**IME I PREZIME:** Ak. god. 2014/2015

**JMBAG:**

**2. domaća zadaća iz Formalnih metoda u oblikovanju sustava**

**Java PathFinder**

Najprije je potrebno instalirati sustav Java PathFinder prema uputama u datoteci

"Java\_PathFinder\_instalacija.pdf"

Svrha 1. dijela 2. domaće zadaće je upoznavanje s projektom jpf-core i pokretanje provjere modela jednostavnih primjera programa. U 2. dijelu domaće zadaće uvode se dodatni razredi slušači koji nadograđuju osnovnu funkcionalnost projekta jpf-core. 3. dio domaće zadaće pokriva izvođenje provjere modela nad primjerima iz projekta jpf-aprop, koji je dodatni projekt koji se može koristiti za provjeru različitih anotacija u programu. Konačno, u 4. dijelu domaće zadaće studenti će napraviti svoj projekt od početka, uključiti korištenje projekata jpf-core i jpf-aprop te provjeravati model zadanog programa uz stalne izmjene.

**1. dio - Provjera modela za jednostavne primjere u projektu jpf-core**

Proučite strukturu projekta jpf-core. Primjeri nad kojim će se raditi provjera modela nalaze se u paketu src/examples. Osim ako nije drugačije zadano, programi se mogu pokrenuti desnim klikom na odgovarajuću \*.jpf datoteku i odabirom opcije "Verify..." Alternativno, ako niste uspjeli podesiti *plugin* "Verify..." da radi (vidjeti instalacijske upute), moguće je pokrenuti program unoseći puni put u komandnoj liniji koji je obično ovakvog oblika:

java -jar C:\Users\Korisnik\projects\jpf\jpf-core\build\RunJPF.jar

+shell.port=4242 C:\Users\Korisnik\Documents\NetBeansProjects\jpf- core\src\examples\HelloWorld.jpf

**1.1.** Otvorite konfiguracijsku datoteku projekta jpf-core koja se naziva jpf.properties i koja se nalazi u korijenskom direktoriju projekta. Do datoteke se može doći putem kartice "Files" koja se nalazi pokraj kartice "Projects" u NetBeansu ili putem Windows Explorera.

Navedite koja se *defaultna* strategija koristi za pretragu prostora stanja. Koja se standardna svojstva provjeravaju prilikom pretrage? Osim naziva strategije i svojstava, navedite i puna kvalificirajuća imena razreda u projektu jpf-core koji za to služe.

Defaultna strategija za pretragu prostora stanja:

gov.nasa.jpf.search.DFSearch.

Standardna svojstva koja se provjeravaju prilikom pretrage:

gov.nasa.jpf.vm.NotDeadlockedProperty

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

**1.2.** Proučite najjednostavniji primjer aplikacije HelloWorld.java. Što se ispisuje pokretanjem provjere modela tog programa u dijelu nakon search started? Koje su pogreške dojavljene?

Ispisuje se ono što program ispisuje, “I won't say it!”. Nakon ispisa programa slijede rezultati - “no errors detected” - nisu pronađene pogreške. Prije ispisa kraja pretrage još se ispisuje statistika.

**1.3.** Proučite primjer ograničenog međuspremnika BoundedBuffer.java. Navedite koji su sudionici u ovom primjeru.

Sudionici u primjeru su proizvođači i potrošači. Proizvođači šalju podatke - string duljine 8 znakova. Međuspremnik u koji se spremaju podaci je ograničen na 2 objekta (2 stringa).

**1.4.** Kojim mehanizmom u Javi su ostvareni ti sudionici? Koje metode koriste koji sudionici?

Sudionici nasljeđuju razred Thread te implementiraju metodu run(). Koriste dvije sinkrone operacije - get() i put() koje koriste semafore za sinkronizaciju pomoću wait() i notify() metoda.

**1.5.** Pokrenite aplikaciju ograničenog međuspremnika. Koje svojstvo je narušeno izvođenjem ovog programa koristeći argumente navedene u konfiguracijskoj datoteci (2,4,1)? Što se dogodilo s pojedinim sudionicima? Kolika je bila veličina međuspremnika bila ovom slučaju?

Veličina međuspremnika je 2, imamo 4 proizvođača i 1 potrošača. Svojstvo koje je narušeno je gov.nasa.jpf.vm.NotDeadlockedProperty - došlo je do potpunog zastoja. Sve dretve (4 proizvođača i 1 potrošač) se nalaze u stanju WAIT.

**1.6.** Pokrenite istu aplikaciju, samo s argumentima (4,1,1). Kakva je sad situacija? (Napomena: NetBeans će vas možda gnjaviti da ne može spremiti izmjene u datoteci \*.jpf jer da je datoteka otvorena vjerojatno samo za čitanje. Obično spremanje promjena ipak uspješno prođe nakon što prođe neko vrijeme (manje od minute), no u slučaju da ne prođe, možete napraviti Save As... i pohraniti datoteku pod drugim imenom)

U ovom slučaju, za veličinu međuspremnika 4, jednog proizvođača i jednog potrošača ne dolazi do potpunog zastoja.

**1.7.** Opišite ukratko što rade i nad čime se pokreću Javine metode wait() i notify().

Metode wait() i notify() se pozivaju nad trenutno aktivnom dretvom (nad dretvom koja je pozvala te metode). (Možda se mogu pozvati nad bilo kojim objektom?)

wait() - dretva prestaje biti vlasnik monitora i čeka dok neka druga dretva ne pozove notify().

notify() - odabire se slučajno jedna dretva koja je čekala i koja se dalje natječe za izvođenje kritičnog odsječka kad trenutna dretva završi s njim.

notifyAll() - bude se sve dretve koje čekaju nad monitorom i onda se dalje natječu za izvođenje kritičnog odječka.

**1.8.** Obrazložite ukratko zašto dolazi do narušavanja svojstva u ovom primjeru.

U (2, 4, 1) dolazi do potpunog zastoja. Akcije koje su se izvele su sljedeće:

PUT from P1

PUT from P1

GET from C1

GET from C1

P1 će napuniti 2 objekta u međuspremnik veličine 2 koji je zatim popunjen. P1 će pozvati notify() te dalje spavati, na što se budi C1. C1 će pročitati dva objekta iz međuspremnika. Nakon toga se više ne pobuđuju prave dretve jer imamo nedeterminizam prilikom odabira dretvi koje će se probuditi i mogućnost zastoja. Ako bismo zamijenili notify() s notifyAll(), ne dolazi do potpunog zastoja. U našem primjeru svi čekaju na BoundedBuffer@163, tko god to bio (nemam pojma otkud taj 163).

**1.9.** Proučite primjer Rand.java i pridruženu konfiguracijsku datoteku Rand.jpf. Što specificira konfiguracijska naredba cg.enumerate\_random = true i zašto je ona bitna za ovaj problem?

Program generira nasumični cijeli broj u rasponu [0, 1] i pridružuje ga varijabli a te nasumični cijeli broj u rasponu [0, 2] i pridružuje ga varijabli b. Zatim se računa varijabla c, prilikom čega u ovisnosti o vrijednostima varijabla a i b može doći do dijeljenja s nulom, što naravno baca iznimku.

Prilikom pokretanje programa na JVM-u moguće je da veoma rijetko ili nikada ne dobijemo slučaj dijeljenja s nulom. Međutim, JPF provjerava i to. Ako postavimo cg.enumerate\_random = true, JPF će razmotriti sve moguće vrijednosti za varijable a i b te će izgraditi stablom i njegovom pretragom naći slučajeve u kojima dolazi do dijeljenja s nulom.

**1.10.** Pokrenite aplikaciju za verifikaciju. Koje svojstvo je ovdje narušeno? Što je programer ovog ili ovome sličnog koda previdio? Kako se mogao unaprijed osigurati da se cijeli sustav ne sruši? Koje su konkretne vrijednosti varijabli a i b srušile program?

Za vrijednosti:

a = 0, b = 2 (a = 1, b = 1 JPF ne pronalazi jer staje nakon prvog exceptiona)

Narušeno je svojstvo:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.ArithmeticException: division by zero

Programer je previdio mogućnost kombinacije vrijednosti varijabli a i b kada dolazi do dijeljenja s nulom. Rješenje je postaviti jedan if ili try-catch.

**1.11.** Sad izbrišite specifikaciju cg.enumerate\_random = true iz konfiguracijske datoteke te pokrenite aplikaciju. Što se sad dogodilo? Objasnite.

Verifikacija ne javlja nikakve probleme jer se ne provjeravaju sve moguće kombinacije vrijednosti nasumično generiranih brojeva.

**2. dio - Provjera modela u projektu jpf-core korištenjem dodatnih slušača**

**2.1.** Proučite primjer Racer.java i konfiguracijsku datoteku Racer.jpf. Napravite kopiju konfiguracijske datoteke koju ćete nazvati Racer\_2.jpf i u kojoj ćete izbrisati liniju listener=gov.nasa.jpf.listener.PreciseRaceDetector

Pokrenite aplikaciju za verifikaciju putem Racer.jpf i potom putem Racer\_2.jpf.

Koje svojstvo je narušeno u slučaju Racer\_2.jpf, a što piše pod error 1 u slučaju

Racer.jpf?

U Racer\_2.jpf je narušeno svojstvo:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.ArithmeticException: division by zero

Greška u Racer.jpf:

gov.nasa.jpf.listener.PreciseRaceDetector

race for field Racer@15b.d

**2.2.** Opišite zašto može doći do problema prilikom izvođenja ovog primjera. Može li instanca dretve t pristupiti liniji koda int c = 420 / racer.d; ?

Do problema utrke dretava dolazi jer one čekaju gotovo isto vrijeme te ovisi o raspoređivaču dretava hoće li se originalna dretva probuditi prije novostvorene dretve. U slučaju da novostvorena dretva dođe prva na red, promijeniti će vrijednost varijable d na 0 te će u originalnoj dretvi doći do dijeljenja s nulom. Ako novostvorena dretva nije stigla postaviti vrijednost na 0, originalna dretva će dijeliti s 42.

Instanca dretve t ne može pristupiti navedenoj liniji koda jer se ona nalazi unutar main funkcije koju izvršava samo početna dretva. Druga dretva koju stvorimo izvršava samo kod unutar svoje run() metode.

**2.3.** Otvorite kod razreda gov.nasa.jpf.listener.PreciseRaceDetector. Ukratko pojasnite (na temelju komentara razreda) koja je ideja kod implementacije detektora utrke za resursom. Također navedite koji Adapter nasljeđuje ovaj slušač i koje metode nadjačava.

Ideja iza detektora utrke za resursom je ta da JPF prvo pronalazi sve „racer“ objekte koji se dijele između dretava (u našem slučaju varijabla d). Zatim izvršava dretve u svim mogućim redoslijedima u kojima se objektu pristupa iz različitih dretvi.

Detektor nasljeđuje adapter PropertyListenerAdapter. Metode koje nadjačava: check, reset, getErrorMessage, choiceGeneratorSet, executeInstruction.

**2.4.** Proučite kod primjera NumericValueChecker.java i konfiguracijsku datoteku

NumericValueChecker.jpf. Zatim pokrenite aplikaciju. Koju grešku je dojavio JPF?

Dojavljenja greška:

gov.nasa.jpf.listener.NumericValueChecker

local variable someVariable out of range: 12345,000000 > 42,000000

**2.5.** Primijetite na koji način je specificirano u konfiguracijskoj datoteci na koju varijablu i na koji način se odnosi provjeravanje raspona numeričkih vrijednosti. Pogledajte sad kod odgovarajućeg slušača koji implementira provjeravanje raspona vrijednosti. Koje su dvije mogućnosti rada tog slušača (što se može provjeriti)? Navedite i sintaksu tih provjera.

range.vars = 1

range.1.var = NumericValueCheck.main(java.lang.String[]):someVariable

range.1.min = 0

range.1.max = 42

ili

range.vars = someVariable

range.someVariable.var = NumericValueCheck.main(java.lang.String[]):someVariable

range.someVariable.min = 0

range.someVariable.max = 42

Slušač može provjeriti je li vrijednost varijable iznad maksimalne ili ispod minimalne zadane vrijednosti.

**2.6.** Proučite kod primjera TestExample.java i konfiguracijsku datoteku TestExample-coverage.jpf. Korištenjem slušača CoverageAnalyzer omogućena je analiza pokrivanja koda. Pokrenite aplikaciju i promotrite rezultantnu tablicu koju je ispisao CoverageAnalyzer. Koji razred je bio bolje pokriven s ispitnim primjerima u metodi main? Koje sučelje je morao implementirati ovaj slušač kako bi izmijenio izgled ispisa izvještaja?

Od T1 su pokrivene 3/3 metode (9/11 LOC), a od T2 su pokrivene samo 2/3 metode (2/6 LOC) => T1 je bolje pokriven ispitnim primjerima

Da bi slušač mogao izmijeniti izgled ispisa izvještaja mora implementirati PublisherExtension sučelje.

**2.7.** S obzirom na rezultate ispisa i dani kod, koja bi to bila metoda <init>() koja piše u tablici?

Defaultni konstruktor razreda.

**2.8.** Dodajte u konfiguracijsku datoteku TestExample-coverage.jpf pod razrede koje treba uključiti za provjeru dodatno i sam razred TestExample, uz postojeće razrede T1 i T2. Spremite datoteku i pokrenite aplikaciju za verifikaciju. Proučite rezultat. Iz samog koda, očito je da će se proći kroz sve linije metode main. Koje naredbe (linije koda) analizator pokrivanja preskače kad izvještava da je prošao kroz samo 3/8 linija koda metode main (osma linija koda uključuje implicitni return;)? Kojom specifikacijom bi isključili ispisivanje provjeravanja pokrivenosti grana u izlaznoj tablici?

Prošao je kroz naredbe definiranja varijabli i stvaranju njima pripadajućih objekata (+ return;). Nije prolazio kroz assert naredbe.

Specifikacijom coverage.show\_branches = false isključujemo ispisivanje provjeravanja pokrivenosti grana u izlaznoj tablici.

**3. dio - Provjera modela u projektu jpf-aprop**

Instalirajte projekt (paket) jpf-aprop koji služi za provjeru specifičnih svojstava programa pisanih u Javi koja su zadana u kodu u obliku anotacija (oznaka @). Za instalaciju odaberite redom "Team" -> "Remote" -> "Clone other..." i upišite pod Repository URL:

<http://babelfish.arc.nasa.gov/hg/jpf/jpf-aprop>

Po završetku kloniranja, nakon što se projekt jpf-aprop našao u otvorenim projektima, dodajte u datoteku site.properties, koju ste ranije stvorili u postupku instalacije, sljedeće retke na kraj datoteke i pohranite promjene.

# annotation properties extension jpf-aprop = ${jpf.home}/jpf-aprop extensions+=,${jpf-aprop}

Prvo što možete zamijetiti sa skinutim paketom je to da postoje uskličnici uz dosta razreda koda paketa jpf-aprop. Prvo što trebate napraviti je da otvorite skriptu build.xml od projekta jpf-aprop i da potražite redak property name="src\_level" koji postavite na vrijednost 8 (umjesto dosadašnjih 6). Zatim desnim klikom na projekt jpf-aprop pa redom "Properties" -> "Java sources" -> "Source Level" postavite na JDK 1.8. Sada pokrenite skriptu build.xml (desni klik pa "Run Target" -> "build").

Skripta bi trebala vratiti 62 pogreške i neuspjelu izgradnju. Razlog tome je taj što se jpf-

aprop paket nije mijenjao u zadnjih nekoliko godina u odnosu na jpf-core koji je više- manje stalno evoluirao. Ipak, razlika nije toliko drastična da se ne bi mogla uglavnom popraviti pažljivim izmjenama u kodu.

Promotrite pogreške koje su vam dojavljene prilikom izgradnje. Većina pogrešaka vezana je uz import razreda InvokeInstruction, ReturnInstruction, FieldInstruction i InstanceFieldInstruction i to iz paketa gov.nasa.jpf.jvm.bytecode.

Kako bi to riješili, u svim razredima za koje je dojavljen taj problem potrebno je ručno

napraviti izmjene (najlakše uz pomoć CTRL+H, pri čemu treba zamijeniti sva pojavljivanja (Replace All) razreda InvokeInstruction s JVMInvokeInstruction, ReturnInstruction s JVMReturnInstruction, FieldInstruction s JVMFieldInstruction te InstanceFieldInstruction s JVMInstanceFieldInstruction iz istoga paketa (uvjerite se da ti razredi zaista

postoje u jpf-coreu). Možete povremeno pokrenuti nanovo build.xml da vidite kako se broj prijavljenih pogrešaka smanjuje kako radite ispravke.

Na kraju bi trebale ostati tri pogreške, dvije vezane uz razred ArrayInstruction i jedna vezana uz razred JVMInvokeInstruction. Prve dvije ćete ispraviti tako što ćete preimenovati razred ArrayInstruction u ARRAYLENGTH is istog paketa. Treću pogrešku (poziv metode hasAttrRefArgument) vjerojatno nećete moći ispraviti. Potrebno je zakomentirati (uz pomoć /\* ... \*/ ) dio metode executeInstruction u razredu koji prijavljuje tu pogrešku i to od linije koja počinje s "if(call.hasAttrRefArgument

..." pa do uključivo pretpretzadnje vitičaste zagrade. Sad bi build.xml trebao uspješno proći izgradnju projekta i svi preostali uskličnici (pogreške) bi trebali nestati.

**3.1.** Proučite datoteku jpf.properties projekta jpf-aprop. Koja se standardna svojstva provjeravaju prilikom pretrage kad se koristi projekt jpf-aprop? Gdje je to uopće definirano? Također, navedite put do direktorija s razredima koji se kao primjeri provjeravaju uz pomoć jpf-aprop. Koje svojstvo to pokazuje u datoteci jpf.properties?

jpf.properties projekta jpf-aprop ne navodi nikakva standardna svojstva koja se provjeravaju. Svojstva koja će se provjeravati standardno nisu definirana u ekstenziji, pa će se (pretpostavljam) provjeravati ona koja su postavljenja u jpf-core, a to su:

gov.nasa.jpf.vm.NotDeadlockedProperty,\

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

jpf-core standardna svojstva su definirana pod search.properties u jpf.properties projekta jpf-core.

Put do direktorija s razredima koji se kao primjeri provjeravaju uz pomoć jpf-aprop: src/examples. Svojstvo koje to pokazuje je: jpf-aprop.sourcepath.

**3.2.** Proučite jednostavni primjer ConstViolation.java i pridruženu konfiguracijsku datoteku ConstViolation.jpf te odgovarajućeg slušača. Opišite što se događa u kodu razreda ConstViolation.java. Koja metoda je označena s [@Const](mailto:@Const) anotacijom i što to točno znači?

Metoda razreda ConstViolation označena s @Const anotacijom je dontDoThis() koja poziva metodu foo() koja u svome tijelu mijenja vrijednost podatkovnog člana d razreda ConstViolation na 42. U main funkciji stvara se objekt razreda ConstViolation te se nad njime poziva metoda dontDoThis().

Metode označene s @Const imaju ograničenje da ne smiju mijenjati članove svog razreda (bilo statičke razreda ili dinamičke konkretnog objekta). U slučaju kršenja podiže se iznimka AssertionError. Napomena: čak i ako funkcija ne mijenja vrijednosti unutar svog tijela, iznimka će se podići i u slučaju da metode pozvane iz @Const-označene metode također mijenjaju vrijednosti. Znači - gleda se stanje neposredno nakon ulaska u @Const-označenu metodu te se uspoređuje sa stanjem neposredno prije izlaska iz @Const-označene metode.

**3.3.** Pokrenite tu aplikaciju za verifikaciju. Koja vrsta pogreške je dojavljena?

Bačena je iznimka:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError: instance field write within const context: int ConstViolation.d

**3.4.** Proučite primjer ContractViolation.java i pridruženu konfiguracijsku datoteku ContractViolation.jpf. Navedite koje sve slušače koristi ovaj program (puna kvalificirajuća imena).

gov.nasa.jpf aprop.listener.NonnullChecker

gov.nasa.jpf.aprop.listener.LockChecker

gov.nasa.jpf.aprop.listener.ConstChecker

gov.nasa.jpf.aprop.listener.NonSharedChecker

gov.nasa.jpf.aprop.listener.ContractVerifier@pbc

**3.5.** Pronađite u strukturi projekta odgovarajućeg slušača u kojem se provjerava svojstvo nonshared.throw\_on\_cycle. U kodu pronađite i napišite koju bi iznimku bacio JPF ako bi dretva bila uhvaćena u ciklusu nad objektom koji nije predviđen za višedretvenost.

Slušač: gov.nasa.jpf.aprop.listener.NonSharedChecker

U slučaju da je dretva uhvaćena u ciklusu nad objektom koji nije predviđen za višedretvenost slušač baca pogrešku java.lang.AssertionError s porukom „NonShared object: <oznaka\_objekta> accessed in live thread cycle: <oznaka\_dretve>“.

**3.6.** Što znače i čemu služe anotacije [@Requires,](mailto:@Requires) [@Invarian](mailto:@Invariant)[t i @Ensures](mailto:@Ensures) u kodu programa

ContractViolation.java?

Prilikom oblikovanja putem ugovora metode imaju određene preduvjete koji moraju biti ispunjeni da bi se one pokrenule izvršavati (@Requires, precondition). Nakon što su gotove s izvršavanjem, osiguravaju određena svojstva (@Ensures, postcondition). Tokom izvođenja metode, osigurava se da su u svim trenutcima zadovoljenja svojstva invarijante (@Invariant).

**3.7.** Pokrenite aplikaciju za verifikaciju. Koji dio ugovora je narušen? Napišite pogrešku koja je dojavljena.

Dojavljuje se jedna greška vezana uz funkciju getLoopCount():

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError: postcondition violated: "(Result >= 0) AND (Result > 0)", values={Result=0}

**3.8.** Pažljivo proučite dojavljenu pogrešku. Metoda kojeg točno razreda je izazvala narušavanje ugovora?

Pozvala se metoda getLoopCount() razreda ContractViolation. Iako metoda tog razreda kao postcontidion dozvoljava vrijednost 0, postcondition nadrazreda ContractViolationBase je stroži te ne dozvoljava nulu pa je zapravo zbog toga došlo do pogreške. Generalno je dobra ideja uvjete i invarijante nadrazreda postaviti slabije od onih u izvedenima (ovdje je to napravljeno obratno).

JPF radi konjunkciju svih uvjeta u hijerarhiji razreda za zajedničke metode koje niži razredi overrideaju. U našem primjeru se tako provjeravalo (Result >= 0) AND (Result > 0). >= je iz ContractViolation, a > iz ContractViolationBase.

**3.9.** Promijenite ugovore dviju metoda tako da obje ispituju uvjet (Result>=0). Spremite izmijenjenu datoteku ContractViolation.java. Pokrenite ponovno skriptu build.xml. Kad se izmijenjeni primjer preveo, ponovno ga pokrenite. Kakva je sad situacija?

Sada nema pogrešaka.

**4. dio - Provjera modela vlastitog projekta**

**4.1.** U NetBeansu napravite novi projekt ("File" -> "New Project" -> "Java Application") koji ćete nazvati JavaFV. Napravite ga bez razreda JavaFV s metodom main. Zatim desnim klikom na Source Packages napravite novi paket pod nazivom fv, a onda desnim klikom na paket fv napravite novi razred pod imenom Verifikacija.java i statičkom metodom main (unutar razreda napišite public static void main(String[] args){}). Isto tako, izbrišite redak package fv; iz napravljene datoteke Verifikacija.java.

Napravite unutar istoga paketa novu datoteku (desni klik na fv, pa "New" -> "Other..." ->

"Other" -> "Empty File" i nazovite ga Verifikacija.jpf. U tu datoteku dodajte zasad

samo jedan redak kojim ćete omogućiti pokretanje razreda Verifikacija.java i spremite ju. Kako izgleda taj redak?

target = Verifikacija

U korijenskom direktoriju projekta JavaFV zatim napravite datoteku jpf.properties

jednostavnog sadržaja:

JavaFV = ${config\_path} JavaFV.classpath=\

${JavaFV}/build/classes

JavaFV.sourcepath=\

${JavaFV}/src/fv

Na kartici "Projects" kliknite desnim klikom na vaš projekt JavaFV i odaberite "Clean and build".

Nakon što se projekt izgradio, provjerite da se međukod Verifikacija.class nalazi

pod direktorijem build/classes. Probajte pokrenuti verifikaciju koja bi trebala proći bez pogrešaka.

Objasnite zašto je redak JavaFV.classpath=\

${JavaFV}/build/classes

nužno navesti u datoteci jpf.properties?

Nužno je navesti putanju do međukoda kako bi ga JPF mogao učitati.

**4.2.** Sada izmijenite sadržaj datoteke Verifikacija.java tako da sadrži kod koji se nalazi u istoimenoj datoteci koja se nalazi u repozitoriju kolegija FMUOS (pod 2. DZ). Također, izmijenit ćete sadržaj datoteke Verifikacija.jpf tako da sadrži specifikacije prema istoimenoj datoteci koja se nalazi u repozitoriju kolegija. Nakon kopiranja koda i specifikacija spremite datoteke, no nećete moći prevesti datoteku Verifikacija.java budući da sadrži importe koji su nepoznati samom projektu JavaFV.

**4.3.** Potrebno je uključiti izgrađene knjižnice (.jar) od jpf-core i jpf-aprop u projekt JavaFV kako bi se kod razreda Verifikacija.java mogao prevesti. To se radi tako da odaberete JavaFV pa desni klik, a zatim "Properties" -> "Libraries" -> "Add JAR/Folder". Pronađite u direktoriju jpf-core -> build -> jpf.jar, jpf-classes.jar i jpf- annotations.jar te ih dodajte. Ostali JAR-ovi nisu bitni. Od projekta jpf-aprop potrebno je dodati samo JAR jpf-aprop-annotations.jar. Odaberite "Ok". Pogreške bi sada trebale nestati. Sad prevedite Verifikacija.java (desni klik -> "Compile File").

**4.4.** Pokrenite aplikaciju za verifikaciju. Koja pogreška vam je dojavljena? Objasnite zašto je došlo do te pogreške s obzirom na konfiguracijsku datoteku i zadani kod.

Javljena je pogreška:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError: null assignment to @Nonnull instance field: Verifikacija.id

prilikom poziva v.setId(null)

Pogreška je javljana jer imamo postavljen slušač gov.nasa.jpf.aprop.listener.NonnullChecker te u programskom kodu imamo anotaciju @Nonnull private String id koja provjerava da id ne bude postavljen na NULL vrijednost.

**4.5.** U konfiguracijsku datoteku dodajte ovaj redak na kraj:

search.multiple\_errors = true

Ovime se prolazi svim putevima izvođenja kroz program i dojavljuje se za svaki put

izvođenja prva pogreška na koju se naletilo. Pokrenite aplikaciju za verifikaciju. Koja je razlika između prethodnog ispisa pogrešaka i sadašnjega?

Pojavile su se još dvije pogreške (jednake). Prva je ista kao i prije, a pojavile se:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError: null assignment to @Nonnull instance field: Verifikacija.id

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError: null assignment to @Nonnull instance field: Verifikacija.id

**4.6.** Uklonite redak search.multiple\_errors = true te zakomentirajte redak u kodu koji smatrate odgovornim za dojavu ove vrste pogreške. Prevedite kod i pokrenite ponovno aplikaciju za verifikaciju. Koja se sada pogreška pojavila, na kojem retku koda i zašto je prijavljena?

Javila se pogreška:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError: write of const instance field outside constructor: java.lang.String Verifikacija.DNA

Pozivom v.setDNA("CTGA") pokušava se promijeniti vrijednost varijable DNA razreda Verifikacija, što nije dopušteno pošto je ta varijabla označena @Const anotacijom.

**4.7.** Zakomentirajte redak u kodu koji smatrate odgovornim za dojavu ove vrste pogreške. Prevedite kod i pokrenite ponovno aplikaciju za verifikaciju. Koja je sada pogreška dojavljena? Objasnite zašto se ova pogreška dogodila.

Javila se pogreška:

gov.nasa.jpf.vm.NoUncaughtExceptionsProperty

java.lang.AssertionError

Prilikom assert (vel < 100.0)

Raspon vrijednosti za vel je definiran u .jpf datoteci te je [20, 100]. 100 < 100 = false pa assert ne prolazi.

**4.8.** Izmijenite naredbu (assert) u kodu tako da više ne dolazi do ove vrste pogreške. Prevedite kod i provjerite da se pogreška zaista više ne događa. Koji ste broj trebali navesti kao uvjet u naredbi (assert) da ne dođe do pogreške?

assert (vel <= 100.0) (ili manje točno assert (vel < 101.0))

Više nema pogrešaka prilikom verifikacije.

**4.9.** Provjerite metode razreda gov.nasa.jpf.vm.Verify. Postoje li metode getDouble() i getInt() bez argumenata? Objasnite. Objasnite na primjeru što su to generatori izbora i koja je namjena navođenja heuristika pri korištenju generatora izbora. Koja se heuristika koristila u primjeru Verifikacija.java?

Ne postoje metode getDouble() i getInt() bez argumenata. Ako predamo prazan argument, zapravo se radi o generatorima nasumičnih brojeva. Potrebni su argumenti za raspon ili za ključ svojstva definiranog u .jpf datoteci (kao što je u našem primjeru „velocity“.

ChoiceGenerators je mehanizam JPF-a da sustavno istražuje prostor stanja kako bi došao do rješenja.Kako bi se smanjio broj mogućnosti dodjele vrijednosti pri provjeri int, double i drugih tipova varijabli, uvode se heuristike pretraživanja. Primjenom heuristike odstupa se od temeljnog (ali idealističkog) zahtjeva kod provjere modela: da se ispitaju svi putevi kroz program. Ispituju se samo „zanimljivi“ putevi.

Pojašnjenje: varijable tipa double mogu poprimiti mnoštvo vrijednosti. Generatorom izbora suzimo moguće vrijednosti na neki „zanimljivi“ ograničeni skup kako bi se heurističkim pretraživanjem brže došlo do rješenja prilikom provjere modela.

Implicitno je korištena heuristika nasumičnog odabira.