AUTOMATIZAREA VERIFICĂRII CORECTITUDINII PROGRAMELOR SCRISE ÎN LIMBAJ IMPERATIV

ing. Horaţiu Voicu

Universitatea Tehnică "Gh. Asachi", Iaşi
Facultatea de Automatică şi Calculatoare

Rezumat: Lucrarea prezintă o modalitate în care se poate realiza verificarea automată a corectitudinii programelor scrise în limbaje imperative. Pornind de la noţiunile teoretice necesare iniţierii în domeniu (logica Floyd-Hoare [1]), se prezintă o implementare în limbajul CAML Light a unui verificator de programe care este format din două funcţii principale : un demonstrator de teoreme (a) şi un generator de condiţii de verificare (b). Programul (algoritmul care se doreşte a fi testat) este furnizat verificatorului de programe, care prin intermediul funcţiei (b), generează o serie de condiţii de verificare, furnizate funcţiei (a) care încearcă să le demonstreze. În caz că reuşeşte, înseamnă că programul este corect. În caz contrar, nu se poate trage nici o concluzie.Un astfel de program este util în practică atunci când o testare exhaustivă a unui algoritm este imposibil de realizat şi este utilizat, mai ales, în domeniul producerii software-ului care controlează activităţi potenţial dăunătoare omului (convertirea energiei atomice în energie electrică, ghidarea navetelor spaţiale).
Cuvinte cheie: condiţii de verificare, demonstrator de teoreme, ß-conversie, specificaţie, adnotare, limbaj funcţional.

1. Introducere

Limbajul ML este un limbaj funcţional şi a fost special proiectat pentru a construi demonstratoare de teoreme. Limbajele de programare se numesc funcţionale atunci când la baza structurii programelor stă noţiunea de funcţie, şi structura esenţială de control este apelul funcţiilor. Proiectarea verificatorului s-a realizat utilizând o metodologie bottom-up, care este în concordanţă cu structura sistemului CAML Light, acesta constând dintr-un nucleu extins cu câteva tipuri de date şi funcţii standard [2], [3].

Ideea care stă la baza verificatorului de programe construit constă în faptul că execuţia unui program pe o maşină de tip von Neumann este determinată de programul care se execută şi starea iniţială din care a plecat maşina. Prin stare iniţială se înţelege starea procesorului şi valorile variabilelor globale ale programului în cauză. Verificatorului de programe i se va furniza programul care se doreşte a fi testat, starea iniţială şi starea finală la care se doreşte să se ajungă. Având la dispoziţie aceste date, verificatorul va genera o serie de condiţii de verificare (i.e. propoziţii logice), folosind reguli specifice fiecărui tip de instrucţiune, condiţii care vor fi furnizate unui demonstrator de teoreme care va încerca să le demonstreze. În caz că reuşeşte, înseamnă că există o demonstraţie riguroasă a corectitudinii programului în cauză. În caz că nu reuşeşte, programul poate fi corect sau nu. Puterea de demonstrare este dependentă de numărul şi de felul axiomelor cu care este înzestrat demonstratorul de teoreme.

2. Suportul teoretic al implementării verificatorului.

2.1 Definiţii şi reguli

Noţiunile teoretice ce se prezintă în continuare au în vedere faptul că algoritmul ce se doreşte a fi testat este scris într-un limbaj imperativ minimal (cuprinde doar comenzi de tipul : assign, if, while şi for). O prezentare completă a lor se găseşte în [4].

Definiţie : Se numeşte *specificaţie*

$\{ P \}\ C\ \{ Q \}$,

unde :

C este un program ce se doreşte a fi specificat.

P şi Q sunt condiţii asupra variabilelor programului C. ■

Condiţiile asupra variabilelor programului sunt scrise folosind simboluri matematice standard împreună cu operatorii logici $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$.

Definiţie: Dacă $\{ P \}\ C\ \{ Q \}$ este o specificaţie, atunci P se numeşte *precondiţie,* iar Q *postcondiţie.* ■

Definiţie: Spunem că $\{ P \}\ C\ \{ Q \}$ este o *specificaţie parţial corectă* atunci când C este executată cu o stare iniţială în care P este adevărată, iar în eventuala stare finală (C poate să nu se termine niciodată) propoziţia Q este adevărată. ■

Definiţie: Expresia $[P]\ C\ [Q]$ se numeşte *specificaţie totală.* ■

Definiţie: O specificaţie totală $[P]\ C\ [Q]$ este corectă dacă următoarele două condiţii sunt adevărate:

(i) Oricând C este executată într-o stare care satisface P atunci C se termină în timp finit.

(ii) După terminare Q este adevărată. ■

Relaţia dintre corectitudinea totală şi cea parţială poate fi exprimată informal prin ecuaţia :

Corectitudine Totală = Terminare + Corectitudine Parţială

Definiţie: Se numeşte *demonstraţie formală* o succesiune de formule, fiecare fiind, ori o aplicare a unei axiome, ori este dedusă din liniile anterioare printr-o regulă de inferenţă. ■

Motivul pentru care se construiesc demonstraţii formale este acela de a fi siguri că se folosesc doar metode de deducţie riguroase. Având axiome şi reguli de inferenţă riguroase este sigur că deducţiile

sunt adevărate. Alegerea regulilor de inferență trebuie să fie făcută cu grijă pentru a nu genera informații de corectitudine false.

În continuare se introduc progresiv regulile de inferență pentru fiecare comandă a limbajului imperativ minimal. Ele se găsesc de asemenea în [4]. Regulile de inferență vor fi scrise în forma:

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}.$$

Aceasta înseamnă că teorema S este dedusă din ipotezele $S_1,\ldots,S_n$. Ipotezele pot fi axiome din logica Floyd-Hoare sau din alte teoreme deduse anterior.

Regula asignării

Axioma asignării reprezintă faptul că valoarea unei variabile V după executarea unei comenzi **assign V E** este egală cu valoarea expresiei E în starea anterioară execuției comenzii. Pentru a formaliza acest lucru trebuie să se observe că, dacă o propoziție P este adevărată după asignare, atunci propoziția obținută prin substituția lui V cu E în P trebuie să fie adevărată înaintea asignării. Se notează cu P[E/V] rezultatul înlocuirii tuturor aparițiilor lui V în P prin E. ■

Exemplu:

$(a + 1 > a)[b + c / a] = ((b + c) + 1 > b + c)$.

Axioma asignării este :

$$\vdash \{ P[E / V] \} \text{ assign } V \; E \{ P \}, \qquad (1.1)$$

unde V este orice variabilă, E este orice expresie. ■

Regula secvenței

$$\frac{\vdash \{P\} \; C_1 \{Q\}, \vdash \{Q\} \; C_2 \{R\}}{\vdash \{P\} \; C_1 ; C_2 \{R\}} \qquad (1.2)$$ ■

În această regulă se arată faptul că dacă două specificații pentru care postcondiția uneia este identică cu precondiția celeilalte sunt corecte, atunci și specificația care are precondiția identică cu precondiția primei specificații și postcondiția identică cu postcondiția celei de-a doua specificații și programul format din programele celor două specificații scrise în ordine corespunzătoare este corectă.

Regulile condiționale

Există două feluri de comenzi condiționale : cu o singură ramură și cu două ramuri. Deci, sunt prezentate două reguli pentru comenzi condiționale:

$$\frac{\vdash \{P \wedge S\} \; C \; \{Q\}, \; P \wedge \neg S \Rightarrow Q}{\vdash \{P\} \text{ if } S \text{ then } C \{Q\}}; \qquad (1.3)$$

$$\frac{\vdash \{P \wedge S\} \; C_1 \; \{Q\}, \; P \wedge \neg S \; \{C_2\} Q}{\vdash \{P\} \text{ if } S \text{ then } C_1 \text{ else } C_2 \{Q\}}. \qquad (1.4)$$

Regula WHILE

$$\frac{\vdash \{P \wedge S\} \; C \; \{P\}}{\vdash \{P\} \text{ while } S \text{ do } C \; \{P \wedge \neg S\}} \qquad (1.5)$$

Cu excepția comenzii WHILE, toate axiomele și regulile descrise până acum sunt riguroase din punctul de vedere al corectitudinii totale ca și al corectitudinii parțiale. Regula WHILE nu este valabilă pentru corectitudinea totală, ci doar pentru corectitudinea parțială.

Regula FOR

$$\frac{\vdash \{P \wedge (E_1 \le V) \wedge (V \le E_2)\} \; C \; \{P[V+1/V]\}}{\vdash \{P[E_1/V] \wedge (E_1 \le E_2)\} \text{ for } V = E_1 \text{ until } E_2 \text{ do } C \{P[E_2 + 1/V]\}}, \qquad (1.6)$$

unde nici V, nici variabilele care apar în E_1 sau E_2 nu sunt asignate în comanda C. ■

Această regulă nu permite deducerea nici unui lucru despre comenzile FOR în care se asignează variabile din expresiile ce desemnează limitele intervalului. În expresiile ce delimitează limitele intervalului comenzii FOR nu se pot face asignări.

Pentru a include și cazul când $E_2 < E_1$ este nevoie de axioma de mai jos :

$$\vdash \{P \wedge (E2 < E1)\} \text{ for } V = E1 \text{ until } E2 \text{ do } C\{P\}. \qquad (1.7)$$

Această axiomă se traduce prin : dacă $E_2 < E_1$ atunci comanda **for** nu are nici un efect.

2.2 Automatizarea verificării programelor

Se poate demonstra că este imposibil de construit o procedură care să determine automat valoarea de adevăr a unei propoziții matematice. Aceasta nu înseamnă că nu există proceduri care pot demonstra multe teoreme folositoare. Inexistența unei proceduri generale de decizie arată că nu se poate spera să se demonstreze automat orice. Însă, în practică, este posibil de a automatiza multe din aspectele de rutină ale verificării programelor. Deși este imposibil de a determina valoarea de adevăr a unei propoziții arbitrare este totuși posibil de a verifica dacă o demonstrație formală este validă.

Tehnica adoptată în realizarea verificatorului automat este înlănțuirea înapoi (backward chaining), adică se pornește de la concluzie și se ajunge la ipoteză.

Noțiunile necesare în limbajul descrierii generării condițiilor de verificare se introduc prin următoarele trei definiții.

Definiție: O comandă *adnotată* este o comandă cu propoziții inserate în ea. Adnotările încep cu { și se termină cu }. ■

Definiție: Se spune că o comandă este *adnotată propriu-zis* dacă propozițiile au fost inserate în următoarele locuri :

(i) Înaintea fiecărei comenzi C_i (unde $i > 1$) într-o succesiune C_1; C_2; ... C_n exceptând cazul în care C_i este o comandă de asignare.

(ii) După cuvântul **do** in comenzile **while** şi **for**.■

Intuitiv propoziţiile inserate ar trebui să exprime condiţiile care trebuie să fie adevărate la un anumit punct în program.

Definiţie: O *specificaţie adnotată propriu-zis* este o specificaţie {P} C {Q} în care C este o comandă adnotată propriu-zis. ■

Exemple de adnotări se află în ultima secţiune a articolului unde sunt listate exemple de algoritmi testaţi de verificator.

Generarea condiţiilor de verificare

Se prezintă generarea condiţiilor de verificare pentru fiecare comandă din limbajul definit. Acestea se deduc din regulile (1.1) , ... , (1.7)

Comanda de asignare:

Specificaţia {P} assign V E {Q} generează condiţia de verificare :

$$P \Rightarrow Q[E/V]. \qquad (2.1)\blacksquare$$

Comanda condiţională cu o singură ramură:

Specificaţia {P} if S then C {Q} generează condiţiile :

(i) $(P \wedge \neg S) \Rightarrow Q$.

(ii) condiţiile de verificare generate de:

$\{P \wedge S\}$ C $\{Q\}$. (2.2)■

Comanda condiţională cu două ramuri:

Specificaţia {P} If S then C_1 else C_2 {Q} generează condiţiile:

(i) condiţiile generate de $\{ P \wedge S \}$ C_1 $\{ Q \}$.

(ii)condiţiile generate de:

$\{P \wedge \neg S\}$ C_2 $\{Q\}$. (2.3)■

Secvenţe:

Se disting două cazuri :

(a) Condiţiile de verificare generate de:

$\{ P \}$ seq C_1; ... C_{n-1}; $\{ R \}$ C_n ; : $\{ Q \}$

(unde C_n nu este o asignare) sunt :

(i) Condiţiile de verificare generate de:

$\{ P \}$ seq C_1; ... C_{n-1} ; : $\{ R \}$.

(ii) Condiţiile de verificare generate de:

$\{ R \}$ C_n $\{ Q \}$.

b) Condiţiile de verificare generate de:

$\{ P \}$ seq C_1; ... C_{n-1}; assign I E ; : $\{ Q \}$

sunt condiţiile de verificare generate de:

$\{ P \}$ seq C_1; ... C_{n-1}; : $\{ Q[E/V] \}$. (2.4)■

Comenzile WHILE:

Condiţiile de verificare generate de:

$\{ P \}$ while S do $\{ R \}$ C $\{ Q \}$

sunt:

(i) $P \Rightarrow R$

(ii) $(R \wedge \neg S) \Rightarrow Q$

(iii) condiţiile de verificare generate de:

$\{ R \wedge S \}$ C $\{ R \}$. (2.5)■

Comenzile FOR:

Condiţiile de verificare generate de:

$\{ P \}$ for $V = E_1$ until E_2 do $\{ R \}$ C $\{ Q \}$

sunt:

(i) $P \Rightarrow R[E_1/V]$,

(ii) $R[E_2+1/V] \Rightarrow Q$,

(iii) $P \wedge (E_2 < E_1) \Rightarrow Q$,

(iv) condiţiile de verificare generate de

$\{ R \wedge (E_1 \leq V) \wedge (V \leq E_2) \}$ C $\{ R[V+1/V] \}$;

(v) condiţiile sintactice: nici V nici o altă variabilă ce apare în E_1 sau E_2 nu este asignată în interiorul lui C. (2.6)■

Teorema care stă la baza automatizării verificării programelor este următoarea:

Teoremă: O specificaţie adnotată $\{ P\}$ C $\{ Q\}$ este demonstrabilă în logica Floyd-Hoare (adică $\vdash \{P\}$ C $\{Q\}$), dacă condiţiile de verificare generate de ea sunt demonstrabile. ■

Aşadar, demonstrabilitatea condiţiilor de verificare este o condiţie suficientă, dar nu şi necesară. De fapt, ea reprezintă cea mai slabă condiţie suficientă.

Deci, se poate afirma că, pentru a demonstra corectitudinea specificaţiei $\{ P \}$ C $\{ Q \}$, trebuie să se parcurgă următoarele trei etape :

(i) Programul C trebuie să fie *adnotat* prin inserarea de propoziţii care trebuie să fie adevărate la diferite puncte în program. Această fază poate fi înşelătoare, automatizarea ei este o problemă de inteligenţă artificială (pentru verificatorul în cauză ea este efectuată de către expertul uman).

(ii) Se generează o mulţime de propoziţii numite *condiţii de verificare*. Acest proces este pur mecanic şi, deci, uşor de implementat.

(iii) Sunt demonstrate condiţiile de verificare. Automatizarea acestui lucru este, de asemenea, o problemă de inteligenţă artificială.

3. Implementarea în limbajul ML a demonstratorului de teoreme

Scopul demonstratorului este de a demonstra automat cât mai multe condiţii de verificare pe care le produce generatorul de condiţii de verificare.

Demonstratorul care se construieşte este bazat pe reguli de rescriere. O afirmaţie este demonstrată prin înlocuirea repetată a subexpresiilor cu subexpresii echivalente până când se determină că afirmaţia este adevărată sau demonstratorul se opreşte.

Demonstratorul înlocuieşte E_1 cu E_2 dacă :

1) ($E_1 = E_2$) este o axiomă pe care sistemul o posedă;

sau

2) E_1 reprezintă $S_1 \Rightarrow S_2$ şi E_2 repezintă $S_1 \Rightarrow S_2[true/S_1]$;

sau

3) E_1 reprezintă ($x = E$) $\Rightarrow$ S şi E_2 reprezintă $(x = E) \Rightarrow S[E/x]$,

unde, ca de obicei, $S[S_1/S_2]$ notează rezultatul înlocuirii lui S_2 cu S_1 în S.

În teoria lambda calculului [4] se defineşte ß-conversia care înseamnă o evaluare a unei expresii E_1 la o expresie E_2 ţinând cont de un anumit mediu (i.e. reguli de rescriere). Deci, regula de inferenţă a demonstratorului care se proiectează este ß-conversia [4]. Demonstratorul evaluează expresiile şi încearcă să obţină valoarea true (aceasta însemnând că teorema care este codificată prin expresia respectivă este demonstrată).

Cea mai mare parte a demonstratorului este un motor de rescriere. Acest motor de rescriere primeşte la intrare un set de ecuaţii (regulile de rescriere sau axiome : ($E_{11} = E_{12}$) ... ($E_{n1} = E_{n2}$)) şi o expresie E pentru a fi rescrisă şi apoi înlocuieşte repetitiv apariţii ale lui E_{1i} cu expresia E_{2i} până când nu mai apar schimbări.

Lista cu axiome a fost "acordată" (în conţinut şi ordine) astfel încât să rezolve cât mai multe condiţii de verificare (aserţiuni matematice). Lista se poate îmbunătăţi prin adăugarea de noi ecuaţii. Însă această îmbunătăţire trebuie făcută cu atenţie pentru a nu micşora mulţimea teoremelor pe care demonstratorul le poate rezolva. Deoarece demonstratorul parcurge lista cu axiome în ordinea în care acestea sunt scrise, rezultă că trebuie respectate următoarele reguli atunci când se scriu axiomele:

(i) primele axiome vor fi de uz general şi vor avea o doză mare de generalitate;

(ii) următoarele axiome vor fi puse în grupuri specializate pe anumite probleme. În interiorul acestor grupuri axiomele care particularizează cel mai mult vor fi puse primele pentru a nu fi obturate de cele cu un caracter mai general.

Se listează mai jos câteva din axiomele cu care este înzestrat demonstratorul de teoreme :

(((x & y) $\Rightarrow$ z) = (x $\Rightarrow$ (y $\Rightarrow$ z)))

((!(! x)) = x)

(((x $\leq$ y) $\Rightarrow$ ((y $\leq$ z) $\Rightarrow$ (x $\leq$ z))) = true)

(((x $\leq$ y) $\Rightarrow$ ((y < z) $\Rightarrow$ (x < z))) = true)

(((x $\leq$ m) $\Rightarrow$ ((y $\leq$ m) $\Rightarrow$ ((m < z) $\Rightarrow$ ((x $\leq$ z) & (y $\leq$ z))))) = true)

(((x gcd y) = (y gcd x)) = true)

((x gcd x) = x)

(((x > y) $\Rightarrow$ (d = ((x - y) gcd y))) = (d = (x gcd y)))

(((x > y) $\Rightarrow$ (d = (y gcd (x - y)))) = (d = (x gcd y)))

Mai jos se descrie în pseudocod algoritmul care stă la baza demonstratorului. Ordinea în care sunt listate funcţiile este top-down.

```
   demonstrează (ecuaţii,expr)
{
expl =
rescrie_în_adâncime(ecuaţii,
                    simplificarea_implicaţiei(expr));
while (expl != expr )
{
expl = expr;
expr =
rescrie_în_adâncime(ecuaţii,
                    simplificarea_implicaţiei(expr));
}
return(expr);
}
```

Această funcţie simplifică în adâncime (i.e. toate subexpresiile) toate implicaţiile (vezi funcţia **simplificarea_implicaţiei**) şi rescrie în adâncime expresia până obţine o expresie care nu mai poate fi modificată (vezi funcţia **rescrie_în_adâncime**).

```
   rescrie_în_adâncime(ecuaţii,expr)
{
expl = repetă_rescrie(ecuaţii,expr);
if ( expl este atom ) then return(expl);
else
repetă_rescrie(ecuaţii,
     cons(rescrie_în_adâncime(ecuaţii,first(expl)),
          rescrie_în_adâncime(ecuaţii,rest(expl))
                )
          )
}
```

Această funcție rescrie expresia **expr** pe care o primește, în funcție de axiomele **ecuații**, până când nu se mai produce nici o schimbare. Dacă rezultatul obținut este un atom (adică o variabilă sau o constantă), atunci funcția reîntoarce expresia obținută mai sus. Dacă expresia obținută nu este un atom, atunci se aplică funcția recursiv pe subexpresii. Subexpresiile sunt subarborii stângi și drepți ai arborelui care codifică expresia.

repetă_rescrie(ecuații,expr)

```
{
exp1 = rescrie(ecuații,expr);
while (exp1 != expr )
{
exp1 = expr;
expr = rescrie(ecuații,expr);
}
return(expr);
}
```

Această funcție rescrie ecuația până când nu mai are loc nici o schimbare.

rescrie(ecuații,expr)

```
{
if (ecuații = listă vidă)  then return(expr)
else rescrie(rest(ecuații),rescrie1(first(ecuații),exp))
}
```

rescrie1(ecuație,expr)

```
{
l = membrul stâng al ecuației;
r = membrul drept al ecuației;
sub = match(l,expr)          // sub reprezintă o listă
cu
                  // substituțiile care trebuie
                  // făcute pentru ca l să se
                  // potrivească cu expr.
if (sub = listă vidă) then return(expr);
else
        return(substituie_din_lista(sub,r);
        // întoarce membrul drept al axiomei
(ecuației)
        // dar cu substituirile făcute
}
```

Funcția **simplificarea_implicației** transformă $P \Rightarrow Q$ în $P \Rightarrow Q[true/P]$ și $(X = E) \Rightarrow Q$ în $(X = E) \Rightarrow Q[E/X]$.

Funcțiile **cons**, **first**, **rest** au semnificația din limbajul Lisp adică **cons** concatenează două expresii, **first** furnizează cel mai din stânga element dintr-o expresie, iar **rest** furnizează ceea ce rămâne dintr-o expresie dacă se înlătură cel mai din stânga element.

4. Implementarea generatorului de condiții în ML

Generatorul de condiții de verificare construit este bazat pe principiile explicate în introducere. Mai jos, este descris în pseudocod algoritmul care stă la baza generării condițiilor de verificare.

gen_cond_ver(p,c,q)

```
{       // p este precondiția, c este comanda, iar q este
        // postcondiția.
        switch ( tip_comandă(c)) {
        case'assign':assign_gen_cond_ver(p,c,q); break;
        case 'if'  : if_gen_cond_ver(p,c,q);break;
        case 'If'  : If_gen_cond_ver(p,c,q);break;
        case'while':assign_gen_cond_ver(p,c,q); break;
        case 'for' : for_gen_cond_ver(p,c,q);break;
        case 'seq': seq_gen_cond_ver(p,c,q);break;
}
```

Funcțiile de generare pentru fiecare comandă în parte sunt scrise astfel încât să respecte regulile (2.1), ... , (2.6). Funcția gen_cond_ver și toate celelalte funcții de generare a condițiilor de verificare pentru comenzi (în afară de assign_gen_cond_ver) sunt mutual recursive deoarece sunt instrucțiuni care pot conține în interiorul lor alte instrucțiuni.

5. Implementarea interfeței cu utilizatorul

Legătura dintre utilizator și funcția care execută verificarea efectivă este realizată de o interfață ce conține un analizor lexical, un analizor sintactic și un mediu integrat simplu pentru dezvoltarea algoritmilor.

Programul principal

Programul principal este un fișier de comenzi în Unix care asigură o interfață prietenoasă cu utilizatorul.

Deoarece programul CAML LIGHT există, atât sub sistemul de operare Unix, cât și sub sistemul de operare Dos și acest verificator poate fi obținut în cele două versiuni.

Sub sistemul de operare Unix programul obținut prin compilare se numește main. Acesta poate fi folosit cu următoarea sintaxă:

main nume_fișier [-opțiuni] ,

unde nume_fișier este numele fișierului care conține algoritmul ce se dorește a fi verificat.

Opțiuni :

-d se generează doar demonstrația;

-v se generează doar condițiile de verificare;

-f ieşirea verificatorului (stdout) devine fişierul care are numele nume_fişier, dar cu extensia schimbată în funcție de ce s-a generat **.demo** (pentru demonstrație) şi **.ver** pentru condiții de verificare.

Pentru o mai uşoară manipulare în versiunea de sub Unix există fişierul batch **verifier,** care este o interfață prietenoasă cu utilizatorul.

6. Exemple de algoritmi care au fost testați cu verificatorul

Exemplul 1:

```
# Algoritm ce calculează cel mai mare divizor
comun # a două numere.
{ (((A = a) & (B = b)) & (d = (a gcd b))) }
 seq
        while !(A = B) do
        { (d = (A gcd B))  }
        seq
                while ( A > B ) do
                { (d = (A gcd B))  }
                        assign A (A - B )
                ;
                { (d = (A gcd B))  };
                while ( B > A ) do
                { (d = (A gcd B))  }
                        assign B (B - A)
                ;
        :
        ;
:
{ (( A = B ) & ( A = d)) }
```

Condițiile de verificare pentru acest algoritm sunt:

(((d =(A gcd B))&(B > A))⇒ (d =(A gcd(B-A))))

(((d =(A gcd B))&(!(B > A))) ⇒ (d =(A gcd B)))

((d =(A gcd B)) ⇒ (d =(A gcd B)))

(((d =(A gcd B))&(A > B)) ⇒ (d =((A-B)gcd B)))

(((d =(A gcd B))&(! (A > B)))⇒ (d =(A gcd B)))

(((d =(A gcd B))&(! (A = B))) ⇒ (d =(A gcd B)))

(((d =(A gcd B))&(!(!(A=B))))⇒((A=B)&(A=d)))

((((A=a)&(B=b))&(d =(a gcd b)))⇒(d=(A gcd B)))

Exemplul 2 :

```
# Algoritm de sortare a trei numere.
{ true }
 seq
        if (x2 < x1) then
                seq
                 assign r x1;
                 assign x1 x2;
                 assign x2 r;
                :
        ;
        { (x1 <= x2) };
        if (x3 < x2) then
                seq
                 assign r x2;
                 assign x2 x3;
                 assign x3 r;
                :
        ;
        { ((x2 <= x3) & (x1 <= x3)) };
        if (x2 < x1) then
                seq
                 assign r x1;
                 assign x1 x2;
                 assign x2 r;
                :
        ;
 :
{ (((x1 <= x2) & (x2 <= x3)) & (x1 <= x3)) }
```

Condițiile de verificare pentru acest program sunt:

(((x2 ≤ x3)&(x1 ≤ x3))&(x2 < x1))⇒(((x2 ≤ x1) & (x1 ≤ x3))&(x2 ≤ x3)))

(((x2 ≤ x3)&(x1 ≤ x3))&(!(x2 < x1)))⇒(((x1 ≤ x2) & (x2 ≤ x3))&(x1 ≤ x3)))

(((x1≤ x2)&(x3<x2))⇒((x3≤x2)&(x1≤x2)))

$(((x1 \le x2) \& (!(x3 < x2))) \Rightarrow ((x2 \le x3) \& (x1 \le x3)))$

$((\text{true} \& (x2 < x1)) \Rightarrow (x2 \le x1))$

$((\text{true} \& (!(x2 < x1))) \Rightarrow (x1 \le x2))$

Exemplul 3 :

```
# Algoritm de calcul al maximului dintre trei
# numere.
{ true }
        seq
         assign max x1;
         { (x1 <= max)};
         if (max < x2) then
                 assign max x2;
         { (( x1 <= max ) & (x2 <= max))};
        if ( max < x3 ) then
                 assign max x3;
        :
{ (( x1 ≤ max) & ((x2 ≤ max ) & ( x3 ≤ max))) }
```

Condițiile de verificare pentru acest algoritm sunt:

$((((x1 \le max) \& (x2 \le max)) \& (max < x3)) \Rightarrow$

$((x1 \le x3) \& ((x2 \le x3) \& (x3 \le x3))))$

$((((x1 \le max) \& (x2 \le max)) \& (!(max < x3))) \Rightarrow$
$((x1 \le max) \& ((x2 \le max) \& (x3 \le max))))$

$(((x1 \le max) \& (max < x2)) \Rightarrow ((x1 \le x2) \& (x2 \le x2)))$

$(((x1 \le max) \& (!(max < x2))) \Rightarrow ((x1 \le max) \& (x2 \le max)))$

$(\text{true} \Rightarrow (x1 \le x1))$

Din motive de spațiu nu s-au inclus demonstrațiile pentru fiecare condiție de verificare.

7. Concluzii

Partea cea mai importantă din verificator este demonstratorul de teoreme. Metoda de proiectare a demonstratorului este simplă comparativ cu metodele ce se bazează pe principiul rezoluției [5]), fapt care conferă demonstratorului o flexibilitate mai mare prin facilitatea implementării unor diverse sisteme logice. Numărul axiomelor cu care este înzestrat demonstratorul se poate mări astfel ca acesta să poată demonstra un număr cât mai mare de teoreme (se pot introduce axiome care să cuprindă toți operatorii matematici). Demonstratorul de teoreme care s-a construit generează verificări pentru algoritmi de mărimea celor prezentați în exemple într-un timp de ordinul minutelor pe o mașină IBM și un timp de ordinul zecilor de secunde pe o mașină RS6000.

Pentru a face un pas în plus în asigurarea corectitudinii programelor, pe lângă verificatorul de programe se poate construi un verificator de demonstrații, care să verifice corectitudinea demonstrațiilor generate de verificatorul de programe. Având în vedere faptul că verificarea unei demonstrații constă doar din operații mecanice, probabilitatea strecurării unei greșeli în implementare scade foarte mult.
Verificarea automată a programelor constituie o unealtă foarte utilă atunci când se dorește realizarea unui algoritm corect.

Bibliografie

1. **DIJKSTRA, E.W.:** A Discipline of Programming, Prentice Hall, Englewood Cliffs, N.J., 1976.
2. **LEROY, X.:** The Caml Light System Release 0.6. Documentation and User's Manual, September 1993.
3. **MAUNY, M:** Functional Programming Using Caml Light, September 1993.
4. **GORDON, J.C.:** Programming Language Theory and its Implementation, Prentice Hall, 1988.
5. **PAULSON, L.:** Ml for the Working Programmer, Cambridge University Press, Cambridge, 1991.

NOTĂ

Cei ce doresc să obțină o copie a verificatorului (inclusiv sursele) pot face o cerere la adresa de email : hvoicu@tuiasi.ro

Lista de simboluri folosite în text :

$\vee$	operatorul și
$\wedge$	operatorul sau
$\neg$	operatorul not
$\Rightarrow$	operatorul implică
$\Leftarrow$	operatorul implică
$\Leftrightarrow$	operatorul echivalență
$\vdash$	operatorul aserțiune
■	semnifică sfârșitul unui enunț (definiție, teoremă, etc)