Osztott jáva programok automatikus tesztelése

Matkó Imre
BBTE, Kolozsvár
Informatika szak, IV. Év
2007 – január

Tartalom

- Osztott alkalmazások
- Automatikus tesztelés
- Tesztelés – heurisztikus zaj keltés
- Tesztelés – genetikus algoritmus
- Tesztelés – jelölések
- Tesztelés – alkalmassági függvény
- Tesztelés – genetikus keresés
- Tesztelés – következtetések
- Források

Osztott alkalmazások

A több szálas adatfeldolgozás illetve a hálózatok fejlődése és az internet elterjedése miatt már lépten-nyomon találkozunk velük:

- Szerver technológiák
- Osztott adatbázis
- Hálózati játékok
- Osztott (párhuzamosan szerkeszthető) dokumentumok
- Stb. stb. stb.

Osztott alkalmazások

A folyamatok, futási szálak ütemezése egy nem determinisztikus program futást eredményez, ami konkurencia problémákhoz vezethet, ezek vizsgálata fontos.

Adott programok esetén az összes lehetséges ütemezés vizsgálata túl költséges és fölösleges, mivel csak néhány esetben léphet fel holtpont vagy verseny az erőforrásokért.

Automatikus tesztelés

Egyszerű tesztelés: sokszor futtassunk le egy tesztet az osztott alkalmazásra.

Probléma: a JVM ütemező többnyire determinisztikusan viselkedik, ezért sok eset kimarad. Nehéz a hiba pillanatának az állapotát reprodukálni (mikor, hogyan vált az ütemező?) .

Megoldás: A teszthez generáljunk „zajos” kódot, vagyis helyezzünk el a kritikus pontokba szál / folyamat váltást kényszerítő elemeket. (pl. yield(), wait(), sleep(), suspend())

Tesztelés – heurisztikus zaj keltés

Ilyen „zaj” műveleteket helyezünk el a kód néhány olyan része elé vagy után, amelyek osztott műveleteket végeznek.

Néhány lehetséges elhelyezés nem befolyásolja a hibák előfordulását, illetve lehet, hogy éppen nem is futtatható. Ezért az összes ilyen hely kiválasztása, illetve a megfelelő típusú „zaj” meghatározása, „a-priori” nagyon bonyolult. A gyakorlatban ezt véletlenített algoritmusok végzik.

Tesztelés – heurisztikus zaj keltés

Esemény – valamilyen típusú szál váltást kényszerítő művelet beékelése a programba

Jó esemény – növeli az osztott hibák előfordulásának a valószínűségét

Rossz esemény – csökkenti egy osztott hiba előfordulásának a valószínűségét

Semleges esemény – nem befolyásolja a hibák előfordulását

Tesztelés – heurisztikus zaj keltés

Példa:

Szál1	**Szál2**
(1) If (x != 0) {	(3) x = 1;
(2) y = 1 / x;	(4) x = 0;
}	(5) z = 8;

Az (1) és (3) események jók, mert növelik a nullával való osztás valószínűségét.
Ha az első szálba „ if (x == 0) ” -t írunk, az (1) rossz eseménnyé válik.

Tesztelés – genetikus algoritmus

Az egyszerű véletlenített algoritmusok viszonylag kis szignifikanciával állapítják meg az egyes hibák előfordulásának a valószínűségét és néhány eset még így is kimarad.

Ennek a javítására, a feladatot keresési feladatként is felfoghatjuk és ekkor egy genetikus algoritmust alkalmazunk a legjobb tesztesetek előállítására.

Tesztelés – genetikus algoritmus

Genetikus algoritmus:

Adott egy populáció – itt a tesztesetek (a zaj egy-egy lehetséges elosztása) egy véletlenszerűen előállított halmaza. A halmaz elemei jelképezik a kromoszómákat.

Egy tesztesethez egy megfelelőségi függvényt, heurisztikát rendelünk: mennyire szolgálja az adott eset az elérni kívánt célokat.

Tesztelés – genetikus algoritmus

A fejlődés során a legjobb egyedek fennmaradnak és új, mutációkat szenvedett egyedek jelennek meg, a legéletképtelenebbek elpusztulnak illetve az örökítőanyagok keveredésével szaporodnak az életben maradt egyedek.

Tesztelés – genetikus algoritmus

Az algoritmus céljai:

(1) növeljük a valószínűségét annak, hogy egy hiba minden generált teszt eset futtatásánál előfordul
(2) minél több információt adjunk a felhasználónak a hibákról

Tesztelés – jelölések

Legyen $\mathcal{P}$ a jáva programok halmaza, ahol minden $p \in \mathcal{P}$ egy teszt eset.

Minden p-re definiáljuk a következő halmazokat:

ν - a program változói

S_V- értékpárok, a változók halmazára a programban, a hozzáférés előtti és utáni értékkel

$\mathcal{L}$ – program részletek halmaza, értékpárokkal – végrehajtás előtti és utáni hely

Tesztelés – jelölések

$\mathcal{H}$ – a felhasználható esemény típusok (pl. yield(), wait())

$\mathcal{N}$- a lehetséges zaj erősségek (pl. $\mathcal{N}$= {0..100})

$S_L = \mathcal{L} \times \mathcal{H} \times \mathcal{N}$

$S_V = \nu \times \mathcal{H} \times \mathcal{N}$

$S_S = S_V \times \mathcal{H} \times \mathcal{N}$

Legyen $\mathcal{C}$ a három S halmaz diszjunkt uniója.

Ekkor $\mathcal{C}$ a lehetséges tesztesetek halmaza lesz (az összes esemény elhelyezéseinek a módjai) .

Tesztelés – jelölések

Legyen $C_i(t)$ az i. konfiguráció végrehajtásainak (a teszteset futtatásának) száma t lépés után. Röviden jelöljük C_i -vel.

Legyen $f_b(C_i)$ a b. hiba megjelenéseinek a száma t. végrehajtás után.

Legyen $N_{i,j}$ az i. konfiguráció j. eseményére vonatkozó $\mathcal{N}$.

Legyen $C_{i,j}$ az i, konfiguráció j. eseményének a bekövetkezéseinek a száma.

Tesztelés – alkalmassági függvény

Alkalmassági függvény:

(1) $$méret = \sum_{j=1}^{|C_i|} N_{i,j} * elemMéret(C_{i,j})$$

elemMéret – minden j-hez egy tetszőleges, előre adott értéket rendel

méret – az alkalmazott zaj erősségek súlyozott összege

(2) $$entrópia = -\sum_{j=1}^{|C_i|} \frac{N_{i,j}}{méret(C_i)} \log\left(\frac{N_{(i,j)}}{méret(C_i)}\right)$$

entrópia – a zaj eloszlása

Tesztelés – alkalmassági függvény

Alkalmassági függvény:

(3) $f_s = méret(C_i) + \frac{1}{entrópia(C_i)}$

a méret és az entrópia közötti összefüggés

(4) $f_p = f_b \frac{(C_i)}{C_i(t)} + \log(C_i(t))$

hibák előfordulásának a valószínűsége

Tesztelés – genetikus keresés

- 1. Véletlenszerűen előállítunk néhány teszt esetet
- 2. A fenti halmaz egy generáció,amely programjait egy előre megadott szám-szor futtatjuk.
- 3. A megfelelőség függvény alapján rendezzük a kromoszómákat.
- 4. A legjobbakat megtartjuk, a leggyengébbeket töröljük. Új genetikus információt hozunk be, véletlenszerűen előállított kromoszómákkal, illetve néhány életben maradt kromoszómát párosítunk, ezzel mutációkat hozva létre.
- 5. Előre megadott számú generációt állítunk elő.

Tesztelés – következtetések

Futás közben érdemes figyelni:

- Minden generáció legjobb kromoszómáját.
- Generációnként a kromoszómák megfelelőség átlagát.
- Átlagosan hány kromoszóma maradt változatlanul az egymásutáni generációkban.
- A kromoszómák életkorát.

Az algoritmus a gyakorlatban az olyan populáció előállítása felé konvergál, amelynek a kromoszómái, tesztesetei csak, a hibás részeket jelölik.

Források

[1] Concurrent Java Test Generation as a Search Problem -Yaniv Eytani www.elsevier.nl/locate/entcs, cs.haifa.ac.il/~ieytani/

[2] Heuristics For Finding Concurrent Bugs – Yaniv Eytani, Eitan Farchi, Yosi Ben-Asher - Workshop on Parallel and Distributed Testing and Debugging, Nice, 2003

[3] Yaniv Eytani, and Shmuel Ur, “Compiling a Benchmark of Documented Multi-threaded Bugs,” Workshop on Parallel and Distributed Testing and Debugging, Santa Fe, 2004

[4] Multithreaded unit testing with ConTest – Yarden Nir-BuchBinder, Shmuel Ur – 2006 Apr 04.

[5] Operating systems – William Stallings, Prentice hall – Fourth edition; Part Two, Part Four

[6] JavaBy Example - Clayton Walnum , Que Corporation, Macmillan Computer Publishing