Verificare și validare software

Testare white-box. Acoperire cu teste

10 octombrie 2013

Testarea white-box

Generează teste bazat pe *structura internă* a codului
Alte denumiri: glass box, clear box, open box

Terminologie alternativă
testare *comportamentală* (black-box) / *structurală* (white-box)

Comparație:
– black-box: la orice nivel / white-box: mai ales la nivel de modul
– white-box: modificarea codului $\Rightarrow$ modificarea testelor
– white-box: ușurează detectarea *erorilor de codare*,
dar nu poate detecta *erori de omisiune* (în cod sau specificații)

Structura unui program: terminologie

Graful de flux de control (control flow graph)

Reprezentare sub formă de graf a structurii programului
și implicit a căilor sale de execuție

Noduri = instructiuni

Muchii (evtl. etichetate cu condiții): secvențierea între instrucțiuni

Uzual, instrucțiunile "în linie dreaptă": grupate împreună $\Rightarrow$

basic block =

secvență de instrucțiuni cu un singur punct de intrare și de ieșire
(fără salturi în mijlocul codului, sau din mijloc afară)

Acoperirea prin teste: (code) coverage

= criteriu pentru a măsura *dacă un set de teste e adecvat*

La ce bun astfel de criterii ? [Burnstein]

Ce proprietăți ale programului să examinăm ?

Ce date de test selectăm pentru aceste proprietăți ?

Ce obiective *cantitative* stabilim pentru testare ?

S-a testat suficient sau nu ?

Idealul (... de neatins)

= testarea tuturor execuțiilor programului

⇒ adică a *tuturor căilor* prin graful de flux de control (CFG)

Dar: numărul căilor prin program e în general infinit (cicluri, etc.)

⇒ trebuie să ne multumim cu criterii structurale mai modeste

⇒ dar nu la întâmplare – trebuie alese *judicios*

Axiome ale testării (Weyuker)

Antiextensionalitate:

Există programe echivalente P și Q așa încât o anumită suită de teste T e adecvată pentru P dar nu pentru Q.

adică: programe echivalente *semantic* pot necesita teste diferite

General Multiple Change:

Există programe P și Q care au aceeași formă (structură) și o suită de teste T care e adecvată pentru P dar nu pentru Q.

adică: programe apropiate *sintactic* pot necesita teste diferite

Acoperirea la nivel de instrucțiune

statement coverage (line coverage, basic block coverage)

- Teste suficiente pentru a executa fiecare instructiune de program
- Criteriu evident necesar (cod neexecutat = cod netestat)
- De regulă evident și insuficient

```
char a[5], *s = NULL;
if (len < 5)
  s = a;
*s = 't';
```

Un test cu `len = 4` acoperă toate instrucțiunile; nu găsește eroarea

Acoperirea la nivel de ramificație

branch coverage (decision coverage)
Testează fiecare valoare posibilă a unei decizii (true/false)
Def. mai precisă: testează *și* fiecare intrare/ieșire din program
  în general, implică acoperirea instrucțiunilor
  (fiecare instrucțiune e pe o ramură; v. excepție mai jos)

În categoria deciziilor / ramificațiilor se includ:
  instrucțiuni `switch/case` (ramificații multiple)
  generarea de *excepții* (greu de testat, des neglijată)
    (fiecare excepție *posibilă* e un punct de ramificare)

Atenție: funcții sau efecte laterale în expresii de decizie
```
if (a && (b || f(x, y))
```
  nu execută `f` dacă `a` și `b` sunt adevărate
  $\Rightarrow$ în general nu implică acoperirea instrucțiunilor

Acoperirea la nivel de condiție

condition coverage

O *condiție* e o *expresie booleană elementară* dintr-o *decizie*

Necesită teste pentru fiecare valoare posibilă a fiecărei condiții

Aparent mai complexă decât decision coverage, dar NU o include!

Exemplu:

```
if (x > 5 && y == 3) /*ceva cod */
```

Două teste: `x = 6, y = 2` și `x = 4, y = 3`

generează toate valorile posibile (T și F, respectiv F și T)

Dar în ambele cazuri, decizia urmează ramura “fals”

Acoperirea mixtă decizie/condiție

condition/decision coverage

Necesită acoperirea simultană a *ambelor* criterii

Fața de cele două criterii individuale poate necesita teste suplimentare, sau doar recombinarea lor

de ex. pentru

```
if (x > 5 && y == 3) /*ceva cod */
```

se poate atinge tot cu două teste: `x = 6, y = 3`, și `x = 4, y = 2`

Poate fi insuficientă: efectul unor condiții le poate *masca* pe altele

Multiple condition coverage

Testează toate combinațiile posibile pentru *subexpresiile* deciziei

Exponentială in numarul de conditii (2^n teste pentru n condiții)
$\Rightarrow$ adesea nefezabil de implementat

Practic, toate cele 2^n combinații
– pot fi irealizabile (când condițiile nu sunt independente)
– pot fi irelevante (limbaje cu evaluare în scurt-circuit)
$\Rightarrow$ în general, cerința nu se justifică

Modified Condition/Decision Coverage

Unul din cele mai puternice criterii; dezvoltat la Boeing, e o cerință pentru software în domeniul aeronautic (standard DO-178B)
Cerințele complete pentru o suită de teste MC/DC:

- Toate punctele de intrare și ieșire din program atinse
- Fiecare decizie exercitată pe ambele ramuri
- Fiecare condiție ia ambele valori
- *Fiecare condiție e dovedită să afecteze decizia din care face parte (se păstrează fixe celălalte condiții, variind condiția de interes)*

Modified Condition/Decision Coverage (cont'd)

Construirea unei suite MC/DC:

Pentru a `&&` b sunt necesare testele: f t t f t t

Pentru a `||` b sunt necesare testele: f f f t t f

Suita rezultată e aceeași pentru evaluare scurt-circuit sau nu.

a	b	a && b	
f	t	f	(1)
t	f	f	(2)
t	t	t	(3)

Indicăm perechea de teste din suită relevantă pentru fiecare condiție:

(1) și (3) arată că a poate influența decizia

(2) și (3) arată că b poate influența decizia

Construirea unei acoperiri MC/DC

Un test e un șir de valori f și t pentru cele n variabile.
Formăm recursiv testele pentru o formulă alăturând testele (șirurile) pentru subformule.

În lista L de teste pentru o expresie, marcăm un test F cu rezultat f, și un test T cu rezultat t.

Cazul de bază: o singură variabilă: $F = \text{f}$, $T = \text{t}$, $R = \emptyset$ (doar 2 teste).

Cazul e_1 && e_2. Alăturăm testele pentru e_1 și e_2:
- combinăm lista $L_1 \setminus \{T_1\}$ cu T_2 (vor avea aceeași valoare ca în L_1, pentru că $e_2 = t$). Din acestea, marcăm $F = F_1 T_2$ ca test cu valoarea f.
- simetric, combinăm T_1 cu toată lista $L_2 \setminus \{T_2\}$
- formăm testul $T = T_1 T_2$ care dă valoarea t.

Cazul e_1 || e_2. Similar:
- formăm testul $F = F_1 F_2$ care dă valoarea f.
- combinăm testul F_1 cu lista $L_2 \setminus \{F_2\}$, și lista $L_1 \setminus \{F_1\}$ cu testul F_2.

Din acestea, marcăm $T = T_1 F_2$ ca test cu valoarea t.

Exemplu

Fie formula `a && b && (c || d && e)`

Aplicăm regula pentru && la testele $F_a = f$, $T_a = t$, $F_b = f$, $T_b = t$:

	a	b	a && b	
F_{ab}	f	t	f	(0)
	t	f	f	(1)
T_{ab}	t	t	t	(2)

cu perechile a: (0, 2) b: (1, 2)

și la fel

	d	e	d && e	
F_{de}	f	t	f	(0)
	t	f	f	(1)
T_{de}	t	t	t	(2)

perechi: d: (0, 2) e: (1, 2)

Formăm acum testele pentru `c || (d && e)`

Colorăm fragmentele care provin din testele F_c, F_{de}, T_c

	c	d	e	c || d && e	
F_{cde}	f	f	t	f	(0)
	f	t	f	f	(1)
	f	t	t	t	(2)
T_{cde}	t	f	t	t	(3)

perechi de teste

c: (0, 3)

d: (0, 2)

e: (1, 2)

Exemplu (continuare)

Combinăm testele construite pentru `a && b` și `c || d && e`.
Am marcat folosirea testelor F_{ab} și T_{ab}, T_{cde}.

	a	b	c	d	e	rez	
F	f	t	t	f	t	f	(0)
	t	f	t	f	t	f	(1)
	t	t	f	f	t	f	(2)
	t	t	f	t	f	f	(3)
	t	t	f	t	t	t	(4)
T	t	t	t	f	t	t	(5)

perechi de teste
a: (0, 4)
b: (1, 4)
c: (2, 5)
d: (2, 4)
e: (3, 4)

Din construcție reiese direct că n variabile necesită $n + 1$ teste.

MC/DC în cazuri reale

Analiza anterioară e valabilă pentru *condiții independente*

e întotdeauna posibil să generăm testele proiectate În realitate, condițiile pot fi *cuplate* (corelate)

Exemplu: `(z - x >= 3 && z - y >= 1 || y < 5) && x <= 3`

Pentru ca `z - x >= 3` să influențeze condiția ar trebui să avem:

MC/DC în cazuri reale

Analiza anterioară e valabilă pentru *condiții independente*
e întotdeauna posibil să generăm testele proiectate În realitate, condițiile pot fi *cuplate* (corelate)

Exemplu: `(z - x >= 3 && z - y >= 1 || y < 5) && x <= 3`
Pentru ca `z - x >= 3` să influențeze condiția ar trebui să avem:
`x <= 3`, și `y >= 5`, și `z - y >= 1`
Dar de aici deducem `z - x >= 3`, deci condiția nu poate fi falsă, și deci nu poate influența decizia!

$\Rightarrow$ încercând o acoperire MC/DC putem detecta dacă o condiție e scrisă prea complicat, sau are părți irelevante (o posibilă eroare de logică).

În acest caz, cum `z - x < 3` nu poate avea efect (deși poate apărea), condiția se poate rescrie înlocuind `z - x >= 3` cu *true*:

`(z - y >= 1 || y < 5) && x <= 3`

Unique-Cause MC/DC vs. Masking MC/DC

Unique-Cause MC/DC

varianta prezentată inițial, influența unei condiții trebuie demonstrată păstrând toate celelalte la fel

uneori imposibil pentru condiții cuplate

Masking MC/DC

variantă relaxată: în perechea de teste, nu toate celelalte condiții trebuie să fie la fel, dar ambele combinații trebuie să evidențieze efectul condiției analizate

Combinație

cauză unică pentru toate condițiile independente

masking pentru condițiile cuplate

Acoperirea bazată pe predicate

predicate(-complete) coverage, [T. Ball, 2004]

Criteriile anterioare: *NU* corelează între ele deciziile.
$\Rightarrow$ ex. combinații a două decizii succesive în program
$\Rightarrow$ necesar un criteriu mai apropiat de *path coverage*
(care ar acoperi toate căile de execuție)

Se identifică n *predicate* (condiții) relevante în program
Se încearcă generarea tuturor combinațiilor posibile din cele $S \cdot 2^n$
(S stări = locații în program, n predicate)
$\Rightarrow$ corelează între ele *toate* condițiile și locațiile din program

Predicate coverage: exemplu [T. Ball]

```
void partition(int a[], int n) { // assume(n>2);
  int pivot = a[0];
  int lo = 1, hi = n-1;
  while (lo <= hi) {
    while (a[lo] <= pivot)
      lo++;
    while (a[hi] > pivot)
      hi--;
  if (lo < hi)
    swap(a,lo,hi);
  }
}
```

E corect ? Detectați o eroare ?

Predicate relevante; condițiile din program:
`lo <= hi, lo < hi, a[lo] <= pivot, a[hi] > pivot`

Criterii de acoperire pentru cicluri

[Beizer, Software Testing Techniques]
Pentru cicluri simple:
– zero iterații (nu se intră în ciclu)
  suplimentar: contor negativ: se comportă corect ?
– o iterație
– două iterații (poate prinde *erori de inițializare*)
– o valoare tipică intermediară
– N-1 iterații
– N iterații
– se încearcă N+1 iterații (mai mult decât nr. maxim presupus)

Pentru minim nonzero: se încearcă min-1, min, min+1 ...

Acoperire pentru cicluri multiple [Beizer]

1. număr minim de iterații exterioare
   testați complet ciclul interior (ca și ciclu independent)
2. continuă mergând în exterior pentru cicluri:
   – cu ciclurile interioare la o valoare tipică
   – și variind ciclul curent
3. în final, variază simultan toate ciclurile de la min la max

Alte criterii legate de căi (path testing)

Boundary interior path testing

- toate căile care parcurg un ciclu o dată, fără repetiție (boundary test)
- toate căile care repetă un test, cel mult de două ori (interior test)

Linear Code Sequence and Jump (LCSAJ)

o secvență LCSAJ: cod liniar urmat de un salt

criteriu LCSAJ de lungime N: N astfel de secvențe consecutive
N = 1 asigură acoperirea instrucțiunilor; N = 2 cea a ramificațiilor (chiar mai mult)

Acoperirea prin mutații

Se încearcă modificarea deciziilor / instrucțiunilor după anumite tipare, pentru a se detecta dacă programul functionează diferit
Exemple:
– $<$ modificat în $<=$, etc.
– $+1$ modificat în -1 sau ignorat
– limite modificate cu $\pm$ 1
– `a || b` modificat în `a`, resp. `b`
(e relevant testul eliminat?); la fel pentru `a && b`

Dacă vreo mutație nu e prinsă măcar de un test:
$\Rightarrow$ fie testele sunt insuficiente
$\Rightarrow$ fie programul e potențial gresit (sau are cod irelevant)

Criterii de acoperire pentru date

Criteriile de până acum: legate de *fluxul de control* al programului
O alternativă: criterii legate de *fluxul de date* (*dataflow coverage*)
Noțiuni cheie:
– *definirea* unei variabile (**def**): un loc unde e atribuită
– *utilizarea* unei variabile (**use**): un loc unde e citită
(folosită într-o expresie, testată într-o condiție)
– diverse criterii de acoperire, ex.: all-defs, all-uses
def-use coverage: acoperire cu un caz de test pentru fiecare pereche (fezabilă) de definire-utilizare

Cât de relevantă e acoperirea ?

– complexitatea ciclomatică a CFG-ului.

Def. într-un graf, complexitatea ciclomatică e E - V + 2
(E = nr. de muchii, V = nr. de noduri)
– măsură bună pt. complexitatea unui program
– dorim acoperire mai mare pt. fragmentele mai complexe