Dezvoltare de algoritmi corecţi din specificaţii

Elemente de teorie

• Notatie: prin [c1, c2] este notata o propozitie Pseudocod nestandard in care executia porneste dintr-o stare in care preconditia **c1** este adevarat si se termina intr-o stare in care postconditia **c2** este adevarata
• Daca η este un predicat, atunci prin η[x|e] se noteaza rezultatul substituirii in predicatul η a variabilelor x cu expresiile e
• Daca variabila v apare si in preconditia si in postconditia unei parti de algoritm, in postconditie ea este notata prin v' pentru a sublinia ca are alta valoare dupa executia acestei parti de algoritm
• Reguli de rafinare

R1:regula atribuirii:
[c1, c2] poate fi rafinat prin
v←exp
daca c2 devine adevarata dupa atribuirea v←exp, sau formal
c1 → c2(v | exp)
(a se citi "c1 implică c2 devine adevărată după substituirea lui v prin exp")

R2: regula compunerii secventiale: [c1, c2] este rafinat la
[c1, mijloc]
[mijloc, c2] pentru orice conditie **mijloc**

R3: regula alternantei (DACA)
Fie c o conditie (expresie booleana) si ¬ b negatia sa. Atunci
[c1, c2] poate fi rafinat la
DACA c
atunci [c ∧ c1, c2]
altfel [¬ c ∧ c1, c2]
SFDACA

R4: regula iteratiei: [c1, c1 şi ¬ c] este rafinat la
câttimp c execută
[c1 şi c, c1 şi cond.de terminare]
sfcât
{aici ¬ c este adevarat}
c este o conditie (expresie booleana)

Exemplu 1

Inserare

Fie A = (a_1, a_2,...,a_n) un vector cu n componente ordonate nedescrescător şi fie x o valoare arbitrară. Inseraţi în vectorul A pe o poziţie potrivită pe x astfel ca A să rămână ordonat şi să conţină o nouă componentă egală cu x.

Fie ORD următorul predicat:

ORD(n,A) ::= ($\forall i,j: 1 \le i,j \le n,\ i \le j \Rightarrow a_i \le a_j$)

care este adevărat când componentele vectorului A sunt ordonate nedescrescător.

φ::= ORD(n,A) $\wedge$ (n natural)
ψ ::=ORD(n+1,A) şi (A conţine componentele iniţiale plus o componentă egală cu x)

Prin

A_0:	[φ, ψ]

notăm un algoritm abstract care cere inserarea lui x în vectorul ordonat A cu păstrarea ordinei.

Deoarece sunt 2 posibilitati, ($x<a_n$ şi $n\neq 0$) sau ($x\geq a_n$ sau ($n=0$))avem

A_1:	Subalgoritmul Insert(n,A,x) este: Dacă $x<a_n$ şi $n\neq 0$ atunci [$\varphi \wedge (x<a_n) \wedge n\neq 0, \psi$] altfel [$\varphi \wedge ((x\geq a_n)\vee(n=0)), \psi$] sfdacă sf-Insert

A doua propoziţie nestandard se rafinează printr-o atribuire

A_2:	Subalgoritmul Insert(n,A,x) este: Dacă $x<a_n$ şi $n\neq 0$ atunci [$\varphi \wedge (x<a_n) \wedge n\neq 0, \psi$] altfel $(n,a_{n+1}) \leftarrow (n+1,x)$ sfdacă sf-Insert

Să notăm prin η următorul predicat

$$ORD(n,A) \wedge [(x<a_1) \wedge (p=1) \vee (a_{p-1}\leq x<a_p) \wedge (1<p\leq n)]$$

Care este o postconditie pentru o problemă de căutare şi să folosim regula secvenţei. Ajungem la:

A_3:	Subalgoritmul Insert(n,A,x) este: Dacă $x<a_n$ şi $n\neq 0$ atunci [$\varphi \wedge (x<a_n) \wedge n\neq 0, \eta$] [$\eta$, ψ] altfel $(n,a_{n+1}) \leftarrow (n+1,x)$ sfdacă sf-Insert

Vom satisface postcondiţia η în urma apelului subalgoritmului de căutare, astfel că ajungem la:

A_4:	Subalgoritmul Insert(n,A,x) este:
	  Dacă $x<a_n$ şi $n\neq 0$
	  atunci Cheamă SEARCH(x,n,A,p)
	  [η , ψ]
	  altfel $(n,a_{n+1}) \leftarrow (n+1,x)$
	  sfdacă
	sf-Insert

In urma căutării ştim că x se află între a_{p-1} şi a_p astfel că x trebuie inserat pe poziţia p, deci

$a'_{i+1} \leftarrow a_i$, pentru i=n,n-1,...,p

iar $a'_p \leftarrow x$.

$n' \leftarrow n+1$

Realizăm aceasta prin atribuirile

i←n;
Câttimp i≥p execută
$a_{i+1} \leftarrow a_i$
i←i-1
sfcât

astfel că ajungem la versiunea finală:

A_5:	Subalgoritmul Insert(n,A,x) este:
	  Dacă $x<a_n$ şi $n\neq 0$
	  atunci Cheamă SEARCH(x,n,A,p)
	  i←n
	  Câttimp i≥p execută {i scade}
	  $a_{i+1} \leftarrow a_i$
	  i←i-1
	  sfcât
	  $a_p \leftarrow x$
	  n←n+1
	  altfel $(n,a_{n+1}) \leftarrow (n+1,x)$
	  sfdacă
	sf-Insert

Mai putem rafina acest algoritm observând că atribuirea n←n+1 apare pe ambele ramuri ale propoziţiei DACA, astfel că o putem scrie o singură dată după această propoziţie. Obţinem

A_5:	Subalgoritmul Insert(n,A,x) este: Dacă $x<a_n$ şi $n\neq 0$ atunci Cheamă SEARCH(x,n,A,p) $i \leftarrow n$ Câttimp $i \geq p$ execută $a_{i+1} \leftarrow a_i$ $i \leftarrow i-1$ sfcât $a_p \leftarrow x$ altfel $a_{n+1} \leftarrow x$ sfdacă Fie $n \leftarrow n+1$ sf-Insert

Exemplu 2

Sortare prin Inserare

Fie $A = (a_1, a_2,...,a_n)$ un vector cu n componente întregi . Se cere să ordonăm crescător componentele vectorului A, deci să scriem un algoritm pentru următoarea specificatie:

$\varphi ::= n \geq 2$, A are elemente numere intregi
$\psi ::=$ ORD(n,A) şi A are aceleaşi elemente ca la început
deci

A_0:	$[\varphi, \psi]$

Folosind regula secvenţei şi observând că

$$ORD(k,A) \wedge (k=n) \Rightarrow \psi$$

obţinem

A_1:	Subalgoritmul InsertSort(n,A) este: $[\varphi$, ORD(k,A)] [ORD(k,A), ORD(k, A) şi (n=k)] sf-InsertSort

Prima propoziţie nestandard se rafinează la atribuirea $k \leftarrow 1$, astfel că obţinem

A_2:	Subalgoritmul InsertSort(n,A) este: Fie $k \leftarrow 1$ [ORD(k,A), ORD(k, A) şi (n=k)] sf-InsertSort

Propoziţia nestandard rămasă se rafinează cu regula iteraţiei şi obţinem:

A_3:	Subalgoritmul InsertSort(n,A) este: Fie k←1 Câttimp k<n execută [ORD(k,A) si k<n, ORD(k,A) şi θ] sfcât sf-InsertSort

Pentru a asigura terminarea ciclului condiţia θ k trebuie sa creasca. O prima posibilitate ar fi k←k+1. Pentru a mentine predicatul η(k)::=ORD(k,A) invariant trebuie ca prin inlocuirea lui k cu k+1 predicatul η(k|k+1) sa fie adevarat. Ajungem la versiunea

A_4:	Subalgoritmul InsertSort(n,A) este: Fie k←1 Câttimp k<n execută [k<n şi η(k), η(k|k+1)] sfcât sf-InsertSort

Să observăm că propoziţia nestandard

$$[\ (k<n) \wedge \ ORD(k,A),\ ORD(k+1,A)]$$

correspunde următoarei probleme, care pentru noi devine o subproblemă:

Dacă ORD(k,A) (deci primele k elemente ale vectorului A sunt ordonate) atunci faceţi ca primele k+1 elemente sfie ordonate. Aceasta se poate face prin apelul unui subalgoritm de inserare a componentei a_{k+1} astfel ca după inserare să fie adevată postcondiţia ORD(k+1,A).

A_4:	Subalgoritmul InsertSort(n,A) este: Fie k←1 Câttimp k<n execută {n-k scade} Cheamă INSERT(k,A, a_{k+1}) sfcât sf-InsertSort

Exemplu 3

Impartirea întreagă prin scăderi repetate

Specificare

φ: $(x\geq 0) \wedge (y>0)$

ψ: $(x=q*y+r) \ \wedge \ (0\leq r<y)$

A_0:	Subalgoritmul CATREST(x,y,q,r) este: [φ, ψ] sf-CATREST

Să considerăm următorul predicat

η: $(x=q*y+r) \ \wedge (0\leq r)$

şi să folosim regula secvenţei R1:

A_1:	Subalgoritmul CATREST(x,y,q,r) este: [φ, η] [η, ψ] sf-CATREST

Intrucât η devine adevărat prin atribuirea (q,r):= (0,x), iar $(r<y) \wedge \eta$ implică ψ, avem:

A_2:	Subalgoritmul CATREST(x,y,q,r) este: Fie (q,r)←(0,x) [η, η şi r<y] sf-CATREST

Observăm că η este predicat invariant şi putem folosi regula iteraţiei, pentru a ajunge la

A_3:	Subalgoritmul CATREST(x,y,q,r) este: Fie (q,r) ← (0,x) Câttimp r≥y execută [r≥y şi η, η şi r scade] sfcât sf-CATREST

Observăm că terminarea execuţiei ciclului se obţine dacă r scade, ceea ce se poate obţine prin atribuirea r←r-y întrucât r≥y. Pentru ca invariantul η să rămână adevărat va trebui să schimbăm şi pe q. Intrucât

$$q*y+r = (q+1)*y + (r-y)$$

o soluţie posibilă este atribuirea (q,r) ← (q+1, r-y), astfel că ajungem la versiunea finală:

A_3:	Subalgoritmul CATREST(x,y,q,r) este: Fie (q,r) ← (0,x) Câttimp r≥y execută {r scade} (q,r) ← (q+1,r-y) sfcât sf-CATREST

Exemplu 4

Cel mai mare divizor comun

In cele ce urmează vom nota prin cmmdc(m,n) cel mai mare divizor comun al numerelor m şi n.

Specificarea problemei:

φ : x>0, y > 0

ψ : d=cmmdc(x,y)

A_0

Subalgoritmul CMMDC(x,y,d) este:
[φ, ψ]
sf-CMMDC

Predicatul

η::= cmmdc(d,s)=cmmdc(x,y)

poate deveni adevarat pentru (d,s)=(x,y), si daca d=s atunci η implica ψ. Ca urmare, o versiune corecta a algoritmului este:

A_1

```
Subalgoritmul CMMDC(x,y,d) este:
        [φ, η]
        [η,η ∧ (d=s)]
sf-CMMDC
```

A_2

```
Subalgoritmul CMMDC(x,y,d) este:
        {φ e adevarat}
        Fie (d,s) ← (x,y)   {η devine adevarat}
        Câttimp d≠s execută
                @Pastreaza η adevarat si asigura terminarea
        sfcât
        {Aici d=s, ca urmare η implica ψ}
        {ψ}
sf-CMMDC
```

Pentru d≠s avem d>s sau d<s. Ca urmare, din matematica stim ca pentru d>s avem

cmmdc(d,s)=cmmdc(d-s,s)

ca urmare atribuirea d ←d-s pastreaza η invariant. Ca urmare, o varianta corecta (finala) a algoritmului este:

A_3

```
Subalgoritmul CMMDC(x,y,d) este:
        {φ e adevarat}
        Fie (d,s) ← (x,y)   {η devine adevarat}
        Câttimp d≠s execută     {s+d scade}
                Dacă d>s
                      atunci d←d-s   {Pastreaza η adevarat}
                       altfel s←s-d     {Pastreaza η adevarat}
                sfdacă
                {d+s descreste, si acest lucru asigura terminarea}
        sfcât
        {Aici d=s, ca urmare η implica ψ}
        {ψ}
sf-CMMDC
```

Exemplu 5

Rădăcina pătrată întreagă.

Se cere să se determine partea întreagă din radical din *n*.

Dacă notăm prin r această valoare, trebuie să avem

$$r \leq \textit{radical din } n < r+1$$

deci predicatul de ieşire este

$$r^2 \leq n < (r+1)^2$$

Specificarea:

φ: n>1

ψ: $r^2 \leq n < (r+1)^2$

A_0

```
Subalgoritmul RADICALINT(n,r) este:
        [φ, ψ]
sf-RADICALINT
```

Pentru a-l găsi pe r îl vom iniţializa cu o anumită valoare cerută de contextul programului şi vom modifica această valoare până când ea va satisface predicatul de ieşire. Întrucât în forma în care se află nu putem ghici valoarea care satisface acest predicat, vom introduce o nouă variabilă care să ne ajute la determinarea unui predicat invariant şi în final a valorii dorite pentru r. Mai exact, predicatul de ieşire va fi adevărat dacă

$(r^2 <= n < q^2) \wedge (q=r+1)$

Predicatul de mai sus nu poate fi satisfăcut direct întrucât e imposibil să "ghicim" valori pentru *q* şi *r* care să satisfacă ambii termeni ai conjuncţiei. E mult mai uşor să satisfacem doar prima paranteză, motiv pentru care predicatul

$\eta ::= (r^2 \leq n < q^2) \wedge (q \geq r+1)$

ne sugerează un invariant pentru programul dorit.

A_1

```
Subalgoritmul RADICALINT(n,r) este:
        [φ, η]
        [η,η ∧ (q=r+1)]
        {ψ}
sf-RADICALINT
```

Observăm că η devine True pentru r=0 şi q=n+1, deci obţinem următoarea variantă:

A_2

```
Subalgoritmul RADICALINT(n,r) este:
        {φ = ...}
        Fie (q,r) ← (n+1,0)
        Câttimp q>r+1 execută
            @Pastreaza η adev si asigura terminarea micsorand diferenta q-r
        sfcât
        {Aici q=r+1, ca urmare η implica ψ}
        {ψ}
sf-RADICALINT
```

Propoziţia nestandard din ciclu cere ca invariantul η să rămână invariant, iar condiţia de terminare fiind q=r+1 sugerează funcţia de terminare t=q-r, care în final q-r trebuie să devină 1.

Intrucât valoarea p=(q+r)/2 satisface inegalitatea r<p<q, diferenţa q-r se înjumătăţeşte dacă schimbăm intervalul [r,q] cu unul din subintervalele [r,p], respectiv [p,q], ceea ce ar asigura terminarea.

Pentru a pastra η invariant avem nevoie sa stim daca $(p^2 \leq n)$ sau $(p^2 > n)$. Daca $(p^2 \leq n)$ atunci atribuirea $r \leftarrow p$ pastreaza η invariant. Daca $(p^2 > n)$ atunci atribuirea $q \leftarrow p$ pastreaza η invariant. Ca urmare algoritmul este:

A_3

```
Subalgoritmul RADICALINT(n,r) este:
        {φ = ...}
        Fie (q,r) ← (n+1,0)
        Câttimp q>r+1 execută  {q-r scade}
                Fie p← (q+r)/2
                Dacă  p²≤n atunci  r←p
                                altfel  q←p
                sfdacă   {η ramane adev si q-r descreste}
        sfcât
        {Aici q=r+1, ca urmare η implica ψ}
        {ψ}
sf-RADICALINT
```

Exemplu 6

Ridicarea la putere prin inmultiri repetate (rapide)

Calculati $z = x^y$ prin inmultiri repetate.

Specificare:

A0:	φ : $(x>0) \wedge (y \geq 0)$ ψ : $z = x^y$

Observam ca predicatul

$\eta ::= (z*u^v = x^y) \wedge (v \geq 0)$

implica ψ daca v=0. Folosindu-l ca predicat de mijloc (invariant), putem aplica regula compunerii secventiale si obtinem versiunea:

A1:	$[\varphi, \eta]$ $[\eta, \psi]$

Deoarece η e adevarat daca (z,u,v) = (1,x,y) prima linie trebuie inlocuita cu o atribuire multipla. Ca urmare, urmatoarea versiune corecta a algoritmului, definit sub forma unui subalgoritm, este:

A2:	Subalgoritmul Putere(x,y,z) este: {A: φ is true} (z,u,v) ← (1,x,y) $[\eta, \psi]$ sf-Putere

In propozitia nestandard $[\eta, \psi]$ predicatul η este un invariant, a.i. este rafinat la o structura

iterativa Cattimp:

A3:	Subalgoritmul Putere(x,y,z) este: {A: φ is true} (z,u,v) ← (1,x,y) Câttimp v≠0 execută {B: η is true} @Păstrează η invariant şi asigură terminarea ciclului sfcâtimp {C: ψ is true} sf-Putere

Pentru a asigura iesirea din ciclu, adica v sa devina 0, v trebuie sa descreasca. Cel mai simplu mod de a asifura acest lucru este prin atribuirea v←v-1. Deoarece $z*u^v = z * u*u^{v-1}$ atribuirea (z,v)←(z*u,v-1) pastreaza η invariant. Obtinem urmatorea versiune corecta:

A4:	Subalgoritmul Putere(x,y,z) este: {A: φ e adevarata} (z,u,v) ← (1,x,y) Câttimp v≠0 execută {B: η e adevarat} (z,v) ← (z*u,v-1) sfcâtimp {C: ψ e adevarat} sf-Putere

Urmatoarea posibilitate de a descreste v ar fi, daca v este par, atribuirea v←v div 2. Deoarece $z*u^v = z* (u*u)^{v/2}$ e nevoie de atribuirea (u,v) ← (u*u,v/2) pentru a pastra η invariant, in acest caz. Ca urmare, obtinem a doua varianta:

```
Subalgoritmul Putere2(x,y,z) este:
        {A: φ e adevarat}
        (z,u,v) ← (1,x,y)
        Câttimp v≠0 execută
                Câttimp v is even execută        {v scade}
                        (u,v) ← (u*u,v/2)   {η is true}
                sfcât {Aici v e impar si il putem descreste cu 1}
                (z,v) ← (z*u,v-1)        {η e adevarat}
        sfcâtimp
        {C: ψ e adevarat}
sf-Putere2
```

Exemplu 7

Inmulţire prin adunări repetate

Specificare:

φ : $(x \geq 0) \wedge (y \geq 0)$

ψ : $z = x*y$

A_0

Subalgoritmul Produs(x,y,z) este:
[φ, ψ]
sf-Produs

Întrucât postcondiţia ψ nu poate fi satisfacută direct printr-o atribuire vom introduce următorul predicat:

$\eta ::= (z+u*v = x*y) \wedge (v \geq 0)$

Acest predicat devine adevărat prin atribuirea

$(u,v,z) \leftarrow (x,y,0)$

iar

$\eta \wedge (v=0) \rightarrow \psi$

ceea ce ne sugerează următoarea versiune de subalgoritm:

A_1

Subalgoritmul Produs(x,y,z) este:
[φ, η]
[η,η ∧ (v=0)]
{ψ}
sf-Produs

Folosind regula iteratiei, obtinem

A_2

```
Subalgoritmul Produs(x,y,z) este:
    (z,u,v) ← (0,x,y)                    {η e adevărat}
    Câttimp v≠0 execută
      @Păstrează z+u*v = x*y invariant adevărat şi micşorează pe v
    sfcât {aici v=0 şi η e adevărat, deci ψ va fi adevărat}
    {ψ}
sf-Produs
```

Există două posibilităţi de a-l micşora pe v. Prima, cu valoarea 1 prin atribuirea v:=v-1. In acest caz, pentru a-l părstra pe η adevărat este necesară şi atribuirea z:=z+u deoarece

$z+u*v = z + u + u*(v-1)$

A doua se poate aplica numai pentru v par când v poate fi împărţit cu 2. Cum

$z+u*v = z + (u*2)* v/2$

rezultă că η rămâne adevărat la atribuirea (u,v):=(u+u, v/2). Ajungem la varianta finală

A_3

```
Subalgoritmul Produs(x,y,z) este:
    (z,u,v) ← (0,x,y)                    {η e adevărat}
    Câttimp v>0 execută      {v scade}
        Câttimp v e par execută
            (u,v) ← (u+u,v div 2)
        sfcât          { z+u*v = x*y e invariant}
```

```
            {aici v e impar}
            (z,v) ← ( z+u, v-1)
        sfcât {aici v=0 şi η e adevărat, deci ψ va fi adevărat}
        {ψ}
sf-Produs
```

Exemplu 8

Cautarea pozitiei de inserare a unui element x intr-un sir A ordonat crescator.

$\varphi ::= ORD(n,A) \wedge n \geq 1$

$\psi ::= [(x<a_1) \wedge (p=1)] \vee [(a_{p-1} \leq x < a_p) \wedge (1<p\leq n)] \vee [(x \geq a_n) \wedge (p=n+1)]$