26. UTILIZAREA STRUCTURILOR DE DATE ÎN TESTAREA SOFTWARE

26.1 Generatoare de date de test (GDT)

Diversităţii tipurilor de programe le corespunde o diversitate a generatoarelor de date de test. Generatoare date de test sunt programe care au un rol important în automatizarea testării software.

Dacă software lucrează cu fişiere generatorul date de test creează un fişier. Dacă software lucrează cu matrice, GDT generează matricele.

Fiecărui sistem de programe i se asociază o structură de tip graf (arbore), nodurile corespunzând punctelor de test ale structurilor de control.

Astfel secvenţei:

```
if (a>b)
   if (c>d)
     e=1;
   else
     e=2;
else
   if (x<y)
     e=3;
   else
     e=4;
```

îi va corespunde structura de graf din figura 26.1.

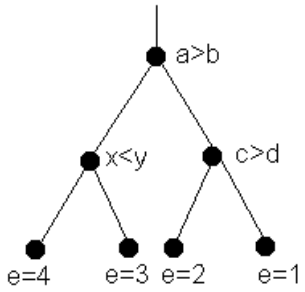

Figura 26.1 *Structură arborescentă asociată secvenţei*

În programe apar instrucţiuni de atribuire şi de control. Astfel programul:

```
void main()
{
 int a=1,b=2,c;
 c=a+b;
 printf("c=%d",c);
}
```

conţine numai expresii de atribuire şi apelarea funcţiei de afişare. Flexibilitatea este limitată, valoarea care se afişează fiind aceeaşi, indiferent de numărul rulărilor.

În acest caz nu se va vorbi de date de intrare, deci nici de date de test.

Testarea programului este simplificată întrucât se urmăreşte comportamentul acestuia pas cu pas numai pentru valorile impuse.

În cazul în care programul (modulul) are valori instrucţiuni de intrare/ieşire sau parametri care modifică nivelele variabilelor efective ale sale, testarea presupune definirea de seturi de date de test (baterii).

În structura arborescentă asociată unui program se identifică următoarele tipuri de noduri:

- un nod rădăcină, corespunzător primei secvenţe (instrucţiuni) din modulul (programul) apelator;
- noduri intermediare, corespunzătoare instrucţiunilor (secvenţelor/modulelor) care sunt urmate şi de alte instrucţiuni (secvenţe/module);
- noduri terminale (frunze), ce corespund instrucţiunilor (secvenţelor/modulelor) care încheie o prelucrare.

Nodul din graful corespunzător programului se asociază unei instrucţiuni, unei secvenţe de instrucţiuni sau unor funcţii, depinzând de gradul de detaliere cu care se doreşte a fi abordat un program.

Bateria de date de test este alcătuită dintr-un set de date care asigură execuţia instrucţiunilor care formează drumul de la nodul rădăcină până la un nod frunză.

26.2 Generator de fişiere de test

Aplicaţiile complexe presupun lucrul cu fişiere. Fişierele sunt formate din articole a căror structură conţine câmpuri.

În multe situaţii, proiectarea de fişiere are la bază date din evidenţele contabile ale societăţilor comerciale. Forma concretă de prezentare a valorilor efective este tabelul (raportul).

Studiul rapoartelor permite stabilirea proprietăţilor, a domeniilor de variaţie a valorilor fiecărui câmp dintr-un articol.

Programul generează fişiere cu date de test aleatoare. Mai întâi este citit numele extern al fişierului.

Structura articolelor fişierului generat se introduce de la tastatură. Câmpurile articolelor sunt de următoarele tipuri:

- alfabetic;
- alfanumeric;
- întreg (de tip *int*).

Câmpurilor alfabetice şi alfanumerice li se precizează lungimea. Pentru aceste tipuri se generează datele în modurile următoare:

- aleator;
- aleator dintr-o listă de valori specificate;

Pentru datele de tip întreg, exista şi o a treia posibilitate de generare, între o limită inferioară şi o limită superioară.

În final se citeşte numărul de articole de generat.

Programul parcurge următorii paşi:

a) introducere nume extern de fişier (FMAT.DAT) ca şir de caractere de maximum 127 elemente;
b) definire număr, tipuri şi domenii ale câmpurilor;
c) generare şiruri alfabetice;
d) generare şiruri alfanumerice;
e) generare valori întregi cuprinse între limitele specificate;
f) înscrierea articolelor în fişier;
g) închiderea fişierului.

De exemplu, pentru gestiunea stocurilor de materiale de la întreprinderea X se consideră fişierul PTEST.DAT ale cărui articole au structura:

```
struct Pers
{
   char nume[30];
   int virsta;
   int vech;
   int copii;
   char sex[1];
};
```

Condiţiile pentru crearea articolelor sunt:
- vârsta cuprinsă între 16 şi 62;
- vechimea între 0 şi 46;
- copii între 0 şi 10;
- sex 'M' sau 'F';

Pentru crearea fişierului "PTEST.DAT" având de 50 de articole este purtat următorul dialog între utilizator şi program:

Nume fisier: pmat.dat
Numarul de campuri :5
Numar articole de generat: 50

Tipul cimpului 1
0-intreg
1-alfabetic
2-alfanumeric
-->1
Lungime: 30
Generare
0-aleator
1-din lista de valori
-->0

Tipul cimpului 2
0-intreg
1-alfabetic
2-alfanumeric
-->0
Generare
0-aleator
1-in interval
2-din lista de valori
-->1
Limita inferioara:16
Limita superioara:46

Tipul cimpului 3
0-intreg
1-alfabetic
2-alfanumeric
-->0
Generare
0-aleator
1-in interval
2-din lista de valori
-->1
Limita inferioara:0
Limita superioara:46

Tipul cimpului 4
0-intreg
1-alfabetic
2-alfanumeric
-->0
Generare
0-aleator
1-in interval
2-din lista de valori
-->1
Limita inferioara: 0
Limita superioara: 10
Tipul cimpului 5
0-intreg
1-alfabetic
2-alfanumeric
-->1
Lungime: 1

Limita superioara: 62

Tipul cimpului 3
0-intreg
1-alfabetic
2-alfanumeric
-->0

Generare
0-aleator
1-din lista de valori
-->1
Numar elemente: 2
Element 1: M
Element 2: F

În continuare este prezentat codul sursă al programului care generează fişiere de test, programul fiind scris în limbajul C++.

```
#include <string.h>
#include <limits.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>

using namespace std;

int random(int li, int ls)
{
	return (li+(rand()/(float)RAND_MAX)*ls);
}

/* defineste structura unui cimp care urmeaza sa aiba valori generate
aleator */
class Cimp
{
	int tip;//0-intreg, 1-alfabetic, 2-alfanumeric
	int mod;//0 - aleator, 1 - lista
	struct InfoAlf
	{
		int lg;
		int ne;//numarul de elemente din lista de valori
		char **lv;
	};
	//informatii despre cimpurile intregi

	struct InfoInt
	{
		int li,ls;
		int ne;
		int *lv;
	};
	//informatii despre un cimp din articol

	struct InfoInt *intreg;
	struct InfoAlf *alfa;

	void SetTip(int _tip)
	{
		tip=_tip;
		switch(tip)
		{
		case 0:
			{
				//intreg
				intreg=new InfoInt();
				alfa=NULL;
```

```
                        break;
                }
        case 1:
        case 2:
                {
                        intreg=NULL;
                        alfa=new InfoAlf();
                        break;
                }

        }
}

void SetMod(int _mod)
{
        mod=_mod;
}

Cimp()
{

}
Cimp(int _tip,int _mod):tip(_tip),mod(_mod)
{
        switch(tip)
        {
        case 0:
                {
                        //intreg
                        intreg=new InfoInt();
                        alfa=NULL;
                        break;
                }
        case 1:
        case 2:
                {
                        intreg=NULL;
                        alfa=new InfoAlf();
                        break;
                }

        }
}

~Cimp()
{
        //eliberare memorie

        switch(tip)
        {
        case 0:
                if(intreg->ne!=-1)
                        delete [] intreg->lv;
                delete intreg;
                break;
        case 1:
        case 2:
                if(alfa->ne!=-1)
                {
                        for(int j=0;j<alfa->ne;j++)
                                delete [] alfa->lv[j];
```

```
                        delete [] alfa->lv;
                }
                delete alfa;
                break;
        }

    }

    void SetIntregRnd(int _li=0,int _ls=INT_MAX-1)
    {
        //limita inf=0 limita sup=intregul maixim

        intreg->li=_li;
        intreg->ls=_ls;
        intreg->ne=-1;
    }

    void SetIntregRndLV(int _ne, int lv[])
    {
        //citesc numarul si elem din lista de val posibile

        intreg->ne=_ne;
        //aloc pentru numarul de elemente
        intreg->lv=new int[_ne];
        for(int j=0;j<intreg->ne;j++)
        {
            intreg->lv[j]=lv[j];
        }
    }

    void SetAlfaRnd(int _lg)
    {
        alfa->lg=_lg;
        alfa->ne=-1;
    }

    void SetAlfaRndLV(int _lg,int _ne,char **lv)
    {
        alfa->ne=_ne;
        alfa->lg=_lg;
        //aloc pentru numarul de elemente
        alfa->lv=new char*[_ne];
        for(int j=0;j<alfa->ne;j++)
        {
            //aloc pentru lungimea fiecarul element;
            alfa->lv[j]=new char[alfa->lg+1];
            strncpy(alfa->lv[j],lv[j],alfa->lg);
        }


    }

    friend class GenTest;
};

class GenTest
{
    FILE * _pf;
    char NumeFisier[255];
    int _nCimpuri;
    int _nArticole;
```

```
	Cimp *art;
public:
	GenTest(char * nFisier,int nCimpuri,int nArticole)
	{
		srand( (unsigned)time( NULL ) );
		strcpy(NumeFisier,nFisier);
		_pf=fopen(nFisier,"w+b");
		_nCimpuri=nCimpuri;
		_nArticole=nArticole;

		//!!!!!!!!!!!!!!!!!!!
		art=new Cimp[nCimpuri];
	}


	~GenTest()
	{
		if(_pf!=NULL)
			fclose(_pf);
		if(art!=NULL)
			delete [] art;

	}

	static char Caractere[63];


	void GenIntreg(int _idxCimp)
	{
		int num,i;
		if(art[_idxCimp].intreg->ne==-1)
			//gen nr aleat cuprins intre linf si lsup
			num=random(art[_idxCimp].intreg-
>li,art[_idxCimp].intreg->ls);
		else
		{
			i=random(0,art[_idxCimp].intreg->ne);
			num=art[_idxCimp].intreg->lv[i];
		}
		fwrite(&num,sizeof(int),1,_pf);
	}

	void GenAlfa(int _idxCimp)
	{
		int i,j,l,k;
		char *car=new char[art[_idxCimp].alfa->lg+1];
```

26.3. Generatoare de matrice de test

Numeroase programe prelucrează matrice şi vectori între care există anumite legături. Este preferabil ca matricele de intrare mai ales atunci când sunt de mari dimensiuni să fie rezultatul unor procese de generare. De exemplu, se consideră un program destinat rezolvării unor sisteme de ecuaţii liniare prin metoda Jacobi iterativă.

Pentru testarea acestui tip de program se iau în considerare două modalităţi şi anume:

- preluarea de exemple de sisteme liniare din cărţile de analiză matematică;
- generarea matricei A a sistemului, generarea vectorului de soluţii X; se efectuează produsul A*X şi se obţine vectorul B al termenilor liberi.

Programul preia ca intrări din fişier matricea A, vectorul termenilor liberi B, rezolvă sistemul şi compară soluţia sa X_0 cu valoarea vectorului X generat.

Pentru situaţii speciale, generatorul de matrice permite:

- generare de matrice simetrice pozitiv definite;
- generarea de matrice unitate;
- generarea matrice triangulare;
- generarea matrice care se bucură de proprietatea conform căreia suma elementelor de pe o linie este egală cu o valoare dată (matricele markoviene sunt caz particular, suma elementelor de pe linie este 1);
- generarea de matrice cu elemente maxime pe diagonala principală;
- generarea de matrice în care se specifică liniile sau coloanele identice.

Programul MATEST.CPP este un exemplu de generator de date de test matriceale. Programul prezintă doar câteva din aceste posibilităţi, permiţând şi includerea altor tipuri de generări de matrice.

```
#include <iostream>
#include <time.h>

#define NMAX 20

using namespace std;


int random(int li, int ls)
{
	return (li+(rand()/(float)RAND_MAX)*ls);
}

class GenMat
{
	int a[NMAX][NMAX];
	int n,li,ls;

public:
	GenMat(int _n,int _li,int _ls):n(_n),li(_li),ls(_ls){}

	//Generaza o matrice cu elemente aleatoare cuprinse intre li si
ls
	void GenMatRnd()
	{
		for(int i=0;i<n;i++)
			for(int j=0;j<n;j++)
				a[i][j]=random(li,ls);
	}

	//Genereaza matricea unitate
	void GenUnit()
	{
```

```
	for(int i=0;i<n;i++)
		for(int j=0;j<n;j++)
		{
			if(i==j)
				a[i][j]=1;
			else
				a[i][j]=0;
		}
}
//Genereaza matricea simetrica
void GenSim()
{
	for(int i=0;i<n;i++)
		for(int j=i;j<n;j++)
		{
			if(i==j)
				a[i][j]=random(li,ls);
			else
				a[i][j]=a[j][i]=random(li,ls);
		}
}

//Genereaza matricea triangulara
void GenTriang()
{
	int i,j;
	for(i=0;i<n;i++)
		for(j=0;j<n;j++)
		{
			if((i+j)>=n)
				a[i][j]=0;
			else
				a[i][j]=random(li,ls);
		}
}
//Genereaza matrice cu elementele maxime pe diagonala
void GenMaxDiag()
{
	int jum=(ls-li)/2;
	for(int i=0;i<n;i++)
		for(int j=0;j<n;j++)
		{
			if(i==j)
				a[i][j]=jum+random(li,ls-jum);
			else
				a[i][j]=random(li,jum);
		}
}

//Adiseaza matricea generata la consola/scriere in fisier
void scrieA(FILE *f=stdout)
{
	if(f!=stdout)
	{
		f=fopen("matr.txt","wt");
		if(!f)
		{
			printf("Eroare la creare fisier !\n");
			return;
		}
	}
```

```
            for(int i=0;i<n;i++)
            {
                  for(int j=0;j<n-1;j++)
                        fprintf(f,"%d ",a[i][j]);
                  fprintf(f,"%d\n",a[i][j]);
            }
            if(f!=stdout)
                  fclose(f);
      }

};

void main()
{
      int opt,n,li,ls;

      srand( (unsigned)time( NULL ) );
      do
      {
            cout<<"Tipul de matrice:"<<endl;
            cout<<"1. Unitate"<<endl;
            cout<<"2. Simetrica"<<endl;
            cout<<"3. Triangulara"<<endl;
            cout<<"4. Maxim pe diagonala"<<endl;
            cout<<"5. Elemente aleatoare"<<endl;
            cout<<"-->";
            cin>>opt;
      }
      while(opt<1&&opt>5);

      cout<<("Numarul de linii/coloane: ");
      cin>>n;
      cout<<("Limita inferioara: ");
      cin>>li;
      cout<<("Limita superioara: ");
      cin>>ls;

      GenMat gm(n,li,ls);

      switch(opt)
      {
      case 1:gm.GenUnit();break;
      case 2:gm.GenSim();break;
      case 3:gm.GenTriang();break;
      case 4:gm.GenMaxDiag();break;
      case 5:gm.GenMatRnd();break;
      }
      gm.scrieA();
      cin.get();
      }
```

Programul generează matrice pătratice si include următoarele tipuri de matrice: matrice unitate, matrice simetrică, matrice triangulară, matrice cu valorile maxime pe diagonală şi matrice cu elemente generate aleator.

26.4 Fişiere date de test livrate

La elaborarea de specificaţii pentru dezvoltarea de software apare necesitatea creării unui cadru complet. Unul dintre elementele esenţiale îl reprezintă datele de test livrate. Aceste sunt fişiere ce conţin o multitudine de baterii de date de test.

Din specificaţii rezultă care sunt ieşirile software-ului corect şi modul de întoarcere a rezultatelor proprii obţinute prin tratarea datelor de test.

Se consideră problema P căreia i se asociază datele de test livrate sub forma copiilor fişierului F, achiziţionate de *n* producători de software. Rezultatele standard ale bateriilor de test din fişierul F se constituie sub forma de exemple de verificare în fişierul R.

Software-ul obţinut de la cei *n* producători pentru a participa la licitaţie este dat în forma de variante V_1, V_2, .., V_n. Producătorii de software obţin pe baza intrărilor din fişierul F rezultate pe care le stochează în fişierele r_1, r_2, .., r_n.

Depunerea spre recepţie a software-ului în variantele V_1, V_2, .., V_n este precedată de analiza calitativă şi cantitativă a conţinutului fişierelor r_1, r_2, .., r_n. Numai în cazul încadrării între limitele precizate a calităţii rezultatelor se acceptă preluarea variantelor V_1, V_2, .., V_n şi deci participarea efectivă la licitaţie cu produse program existente.

Fişierele livrate cu date de test generează un proces continuu de testare, întrucât nu este exclusă existenţa de inconsistenţe chiar în fişierul R, figura 26.2.

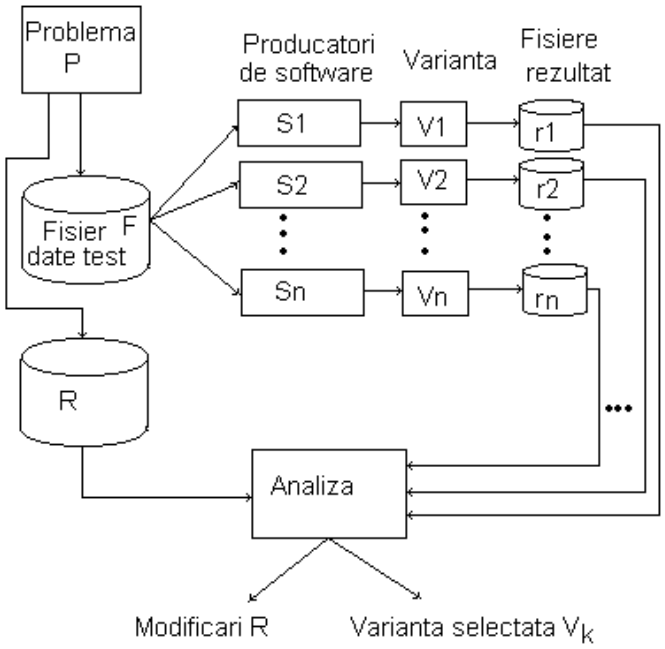

Figura 26.2 *Proces iterativ de generare şi selecţie software*

În contextul analizei, proiectării şi programării orientate obiect cu atât mai mult se impune dezvoltarea sub formă de variante prin referire de membri ai claselor, ştiut fiind faptul că se elaborează din start software pentru un număr foarte mare de utilizatori.

Schimbarea concepţiei privind dezvoltarea de software pentru calcul salarii, gestiune mijloace fixe, gestiune materiale, comenzi, contracte şi evidenţă contabilă permite o altfel de abordare.

Dezvoltarea de software direcţionat spre utilizare generală impune licitaţii pe produse existente pentru a obţine efecte pozitive în mod real. Înseamnă deci, că se creează premisele achiziţionării foarte avantajoase a unui anumit produs şi trecerea la utilizarea sa.

În primul rând se obţine propagarea la nivelul utilizatorilor a tuturor efectelor pozitive pe care le are incorporate varianta de produs program selectată.

În al doilea rând dispare paralelismul abordării permanente a unei aceeaşi probleme de către mai multe case producătoare de software.

În al treilea rând se obţine un nivel de omogenitate a ieşirilor informaţionale la toţi utilizatorii de programe care rezolvă o aceeaşi problemă, ceea ce permite continuarea de aplicaţii integrate pe diferite nivele.

În al patrulea rând producătorii de software se eliberează de problematici comune, putând să-şi canalizeze eforturile în alte zone ale procesului de informatizare.

Toate acestea sunt posibile numai prin planificarea fazei de testare software care garantează robusteţea şi fiabilitatea acestuia, premise ale utilizării de software independent de producător.

26.5 Testarea programelor cu structuri statice

Testarea produselor program reprezintă una dintre etapele căreia dezvoltatorii de software trebuie să-i acorde maximă importanţă, întrucât comportamentul la utilizatori este influenţat de existenţa sau inexistenţa erorilor. Cu cât numărul de utilizatori este mai mare şi în procesul de testare nu au fost identificate şi corectate erori, cu atât efectele negative de antrenare multiplă sunt mai ample.

Produsele program construite cu structuri statice caracterizează acea concepţie conform căreia dimensiunile problemelor de rezolvat se estimează cu suficientă precizie, iar raportul dintre zonele de memorie alocate şi cele strict utilizate se gestionează, asigurându-se un grad de folosire cel puţin satisfăcător.

Testarea programelor care includ structuri de date alocate static prezintă particularităţi specifice, accentul punându-se fie pe încadrarea câmpurilor referite în alocările existente, fie pe stabilirea gradului de apartenenţă a operanzilor la domenii specificate.

Structurile de date statice sunt: masive unidimensionale, masive bidimensionale, masive multidimensionale, articole, masive de articole şi fişiere cu moduri de organizare şi tipuri de acces cărora le corespund funcţii sau instrucţiuni de limbaj.

Programele care includ masive unidimensionale conţin secvenţe pentru:

- iniţializarea din fişiere sau de la tastatură a elementelor masivelor;
- traversarea masivelor în vederea efectuării de calcule element cu element;
- iniţializarea prin atribuire a elementelor din masive;

- afişarea conţinutului elementelor.

Testarea vizează stabilirea concordanţei între specificaţiile de program şi prelucrările efectiv realizate de secvenţele programelor. În cazul în care programele alocă zone de memorie corespunzătoare unor dimensiuni considerate maxime ale problemelor, programatorii trebuie să verifice dacă numărul efectiv al componentelor referite din masive corespunde definirii iniţiale. În secvenţa:

```
…
public static int PMAX=20;

Produs produse[];
int n=0;
…
public boolean AdaugaProdus(Produs p)
{
      produse[n] = p;
      n++;
}
…
```

se observă că după un număr de 20 de apeluri se ajunge la referirea unei zone de memorie adiacentă masivului *produse[]*. De aceea este necesară existenţa unui test pentru a evita depăşirea limitelor masivului.

Testarea are rolul de a evidenţia prezenţa sau absenţa secvenţelor de validare a încadrării expresiilor indiciale în limite impuse de definirea masivelor unidimensionale.

În cazul în care iniţializarea masivului unidimensional ia în considerare n componente, iar referirea vizează m componente şi m>n, rezultă că la evaluarea expresiilor se utilizează şi câmpuri neiniţializate. De exemplu secvenţa:

```
#define N 10

int x[N]={1,2,3,4,5,6,7}, n=5, m=7, y[N];
for(int i=0;i<n;i++)
      y[i]=i*i*i+3;
for(i=0;i<m;i++)
      x[i]+=y[i];
```

conduce la obţinerea de rezultate incorecte în legătură cu componentele *x[5]* şi *x[6]*. În cazul în care în program există definite teste de validare a apartenenţei expresiilor de referire la valori care corespund componentelor iniţializate corect, testele vor determina apariţia de mesaje specifice tentativei de ieşire din aceste intervale.

Programele care conţin definiri de masive bidimensionale trebuie să includă o serie de expresii care să stabilească concordanţa dintre aceste masive şi masivele unidimensionale. De exemplu se consideră expresia matriceală

$$b = (A'A)^{-1}B \qquad (26.1)$$

unde:

- A – masiv bidimensional având m linii şi n coloane;
- A’ – transpusa masivului bidimensional A;

- B – masiv unidimensional având n elemente.

Dacă masivul A este iniţializat pe m linii şi n coloane, iar masivul B este iniţializat pe k elemente, trebuie validată existenţa egalităţii lui n cu k.

Testarea programelor care manipulează elemente ale masivelor bidimensionale trebuie să scoată în evidenţă concordanţa dintre modul în care sunt iniţializate blocuri şi modul în care acestea sunt întrebuinţate.

Definirea diferită de iniţializare conduce la rezultate incorecte afectate de regulile de referire, diferite de la o etapă la alta.

Lucrul cu articole ca tipuri de date neomogene impune construirea de expresii de referire care să ofere o flexibilitate suficient de mare pentru structurile instabile în care se adaugă câmpuri, se inserează câmpuri, se interschimbă câmpuri, se elimină câmpuri, se modifică tipul sau lungimea câmpurilor existente.

Testarea programelor care utilizează structuri de tip articol are rolul de a scoate în evidenţă concordanţa dintre tipul câmpului şi conţinutul acestuia.

În cazul structurilor statice testarea conţinutului operanzilor prezintă importanţă, întrucât numai apartenenţa la domeniul specific aplicaţiei garantează calitatea şi completitudinea rezultatelor.

În cazul masivelor de articole este important ca expresiile de referire să asigure traversarea completă a componentelor iniţializate în contextul existent în specificaţiile de programare. Costul testării programelor care utilizează structuri statice variază în funcţie de tipul structurilor, ţinând cont de faptul că în cazul fişierelor problemele care pot să apară sunt mai numeroase, decât de exemplu în cazul masivelor unidimensionale.

26.6 Particularităţi ale testării programelor cu structuri dinamice

Structurile dinamice, liste simple, liste dublu înlănţuite, arbori binari, grafuri etc. sunt rezultatul necesităţii creşterii gradului de utilizare a zonelor de memorie definite şi referite. Dacă în cazul unui masiv unidimensional cu maxim N componente sunt referite primele n componente, n<N, în cazul unei liste simplu înlănţuită cu m componente, toate cele m componente fac obiectul referirii în program.

Spre deosebire de structurile statice, structurile dinamice folosesc variabile pointer, al căror conţinut sunt deplasări (adrese relative). Testele programelor care conţin structuri dinamice au dublu rol. Pe de o parte vizează conţinutul câmpurilor aferente informaţiilor utile scopului pentru care a fost elaborat programul şi pe de altă parte, vizează conţinutul variabilelor pointer care trebuie să fie NULL sau o deplasare care să permită referirea elementului următor, referirea elementului precedent sau a unuia descendent, în funcţie de tipul structurii dinamice.

În cazul în care legăturile nu sunt corect realizate, funcţiile de traversare conduc la referiri parţiale şi la întreruperi necontrolate ale execuţiei.

Există situaţii în care, printr-o gestionare defectuoasă a variabilelor pointer se pierd adresele capetelor de listă sau adresa rădăcinii arborelui. Pentru evitarea unor incidente, în faza de testare a produsului program informaţiile utile trebuie să fie preluate din fişiere, până la stabilizarea mecanismelor de construire şi actualizare a structurilor dinamice.

În limbajul Java, spre deosebire de C++, nu există variabile de tip pointer şi eliberarea memoriei alocate se face automat de către o componentă specializată, ceea ce conduce la posibilităţi reduse de apariţie a erorilor din aceste cauze, testarea concentrându-se asupra altor aspecte.

Dacă se lucrează cu componente adiacente trebuie gestionat comportamentul programului în raport cu primul element, respectiv cu ultimul element. Secvenţa C++ care include expresii de forma:

```
pcl->next->next
```

sau:

```
pcl->prev->prev
```

trebuie să verifice dacă este atinsă una dintre limite. Absenţa unui astfel de test generează tentative de referire a unor componente care nu aparţin structurii de date creată prin programul scris în cadrul aplicaţiei.

Structurile dinamice presupun referirea funcţiilor de alocare a memoriei **`malloc()`**, **`calloc()`**, operatorul **`new`**, precum şi funcţia de eliberare a memoriei **`free()`** sau operatorul de eliberare a memorie **`delete`**. Oricărei alocări trebuie să-i corespundă într-un alt moment al execuţiei o eliberare de memorie. În cazul în care se înregistrează dezechilibre, fie are loc umplerea stivei, fie are loc golirea prematură a acesteia. Testarea trebuie să vizeze simetria dintre procesele de alocare şi procesele de eliberare a memoriei.

Sunt numeroase situaţiile în care nivelurile de indirectare sunt mai mari decât unu, ceea ce impune stabilizarea aritmeticilor de pointeri pentru a realiza concordanţa între modul în care sunt definite şi iniţializate pe toate nivelurile aceste variabile şi, respectiv, evaluările expresiilor de referire.

Bibliotecile de funcţii create pentru manipularea elementelor constitutive ale structurilor dinamice sau funcţiile membre ale claselor asociate acestor structuri, fie în forma directă, fie în forma recursivă, trebuie să asigure obţinerea de adrese sau de poziţii ale elementelor. După fiecare funcţie trebuie realizate teste care determină eliminarea fluxurilor de prelucrare generatoare de incidente, atunci când referirile de variabile pointer nu mai aparţin domeniului specific unui segment de memorie.

Testarea programelor cu structuri dinamice implică un efort de testare mai mare decât în cazul structurilor statice, având în vedere faptul că algoritmii utilizaţi în manipularea structurilor de date dinamice sunt mai complecşi, şi de aceea, şi costul asociat testării acestor programe este mai ridicat.

26.7 Procese de testare pentru programele cu structuri agregate

Structurile de date agregate sunt vectorii de structură, vectorii de pointeri spre structuri, spre funcţii sau alte modalităţi de concatenare a structurilor neomogene. Testarea structurilor de date agregate vizează, pe de o parte, verificarea dacă agregarea corespunde cerinţelor enunţate în problemă şi, pe de altă parte, vizează testarea conţinutului câmpurilor

referite pentru a vedea dacă există concordanţă între modul de iniţializare şi modul de utilizare.

Numeroase programe conţin structuri de date statice şi structuri de date dinamice agregate. De exemplu pentru construirea unui graf se iau în considerare informaţiile de descriere a nodurilor sub forma unui masiv de articole ca structură statică, de care se conexează liste care descriu arcele incidente spre exterior fiecărui nod în parte.

Există situaţii în care structurile arborescente oarecare au asociate noduri alocate dinamic şi legături între noduri, construite în acelaşi fel. Fiecare nod conţine un câmp care indică numărul de pointeri iniţializaţi cu adresele descendenţilor şi un vector de pointeri ca structură statică în care se stochează adrese de descendenţi.

Testele care se efectuează pentru structurile dinamice agregate au rolul de a stabili că legăturile dintre noduri sunt cele existente în realitate şi mai ales că proprietăţile structurii redau proprietăţi existente între elementele din realitate care fac obiectul reflectării în plan informatic.

Agregarea de tip omogen care conduce la obţinerea vectorilor de vectori, listelor de liste, articolelor de articole, fişierelor de fişiere au corespondent în planul testării includerea de structuri repetitive imbricate, în număr egal cu numărul gradului de agregare.

Secvenţa de program:

```
#include <stdio.h>

struct dlista
{
struct dlista *dprev;
int info;
struct dlista *dnext;
};

struct lista
{
      struct dlista * plista;
      struct lista *next;
};

void main()
{
struct dlista x11,x12,x13, x21,x22,x23,x24, x31,x32,x33,x34;
struct lista *py, y1, y2, y3;

py=&y1;
y1.next=&y2;
y2.next=&y3;
y3.next=NULL;

y1.plista=&x11;x11.dnext=&x12;x12.dnext=&x13; x13.dnext=NULL;
y2.plista=&x21;x21.dnext=&x22;x22.dnext=&x23;x23.dnext=&x24;
x24.dnext=NULL;
y3.plista=&x31;x31.dnext=&x32;x32.dnext=&x33;x33.dnext=&x34;
x34.dnext=NULL;
…
while(py!=NULL)
{
 …
 while(py->plista!=NULL)
```

```
 {
      …
      py->plista=py->plista->dnext;
 }
 py=py->next;
}
}
```

prezintă interes deosebit stabilirea modului în care comutativitatea operaţiilor de agregare implică comutativitate în expresiile de referire. Testarea acestei structuri prezintă o serie de dificultăţi, având în vedere atât gradul de agregare, cât şi utilizarea variabilelor de tip pointer.

În cazul în care cercetările evidenţiază astfel de proprietăţi, în procesul de testare se reflectă prin modul de combinare a valorilor existenţa capacităţii de traversare a structurilor arborescente asociate prelucrărilor în care operatorii sunt componente referite prin intermediul variabilelor pointer.

Costul testării programelor care conţin structuri de date agregate este mai mare decât în cazul programelor care utilizează structuri de date statice. Numărul de cazuri de test este mai mare având în vedere complexitatea ridicată a programelor şi numărul crescut de posibilităţi de eroare.

Testarea tipologiilor de proceduri urmează necesităţii definirii de construcţii software înzestrate cu proprietatea de coeziune maximă. Fiecare procedură este ortogonală celorlalte componente din produsul software care se realizează. Înseamnă că fiecare procedură realizează un anumit tip de prelucrare bine definit, care nu se regăseşte sub nici o formă în alte proceduri.

Acest tip de structuri omogene de proceduri, omogenitatea fiind specifică secvenţelor care se includ în procedură, creează un nou mod de abordare a activităţii de programare, caracterizat prin controlul redundanţei şi orientarea acesteia spre niveluri scăzute.

Specializarea echipelor pe tipologii de proceduri conduce la creşterea productivităţii în procesul de testare şi în utilizarea eficientă a mediilor de asistare a acestui proces.

Când se elaborează un produs program complex format din module neomogene în raport cu tipurile de date utilizate se urmăreşte folosirea diferenţiată a tehnicilor de testare pornind de la tipurile de date preponderente. Se impune utilizarea la nivelul modulelor de structuri de date de aceeaşi categorie şi se obţine drept consecinţă definirea de date de test specifice, exclusiv lucrului cu matrice sau exclusiv lucrului cu liste sau exclusiv lucrului cu structuri arborescente. Deci, vor exista module care utilizează masive unidimensionale, module care operează cu masive bidimensionale, module care utilizează fişiere, module care lucrează cu liste simple, module care lucrează cu liste dublu înlănţuite, module destinate lucrului cu arbori şi module destinate lucrului cu grafuri, fără a se face combinaţii între structuri. Este numai o aparenţă fărâmiţarea produsului program într-un număr foarte mare de module, cu creşterea numărului de niveluri, pentru că în realitate prin creşterea omogenităţii modulelor creşte eficacitatea procesului de testare şi, implicit, calitatea produsului program în ansamblu.

Pentru fiecare tip de structură se definesc un set de obiective ale testării şi proceduri de testare care trebuie urmate. Rezultatul testării trebuie să impună modificări în textele sursă care ameliorează prelucrările şi la reluarea procesului de testare nu se mai obţin aceleaşi erori.

Ştiindu-se că structurile de control sunt concepute diferenţiat pentru implementarea operatorilor pe structuri de date, o combinare adecvată a testării orientate pe structuri de control cu testarea orientată pe structuri de date va avea efecte deosebite, chiar în condiţiile în care complexitatea pe ansamblu a produsului program final este deosebit de ridicată. Mai mult, sunt create în acest fel condiţii de testare pe trei categorii specializate de structuri de control şi pe un număr K de structuri de date, ceea ce conduce la existenţa unui număr de 3K tipologii combinate de date de test. Oricare ar fi modulul, va exista printre cele 3K tipologii de date de test exemplul care să permită evidenţierea erorilor, prin teste mecanice, încă de la început. Etapa de testare trece din planul artei dezvoltării de software în planul proceselor coerente, previzibile.