura ELEMENT_ML
    câmpuri: șir de simboluri
    memorii_alfa: listă de MEMORIE_ALFA
    potriviri: listă de POTRIVIRE
    rezultate_negate: listă de REZULTAT_JONȚIUNE_NEGATIVĂ
sfârșit

```

Structura de date pentru un nod negativ arată ca o combinație a structurilor pentru nod de joncțiune și memorie beta

```

structura NOD_NEGATIV
    potriviri: listă de POTRIVIRE
    mem_alfa: MEMORIE_ALFA
    teste: listă de TEST_JONȚIUNE
sfârșit

```

În continuare prezentăm proceduri pentru activarea din stânga și dreapta a nodurilor negative. În cazul activării la stânga, construim și reținem o nouă potrivire, realizăm joncțiunea pentru potrivire, stocăm rezultatul potrivirii în structură și propagăm potrivirea la nodurile succesori dacă nu am obținut rezultate în urma joncțiunii.

```

procedura NOD_NEGATIV_ACTIVARE_STÂNGA(nod:NOD_NEGATIV,
                                         p:POTRIVIRE,
                                         e:ELEMENT_ML)

    potrivire_nouă=alocă_memorie(POTRIVIRE)
    inserează potrivire_nouă în nod.potriviri
    potrivire_nouă.rezultate_joncțiune=NULL
    pentru fiecare element din nod.alfa_mem.elemente
        dacă E_POTRIVIRE(nod.teste,
                          potrivire_nouă,
                          element) atunci
            rezultat=alocă_memorie(REZULTAT_JONȚIUNE_NEGATIVĂ)
            rezultat.pot=alocă_memorie(POTRIVIRE)
            rezultat.element=e
            inserează rezultat în potrivire_nouă.rezultate_negate
            inserează rezultat în e.rezultate_negate
        sfârșit dacă
    sfârșit pentru fiecare
    dacă potrivire_nouă.rezultate nu conține elemente atunci

```

```

    pentru fiecare copil din nod.copii
        MB_ACTIVARE_STÂNGA(copil, potrivire_nouă, NULL)
    sfârșit pentru fiecare
sfârșit dacă
sfârșit

```

Trebuie remarcat faptul că ultimul apel de procedură din pseudocodul anterior transmite NULL ca ultim parametru deoarece nici un element al ML nu a fost potrivit. Acest lucru duce la două consecințe: trebuie să modificăm procedurile CREAȚĂ_POTRIVIRE și ȘTERGE_POTRIVIRE pentru a trata și cazul parametrului nul. (O implementare alternativă ar putea fi transmiterea unui element special, gol, în locul valorii nule[2].)

```

procedura CREAȚĂ_POTRIVIRE(nod:NOD_REȚE,
                          părinte:POTRIVIRE,
                          e:ELEMENT_ML)
    pot=alocă_memorie(POTRIVIRE)
    pot.părinte=părinte
    pot.element=e
    pot.nod=nod
    pot.copii=NULL
    inserează pot în părinte.copii
    dacă e <> NULL atunci
        inserează pot în e.potriviri
    sfârșit dacă
    returnează pot
sfârșit

procedura ȘTERGE_POTRIVIRE(p:POTRIVIRE)
    cât timp p.copii conține elemente
        ȘTERGE_POTRIVIRE(prima potrivire din p.copii)
    sfârșit cât timp
    șterge p din lista p.nod.elemente
    dacă p.element <> NULL atunci
        șterge p din lista p.element.potriviri
    sfârșit dacă
    șterge p din lista p.părinte.copii
    dealocă_memorie(p)
sfârșit

```

Atunci când activăm din dreapta nodul negativ (când un element ML este adăugat într-o memorie alfa) trebuie să ne uităm la toate potrivirile din nod pentru a determina dacă există vreuna care se potrivește cu elementul. Dacă găsim vreo astfel de potrivire, atunci adăugăm elementul în memoria ei. De asemenea, avem și o situație specială, dacă numărul de elemente ale unei memorii de potrivire devine diferit de zero trebuie să apelăm procedura ȘTERGE_DESCENDENȚI_POTRIVIRE pentru a șterge vechile potriviri. În această ultimă situație toate modificările făcute de regula respectivă trebuie revocate.

```

procedura NOD_NEGATIV_ACTIVARE_DREAPTA(nod:NOD_NEGATIV,
                                         e:ELEMENT_ML)
    pentru fiecare p din nod.potriviri
        dacă E_POTRIVIRE(nod.teste, p, e) atunci
            dacă p.rezultate_negate<>NULL atunci
                ȘTERGE_DESCENDENȚI_POTRIVIRE(p)
            sfârșit dacă
            rezultat=alocă_memorie(REZULTAT_JONȚIUNE_NEGATIVĂ)
            rezultat.pot=p
            rezultat.element=e
            inserează rezultat în p.rezultate_negate
            inserează rezultat în e.rezultate_negate
        sfârșit dacă
    sfârșit pentru fiecare
sfârșit

procedura ȘTERGE_DESCENDENȚI_POTRIVIRE(p:POTRIVIRE)
    cât timp p.copii conține elemente
        ȘTERGE_DESCENDENȚI_POTRIVIRE(primul din p.potriviri)
    sfârșit cât timp
sfârșit

```

Până acum am discutat despre adăugarea elementelor ML și a potrivirilor atunci când avem noduri negative. Acum vom aborda problema și din perspectiva ștergerii. Atunci când un element al ML este șters, trebuie să actualizăm toate nodurile ce îl implică. Putem localiza eficient toate nodurile de joncțiune ce trebuie actualizate prin interogarea câmpului rezultate_negate din cadrul structurii ELEMENT_ML. Vom șterge fiecare rezultat de joncțiune; când ștergem o potrivire dintr-un nod negativ și

memoria asociată potrivirii ajunge la zero elemente, atunci vom informa toți copiii nodului negativ despre acest lucru.

```

procedura ȘTERGE_ELEMENT_ML(e:ELEMENT_ML)
    pentru fiecare alfa_mem din e.alfa_memorii
        șterge e din lista alfa_mem.elemente
    sfârșit pentru fiecare
    cât timp e.potriviri conține elemente
        ȘTERGE_POTRIVIRE(prima potrivire din e.potriviri)
    sfârșit cât timp
    pentru fiecare rez_negat din e.rezultate_negate
        șterge rez_negat din rez_negat.pot.rezultate_negate
        dacă rez_negat.pot.rezultate_negate=NULL atunci
            pentru fiecare copil din rez_negat.pot.nod.copii
                MB_ACTIVARE_STÂNGA(copil, rez_negat.pot, NULL)
            sfârșit pentru fiecare
        sfârșit dacă
    sfârșit pentru fiecare
sfârșit

```

Trebuie să modificăm din nou procedura ȘTERGE_POTRIVIRE pentru a trata corect cazul nodurilor negative:

```

procedura ȘTERGE_POTRIVIRE(p:POTRIVIRE)
    cât timp p.copii conține elemente
        ȘTERGE_POTRIVIRE(prima potrivire din p.copii)
    sfârșit cât timp
    șterge p din lista p.nod.elemente
    dacă p.element <> NULL atunci
        șterge p din lista p.element.potriviri
    sfârșit dacă
    șterge p din lista p.părinte.copii
    dacă p.nod este NOD_NEGATIV atunci
        pentru fiecare rez_negat din p.rezultate_negate
            șterge rez_negat din
                rez_negat.element.rezultate_negate
            dealocă_memorie(rez_negat)
        sfârșit pentru fiecare
    sfârșit dacă
    dealocă_memorie(p)
sfârșit

```

2.7 Conjunții negate

Această secțiune va prezenta modul de implementare a condițiilor conjugate negate (CCN), care testează absența unei combinații de elemente ML. (CCN cu un singur element sunt echivalente semantic cu condițiile negate prezentate în secțiunea anterioară). Teza originală (Forgy, 1979), care prezenta algoritmul Rete nu tratează decât teoretic problema CCN, iar implementarea în pseudocod a fost introdusă de (Doorenbos, 1995). Această primă implementare va fi prezentată în continuare.

Pentru a exemplifica CCN vom adăuga un câmp suplimentar pentru fiecare cub, câmp numit **culoare**, care va reține culoarea cubului. Așadar, modelul cub devine:

```
Cub
- Latură [numeric]
- Înălțime [numeric]
- VecinStânga [Cub]
- VecinDreapta [Cub]
- VecinJos [Cub]
- VecinSus [Cub]
- Culoare [Culoare]
```

Având noul model vom crea o producție din P'_1 care va fi conține o CCN.

$$P''_1 : \text{DACĂ } (\langle \text{cub1} \rangle \text{ VecinStânga} = \langle \text{cub2} \rangle) \quad [C_1]$$

$$(\langle \text{cub1} \rangle \text{ VecinSus} = \langle \text{cub3} \rangle) \quad [C_2]$$

$$-\{ (\langle \text{cub3} \rangle \text{ VecinStânga} = \langle \text{cub4} \rangle) \quad [C_3]$$

$$(\langle \text{cub4} \rangle \text{ Culoare MOV}) \} \quad [C_4]$$

Această producție este exact ca cea din figura 2.5 cu diferență că al patrulea cub poate fi de orice culoare mai puțin mov.

Vom numi conjuncțiile dintr-o CCN “subcondiții”. Nu există restricții despre poziția sau tipul (pozitiv/negativ) subcondițiilor. De asemenea, subcondițiile pot fi imbricate până la orice adâncime. Important, posibilitatea de a imbrica conjuncții negate ne permite să creăm condiții cu număr arbitrar de cuantificatori \exists și \forall : informal, condițiile pozitive au semantica $\exists xP(x)$, pe când $\forall xP(x)$ poate fi rescris folosind negația conjunctivă ca $\neg \exists x \neg P(x)$. Ex[2]: “Fiecare cub roșu are un cub albastru deasupra lui” poate fi rescrisă ca “Nu există nici un cub roșu care nu are un cub albastru deasupra lui”.

Idea care stă la baza implementării CCN este o generalizare a negației prezentate în secțiunea anterioară. În cazul negațiilor individuale, pentru

fiecare potrivire realizăm joncțiunea ca și când nu am fi avut negație, salvăm rezultatul într-o memorie locală și trimitem mai departe în rețea doar potrivirile cu memorie locală goală. Vom folosi aceeași metodă și la CNN cu diferența că în locul unui singur nod negativ vom avea o subrețea care calculează joncțiunea subcondițiilor (figura 2.7). Un nod CCN este format dintr-un copil al nodului de joncțiune ce precede CCN și un nod partener care este cel mai de jos nod din subrețea.

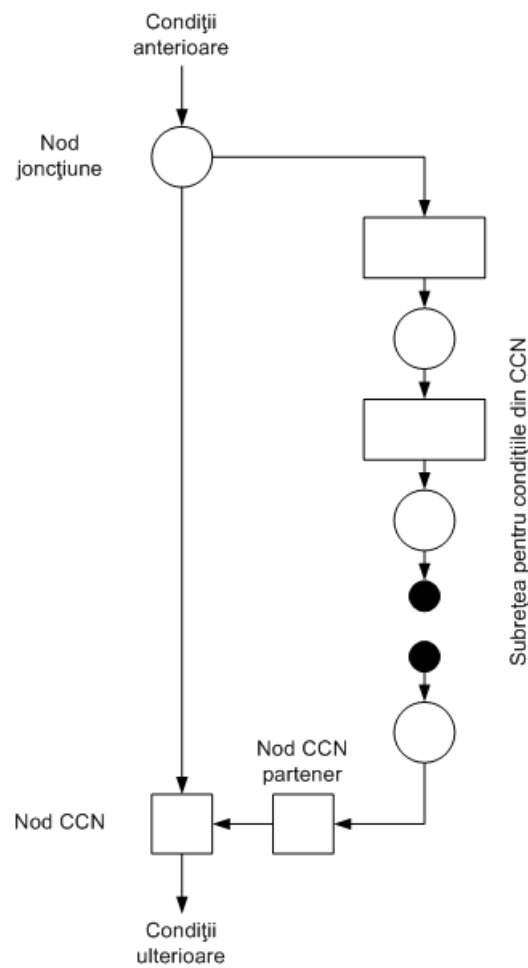


Figura 2.7: Rețeaua Rete pentru CCN

Folosim două noduri (CCN și CCN partener) deoarece în acea zonă a rețelei trebuie să primim activări din două surse: nodul de joncțiune care precede CCN și nodul de jos al rețelei. În ambele cazuri, activarea va

fi făcută prin activare din stânga și trebuie să tratăm aceste activări în mod diferit. O altă posibilitate ar fi să folosim un singur nod CCN care să suporte activare din stânga și din dreapta însă acest lucru ar însemna că nodurile de joncțiune reține două liste separate de copii (o listă pentru activări din stânga și una pentru activări din dreapta). Această ultimă soluție ar necesita mai mult spațiu de memorie. O a treia abordare ar fi ca procedura de activare să recunoască sursa activării și să efectueze activare la stânga sau la dreapta. Această metoda face codul mai complex și ceva mai încet[2].

Trebuie remarcat că atunci când avem CCN, partea beta a rețelei Rete nu mai este un arbore așa cum s-a menționat anterior. Vom presupune pentru moment că ordonăm joncțiunile precedente în așa fel încât nodul CCN să fie înaintea nodului cel mai de sus al subrețelei. Nodul CCN este activat din stânga atunci când o nouă potrivire pentru condițiile anterioare este găsită; el reține această potrivire ca o potrivire nouă și inițializează memoria locală a potrivirii. Noua potrivire este trimisă copiilor nodului (deoarece memoria de rezultate este vidă în acest moment). Nodul partener CCN servește colectării de potriviri din subcondiții; de fiecare dată când o potrivire este găsită, nodul partener CCN este activat. Scopul lui este de a adăuga noua potrivire în memoria locală specifică potrivirii din nodul CCN. Cum determinăm care potrivire din nodul CCN este cea “potrivită”? Considerând producția anterioară, o potrivire pentru ea are forma (e_1, e_2, e_3, e_4) unde e_1 și e_2 se potrivesc primelor două condiții, iar e_3 și e_4 se potrivesc condițiilor din CCN. Nodul partener CCN scoate ultimele j elemente ale ML din noile potriviri ale subcondițiilor, unde j este numărul de conjuncții din CCN, și găsește potrivirile corespunzătoare în secvența de elemente ML rămasă. Memoria potrivirii găsite este actualizată și, dacă este nevoie, copiii nodului CCN sunt notificați.

Un nod CCN reține o listă a potrivirilor și o referință către nodul părinte:

```
structura NOD_CCN
  potriviri: listă de POTRIVIRE
  partener: NOD_RETE
sfârșit
```

Vom modifica structura POTRIVIRE pentru a include două câmpuri suplimentare. REZULTATE_CCN va fi memoria locală pentru o potrivire dintr-un nod CCN. Din moment ce o potrivire poate fi acum un rezultat local, trebuie să adăugăm un câmp `pot` care are același rol cu câmpul `pot` din structura REZULTAT_JONCȚIUNE_NEGATIVĂ:

```

structura POTRIVIRE
  părinte: legătură către o POTRIVIRE de mai sus în arbore
  element: ELEMENT_ML
  nod: NOD_RETE
  copii: listă de POTRIVIRE
  rezultate_negate: listă de REZULTAT_JONCȚIUNE_NEGATIVĂ
  rezultate_ccn: listă de POTRIVIRE
  pot: POTRIVIRE
sfârșit

```

Un nod partener CCN reține o referință către nodul CCN de care este legat, plus un contor al numărului de conjuncții din CCN. De asemenea, vom mai adăuga un câmp `tampon_rezultate`, care este o memorie temporară folosită între momentul când subrețeaua a fost activată cu o nouă potrivire și momentul în care nodul CCN este activat de acea potrivire.

```

structura NOD_CCN_PARTENER
  nod_ccn: NOD_CCN
  număr_conjuncții: întreg
  tampon_rezultate: listă de POTRIVIRE
sfârșit

```

Procedura pentru activarea din stânga a nodului CCN este similară cu cea a activării unui nod negativ. În ambele cazuri, trebuie să găsim rezultatele joncțiunii pentru o nouă potrivire. Pentru nodurile negative, realizăm joncțiunea prin interogarea memoriei alfa și realizarea testelor pe ea. Pentru nodurile CCN, rezultatele joncțiunii au fost deja realizate de către subrețea, așadar ne vom uita în lista `tampon_rezultate` din nodul partener după ele.

```

procedura NOD_CCN_ACTIVARE_STÂNGA(nod:NOD_CCN,
                                   p:POTRIVIRE,
                                   e:ELEMENT_ML)
  pot=CREAZĂ_POTRIVIRE(nod,p,e)
  inserează pot în nod.potriviri
  pot.tampon_rezultate=NULL
  pentru fiecare rezultat din nod.partener.tampon_rezultate
    șterge rezultat din nod.partener.tampon_rezultate
    inserează rezultat în pot.tampon_rezultate
    rezultat.potrivire=pot
  sfârșit pentru fiecare

```

```

dacă pot.tampon_rezultate e goală atunci
    pentru fiecare copil din nod.copii
        MB_ACTIVARE_STÂNGA(copil, rez_negat.pot, NULL)
    sfârșit pentru fiecare
sfârșit dacă
sfârșit

```

Pentru a putea activa (din stânga) nodul partener CCN, luăm noua potrivire din subrețea și construim o potrivire “rezultat” pentru a o reține. După aceea, încercăm să găsim potrivirea corespunzătoare din memoria nodului CCN. Dacă găsim un corespondent, atunci adăugăm noul rezultat în memoria potrivirii; dacă numărul de rezultate din memorie s-a schimbat de la zero la unu, situație ce indică faptul că CCN a fost adevărată anterior și acum este falsă, vom apela procedura de ștergere a potrivirilor din nodurile de mai jos din rețea. Dacă în schimb nu avem nici o potrivire corespondentă atunci vom pune potrivirea în memoria de rezultate tampon.

Vom modifica procedura ȘTERGE_POTRIVIRI pentru a trata corect cazurile nodurilor CCN și partener CCN. Trei aspecte vor trebui luate în considerare. În primul rând, când o potrivire dintr-un nod CCN este ștearsă, trebuie să “curățăm” și memoria locală asociată ei prin ștergerea tuturor potrivirilor din ea. În al doilea rând, când o potrivire dintr-un nod partener CCN este ștearsă, în loc să o ștergem dintr-o listă `nod.potriviri`, o vom șterge din `nod.tampon_rezultate`. În cel de-al treilea rând, când ștergem, dacă numărul de potriviri se schimbă de la unu la zero vom informa toți copiii nodului CCN.

```

procedura NOD_P_CCN_ACTIVARE_STÂNGA(partener:NOD_RETE,
                                     p:POTRIVIRE,
                                     e:ELEMENT_ML)

    nod_ccn=partener.nod_ccn
    rezultat=CREAZĂ_POTRIVIRE(partener,p,e)
    pot_asociată=p
    elem_asociat=e
    pentru i=1 la partener.număr_conjuncții
        elem_asociat=pot_asociată.element
        pot_asociată=pot_asociată.părinte
    sfârșit pentru
    dacă (există potrivire în nod_ccn.pot unde
        potrivire.pot=pot_asociată și
        potrivire.element=elem_asociat) atunci

```

```

    inserează rezultat în potrivire.rezultate_ccn
    rezultat.pot=potrivire
    ȘTEREGE_POTRIVIRE(potrivire)
altfel
    inserează rezultat în partener.tampon_rezultate
sfârșit dacă
sfârșit

```

Trebuie remarcat faptul că o singură condiție negativă este un caz particular de CCN în care subrețeaua conține o singură condiție. Condițiile negative ar fi putut fi implementate folosind aceleași proceduri ca la CCN însă acest lucru ar necesita mai mult spațiu și timp deoarece cazul general necesită mai multe noduri și activări. Exceptând cazul în care condițiile negative singulare sunt extrem de rare, merită să le tratăm separat.

2.8 Optimizări ale algoritmului Rete

Trei algoritmi de potrivire au fost dezvoltați ca alternative ale Rete: Treat (Miranker, 1990), Match Box (Perlin și Debaud, 1989) și Tree (Bouaud, 1993).

Există o serie de (posibile) îmbunătățiri ale algoritmului Rete, care au fost inventate în timp. Le vom menționa în continuare fără a detalia prea mult:

- *Modificare “in place”*¹: extinde implementarea procedurilor Rete de interpretare pentru a realiza modificările elementelor ML direct. Pseudocodul prezentat în această lucrare necesită ca modificările să fie făcute indirect, prin eliminarea elementului original urmat de adăugarea elementului modificat.
- *“Scaffolding”*²: este o metoda utilă atunci când elementele ML sunt adăugate și eliminate repetat din memoria de lucru. În loc să se realizeze ștergerea, potrivirile sunt doar marcate ca inactive iar reactivarea lor durează mult mai puțin decât recreerea.
- O altă metodă este implementarea memoriei alfa ca un arbore de decizie³. Acesta funcționează asemănător cu rețeaua de date cu indexare

¹Schor et al, 1986

²Perlin, 1990a

³Ghallab, 1981; Nishiyama, 1991

discutată în secțiunea 2.2.2, dar are o complexitate mult mai bună de timp: este garantat că rulează în timp linear cu adâncimea arborelui. Dezavantajul este că adăugarea și eliminarea producțiilor în timpul rulării este mult mai complicată și arborele de decizie poate necesita spațiu exponențial în cazul cel mai defavorabil.

- În sistemele în care se folosește o metodă de rezolvare a conflictelor⁴, procedurile de potrivire pot fi îmbunătățite prin încorporarea unor cunoștințe despre tipuri particulare de conflicte și rezolvări ale lor. O strategie de rezolvare a conflictelor este o modalitate de a selecta o singură potrivire completă a unei producții în locul tuturor potrivirilor.
- În sistemele cu memorie de lucru (ML) foarte mare, costul operațiilor de joncțiune individuale poate deveni o problemă din cauza produsului cartezian generat. “Collection Rete”⁵ este o modalitate de a reduce costurile prin structurarea conținutului memoriei beta într-un set de colecții de potriviri în locul potrivirilor individuale.
- Un alt mod de a reduce costurile operațiilor de joncțiune este adăugarea consistenței arcurilor factorizate⁶. Aceasta adaugă un algoritm de consistență a arcurilor pentru a elimina multe combinații de elemente ale ML și potriviri fără a le mai testa individual. Deși este o decizie costisitoare s-a dovedit că, în practică, beneficiile sunt mai mari decât costurile.
- O altă metodă de a evita operațiile de joncțiune costisitoare este restricționarea conținutului ML folosind atribute unice⁷. Algoritmul Rete poate fi făcut să profite de această specializare, rezultând “Uni-Rete”⁸. Uni-Rete este cu siguranță mai rapid decât implementare Rete clasică dar nu poate fi folosit în scopuri general ci doar pe anumite clase de probleme.
- În algoritmul Rete clasic, nodurile rețelei beta pentru o producție sunt liniare. Cercetătorii⁹ au investigat utilizarea unor topologii neliniare.

⁴McDermott și Forgy, 1978

⁵Acharya și Tambe, 1992

⁶Perlin, 1992

⁷Tambe et al., 1990

⁸Tambe et al., 1992

⁹Schor et al., 1986; Scales, 1986; Gupta, 1987; Ishida, 1988; Tambe et al., 1991

Din nefericire, nu se cunoaște un algoritm eficient pentru a găsi cea mai bună topologie pentru o anumită producție (diferite producțiile se comportă diferit folosind diferite topologii), iar utilizarea unei topologii neliniare poate duce la rezultate foarte defavorabile. Tehnicile de găsim eficientă a unei topologii rămân domeniu de studiu pentru cercetarea curentă.

Capitolul 3

O Implementare Modernă

În acest capitol vom prezenta o implementare inedită a unui sistem expert folosind tehnologii moderne (cloud computing și software as a service). Implementarea s-a făcut pe platforma de cloud computing, Azure, oferită de către Microsoft.

3.1 De ce cloud computing?

Cloud Computing-ul nu este un concept nou, a fost inventat acum mulți ani, însă în ultima perioadă a revenit în atenția dezvoltatorilor din cauza beneficiilor pe care le poate avea pentru o afacere. Cloud computing presupune existența unui set de resurse scalabile, adesea virtualizate, oferite sub formă de serviciu.

Conceptul este o combinație a altor concepte apărute în trecut:

- Infrastructură ca serviciu (eng. Infrastructure as a Service - IaaS)
- Platformă ca serviciu (eng. Platform as a Service - PaaS)
- Software ca serviciu (eng. Software as a Service - SaaS)

Termenul “nor” (eng. cloud) este folosit ca o metaforă pentru Internet, bazându-se pe diagramele care îl reprezintă ca o rețea de calculatoare.

Unul din principalele motive pentru care companiile recurg la furnizorii de servicii de cloud computing este elasticitatea resurselor. Putem asemăna un serviciu de cloud computing (adesea numit “utility computing”) cu o utilitate publică precum curentul electric. Dacă o persoană consumă mult curent atunci va plăti mult, pe când dacă nu consumă deloc nu va plăti

nimic. În același mod, serviciile de cloud se plătesc în funcție de utilizare și permit, prin intermediul unor API-uri, fără intervenție umană, scalarea pe verticală/orizontală a resurselor alocate. Acest lucru este benefic în special pentru afaceri care sunt la început. Un business online, aflat la început, evoluează precum se vede în figura 3.1; la început numărul de accesări (cereri) este într-o continuă creștere, apoi există un punct de apogeu iar în final, inevitabil, există un segment de recesiune. Vom prezenta acțiunile investitorului în fiecare din cele trei situații atât din punct de vedere al soluției, clasice, on premise, cât și a utility computing-ului.

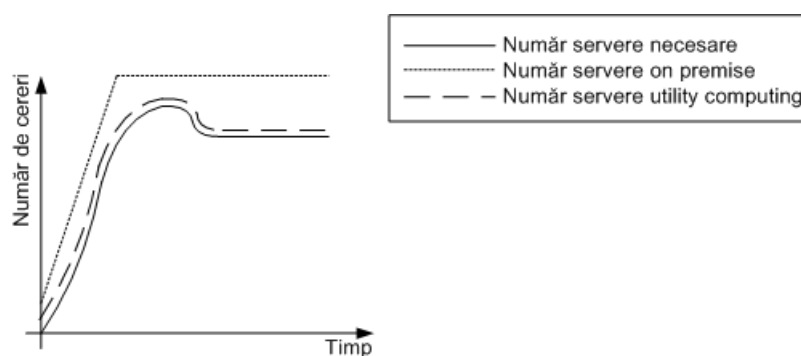


Figura 3.1: Evoluția (felicită) a unui startup

Acțiunile pe care le întreprinde un investitor având o soluție on premise:

1. *Perioada 1 - Creșterea continuă a numărului de cereri*
Investește în servere pentru a putea servi clienții cu un timp cât mai bun de răspuns. În general se cumpără mai multă putere de procesare pentru a putea acomoda orice exces de trafic.
2. *Perioada 2 - Apogeul*
Investitorul, fiind copleșit de perioada anterioară investește în continuare și cumpără, pe baza unor predicții, mai multă putere de procesare. O parte din capacitatea de calcul deja nu este folosită.
3. *Perioada 3 - Recesiunea*
Investitorul realizează că afacerea nu duce în sus și are un ușor trend negativ. Oprește sau reduce investițiile, dar capacitatea de calcul nefolosită este în creștere; de obicei nu o reduce pentru că se așteaptă la o revenire. Timpul trece, echipamentele se învechesc, iar când investitorul se hotărăște să le vândă nu își poate acoperi pierderile pentru

că în tot acest timp mașinile au creat cheltuieli (de întreținere, de rulare, etc.).

Acțiunile pe care le întreprinde un investitor având o soluție in the cloud:

1. *Perioada 1 - Creșterea continuă a numărului de cereri*
Realizează aplicația să scaleze în sus resursele astfel încât să acomodeze foarte puțin trafic peste numărul actual de cereri. Costurile cresc o dată cu profitul.
2. *Perioada 2 - Apogeul*
Aplicația se oprește din scalat. Costurile devin constante.
3. *Perioada 3 - Recesiunea*
Aplicația scalează în jos pentru nu acomoda decât puțin peste numărul curent de cereri. Profitul scade dar o dată cu aceasta și costurile.

Se poate observa ușor că o soluție cu resurse scalabile este mult mai avantajoasă, mai ales pentru afaceri aflate la început deoarece lipsa profitului/succesului nu duce la creșterea cheltuielilor; am putea zice chiar că succesul este direct proporțional cu cheltuielile. Un alt avantaj al norului este lipsa necesității de personal care să administreze infrastructura din partea investitorului. Provider-ul de cloud asigură toate cele necesare, el se ocupă de reparații, mentenanță, etc.

Totuși, ca orice lucru aparent ideal, cloud computing-ul are și dezavantaje. Cel mai mare dezavantaj este lipsa controlului. Asupra platformei pe care rulează aplicațiile din nor nu se prea poate interveni, nu se pot stabili ce actualizări să se instaleze, ce alte aplicații suplimentare să ruleze sau cum este configurată mașina (virtuală). Lipsa controlului asupra mașinii duce la lipsa controlului asupra securității, ceea ce pentru unele corporații este inadmisibil. Un alt dezavantaj este dependența de conexiune la internet; pe când o soluție on premise poate rula în interiorul companiei chiar dacă este pierdută legătura cu lumea exterioră, aplicațiile din nor pot rula doar cu conexiune permanentă. O combinație de soluție in the cloud și on premise o prezintă conceptul de Software + Services¹, adică software și servicii, concept ce desemnează o suită de aplicații offline ce pot rula la capacitate

¹Subiectul Software + Service nu este acoperit de această lucrare însă, pentru mai multe detalii, se pot vizita adresele: <http://victorhurduagaci.com/software-services-ama-p1/> și http://en.wikipedia.org/wiki/Software_plus_services

limitată într-un scenariu deconectat și la capacitate maximă atunci când pot ajunge la serviciu.

Dacă totuși aceste probleme pot fi ocolite iar politica companiei nu impune vreo restricție de rulare on premise atunci o soluție din nor este o investiție foarte bună.

3.2 Windows Azure

Azure Services Platform² este platforma “din nor” oferită de către Microsoft care oferă o gamă largă de servicii de Internet ce pot fi consumate atât din medii on-premise cât și online.

Azure este o platformă de aplicații ce permite ținerea și rularea aplicațiilor în centrele de date Microsoft. Toate serviciile Azure, inclusiv cele dezvoltate de terțe părți, rulează pe sistemul de operare Windows Azure iar figura 3.2 prezintă cele două straturi ale norului având la bază sistemul de operare pe care rulează o serie de servicii predefinite.

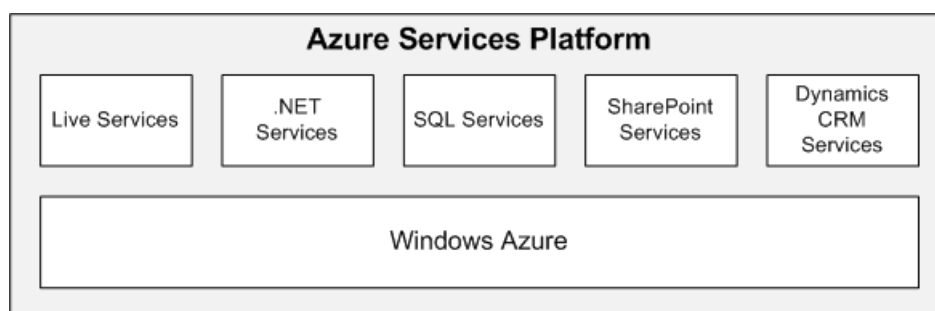


Figura 3.2: Azure Services Platform

Windows Azure este sistemul de operare strict destinat norului Microsoft și nu va fi făcut public pentru a se crea nori on premise.

Orice aplicație care rulează pe Windows Azure poate fi compusă din două componente, fiecare scalabilă individual și/sau un serviciu de stocare. Fiecare componentă este reprezentată de o instanță a unei mașini virtuale rulând într-un centru de date.

Web Role este prima componentă scalabilă a aplicațiilor Azure. Este singurul punct unde o aplicație poate primi cereri (HTTP sau HTTPS) din exterior. De fapt, este o mașină virtuală cu Windows Server 2008 și IIS pe

²<http://www.microsoft.com/azure/>

care rulează o aplicație web ASP.NET și/sau un serviciu Windows Communication Foundation (WCF). Web Role-ul poate crea conexiuni HTTP(S) / TCP sper exterior.

Worker Role este o componentă, a unei aplicații scalabile ce rulează în nor, care nu poate primi cereri din afară. Poate fi asemănat cu un fir de execuție ce rulează în background, fir de execuție ce procesează informațiile consumatoare de timp. Și Worker Role-ul poate crea conexiuni HTTP(S)/TCP sper exterior.

Trebuie menționat că cele două componente prezentate anterior sunt fără stare, adică nimic din ce se stochează pe ele nu se păstrează iar două apeluri consecutive, de la aceeași sursă pot ajunge la instanțe diferite. Pentru a persista date se folosesc serviciile de stocare.

Serviciile de stocare reprezintă singurul mod în care informația poate fi păstrată în nor. Toate modalitățile de stocare sunt scalabile, scalarea realizându-se transparent pentru utilizator, durabile și disponibile. Orice informație stocată în nor este replicată în trei exemplare astfel încât atunci când una din sursele de date este afectată există alte două de unde se pot prelua datele. Disponibilitatea este asigurată de faptul că datele sunt stocate tot în nor, sub formă de serviciu.

Există trei forme în care se pot stoca informațiile:

- *Cozi de mesaje*

Cozile de mesaj sunt structuri de date care funcționează pe principiul FIFO³ și care rețin date ce trebuie procesate în ordinea în care au ajuns. Totodată, cozile de mesaje rețin informații de dimensiune mică (max 8K). Deși sunt scalabile și pot fi accesate de mai multe componente (Web sau Worker) simultan, este garantat că un mesaj va fi preluat de o instanță a unei componente.

- *Tabele*

Tabele sunt colecții structurate de date. Trebuie reținut că tabelele din serviciul de stocare Azure *NU* sunt relaționale sau tranzacționale. Ele sunt doar niște colecții de rânduri optimizate pentru miliarde de înregistrări. Tabelele pot fi partiționate pentru a optimiza performanțele.

- *Blob-uri*

Blob-urile sunt informații de dimensiune mare (fișiere video, imagini, etc.). Ele pot fi asemănat cu fișierele din sistemul local de fișiere iar

³FIFO = First In First Out

organizarea lor se face pe bază de cataloage care sunt echivalente cu directoarele.

Întreg serviciul de stocare poate fi folosit independent de Web Role și Worker Role și atât timp cât nu este privat poate fi accesat de oricine prin niște interfețe REST⁴.

Posibilitatea de folosi cele trei mari elemente ale Windows Azure în mod independent creează posibilități incredibile pentru Software + Services. De exemplu, putem avea aplicații care își expun datele printr-un serviciu de cloud (Web Role), date care sunt aduse de către Worker Role dintr-un centru de date on premise.

3.3 CloudRuleBasedSystem (CRBS)

Sau sistemul bazat pe reguli ce rulează în nor. Așa se numește aplicația dezvoltată pentru exemplificarea celor prezentate în această teză. Este vorba despre un sistem expert scalabil, oferit sub formă de serviciu, rulând în nor, al cărui proces de inferență are la bază algoritmul Rete.

3.3.1 De ce serviciu (scalabil)?

În urma unor discuții avute cu niște experți a reieșit că aceștia nu vor să își ofere cunoștințele. Ei preferă să le țină pentru ei și să le ofere sub formă de consultanță. Un sistem expert care rulează în nor nu îi va permite utilizatorului accesul la baza de cunoștințe și la procesul de inferență ci doar la rezultate. În acest fel, expertul poate lipsi atunci când se oferă consultanța digitală dar totuși deține control asupra ceea ce îi este oferit utilizatorului.

Un alt motiv pentru care am ales centralizarea datelor este reprezentat de schimbările regulilor. Dacă fiecare aplicație client ar fi avut regulile în ea atunci ar fi fost foarte greu, dacă nu chiar imposibil, să se sincronizeze schimbările cu baza de cunoștințe centrală. Faptul că doi clienți folosesc reguli diferite poate fi dezastruos deoarece pe același set de fapte ei vor obține rezultate diferite, ceea ce duce la dezinformare iar scopul unui sistem expert este complet opus dezinformării.

Ultimul, dar nu cel din urmă motiv pentru am considerat că sistemul va funcționa mai bine dacă este oferit ca serviciu (scalabil) îl reprezintă

⁴REST = REpresentational State Transfer (http://en.wikipedia.org/wiki/Representational_State_Transfer)

numărul de cereri care nu poate fi prevăzut a priori. Folosind o soluție care se scalează automat, putem regla resursele alocate și implicit costurile pentru ele astfel încât să fie la pragul cel mai mic admisibil.

3.3.2 Arhitectura

CRBS este format din două componente mari: clientul și serviciul.

Clientul este o aplicație ce poate fi dezvoltată de furnizorul serviciului sau de terțe părți. Rolul clientului este de a facilita introducerea datelor și prezentarea rezultatelor într-un mod specific problemei. Pentru lumea cuburilor prezentată la pagina 12 clientul permite introducerea datelor despre cuburi prin selecție din liste iar afișarea este grafică. Pentru un sistem expert despre legislație s-ar putea introduce niște informații despre situație în mod text iar rezultatele să fie tot text. Trebuie menționat că orice instanță a serviciului poate avea mai mulți clienți. Se conectează la serviciul de inferență prin HTTP și trebuie să fie făcut în concordanță cu modelele de pe server.

Serviciul este nucleul sistemului expert. Aici avem baza de cunoștințe, motorul de inferență, baza de date a utilizatorilor, sistemul de permisiuni și uneltele de administrare. De asemenea toată procesarea se realizează aici, scutind clientul de muncă suplimentară. Funcționalitatea pentru client este expusă printr-un serviciu WCF iar uneltele de administrare sunt accesibile printr-o interfață web ASP.NET.

După cum am menționat anterior Web Role-ul și Worker-ul sunt componente Azure scalabile care nu comunică în mod direct. În postura web role-ului avem interfața web și interfața WCF iar worker-ul este reprezentat de motorul de inferență. S-a ales o astfel de separare deoarece procesul de inferență durează destul de mult iar numărul de accesări nu este foarte mare. În acest context, putem avea un număr mic de roluri web și un număr mult mai mare de worker-i astfel încât fiecare cerere să fie preluată de un worker și să se dea un răspuns rapid.

Modul cum interacționează clientul cu serviciul dar și componentele interne ale serviciului sunt prezentate în figura 3.3. Se poate observa că unicul mod de comunicare dintre instanțele interfeței web și ale motorului de inferență este serviciul de stocare. Deși nu este reprezentat în figură, fiecare interfață web poate avea mai multe instanțe; același lucru este valabil și pentru motorul de inferență.

Se poate observa că, în cazul cozii de mesaje, avem o comunicare unidirecțională (de la interfața web, la motorul de inferență), asincronă. Cititorul

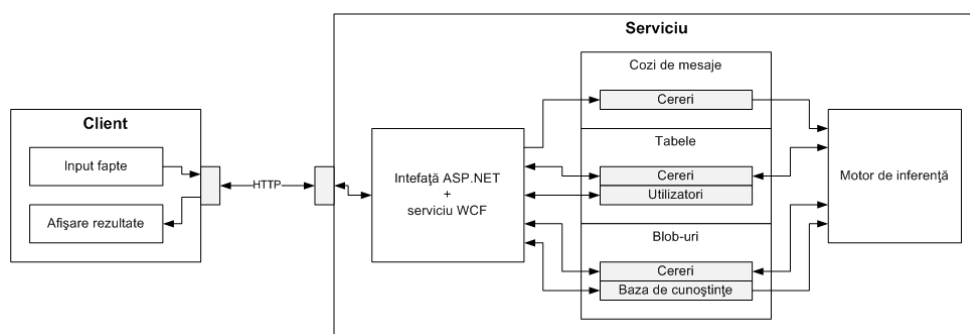


Figura 3.3: Componentele sistemului și modul cum interacționează ele

atent a sesizat până în acest moment că este vorba despre un șablon arhitectural extrem de cunoscut dacă numim interfață web “producător” și motorul de inferență, “consumator”. Este vorba de șablonul (design pattern-ul) producător-consumator cu producători și consumatori multipli; acesta este principalul model arhitectural de comunicare dintre componentele Windows Azure.

Mai trebuie remarcat că motorul de inferență nu alterează în nici un fel baza de cunoștințe și nici nu are acces la tabela de utilizatori. Aceste două constrângeri sunt naturale și sunt impuse deoarece s-a adoptat principiul privilegiului minim.

Atunci când o nouă cerere de inferență este primită prin serviciul web, instanța Web Role-ului care a primit-o face următoarele, în ordine (se consideră că nu se trece la un pas ulterior dacă pasul anterior nu a fost finalizat cu succes):

1. Verifică credențialele (codul de securitate al cererii; va fi discutat în secțiunea 3.3.3)
2. Verifică dacă versiunea bazei de cunoștințe căreia se adresează cererea este pe server
3. Salvează detaliile cererii (faptele) în blob-uri
4. Salvează detaliile cererii (identitatea celui care a făcut cererea, data, versiunea de cunoștințe căreia se adresează, numele blob-ului, statusul cererii ca “neprocesată”) în tabela de cereri
5. Pune un nou mesaj în coada de mesaje conținând identificatorul liniei din tabelă unde se află detaliile

Cât timp o instanță a motorului de inferență lucrează, nu este disponibilă pentru a prelua noi cereri. Atunci când devine disponibilă, ea realizează următoarele (considerând că există mesaje în coada ce cereri):

1. Ia primul mesaj din coada de cereri
2. Extrage identitatea cererii și o marchează în tabelă ca “în curs de procesare” astfel încât dacă se întâmplă ca procesarea să dureze mai mult de 30 de secunde și altă instanță preia mesajul să îl arunce deoarece altcineva se ocupă de cererea asociată
3. Preia blob-ul asociat din serviciul de stocare și extrage versiunea bazei de cunoștințe (doar numărul)
4. Crează o nouă instanță a unui proces de inferență folosind versiunea extrasă anterior
5. Trimite spre procesare informațiile din blob și așteaptă confirmarea de finalizare
6. Scrie în tabelă rezultatul procesării, marchează cererea ca “Finalizată” și, dacă nu au fost erori, scrie rezultatul într-un blob

3.3.3 Securitate

Ca orice sistem care expune informații în Internet, CRBS are mai multe facilități de securizare și protejare a datelor. În primul rând, prin intermediul serviciului web nu este permisă alterarea datelor bazei de cunoștințe, ci se permite doar cerearea de procesări noi; acest lucru împiedică “scurgerea” de informații private; tot în acest scop, spre client, nu se transmit niciodată reguli sau informații privind procesul de inferență. Serviciul web, așadar, oferă doar metode pentru crearea unei cereri noi, interogarea pentru rezultate și anularea unei cereri.

Am menționat în secțiunea anterioară despre codul de securitate care este trimis de către client la server. Acest cod este un GUID⁵ unic pentru fiecare utilizator în parte. Este nevoie ca el să fie transmis la fiecare apel al serviciului web pentru a verifica identitatea celui care apelează. Am recurs la această metodă și nu la transmiterea credențialelor (nume de utilizator și parolă) pentru a nu permite unui atacator să afle cine este posesorul unui

⁵Globally Unique Identifier http://en.wikipedia.org/wiki/Globally_Unique_Identifier

cod de securitate dar și pentru a-i anula orice posibilitate de revenire la schimbarea codului. Dacă s-ar fi ales soluția cu transmiterea credențialelor atunci, chiar și la schimbarea parolei, atacatorul tot ar fi avut numele de utilizator.

Fie u_1, u_2, \dots, u_n utilizatorii sistemului expert. Pentru fiecare utilizator $u_i; i = 1, \dots, n$ există asociat un cod unic de securitate $s_i; i = 1, \dots, n$. Nu există nici o relație, în afara celei de asociere între u_i și s_i iar două coduri de securitate alăturate s_i și s_{i+1} nu se află într-o relație de ordine. GUID-urile de securitate sunt generate și distribuite aleator astfel încât generarea unuia nou să nu poată fi prezisă pe baza celui anterior.

În orice moment, un utilizator poate merge pe pagina sa de administrare pentru a reseta codul. O dată ce un cod a fost resetat, toți clienții care îl folosesc trebuie să schimbe manual vechiul cod. Noul cod nu se poate obține prin serviciul web și cum un atacator nu poate accesa pagina de administrare deoarece nu poate asocia codul de securitate cu credențiale rezultă că nu va putea să mai acceseze sistemul.

Posibilitatea de a ataca serviciul prin forță brută (generarea de GUID-uri până când se găsește unul potrivit) este în zadar deoarece există nu mai puțin de 2^{128} sau 3.4×10^{38} GUID-uri posibile iar probabilitatea de a nimeri un GUID potrivit chiar și pentru un număr destul de mare de utilizatori este extrem de mică. Pentru ca cititorul să își facă o idee despre cât de mare este numărul de GUID-uri posibile s-a arătat că pentru fiecare din cele 5×10^{22} stele ale universului observabil putem alocă nu mai puțin de 6.8×10^{15} GUID-uri unice.

Fiecare cerere de procesare are un identificator unic tot de tip GUID. Pentru obținerea rezultatelor procesării unei astfel de cereri, cel care apelează serviciul trebuie să prezinte atât identificatorul unic al cererii cât și codul de securitate al utilizatorului care a inițiat cererea inițială. Este imposibil a se obține rezultatele unei procesări prin atacuri de tip forță brută deoarece în acest caz avem nu mai puțin de $2^{128} \times 2^{128}$ combinații posibile.

Accesarea paginii de administrare se realizează pe bază de nume de utilizator și parolă. Există trei tipuri de utilizatori, fiecare având drepturile prezentate în tabela 3.1.

În tabela cu utilizatori din serviciul de stocare Azure nu se rețin parolele ci doar un hash al lor generat folosind un algoritm de clasă SHA-2 ce generează un mesaj de pe 256 biți, SHA256. Atunci când un utilizator încearcă autentificarea, se generează folosind același algoritm, un hash pentru parola introdusă și se compară cu cel din baza de date. În acest fel parola originală nu poate fi obținută deoarece algoritmi de hash, per se, nu

Tabela 3.1: Tipuri de utilizatori în CRBS (Utilizator = U; Inginer cunoștințe = IC; Administrator = A)

Drepturi	U	IC	A
Apelare webservice	x	x	x
Accesare interfață web	x	x	x
Resetare parolă/cod securitate propriu	x	x	x
Vizualizare istoric cereri proprii	x	x	x
Modificare bază de cunoștințe		x	x
Creare de utilizatori			x
Resetare parolă/cod securitate ale altui utilizator			x
Vizualizare istoric cereri ale altui utilizator			x
Anulare cereri ale altui utilizator			x

permit obținerea mesajului original din textul rezultat. Singura modalitate de a “sparge” parola unui utilizator este prin forță brută. Pentru algoritmul SHA256 nu se cunoaște, încă, un mod de a genera coliziuni.

3.3.4 Ce aduce nou?

CRBS se dorește a fi un sistem expert adaptabil oricărei probleme. Mai mult, oricine ar trebui să poată crea o bază de cunoștințe (BC) folosind interfața de administrare⁶.

Bazându-se pe reguli care sunt interpretate din fișiere XML, BC a CRBS poate fi schimbată oricând, cu efort minim chiar folosind un editor de texte. Regulile și modelele sunt stocate sub formă arborescentă și pot fi interpretate chiar și de alte aplicații. Orice schimbare a BC nu necesită compilare/restartare. Pe server pot fi ținute mai multe versiuni ale BC astfel încât să se păstreze compatibilitatea cu vechii clienți dar și cei noi să poată beneficia de regulile noi.

În viitor, interfața pentru introducerea regulilor va fi una intuitivă, restrictivă la doar ceea ce se poate folosi într-un context dat (de exemplu va împiedica adunarea dintre un întreg și un Cub) tocmai pentru a reduce numărul de greșeli pe care le-ar putea face un utilizator. De asemenea, se dorește ca introducerea regulilor să fie atât text pentru utilizatori avansați cât și în mod grafic pentru începători.

⁶În momentul scrierii acestei lucrări, interfața de introducere a regulilor era în curs de dezvoltare

Capitolul 4

Pseudocodul final

```
structura NOD_TC
  TESTEAZĂ: ‘‘Atribut’’, ‘‘Model’’, ‘‘Valoare’’ sau ‘‘Nimic’’
  REZULTAT_AȘTEPTAT: <atribut>, <model> sau <valoare>
  COPII: listă de noduri sau NULL
  MEMORIE_ALFA: referință la MA sau NULL
sfârșit
```

```
structura MEMORIE_ALFA
  elemente: listă de ELEMENT_ML
  succesori: listă de NODURI_JOIN
sfârșit
```

```
structura NOD_RETE
  tip: ‘‘memorie-beta’’, ‘‘nod-joncțiune’’, ‘‘nod-productie’’
  copii: listă de NOD_RETE
  părinte: NOD_RETE
  ...
sfârșit
```

```
structura MEMORIE_BETA
  potriviri: listă de POTRIVIRE
sfârșit
```

```
structura NOD_JONCȚIUNE
  mem_alfa: MEMORIE_ALFA
  teste: listă de TEST_JONCȚIUNE
```

sfârșit

structura TEST_JONȚIUNE

câmp1: ‘obiect’, ‘atribut’ sau ‘valoare’

câmp2: ‘obiect’, ‘atribut’ sau ‘valoare’

locație_câmp_2: întreg

sfârșit

structura TEST_JONȚIUNE_ULTIMA_CONDIȚIE

câmp1: ‘obiect’

câmp2: ‘valoare’

locație_câmp_2: 2

sfârșit

structura REZULTAT_JONȚIUNE_NEGATIVĂ

pot: POTRIVIRE

element: ELEMENT_ML

sfârșit

structura ELEMENT_ML

câmpuri: șir de simboluri

memorii_alfa: listă de MEMORIE_ALFA

potriviri: listă de POTRIVIRE

rezultate_negate: listă de REZULTAT_JONȚIUNE_NEGATIVĂ

sfârșit

structura NOD_CCN

potriviri: listă de POTRIVIRE

partener: NOD_RETE

sfârșit

structura POTRIVIRE

părinte: legătură către o POTRIVIRE de mai sus în arbore

element: ELEMENT_ML

nod: NOD_RETE

copii: listă de POTRIVIRE

rezultate_negate: listă de REZULTAT_JONȚIUNE_NEGATIVĂ

rezultate_ccn: listă de POTRIVIRE

pot: POTRIVIRE

sfârșit

```
structura NOD_CCN_PARTENER
  nod_ccn: NOD_CCN
  număr_conjuncții: întreg
  tampon_rezultate: listă de POTRIVIRE
sfârșit
```

```
procedura ADAUGĂ_ELEMENT_ML(elem:ELEMENT_ML)
  ACTIVEZĂ_NODUL(nod_părinte, elem)
sfârșit
```

```
procedura ACTIVEZĂ_NODUL(nod:NOD_TC,
                          elem:ELEMENT_ML)
  rezultat = nod.REZULTAT_AȘTEPTAT
  câmp = nod.TESTEZĂ
  dacă (CÂMP(câmp, elem) == rezultat) sau
    (nod.TESTEAZĂ == 'Nimic') atunci
    dacă număr(nod.COPII) <> 0 atunci
      pentru fiecare nod_copil din nod.COPII
        ACTIVEZĂ_NODUL(nod_copil, elem)
      sfârșit pentru fiecare
    altfel
      ACTIVEAZĂ_MA(elem)
    sfârșit dacă
  altfel
    returnează NEPOTRIVIRE
  sfârșit dacă
sfârșit
```

```
procedura CÂMP(câmp:NOD_TC.TESTEZĂ,
               elem_al_ml:ELEMENT_ML)
  selectează (câmp)
  cazul 'Atribut':
    returnează elem_al_ml.ATRIBUT
  cazul 'Model':
    returnează elem_al_ml.MODEL
  cazul 'Valoare':
    returnează elem_al_ml.VALOARE
```



```

pentru fiecare elem_ml din nod.mem_alfa.elemente
  dacă E_POTRIVIRE(nod.teste, p, elem_ml) atunci
    pentru fiecare copil din nod.copii
      MB_ACTIVARE_STÂNGA(copil, p, elem_ml)
    sfârșit pentru fiecare
  sfârșit dacă
sfârșit pentru fiecare
sfârșit

```

```

procedura E_POTRIVIRE(teste:listă de TEST_JONȚIUNE,
                    p: POTRIVIRE,
                    e: ELEMENT_ML)
  pentru fiecare test din teste
    arg1=e[test.câmp1]
    condiție=selectează condiția a [test.locație_câmp_2]-ia
    arg2=condiție[test.câmp2]
    dacă arg1<>arg2 atunci
      returnează FALS;
    sfârșit dacă
  sfârșit pentru fiecare
  returnează ADEVĂRAT
sfârșit

```

```

procedura ACTIVARE_MA(a: MEMORIE_ALFA,
                    e:ELEMENT_ML)
  inserează e în a.elemente
  inserează a în e.memorii_alfa
  pentru fiecare copil din a.sucesori
    ACTIVARE_DREAPTA_NOD_JONȚIUNE(copil, e)
  sfârșit pentru fiecare
sfârșit

```

```

procedura NOD_NEGATIV_ACTIVARE_STÂNGA(nod:NOD_NEGATIV,
                                       p:POTRIVIRE,
                                       e:ELEMENT_ML)
  potrivire_nouă=alocă_memorie(POTRIVIRE)
  inserează potrivire_nouă în nod.potriviri
  potrivire_nouă.rezultate_jonțione=NULL
  pentru fiecare element din nod.alfa_mem.elemente

```

```

dacă E_POTRIVIRE(nod.teste,
                 potrivire_nouă,
                 element) atunci
    rezultat=alocă_memorie(REZULTAT_JONCTIUNE_NEGATIVĂ)
    rezultat.pot=alocă_memorie(POTRIVIRE)
    rezultat.element=e
    inserează rezultat în potrivire_nouă.rezultate_negate
    inserează rezultat în e.rezultate_negate
sfârșit dacă
sfârșit pentru fiecare
dacă potrivire_nouă.rezultate nu conține elemente atunci
    pentru fiecare copil din nod.copii
        MB_ACTIVARE_STÂNGA(copil, potrivire_nouă, NULL)
    sfârșit pentru fiecare
sfârșit dacă
sfârșit

procedura CREAZĂ_POTRIVIRE(nod:NOD_RETE,
                           părinte:POTRIVIRE,
                           e:ELEMENT_ML)

    pot=alocă_memorie(POTRIVIRE)
    pot.părinte=părinte
    pot.element=e
    pot.nod=nod
    pot.copii=NULL
    inserează pot în părinte.copii
    dacă e <> NULL atunci
        inserează pot în e.potriviri
    sfârșit dacă
    returnează pot
sfârșit

procedura NOD_NEGATIV_ACTIVARE_DREAPTA(nod:NOD_NEGATIV,
                                         e:ELEMENT_ML)

    pentru fiecare p din nod.potriviri
        dacă E_POTRIVIRE(nod.teste, p, e) atunci
            dacă p.rezultate_negate<>NULL atunci
                ȘTERGE_DESCENDENȚI_POTRIVIRE(p)
            sfârșit dacă

```

```

rezultat=alocă_memorie(REZULTAT_JONCTIUNE_NEGATIVĂ)
rezultat.pot=p
rezultat.element=e
inserează rezultat în p.rezultate_negate
inserează rezultat în e.rezultate_negate
sfârșit dacă
sfârșit pentru fiecare
sfârșit

procedura ȘTERGE_DESCENDENȚI_POTRIVIRE(p:POTRIVIRE)
cât timp p.copii conține elemente
    ȘTERGE_DESCENDENȚI_POTRIVIRE(primul din p.potriviri)
sfârșit cât timp
sfârșit

procedura ȘTERGE_ELEMENT_ML(e:ELEMENT_ML)
pentru fiecare alfa_mem din e.alfa_memorii
    șterge e din lista alfa_mem.elemente
sfârșit pentru fiecare
cât timp e.potriviri conține elemente
    ȘTERGE_POTRIVIRE(prima potrivire din e.potriviri)
sfârșit cât timp
pentru fiecare rez_negat din e.rezultate_negate
    șterge rez_negat din rez_negat.pot.rezultate_negate
    dacă rez_negat.pot.rezultate_negate=NULL atunci
        pentru fiecare copil din rez_negat.pot.nod.copii
            MB_ACTIVARE_STÂNGA(copil, rez_negat.pot, NULL)
        sfârșit pentru fiecare
    sfârșit dacă
sfârșit pentru fiecare
sfârșit

procedura ȘTERGE_POTRIVIRE(p:POTRIVIRE)
cât timp p.copii conține elemente
    ȘTERGE_POTRIVIRE(prima potrivire din p.copii)
sfârșit cât timp
șterge p din lista p.nod.elemente
dacă p.element <> NULL atunci
    șterge p din lista p.element.potriviri

```

```

sfârșit dacă
șterge p din lista p.părinte.copii
dacă p.nod este NOD_NEGATIV atunci
    pentru fiecare rez_negat din p.rezultate_negate
        șterge rez_negat din
            rez_negat.element.rezultate_negate
        dealocă_memorie(rez_negat)
    sfârșit pentru fiecare
sfârșit dacă
dealocă_memorie(p)
sfârșit

procedura NOD_CCN_ACTIVARE_STÂNGA(nod:NOD_CCN,
                                   p:POTRIVIRE,
                                   e:ELEMENT_ML)

    pot=CREAZĂ_POTRIVIRE(nod,p,e)
    inserează pot în nod.potriviri
    pot.tampon_rezultate=NULL
    pentru fiecare rezultat din nod.partener.tampon_rezultate
        șterge rezultat din nod.partener.tampon_rezultate
        inserează rezultat în pot.tampon_rezultate
        rezultat.potrivire=pot
    sfârșit pentru fiecare
    dacă pot.tampon_rezultate e goală atunci
        pentru fiecare copil din nod.copii
            MB_ACTIVARE_STÂNGA(copil, rez_negat.pot, NULL)
        sfârșit pentru fiecare
    sfârșit dacă
sfârșit

procedura NOD_P_CCN_ACTIVARE_STÂNGA(partener:NOD_RETE,
                                       p:POTRIVIRE,
                                       e:ELEMENT_ML)

    nod_ccn=partener.nod_ccn
    rezultat=CREAZĂ_POTRIVIRE(partener,p,e)
    pot_asociată=p
    elem_asociat=e
    pentru i=1 la partener.număr_conjuncții
        elem_asociat=pot_asociată.element

```

```
    pot_asociată=pot_asociată.părinte
sfârșit pentru
dacă (există potrivire în nod_ccn.pot unde
    potrivire.pot=pot_asociată și
    potrivire.element=elem_asociat) atunci
    inserează rezultat în potrivire.rezultate_ccn
    rezultat.pot=potrivire
    ȘTEREGE_POTRIVIRE(potrivire)
altfel
    inserează rezultat în partener.tampon_rezultate
sfârșit dacă
sfârșit
```


Bibliografie

- [1] Barachini F. (1991), *The evolution of PAMELA*
- [2] Doorenbos R. B. (1995), *Production Matching for Large Learning Systems*, Carnegie Mellon University
- [3] Forgy C. L. (1979), *On the Efficient Implementation of Production Systems*, Carnegie Mellon University
- [4] Gupta A., Tambe M., Kalp D., Forgy C. L. și Newell, A. (1988), *Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis*, International Journal of Parallel Programming
- [5] Krishnamoorthy C. S. și Rajeev S. (1996), *Artificial Intelligence and Expert Systems for Engineers*, CRC Press
- [6] Norvig P., Russel S. (2003). *Artificial Intelligence - A Modern Approach Second Edition*, Prentice Hall
- [7] Perlin M. (1990b), *Topologically traversing the Rete network*, Taylor & Francis, Inc
- [8] Scales D. J., (1986), *Efficient matching algorithms for the Soar/Ops5 production system*, Stanford University.