Semantika višedretvenih programa - nastavak

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

14. prosinca, 2022.

Semantika višedretvenih programa

Svaki objekt uz pridruženi monitor ima i **pridruženi skup čekanja dretvi**. Pri kreiranju objekta, skup čekanja je prazan. Akcije koje dodaju/uklanjaju dretvu iz skupa čekanja su **atomarne (nedijeljive)**. Skupovima za čekanje se može upravljati jedino kroz metode `Object.wait`, `Object.notify` i `Object.notifyAll`. Na skup čekanja može utjecati i **status prekida** (eng. *interrupt status*) dretve, te funkcije klase koje obrađuju signale prekida. Metode za pauziranje i spajanje dretvi imaju svojstva izvedena iz metoda za čekanje (`wait`) i dojavljivanje (`notify`).

Akcije čekanja se događaju nakon poziva metode `wait()` ili varijanti `wait(long milisecs)` i `wait(long milisecs, int nanosecs)` (zadnje dvije varijante metode s parametrima 0 su ekvivalentnte prvoj). Dretva izlazi iz čekanja normalno ukoliko se na javi `InterruptedException`.

Neka je d dretva koja izvršava `wait` na objektu m i neka je n broj poziva akcije zaključavanja od strane dretve d na monitoru objekta m za koje nije pozvana odgovarajuća akcija otključavanja. Tada se događa jedno od sljedećeg:

Semantika višedretvenih programa

- Ukoliko je $n = 0$, prijavljuje se `IllegalMonitorStateException`.
- Ukoliko se radi o vremenski ograničenoj `wait` naredbi i argument nanosekundi nije u rasponu $0 - 999999$ ili je argument milisekundi negativan prijavljuje se `IllegalArgumentException`.
- Ukoliko je dretva d prekinuta, tada se prijavljuje `InterruptedException` i status prekinutosti dretve se postavlja na `false`.
- Inače se izvršava sljedeći niz akcija:
  - Dretva d se dodaje u skup čekanja objekta m i izvršava n poziva otključavanja monitora objekta m.
  - Dretva d ne izvodi naredbe dok se ne ukloni iz skupa čekanja objekta m. Dretva se može ukloniti iz skupa čekanja ukoliko se dogodi neka od sljedećih akcija:
    - Izvršena je naredba dojavljivanja (`notify`) na objektu m kojom je d odabrana za uklanjanje iz skupa čekanja.
    - Izvršena je naredba `notifyAll` na objektu m.
    - Došlo je do izvršavanja akcije koja prekida dretvu d.
    - Kod vremenski ograničene naredbe `wait`, d se uklanja nakon *milisec* milisekundi i *nanosec* nanosekundi od početka izvođenja naredbe.

Semantika višedretvenih programa

- (nastavak)
  - Može doći do interne akcije implementacije (dozvoljeno je da interna implementacija samostalno pozove izbacivanja iz skupova za čekanje - iako to nije preporučeno).

  Svaka dretva određuje poredak događaja (akcija) koji mogu uzrokovati njezino izbacivanje iz skupa čekanja. Poredak ne mora biti konzistentan s drugim poretcima, ali dretva se mora ponašati kao da su se događaji dogodili u tom (odabranom) poretku. Npr. ukoliko je d u skupu čekanja od m te se istovremeno dogodi akcija prekida dretve i dojavljivanja, tada mora biti definiran poredak izvršavanja naredbi. Ukoliko se prekid nađe prvi u poretku, tada se d uklanja sa skupa čekanja i prijavljuje `InterruptedException` a neka druga dretva u skupu čekanja m (ukoliko postoji) mora dobiti dojavu za napuštanje skupa čekanja. Ukoliko je dojava prva u poretku, tada se d normalno uklanja iz skupa čekanja a tek nakon se izvršavaju naredbe vezane uz prekid dretve.
- Dretva d izvršava n naredbi zaključavanja monitora od m.

Semantika višedretvenih programa

- Ukoliko je d izbačen iz skupa čekanja objekta m zbog prekida (`Interrupt`), tada se status prekida od d postavlja na `false` i metoda `wait` prijavljuje `InterruptedException`.

Dojavljivanje se izvršava nakon poziva metoda `notify` i `notifyAll`. Za dretvu d koja izvršava jednu od naredbi dojavljivanja na objektu m i koja je pozvala n naredbi zaključavanja na m za koje nije pozvana odgovarajuća naredba otključavanja vrijedi nešto od:
- Ako je $n = 0$ prijavljuje se `IllegalMonitorStateException`. To se događa jer d ne posjeduje lokot monitora objekta m.
- Za $n > 0$ i naredbu `notify`, te neprazan skup čekanja objekta m, izabire se dretva u, element skupa čekanja m i uklanja se iz skupa čekanja. Ne postoji poredak uklanjanja dretvi iz skupa čekanja. u izvršava naredbe vezane uz objekt m i nastavlja izvršavanje. Daljnje naredbe zaključavanja lokota monitora od m mogu uspjeti tek nakon što d u potpunosti otključa lokot monitora od m.

Semantika višedretvenih programa

- Za $n > 0$ i naredbu `notifyAll`, sve dretve iz skupa čekanja objekta m nastavljaju izvršavanje. Samo jedna dretva u danom trenutku može zaključati monitor objekta m potreban za izvršavanje naredbi nad zaštičenim resursom.

Akcije prekida se događaju nakon poziva naredbe `Thread.interrupt` kao i naredbi koje ju pozivaju (npr. `ThreadGroup.interrupt`). Neka je d dretva koja poziva naredbu `u.interrupt` za neku dretvu u, gdje može biti i $d = u$. Nakon poziva se status prekida dretve u postavlja na `true`. Ukoliko postoji objekt m čiji skup čekanja sadrži u, tada se u uklanja iz skupa čekanja od m. Nakon nastavka izvršavanja i ponovnog zaključavanja monitora od m, `wait` će prijaviti `InterruptedException`. Poziv `Thread.isInterrupted` može utvrditi status prekida dretve. Statička metoda `Thread.interrupted` se može koristiti od strane dretve za utvrđivanje i poništavanje stanja prekida.

Semantika višedretvenih programa

Interakcija naredbi `wait`, `notify` i `interrupt`:
Ukoliko je dretva i dobila dojavu i prekinuta dok čeka u skupu čekanja objekta ona može ili:

- normalno prekinuti čekanje ali i dalje imati neobrađen zahtjev za prekid (poziv metode `Thread.interrupted` bi vratio `true`).
- prekinuti čekanje uz prijavu `InterruptedException`.

Dretva ne može resetirati svoj status prekida i normalno prekinuti čekanje u redu čekanja. Dojave se ne mogu izgubiti zbog prekida. Za skup dretvi s u skupu čekanja objekta m, ukoliko neka dretva pozove `notify` na m, a) barem jedna dretva prekida čekanje normalno ili b) sve dretve prekidaju čekanje i javljaju `InterruptedException`.

`Thread.sleep` **privremeno obustavlja izvođenje dretve na određeno vrijeme**. Dretva **ne gubi vlasništvo nad monitorima** a nastavak izvođenja ovisi o sustavu za dodjelu zadataka i dostupnosti procesora.
`Thread.yield` **dojavljuje sustavu za dodjelu zadataka da je spremna osloboditi procesor**, ali bi htjela biti **ponovno dodijeljena procesoru što je prije moguće**.

Semantika višedretvenih programa

`Thread.sleep` i `Thread.yield` nemaju definiranu semantiku sinkronizacije. Prevodioc ne mora zapisati sve informacije iz registara u glavnu memoriju prije poziva tih naredbi niti ne mora ponovno učitati vrijednosti koje su bile u registrima nakon poziva tih naredbi.

```
1 while (!this.done)
2     Thread.sleep(1000);
```

Primjer svojstva naredbi sleep i yield.

Ukoliko `this.done` nije `volatile`, prevodioc može iskoristiti optimizaciju i učitati vrijednost tog polja u registre ili *cache* memoriju procesora. Zbog toga, bilo koja promjena navedenog polja od strane bilo koje druge dretve nebi bila vidljiva i petlja bi se potencijalno izvodila bez prekida.

Memorijski model

Za zadani **tijek izvršavanja programa** (uređeni niz izvršenih naredbi u programu), **memorijski model** opisuje **da li tijek izvršavanja predstavlja ispravno izvršen program**. Memorijski model programskog jezika *Java* **provjerava svako čitanje u tijeku izvršavanja** i provjerava je li **zapisana vrijednost** koja je pročitana naredbama čitanja **u skladu sa zadanim skupom pravila**. Memorijski model **opisuje moguće ponašanje programa**. Implementacija može proizvesti **proizvoljan kod**, međutim **sva konačna izvršavanja** moraju proizvesti rezultat koji je **predviđen od strane memorijskog modela**.

Izvršavanje svake dretve u **izolaciji** mora slijediti **semantiku te dretve** (semantika unutar dretve - **semantika izvršavanja jednodretvenih programa**, omogućava potpuno opisivanje ponašanja dretve koristeći informaciju o vrijednostima učitanim od strane naredbi čitanja te dretve), međutim **vrijednosti koje se čitaju naredbama za čitanje se određuju memorijskim modelom**.

Memorijski model

Određivanje korektnosti naredbe dretve *d* se provodi tako da se **evaluira implementacija** kao da se izvodi u kontekstu **jednodretvenog izvršavanja**. Svaki puta kada *d* izvršava naredbu koja ima **među-dretveni (globalni) učinak**, **mora odgovarati među-dretvenoj naredbi** *a* dretve *d* koja dolazi sljedeća u tijeku programa. Ukoliko je *a* naredba čitanja, pročitana vrijednost (određena od strane memorijskog modela) se koristi za daljnju evaluaciju dretve *d*.

Memorija koju dijele dretve se zove **dijeljena memorija** ili **memorija hrpe**. Svi elementi članovi klasa, statička polja i elementi polja su spremljeni na hrpi dok lokalne varijable, formalni parametri metoda i parametri iznimki nisu dijeljeni među dretvama. Dva pristupa (čitanje ili pisanje) su u **konfliktu** ukoliko je **barem jedan pristup pisanje**.

Memorijski model

Postoji **nekoliko vrsta akcija** između dretava koje se mogu dogoditi u programu:
- Čitanje varijable (normalno ili ne `volatile`)
- Upisivanje vrijednosti u varijablu (normalno ili ne `volatile`)
- Akcije sinkronizacije:
  - `Volatile` čitanje
  - `Volatile` pisanje
  - Zaključavanje (monitora)
  - Otključavanje (monitora)
  - Umjetna prva ili zadnja akcija dretve.
  - Akcije koje pokreću dretvu ili detektiraju kraj izvršavanja dretve.
- Vanjske akcije - akcije koje se mogu uočiti i van izvođenja (rezultat je baziran na okruženju van izvođenja, npr. pozivi preko JNI).
- Akcije divergencije - događaju se kada dretva izvršava beskonačnu petlju ali ne koristi memoriju, sinkronizaciju ili eksterne akcije (u tom slučaju se može dogoditi da dretva blokira ostale dretve).

Memorijski model

Akcija a je **uređena četvorka** (d, t, v, id), gdje je d dretva, t tip akcije, v varijabla (oznaka označava i monitor pridružen objektu, ukoliko je akcija čitanje - varijablu koja se čita a ukoliko je akcija pisanje - varijablu u koju se piše), te id (jedinstveni identifikator akcije). Eksterne akcije su reprezentirane kao **uređene petorke**, gdje zadnja komponenta sadrži rezultat koji uočava dretva koja izvodi akciju (može sadržavati i informaciju o (ne)uspješnom izvođenju). **Kod izvršavanja koje ne završavaju** ne mogu se identificirati sve eksterne akcije.

Uređaj programa dretve d je onaj **potpuni uređaj** (od svih mogućih među-dretvenih akcija svake dretve d) koji **reprezentira uređaj** u kojem bi se te akcije izvršile u skladu sa **semantikom individualnog izvršavanja** (u izolaciji) dretve d.

Memorijski model

Skup akcija je **sekvencijalno konzistentan** ako se sve akcije javljaju u **potpunom uređaju** (poretku izvršavanja), **konzistentan je s uređajem programa** i svako čitanje r varijable v **ima informaciju o vrijednosti upisanoj u** v od strane pisanja w tako da:
- w se u poretku izvršavanja nalazi **prije** r,
- ne postoji drugo pisanje w' takvo da se w dogodi prije w' i w' prije r u poretku izvršavanja.

Kod sekvencijalne konzistentnosti je **svaka akcija atomarna** (nedjeljiva i vidljiva svim dretvama). Ukoliko u programu nema **stanja utrkivanja za pristup resursima** (eng. *data races*), tada će **sva izvršavanja** biti sekvencijalno konzistentna. Sekvencijalna konzistentnost i/ili odsustvo stanja utrkivanja za pristup resursima i dalje **ne otklanjaju pogreške koje se događaju kada bi se grupa operacija trebala izvršiti kao atomarna operacija ali to nije** (npr. primjena inkrement/dekrement operatora).

Memorijski model

Svako izvršavanje ima **uređaj sinkronizacije** (potpuni uređaj nad svim akcijama sinkronizacije kod izvršavanja). Uređaj sinkronizacije sinkronizacijskih akcija dretve d je **konzistentan s uređajem programa**. Akcije sinkronizacije induciraju `sinkroniziran-s` relaciju na akcijama. Ta relacija se definira na sljedeći način:

- Akcija otključavanja monitora od m je u relaciji `sinkroniziran-s` sa svim daljnjim akcijama zaključavanja lokota od m (daljnji je u skladu sa uređajem sinkronizacije).
- Pisanje u varijablu v koja je `volatile` je u relaciji `sinkroniziran-s` sa svim daljnjim akcijama čitanja varijable v bilo koje dretve (daljnji je u skladu sa uređajem sinkronizacije).
- Akcija koja pokreće dretvu je u relaciji `sinkroniziran-s` sa prvom akcijom u dretvi koju pokreće.
- Pisanje standardne vrijednosti (nula, `false` ili `null`) u proizvoljnu varijablu je u relaciji `sinkroniziran-s` sa prvom akcijom u svakoj dretvi.
- Završna akcija dretve d_1 je u relaciji `sinkroniziran-s` sa proizvoljnom akcijom dretve d_2 koja detektira da je d_1 završila izvođenje.

Memorijski model

- Ukoliko dretva d_1 pozove prekid dretve d_2, prekid od d_1 je u relaciji `sinkroniziran-s` sa proizvoljnom naredbom u kojoj proizvoljna dretva (uključujući d_2) otkrije da je u statusu prekida (može se dogoditi prijavljivanjem `InterruptedException` ili pozivom `Thread.interrupted`, `Thread.isInterrupted`.

Izvor brida relacije `sinkroniziran-s` se zove *otpuštanje* (eng. *release*) a odredište `pribavljanje` (eng. *acquire*).

Dvije akcije mogu biti uređene `dogodilo-se-prije` (eng. *happen-before*) relacijom. Ukoliko je jedna akcija u relaciji `dogodilo-se-prije` druge akcije, tada je **prva vidljiva drugoj i u uređaju prije nje**. Koristimo oznaku $dp(x, y)$ da označimo da je x u relaciji `dogodilo-se-prije` s y.
- Ukoliko su x i y akcije iste dretve i x se javlja prije y u uređaju programa, tada vrijedi $dp(x, y)$.
- Postoji `dogodilo-se-prije` brid od završetka konstruktora do početka destruktora objekta.

Memorijski model

- Ukoliko je akcija x `sinkroniziran-s` sljedećom akcijom y tada vrijedi i $dp(x, y)$.
- Ako $dp(x, y)$ i $dp(y, z)$ tada $dp(x, z)$.

`wait` metoda klase `Object` ima **povezane akcije zaključavanja i otključavanja** (njihove `dogodilo-se-prije` relacije su definirane preko tih povezanih akcija).

Prisutnost `dogodilo-se-prije` relacije između dvije akcije **ne mora nužno povlačiti** da se one moraju odvijati u tom poretku u implementaciji. **Dozvoljeno** je da dođe do **promjena u poretku** ukoliko su one **konzistentne s ispravnim izvođenjem** (npr. inicijalizacija članova kreiranog objekta se ne mora odvijati prije početka izvođenja dretve pod uvjetom da nema čitanja tih članova prije početka izvođenja dretve). Ukoliko su dvije akcije u `dogodilo-se-prije` relaciji, to **ne mora biti uočljivo iz koda s kojim te akcije nisu u toj relaciji**.

Memorijski model

Npr. ukoliko je došlo do stanja utrkivanja za pristup resursima između pisanja iz jedne dretve i čitanja iz druge dretve, tada pisanja mogu biti van relacije `dogodilo-se-prije` za navedena čitanja.

Skup sinkronizacijskih bridova S je **dovoljan** ukoliko je **minimalan skup** takav da **tranzitivno zatvorenje** od S uz **uređaj programa** određuje sve `dogodilo-se-prije` bridove izvršavanja. Taj skup je **jedinstven**.

Iz gornjih definicija slijedi:
- Otključavanje monitora je u relaciji `dogodilo-se-prije` sa svakim daljnjim zaključavanjem tog monitora.
- Pisanje u element član klase koji je `volatile` je u `dogodilo-se-prije` relaciji sa svakim daljnjim čitanjem tog elementa člana klase.
- Poziv `start()` metode na dretvi je u relaciji `dogodilo-se-prije` sa svakom akcijom u pokrenutoj dretvi.
- Sve akcije u dretvi su u relaciji `dogodilo-se-prije` od uspješnog vraćanja neke druge proizvoljne dretve iz poziva naredbe `join()` na toj dretvi.

Memorijski model

- Standardna inicijalizacija proizvoljnog objekta je u relaciji `dogodilo-se-prije` u odnosu na bilo koju drugu akciju u programu (osim standardnog-pisanja pri inicijalizaciji).

Kada program sadrži dva konfliktna pristupa koji **nisu uređeni** `dogodilo-se-prije` relacijom, **kažemo da program ima stanje utrkivanja za pristup resursima**.

Operacije koje **nisu među-dretvene** kao **čitanje duljine polja**, **primjena operatora pretvorbe uz provjeru tipova** i **pozivi virtualnih metoda nisu direktno podložni stanju utrkivanja za pristup resursima**.

Program je točno sinkroniziran ako i samo ako **sva sekvencijalno konzistentna izvršavanja nemaju stanja utrkivanja za pristup resursima**.

Memorijski model

Kažemo da čitanje r varijable v **može uočiti** pisanje w u v ako u `dogodilo-se-prije` parcijalnom uređaju izvršavanja:
- r nije u poretku prije w (ne vrijedi $dp(r, w)$).
- ne postoji pisanje w' u v takvo da $dp(w, w')$ i $dp(w', r)$,

Skup akcija A je `dogodilo-se-prije` **konzistentan** ako za sva čitanja $r \in A$, gdje $w(r)$ označava akciju pisanja koju r uočava, ne vrijedi $dp(r, w(r))$ niti da postoji pisanje $w \in A$ takvo da $w.v = r.v$ i $dp(w(r), w)$ i $dp(w, r)$.
Kod `dogodilo-se-prije` konzistentnih akcija, svako čitanje uočava pisanje koje smije vidjeti prema `dogodilo-se-prije` uređaju.

Primjer `dogodilo-se-prije` konzistentnog koda koji nije sekvencijalno konzistentan

Dretva D_1	Dretva D_2
$B = 1;$	$A = 2;$
$r_2 = A;$	$r_1 = B;$

U primjeru gore su A i B dijeljene varijable (očito imamo utrkivanja za pristup resursima, stoga odgovarajuća pisanja/čitanja nisu u `dogodilo-se-prije` relaciji).

```
1: B = 1;
3: A = 2;
2: r2 = A;  // vidi inicijaliziranu vrijednost 0
4: r1 = B;  // vidi inicijaliziranu vrijednost 0
```

Program nije sinkroniziran pa čitanja mogu vidjeti ili inicijaliziranu vrijednost ili vrijednost zapisanu od strane druge dretve.

Primjer dogodilo-se-prije konzistentnog koda koji nije sekvencijalno konzistentan

```
1: r2 = A; // vidi zapisanu vrijednost A=2
3: r1 = B; // vidi zapisanu vrijednost B=1
2: B = 1;
4: A = 2;
```

Program nije sinkroniziran pa čitanja mogu vidjeti ili inicijaliziranu vrijednost ili vrijednost zapisanu od strane druge dretve.

U ovom primjeru je prvo došlo do optimizacije prevodioca (gledano u izolaciji je svejedno hoće li se prvo izvršiti $B = 1$ ili $r_2 = A$). Pošto nema sinkronizacije u programu, svako čitanje može vidjeti ili inicijaliziranu vrijednost ili upisanu vrijednost od strane druge dretve. Međutim, program je i dalje dogodilo-se-prije konzistentan!

Izvođenje

Izvođenje I je **opisano uređenom osmorkom** $(P, A, pu, su, UP, ZV, ss, dp)$, gdje P predstavlja program, A skup akcija, pu uređaj programa, su uređaj sinkronizacije, UP funkciju uočenog pisanja (za svako čitanje $r \in A$, vraća $UP(r)$ akciju pisanja koja je uočena od $r \in I$), ZV funkciju upisanih vrijednosti koja za svako pisanje $w \in A$ vraća $ZV(w)$ - upisanu vrijednost $w \in I$, ss je relacija `sinkroniziran-s`, dp je relacija `dogodilo-se-prije`.
Izvršavanje I je `dogodilo-se-prije` **konzistentno** ukoliko je **njegov skup akcija** `dogodilo-se-prije` **konzistentan**.

Izvršavanje I je **dobro utemeljeno** ako vrijedi:
- **Svako čitanje uočava pisanje u istu varijablu tijekom izvršavanja**.
Sva čitanja i pisanja varijabli koje su `volatile` su `volatile` akcije. Za sva čitanja $r \in A$ vrijedi $UP(r) \in A$ i $UP(r).v = r.v$. Varijabla $r.v$ je `volatile` ako i samo ako je r čitanje koje je `volatile` a varijabla $w.v$ je `volatile` ako i samo ako je w pisanje koje je `volatile`.

Izvođenje

- Uređaj `dogodilo-se-prije` je **parcijalni uređaj**. **Zadan je tranzitivnim zatvorenjem bridova relacije** `sinkroniziran-s` i **uređajem programa**. Mora biti valjani parcijalni uređaj: **refleksivan, tranzitivan i asimetričan**.
- Izvršavanje je **konzistentno u pogledu izvođenja dretve u izolaciji**. Za svaku dretvu d, **akcije** koje izvodi d (elementi skupa A) su **identični** kao što bi bili **generirani od strane te dretve po uređaju programa u kada bi se izvodila u izolaciji**. Svako pisanje **zapisuje** vrijednost $ZV(w)$ uz uvjet da svako čitanje r **uočava** vrijednost $ZV(UP(r))$.
- Izvođenje je `dogodilo-se-prije` **konzistentno**.
- Izvođenje je **konzistentno s obzirom na sinkronizacijski uređaj**. Za sva čitanja $r \in A$ koja su `volatile`, nije slučaj da je $su(r, UP(r))$ ili da postoji pisanje $w \in A$ takvo da $w.v = r.v$ i $su(UP(r), w)$ i $su(w, r)$.

Kod dobro utemeljenih izvršavanja su elementi relacije `sinkroniziran-s` i `dogodilo-se-prije` **jedinstveno određeni drugim komponentama izvođenja**.

Izvođenje i zahtjevi kauzalnosti

$f|_d$ predstavlja restrikciju funkcije f na d, analogno $p|_d$ označava restrikciju parcijalnog uređaja p elemenata iz d.

Dobro-utemeljeno izvođenje $E = (P, A, pu, su, UP, ZV, ss, dp)$ se validira tako da **izvršavamo akcije** iz A. Ukoliko sve akcije iz A **uspješno izvršimo**, tada **izvršavanje zadovoljava uvjete kauzalnosti** memorijskog modela programskom jeziku *Java*.

Krećemo od praznog skupa C_0 te izvodimo niz koraka u kojima akcije iz skupa akcija A dodajemo u skup izvršenih akcija C_i da bi dobili skup izvršenih akcija C_{i+1}. Smislenost postupka dokazujemo **traženjem izvršavanja** E koje sadrži C_i i **zadovoljava određene uvjete**.

Izvršavanje E zadovoljava uvjete kauzalnosti programskog jezika *Java* ako i samo ako postoji:
- **Skupovi akcija** $C_0,\ C_1,\ \ldots$ takvi da: a) $C_0 = \emptyset$, b) $C_i \subset C_{i+1}$, c) $A = \bigcup_i C_i$.
Ukoliko je A konačan, tada je niz $C_0,\ C_1,\ \ldots$ konačan i završava skupom $C_n = A$.

Izvođenje i zahtjevi kauzalnosti

Ukoliko je A beskonačan, tada $C_0, C_1, \ldots$ može biti beskonačan i mora vrijediti $\bigcup_i C_i = A$.
- **Dobro-utemeljena izvođenja** $E_1, \ldots, ,$ gdje
$E_i = (P, A_i, pu_i, su_i, UP_i, ZV_i, ss_i, dp_i)$.
Uz dane skupove akcija $C_0, \ldots$ i izvođenja $E_1, \ldots,$ svaka akcija u C_i **mora biti jedna akcija** u E_i. Sve akcije u C_i **moraju dijeliti isti relativni** `dogodilo-se-prije` **uređaj i sinkronizacijski uređaj** u E_i i E.
d) $C_i \subset A_i$, e) $dp_i|_{C_i} = dp|_{C_i}$, f) $su_i|_{C_i} = su|_{C_i}$.
Vrijednosti zapisane od strane pisanja u C_i **moraju biti identične** u E_i i E. Samo čitanja u C_{i-1} **trebaju vidjeti ista pisanja** iz C_{i-1} u E_i i E.
g) $ZV_i|_{C_i} = ZV|_{C_i}$ h) $UP_i|C_{i-1} = UP|C_{i-1}$
Čitanja u E_i koja nisu u C_{i-1} **moraju vidjeti pisanja koja su u relaciji** `dogodilo-se-prije` s njima. Svako čitanje $r \in C_i \setminus C_{i-1}$ **mora vidjeti pisanja** iz C_{i-1} i u E_i i u E, ali može vidjeti **drugačije pisanje** u E_i od onog u E. i) za svako čitanje r iz $A_i \setminus C_{i-1}$ vrijedi $dp_i(UP_i(r), r)$, j) za svako čitanje r iz $C_i \setminus C_{i-1}$ vrijedi $UP_i(r) \in C_{i-1}$ i $UP(r) \in C_{i-1}$.

Izvođenje i zahtjevi kauzalnosti

Uz zadani skup dovoljnih bridova relacije `sinkroniziran-s` za E_i, ukoliko postoji **otpusti-dohvati** (`release-acquire`) par koji je u relaciji `dogodilo-se-prije` s akcijom koju izvodimo, tada **taj par mora biti prisutan** u svim E_j, gdje $j \geq i$. k) Neka su ssw_i oni ss_i bridovi koji su u tranzitivnoj redukciji dp_i ali ne u pu. ssw_i se zove **dovoljan skup bridova** relacije `sinkroniziran-s` za E_i. Ako $ssw_i(x, y)$ i $dp(y, z)$ i $z \in C_i$, tada $ss_j(x, y)$ za svaki $j \geq i$. Ukoliko se izvodi akcija y, **sve eksterne akcije** koje su u relaciji `dogodilo-se-prije` y su se **isto izvodile**. l) Ako $y \in C_i$, x je eksterna akcija i $dp(x, y)$, tada $x \in C_i$.

Uočljivo ponašanje i izvođenja koja ne završavaju

Za programe koji uvijek završe izvođenje u ograničenom, konačnom vremenu, njihovo ponašanje možemo **neformalno razumjeti** promatrajući njihova **dopuštena izvršavanja**. Stvari nisu toliko jednostavne kod programa koji mogu imati neograničeno vrijeme izvršavanja.

Uočljivo ponašanje programa se **definira preko konačnog skupa eksternih akcija koje program može izvršiti**. Program koji zauvijek ispisuje *Pozdrav* se opisuje tako da za svaki ne negativni cijeli broj i ispisuje poruku *Pozdrav* i puta.

Završetak izvođenja se ne **modelira eksplicitno**, međutim program se može proširiti **dodatnom akcijom** `krajIzvođenja` koja se izvrši **nakon završetka izvođenja svih dretvi**. Definirana je posebna akcija **blokiranja**. Ukoliko je ponašanje programa opisano **skupom eksternih akcija koje uključuju akciju blokiranja**, onda opisuje ponašanje u kojem se, **nakon opažanja eksternih akcija**, **program može izvršavati neograničeno dugo bez izvođenja dodatnih eksternih akcija ili završavanja**.

Uočljivo ponašanje i izvođenja koja ne završavaju

Program se može zablokirati ukoliko su **sve dretve blokirane** ili ako **program može izvršiti neograničeni broj akcija bez izvođena eksternih akcija**.

Dretva **može biti blokirana** kada **pokuša dohvatiti lokot** ili **izvršiti eksternu akciju** (npr. čitanje) koje ovisi o eksternim podacima. Dretva **može biti blokirana i zauvijek** (bez završetka izvođenja). U takvim slučajevima, skup akcija generiranih od strane blokirane dretve se mora sastojati od **svih akcija do i uključujući akciju koja je uzrokovala blokiranje**. Taj skup **ne smije sadržavati niti jednu akciju** koja bi bila generirana od strane te dretve **nakon akcije koja je uzrokovala blokiranje**.

Ukoliko je O skup uočljivih akcija za izvršavanje E, tada $O \subseteq A$ i O **mora sadržavati samo konačan broj akcija** (čak i ako A sadrži beskonačan broj akcija). Ako je akcija $y \in O$ i $dp(x, y)$ ili $su(x, y)$ tada $x \in O$. Skup uočljivih akcija nije ograničen na eksterne akcije već samo eksterne akcije koje su u skupu uočljivih akcija čine uočljive eksterne akcije.

Uočljivo ponašanje i izvođenja koja ne završavaju

Ponašanje B je **dopustivo ponašanje** programa P ako i samo ako je B konačan skup eksternih akcija i vrijedi jedno od navedenog:
- Postoji izvršavanje E od P i skup O uočljivih akcija od E i B je skup eksternih akcija iz O (ukoliko neka dretva iz E završi u blokiranom stanju i O sadrži sve akcije iz E, tada B može sadržavati akciju blokiranja).
- Postoji skup O akcija takvih da B sadrži akciju blokiranja i sve eksterne akcije u O i $\forall k \geq |O|$, postoji izvođenje E programa P sa skupom akcija A i postoji skup akcija O' takvih da:

- $O,\ O' \subseteq A$ su skupovi uočljivih akcija
- $O \subseteq O' \subseteq A$
- $|O'| \geq k$
- $O' \setminus O$ ne sadrži eksterne akcije

B ne opisuje poredak u kojem su uočene eksterne akcije iz B, međutim (unutarnja) pravila koja uređuju način generiranja eksternih akcija mogu uvesti ograničenja na poredak.

Semantika `final` elemenata

Elementi članovi koji su deklarirani kao `final` se **inicijaliziraju jednom** ali im se **vrijednost ne mijenja u normalnim okolnostima**. Zbog toga se njihova semantika razlikuje od semantike normalnih članova (prevodioc može čitanja `final` elemenata **pomicati kroz sinkronizacijske barijere** i u **pozive proizvoljnih ili nepoznatih metoda**). Prevodioc može spremiti vrijednost `final` varijable u **registar** i **ne učitavati njenu vrijednost ponovo iz memorije** čak niti u slučajevima kada bi se to radilo kod normalnih varijabli.

`final` elementi također dopuštaju programeru **implementiranje nepromijenjivih, sigurnih objekata, s obzirom na pristup iz drugih dretava, bez korištenja sinkronizacijskih mehanizama**. Takav objekt je **nepromijenjiv za sve dretve**, čak i ako se reference proslijeđuju drugim dretvama u stanju utrkivanja za pristup resursima. Time dobijemo garancije protiv zloupotrebe nepromijenjivih klasa od strane netočnog ili malicioznog koda.

Semantika `final` elemenata

Objekt je potpuno inicijaliziran **kada se prestanu izvoditi njegovi konstruktori**. Dretva koja može vidjeti referencu na objekt nakon što je taj objekt potpuno inicijaliziran sigurno vidi korektno inicijalizirane vrijednosti elemenata članova tog objekta koji su `final`.

Način korištenja `final` elemenata za sinkronizaciju:
- Inicijaliziramo vrijednosti `final` polja u konstruktoru objekta.
- Ne zapisujemo referencu na objekt koji konstruiramo na mjesto gdje ju je moguće dohvatiti od strane druge dretve (dok se ne izvrši konstruktor objekta).
- Ukoliko su slijeđena prethodna dva koraka, sve dretve vide korektno konstruiranu verziju `final` elemenata tog objekta. Vidjet će i proizvoljni objekt ili polje referencirano od strane tih `final` polja koja su ažurna kao i `final` elementi.

Semantika final elemenata

```
1  class FinalElement {
2      final int x;
3      int y;
4      static FinalElement f;
5
6      public FinalElement() { x = 3;  y = 4; }
7     //pretpostavimo da pozovemo u dretvi 1
8      static void stvori() { f = new FinalElement(); }
9  //pretpostavimo da pozovemo u dretvi 2
10     static void koristi() {
11         if (f != null) {
12             int i = f.x;  // garancija da je vrijednost 3
13             int j = f.y;  // 0 ili 4 (nema garancije)
14         }
15     }
16 }
```

Korištenje final elemenata.

Semantika `final` elemenata

Neka je o objekt i c konstruktor za o u kojem se zapisuje `final` element f. **Akcija zamrzavanja** `final` elementa od o se događa kada c **prestane sa izvršavanjem ili normalno ili neočekivano**. Ukoliko jedan konstruktor pozove drugi konstruktor i pozvani konstruktor zapiše vrijednost `final` elementa, tada se akcija zamrzavanja događa na kraju pozvanog konstruktora.

Kod svakog izvršavanja na čitanja utječu i dodatna dva parcijalna uređaja, **lanac dereferenciranja** `dereferenciranje()` i **memorijski lanac** `ml()` koji se smatraju dijelom izvođenja. Za njih mora vrijediti:
- **Lanac dereferenciranja:** za akciju čitanja ili pisanja a, elementa člana objekta o, od strane dretve d, koja nije inicijalizirala o, mora postojati čitanje r od strane dretve d koje vidi adresu od o tako da za r vrijedi `dereferenciranje`(r, a).

Semantika `final` elemenata

- **Memorijski lanac:**
  - Ukoliko je r čitanje koje vidi pisanje w, tada mora biti $\texttt{ml}(w, r)$.
  - Ukoliko su r i a akcije takve da vrijedi $\texttt{dereferenciranje}(r, a)$ tada mora vrijediti $\texttt{ml}(r, a)$.
  - Ukoliko je w pisanje adrese objekta o od strane dretve d koji nije inicijalizirao o, tada mora postojati čitanje r od strane dretve d koja vidi adresu od o tako da $\texttt{ml}(r, w)$.

Za dano pisanje w, akciju zamrzavanja f, akciju a koja nije čitanje elementa člana koji je `final`, čitanje r_1 `final` elementa koji je zamrznut od f i čitanje r_2 takvo da $dp(w, f)$, $dp(f, a)$, $ml(a, r_1)$ i $\texttt{dereferenciranje}(r_1, r_2)$ tada pri određivanju koje vrijednosti vidi r_2 promatramo $dp(w, r_2)$. Uređaj `dereferenciranje()` je **refleksivan**. Jedina pisanja koja smiju prethoditi čitanju `final` elemenata su ona **izvedena iz semantike** `final` elemenata.

Ostala svojstva `final` elemenata

Čitanje `final` elementa objekta unutar dretve koja konstruira objekt je **uređeno relacijom** `dogodilo-se-prije` s obzirom na inicijalizaciju tog elementa unutar konstruktora. Ukoliko se čitanje dogodi nakon postavljanja vrijednosti u konstruktoru, **vidi konačno pridijeljenu vrijednost**, inače **vidi standardno pridijeljenu vrijednost**.

U nekim slučajevima (kao npr. **deserijalizacija**) je **potrebno promijeniti vrijednost** `final` elementa objekta **nakon stvaranja**. `final` elementi se mogu mijenjati kroz npr. refleksiju. Jedina smislena promjena vrijednosti je ona u kojoj je objekt prvo kreiran (`final` element inicijaliziran) a **zatim su ažurirane vrijednosti** `final` elementa. Objekt **ne smije biti vidljiv drugim dretvama** i `final` elementi se **ne smiju čitati dok nije završeno pridijeljivanje vrijednosti**. Akcija zamrzavanja `final` elementa se događa **na kraju konstruktora** u kojem se postavlja vrijednost `final` elementa i **neposredno nakon svake promijene vrijednosti korištenjem refleksije ili drugih mehanizama**.

Ostala svojstva `final` elemenata

I kod takvog pristupa dolazi do problema. Ukoliko je `final` element **inicijaliziran na konstantan izraz pri deklaraciji**, tada **promjene toga elementa ne moraju nužno biti vidljive** (prevodioc može sva **korištenja** `final` elementa **zamijeniti odgovarajućim konstantnim izrazom kojim je inicijaliziran**). Drugi problem je da **specifikacija dozvoljava dosta neograničenu optimizaciju** `final` elemenata (prevodioc može **preurediti poredak čitanja** `final` elementa i **naredbi koje mijenjaju vrijednost tog elementa izvan konstruktora**).

Java omogućava mijenjanje određenih posebnih `final static` polja - `System.in`, `System.out` i `System.err` korištenjem naredbi `System.setIn`, `System.setOut` i `System.setErr` stoga su ta polja **zaštičena od pisanja**. Korisnički kod ne može mijenjati vrijednost tim elementima osim ako nije u klasi `java.lang.System`.

Ostala svojstva `final` elemenata

```
1  class Prva {
2      final int x;
3      Prva() { x = 1; }
4
5      int f() {  return d(this,this); }
6  //prevodioc moze bez ogranicenja preurediti naredbe citanja
       a1.x, a2.x i poziv funkcije g, stoga su kod new A().f()
       moguce povratne vrijednosti funkcije d: -1, 0 i 1
7      int d(Prva a1, Prva a2) {
8          int i = a1.x;
9          g(a1);
10         int j = a2.x;
11         return j - i;
12     }
13
14     static void g(Prva a) {
15      // koristimo refleksiju za dodijeljivanje a.x = 2
16     }
17 }
```

Primjer moguće reorganizacije koda od strane prevodioca.

Razlamanje riječi

Svaki element član klase ili element polja se smatraju **posebnim elementom**. **Ažuriranje vrijednosti jednog elementa ne smije utjecati na ažuriranje ili čitanje bilo kojeg drugog elementa**. Dretve koje ažuriraju susjedne elemente polja *byte*-ova ne smiju utjecati jedna na drugu i **takav pristup mora osiguravati sekvencijalnu konzistentnost bez sinkronizacije**.

Neki procesori ne omogućavaju pisanje jednog *byte*-a. Kod takvih procesora bi bilo **nepravilno** pročitati cijelu riječ, upisati vrijednost *byte*-a i zatim riječ spremiti natrag u memoriju. Problem neovisnog pristupa različitim elementima se zove **razlamanje riječi**.

```
1 public class Razlamanje extends Thread {
2     static final int DULJINA = 8;
3     static final int ITERACIJA  = 1000000;
4     static byte[] brojaci    = new byte[DULJINA];
5     static Thread[] dretve = new Thread[DULJINA];
6     final int id;
```

Kod koji detektira razlamanje riječi.

Razlamanje riječi

```
1     Razlamanje(int i) { id = i; }
2
3     public void run() {
4         byte v = 0;
5         for (int i = 0; i < ITERACIJA; i++) {
6             byte v2 = brojaci[id];
7             if (v != v2) {
8                 System.err.println("Pronadeno razlamanje
    rijeci: brojaci["+ id +"] = "+ v2 +", treba biti "+ v);
9                 return;
10            }
11            v++;
12            brojaci[id] = v; } }
13
14    public static void main(String[] args) {
15        for (int i = 0; i < DULJINA; ++i)
16            (dretve[i] = new Razlamanje(i)).start();
17    }
18 }
```

Kod koji detektira razlamanje riječi.

Korištenje wait i notify

```
1  public class Skladiste {
2      private String poruka;
3      private boolean prazno = true;
4
5   public synchronized String uzmi() {
6     while (prazno) {
7      try { wait(); } catch (InterruptedException e) {} }
8          prazno = true; notifyAll();
9          return poruka; }
10
11  public synchronized void stavi(String porukaN) {
12    while (!prazno) {
13     try { wait(); } catch (InterruptedException e) {} }
14         prazno = false; this.poruka = porukaN;
15         notifyAll();
16     }
17 }
```

Implementacija problema proizvođača i potrošača.

Korištenje wait i notify

```
1  public class Proizvodac implements Runnable{
2    private Skladiste skladiste;
3
4    public Proizvodac(Skladiste skladiste) {
5          this.skladiste = skladiste; }
6
7    @Override
8    public void run() {
9      String poruke[] = { "Poruka1", "Poruka2", "Poruka3", "
       Poruka4"; };
10
11         Random random = new Random();
12         for (int i = 0; i < poruke.length; i++) {
13             skladiste.stavi(poruke[i]);
14             try {
15                 Thread.sleep(random.nextInt(5000));
16             } catch (InterruptedException e) {}
17         }
18         skladiste.stavi("GOTOVO"); } }
```

Implementacija problema proizvođača i potrošača.

Korištenje wait i notify

```
1  public class Potrosac implements Runnable{
2      private Skladiste skladiste;
3
4      public Potrosac(Skladiste skladiste) {
5          this.skladiste = skladiste;
6      }
7
8      @Override public void run() {
9          Random random = new Random();
10         for (String poruka = skladiste.uzmi();
11              ! poruka.equals("GOTOVO");
12              poruka = skladiste.uzmi()) {
13             System.out.format("Primljena poruka: %s%n",
       poruka);
14             try {
15                 Thread.sleep(random.nextInt(5000));
16             } catch (InterruptedException e) {}
17         } } }
```

Implementacija problema proizvođača i potrošača.

Korištenje nepromijenjivih objekata u višedretvenom programiranju

Za kreiranje ispravnih, nepromijenjivih objekata koji osiguravaju da ne može doći do nepravilnog korištenja u višedretvenim programima treba:

1. Ispustiti sve metode koje postavljaju vrijednosti elementima članovima klase ili objektima referiranim unutar klase (tzv. *setter* metode).
2. Svi elementi članovi bi trebali biti privatni i konstantni.
3. Nebi trebalo dopustiti podklasama da nadjačavaju metode (npr. stvaranjem klase koja je `final` ili kreiranjem privatnog konstruktora i definicijom odgovarajućih metoda za stvaranje objekata).
4. Ukoliko elementi članovi instance sadrže reference na promijenjive objekte, nebi se smjelo dopustiti mijenjanje tih objekata.
   - Nebi trebalo implementirati metode koje mijenjaju promijenjive objekte.
   - Nebi trebalo dijeliti reference promijenjivih objekata (npr. spremati u eksterne reference koje se koriste u konstruktorima). Treba kreirati kopije po potrebi i spremati reference u kopije. Po potrebi treba vraćati kopije internih promijenjivih objekata da bi se izbjeglo vraćanje originala.

Korištenje nepromijenjivih objekata u višedretvenom programiranju

```java
public final class NepromijenjivaTocka {
    final int x, y;

    public NepromijenjivaTocka(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public NepromijenjivaTocka translatirajOkoX(){
        return new NepromijenjivaTocka(x,-y);
    }

    public NepromijenjivaTocka translatirajOkoY(){
        return new NepromijenjivaTocka(-x,y);
    }
}
```

Osiguravanje korektnosti korištenjem nepromijenjivih objekata.