Capitolul 2

Verificarea corectitudinii algoritmilor

Analiza unui algoritm presupune verificarea câtorva caracteristici ale acestuia: *corectitudine*, *eficienţă*, *simplitate*, *claritate*. Dintre acestea primele două pot fi evaluate în manieră obiectivă pe când ultimele au şi o componentă subiectivă care este mai dificil de evaluat. În acest capitol este prezentată o introducere în verificarea corectitudinii unui algoritm iar în capitolul următor este prezentat modul în care poate fi evaluată eficienţa unui algoritm.

2.1 Etapele verificării corectitudinii

Un algoritm este considerat corect dacă permite obţinerea rezultatului problemei pornind de la datele iniţiale ale acesteia. Pentru a obţine informaţii privind abilitatea unui algoritm de a rezolva problema pentru care a fost proiectat se poate alege una dintre următoarele variante:

- *Experimentală.* Se testează algoritmul pentru diverse instanţe ale problemei. Principalul avantaj al acestei abordări îl reprezintă faptul că nu necesită tehnici speciale pentru a fi aplicată pe când principalul dezavantaj îl reprezintă faptul că testarea nu poate acoperi întotdeauna toate variantele posibile de date de intrare. Varianta experimentală permite uneori identificarea situaţiilor pentru care algoritmul nu funcţionează corect însă nu garantează întotdeauna corectitudinea.

- *Analitică.* Se demonstrează că algoritmul funcţionează corect pentru orice date de intrare. Principalul avantaj îl reprezintă faptul că este garantată corectitudinea pentru orice date de intrare. Dezavantajul este reprezentat de faptul că uneori este dificil de găsit o demonstraţie. Abordarea analitică poate însă conduce la o mai bună înţelegere a algoritmului

şi la identificarea porţiunilor ce conţin eventuale erori. Demonstrarea corectitudinii poate fi dificilă în cazul algoritmilor complecşi. În această situaţie algoritmul se descompune în subalgoritmi şi se demonstrează pentru fiecare în parte corectitudinea.

În verificarea analitică a corectitudinii se parcurg următoarele etape principale:

- Identificarea proprietăţilor datelor de intrare (*precondiţiile problemei*).
- Identificarea proprietăţilor pe care trebuie să le satisfacă datele de ieşire (*postcondiţiile problemei*).
- Demonstrarea faptului că pornind de la precondiţii şi aplicând prelucrările specificate în algoritm se ajunge la satisfacerea postcondiţiilor.

În analiza corectitudinii este utilă noţiunea de *stare* a unui algoritm înţeleasă ca ansamblul valorilor pe care le au variabilele prelucrate în cadrul algoritmului la un moment dat (pas al prelucrării). Ideea verificării este de a stabili care trebuie să fie starea la fiecare moment astfel încât în final să fie satisfăcute postcondiţiile. O dată stabilite aceste stări intermediare este suficient să se verifice că fiecare prelucrare asigură transformarea stării care o precede în cea care o succede. Starea algoritmului este modificată prin atribuiri de valori variabilelor. În cazul prelucrărilor secvenţiale (succesiuni de atribuiri), verificarea este în general simplă constând în a analiza succesiv efectul fiecărei atribuiri asupra stării algoritmului. Principala sursă de erori o reprezintă prelucrările repetitive pentru care există riscul de a specifica în mod greşit iniţializările, prelucrările din corpul ciclului sau condiţia de oprire (continuare). Demonstrarea corectitudinii unei prelucrări repetitive se bazează pe principiul inducţiei matematice.
Exemplu. Considerăm următorul algoritm, care determină minimul unui şir finit de numere reale:

```
minim(real a[1..n])
  min ← a[1]
  for i ← 2, n do
    if min > a[i] then
      min ← a[i]
    end if
  end for
  return min
```

Enunţul problemei nu impune nici o restricţie asupra tabloului $a[1..n]$ astfel că setul de precondiţii este constituit doar din $\{n \geq 1\}$ (şirul nu este vid). Postcondiţia este reprezentată de proprietatea valorii minime: $min \leq a[i]$ pentru orice $i = \overline{1, n}$. Rămâne să arătăm că postcondiţia rezultă în urma aplicării

algoritmului. Demonstrăm prin inducţie matematică după i că $min \leq a[i]$, $i = \overline{1,n}$. Cum $min = a[1]$ şi valoarea lui min este înlocuită doar cu una mai mică rezultă că $min \leq a[1]$. Presupunem că $min \leq a[k]$ pentru orice $k = \overline{1,i}$. Rămâne să arătăm că $min \leq a[i+1]$. La pasul i al ciclului **for** valoarea lui min se modifică astfel:

- Dacă $min \leq a[i+1]$ atunci min rămâne nemodificat.
- Dacă $min > a[i+1]$ atunci min se înlocuieşte cu $a[i+1]$.

Astfel în oricare dintre variante se obţine că $min \leq a[i+1]$. Conform metodei inducţiei matematice rezultă că postcondiţia este satisfăcută, deci, în ipoteza că se ajunge la un rezultat acesta va fi corect. Să considerăm acum algoritmul:

```
minim(real a[1..n])
  for i ← 1, n − 1 do
    if a[i] < a[i + 1] then
      min ← a[i]
    else
      min ← a[i + 1]
    end if
  end for
  return min
```

În acest caz prelucrarea din corpul ciclului conduce la $min = \min\{a[i], a[i+1]\}$ astfel că nu se mai poate demonstra pornind de la $min = \min\{a[1..i]\}$ că $min = \min\{a[1..i+1]\}$. Dealtfel este uşor de găsit un contraexemplu (de exemplu şirul $(2, 1, 3, 5, 4)$) pentru care algoritmul de mai sus nu funcţionează corect. În schimb se poate demonstra că algoritmul returnează $\min\{a[n-1], a[n]\}$.

2.2 Elemente de analiză formală a corectitudinii

Pentru verificarea analitică a corectitudinii algoritmilor există un metode formale bazate pe logica Floyd-Hoare. Aceste metode folosesc următoarea strategie:

- pe baza precondiţiilor şi postcondiţiilor problemei se stabilesc pentru fiecare prelucrare *aserţiuni* privind valorile datelor şi relaţiile dintre ele;
- pentru fiecare prelucrare se demonstrează că asigură satisfacerea aserţiunii care o urmează dacă este satisfăcută aserţiunea care o precede.

O noţiune importantă în acest context este cea de *triplet Hoare* definit ca fiind $\langle P, A, Q\rangle$, unde P reprezintă precondiţiile problemei, A algoritmul iar Q postcondiţiile. Se spune că tripletul $\langle P, A, Q\rangle$ reprezintă un algoritm corect, şi se notează acest lucru cu $P \xrightarrow{A} Q$, dacă următoarea afirmaţie este adevărată:

> Dacă datele problemei satisfac precondiţiile P atunci:
> (i) rezultatul satisface postcondiţiile Q;
> (ii) algoritmul A se termină după un număr finit de paşi.

Dacă algoritmul satisface condiţia (i) însă nu s-a demonstrat că satisface şi condiţia (ii) (numită condiţie de terminare) atunci este denumit algoritm *parţial corect*. În cazul în care se demonstrează ambele proprietăţi atunci algoritmul este considerat *complet corect*.

Verificarea corectitudinii complete a unui algoritm este un demers dificil în principal datorită dificultăţii demonstrării faptului că algoritmul se termină. Adesea demonstrarea terminării unui algoritm este echivalentă cu rezolvarea unor probleme încă deschise în matematică. De exemplu este relativ uşor de proiectat şi de demonstrat corectitudinea parţială a unui algoritm care caută un număr perfect impar (un număr natural este denumit perfect dacă coincide cu suma divizorilor săi proprii). Faptul că algoritmul se termină nu poate fi demonstrat la ora actuală, atât timp cât încă nu se ştie dacă un astfel de număr există.

La rândul ei, verificarea corectitudinii parţiale poate fi dificilă în cazul algoritmilor care implică multe prelucrări. În astfel de situaţii se realizează verificarea pe porţiuni până se ajunge la nivelul structurilor fundamentale de prelucrare (secvenţială, alternativă, repetitivă). Pentru fiecare dintre aceste structuri de prelucrare există reguli care uşurează verificarea.

Regula structurii secvenţiale. Fie algoritmul A constituit din succesiunea de prelucrări $(A_1, A_2, \ldots, A_n)$. Notăm cu P_{i-1} şi P_i starea algoritmului înainte şi respectiv după efectuarea prelucrării A_i. Regula structurii secvenţiale poate fi enunţată astfel:

> Dacă $P \Rightarrow P_0$, $P_{i-1} \xrightarrow{A_i} P_i$ pentru $i = \overline{1, n}$ şi $P_n \Rightarrow Q$ atunci $P \xrightarrow{A} Q$

Această regulă afirmă că dacă precondiţia implică starea iniţială, fiecare prelucrare din secvenţă conduce de la aserţiunea care o precede la cea care o succede, iar ultima aserţiune implică postcondiţia atunci secvenţa este corectă.
Exemplu. Fie a şi b două valori, iar x şi y două variabile ce conţin valorile a şi b. Să se interschimbe valorile celor două variabile.

Precondiţiile problemei sunt: $P = \{x = a, y = b\}$ iar postcondiţiile sunt $Q = \{x = b, y = a\}$. Considerăm următoarele prelucrări:

$x \leftrightarrow y$	
$A_1:$	$aux \leftarrow x$
$A_2:$	$x \leftarrow y$
$A_3:$	$y \leftarrow aux$

Aserţiunile privind stările algoritmului sunt următoarele: $P_0 = \{x = a, y = b\}$ (înainte de A_1), $P_1 = \{aux = a, x = a, y = b\}$ (după A_1), $P_2 = \{aux = a, x = b, y = b\}$ (după A_2), $P_3 = \{aux = a, x = b, y = a\}$ (după A_3). Este uşor de remarcat că $P = P_0$, $P_3 \Rightarrow Q$ şi $P_{i-1} \xrightarrow{A_i} P_i$ pentru $i = \overline{1,3}$. Conform regulii structurii secvenţiale rezultă că secvenţa realizează corect interschimbarea valorilor celor două variabile. Dacă însă se consideră secvenţa de prelucrări:

$A_1:$	$x \leftarrow y$
$A_2:$	$y \leftarrow x$

aserţiunile corespunzătoare sunt: $\{x = b, y = b\}$ (atât după A_1 cât şi după A_2). În acest caz A_1 şi A_2 nu pot conduce la satisfacerea postcondiţiilor.

Aceeaşi problemă poate fi rezolvată, în cazul variabilelor de tip numeric, fără a folosi o variabilă auxiliară efectuând prelucrările:

$A_1:$	$x \leftarrow x - y$
$A_2:$	$y \leftarrow x + y$
$A_3:$	$x \leftarrow y - x$

Stările algoritmului sunt următoarele: $P_0 = \{x = a, y = b\}$, $P_1 = \{x = a - b, y = b\}$, $P_2 = \{x = a - b, y = a\}$, $P_3 = \{x = b, y = a\}$. Este uşor de remarcat că $P = P_0$, $P_3 = Q$ şi $P_{i-1} \xrightarrow{A_i} P_i$ pentru $i = \overline{1,3}$.

Uneori sunt utile şi următoarele reguli:

Întărirea precondiţiei. Dacă $R \Rightarrow P$ şi $P \xrightarrow{A} Q$ atunci $R \xrightarrow{A} Q$ (algoritmul rămâne corect dacă se particularizează datele problemei).

Slăbirea postcondiţiei. Dacă $P \xrightarrow{A} Q$ şi $Q \Rightarrow R$ atunci $P \xrightarrow{A} R$ (algoritmul rămâne corect dacă se restrâng cerinţele problemei).

Proprietăţi însoţitoare. Dacă $P_1 \xrightarrow{A} Q_1$ şi $P_2 \xrightarrow{A} Q_2$ atunci $P_1 \wedge P_2 \xrightarrow{A} Q_1 \wedge Q_2$.

Regula structurii alternative. Considerăm structura:

`S:` **if** c **then** A_1 **else** A_2 **end if**

Dacă P şi Q sunt precondiţiile respectiv postcondiţiile atunci regula poate fi enunţată în modul următor:

Dacă c este bine definită (poate fi evaluată), $P \wedge c \xrightarrow{A_1} Q$ şi $P \wedge \overline{c} \xrightarrow{A_2} Q$ atunci $P \xrightarrow{A} Q$.

Regula sugerează că trebuie verificată corectitudinea fiecărei ramuri (atât când c este adevărată cât şi în cazul în care este falsă).
Exemplu. Considerăm problema determinării minimului dintre două valori reale, distincte:

---

if $a < b$ **then**
  $A_1 :\quad m \leftarrow a \qquad \{\, a < b,\, m = a\}$
else
  $A_2 :\quad m \leftarrow b \qquad \{\, a \geq b,\, m = b\}$
end if

Precondiţiile sunt $P = \{a \in \mathbb{R},\ b \in \mathbb{R},\ a \neq b\}$ iar postcondiţia este $Q = \{m = \min\{a, b\}\}$. Condiţia c este $a < b$. Dacă $a < b$ atunci $m = a < b$ deci $m = \min\{a, b\}$. În caz contrar este adevărată relaţia $a > b$ iar prelucrarea A_2 conduce la $m = b < a$ deci din nou $m = \min\{a, b\}$.

Regula structurii repetitive. Întrucât toate structurile repetitive pot fi reorganizate astfel încât să conţină o structură condiţionată anterior (de tip **while**) o vom analiza doar pe aceasta. Considerăm structura:
S: **while** c **do** A **end while**,

precondiţia P şi postcondiţia Q. In cazul prelucrărilor repetitive intervine problema terminării algoritmului astfel că trebuie analizate două aspecte: faptul că algoritmul conduce la postcondiţie în cazul în care se termină şi faptul că algoritmul se termină după un număr finit de prelucrări. Pentru a demonstra că o structura repetitivă de tip **while** este corectă este suficient să se arate că există o aserţiune I (numită *invariant* sau *proprietate invariantă* a prelucrării repetitive) asociată stării algoritmului şi o funcţie $t : \mathbb{N} \to \mathbb{N}$ (numită *funcţie de terminare* şi care depinde de numărul de ordine al iteraţiei, notat în continuare cu p) care satisfac proprietăţile:

(a) Aserţiunea I este adevărată la începutul prelucrării repetitive ($P \Rightarrow I$).

(b) Aserţiunea I este *invariantă*: dacă I este adevărată înainte de efectuarea prelucrării A iar c este adevărată atunci I rămâne adevărată şi după efectuarea lui A ($I \wedge c \xrightarrow{A} I$).

(c) La sfârşitul structurii repetitive (când c devine falsă) postcondiţia Q poate fi dedusă din I ($I \wedge \overline{c} \Rightarrow Q$).

(d) După efectuarea prelucrării A valoarea lui t descreşte ($c \wedge (t(p) = k) \xrightarrow{A} t(p+1) < k$)

(e) Dacă c este adevărată atunci $t(p) \geq 1$ (mai există cel puţin o iteraţie) iar când valoarea lui t este 0 condiţia c devine falsă (algoritmul se termină după un număr finit de iteraţii).

Elementul esenţial în demonstrarea corectitudinii unei prelucrări repetitive îl reprezintă găsirea proprietăţii invariante. Aceasta este o aserţiune particulară privind relaţiile dintre variabilele algoritmului care este adevărată înainte de intrarea in prelucrarea repetitivă, rămâne adevărată după fiecare iteraţie iar când condiţia c devine falsă implică postcondiţiile. Găsirea unui invariant elimină necesitatea demonstraţiei explicite prin inducţie matematică. Din perspectiva demonstrării corectitudinii condiţia cea mai importantă pe care trebuie să o satisfacă un invariant este să implice postcondiţia la sfârşitul prelucrării repetitive.

Dacă se identifică invariantul unei prelucrări repetitive atunci aceasta este cel puţin parţial corectă. Identificarea unei funcţii de terminare este fie foarte simplă (în special în cazul prelucrărilor repetitive ce folosesc un contor) sau foarte dificilă.

Exemplul 2.1 Reluăm problema determinării minimului dintr-un şir finit de valori, rescrisă de această dată folosind **while** şi analizăm algoritmul corespunzător.

`minim`(**real** $a[1..n]$)		
A_1 :	$min \leftarrow a[1]$	$\{\ min = a[1]\ \}$
A_2 :	$i \leftarrow 2$	$\{\ min = a[1],\ i = 2\ \}$
	while $i \leq n$ **do**	
A_3 :	**if** $min > a[i]$	
	then $min \leftarrow a[i]$	
	end if	$\{\ min \leq a[j],\ j = \overline{1,i}\}$
A_4 :	$i \leftarrow i + 1$	$\{\ min \leq a[j],\ j = \overline{1,i-1}\}$
	end while	
	return min	$\{\ i = n+1,\ min \leq a[j],\ j = \overline{1,i-1}\}$

Precondiţia este $P = \{n \geq 1\}$ iar postcondiţia este $P = \{min \leq x[i],\ i = \overline{1,n}\}$. Aserţiunile marcate după fiecare prelucrare sunt uşor de stabilit. Proprietatea invariantă este $I = \{min \leq a[j],\ j = \overline{1,i-1}\}$. Într-adevăr, după prelucrarea A_2 proprietatea este satisfăcută. Din aserţiunea specificată după A_4 rezultă că proprietatea rămâne adevărată după fiecare iteraţie. Condiţia de oprire este $i = n+1$ şi se observă că atunci când este satisfăcută, aserţiunea finală implică postcondiţia.

În acest caz poate fi identificată uşor o funcţie de terminare. Notând cu p contorul iteraţiei aceasta poate fi descrisă prin $t(p) = n + 1 - i_p$ unde i_p este valoarea indicelui i corespunzător iteraţiei p. Cum $i_{p+1} = i_p + 1$ rezultă că

$t(p+1) < t(p)$ şi descreşterea este cu o unitate la fiecare iteraţie. Aceasta înseamnă că după un anumit număr de iteraţii t se anulează. Când $t(p) = 0$ variabila i satisface $i = n+1$ deci condiţia de continuare devine falsă conducând la terminarea prelucrării repetitive.

Exemplul 2.2 Analizăm algoritmul lui Euclid pentru determinarea celui mai mare divizor comun a două numere naturale nenule:

```
cmmdc (integer a, b)
  d ← a
  i ← b
  r ← d MOD i
  while r ≠ 0 do
    d ← i
    i ← r
    r ← d MOD i
  end while
  return i
```

Precondiţiile sunt $P = \{a, b \in \mathbb{N}^*\}$, iar postcondiţia este $Q = \{i = \text{cmmdc}(a, b)\}$. Considerăm proprietatea I ca fiind $\text{cmmdc}(a, b) = \text{cmmdc}(d, i)$ şi funcţia t definită prin $t(p) = r_p$ unde r_p este restul obţinut la iteraţia p. Inainte de intrarea în prelucrarea repetitivă este adevărată aserţiunea $\{d = a, i = b\}$ deci $\text{cmmdc}(a, b) = \text{cmmdc}(d, i)$. Când condiţia de continuare devine falsă $(r = 0)$ se obţine că $\text{cmmdc}(i, r) = i$ adică postcondiţia $\text{cmmdc}(a, b) = i$.

Pentru a arăta că I este o proprietate invariantă este suficient să arătăm că dacă $r \neq 0$ atunci $\text{cmmdc}(d, i) = \text{cmmdc}(i, r)$. Notăm $d_1 = \text{cmmdc}(d, i)$ şi $d_2 = \text{cmmdc}(i, r)$. Pe de altă parte $d = i \cdot q + r$. Se pot verifica uşor următoarele implicaţii:

$$d_1|d \text{ şi } d_1|i \Rightarrow d_1|d - i \cdot q = r, d_1|i \Rightarrow d_1|i \text{ şi } d_1|r \Rightarrow d_1 \leq d_2,$$

$$d_2|i \text{ şi } d_2|r \Rightarrow d_2|i \cdot q + r, d_2|i \Rightarrow d_2|d \text{ şi } d_2|r \Rightarrow d_2 \leq d_1.$$

Rezultă că $d_1 = d_2$, deci proprietatea I este într-adevăr invariantă. Cum funcţia de terminare este chiar valoarea restului, iar acesta este un număr natural care descreşte de la o iteraţie la alta, va ajunge să se anuleze după un anumit număr de iteraţii. Prin urmare este satisfăcută şi condiţia de terminare.

Observaţie. Invarianţii pot fi utilizaţi nu numai pentru a determina corectitudinea unui algoritm ci şi ca instrument de proiectare a algoritmilor, în sensul că identificarea unei proprietăţi invariante poate fi văzută ca o etapă preliminară elaborării unui ciclu. Invariantul poate fi considerat ca o specificaţie

pentru ciclul respectiv şi poate fi utilizat în identificarea prelucrărilor necesare pentru iniţializare, pentru corpul ciclului şi în identificarea condiţiei de oprire (continuare) a ciclului.

Exemplul 2.3 Să considerăm calculul sumei, S, a primelor n numere naturale ($n \geq 1$).

Precondiţia este $P = \{n \geq 1\}$, iar postcondiţia este $Q = \{S = 1 + 2 + ... + n\}$. Întrucât suma va fi calculată adăugând succesiv termenul curent, i, un invariant adecvat ar fi $S = 1+2+\ldots+i$ (exprimă faptul ca la sfârşitul iteraţiei i variabila S conţine suma parţiala până la termenul i inclusiv). Pentru a asigura validitatea acestui invariant termenul curent trebuie pregătit chiar înainte de a fi adăugat. Pe de altă parte pentru a ne asigura că la finalul ciclului postcondiţia este satisfăcută rezultă că la ieşirea din ciclu variabila i trebuie să aibă valoarea n. Astfel condiţia de continuare a ciclului **while** ar trebui să fie $i < n$. Toate aceste observaţii conduc la algoritmul `suma1`.

`suma1` (**integer** n)	`suma2` (**integer** n)
$i \leftarrow 1$	$S \leftarrow 0$
$S \leftarrow 1$	$i \leftarrow 1$
while $i < n$ **do**	**while** $i \leq n$ **do**
$i \leftarrow i + 1$	$S \leftarrow S + i$
$S \leftarrow S + i$	$i \leftarrow i + 1$
end while	**end while**
return S	**return** S

Varianta cea mai frecvent utilizată în practică este cea descrisă în algoritmul `suma2`. Motivaţia provine din faptul că această variantă provine din descrierea clasică bazată pe utilizarea instrucţiunii **for**. Un invariant corespunzător acestei variante este $S = 1 + 2 + ... + (i - 1)$, iar o funcţie de terminare este $t(p) = (n + 1) - i_p$.