IME I PREZIME:_____ Ak. god. 2019./2020.

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 potrebne izmjene.

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

Proučite strukturu projekta `jpf-core`. Primjeri nad kojima će se raditi provjera modela nalaze se u paketu `src/examples`. Osim ako nije drugačije zadano, programi se u NetBeansu 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.

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

Defaultna strategija koja se koristi za pretragu prostora stanja u JPF-u je *DFSearch* (*gov.nasa.jpf.search.DFSearch*).

Standardna svojstva koja se provjeravaju prilikom pretrage su:
- potpuni zastoj
(*gov.nasa.jpf.vm.NotDeadlockedProperty*)
- neuhvaćene iznimke
(*gov.nasa.jpf.vm.NoUncaughtExceptionsProperty*)

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

Nakon *search started* ispisuje se *„I won't say it!"* te rezultati i statistika provjere modela. U odjeljku rezultata niti jedna pogreška nije dojavljena (*„no errors detected"*).

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

Sudionici u ovom primjeru su proizvođači (*producers*) i potrošači (*consumers*).

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

Sudionici su ostvareni kao dretve u sinkronizacijskom mehanizmu pristupa kritičnom odsječku. Sudionici koriste metode *get()* i *put()* označene sa *synchronized*, a te metode u sebi pozivaju metode za promjene stanja dretvi: *wait()* i *notify()*.

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 u ovom slučaju?

Narušeno je svojstvo potpunog zastoja, odnosno došlo je do potpunog zastoja.

Dretve svih sudionika (4 proizvođača i 1 potrošač) stavljene su u stanje čekanja (*waiting*).

Veličina međuspremnika je bila 2.

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)

Nema dojavljenih pogrešaka, odnosno nije došlo do potpunog zastoja.

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

Dretva koja se nalazi u kritičnom odsječku (vlasnik je monitora) može nad objektom pozvati metode: *wait()*, *notify()* i *notifyAll()*.

wait() - dretva prestaje biti vlasnik monitora i čeka dok neka druga dretva ne pozove *notify()* nad tim objektom

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

1.8. Obrazložite ukratko (i precizno) zašto dolazi do narušavanja svojstva u ovom primjeru.

Do narušavanja svojstva potpunog zastoja dolazi zbog korištenja metode *notify()*. Događa se slučaj da pozivanjem metode *notify()* slučajno odabrana dretva propusti svoju obavijest i onda čeka da netko drugi pozove metodu *notify()*, ali to ne se može dogoditi, jer zbog njenog propuštanja obavijesti nitko drugi ne može više pozvati metodu *notify()*. Metodom *notifyAll()* izbjegao bi se taj slučaj, jer je jako mala vjerojatnost da sve dretve propuste svoju obavijest.

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?

Navedena konfiguracijska naredba specificira svojstva da se gledaju sve mogućnosti prilikom slučajnog izbora koji se pojavljuje unutar programa (npr. za *Random.nextInt(2)* postoje mogućnosti 0 ili 1). Za ovaj problem bitna je zbog provjeravanja eventualnog 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?

Ovdje je narušeno svojstvo neuhvaćenih iznimki, a u ovom slučaju neuhvaćena iznimka bila je dijeljenje s nulom:
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero

Programer je previdio slučaj gdje može doći do dijeljenja s nulom, konkretno u ovoj aplikaciji do toga može doći u formuli *c=a/(b+a-2)*

Izbjegavanje rušenja cijelog programa najlakše se moglo postići stavljanjem navedene formule u *try-catch* blok.

Vrijednosti *a=0* i *b=2* srušile su program.

1.11. Sad izbrišite specifikaciju `cg.enumerate_random = true` iz konfiguracijske datoteke te pokrenite aplikaciju. Što se sad dogodilo? Objasnite.

Kako se brisanjem navedene specifikacije ne gledaju sve mogućnosti generiranja slučajnih brojeva, verifikacija ne dojavljuje niti jednu pogrešku.

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 zatim putem `Racer_2.jpf`. Koje svojstvo je narušeno u slučaju `Racer_2.jpf`, a što piše pod error 1 u slučaju `Racer.jpf`?

U slučaju *Racer_2.jpf* narušeno je svojstvo neuhvaćenih iznimki, a u ovom slučaju neuhvaćena iznimka bila je dijeljenje s nulom:
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero

U slučaju *Racer.jpf* narušeno je svojstvo utrke, a pod *error 1* piše koji retci koda su uzrokovali utrku dretvi:
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer@18f.d
 main at Racer.main(Racer.java:35)
 "int c = 420 / racer.d; // (4)" READ: getfield Racer.d
 Thread-1 at Racer.run(Racer.java:26)
 "d = 0; // (2)" WRITE: putfield Racer.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 može doći, jer instanca dretve *t* i glavna dretva pristupaju istoj varijabli *d*, odnosno dolazi do utrke dretvi. Do dijeljenja s nulom može doći ako instanca dretve *t* izvede naredbu *d = 0* prije nego glavna dretva izvede naredbu *int c = 420 / racer.d.* To je ostvarivo zbog načina kako se dretve raspoređuju na izvođenje.

Instanca dretve *t* ne može pristupiti navedenoj liniji koda, jer je ona napisana u *main()* metodi, tako da joj može pristupiti samo glavna dretva. Instanca dretve *t* može pristupiti samo linijama koda koje su napisane u metodi *run()*.

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.

Detektor utrke za resursom najprije pronalazi objekte nad kojima više od jedne dretve rade čitanje i pisanje. Nakon

toga, izvršava dretve u različitim redoslijedima i provjerava ishode zajedničkog pristupanja tim objektima. *PreciseRaceDetector* nasljeđuje *PropertyListenerAdapter* i nadjačava njegove metode:

- *boolean check(Search search, VM vm)*
- *void reset()*
- *void choiceGeneratorSet(VM vm, ChoiceGenerator<?> newCG)*
- *void executeInstruction(VM vm, ThreadInfo ti, Instruction insnToExecute)*

2.4. Proučite kod primjera `NumericValueCheck.java` i konfiguracijsku datoteku `NumericValueCheck.jpf`. Zatim pokrenite aplikaciju. Koju grešku je dojavio JPF?

JPF je dojavio grešku da je vrijednost varijable *someVariable* izvan raspona:

```
gov.nasa.jpf.listener.NumericValueChecker
local variable someVariable out of range: 12345,000000 > 42,000000
        at NumericValueCheck.main(NumericValueCheck.java:29)
```

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 (na koje dijelove nekog razreda se može primijeniti)? Navedite i sintaksu tih provjera.

Slušač se može primijeniti na polja (*FieldCheck*) i varijable (*VarCheck*).

Prvi način pisanja sintakse:

```
range.vars = 1
range.1.var = NumericValueCheck.main(java.lang.String[]):someVariable
range.1.min = 0
range.1.max = 42
```

Drugi način pisanja sintakse:

```
range.vars = someVariable
range.someVariable.var = NumericValueCheck.main(java.lang.String[]):someVariable
range.someVariable.min = 0
range.someVariable.max = 42
```

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?

Razred *T1* bio je bolje pokriven od razreda *T2*. U razredu *T1* pokrivene su 3/3 metode, a u razredu *T2* 2/3 metode.

Slušač *CoverageAnalyzer* morao je implementirati sučelje *PublisherExtension* kako bi izmijenio izgled ispita izvještaja.

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

Metoda *<init>()* je pretpostavljeni konstruktor razreda. U *main()* metodi pozivani su pretpostavljeni konstruktori razreda *T1* i *T2*, što je u tablici označeno kao *<init>()* za svaki od tih 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?

U tablici piše da je analizator prošao kroz 3/9 linija koda, umjesto 3/8 kao što je navedeno u ovom zadatku. Analizator je preskočio linije koda u kojima se radi *assert*, odnosno pokrio je samo kroz linije koda stvaranja razreda i implicitni *return*.

Postavljanjem specifikacije *coverage.show_branches = false* isključili bi 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 @). Projekt možete naći u zip obliku na stranicama predmeta FMUOS, u materijalima za DZ2. Projekt raspakirajte u isti direktorij gdje se nalazi i projekt `jpf-core` (npr. NetBeansProjects).

U NetBeansu otvorite novi projekt: odaberite "File" -> "New Project" -> "Java" -> "Java Free-Form Project". Pod "Location" odaberite direktorij gdje se nalazi `jpf-aprop`. Ostale sve rubrike bi se trebale popuniti automatski. Nastavite dalje. U koraku "Source Package Folder" pod "Source level" izaberite JDK 1.8. Zatim odaberite "Finish". `jpf-aprop` bi se trebao naći 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}
```

Sada pokrenite skriptu `build.xml` koja se nalazi u korijenskom direktoriju projekta `jpf-aprop` (desni klik pa "Run Target" -> "build"). Prevođenje datoteka i izgradnja tri .jar datoteke trebalo bi proći bez pogrešaka.

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 međukodom razreda koji se kao primjeri provjeravaju uz pomoć `jpf-aprop`. Koje svojstvo pokazuje taj put u datoteci `jpf.properties`?

U datoteci *jpf.properties* projekta *jpf-aprop* nisu navedena standardna svojstva koja se provjeravaju prilikom pretrage. S obzirom da se u *build.xml* datoteci referencira datoteka *jpf.properties* projekta *jpf-core*, standardna svojstva koja će se provjeravati su ona navedena u toj datoteci. To su potpuni zastoj (*gov.nasa.jpf.vm.NotDeadlockedProperty*) i neuhvaćene iznimke (*gov.nasa.jpf.vm.NoUncaughtExceptionsProperty*).

Put do direktorija s međukodom razreda koji se kao primjeri provjeravaju uz pomoć *jpf-aprop* je *src/examples*. Svojstvo koje pokazuje taj put u datoteci *jpf.properties* 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` anotacijom i što to točno znači?

U kodu razreda *ConstViolation*, unutar *main()* metode kreira se instanca razreda *ConstViolation* te se nad njom poziva metoda

dontDoThis(). U metodi *dontDoThis()* poziva se metoda *foo()* koja postavlja vrijednost varijable *d* u 42.

Metoda *dontDoThis()* označena je anotacijom @Const, čime se provjerava anotacijsko svojstvo promjenjivosti objekta. Metode označene tom anotacijom i sve ostale metode koje pozivaju tu metodu ne smiju mijenjati vrijednosti varijabli razreda ili instance.

3.3. Pokrenite tu aplikaciju za verifikaciju. Navedite pogrešku koja je dojavljena.

Dojavljena je neuhvaćena iznimka *AssertionError*:
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: instance field write within const context: int ConstViolation.d
    at ConstViolation.foo(ConstViolation.java:15)
    at ConstViolation.dontDoThis(ConstViolation.java:11)
    at ConstViolation.main(ConstViolation.java:20)

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

Ovaj program koristi sljedeće slušače:
- *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 vrstu iznimke bacio JPF ako bi dretva bila uhvaćena u ciklusu nad objektom koji nije predviđen za višedretvenost.

Navedeno svojstvo provjerava se u slušaču *gov.nasa.jpf.aprop.listener.NonSharedChecker*.

Ako bi dretva bila uhvaćena u ciklusu nad objektom koji nije predviđen za višedretvenost, JPF bi bacio iznimku *java.lang.AssertionError* s pripadnom porukom.

3.6. Što znače anotacije @Requires, @Invariant i @Ensures u kodu programa ContractViolation.java? Koju vrstu dobrog oblikovanja programske potpore ostvaruju ove anotacije?

@Requires – uvjeti koji moraju biti ispunjeni prije izvršavanja metode (*preconditions*)

@Invariant – uvjeti koji moraju biti ispunjeni u svakom trenutku izvršavanja metode

@Ensures – uvjeti koji moraju biti ispunjeni nakon izvršavanja metode (*postconditions*)

Ove anotacije ostvaruju to da se u svakom trenutku izvršavanja metode zna koja su moguća stanja u kojima se varijable mogu naći.

3.7. Pokrenite aplikaciju za verifikaciju. Koja anotacija je narušena? Napišite pogrešku koja je dojavljena.

Narušena je anotacija *@Ensures*.

Dojavljena je neuhvaćena iznimka *AssertionError*:
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: postcondition violated: "(Result >= 0) AND (Result > 0)", values={Result=0}
at ContractViolation.getLoopCount(ContractViolation.java:29)
at ContractViolation.main(ContractViolation.java:42)

3.8. Pažljivo proučite dojavljenu pogrešku. Koja metoda kojeg točno razreda je izazvala narušavanje ugovora?

Metoda *getLoopCount()* razreda *ContractViolationBase* izazvala je narušavanje ugovora, jer je tamo postavljena anotacija *@Ensures(„Result > 0")*.

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?

Nema dojavljenih pogrešaka, svi ugovori su zadovoljeni.

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){}`).
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` iz paketa `fv` 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/fv`. Probajte pokrenuti verifikaciju koja bi trebala proći bez pogrešaka.
Objasnite zašto je redak `JavaFV.classpath=\`
`${JavaFV}/build/classes`
nužno navesti u datoteci `jpf.properties`?

Da bi JPF mogao učitati međukod, potrebno je navesti njegovu putanju u datoteci *jpf.properties*.

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 DZ2). 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 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. Ostale knjižnice nisu bitne. Od projekta `jpf-aprop` potrebno je dodati samo knjižnicu `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.

Dojavljena je neuhvaćena iznimka *AssertionError:*

```
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: null assignment to @Nonnull instance field: fv.Verifikacija.id
        at fv.Verifikacija.setId(fv/Verifikacija.java:30)
        at fv.Verifikacija.main(fv/Verifikacija.java:26)
```

Do navedene pogreške došlo je zbog pokušaja postavljanja vrijednosti varijable *id* instance razreda *Verifikacija* u *null*. U konfiguracijskoj datoteci imamo postavljen slušač *gov.nasa.jpf.aprop.listener.NonnullChecker* koji prati je li se nekoj varijabli označenoj s anotacijom *@Nonnull* pokušala dodijeliti vrijednost *null*. U kodu ove aplikacije ispred definirane varijable *id* stavljena je anotacija *@Nonnull*.

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?

Za razliku od prethodnog ispisa u kojem je dojavljena samo jedna neuhvaćena iznimka *AssertionError*, u ovom ispisu dojavljene su tri neuhvaćene iznimke *AssertionError*:

```
========================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: null assignment to @Nonnull instance field: fv.Verifikacija.id
        at fv.Verifikacija.setId(fv/Verifikacija.java:30)
        at fv.Verifikacija.main(fv/Verifikacija.java:26)

========================================================== error 2
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: null assignment to @Nonnull instance field: fv.Verifikacija.id
        at gov.nasa.jpf.vm.Verify.getDouble(gov.nasa.jpf.vm.JPF_gov_nasa_jpf_vm_Verify)
        at fv.Verifikacija.main(fv/Verifikacija.java:21)
```

== error 3
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: null assignment to @Nonnull instance field: fv.Verifikacija.id
	at gov.nasa.jpf.vm.Verify.getDouble(gov.nasa.jpf.vm.JPF_gov_nasa_jpf_vm_Verify)
	at fv.Verifikacija.main(fv/Verifikacija.java:21)

4.6. Uklonite redak `search.multiple_errors = true` te zakomentirajte redak u kodu koji smatrate odgovornim za dojavu pogreške iz zadatka 4.4. Prevedite kod i pokrenite ponovno aplikaciju za verifikaciju. Koja se sada pogreška pojavila, na kojem retku koda i zašto je prijavljena?

Dojavljena je neuhvaćena iznimka *AssertionError:*
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError: write of const instance field outside constructor: java.lang.String fv.Verifikacija.DNA
	at fv.Verifikacija.setDNA(fv/Verifikacija.java:33)
	at fv.Verifikacija.main(fv/Verifikacija.java:27)

Do navedene pogreške došlo je u 27. retku koda zbog pokušaja promjene vrijednosti varijable *DNA* instance razreda *Verifikacija*. U konfiguracijskoj datoteci imamo postavljen slušač *gov.nasa.jpf.aprop.listener.ConstChecker* koji prati je li se nekoj varijabli označenoj s anotacijom *@Const* pokušala promijeniti vrijednost. U kodu ove aplikacije ispred definirane varijable *DNA* stavljena je anotacija *@Const*.

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.

Dojavljena je neuhvaćena iznimka *AssertionError:*
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError
	at fv.Verifikacija.main(fv/Verifikacija.java:23)

U konfiguracijskoj datoteci definirali smo *velocity* kao *gov.nasa.jpf.vm.choice.DoubleThresholdGenerator* i odredili njenu najnižu (20.0) i najvišu (100.0) vrijednost. U kodu smo dohvatili najvišu vrijednost i spremili u varijablu *vel* te pomoću *assert* ispitali tvrdnju *vel < 100.0*. Kako je vrijednost varijable *vel* jednaka 100.0, *assert* vraća da tvrdnja nije istinita, što dovodi do navedene iznimke *AssertionError*.

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?

U naredbi *assert (vel < 100.0)* umjesto broja 100 može se staviti bilo koji broj veći od njega da bi tvrdnja bila istinita. Za npr. broj 101 tvrdnja je istinita i ne dolazi do pogreške.

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`?

U razredu *gov.nasa.jpf.vm.Verify* ne postoje metode *getDouble()* i *getInt()* bez argumenata. Arugmenti su potrebni za raspon ili za ključ svojstva definiranog u konfiguacijskoj datoteci.

Generatori izbora (engl. *ChoiceGenerators*) predstavljaju mehanizam JPF-a koji sustavno istražuje prostor stanja kako bi došao do rješenja. Heuristike se navode s ciljem smanjenja broja mogućnosti dodjele vrijednosti pri provjeri tipova varijabli kao što su *int*, *double* i drugi tipovi varijabli. Primjenom heuristike ne ispituju se svi putevi kroz program, nego samo oni koje heuristika smatra zanimljivima.

U primjeru *Verifikacija.java* koristila se heuristika *DoubleTresholdGenerator*.