**Testarea aleatorie**

**Testarea aleatorie** este o tehnică de testare software cu cutie neagră în care programele sunt testate prin generarea de intrări aleatorii și independente. Rezultatele ieșirii sunt comparate cu specificațiile software-ului pentru a verifica dacă ieșirea de test este trecută sau eșuată. În cazul absenței specificațiilor, se utilizează excepții ale limbajului, ceea ce înseamnă că, dacă apare o excepție în timpul executării testului, înseamnă că există o defecțiune în program, este utilizată și ca modalitate de a evita testarea părtinitoare.

**Istoricul testelor aleatorii**

Testarea aleatorie a hardware-ului a fost examinată pentru prima dată de Melvin Breuer în 1971, iar efortul inițial de evaluare a eficacității sale a fost făcut de Pratima și Vishwani Agrawal în 1975.

În software, Duran și Ntafos au examinat testarea aleatorie în 1984.

**Prezentare generală**

Luați în considerare următoarea funcție C ++:

int myAbs ( int x ) {

 **if**  ( x > 0 ) {

 **return**  x ;

 }

 **else**  {

 **return**  x ; *// bug: ar trebui să fie '-x'*

 }

}

Acum testele aleatorii pentru această funcție ar putea fi {123, 36, -35, 48, 0}. Numai valoarea „-35” declanșează eroarea. Dacă nu există nicio implementare de referință pentru a verifica rezultatul, eroarea ar putea trece neobservată. Cu toate acestea, ar putea fi adăugată o afirmație pentru a verifica rezultatele, cum ar fi:

void testAbs ( int n ) {

 **for**  ( int i = 0 ; i < n ; i ++ ) {

 int x = getRandomInput ();

 int result = myAbs ( x );

 afirma ( rezultat > = 0 );

 }

}

Implementarea de referință este uneori disponibilă, de exemplu atunci când implementați un algoritm simplu într-un mod mult mai complex pentru o performanță mai bună. De exemplu, pentru a testa o implementare a algoritmului Schönhage – Strassen , se poate utiliza operația standard „\*” pe numere întregi:

int getRandomInput () {

 *// ...*

}

void testFastMultiplication ( int n ) {

 **for**  ( int i = 0 ; i < n ; i ++ ) {

 long x = getRandomInput ();

 y lung = getRandomInput (); rezultat lung = FastMultiplication ( x , y ); afirmă ( x \* y == rezultat ); } }

În timp ce acest exemplu este limitat la tipuri simple (pentru care poate fi utilizat un generator aleatoriu simplu), instrumentele care vizează limbaje orientate pe obiecte explorează de obicei programul pentru a testa și găsi generatoare (constructori sau metode care returnează obiecte de acel tip) și le apelează folosind aleator intrări (fie ele înșiși generate în același mod, fie generate folosind un generator pseudo-aleator, dacă este posibil). Astfel de abordări mențin apoi un grup de obiecte generate aleatoriu și utilizează o probabilitate fie pentru reutilizarea unui obiect generat, fie pentru crearea unuia nou.

**Despre întâmplare**

Conform lucrării seminale despre testarea aleatorie de D. Hamlet

[..] semnificația tehnică, matematică a „testării aleatorii” se referă la o lipsă explicită de „sistem” în alegerea datelor de testare, astfel încât nu există nicio corelație între diferite teste.

**Punctele forte și punctele slabe**

Testarea aleatorie este de obicei lăudată pentru următoarele puncte forte:

* Este ieftin de utilizat: nu trebuie să fie inteligent cu privire la programul testat.
* Nu are nici o părtinire: spre deosebire de testarea manuală, nu trece cu vederea erorile, deoarece există o încredere greșită în unele coduri.
* Este rapid să găsiți candidați la erori: durează de obicei câteva minute pentru a efectua o sesiune de testare.
* Dacă software-ul este specificat corect: găsește erori reale.

Următoarele puncte slabe sunt de obicei subliniate de detractori:

* Găsește doar erori de bază (ex. identificarea pointerului null ).
* Este la fel de precisă pe cât specificațiile și specificațiile sunt de obicei imprecise.
* Se compară slab cu alte tehnici pentru a găsi erori (ex. analiza statică a programului ).
* Dacă sunt selectate aleatoriu intrări diferite la fiecare test, acest lucru poate crea probleme pentru integrare continuă, deoarece aceleași teste vor trece sau vor eșua aleatoriu.
* Unii susțin că ar fi mai bine să acoperiți cu grijă toate cazurile relevante cu teste construite manual într-o manieră albă, decât să vă bazați pe întâmplare.

**Tipuri de testare aleatorie**

**În ceea ce privește intrarea**

* Generarea aleatorie de secvențe de intrare (adică o secvență de apeluri de metodă)
* Secvență aleatorie de intrări de date (numită uneori testare stocastică) - f.ex. o secvență aleatorie de apeluri de metodă
* Selectarea aleatorie a datelor din baza de date existentă

**Ghidat vs neguidat**

* generarea de teste aleatorii nedirecționate - fără euristică care să ghideze căutarea acestuia
* generare de teste aleatorii direcționate - f.ex. „generarea testului aleatoriu orientat prin feedback”  sau „testarea aleatorie adaptativă”

**Implementări**

Unele instrumente care implementează testarea aleatorie:

* QuickCheck - un instrument de test celebru, dezvoltat inițial pentru Haskell, dar portat în multe alte limbi, care generează secvențe aleatorii de apeluri API bazate pe un model și verifică proprietățile sistemului care ar trebui să fie valabile după fiecare rulare.
* Randoop - generează secvențe de metode și invocații de constructori pentru clasele testate și creează teste JUnit din acestea
* Simulant - un instrument Clojure care rulează simulări ale diferiților agenți (f.ex. utilizatori cu profiluri comportamentale diferite) pe baza unui model statistic al comportamentului lor, înregistrând toate acțiunile și rezultatele într-o bază de date pentru explorare și verificare ulterioară
* AutoTest - un instrument integrat EiffelStudio care testează automat codul Eiffel cu contracte bazate pe prototipul de cercetare omonim.
* York Extensible Testing Infrastructure (YETI) - un instrument de limbaj agnostic care vizează diferite limbaje de programare (Java, JML, CoFoJa, .NET, C, Kermeta).
* GramTest - un instrument de testare aleatorie bazat pe gramatică scris în Java, folosește notația BNF pentru a specifica gramaticile de intrare.

**Critică**

Testarea aleatorie are doar o nișă specializată în practică, mai ales pentru că un oracol eficient este rareori disponibil, dar și din cauza dificultăților cu profilul operațional și cu generarea de valori de intrare pseudorandom.

Un oracol de testare este un instrument pentru verificarea dacă rezultatele se potrivesc sau nu cu specificațiile programului. Un profil de operație este cunoștința despre tiparele de utilizare ale programului și, prin urmare, ce părți sunt mai importante.

Pentru limbajele și platformele de programare care au contracte (de exemplu Eiffel. .NET sau diverse extensii Java cum ar fi JML, CoFoJa ...) contractele acționează ca oracole naturale și abordarea a fost aplicată cu succes. În special, testarea aleatorie găsește mai multe erori decât inspecțiile manuale sau rapoartele utilizatorilor (deși diferite).