Automatizarea testării

19 decembrie 2013

Problematica automatizării

Testarea are componente repetitive, deci se justifică automatizarea.

Problema e *evaluarea cost-beneficiu* a automatizării [Kaner]

Cer timp: crearea, verificarea funcționării, documentarea testelor

E reutilizabilă automatizarea? (Is modificarea programului)

Necesită mentenanță? (modificare GUI, internaționalizare)
$\Rightarrow$ *Automatizarea testării: tratată ca orice proces de dezvoltare*

Întârzie găsirea erorilor? (scade resursele pentru rularea testelor)

Găsește suficiente erori? Sau greul ramâne pe testarea manuală?

E suficient de puternică? Sau automatizează doar testele "ușoare"?

Exemplu: Capture-replay

1) înregistrează acțiunile utilizatorului (*mouse/tastatură*), și ecranul rezultat (bitmap) $\Rightarrow$ nivelul cel mai primitiv
   – alte verificări: cu efortul testorului (întrerupere/inserare)
   – susceptibilă la orice modificare în produs
   – posibile erori de comparare în imaginea rezultată

2) script cu *acțiuni la nivel înalt* (selectare meniu/buton)
   – mai flexibile, dar nu verifică implicit aspectul grafic (nivel scăzut: font, dimensiune/suprascriere text, etc.)

3) *limbaj de scripting* pentru a genera automat teste noi

Dezavantajele sistemului capture-replay
   Nu poate continua din erori
      $\Rightarrow$ erorile sunt găsite în procesul manual de înregistrare
      $\Rightarrow$ se automatizează doar reluarea testului "bun" (regresie)
   Nu poate defini testele *implicite* pentru om ("tot restul e OK") (nu detectează erori nespecificate, sau e inflexibilă – ex. bitmap)

Exemplu: Test monkeys

Utilitare automate care execută acțiuni la întâmplare
(fără cunoștințele unui testor despre funcționalitatea produsului)
Dumb monkeys: ignoră complet utilitatea (doar mouse/tastatură)
dar pot avea noțiuni de bază despre ferestremeniuri/butoane
Smart monkeys: au un *model* cu stări al aplicației, exercită tranziții între aceste stări

++ pot găsi uneori 10-20% din erori [Nyman, Microsoft, 2000]
++ bună acoperire preliminară (ex: 65% în 15 min, editor texte)
++ teste complet automate, fără efort uman de înregistrare
−− "dumb" monkeys nu știu ce e eroare: doar când sistemul crapă
−− $\Rightarrow$ erorile sunt dificil de înregistrat și reprodus
++ rulează independent, nesupravegheat, minim de resurse (cost)

Ce putem automatiza în testare?

Execuția/rularea testelor

ex. în orice cadru de testare de modul

utilă pentru testarea de regresie

Evaluarea testelor

= problema *oracolului* de testare: testul a trecut sau nu ?

Netrivial. Adesea necesită inspecție manuală. Riscuri:

– erori nedetectate (imprecizie, sau aceeași eroare ca programul)

– avertismente false $\Rightarrow$ costul verificării manuale

Ex: comparare de semnale continue (în industria automotive)

comparare de imagini pentru ecran/imprimantă

Generarea testelor

Relativ facil: generarea de schelete de test (declarații + apeluri)

Mai dificil: generarea inteligentă de date relevante (acoperire)

Alegerea arhitecturii de testare [Kaner]

1) Arhitectură direcționată de date (*data-driven*)
   Separă datele de structura testului (la fel ca în programe)
   Exemplu: Tabel, linii = teste; coloane = parametrii de test
   Un script generează un test pentru fiecare rând din tabel
   Minim rezonabil: acoperă fiecare *pereche* de valori la parametri
(pentru fiecare *combinație* de parametri numărul e exponențial)

2) Definirea unui *cadru* de testare (*framework-based* architecture)
   O bibliotecă de funcții separă testarea de interfața utilizator
   Ex: `open(`*`file`*`)`, independent de acțiunile pentru deschidere
   (meniu, clic pe buton, tastatură, etc.)
   ++ reutilizare pentru acțiunile frecvent folosite
   ++ nivel de indirectare $\Rightarrow$ izolare de utilitarul de testare
   -- costisitor, amortizare doar la versiuni ulterioare

Testarea pornind de la specificații

Automatizabilă (/keykeyword testing) pentru specificații în limbaj definit
Pornind de la documentație: specificație tabelară, ex. [Pettichord]

Test ID	Operation	Table	Name	Type	Nulls
dtbed101	Add Col	TB03	NEW_INT_COL	CHAR(100)	Y

Important: format suficient de ușor inteligibil pentru specificator
Un translator/interpretor de test generează din tabel driverul de test sau face interfața cu utilitarul comercial de testare folosit
++: axat pe cerințe (*ce*, nu *cum*), independență de implementare și utilitar de testare, auto-documentat

Avansat: generare automată din specificații în limbaj formal (ex. tabele de decizie în RSML, la protocolul de aviație TCAS-II)

Testarea bazată pe modele

Modele: automate finite, UML, Statecharts (automate ierarhice), Message Sequence Charts, automate cu timp, rețele Petri, lanțuri Markov...

Criterii de generare: acoperirea satisfăcătoare a modelului
toate stările / tranzițiile; combinații de k tranziții consecutive (*k-switch cover*)

++ facilitează generarea testelor relevante
−− investiție în construirea și mentenanța modelului

Testare prin *model checking*
prin *explorarea spațiului stărilor*, pornind de la specificații:
1) se pune întrebarea: poate modelul să ajungă într-o stare dată ?
2) dacă da, un *model checker* va genera un exemplu = caz de test

Testarea pornind de la implementare: execuție simbolică

Scop: exercitarea programului, satisfăcând un *criteriu de acoperire*
$\Rightarrow$ necesită: instrumentarea pentru măsurarea acoperirii
Cum: *combinație de căi* alese la întâmplare + direcționat
(pentru atingerea ramurilor neparcurse încă)

Execuție simbolică: execuție a programului folosind *expresii* (cu variabile = simboluri), spre deosebire de valori numerice

În execuția simbolică se reține constrângeri (path conditions) corespunzând ramificațiilor urmate de program

Realizabilitatea condițiilor se verifică cu utilitare specializate (satisfiability checkers, constraint solvers)
$\Rightarrow$ generează date de test care vor exercita calea respectivă
sau demonstrează că nu e fezabilă $\Rightarrow$ oprește explorarea acelei căi

Execuția simbolică și testarea programelor

inițial descrisă de James C. King (1976)

programul e executat (de un interpretor special) folosind intrări *symbolice*
⇒ arbore de execuție simbolică
cu *condiții de cale* (path conditions) care acumulează condițiile de la fiecare ramificare/decizie
execuția se oprește când condiția devine inconsistentă (nerealizabilă)

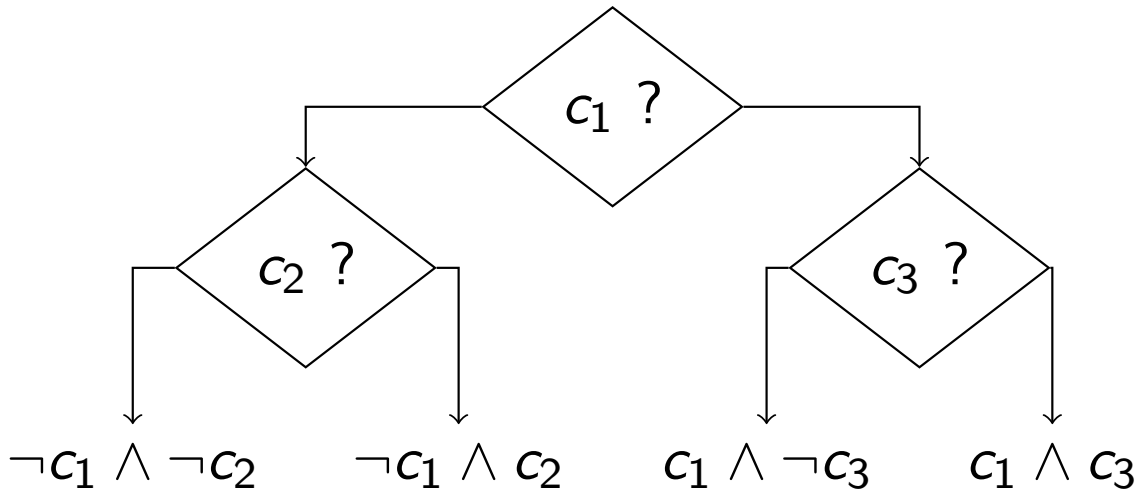
Scopul generării de teste
factor ridicat de *acoperire*
uneori, atingerea unei anumite ramuri

Larg cercetată (150+ articole), tehnică matură, de succes
Utilitare: Java Pathfinder, (j)CUTE, CREST, KLEE, Pex, SAGE, ...
pentru C, C#, Java, mai recent JavaScript

Execuția simbolică: variante

Execuție integral simbolică

explorează independent fiecare cale de execuție



Problema: toată semantica limbajului trebuie exprimată în formule și un rezolvitor pentru formule arbitrare $\Rightarrow$ imposibil

matematică complicată, funcții de bibliotecă, interacțiuni cu mediul

ex. KLEE: modele pentru 40 de apeluri sistem (2500 LOC)

Execuție simbolică dinamică (concolică)

Execuția *simbolică* e direcționată de cea *concreteă*

de aici numele *concolic*

Când modelarea simbolică ar fi prea complicată, înlocuiește pasul respectiv cu execuție concretă

ex. aritmetică neliniară, funcții de bibliotecă

function explore($pc = [c_1, c_2, \ldots, c_n]$)
for $k = n$ downto 1 **do**
    inputs = solve $pc = c_1 \wedge \ldots \wedge c_{k-1} \wedge \neg c_k$ (flip c_k)
    rerun with new inputs; capture new pc'
    explore(pc')

Problema: înlocuind cu valori concrete, se poate să nu mai atingem calea dorită

Concretizarea ca obstacol în execuția simbolică

```
y = hash(x); // nu stim ce face hash => y e concret
if (x + y > 0)
  // calea 1
else
  // calea 2
```

Fie prima intrare: x = 20; y = hash(20) = 13 $\Rightarrow$ *calea 1*
Pentru *calea 2*, negăm $x + y > 0$, cu y concret (constanta 13)
Rezolvitorul poate returna (de ex.) x = -15
dar putem avea hash(15) = 27 și $x + y > 0$ (nu putem prezice)
$\Rightarrow$ execuția urmează *tot calea 1*

$\Rightarrow$ *reîncercare*; în cel mai rău caz, devine testare aleatoare

Când e eficientă automatizarea ?

În *testarea de regresie*: necesită doar stocarea testelor și a rezultatelor așteptate + compararea automată a rezultatelor

Testarea interfețelor utilizator (v. discuția anterioară)

Testarea translatoarelor:

generarea automată de intrări pornind de la gramatica limbajului
se explorează aleator/statistic combinații de neterminale

Load/stress testing: aleatoare, relevantă e cantitatea, nu conținutul

Fuzz testing: generarea de cantități mari de intrări aleatoare/ostile, pentru a detecta erori de validare / vulnerabilități de securitate

Ex: RANDOOP [Microsoft]: 4 mil. teste în 150 ore CPU / 15 ore-om
30 de erori în cod testat 200 pers.-an, cu 20 erori/an găsite manual
v. și `http://research.microsoft.com/en-us/projects/Pex/`

Automatizarea depanării

După automatizarea detecției $\Rightarrow$ sprijinul în *localizarea* erorilor

Minimizarea intrărilor de eroare

prin căutare binară, determinând jumătatea de intrare care cauzează eroarea (ex. în cazul programelor cu intrări fișiere)

Minimizarea diferențelor între o rulare corectă și una eronată

tot căutare binară, pentru două intrări cât mai apropiate

Localizarea erorilor *în spațiu*

comparând în depanator starea de execuție între rulări corecte și eronate

detectând (precis sau statistic) invarianți/tipare corecte, violate de execuția eronată

Localizarea erorilor *în timp*

prin compararea secvențelor eronate și identificarea punctelor în care variabile *infectate* încep să difere / afecteze ieșirea

Delta debugging [Zeller]: implementare parțial automată a acestora