. . . a to znači

- → započeti sa **apstraktnim promišljanjem** o programima: program gledati kroz dijagrame (`UML`, `ER` i sl.)
- → dati veći naglasak cjelokupnom ciklusu razvoja programske potpore (*pp*): od ideje do održavanja s naglaskom na postupke testiranja
- → od sada je važnije **ŠTO** radi *pp* a ne **KAKO** (neovisnost o implementacijskom jeziku)

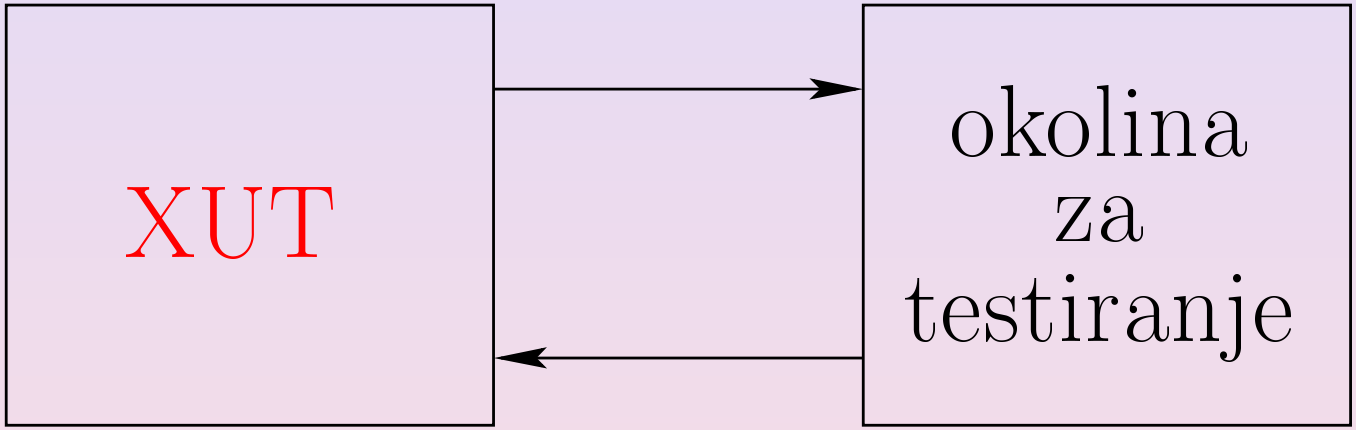
ŠTO program radi opisuju zajedno:

- ⇒ *usmjereni grafovi* $G(V, E)$ za prikaz raznih "`dijagrama`" ("struktura" *pp*)
- ⇒ *logika* za opis funkcionalnosti koju realiziramo ("ponašanje" *pp*)

Raniji primjer je možda hipotetičke naravi ali . . .
. . . da li ste ikada nakon što ste završili svoj prvi primjer u jeziku *Java* pomislili:

- → moj program radi i ne treba više raditi na tome
- → primjer je toliko jednostavan da ga ne treba testirati. . .
- → ne trebam ništa testirati jer sam sve obavio sa "*debuggerom*"
- → ne trebam stavljati komentare, sve se vidi iz programskog koda
- → program je trivijalan i ne treba ga dokumentirati . . .

Okružje za testiranje



Različite razine testiranja

1) jedinično testiranje: *JUnit* (predavano ranije)
2) testiranje većih cjelina: *Expect*
3) prema automatiziranom testiranju: *JavaPathfinder*

→ testiramo klasu po klasu zasebno

→ testiramo *cjeline* unutar sustava, postepeno obuhvaćamo sve više *cjelina* ...

→ potpuno automatsko testiranje ne postoji, ali se može donekle automatizirati

Podjele prema načinu testiranja

- statičko (strukturalno) i funkcionalno
- bijela i crna kutija
- prema obuhvaćenoj cjelini testiranja: komponenta, blok jedinica – regresijsko
- prema automatiziranosti: poluautomatsko ili ručno
- iz kôda ili prema (zadanoj) specifikaciji (conformance)
- mutant, simbolično (concolic), kombinatorno
- kombinacije spomenutoga - prema zadanoj odabranoj strategiji
- testiranje kao dio razvoja: TDD

JUnit, XUT i *programska potpora*

JUnit–Java Unit je program (programsko okružje) za testiranje.

Što znači Unit ?

Unit predstavlja odabranu jedinicu–komponentu koju testiramo.
postoji xUnit podrška za većinu (objektnih) programskih jezika

1. okružje (1) dobivamo dodavanjem klasa u *XUT*
2. testiranje započinje definiranjem test–slučajeva
3. test–slučajevi se prevode (*javac + JUnit.jar*)
4. dobivene klase izvodimo sa *JUnit Test Runner*

Primjer: postotak izgubljenih poruka


```
import java.lang.Math.* ;
public class Ex01LostMessages {

   private int nrec ; // no of received messages
   private int ntot ; // total messages

   // constructor :
   public Ex01LostMessages(int nr, int nt) {
     nrec = nr;
     ntot = nt ;
   }

  /**   * Calculate percentage   *   */
   public double percentLost() {
//       nrec = Math.abs(nrec);
//       ntot = Math.abs(ntot);
      double plost = ((double) nrec / (double) ntot) * 100 ;
      return plost ;
   }

  /**   * Call this to cancel/nulify this .   */
   public void cancel() { ntot = 0 ; }

}
```

Primjer JUnit testovi


```
import org.junit.* ;
import static org.junit.Assert.* ;

public class Ex01LostMessagesTest {

  @Test
  public void test_lessThanOne() {
    System.out.println("TEST-01: (percentage is less or equal 1)") ;
    Ex01LostMessages S01 = new Ex01LostMessages(10,20) ;
    assertTrue(S01.percentLost() <= 100.00) ;
    Ex01LostMessages S02 = new Ex01LostMessages(30,200) ;
    assertTrue(S02.percentLost() <= 100.00) ;
  }

  @Test
  public void test_nonNeg() {
    System.out.println("TEST-02: (non negative and le 100)") ;
    Ex01LostMessages S = new Ex01LostMessages(2,-3) ;
    assertTrue(S.percentLost() >= 0) ;
  }
}
```

. . . od ranije znamo:

Očekivani rezultat mora biti jednak dobivenom rezultatu:

1. *(expectedResult == obtainedResult)*
2. *expectedResult.equals(obtainedResult)*
3. `assertTrue(S02.percentLost() <= 100.00)`

Da li možete dodati slične instrukcije i bez *JUnit okružja* ?

Što je "test suite" (TESTSuite) kod Junit testiranja

"test suite" ili `test` ili `test sekvenca` (`TESTSuite`)

`test sekvenca` ili test – (eng. "test–suite") je:

- $\Rightarrow$ skup test slučajeva i sadržava jedan ili više pojedinačnih `test-slučajeva`
- $\Rightarrow$ testiraju funkcionalnost prema modelu $\mathcal{M}$ ili specifikaciji: mora se znati **što** implementira pp
- $\Rightarrow$ jedan ili viš `testova` ili `test-sekvenci` su sastavni dio dokumentacije koja se predaje testerima ili ili služi za održavanje pp

$\Longrightarrow$ pogledati `Ex01LostMessage` primjer

Model $\mathcal{M}$ za pp i testovi

Odnos programske potpore pp i testova može se prikazati slijedećom formulom:

$$\mathcal{M} \models \varphi$$

gdje su:

(i) model $\mathcal{M}$ je XUT
Model $\mathcal{M}$ je isto što i programski kod. Kasnije se uvode formalni modeli prikaza programske potpore.

(ii) φ je skup logičkih varijabli (φ_i) koje opisuju istinitost testova: ako je test i OK tada je φ_i istinit

(iii) JUnit naredbe `assert`$\star$ nazivamo i `invarijantama` programskog sustava

Kažemo: model $\mathcal{M}$ zadovoljava formulu φ_i.

Još JUnit assert⋆ naredbi prije primjera . . .

(1) `assertEquals`

(2) `assertArrayEquals`

(3) `assertTrue` i `assertFalse`

(4) `assertNull` i `assertNotNull`

(5) `assertSame` i `assertNotSame`

(6) . . . potrebno je stalno pratiti nove inačice JUnit programa

Osim `@Test` postoje još i sledeće direktive za izvođenje testova:

`@BeforeClass`, `@Before`, `@After` `@AfterClass`

Za vježbu:

pogledati primjere `Ex2a` i `Ex2b`

Rješenje:

"Lažni" ili "Mock" objekti

1. "Mock" objekt - lažni, oponašajući, imitirajući . . . objekt
2. nadomjestimo svaku neželjenu popratnu "smetnju" za testiranje "mock objektom"

U našem primjeru . . .

sve što se odnosi na poziv serveru za dobivanje dnevnog tečaja nadomještamo lažnim ili "Mock" objektom

EasyMock i Dummy Mock

U prethodnom primjeru:

- potupak testiranje identičan je ranije rečenom: kreira se XUT i `XUTTest`
- nakon toga potrebno je prolagoditi `ExchangeRateProvider.java` koji postane `DummyProvider.java`
- za složenije slučajeve postoje alati koji olakšavaju pripremu "Mock" objekta

"Mock" programska pomagala

1. *EasyMock*: `http://easymock.org`
2. *jMock*
3. *Mockito*

Primjer: `Ex04: StockExchange`

Što je *testiranjem upravljano programiranje* ?

testiranjem upravljano programiranje
ili
eng. **TDD** – *Test Driven Development*

Ukratko:

1. napisati dovoljno instrukcija zajedno sa (xUnit) testovima da tijekom prvog izvođenja test padne
2. ispraviti što treba pa testirati dok svi test slučajevi ne prođu

Dobro:

nisu potrebna posebna, dodatna pomagala

"TDD" pristup razvoja pp

- ⟶ napisati test slučajeve
- ⟶ napisati jezgru (skeleton) koda take da se pp može pokrenuti
- ⟶ test padne (nema koda za rješenje)
- ⟶ dopuniti kod, testirati
- ⟶ ponavljati dok svi testovi ne postanu uspješni

(1) jedinično testiranje: testiraj svaku metodu unutar jedne klase zasebno kroz niz ("razuman broj") testova

(2) testiranje klase: iduća jedinica složenosti kod OO programa nakon metode je klasa. Kod ne–objektnih programa testiramo komponentu, blok, modul ...

(3) integracijsko testiranje: svaka od klasa je u interakciji s nekim klasama: potrebno je par po par, skup po skup takvih interakcija testirati. Kod neobjektnih programa testiramo interakcije komponenti, blokova, modula, ...

(4) testiranje sustava: testiranje funkcionalnosti sustava, najčešće prema zahtjevima krajnjeg korisnika

(5) test prihvatljivosti krajnjeg korisnika naručioca koji mora odgovoriti na pitanje:
Da li je programska potpora prihvatljiva za krajnjeg korisnika ? Toj grupi pripada tzv. α i β testiranje.

Upravljivost

Upravljivost programske potpore

kolika je složenost određivanja razumnog broja ulaza kao vrijednosti varijabli, operacije ili ponašanje programa. Upravljivost programa koji isključivo primaju ulaze preko tastature je veća nego programa koji primaju ulaze direktno preko senzora, iz sklopovske potpore ili iz raspodijeljenih procesa, niti . . .

Modeli programske potpore

(1) grafovi

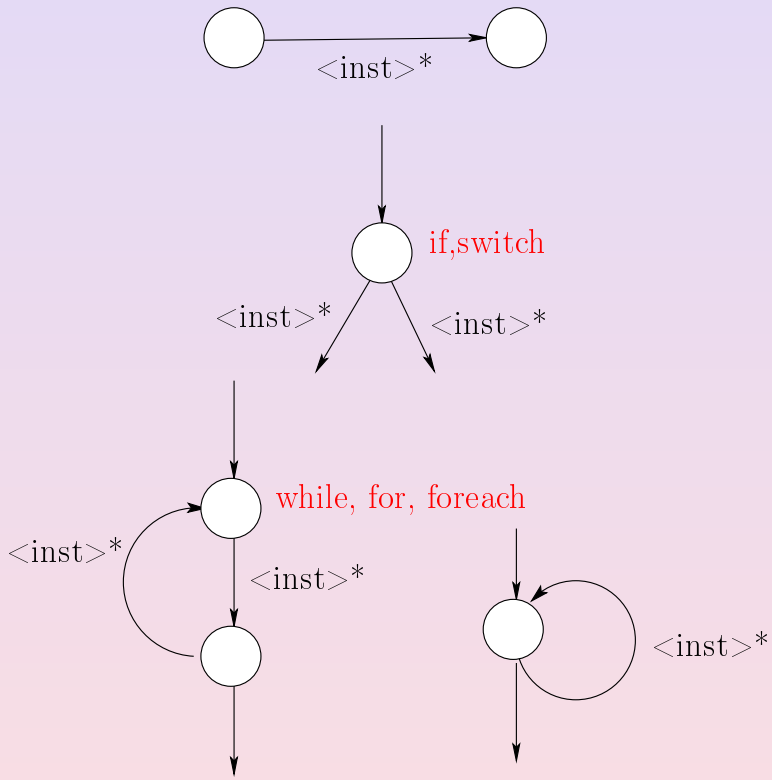
(2) logički izrazi npr.:
$(not \quad X \quad or \quad not \quad Y) \quad and \quad A \quad and \quad B$

(3) karakterizacija ulazne domene: granice varijabli npr.
$A : 0, 1, > 1$
$B : 600, 700, 800$

(4) sintaksne strukture:
$if\ (x > y)$
$z = x - y;$
$else\ z = 2 * x$

U daljnjem dijelu koristiti ćemo sintaksne strukture sa grafom

Kako nacrtati graf ?



- kružići ili čvorovi predstavljaju kontrolne točke (grananja) u programu i sadržavaju vrijednosti svih varijabli
- grane u grafu predstavljaju instrukcije, pozive metoda ili razmjenu poruka
- komunikacijske programe opisujemo sa 2 ili više grafova povezanih komunikacijskim kanalima

Reaktivni sustavi i distribuirana konkurentna pp.

Reaktivni sustav opisuje ponašanje pp.

- → pp. mora *odmah* reagirati na vanjski događaj (prijem poruke, istek vremenske kontrole . . .)
- → poduzete akcije (pozivi metoda, predaja poruka . . .) ovise o unutarnjem stanju u kome se pp. nalazi
- → pp. za reaktivne sustavi najčešče nalazimo kod upravljačkog sloja telekomunikacijske mreže

Primjer:

zahtjev za raskidanjem na početku i kraju transakcije/sesije. . .
Kako zaključiti koji su važeći ishodi ?

- pretvaranje programa u graf nije egzaktno: uvijek postoji više valjanih rješenja za pretvorbu
- važno je dobro poznavati XUT ili imati specifikaciju za XUT
- potpuno testiranje (potpuna prekrivenost grafa) traje previše vremena i ne garantira potpunu odsutnost neispravnosti
- u praksi nastojimo minimalnim brojem testova otkriti što više neispravnosti
- nedostatak: složenost grafova za analizu (potrebni programski alati)

Za one koji hoće više:

Kako bi automatizirali pretvorbu programa u graf ?

Prekrivanja u grafu

(⟶1) prekriti sve instrukcije ili izvesti svaku instrukciju barem jednom

(⟶2) prekriti sve puteve u grafu

(⟶3) prekriti sve čvorove u grafu

(⟶4) prekriti sve grane u programu

(⟶5) izvesti svaku metodu barem jednom

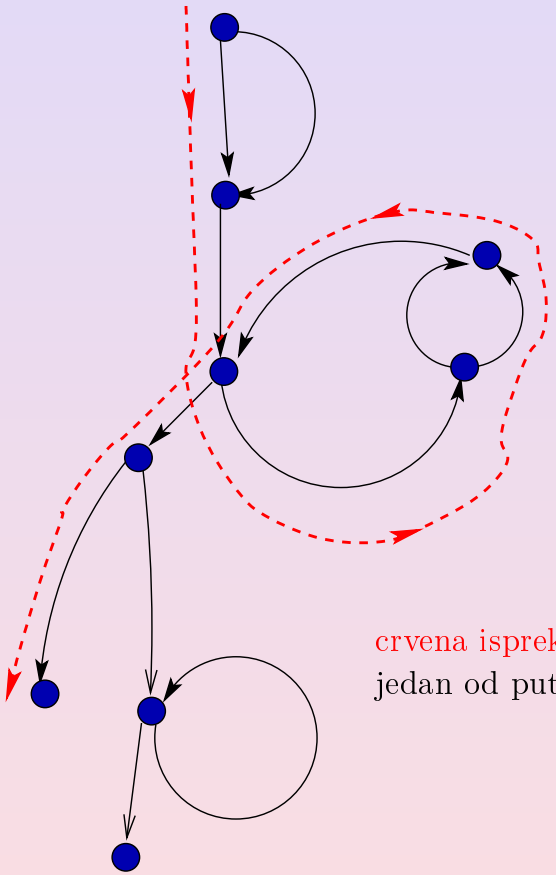
(⟶6) prekriti sve uvjete (naredbe *if, while, for* . . .)

(⟶7) kombinacije navedenog

Još o prekrivanju:

prekrivanje znači izvesti barem jednom . . .

Primjer



crvena isprekidana linija
jedan od puteva

- **složenost**: stotine/tisuće grana i čvorova, nastojimo prikazati samo ono najbitnije
- **početak**: čvor bez ulaznih grana
- **kraj**: čvor bez izlaznih grana

Za vježbu:
Precrtati graf i testirati primjenom svih spomenutih načina prekrivanja

Pomagala za testiranje

Pomagala za testiranje su:

- specijalizirani programski alati (poput uvedenog *JUnit*)
- razni pomoćni alati (spomenuti ćemo *Expect*)
- specijalizirani jezici za testiranje (**TTCN-3** **T**esting and **T**est **C**ontrol **N**otation ver.3)
- alati za provjeru modela (eng. *model checkers*) spomenuti će se *JavaPathfinder*

Expect

- **Expect** je programsko pomagalo za automatizaciju interaktivnih programa
- **Expect** je skriptni jezik koji "razgovara" sa programom
- ne podržava grafička sučelja (GUI) nego isključivo konzole (CLI)
- osnovna primjena: skratiti dugo i zamorno ponavljanje istih sekvenca komandi
- osim osnovne primjene može se koristiti i u testiranju
- u izvornom obliku **Expect** je razvijen kao **TclTk** aplikacija
- više na: `http://expect.nist.gov`

Korištenje *Expecta*

Na primjer automatizacije ftp sesije ilustrirati će se korištenje *Expecta*.
automatizacija ftp sesije je isto što i integracijsko (funkcionalno) testiranje *ftp sesije*. Svaki *Expect* program (ili skripta ili sesija) se sastoji od:

(1) inicijalizacije: (`set`)

(2) `send`: pošalji nalog

(3) `expect`: "očekuje" očekivanu vrijednost

(4) ponavljaj (2) – (3) ili

(5) **END**

Ukoliko *očekivani* rezultat izostane test nije uspio.

O sinkronizaciji

Važno:

- ↪ U stvarnosti nikada nisu unaprijed poznata vremena poziva i izvođenja metoda
- ↪ sinkronizacija XUT i okoline ovisi o okolini (*STDIN i STDOUT* interprocesnoj komunikaciji) kao i o operativnom sustavu
- ↪ pogodno za testiranje zahtjeva (eng. *requirement testing or verification*)
- ↪ potrebno je dodavati u metodeXUT ispise prema `STDOUT` kao i analizirati poruke iz `STDIN`.

Gotovi testovi za TTCN-3

(1) SIP (Session Initiation Protocol)
(2) IP Multimedija Subsystem (IMS)
(3) Internet protocol ver. 6 (IPv6)
(4) WiMax MAC
(5) Digital Mobile Radio (DMR)
(6) Digital Public Mobile Radio (dPMR)
(7) Dynamic Host Configuration Protocol (DHCPv6)
(8) 3GPP LTE TestSuite

Java program i specifikacija

Prije početka testiranja, postoji li specifikacija. model, formalni ili UML opis pprazvijene u Javi ?

> *"Osnovna ideja je (ponekad pomalo fantastična) imati script koji specificira test takav da osoba koja nakon toga testira može biti i virtualni robot koji će učinite sve automatski"*

Očekivanje . . .

Može se očekivati daljnja automatizacija testiranja. Programski alati za automatiziranje pisanja test skripti iz specifikacije ce pri tome imati sve veću ulogu u testiranju.

INFORMACIJA LOGIKA JEZICI

L o g i k a

FER-ZZT-crs-bb
2014/2015

O algebri i logici

Što je algebra ?

(osim algebra rabi se i termin *calculus*)

Neka je zadan skup **A:**

$$\mathbf{A = \{a_1, a_2, \ldots, a_i, \ldots . a_n\}}$$

Skup operacija Ω nad skupom A:

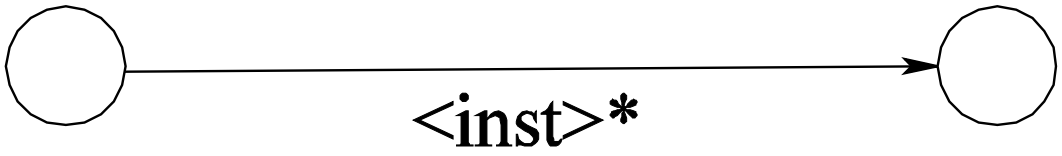
$$\Omega = \{\omega_1, \omega_2, \ldots, \omega_i, \ldots . \omega_n\}$$

Skup aksioma, zakona sa A i Ω

(komutacija, asocijacija,distribucija . . .)

A i Ω zajedno čine *algebru*

Primjer 1: apstrakcija (graf) i matematički aparat (Hoare logika)



graf/automat

{P} C {Q}

logička formula

Floyd-Hoare logika (uređena trojka) i pravila:

$$\{P\}\ C\ \{Q\}$$

- P,Q su tvrdnje (eng. *assert*)
- C je komanda (instrukcija)
- *ako je tvrdnja P točna prije izvršenja C onda je tvrdnja Q točna ili je program u blokadi*
- *tvrdnje nalazimo kao instrukcije u alatima/jezicima*

Primjer 2: logika u C, C++, Javi, . . .

- logički izrazi *u naredbama* imperativnih programskih jezika:
  - if*(uvjet)*
  - switch*(uvjet)*
  - while*(uvjet)*
  - for*(uvjet)*
  - operatori nad bitovima (&,|,^)
  - logički operatori (&&,||)

uvjeti definiraju tijek izvođenja instrukcija (eng. *control-flow*)

Vrste logike

- propozicijska, predikatna
- logika drugog reda
- logike višeg reda
- modalna, temporalna (LTL i CTL)
- ostale: {descriptivna, *fuzzy, kontinuirana}* logika . . .

razvoj novih vrsta logika traje i danas, npr. *interval , dinamička logika . . .*

Boolova algebra

počnimo s poznatim . . .

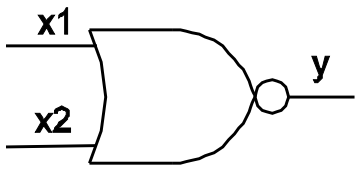
- Boolova algebra (digit. logika)

  $A = \{0,1\}$

  $\Omega = \{I, ILI, neg, komplement, ...\}$

- zakoni/aksiomi:
  - asocijativnost
  - komutativnost
  - apsorcija
  - distributivnost
  - komplement
  - . . .

Boolova logika – pruža temelj i motivaciju za proširenje na programe



shema-simbol: nule i jedinice kao naponske vrijednosti

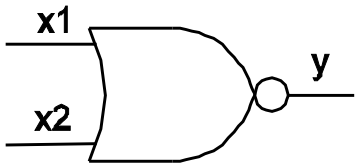
0	0	1
0	1	0
1	0	0
1	1	0

tablica: opisuje sve moguće kombinacije događaja na ulazi i izlazu

(za složenije slučaje ogromne tablice - kombinatorna eksplozija)

y:=x1 NILI x2

logička formula: kraće zapisana tablica



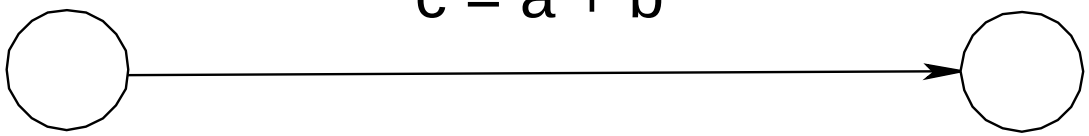
0	0	1
0	1	0
1	0	0
1	1	0

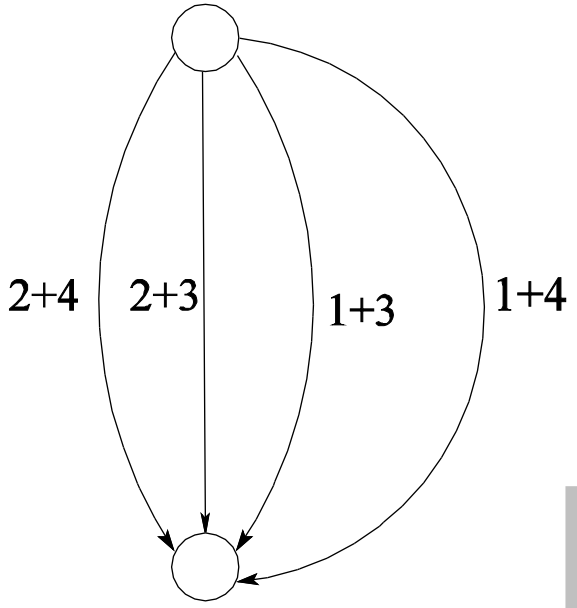
y:=x1 NILI x2

a={1,2} b={3,4}

c={4,5,6}

c = a + b





P≡{a:={1,2};b:={3,4}}

C≡ c = a + b

Q≡{c:={4,5,6}}

Automat (FSM) - definicija

- FSM – konačni diskretni automat
- konačni diskretni automat FSM=(S,s,A,T,L)
  - S – stanje
  - s – početno stanje
  - A – alfabet
  - L – labele (<instrukcije>*)
  - T: $S \times A \rightarrow S$

Striktnija definicija mora još uključiti:

- definiciju *ulaza* i *izlaza*

- definiciju *unutarnjih prijelaza*

Propozicijska logika

- propozicijska logika je algebarski sustav:

$$propL=(A,\Omega,Z,I)$$

gdje su:

- A – simboli ili propozicijske varijable
- Ω – logičke poveznice ili operatori
- Z – pravila zaključivanja *(Reductio ad absurdum, eliminacija konjukcije, dvostruka negativna eliminacija,* ***modus ponens****. . .)*
- I – aksiomi (inicijalne ili početne točke)

Propozicijska logika -primjer

- A = {p,q}
- Ω = { ¬, →} (negacija,implikacija)
- Pravila zaključivanja: **modus ponens**
  - *iz* p, (p→q) slijedi *q*
- |= ∅

u programskoj praksi često se koristi **modus ponens** kao mehanizam zaključivanja *(eng. inference)*

u umjetnoj inteligenciji **modus ponens** se naziva

“rezoniranje unaprijed” (engl. *forward reasoning*)

Predikat

- *p* i *q* iz prethodnog primjera su sudovi
- logiku sudova (često se naziva i algebra sudova) sačinjavaju formule povezane uobičajnim logičkim operatorima (ILI,I,...)
- pogledajmo još jednom:

  p ≡ Danas je srijeda

- ako umjesto srijede uvedemo varijablu $dan:

  p ≡ Danas je $dan

- dobili smo **predikat** *p* za *unifikaciju*

Temporalna logika

- Modalna logika uvodi *modalitete*: moguće, vjerojatno i nužno kao dodatne operatore uz standardne I, ILI, negacija ...
- Temporalna logika je poseban slučaj modalne logike koja modalne operatore interpretira u vremenu
- Temporalna logika uvodi *modalitete* kao *temporalne operatore: slijedeće, uvijek, sve dok, eventualno*

Temporalna logika omogućava verifikaciju i specifikaciju programske opreme

Zadatak- primjer:

Zadan je distribuirani sustav koji čiji je zadatak određivanje nul-točki polinoma (n=2, parabola) preko rješavanje kvadratne jednadžbe. Sustav ima dva podsustava: prvi podsustav definira ulazne podatke, a drugi preuzima te podatke i vraća prvom rješenja.
Potrebno je:
a) opisati i definirati sustav
b) uvesti potreban broj procesa i definirati veze među njima
c) opisati svaki od procesa preko automata (FSM)
d) definirati pripadne grafove i logičke formule
e) predložiti implementaciju kao OO prog.potp.

Rješenje zadatka

a) Sustav kako ga vidi vanjski korisnik:

Kojim UML dijagramom se također može opisati rješenje pod a)?

Rješenje zadatka

b) uvodimo **dva** procesa koja međusobno komuniciraju: za svaki podsustav po jedan

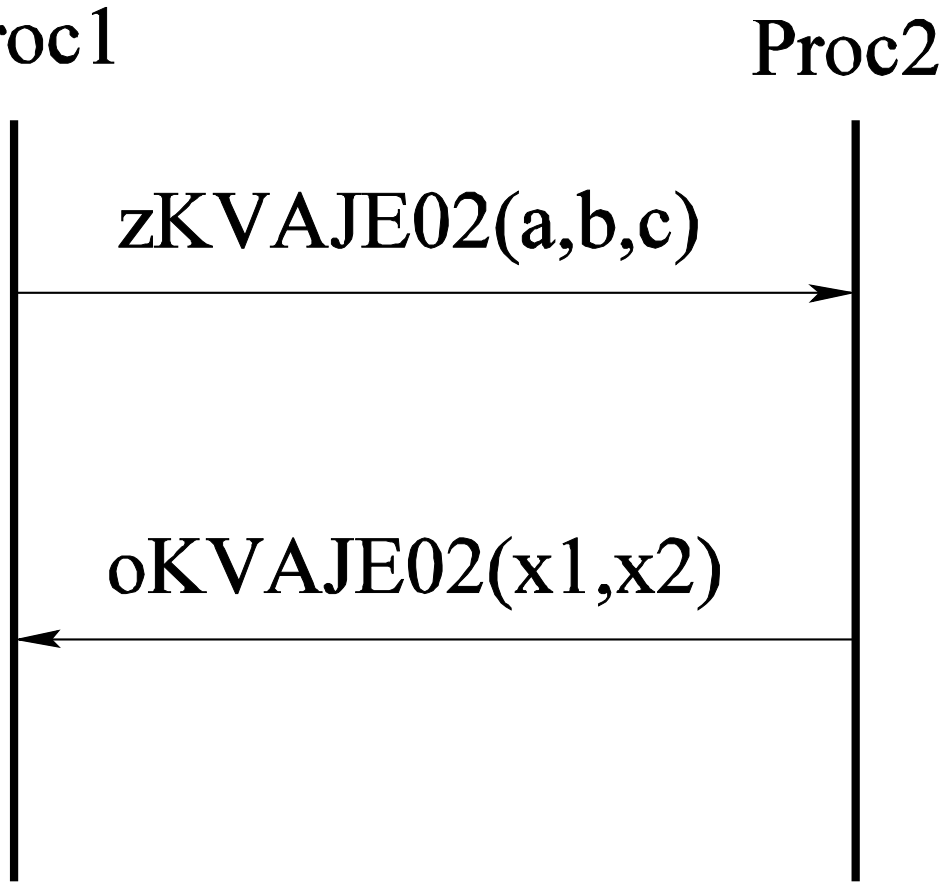
zKVAJE() zahtjev

oKVAJE() odgovor

zKVAJE:

Proc1 šalje

a Proc2 prima



Rješenje zadatka

Pitanja:
Kako su procesi alocirani (arhitektura):
Proc1, Proc2
- klijent – server,
- web pretražnik – davaoc usluga . . .
-distribuirana aplikacija (paralelni program . . .)

Možete navesti još primjera ?
Koji UML dijagram ima sličan zapis ?

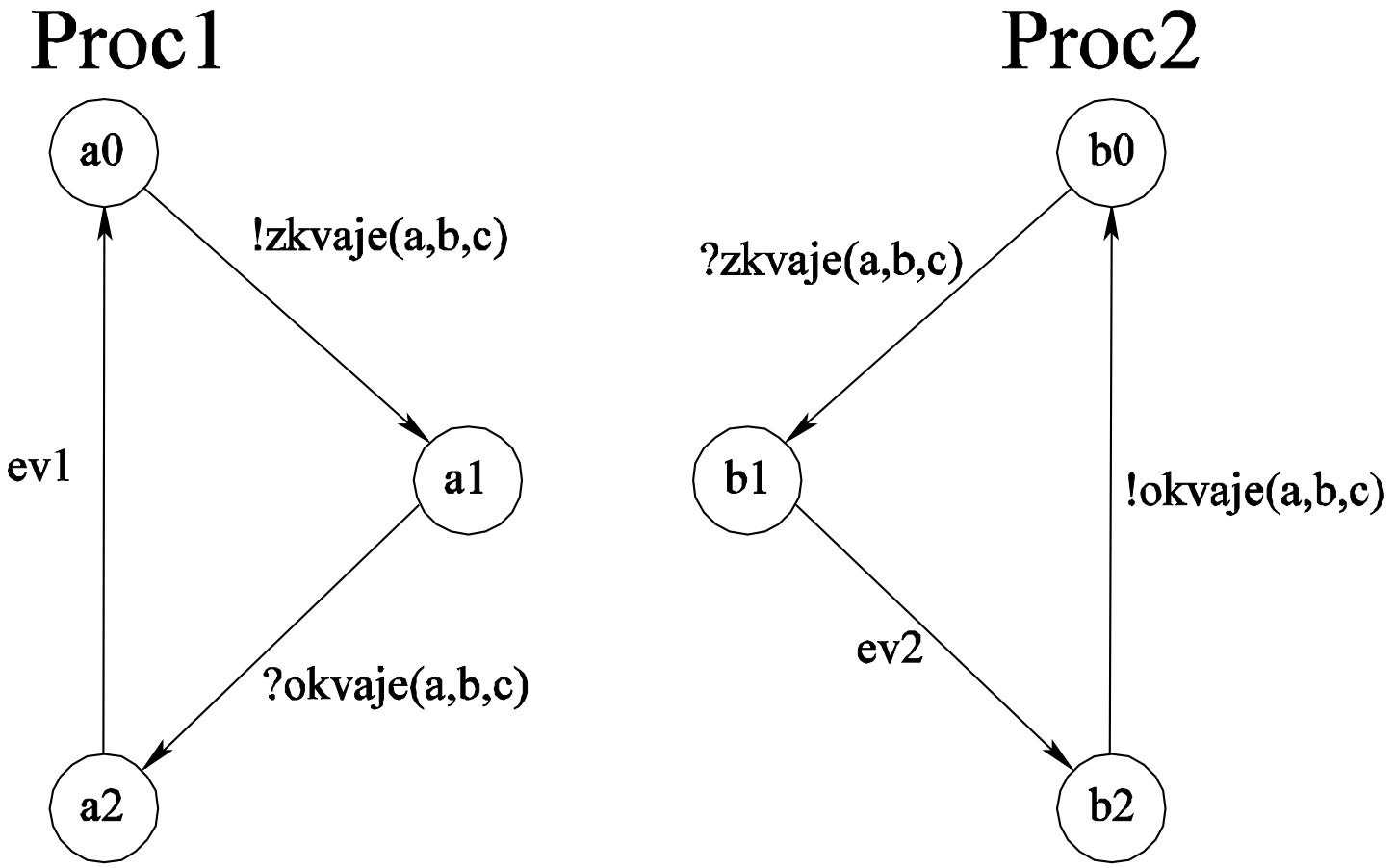
Niti jedna jedina instrukcija nije još napisana ali . . .

Rješenje zadatka

c) procesi prikazani kao grafovi pripadnih diskretnih konačnih automata koji međusobno komuniciraju:

! predaja

? prijem



Rješenje zadatka

Kvadratna jednadžba treba dati rješenja u matematičkom smislu. Praktična upotrebljivost diktira potrebu za odbacivanjem kompleksnih rešenja ili dvostrukih rješenja.

To određuje Proc1 u ev1 unutarnjem prijelazu.

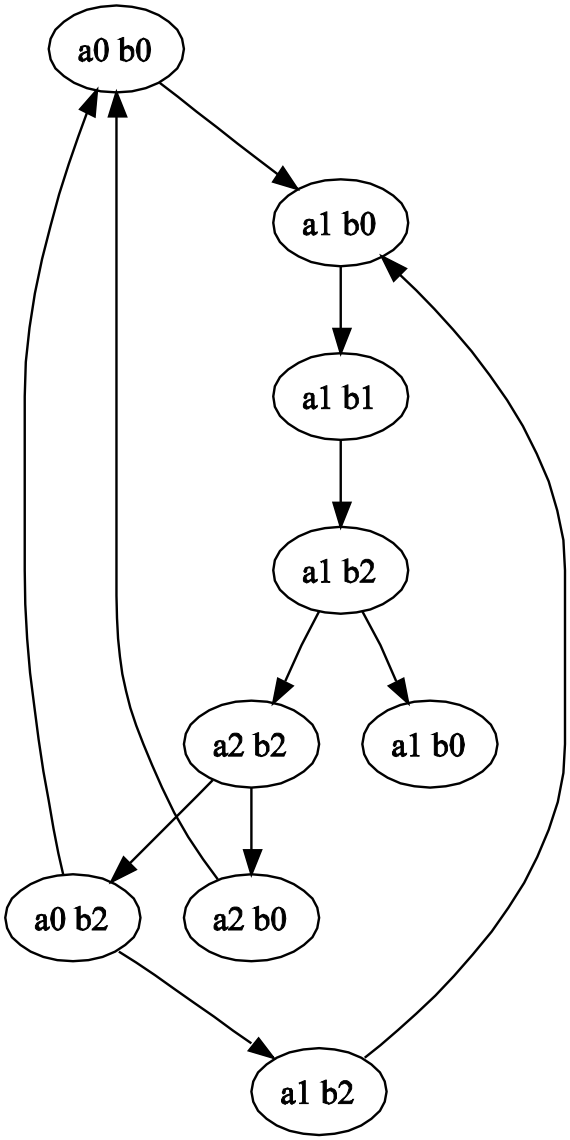
Proc2 u ev2 unutarnjem prijelazu priprema rješenja.

Za vježbu: napisati definicije za oba automata. Koristiti folije #24 i #25

U kojoj mjeri utječe izbor implementacijskog jezika na do sada prikazano ?

Kako prikazati unutarnji prijelaz ev1 i ev2 u dijagramu pod b)?

d) graf dostupnosti za Proc1 i Proc2



Vježba:

dodati labele na grane

e) Expect implementacija KVAJE02

```
#!/bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

package require Expect

set timeout 1

spawn bash
send "./kvaje02\r"
expect "a="
# mora biti iza " blank pa 1  (i u c da ne uzme - opcija !)
send " 1\r"
expect "b="
send   " -5\r"
expect "c="
send " 6\r"
expect "Rez*"
#interact
```

e) logičke formule – verifikacija (provjera za kompleksna rješenja i provjera *da li program regularno završava*)

```
D=b*b-4*a*c;
#define predicat1 D >0
/*assert (predicat1)*/
  if(D < 0) {
/*     printf("Test (D<0) D=%f\n",D); */
    }/*return -1;*/
/*     else  printf("Test (D>=0) D=%f\n",D); */
    x1=(-b + sqrt(D))/(2*a);    x2=(-b - sqrt(D))/(2*a);
tcv = 1;
#define predicat2 tcv == 1
/*assert (predicat2)*/
```

- Možda je za neke ulazne podatke D manji od nule:

  F !predicat1

- Da li će program regularno završavati

  F predicat2

Za vježbu:

(1) Implementirati kao programsku potporu u jeziku Java.

(2) Nacrtati pripadne UML dijagrama

Kako testirati dobivene Java programe ?

Automatizacija

- Formalne metode – područje koje prati razvoj sustava a tako i razvoj programske potpore primjenom rigidnog, strogog matematičkog aparata.
- formalne metoda se mogu primjeniti tijekom svake faze razvoja u “*modelu vodopada*”

Tko želi znati više:

pogledati na www neku od formalnih metoda i razmisliti što znači za primjer KVAJE ili vlastiti program

Programiranje i modeliranje programa

- Postoje dva načina, svaki sa prednostima i manama:
  - Odmah nakon što je problem/zadatak usvojen -razviti traženu programsku potporu. To se svodi na programiranje *algoritma* koji rješava naš zadatak. Velika je vjerojatnost da je netko ranije imao sličan problem koji je rješio i ostavio rješenje za širu upotrebu.
  - Rješiti zadatak/problem bez programiranja: definirati model koji “u sebi” ima rješenje

Iz imperativnog u logički program . . .

(primjer)

Polazimo od poznatog primjera (KVAJE):

```
D=b*b-4*a*c;
#define predicat1 D >0
/*assert (predicat1)*/
  if(D < 0) {
/*     printf("Test (D<0) D=%f\n",D); */
    }/*return -1;*/
/*     else  printf("Test (D>=0) D=%f\n",D); */
    x1=(-b + sqrt(D))/(2*a);    x2=(-b - sqrt(D))/(2*a);
tcv = 1;
#define predicat2 tcv == 1
/*assert (predicat2)*/
```

SAT

CNF sustav formula za 3-SAT ($x_{ij}=\{0,1\}$):

$$(x_{11} \text{ OR } x_{12} \text{ OR } x_{13}) \text{ AND}$$
$$(x_{21} \text{ OR } x_{22} \text{ OR } x_{23}) \text{ AND}$$
$$(x_{i1} \text{ OR } x_{i2} \text{ OR } x_{i3}) \text{ AND}$$
$$(x_{n1} \text{ OR } x_{n2} \text{ OR } x_{n3})$$

x_{ij} može biti i negacija ($\neg x_{ij}$)

- CNF sustav predstavlja u smislu matematičke logike sud

12.3. Semantika

- **Semantika je izvršna**
  - semantika nema ništa zajedničkog sa mogućnošću izvođenja programa
  - semantika nije izvršna specifikacija programa (iako su ponekad vrlo slične na prvi pogled)

- **Semantika opisuje ponašanje**
  - semantika mora moći opisati svaki jezik i svaku paradigmu
  - ponašanje, struktura, arhitektura jezika ili sustava može ali ne mora imati veze sa ponašanjem
  - ako npr. jezik opisuje strukturu ima svoju semantiku, to nije jezik za opis ponašanja nego strukture
  - semantiku i ponašanje nije dobro izjednačiti: ponašanje nije isto što i značenje (smisao)

- **Semantika opisuje ponašanje cijelog sustava**
  - postoji semantika sustava
  - sustav je implementiran u jednom ili više jezika, svaki od njih ima svoju semantiku koja **nije** isto što i semantika sustava
  - dakle za neku cjelinu poput sustava postoji više razina semantike koja ga opisuje

- Semantika opisuje ponašanje individualnih instrukcija
  - možemo govoriti o semantici instrukcije, fraze, sustava, aplikacije . . .
  - semantika pruža dublji uvid u sustav i prelazi okvire semantike pojedinačne instrukcije

- **Semantika znači opisati program matematičkim formalizmom**
  - matematički formalizam ne predstavlja istovremeno semantiku program
  - upotreba matematičkih simbola ne garantira formalnu definiciju semantike programa – matematički formalizam je sredstvo koje pridjeljuje jednoznačno značenje-smisao programu