SISTEM BAZAT PE CUNOŞTINŢE PENTRU INGINERIA INVERSĂ A PROGRAMELOR

dr.ing. Ştefan Trăuşan-Matu

Institutul de Cercetări in Informatică

Rezumat: Lucrarea prezintă un sistem bazat pe cunoştinţe, care poate analiza lexical, sintactic şi semantic şi apoi, pe baza unei baze de cunoştinţe de programare, poate abstractiza un program FORTRAN. Abstractizarea este efectuată cu un analizor de gramatici programate de grafuri şi are drept scop o înţelegere a funcţionalităţii programului analizat. Descrierea de nivel înalt obţinută în urma abstractizării, poate fi utilizată atît pentru validarea programului analizat, cît şi pentru rescrierea programului în alte limbaje cum ar fi C. Prelucrările bazate pe cunoştinţe sunt facilitate de utilizarea mediului de programare orientată spre obiecte XRL, dezvoltat în Common Lisp.

Cuvinte cheie: Ingineria programării, ingineria inversă a programelor, sisteme bazate pe cunoştinţe, reutilizarea programelor, înţelegerea programelor.

1. Introducere

Sintagma de inginerie inversă îşi trage numele de la faptul că, în astfel de sisteme, se parcurg în sens invers fazele dezvoltării unui program, în scopul obţinerii unei descrieri de nivel înalt (la limita o specificaţie a programului respectiv) plecînd de la codul sursă al programului considerat.Lucrarea de faţă prezintă un prototip de sistem de inginerie inversă a programelor FORTRAN [39] Acest sistem a fost experimentat pe mai multe exemple de traducere a unor programe din limbajul FORTRAN în limbajul C.

Interesul în ingineria inversă a programelor este justificat în primul rînd de faptul că, prin această modalitate se pot reutiliza volumele mari de programe scrise în limbaje cum ar fi FORTRAN sau COBOL prin rescriere în alte limbaje. În plus faţă de o simplă rescriere, prin inginerie inversă reutilizarea se poate face, nu numai la nivelul codului, ci la un nivel mai înalt, la care se pot face restructurări complexe sau extinderi ale programului analizat. De exemplu, în [23],[45] se prezintă cum se pot face structurări complexe ale datelor unui program, conform paradigmei programării orientate spre obiecte. Descrierea de nivel înalt, obţinută în urma ingineriei inverse a unui program poate fi folosită, nu numai pentru reutilizare, ci şi ca o anexa în dezvoltarea programului respectiv, pentru verificări, validări şi modificări.

Un sistem inteligent de inginerie inversă include mecanisme de înţelegere a programelor, care necesită tehnici de reprezentarea şi prelucrarea cunoştinţelor. Pe de altă parte, înţelegerea programelor poate fi considerată şi modalitate foarte utilă de achiziţie de cunoştinţe de programare. Aceste consideraţii au stat la baza şi a abordării prezentate în lucrarea de faţă în care sistemul de inginerie inversă realizat este dezvoltat în mediul de reprezentare şi prelucrare a cunoştinţelor XRL, o componentă esenţială a sistemului fiind o bază de cunoştinţe de programare. Această soluţie permite evoluţia sistemului prin adăugarea de noi concepte, de noi abstractizări (adică noi cunoştinţe) sau prin restructurarea celor existente. Acest caracter deschis al sistemului de inginerie inversă se incadrează în concepţia care stă la baza temei de faţă: sprijinirea reutilizării şi evoluţiei programelor, plecînd de la tehnicile de inginerie a cunoştinţelor.

Lucrarea de faţă este structurată, în continuare, după cum urmează: în capitolul al doilea, este făcută o scurtă prezentare a mediului XRL, care a constituit substratul în care a fost dezvoltat sistemul de inginerie inversă. În capitolul al treilea, este prezentat cadrul (dezvoltat în XRL) destinat sprijinirii dezvoltării şi facilitării evoluţiei sistemului de inginerie inversă. În capitolul al patrulea, este prezentat sistemul de inginerie inversă a programelor FORTRAN. Lucrarea este incheiată de un capitol de concluzii.

2. Mediul de reprezentare şi prelucrare a cunoştinţelor XRL

Sistemele bazate pe cunoştinţe sunt programe care rezolvă eficient probleme complexe prin acordarea unei atenţii deosebite reprezentării explicite şi prelucrării prin tehnici specifice a cunoştinţelor implicate în rezolvarea problemei respective. Cele mai puternice medii de programare, destinate dezvoltării de aplicaţii bazate pe cunoştinţe, integrează mai multe modalităţi de reprezentare şi prelucrare a cunoştinţelor, reprezentarea prin obiecte structurate avînd un rol central. În cadrul abordării din această lucrare s-a utilizat mediul de programare orientată spre obiecte structurate XRL [4], [5], [6], [7], [8], [9], [37], [39], dezvoltat în Lisp de un colectiv din care a făcut parte şi autorul prezentei lucrări. Acest mediu oferă multe din serviciile puse la dispoziţie de unul din cele mai puternice medii bazate pe cunoştinţe, destinat dezvoltării programelor, şi anume: sistemul Cake [30]. Sistemul XRL are o arhitectură stratificată (conform figurii 1) şi oferă (printre altele) posibilitatea reprezentării cunoştinţelor prin obiecte structurate, reguli de producţie şi restricţii. Totodată, el pune la dispoziţie mai multe mecanisme de inferenţă pentru rezolvarea de probleme (rafinare concurentă a obiectelor structurate, propagare de restricţii, înlănţuire înainte a regulilor de producţie).

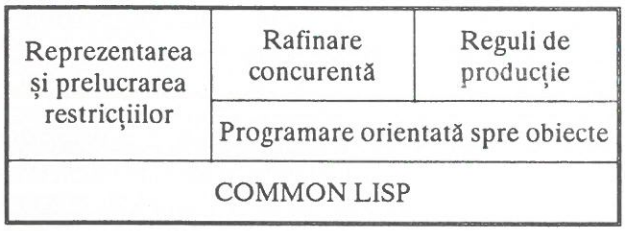

Figura 1. Mediul XRL

2.1 Limbajul de obiecte din XRL

În XRL obiectele pot fi definite ca unități (folosind funcția unit) sau prin copiere (clonare) a unei alte unități (folosind funcția a). În XRL nu există distincția clasa-instanță, existentă în multe limbaje de programare, orientate spre obiecte. Renunțarea la această distincție s-a făcut datorită cerinței de a putea rafina dinamic, de un număr nedefinit de ori, un obiect (ceea ce ar însemna de a considera o instanță a unei clase ca o clasă și de a-i genera dinamic instanțe care, la rindul lor, să fie considerate clase și să li se genereze instanțe, ș.a.m.d. - ceea ce nu este posibil în limbajele bazate pe clase și instanțe). Un obiect are un număr de componente (.sloturi), poate moșteni proprietăți de la un număr de alte obiecte și poate răspunde unui număr de mesaje (care corespund într-o oarecare măsură apelului unei proceduri). Un exemplu de definire a unui obiect care descrie un dreptunghi este dat mai jos:

```
(unit dreptunghi
self (a obiect_grafic
      desenează desen_dreptunghi
      șterge șterge_dreptunghi
      deplasează deplasează_dreptunghi)
x 0
y 0
l nil
h nil)
```

Un dreptunghi particular este declarat după cum urmează:

```
(setq d122 (a dreptunghi
             x 11
             y 15
             l 10
             h 20))
```

Accesarea, respectiv setarea unei componente se face cu:

```
(fslot 'x d122)
 respectiv
(pslot 'x d122 20)
```

În limbajele de programare orientată spre obiecte, prelucrările specifice unui anumit obiect sunt grupate în așa numitele metode. Aceste metode sunt executate ca urmare a trimiterii unui mesaj obiectului respectiv. De exemplu, metoda prin care se realizează deplasarea unui dreptunghi va fi:

```
(defmethod deplaseaza_dreptunghi (selector self
x_nou y_nou)
    (pslot 'x self x_nou)
    (pslot 'y self y_nou))
```

Trimiterea unui mesaj se face prin:

```
(msg 'deplaseaza d122 25 65)
```

Demonii sunt o tehnică de programare larg utilizată în inteligența artificială. Ei reprezintă proceduri activate prin date, respectiv proceduri care sunt executate atunci cînd are loc un anumit tip de operație asupra datelor (fără a fi apelate explicit de utilizator). În XRL există posibilitatea atașării de demoni care intră în acțiune la crearea unui obiect sau slot (așa numiții demoni after- create) și demoni care sunt activați la actualizarea unui slot (demoni after-write). Demonii sunt definiți în XRL ca funcții LISP.

2.2 Reguli de producție în XRL

Regulile de producție sunt probabil cea mai utilizată modalitate de reprezentare a cunoștințelor. În mediul XRL se pot defini reguli de producție (cu înlănțuire înainte), acestea putînd fi grupate în așa numitele interpretoare de reguli. Atît regulile, cît și interpretoarele sunt reprezentate prin obiecte structurate.

Alegerea unei reprezentări bazată pe obiecte pentru reguli are, în primul rind, avantajul obținerii unei uniformități de reprezentare. Aceasta permite, de exemplu, ca obiectele de tip reguli sau sisteme de reguli să poată fi combinate cu alte tipuri de obiecte. În al doilea rînd, există posibilitatea moștenirii de componente de clauze, în acest mod încurajîndu-se crearea de taxonomii de reguli cu efect benefic în structurarea expertizei domeniului avut în vedere.

3. Cadru extensibil de reprezentare și prelucrare a cunoștințelor de programare

3.1 Reprezentarea cunoștințelor de programare

În scopul utilizării tehnicilor de inteligență artificială, care sprijină reutilizarea si evoluția programelor, a fost dezvoltat un cadru extensibil de reprezentare și prelucrare a cunoștințelor de programare. Analizîndu-se clasele de cunoștințe de programare s-a decis ca, în cadrul realizat, să fie incluse facilități de reprezentare și de prelucrare a trei tipuri de cunoștințe: concepte aflate într-o taxonomie pe baza

relației de moștenire, reguli de transformare a unor descrieri în alte descrieri și euristici. Pentru reprezentarea și prelucrarea primului și celui de-al treilea tip de cunoștințe sunt folosite limbajul de obiecte, respectiv, modulul de reguli de producție din XRL. Pentru cel de-al doilea tip de cunoștințe a fost implementat un analizor bazat pe o gramatică programată de grafuri.

Mecanismele de reprezentare și de prelucrare, împreună cu o bază de cunoștințe de programare dezvoltată folosind aceste mecanisme, constituie cadrul destinat sprijinirii activităților tuturor etapelor ciclului de viață al programelor. Acest cadru de reprezentare a cunoștințelor este extensibil, introducerea de noi cunoștințe putînd fi făcută foarte ușor.

3.2 Baza de cunoștințe de concepte de programare

Reprezentarea folosind obiecte structurate este utilizată pentru a descrie un program ca o rețea de obiecte la diverse niveluri de abstractizare (similar și cu calculul cu planuri din Programmer's Apprentice [30], [43]). În acest scop, a fost creată o taxonomie de obiecte structurate, care pot fi utilizate pentru înțelegerea, descrierea și prelucrarea programelor.

Astfel de taxonomii de concepte de programare pot fi considerate ontologii ale domeniului programării. De asemenea, ele pot fi considerate și biblioteci de componente reutilizabile. De fapt, chiar în mediile comerciale de programare au început să apară astfel de biblioteci organizate ca taxonomii de obiecte (de exemplu, TurboVision pentru TurboPascal sau Classlib pentru Borland C++).

În figura 2 este ilustrată o parte din taxonomia de concepte folosite în cadrul sistemului de inginerie inversă. Baza de cunoștințe de programare are drept rădăcină obiectul generic progrConcept. Acest obiect are următorii descendenți direcți (care la rîndul lor, după cum se va discuta mai jos, sunt la baza unor întregi taxonomii de obiecte):

- DATADESCR - obiect generic de la care moștenesc toate obiectele care se referă la date;
- PRGND - obiect generic de la care moștenesc toate obiectele care sunt instrucțiuni sau abstractizări de control;
- FUNCT - obiect generic de la care moștenesc toate obiectele care sunt funcții sau abstractizări procedurale.

Cunoștințele referitoare la structurile de date formează o latice extensibilă de obiecte structurate care corespund tipurilor de date elementare și evoluate. Relația de ordine în latice este relația de moștenire a obiectelor și reprezintă conceptual operații de rafinare/abstractizare. Obiectele care descriu date au mai multe sloturi (componente), unele prezente în toate clasele de obiecte de acest tip, altele specifice unor anumite subclase.

Similar cu baza de cunoștințe referitoare la structurile de date, operațiile și structurile de control sunt reprezentate tot prin obiecte aflate în relații de moștenire, care reflectă relația de abstractizare. Toate obiectele care descriu operații (instrucțiuni) și abstractizări de control moștenesc de la obiectul prgnd.

Obiectele care reprezintă instrucțiuni și abstractizări de control au fost grupate în 6 categorii după cum urmează:

1) instrucțiuni de transfer al controlului, instrucțiuni de intrare/ieșire sau instrucțiuni auxiliare;
2) instrucțiuni de test;
3) grupa instrucțiunilor de atribuire: aceste instrucțiuni au fost grupate într-o taxonomie deoarece, în urma experienței avute, în operațiile de abstractizare (de exemplu, în abstractizarea unei stive sau în abstractizarea ciclurilor) se fac raționamente specifice în funcție de tipul atribuirilor;
4) grupa abstractizărilor de control;
5) grupa obiectelor referitoare la proceduri: PROCDESCR și CALL;
6) obiectele care descriu operații pentru tipuri de date abstracte (s-a reprezentat doar cazul stivei): ISTACK, PUSH, POP.

3.3 Reprezentarea transformărilor de abstractizare si rafinare

O a doua modalitate de reprezentare a cunoștințelor de programare, pusă la dispoziție de cadrul destinat sprijinirii activităților de programare este cea a transformărilor de abstractizare și rafinare. Procesul de abstractizare constă în recunoașterea unor configurații particulare în graful care descrie modelul curent al programului și înlocuirea acestora cu un nod care reprezintă abstractizarea respectivă (pentru rafinare procedîndu-se invers, respectiv înlocuindu-se un nod cu un subgraf). Procesul de abstractizare sau rafinare repetată a modelului unui program poate fi realizat prin intermediul unui sistem de reguli de producție sau prin tehnici de analiză, bazate pe gramatici atribuitate și programate de grafuri [15], [18] în care fiecare transformare este o producție în gramatică.

Din motive de eficiență s-a ales reprezentarea transformărilor de abstractizare sau rafinare prin gramatici programate de grafuri. În cele ce urmează este dat pe scurt formalismul de la care s-a plecat în implementarea unui analizor pentru gramatici programate de grafuri. Acest formalism este un subset al gramaticilor definite în [15].

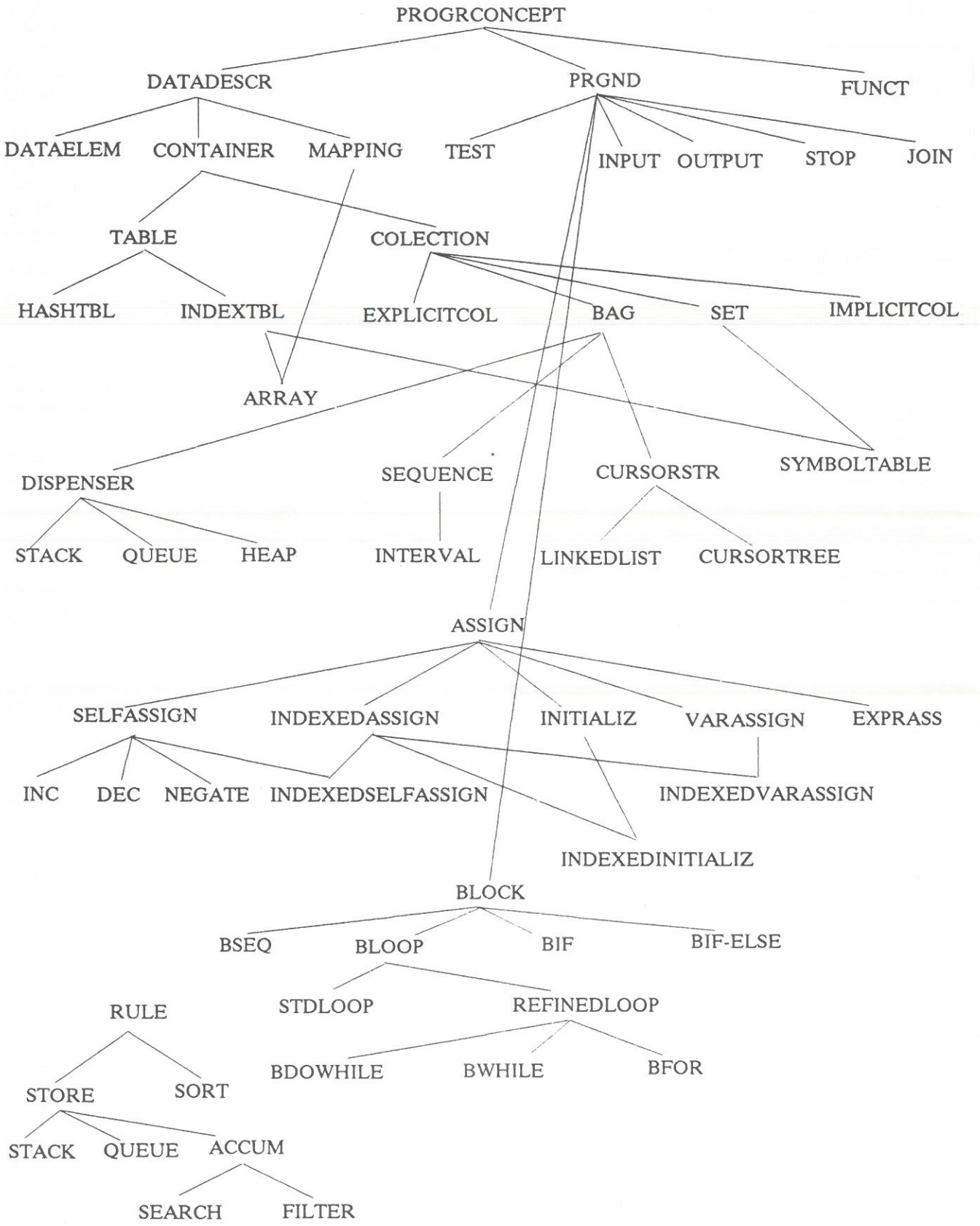

Figura 2. Un fragment din taxonomia de obiecte din baza de cunoștințe de programare

O gramatică programată de grafuri este un cvintuplu g= <V,W,P,S,C>, unde:

1) V si W sunt alfabete pentru etichetarea nodurilor, respectiv arcelor grafului;
2) P este o mulțime finită de producții;
3) S este o mulțime de grafuri inițiale;
4) C este o diagramă de control peste P.

O producție pr este un cuadruplu pr=<gst,n,T,P> în care:

1) gst este un graf care reprezintă partea stîngă a producției;
2) n este un nod care reprezintă cu ce este înlocuită partea stîngă;
3) T={stw,drw| w e W} este transformarea legăturilor;
4) P este predicatul de aplicabilitate al producției.

O diagramă de control peste o mulțime finită de producții, P, este un graf cu mulțimea P U {f} ca etichete de noduri și {da,nu} ca etichete de arce. În plus:

1) unul din nodurile P este declarat ca nod de start;
2) există un unic nod de stop, etichetat cu f, din care nu pleacă nici un arc.

În figura următoare este ilustrată o producție într-o gramatică programată de grafuri.

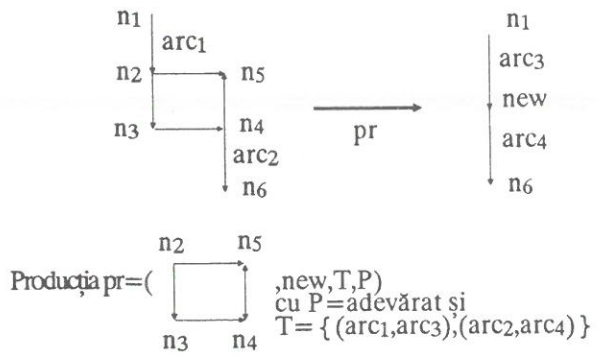

Figura 3. O producție în gramatica programată de grafuri

3.4 Reprezentarea și prelucrarea cunoștințelor de programare folosind reguli de producție

A treia modalitate de reprezentare a cunoștințelor, pusă la dispoziție în sistemul-cadru de sprijinire a activităților de dezvoltare a programelor, prezentat în acest capitol sunt regulile de producție. Prin ele nu se reprezintă explicit concepte, structuri sau taxonomii de concepte, ci elemente procedurale care pot fi însă cuplate cu ierarhii de concepte sau rețele de obiecte structurate.

Ce este particular în abordarea de față este faptul că regulile sunt reprezentate prin obiecte structurate, ceea ce permite organizarea lor în taxonomii. În acest mod se poate spune, dintr- un punct de vedere, că modelul domeniului programării, realizat prin baza de cunoștințe de obiecte structurate este completat cu taxonomia de reguli, care poate fi considerată și ea o taxonomie (implicită) de concepte. De exemplu, în faza de analiză a vectorilor din sistemul de inginerie inversă, care va fi prezentat în capitolul următor, regulile folosite pentru analiza utilizărilor unui vector sunt în următoarea ierarhie de moștenire, fiecare regulă putînd fi considerată și o definiție (alternativă la cele existente) a conceptelor respective.

În urma experimentelor și a evoluției sistemului, taxonomia unei submulțimi de reguli poate sugera definirea unor noi obiecte generice. De exemplu, ca urmare a analizei vectorilor cu regulile aflate în taxonomia ce are ca rădăcină conceptul RULE din figura 2, se poate sugera generarea de noi descendenți ai obiectului generic vector.

4. Sistemul de inginerie inversă a programelor FORTRAN

În cadrul cercetărilor desfășurate în direcția aplicării tehnicilor de inginerie a cunoștințelor în reutilizarea și evoluția programelor, un loc central l-a ocupat elaborarea de sisteme de inginerie inversă a programelor [23],[42]. Sintagma de inginerie inversă își trage numele de la faptul că, în astfel de sisteme, se parcurg în sens invers fazele tipice ingineriei programelor, în scopul obținerii unei descrieri de nivel înalt plecînd de la codul sursă al unui program. Diverse studii și, bineînțeles, cererea de astfel de sisteme, au demonstrat că, în multe cazuri este mai eficient să se dezvolte un program nou plecînd de la un program existent prin inginerie inversă, decît să se dezvolte de la zero.

4.1 Schema bloc a sistemului de inginerie inversă a programelor

Ideea care stă la baza sistemului realizat de inginerie inversă a programelor este utilizarea combinată a tehnicilor de teoria compilării cu cele de inteligență artificială pentru abstractizarea cunoștințelor incluse într-un program dat, în scopul obținerii unei descrieri de nivel înalt a programului respectiv. Funcționarea sistemului constă în parcurgerea mai multor faze de prelucrare, conform schemei bloc din figura 4.

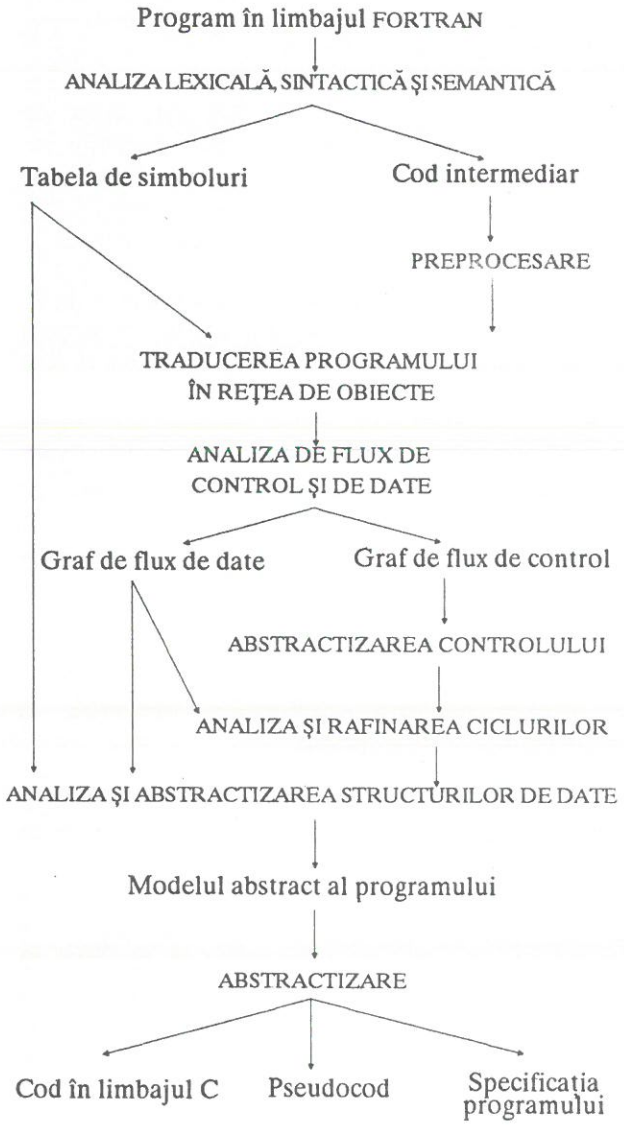

Figura 4. Schema bloc a sistemului de inginerie inversă a programelor

Odată construit un model al programului, se poate continua abstractizarea acestuia într-un ciclu în care se fac abstractizări succesive în scopul obținerii în final a unei descrieri a programului la un nivel cît mai înalt. De exemplu, după cum se va arăta în secțiunea 4.10, se pot abstractiza diverse structuri de date cum ar fi recunoașterea mecanismului LIFO și înlocuirea vectorului corespunzător cu o stivă.

În versiunea actuală, sistemul prezentat în lucrare este destinat analizării programelor scrise în limbajul FORTRAN. Utilizarea sistemului pentru un alt limbaj, de exemplu COBOL, implică rescrierea modulelor de analiză lexicală, sintactică și semantică și de preprocesare, celelalte module putînd fi refolosite.

4.2 Analizorul lexical, sintactic și semantic pentru limbajul FORTRAN

Prima fază a sistemului de inginerie inversă este similară cu faza de analiză dintr-un compilator uzual, scopul urmărit fiind obținerea unei descrieri în cod intermediar a programului analizat și a unei tabele de simboluri. Codul intermediar generat este o descriere în sintaxa Lisp (pentru a putea fi citit ușor de sistemul de înțelegere a programelor) și se oprește cu descompunerea la nivelul expresiilor aritmetice și logice.

Versiunea actuală a analizorului nu ia în considerare declarațiile de COMMON și EQUIVALENCE, de IMPLICIT, instrucțiunile ENCODE, DECODE, GOTO asignat sau calculat precum și instrucțiunile de lucru cu fișiere. Dezvoltarea unei noi versiuni a acestui analizor care să trateze și aceste instrucțiuni nu constituie însă o problemă (cu excepția lui ENCODE și DECODE care însă sunt mai puțin utilizate). Probleme dificile în înțelegerea programului au fost deja considerate. În figura 5. este dat un exemplu de program FORTRAN și instrucțiunile generate în cod intermediar.

Tabela de simboluri, generată pentru programul din figura 5. este:

```
((I %%VAR0 INTVAR 0)
 (S %%VAR1 REALVAR 0)
 (%%VAR2 %%VAR2 VAR 0)
 (%%VAR3 %%VAR3 VAR 0))
```

4.3 Preprocesarea

Preprocesarea are drept scop reducerea numărului de tipuri de construcții de control și de obținere, din codul intermediar, rezultat în urma analizei programului FORTRAN, a unei descrieri independente de un limbaj anume de programare. Prelucrările efectuate sunt:

- înlocuirea ciclurilor DO prin instrucțiuni de atribuire, incrementare a indexului și salt condiționat;
- înlocuirea instrucțiunilor IF aritmetic cu instrucțiuni standard de salt condiționat;
- optimizarea unor secvențe de salturi condiționate și necondiționate și a unor secvențe de etichete.

Tot în figura 5. este descrisă și forma intermediară pentru programul considerat.

Program FORTRAN	Forma intermediară	Forma preprocesată
I=1	EXPR %%VAR0 1)	(EXPR %%VAR0 1)
S=0	(EXPR %%VAR1 0)	(EXPR %%VAR1 0)
2 IF(I.GT.10)GOTO1	2	2
	(LEXPR %%VAR2	(IF (%%VAR0 ***GT*** 10) 1)
	%%VAR0 ***GT*** 10)	
	(IFNOT %%VAR2 -2)	
	(GOTO 1)	
	-2	
I=I+1	(EXPR %%VAR0 %%VAR0 + 1)	(EXPR %%VAR0 %%VAR0 + 1)
S=S+I*I	(EXPR %%VAR1 %%VAR1 +	(EXPR %%VAR1 %%VAR1 +
	%%VAR0 * %%VAR0)	%%VAR0 * %%VAR0)
WRITE(*,3)I	(OUTPUT %%VAR0)	(OUTPUT %%VAR0)
GOTO2	(GOTO 2)	(GOTO 2)
1 IF(S.GT.10)WRITE(*,3)S	1	1
	(LEXPR %%VAR3	
	%%VAR1 ***GT***)	
	(IFNOT %%VAR3 -3)	(IFNOT (%%VAR1 ***GT*** 10) -3)
	(OUTPUT %%VAR1)	(OUTPUT %%VAR1)
	-3	3
3 FORMAT(I2)	(INFO 3 FORMAT(I2))	(INFO 3 FORMAT(I2))
DO 4 I=1,10	(LOOP 4 %%VAR0 1 10 0)	(EXPR %%VAR0 1)
		1.1
		(CONTINUE)
4 WRITE(*,3)I	4	4
	(OUTPUT %%VAR0)	(OUTPUT %%VAR0)
		(EXPR %%VAR0 % 1)
		(IFNOT (%%VAR0 ***GT***10) 1.1)
STOP	(STOP)	(STOP)
END		

Figura 5. Corespondența program FORTRAN, forme intermediare

4.4 Traducerea programului în rețea de obiecte

Faza următoare în analiza unui program este generarea reprezentării interne a programului sub formă de rețea de obiecte XRL. Astfel, pentru fiecare variabilă se generează o copie (clonă) a unui obiect generic pentru date și pentru fiecare instrucțiune din codul intermediar preprocesat se generează o copie (clona) a unui obiect generic din baza de cunoștințe de obiecte, prezentată în capitolul anterior. În plus, față de instrucțiunile în cod intermediar, sunt generate și obiecte de tip JOIN pentru a indica punctele de confluență a mai multor ramuri în program. Obiectele generate pentru instrucțiuni sunt descrise în figura 6.

4.5 Construirea grafului de flux de control și de date

4.5.1 Inferarea fluxului de control și partiționarea în blocuri de bază

Fluxul de control indică posibilitățile de succesiune între diversele entități (instrucțiuni sau blocuri de bază). Fluxul de control este reprezentat printr-un graf în care nodurile sunt entități, iar arcele sunt posibilitățile de succedare sau precedare în decursul execuției programului a acestor entități.

Un bloc de bază este o secvență de instrucțiuni în care fluxul de control intră la prima instrucțiune și iese prin ultima, fără a ajunge la instrucțiuni stop sau return și fără ramificare în altă instrucțiune decît ultima [ASU86]. Partiționarea în blocuri de bază a codului obținut în urma analizei programului sursă este un punct de plecare pentru prelucrările ulterioare de construire a grafurilor de flux de control și de date precum și în diversele modalități de abstractizare.

Gruparea obiectelor generate pentru nodurile programului în blocuri de bază este ilustrată în figura 6. Fluxul de control între blocurile de bază este ilustrat în figura 7.

Forma preprocesată	Obiect generat	Bloc de bază
(EXPR %%VAR0 1)	INITIALIZ-82	BSEQ-850
(EXPR %%VAR1 0)	INITIALIZ-830	
2	JOIN-831	BSEQ-854
(IF (%%VAR0 ***GT*** 10) 1)	IF-832	
(EXPR %%VAR0 %%VAR0 + 1)	INC-833	BSEQ-857
(EXPR %%VAR1 %%VAR1 + %%VAR0 * %%VAR0)	SELFASSIGN-834	
(OUTPUT %%VAR0)	OUTPUT-835	
(GOTO 2)	GOTO-836	
1	JOIN-837	BSEQ-853
(IFNOT (%%VAR1 ***GT*** 10) -3)	IFNOT-838	
(OUTPUT %%VAR1)	OUTPUT-839	BSEQ-856
-3	JOIN-840	BSEQ-852
(INFO 3 format(i2))	INFO-841	
(EXPR %%VAR0 1)	INITIALIZ-842	
1.1	JOIN-843	BSEQ-851
(CONTINUE)	CONTINUE-844	
4	JOIN-845	
(OUTPUT %%VAR0)	OUTPUT-846	
(EXPR %%VAR0 %%VAR0 + 1)	INC-847	
(IFNOT (%%VAR0 ***GT*** 10) 1.1)	IFNOT-848	
(STOP)	STOP-849	BSEQ-855

Figura 6. Corespondența intre forma intermediară, obiecte și blocuri de bază

4.5.2 Numerotarea blocurilor de bază

Ordinea în care sunt tratate blocurile de bază în prelucrările descrise în secțiunile următoare poate avea implicații semnificative asupra timpului de prelucrare. În [1] se demonstrează că, cea mai bună numerotare este conform unei ordini în adîncime. Această ordine este inversul ordinii în care un nod a fost ultima oară vizitat într-o traversare în preordine a grafului de flux de control al blocurilor de bază. În abordarea de față a fost folosită această numerotare.

4.5.3 Detectarea ciclurilor

În cadrul sistemului dezvoltat au fost implementate două posibilități independente de detectare a ciclurilor. Prima este bazată pe găsirea arcelor retrograde (backedges), adică cele care merg înapoi în fluxul controlului. Acestea pot fi detectate plecînd de la proprietatea că virful acestor arce își domină coada. (Un nod d domină nodul n dacă orice cale de la nodul inițial la nodul n al grafului de flux de control trece prin d [1]).Fiecare arc retrograd este generator al unui ciclu denumit ciclu natural [1]. Un ciclu are următoarele proprietăți:

1. are un singur punct de intrare, care domină toate celelalte noduri ale ciclului;
2. există cel puțin o cale înapoi la începutul ciclului.

O a doua posibilitate de detectare a ciclurilor este prin analiza grafului de flux de control cu o gramatică specializată de grafuri (analiza descrisă în faza anterioară a temei de față). Trebuie însă remarcat faptul că pentru această a doua metodă există posibilitatea ca graful să nu poată fi total redus, caz în care graful trebuie modificat prin adăugarea unor noi arce.

4.5.4 Construirea grafului de flux de date

În efectuarea de abstractizări asupra datelor este esențială cunoașterea fluxului datelor (există un flux de date între două instrucțiuni dacă valoarea setată de prima instrucțiune este utilizată în a doua). Graful de flux de date generat are drept noduri instrucțiunile în cod intermediar, iar arcele vor fi de două tipuri: de intrare, grupate pe variabilele de intrare, și de ieșire pentru variabila definită de instrucțiune. Arcele de intrare arată cine contribuie la instrucțiunea respectivă, iar cele de ieșire arată unde va fi utilizată variabila definită.

Construirea grafului de flux de date se face incremental, pentru fiecare bloc de bază în parte, plecîndu-se de la mulțimea variabilelor de intrare și parcurgînd succesiv instrucțiunile din bloc.

Pentru exemplul tratat în secțiunile anterioare se va obține graful de flux de date din figura 7.

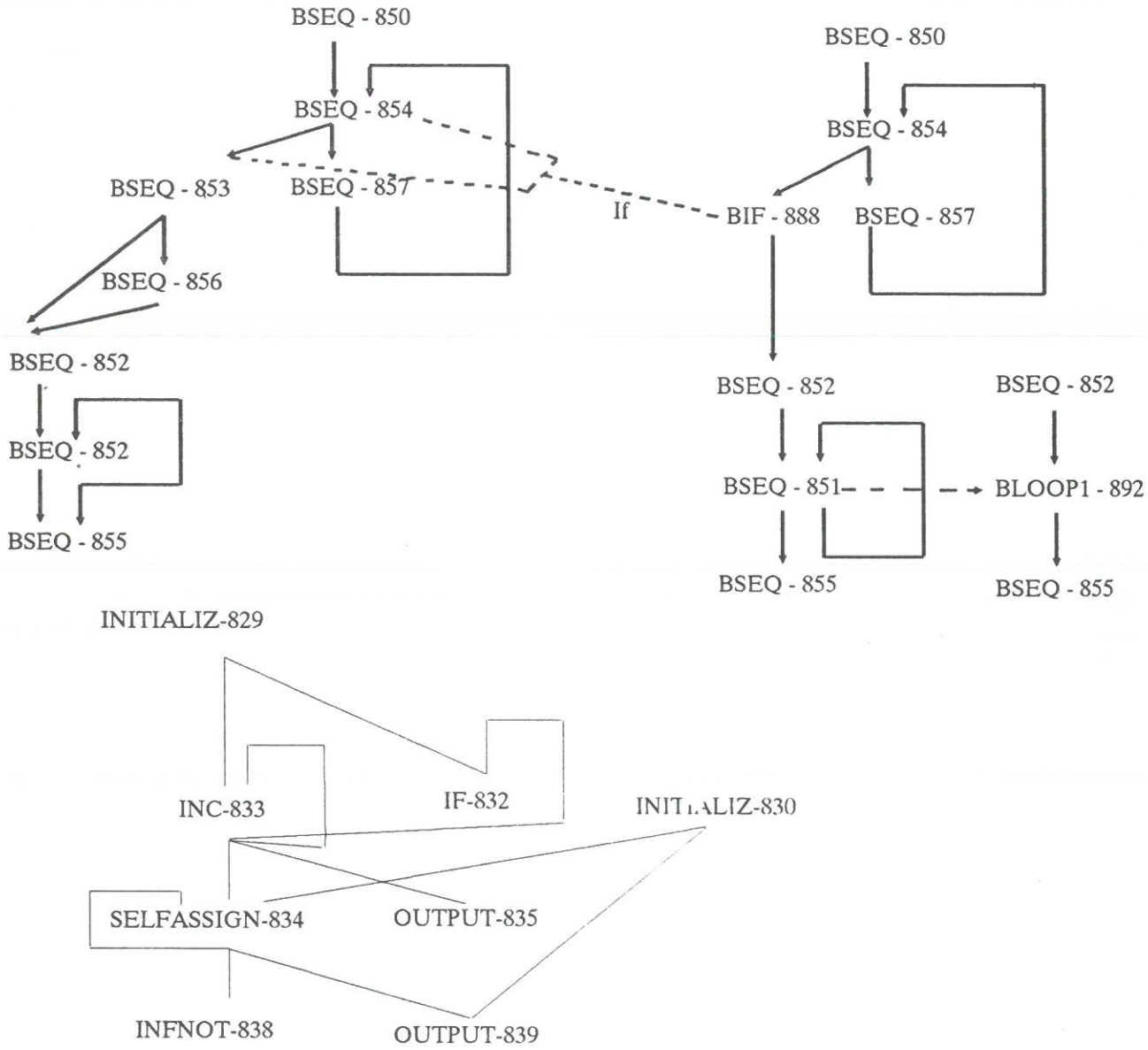

Figura 7. Graful de flux de control și abstractizarea controlului (sus); graful de flux de date (jos)

4.6 Abstractizarea controlului

Abstractizarea controlului constă în recunoașterea și în gruparea obiectelor din descrierea programului, conform structurilor fundamentale de control: secvență, decizie și ciclu. Abstractizarea controlului începe o dată cu detectarea blocurilor de bază (care corespund abstractizării de tip secvență). Pentru abstractizarea în continuare a structurilor de control în rețeaua de obiecte, a fost folosită o gramatică programată de grafuri, similară cu [24] (în care s-a renunțat la două din producțiile date acolo și s-au introdus cîteva producții noi în locul lor). Fiecare producție corespunde la o posibilitate de a abstractiza o configurație de fragment de graf de flux de control la una din structurile fundamentale de control. Efectiv, abstractizarea constă în generarea unui nou obiect care descrie abstracția făcută și gruparea nodurilor fragmentului de graf în acest nou obiect.

În figura 7. este ilustrat procesul de abstractizare a unei structuri If și a unui ciclu (prin linii punctate).

4.7 Analiza și rafinarea ciclurilor

Ciclurile dintr-un program înglobează, de obicei, mai multe decizii de proiectare și de implementare. Totodată, mult din înțelesul unui program poate fi dedus din analiza ciclurilor sale [40].

Analiza ciclurilor în sistemul prezentat are ca punct de plecare tehnicile folosite în optimizarea codului în compilatoare. Astfel, folosind o variantă a algoritmului din [1], pe baza informațiilor furnizate de graful de flux de date pentru fiecare ciclu, plecînd de la cele mai interioare în sus, se detecteaza variabilele care nu sunt modificate în ciclu, variabilele de inducție (variabile care se modifică proporțional cu numărul de iterații efectuate), instrucțiunile invariante și ieșirile din ciclu. Aceste informații, împreună cu graful de flux de date sunt apoi folosite pentru a rafina ciclurile în cicluri cu pre-test (while ... do ...), cu post-test (do ... while ...) și cicluri care depind de un index care variază după o progresie aritmetică.

Rafinarea este efectuată folosind, ca și la abstractizarea controlului, formalismul gramaticilor de grafuri (vezi secțiunea corespunzătoare în capitolul anterior). Pentru fiecare posibilitate de rafinare a unui ciclu într-un ciclu particular a fost definită cite o producție în gramatică.De exemplu, pentru programul analizat pîna acum se vor face următoarele rafinări:

Refinement to a while on blocks (BLOOP1-890) resulting BWHILE-897 nr. -20

Refinement (3) to a for loop on blocks (BLOOP1-892) resulting BFOR-898 nr. -17

4.8 Analiza structurării datelor

Analiza structurării datelor are un rol deosebit în înțelegerea unui program și, în consecință, în abstractizarea sau/și în reutilizarea lui. Dacă se poate spune că, datorită realizărilor în domeniul elaborării compilatoarelor, există un bagaj substanțial de tehnici pentru fazele de analiză lexicală, sintactică și semantică, a construirii grafului de flux de control și de date, a abstractizării controlului și a analizei ciclurilor, nu același lucru poate fi spus despre analiza și abstractizarea structurării datelor. Acest lucru rezultă și din faptul că, în compilatoare nu sunt urmărite restructurări sau abstractizări ale programelor. Cercetările în acest sens au căpătat avînt numai în ultimii ani, ca urmare a interesului sporit arătat în realizarea de sisteme de inginerie inversă [34], [44], [45], [23].

În sistemul de față, analiza datelor poate fi împărțită în mai multe grupe după cum urmează:

1) analiza legăturilor existente între date plecînd de la graful de flux de date: în cadrul acestei analize, în obiectul corespunzător fiecărei variabile din descrierea programului sunt completate următoarele sloturi:
   - alldfi - ce variabile concură (global) la setarea variabilei;
   - alldfo - în calculul căror variabile concură;
   - cobw - împreună cu ce variabile apare în instrucțiuni de atribuire (noțiune introdusă in [45]);
   - alldef - lista obiectelor unde este definită;
   - allref - lista obiectelor unde este referită;
   - alldefproto - lista prototipurilor obiectelor unde este definită;
   - allrefproto - lista prototipurilor obiectelor unde este referită.

De exemplu, pentru programul discutat în secțiunile anterioare, se obțin următoarele informații:

```
((I %%VAR0 INTVAR 0)
   alldfit:(%%VAR0)    alldfot:(%%VAR0
   %%VAR1)
   cobwt:(%%VAR0 %%VAR1)
   alldef:(INITIALIZ INC INITIALIZ INC)
   allref:(IFNOT INC SELFASSIGN OUTPUT
   OUTPUT INC IFNOT)
(S %%VAR1 REALVAR 0)
   alldfit:(%%VAR0 %%VAR1)
   alldfot:(%%VAR1)
   cobwt:(%%VAR0 %%VAR1)
   alldef:(INITIALIZ SELFASSIGN)
   allref:(SELFASSIGN IF OUTPUT)
```

După cum se evidențiază in Ă45Î, informațiile de tip cobw pot fi utilizate în restructurarea unui program conform unei politici orientate spre obiecte. Celelalte informații pot fi utilizate în diverse scopuri, după cum rezultă și din considerațiile de la punctul al 3-lea.

2) pentru o procedură sunt completate sloturile:

- params - parametrii formali ai apelului;
- inv - variabilele de intrare;
- outv - variabile de ieșire;
- iov - variabile de intrare/ieșire;
- auxv - variabile auxiliare (locale).

3) analiza vectorilor: vectorul este cea mai importantă structură de date, în special în înțelegerea programelor FORTRAN, care nu are structuri de date mai complexe. De aceea, în sistemul de față se dă o atenție deosebită analizei vectorilor. Astfel, în primul rînd, pentru vectori sunt completate următoarele sloturi:
   - indxsa - ce indexuri are la accesare;
   - indxss - ce indexuri are la setare;
   - indxnr - numărul de variabile index utilizate;
   - indexes - variabilele index utilizate.

Pentru fiecare vector și index în parte, se analizează dinamica indexului, informație utilă în înțelegerea programului. Datorită faptului că noi considerăm că această analiză poate fi extinsă cu noi posibilități, am implementat-o printr-un sistem de reguli de producție.

Analiza vectorilor poate merge mai departe, inferînd presupuneri asupra a ceea ce face un program. Pentru a exemplifica a fost implementat un mic sistem de reguli de producție, care determină utilizările posibile pentru fiecare vector din program. Pe baza acestor utilizări posibile (memorate într-o listă în slotul usage al vectorului) se poate decide încercarea de a abstractiza vectorul respectiv (împreună cu indexurile sale) la o structură de date mai complexă (după cum vom exemplifica în secțiunea următoare). Acest sistem de reguli determină, de exemplu, dacă se fac interschimbări între elementele unui vector, informație care poate conduce, în caz afirmativ, la posibilitatea ca vectorul să fie utilizat, printre altele, pentru o sortare.

4.9 Generarea de cod C și pseudocod

Descrierea programului, obținută în urma fazelor parcurse pînă acum poate sta la baza unei analize pentru abstractizarea unor structuri de date (după cum se va discuta în secțiunea următoare) sau poate fi utilizată pentru generare de cod într-un limbaj particular. Generarea de cod într-un limbaj structurat este simplificată deoarece controlul a fost abstractizat, atît obiectele care descriau instrucțiunile elementare, cît și fiecare abstracție de control fiind incluse în obiecte care reprezintă abstracția de control înconjurătoare.

Pentru scrierea efectivă de cod au fost atașate pe fiecare obiect generic din baza de cunoștințe metode specializate.

Considerînd tot programul care a parcurs toate fazele în prezentarea de pînă acum, codul C generat este:

```
void main()
 {int I;
 float S;
 I=1;
 S=0;
 while (!(I>10))
        {
        I++;
        S=S+I*I;
        printf("\n I=%d",I);
        }
 if(S>10)
        {
        printf("\n S=%f",S);
       î
 for(I=1; !(I>10); I++)
        {
        printf("\n I=%d",I);
        }
 exit(0);
 }
```

În acest mod, se poate genera cod în orice limbaj, singura adăugare în sistem pentru un nou limbaj fiind scrierea metodelor de tipărire. De exemplu, au fost introduse și metode de tipărire în pseudocod, obținîndu-se pentru program următoarea formă:

```
procedura main()
{
        I ← 1;
        S ← 0;
cît timp !(I>10)) repetă
        {
        I ← I+1;
        S ← S+I*I;
        scrie I
        }
dacă S>10
        atunci
        {
        scrie S
        }
pentru I ← 1; !(I>10); I ← I+1
repetă
        {
        scrie I
        }
stop
}
```

Introducerea de metode pentru tipărirea de pseudocod nu a fost făcută doar de dragul de a mai putea genera și altceva decît cod C, ci și în ideea de a putea descrie un program la un nivel înalt, cu atît mai mult cu cît se pot face și abstractizări ale datelor după cum rezultă din secțiunea următoare.

Referitor la generarea de cod C trebuie făcută o remarcă importantă. Vectorii în C încep de la 0 spre deosebire de FORTRAN unde încep de la 1. Acest fapt implică necesitatea modificării indexărilor pentru a obține aceeași funcționalitate în programul C ca în programul inițial. În acest scop, în sistem, pe baza analizei făcute asupra indexurilor unui vector, se face o analiză mai detaliată a modului de utilizare a acestora din urmă. Au fost identificate două posibilități. Prima este cea în care indexurile sunt folosite numai pentru indexare. În acest caz se poate face o translatare a valorilor lor cu -1 in întreg programul acolo unde sunt inițializați sau testați. A doua posibilitate apare atunci cînd indexurile sunt folosite și la calculul unor expresii.

În acest caz, translatarea nu este indicată, soluția fiind scăderea unei unități la fiecare utilizare a indexului în vectori. De exemplu, programul:

```
      dimension x(4)
      x(1)=10
      x(2)=20
      x(3)=30
      x(4)=40
      w=1
      s=0
      do 1 i=1,4
      s=s+x(i)
1     w=w+s*x(i)
      write(1,1)w
      stop
      end
```

a fost tradus în:

```
void main()
{
int I;
float W,S;
float* X;
X[0]=10;
X[1]=20;
X[2]=30;
X[3]=40;
W=1;
S=0;
for (I=1-1; !((I+1)>4); I++)
        {
         S=S+X[I];
         W=W+S*X[I];
        }
printf("\n W=%f",W);
exit(0);
}
```

în schimb, programul:

```
      dimension a(8)
      ....
      do 4 i=1,n
4     q=q*t+(n+1-i)*a(i)
      ...
      stop
      end
```

a fost tradus în:

```
void main()
{
int N,M,I;
float T,R,P,Q,X;
float* A;
...
I=1;
do
        {
       Q=Q*T+(N+1-(I-1))*A[(I-1)];
        I++;
        }
...
}
```

4.10 Abstractizarea datelor

După cum s-a amintit în secțiunea anterioară, informațiile obținute în urma analizei datelor pot fi punctul de plecare în vederea unor încercări de abstractizare a unui vector împreună cu indexurile sale la o structură de date evoluată, cum ar fi o stivă. Aceste încercări sunt implementate folosind mecanismul gramaticilor programate de grafuri. Astfel, au fost construite producții care recunosc configurațiile din graful de flux de control, susceptibile de a fi transformate la operațiile newstack, push, pop și a testului empty? și, bineînțeles, le înlocuiesc cu obiecte specifice acestor operații. Au fost făcute cîteva experimente reușite în acest sens. De exemplu, în programul FORTRAN următor, (în urma analizei structurării datelor) vectorul a împreună cu indexul i au fost recunoscuți ca urmînd o comportare tipică de stivă și, drept urmare, descrierea programului a fost modificată în sensul eliminării lui a și i și a înlocuirii lor cu o stivă:

```
      dimension a(1000)
      i=0
3     read(*,1)c
1     format(i5)
      if(c.eq.0)stop
      if(c.lt.0)goto 2
      i=i+1
      a(i)=c
      goto3
2     if(i.eq.0)stop
      if(c.eq.-1)goto4
      if(c.lt.-2)goto5
      r=1
7     r=r*a(i)
      i=i-1
      if(i.eq.0)goto6
      goto7
4     r=0
8     r=r+a(i)
      i=i-1
      if(i.eq.0)goto6
```

```
      goto8
6     write(*,9)r
      i=i+1
      a(i)=r
      goto3
5     write(*,9)c
9     format(i3)
      stop
      end
```

pseudocod-ul generat fiind:

```
procedura main()
{
STACK1049 newstack();
repetă
        {
         citeşte C
         dacă C==0
         atunci
                stop
         dacă C
         atunci
                {
                 dacă empty?(STACK1049)
                 atunci
                        stop
                 dacă !(C==-1)
                 atunci
                        {
                         dacă C<-2
                         atunci
                                {
                                 scrie C
                                 stop
                                 }
                           R ← 1;
                           repetă
                             {
                             pop(STACK1049);
                             }
                  cît timp !empty?(STACK1049)
                  }
            altfel
                 {
                 R ← 0;
                 repetă
                 {
                 pop(STACK1049);
                 }
                 cît timp !empty?(STACK1049)
          }
          scrie R
          push (R,STACK1049);
          }
```

```
          altfel
                {
                push (C,STACK1049);
                }
          }
    }
```

4.11 Comparaţie cu alte abordări de inginerie inversă

Preocupările în a construi sisteme de inginerie inversă sunt de dată relativ recentă. Majoritatea sistemelor actuale sunt dedicate, în special, limbajului COBOL datorită numărului foarte mare de programe scrise în acest limbaj. În cazul acestui limbaj problema cea mai importantă este analiza structurării datelor descrise în DATA-DIVISION [23], partea procedurală fiind de obicei mai puţin complexă şi implementînd algoritmi foarte simpli.

Toate sistemele de inginerie inversă folosesc ca o primă parte un analizor care generează o formă intermediară [17],[23],[44]. În continuare, se apelează la diverse metode pentru abstractizarea formei intermediare. Abordarea noastră se apropie cel mai mult de soluţia aleasă în sistemul Programmer's Apprentice. În acest sistem, înţelegerea programelor se face prin analiza pe baza unei gramatici de grafuri [42], [44], opţiune la care am recurs şi noi. Ce ne diferenţiază de ei este că am acordat o atenţie deosebită creării unei taxonomii de obiecte, care reduce spaţiul de căutare, şi prin faptul că utilizăm, în acelaşi scop, minisisteme experte, bazate pe reguli de producţie, care determină care gramatici de grafuri sunt indicate să se încerce (de exemplu, pe baza regulilor care analizează structurarea datelor se poate propune abstractizarea la o stivă). Pe de altă parte, modulul de analiză bazat pe gramatici de grafuri din Programmer's Apprentice este mai complex ca al nostru, ei recurgînd la derivări în paralel pe baza unei agende. Un alt punct tare al sistemului dezvoltat la MIT este formalismul clişeelor din care se pot genera automat producţii în gramatică.

5. Concluzii

Un program este materializarea gîndirii creatoare a omului, a cunoştinţelor existente în domeniul pentru care se dezvoltă programul. Cum inteligenţa artificială investighează gîndirea umană, este natural ca în învestigarea acesteia să fie abordată şi problema dezvoltării programelor. Experienţa în dezvoltarea de sisteme inteligente pentru domeniul ingineriei programării are efecte benefice, atît asupra înţelegerii activităţii de programare cît şi asupra înţelegerii

gîndirii umane şi a dezvoltării domeniului inteligenţei artificiale.

Sistemul de inginerie inversă a programelor este operaţional, el putînd fi utilizat în rescrierea de programe FORTRAN în C. Sistemul este scris in XRL, analizorul lexical, sintactic şi semantic fiind scris în C. Reprezentarea prin obiecte structurate, furnizată de XRL, a simplificat mai multe probleme cum ar fi efectuarea de inferenţe in diversele faze de analiză, problema generării de cod, dezvoltarea evolutivă a sistemului.

Majoritatea abordărilor actuale, de a scrie sisteme de inginerie inversă, pleacă de la limbajul COBOL. Acest fapt este justificat, nu numai prin interesul mai mare de a reutiliza programe în acest limbaj, ci şi prin dificultăţile care apar în abstractizarea programelor FORTRAN. În cadrul programelor COBOL, principala problemă este de a restructura descrierile volumelor mari de descrieri de date conform, de exemplu, paradigmei de programare orientată spre obiecte [23] sau a unor descrieri entitate-relaţie. În acest program, partea procedurală este de obicei foarte simplă, implicînd operaţii aritmetice triviale sau căutări în structurile de date. În cazul programelor FORTRAN, structurile de date cele mai complexe sunt vectori, accentul căzînd pe partea procedurală.

O altă concluzie în urma dezvoltării sistemului de inginerie inversă a fost posibilitatea şi avantajele integrării unor tehnici din teoria compilării cu tehnici din inteligenţa artificială. În acest mod a fost obţinut un sistem care poate efectua analize şi abstractizări în plus faţă de compilatoarele sau analizoarele uzuale.

Bibliografie

1. **AHO, A., SETHI, R., ULLMAN, J.**: Compilers. Principles, Techniques and Tools, Addison Wesley, 1986.
2. **BALZER, R.**: A 15 Year Perspective on Automatic Programming. In: IEEE Transactions on Software Engineering, vol. SE- 11, nr. 11, nov. 1985, pp. 1257-1268.
3. **BARSTOW, D.R.**: Knowledge-Based Program Construction, Elsevier, North-Holland, 1979.
4. **BĂRBUCEANU, M., TRĂUŞAN-MATU, St.**: XRL: An Evolutionary Multi-Paradigm Environment for AI Programming. In: Ph. Jorrand si V. Sgurev (eds.), Artificial Intelligence II: Methodologies, Systems and Applications, North Holland, 1987, pag. 197-205.
5. **BĂRBUCEANU, M., TRĂUŞAN-MATU, St.**: Integrating Declarative Knowledge Programming Styles and Tools in a Structured Object Environment. In: J. Mc.Dermott (ed.) Proceedings of 10-th International Joint Conference on Artificial Intelligence IJCAI'87, Milano, Italia, Morgan Kaufmann Publishers, Inc., 1987.
6. **BĂRBUCEANU, M., TRĂUŞAN-MATU, St.**: XRL2 - manual, ITCI Bucuresti, 1988.
7. **BĂRBUCEANU, M., TRĂUŞAN-MATU, St.**: XRL: A Layered Knowledge Processing Architecture Able to Inhance Itself, Studies and Researches in Computers and Informatics, vol. 2, nr. 2, Bucuresti, 1993, pp. 76- 106.
8. **BĂRBUCEANU, M., TRĂUŞAN-MATU, St., MOLNAR, B.**: Concurrent Refinement: A Model and Shell for Hierarchical Problem Solving. In: J.C. Rault (ed.), Proceedings of 10-th Workshop on Expert Systems and Their Appications, Avignon, Franţa, 1990.
9. **BĂRBUCEANU, M., TRĂUŞAN-MATU, St.**: MODELS: Towards a Language Construction Approach to Expert System Design and Enhancement, Studies and Researches in Computers and Informatics, vol.1, no. 2, iunie 1990, ICI, Bucuresti, pp. 57-76.
10. **BĂRBUCEANU, M., GHICULETE, Gh.**: O3 A Rational Design of an Open Object-Oriented Language, ICI, 1992.
11. **BOYLE, J.M., MURALIDHARAN, M.N.**: Program Reusability through Program Transformation. In: IEEE Transactions on Software Engineering, vol SE-10, no.5, sept. 1984, pp. 574-588;
12. **BROOKS, F.P.**: No Silver Bullet. Essence and Accidents of Software Engineering, Computer, pp. 10-19, apr.1987;
13. **BRACHMAN, R.J., SCHMOLZE, J.G.**: An Overview of the KL-ONE Knowledge Representation System, Cognitive Science, vol. 9, nr. 2, 1985, pp. 171-216.
14. **M. BĂRBUCEANU, TRĂUŞAN-MATU, St., MOLNAR, B.**: Integrating Declarative Knowledge Programming Styles and Tools for Building Expert Systems. In: J.C.Rault (ed.), Proceedings of 7-th International Workshop on Expert Systems and Applications, Avignon, 1987.
15. **BUNKE, H.**: Attributed Programmed Graph Grammars and their Application to Schematic Diagram Interpretation. In: IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 4, no. 6, 1982.
16. **de KLEER, J.**: An Assumption-Based TMS, Artificial Intelligence, 28,(1986), pp. 127-162.
17. *** * *, DOCKET** - concept manual, ESPRIT project DOCKET, 1992.
18. **EHRIG, H., NAGL, M., ROZENBERG, G., ROZENFELD, A.**: Graph- Grammars and Their Application to Computer Science, Lecture Notes in Computer Science 291, Springer, New York, 1986.
19. **FIKES, R., KEHLER, T.**: The Role of Frame-Based Representation in Reasoning, Communications of the ACM, Vol.28, No.9, sept. 1985, pp. 904-920.
20. **GOLDBERG, A.T.**: Knowledge-Based Programming: A Survey of Program Design and Construction Techniques. In: IEEE Transactions on Software Engineering, vol. SE-12, nr. 7, iul. 1986, pp. 752-768.
21. **HOROWITZ, E., MUNSON, J.B.**: An Expansive View of Reusable Software. In: IEEE Transactions on Software Engineering, vol. SE-10, no. 5, sept. 1984, pp. 477-487;

22. **KENNEDY, N.**: A Survey of Data Flow Analysis Techniques. In: Program Flow Analysis: Theory and Applications, Prentice Hall, 1981, pp. 5-54.
23. **LANO, K., BREUER, P.T., HAUGHTON, H.**: Reverse-Engineering and Validating COBOL via Formal Methods, ESPRIT IPSS Results and Progress, 1991, pp. 284-305.
24. **LICHTBLAU, V.**: Decompilation of Control Structures by means of Graph Transformations. In: G. Goos and J. Hartamanis (eds.): Mathematical Fondations of Software Development, Lecture Notes in Computer Science, vol.185, Springer-Verlag, Berlin, 1985.
25. **McALLESTER, D.**: Truth Maintenance. In: Proc. AAAI-90, pp. 1109- 1116.
26. **MUCHNICK, S.S., JONES, N.D.**: Program Flow Analysis: Theory and Applications, Prentice-Hall, 1981.
27. **RAMAMOORTHY, C.V., GARG, V., PRAKASH, A.**: Programming in the Large. In: IEEE Transactions on Software Engineering, vol. SE-12, nr. 7, iulie 1986, pp. 769-784;
28. **REUBENSTEIN, H.B., WATERS, R.C.**: The Requirements Apprentice: Automated Assistance for Requirements Acquisition. In: IEEE Transactions on Software Engineering, vol. 17. no. 3, martie 1991, pp. 226-240.
29. **RAMAMOORTHY, C.V., GARG, V., PRAKASH, A.**: Programming in the Large. In: IEEE Transactions on Software Engineering, vol. SE-12, nr. 7, iulie 1986, pag. 769-784;
30. **RICH, C.**: The Layered Architecture of a System for Reasoning about Programs. In: Proceedings IJCAI'85, pp.540-548.
31. **RICHER, H.**: An Evaluation of Expert System Development Tools. Raport KSL 85-19, Stanford, iunie 1986.
32. **RICH, C., WATERS, R.C.**: Automatic Programming: Myths and Prospects. In: Computer, august 1988;
33. **RICH, C., WATERS, R.C.**: Programmer's Apprentice - A Research Overview. In: Computer, nov. 1988, pp. 11-25;
34. **RICH, C., WATERS, R.C.**: The Programmer's Apprentice, ACM Press, Addison-Wesley, 1990.
35. **SCHINDLER, M.**: Computer-Aided Software Design. Build Quality Software with CASE, John Wiley & Sons, 1990;
36. **SMITH, D. R.**: Research on Knowledge-Based Software Environments at Kestrel Institute. In: IEEE Transactions on Software Engineering, vol. SE-11, nr. 11, nov. 1985, pp. 1278-1295.
37. **SCHOEN, E., SMITH, R. G., BUCHANAN, B. G.**: Design of Knowledge-Based Systems with a Knowledge-Based Assistant. In: IEEE Transactions on Software Engineering, vol. 14, no. 12, dec. 88.
38. **TRĂUȘAN-MATU, St.**: Micro-XRL: An Object-Oriented Programming Language for Microcomputers, Raport de cercetare, Technical Cybernetics Institute, Bratislava, 1989.
39. **TRAUSAN-MATU, St.**: Constraint Processing in the mXRL Object- Oriented Language, Research report, Institute for Technical Cybernetics, Slovak Academy of Sciences, Bratislava, 1989.
40. **TRĂUȘAN-MATU, ST.**: Proiectarea asistată de calculator a programelor. Reproiectarea și reutilizarea programelor prin inginerie inversă, Teza de doctorat, UPB, 1993.
41. **WATERS, R.C.**: A Method for Analyzing Loop Programs. In: IEEE Transactions on Software Engineering, 5(3), mai 1979, pp. 237- 247.
42. **WATERS, R.C.**: The Programmer's Apprentice: A Session with KBEmacs. In: IEEE Transactions on Software Engineering, vol. SE-11, nr. 11, nov. 1985, pp. 1296-1320.
43. **WILLS, L.M.**: Automated Program Recognition: A Feasibility Demonstration, Artificial Intelligence, 45, 1990, pp. 113-171.
44. **WILLS, L.M.**: Automated Program Recognition by Graph Parsing, AI-TR 1358, MIT, 1992.
45. **ZIMMER, J.A.**: Restructuring for Style, Software-Practice and Experience, vol. 20, no. 4, apr. 1990, pp. 365-389.

INFORMAŢII GENERALE

Locul de desfăşurare

UNIVERSITATEA POLITEHNICA BUCUREŞTI
Facultatea de Automatică si Calculatoare
Splaiul Independenţei 313
Bucureşti 77206
ROMANIA

CSCS' 10 se va desfăşura sub patronajul Academiei Române, al Ministerului educaţiei şi Invăţământului, al Ministerului Cercetării şi tehnologiei, al SRAIT şi al secţiunii române a IEEE.

Presedintele conferinţei: Prof. Ion DUMITRACHE
Secretarul conferinţei: Ana Mihaela IONESCU

Lucrări

Lucrările trebuie propuse sub forma unui rezumat care sa nu depăşească 500 de cuvinte, scris în limba engleză. Lucrările selectate vor fi prezentate în cadrul conferinţei şi vor fi publicate în reprint-uri.

Termene limită

- prezentarea titlului şi a rezumatului 30.11.1994
- selecţia preliminară şi informarea acceptului 30.12.1994
- prezentarea întregului text al lucrărilor acceptate 1.02.1995
- program final 31.03.1995

Expoziţie tehnică

În paralel cu conferinţa, va fi deschisă o expoziţie tehnică axată pe tematica acesteia.

Corespondenţa

Toată corespondenţa legată de conferinţă trebuie adresată către:

CSCS' 10
UNIVERSITATEA POLITEHNICA BUCUREŞTI
Facultatea de Automatică şi Calculatoare
Splaiul Independenţei 313
Bucureşti 77206
ROMANIA
Telefon: 401-6314120/324
Fax: 401-3122400
Telex: 1025 2 IPOLB R

TEMATICA GENERALĂ A CONFERINŢEI

1.TEORIA SISTEMELOR APLICATE
- 1.1. Controlul adaptiv şi optimal
- 1.2. Identificare şi modelare
- 1.3. Algoritmi de control
- 1.4. Control robust

2.CONTROLUL PROCESELOR INDUSTRIALE
- 2.1. Sisteme distribuite şi de control prin multiprocesor
- 2.2. Controllere, RISC, ASICS
- 2.3. Algoritmi de control avansat
- 2.4. Control inteligent

3.ROBOŢI SI CELULE FLEXIBILE DE FABRICAŢIE
- 3.1. Controlul roboţilor industriali
- 3.2. Vedere artificiala şi tehnici de IA
- 3.3. Celule flexibile de fabricaţie
- 3.4. Fabricaţia integrată pe calculator

4.ARHITECTURI DE CALCULATOARE ŞI SISTEME DISTRIBUITE
- 4.1. Arhitectura sistemului
- 4.2. Sisteme paralele
- 4.3. Reţele de calculatoare
- 4.4. Sisteme distribuite şi algoritmi

5.PROGRAMAREA SISTEMELOR
- 5.1. Limbaje şi medii de programare
- 5.2. Sisteme de baze de date
- 5.3. Ingineria software-ului şi aplicaţii
- 5.4. Sisteme avansate de operare

6.PROIECTAREA ASISTATĂ DE CALCULATOR
- 6.1. Proiectarea structurii VLSI
- 6.2. CASE (proiectare software asistată de calculator
- 6.3. Sisteme grafice
- 6.4. Proiectarea asistată de calculator a sistemelor de control

7.INTELIGENŢA ARTIFICIALĂ
- 7.1. Limbaje naturale
- 7.2. Sisteme neurale
- 7.3. Sisteme expert

8.BIOENERGIE ŞI SISTEME COGNITIVE
- 8.1. Modelarea proceselor biologice
- 8.2. Biosenzori şi percepţie artificială
- 8.3. Sisteme cognitive şi de instruire
- 8.4. Sisteme expert medicale

A 10-a Conferinţă Internaţională
de
SISTEME DE CONTROL ŞI INFORMATICĂ

CSCS 10

24-26 mai 1995

Vă rugăm să transmiteţi acest formular prin poştă sau fax la:

CSCS' 10
UNIVERSITATEA POLITEHNICA BUCUREŞTI
Facultatea de Automatică şi Calculatoare
Splaiul Independenţei 313
Bucureşti 77206
ROMANIA
Fax: 401-3122400

Nume: .....
Prenume: .....
Firmă/Institutie: .....
Direcţie/Departament: .....
.....
Titlul şi funcţia: .....
Adresa: .....
.....
.....
Telefon ..... Fax: .....
Email: .....
Vă rugăm să marcaţi caseta/casetele corespunzătoare:
Sunt interesat de:
- participarea la conferinţă ☐
- prezentarea unui produs ☐
- prezentarea lucrării/lucrărilor intitulate ☐

.....
.....
.....

Semnătura: Data: