UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milica Kleut

APSTRAKTNA INTERPRETACIJA U VERIFIKACIJI SOFTVERA — ELEKTRONSKA LEKCIJA

master rad

Beograd, 2025.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Jovana Kovačević, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

mast. mat. Milan Čugurović, asistent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: oktobar, 2025.

mom prvom profesoru matematike - mom dedi,
mojoj porodici

Naslov master rada: Apstraktna interpretacija u verifikaciji softvera — elektronska lekcija

Rezime: U ovom radu su objašnjeni osnovni koncepti apstraktne interpretacije u verifikaciji softvera kroz elektronsku lekciju koja sadrži teorijski deo i interaktivni sadržaj. Teorijskim delom se daju osnove apstraktne interpretacije. Zatim se stečeno teorijsko znanje može dodatno utvrditi na konkretnim *Python* primerima koji se pokreću interaktivno u *Jupyter* svesci. Rad sadrži dve implemeatacije metoda apstraktne interpretacije. Prva implementacija demonstrira, na problemu određivanja opsega promenljivih, rano otkrivanje potencijalnih numeričkih grešaka (npr. celobrojnih prekoračenja) i objašnjava uticaj izbora apstraktnog domena na preciznost i skalabilnost. Druga implementacija demonstrira, na problemima koji se javljaju u konkurentnim programima, obrasce pristupa deljenim podacima, kritične sekcije i efekte sinhronizacije. Na taj način se identifikuju tačke mogućih trka za resursima i razgraničava kada sinhronizacija uklanja rizik. Primeri ilustruju kako izabrane apstrakcije utiču na ishod verifikacije, prikazujući i prednosti i mane ove metode.

Ključne reči: verifikacija softvera, statička analiza, apstraktna interpretacija, elektronska lekcija, apstrakcija

Sadržaj

1 Uvod **1**

2 Izbor tehnologija i alata **3**

2.1 Jupyter sveska 3

2.2 Biblioteka *NumPy* 6

2.3 Uputstvo za pokretanje elektornske lekcije 7

3 Teorijska osnova za apstraktnu interpetaciju **10**

3.1 Statička verifikacija 11

3.2 Apstraktna interpretacija 14

3.3 Uređenje skupova i Galoaove veze 31

3.4 Zabranjene zone i lažna upozorenja 40

4 Određivanje opsega promenljivih **44**

4.1 Apstrakcija intervalima za proveru prekoračenja 45

4.2 Apstrakcija intervalima za verifikaciju bankarskog softvera 55

5 Otkrivanje trke za resursima **61**

5.1 Trka za resursima i sinhronizacija 61

5.2 Primeri verifikacije jednostavnih konkurentnih programa 65

6 Zaključak **74**

Literatura **76**

Glava 1

Uvod

U svakodnevnom životu koristimo veliki broj različitih aplikacija. Teško je zamisliti dan bez pametnog telefona, sata ili automobila sa raznim sigurnosnim sistemima i navigacijama. Pored toga sve više se koriste pametni kućni uređaji i pametne kuće, veštačka inteligencija i roboti koji se mogu ponašati slično kao ljudi. Gotovo da više ne postoji delatnost u kojoj nisu prisutni računarski sistemi. Cilj svake kompanije koja proizvodi softver je da obezbedi ponašanje softvera u skladu sa očekivanjima korisnika kako bi korisnici želeli da ga koriste i kako bi se osećali sigurno.

U razvoju softvera kreće se od specifikacije — šta softver treba da radi i na koje zahteve treba da odgovori. Zatim tim programera implementira sam softver i zajedno sa testerima se brine za njegovu ispravnost. Provera kvaliteta softvera i njegovog slaganja sa specifikacijom je deo procesa razvoja softvera koji se naziva *verifikacija softvera*. Ona ima za cilj da utvrdi da li se softver ponaša na predviđen način, da li daje očekivane odgovore, da li postoje anomalije u ponašanju, odnosno da li dati softver zadovoljava predviđenu specifikaciju. Ovo je bitan deo razvoja softvera jer i najmanja greška može naneti velike štete tako da se ne sme dopustiti da softver ima propuste.

U zavisnosti od vrste softvera koji se razvija, korisnici softvera mogu biti nezadovoljni izgledom i funkcionalnostima softvera što uzrokuje gubitak poverenja korisnika čime se smanjuje potražnja pa i profit za proizvod koji se nudi. Takođe, može doći do materijalnih gubitaka poput gubitka podataka. U najgorem slučaju, može doći do smrtnog ishoda što je slučaj koji može da se desi u auto i avio industriji, zdravstvu ili nuklearnim elektranama.

Na primer, jedna od najskupljih softverskih grešaka u istoriji svemirskih misija je pad rakete Evropske svemirske agencije (ESA) *Ariane 5*. Tokom svog prvog leta,

raketa se samouništila svega 37 sekundi nakon poletanja 4. juna 1996. Tom prilikom izgubljena su četiri satelita predviđena za istraživanje Zemljine magnetosfere. Procena štete je bila preko 370 miliona dolara. Uzrok je bio propust u softveru: pokušaj pretvaranja (kastovanja) 64-bitne vrednosti brzine u 16-bitni tip izazvao je numeričko prekoračenje i izuzetak koji nije bio pravilno obrađen. Posledica su bile pogrešne komande, skretanje sa planirane putanje i aktivacija sistema za samouništenje [8].

Kako se kroz vreme sve više okrećemo digitalizaciji i prepuštamo računarima da urade veliki deo posla, programi postaju sve veći. Softverska industrija ima izazov da što bolje verifikuje velike programe. Kada je reč o *velikim* programima to podrazumeva programe sa velikim brojem linija koda (čak do nekoliko desetina miliona linija) što dalje povlači ogroman broj potencijalnih ulaznih vrednosti i ogroman broj putanja izvršavanja programa. Sve ovo otežava nalaženje mogućih grešaka.

Cilj ovog rada je prenošenje znanja o *apstraktnoj interpretaciji*, naprednoj tehnici verifikacije softvera, koja se može primenjivati na programe sa velikim brojem linija koda. Ideja je da se razvije interaktivni alat kako bi čitalac mogao da pokreće algoritme apstraktne interpretacije i sam proveri kako oni rade. Korišćenje interaktivnog okruženja treba da pomogne uvođenje čitaoca u osnove apstraktne interpretacije, da mu predstavi osnovne algoritme koji se koriste u ovoj oblasti, da na interaktivan način pomogne čitaocu da usvoji i razume koncepte apstrakte interpretacije i da pokuša da objasni važnost verifikacije softvera. Takođe, još jedan od ciljeva ovog rada je da podstakne čitaoca na razmišljanje o novim idejama verifikacije softvera i popularizacija ove grane IT[1] industrije kao grane koja je neophodna za proizvodnju kvalitetnog softvera.

U okviru rada razvijena je elektronska lekcija korišćenjem *Jupyter* sveske koja je centralni deo poglavlja 2. Glavnom pojmu ovog rada približićemo se kroz teorijsku osnovu i matematičku potporu koji su deo poglavlja 3. Pored sticanja osnovnih teorijskih znanja potrebno je steći i intuiciju koja je data kroz razne primere u okviru istog tog poglavlja. Centralni deo rada sadrži opis implementacija koje se oslanjaju na apstraktnu interpretaciju. U ovom radu su date dve različite implementacije i one su detaljno objašnjene u okviru poglavlja 4 i 5. Poslednji deo rada, poglavlje 6, sadrži zaključke, doprinose i sumirane rezultate.

[1] IT — informacione tehnologije; eng. *information technology*

Glava 2

Izbor tehnologija i alata

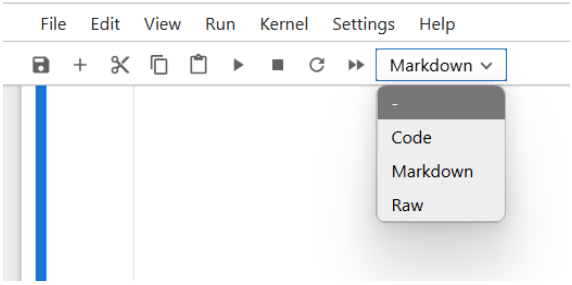
U izradi ovog rada korišćena je *Jupyter sveska* (eng. *Jupyter Notebook*), savremen alat koji omogućava brzu i efikasnu implementaciju programskih rešenja, kao i laku analizu rezultata. Ovaj alat nudi interaktivno okruženje je pogodno je za prenošenje znanja jer omogućava kombinovanje koda, teksta, matematičkih formula, vizuelizacija i multimedijalnih sadržaja u okviru jednog dokumenta.

2.1 Jupyter sveska

Jupyter sveska predstavlja interaktivni veb-zasnovan alat koji omogućava korisnicima da pišu i izvršavaju kôd direktno u veb pregledaču. Glavni delovi *Jupyter* sveske su:

- Alatna traka (eng. *toolbar*) i meni - omogućavaju pristup opcijama za rad sa sveskom, kao što su čuvanje, otvaranje, promena ćelija, pokretanje koda, itd.
- Ćelije (eng. *cells*) - sadrže delove sveske koji se svrstavaju u različite tipove ćelija u zavisnost od sadržaja. Postoje tri tipa ćelija (slika 2.1):
  - *Code*: služe za pisanje i izvršavanje koda.
  - *Markdown*: koriste se za formatirani tekst, kao što su naslovi, liste, slike, matematičke formule itd.
  - *Raw*: sirovi tekst koji se ne interpretira niti izvršava.

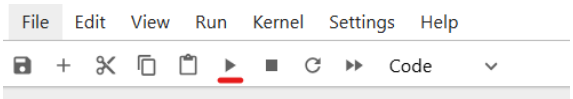
  Ćelija može biti u režimu unosa (eng. *edit mode*) ili u komandnom režimu (eng. *command mode*). Klikom/dvoklikom unutar ćelije prelazi se u režim unosa.



Slika 2.1: Tipovi ćelija

Klikom izvan ćelije ili pritiskom tastera **Esc** prelazi se u komandni režim. Režim unosa dozvoljava izmene unutar selektovane ćelije. Komandni režim dozvoljava menjanje sveske kao jedne celine ali ne i menjanje pojedinačnih ćelija.

Za pokretanje koda iz ćelije dovoljno je izabrati ćeliju i pritisnuti tastersku prečicu **Ctrl** + **Enter** ili kliknuti na dugme **Run** (slika 2.2).



Slika 2.2: Pokretanje koda iz ćelije

Rezultat programa će se videti u redu ispod ćelije. Program čiji su delovi razdvojeni po ćelijama može se pokrenuti tako što se pokrene svaka ćelija zasebno ili tako sto se pokrene izvršavanje koda iz svih ćelija odjednom (klik na opciju **Run** u meniju a zatim **Run All Cells** — slika 2.3).

- Jezgro (eng. *kernel*) — izvršava kôd unutar sveske i obezbeđuje podršku za konkretan programski jezik. Prilikom kreiranja nove sveske, jezgro se bira iz liste dostupnih jezika, koja zavisi od prethodno instaliranih paketa. Na primer, ovo su mogući paketi za neke programske jezike:

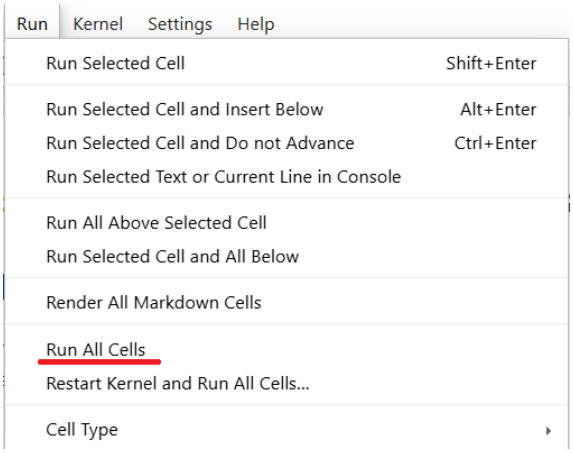
  – programski jezik Python: paket `ipykernel`

  – programski jezik R: paket `IRkernel`

  – programski jezik Julia: paket `IJulia`

  Paketi se mogu instalirati iz terminala pomoću komande *pip install*.

  Primer instalacije paketa `ipykernel`:



Slika 2.3: Pokretanje koda iz svih ćelija

```
pip install ipykernel
```

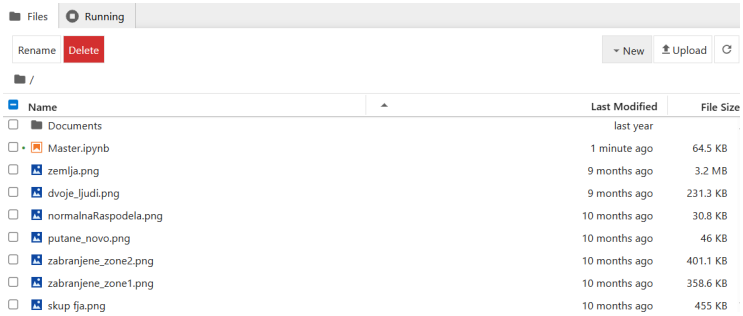
Nakon instalacije jezgro će biti dostupno u *Jupyter* interfejsu.

- Pregledač fajlova (eng. *file browser*) omogućava pregled i upravljanje svim fajlovima i direktorijumima u projektu. Prilikom pokretanja *Jupyter* sveske prva stranica koja se otvori u veb pregledaču je zapravo pregledač fajlova (slika 2.4). Klikom na stavku iz liste fajlova i direktorijuma prikazuju se opcije za otvaranje, preuzimanje, preimenovanje, dupliranje i brisanje.

  Sa desne strane gornjeg dela interfejsa nalaze se dugmad **New** za kreiranje novih svezaka, tekstualnih fajlova ili foldera, i **Upload** za učitavanje fajlova sa lokalnog računara.

  Ovaj deo interfejsa omogućava jednostavno upravljanje projektima i fajlovima što doprinosi većoj pristupačnosti i efikasnosti rada.

Jupyter sveske se ističu svojom fleksibilnošću i podrškom za više programskih jezika, među kojima je i *Python*, koji je korišćen u ovom radu. Interaktivna priroda alata omogućava korisnicima da pokreću delove koda, analiziraju rezultate u realnom vremenu, i na taj način unapređuju i prilagođavaju svoje programe. Ova mogućnost olakšava rad na složenim projektima, jer omogućava brzu iteraciju i testiranje različitih pristupa. Kako je cilj ovog rada prenošenje znanja preporučuje se da čitalac iskoristi interakivnost alata i isproba napisane kodove sa različitim ulaznim vrednostima.



Slika 2.4: Pregledač fajlova

2.2 Biblioteka *NumPy*

U programskom jeziku *Python* jedna od najvažnijih biblioteka za naučne i numeričke proračune je biblioteka *numpy*. Naziv *numpy* označava skraćenicu od *numerical Python* (što znači *numerički Python*). Biblioteka se najčešće koristi za efikasan rad sa velikim numeričkim podacima (višedimenzionalnim nizovima) i izvođenje matematičkih, statističkih i algebarskih operacija. *NumPy* podržava širok spektar matematičkih funkcija, uključujući trigonometrijske, logaritamske i eksponencijalne funkcije, funkcije za zaokruživanje, generisanje nasumičnih brojeva, hiperboličke funkcije, sumiranje, diferenciranje, aritmetičke operacije, rad sa kompleksnim brojevima.

U programskom jeziku *Python* za rad sa kolekcijama podataka često se koriste liste, koje mogu služiti i kao nizovi, ali su relativno spore u obradi većih količina numeričkih podataka. Biblioteka *NumPy* omogućava efikasno rukovanje nizovima putem objekata tipa *ndarray*, koji predstavljaju višedimenzionalne numeričke nizove. Za rad sa ovim objektima dostupne su brojne optimizovane funkcije unutar same biblioteke. Nizovi se široko primenjuju u oblasti nauke o podacima (eng. *data science*), gde su brzina izvršavanja i efikasno korišćenje memorijskih resursa od ključnog značaja. Za razliku od listi, *NumPy* nizovi zauzimaju uzastopne memorijske lokacije, što omogućava brži pristup i manipulaciju podacima. Dodatno, *NumPy* je posebno optimizovan za savremene CPU arhitekture, što dodatno doprinosi njegovoj visokoj performansi.

2.3 Uputstvo za pokretanje elektornske lekcije

Jupyter sveska koja je korišćena u ovom radu dostupna je na github linku: *https://github.com/kleutMilica/MasterRad/blob/main/Master.ipynb*.
Pre svakog primera koda dat je kratak teorijski uvod sa najvažnijim konceptima neophodnim za njegovo razumevanje. Primer kako izgleda teorijski deo elektronske lekcije prikazan je na slici 2.5.

Slika 2.5: Primer teorijskog dela elektronske lekcije

Interaktivni deo čine sami kodovi, koji se mogu izvršavati sa različitim početnim uslovima. U svakom delu sveske opisano je ponašanje koda pri različitim ulazima, a nakon izvršavanja prikazuje se odgovarajući izlaz programa. Na slikama 2.6 i 2.7 dati su primeri interaktivnog dela elektronske lekcije.

Kôd je napisan u programskom jeziku *Python*, te je preduslov za pokretanje sveske instalacija jezika *Python* kao i instalacija *ipykernel*-a koji povezuje *Jupyter* svesku sa *Python* interpreterom, omogućavajući da se kôd iz ćelija u svesci šalje *Python*-u na izvršavanje i dobijene rezultate vraća nazad. Za instalaciju same *Jupyter* sveske koristi se *pip* komanda:

```
pip install jupyter
```

Da bi se sveska otvorila u pretraživaču potrebno je u komandnoj liniji uneti:

- za *Linux* operativni sistem

File Edit View Run Kernel Settings Help Trusted

Markdown JupyterLab Python 3 (ipykernel)

Kôd koji implementira proveru izabranog apstraktnog intervala

[7]:
```
class AbstractDomain:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def check_overflow(self):
        max_a = max(self.a)
        max_b = max(self.b)
        max_c = max(self.c)

        product = max_a * max_b

        if product > 32767:
            return True

        if product + max_c > 32767:
            return True

        return False

def main():
    a_domain = range(1, 101) # [1, 100]
    b_domain = range(1, 11)  # [1, 10]
    c_domain = range(1, 32768) # [1, 32767]

    abstract_domain = AbstractDomain(a_domain, b_domain, c_domain)

    if abstract_domain.check_overflow():
        print("Upozorenje: Vrednosti mogu uzrokovati prekoracenje za 16-obitne cele brojeve!")
    else:
        print("Vrednosti su bezbedne za 16-obitne cele brojeve.")
```

Slika 2.6: Primer interaktivnog dela elektronske lekcije — provera izabranog apstraktnog intervala

File Edit View Run Kernel Settings Help Trusted

Markdown JupyterLab Python 3 (ipykernel)

Kôd koji prikazuje korišćenje mehanizma Lock u Python-u

[10]:
```
import threading

x = 0  # zajednicka promenljiva
lock = threading.Lock()

def increment():
    global x
    for _ in range(100000):
        with lock:   # samo jedna nit u kriticnom delu
            x += 1

t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=increment)

t1.start()
t2.start()
t1.join()
t2.join()

print("Konacna vrednost:", x)
```

```
Konacna vrednost: 200000
```

U ovom primeru, zahvaljujući primeni zaključavanja, niti ne mogu istovremeno pristupiti i modifikovati promenljivu x. Konačna vrednost je uvek ispravna i jednaka 200000, jer je eliminisana mogućnost trke za resursima.

Primeri verifikacije jednostavnih konkurentnih programa

Implementacija apstraktne interpretacije za konkurentne programe je korisna za detekciju problema u konkurentnom programiranju. Primeri koji slede demonstriraju kako se apstraktna interpretacija može koristiti za identifikaciju potencijalnih trka za resursima. Analiza prati pristupe deljenim promenljivama, kao i pristupe kritičnim delovima koda.

Slika 2.7: Primer interaktivnog dela elektronske lekcije — korišćenje mehanizma `Lock` u Python-u

```
jupyter notebook
```

- za *Winodws* operativni sistem

```
python -m notebook
```

Ukoliko se kôd pokreće u *Jupyter* svesci za proveru da li je već instalirana *numpy* biblioteka koristi se komanda:

```
!pip show numpy
```

Ova komanda će prikazati informacije o instalaciji *numpy* paketa. Ako se kao izlaz komande ispisuju informacije o *numpy*-u, to znači da je instaliran. Ukoliko biblioteka nije instalirana, može se instalirati direktno iz *Jupyter* sveske komandom:

```
!pip install numpy
```

Nakon instalacije *numpy-a* treba restartovati *Jupyter* kernel da bi se osvežilo okruženje. To se može uraditi iz menija „`Kernel`” > „`Restart Kernel...`”. Nakon toga je biblioteka spremna za korišćenje.

Glava 3

Teorijska osnova za apstraktnu interpetaciju

Verifikacija softvera može biti dinamička i statička. Tehnike dinamičke verifikacije softvera utvrđuju ispravnost softvera izvršavajući ga i proveravajući da li se izlaz poklapa sa onim što se očekuje na osnovu specifikacije. Najčešći vid dinamičke verifikacije softvera je testiranje. Testiranje se često koristi kao sinonim za verifikaciju softvera. Testiranje je tehnika u kojoj se izvršava program sa namerom da se pronađe što više mogućih defekata ili da se stekne dovoljno poverenja u sistem koji se testira. Pravilnim i sistematičnim testiranjem se podiže nivo pouzdanosti i smanjuje se verovatnoća postojanja greške [8]. Dinamička verifikacija na velikim projektima iziskuje nezanemarljivu količinu resursa.

Sa druge strane, statička verifikacija podrazumeva proveru ispravnosti programa bez njegovog izvršavanja, odnosno analizira se izvorni kôd programa. Statička verifikacija može da koristi ručne provere i preglede koda ili formale metode.

Formalne metode opisuju kôd na jeziku neke matematičke teorije i koriste se sa idejom da se ceo proces automatizuje. To nije u potpunosti moguće zbog fundamentalnih ograničenja. Zarad automatizacije potrebno je odreći se potpune preciznosti, a ono što se postiže je da se u konačnom vremenu, koristeći konačne resurse, dobijaju važne informacije o ispravnosti programa. Jedna metoda iz ove grupe je apstraktna interpretacija koja se oslanja na teoriju aproksimacije.

3.1 Statička verifikacija

Statička analiza omogućava otkrivanje i rešavanje grešaka u kodu u ranim fazama razvoja softvera. Postoje različite vrste statičke verifikacije koje se mogu podeliti u dve osnovne grupe [8]:

Ručne provere i pregledi koda — svakodnevno se primenjuju u okviru procesa razvoja softvera.

Formalne metode verifikacije softvera — uslovi ispravnosti softvera iskazuju se u terminima matematičkih tvrđenja na striktno definisanom formalnom jeziku izabrane matematičke teorije.

U nastavku je dat kratak pregled statičkih metoda (sem apstraktne interpretacije, koja je detaljno opisana u poglavlju 3.2) kako bi se stekao uvid u to šta se sve može otkriti, zaključiti i popraviti ovim metodama. Pređenje metoda sumirano je u tabeli 3.1. Posmatranjem tabele mogu se lako uporediti metode po različitim kriterijumima i uočiti prednosti i mane za svaku od njih. Ovakva vrsta poređenja korisna je pri odlučivanju meotde koja treba da se koristi za konkretan problem.

Pregled koda

Pregled koda (eng. *code review*) predstavlja jednu od najstarijih tehnika statičke analize softvera. Ova metoda se zasniva na sistematskom, pažljivom pregledu izvornog koda od strane jednog ili više stručnjaka sa ciljem identifikacije grešaka, sigurnosnih propusta, nedoslednosti, kršenja stilskih pravila, kao i potencijalnih problema u vezi sa logikom programa. Nakon pregleda, kôd treba da se uklapa u programerske konvencije, da bude kvalitetan i čitljiv svim programerima na projektu.

Za razliku od automatskih alata za statičku analizu, pregled koda se oslanja na ljudsko znanje i iskustvo, što omogućava da detektuje kompleksne i kontekstualno zavisne greške koje često izmiču automatizovanim tehnikama. Pregled koda se može obavljati neformalno (eng. *ad-hoc*) ili kroz formalizovane procese kao što su pregled od strane kolega (eng. *peer review*), vođeni pregled (eng. *walkthrough*) i inspekcija (eng. *inspection*), pri čemu svaki od njih podrazumeva različit stepen strukturiranosti i formalnosti.

Postoje i brojni alati za pregled koda koji ubrzavaju ovaj proces. Takođe, brojna razvojna okruženja daju podršku za automatsko prepoznavanje grešaka u stilu, konvencijama i čitljivosti.

Kriterijum	**Pregled koda**	**Simboličko izvršavanje**	**Proveravanje modela**
Vrsta metode	Statička, neformalna/ formalizovana	Statička, formalna	Statička, formalna
Automatizacija	Niska	Visoka	Visoka
Pokrivenost koda	Delimična	Visoka (sve putanje do dubine)	Potpuna nad modelom
Skalabilnost	Dobra za male sisteme	Ograničena (eksplozija putanja)	Ograničena (eksplozija stanja)
Formalni model potreban	Ne	Ne	Da
Otkrivanje semantičkih grešaka	Visoko	Visoko	Srednje (zavisi od modela)
Generisanje kontra-primera	Ne	Da (test ulazi)	Da (sled stanja)
Lakoća implementacije	Jednostavna	Umereno složena	Složena
Primena u praksi	Kritične revizije, sigurnost	Sigurnosna analiza, automatsko testiranje	Reaktivni i vremenski zavisni sistemi
Prednosti	Fleksibilna, ljudski uvid	Visoka preciznost, automatski testovi	Potpuna pretraga modela, kontra-primeri
Mane	Subjektivna, sklona previdima	Eksplozija putanja, složena konfiguracija	Eksplozija stanja, potreba za modelovanjem

Tabela 3.1: Poređenje tehnika statičke verifikacije softvera

Simboličko izvršavanje

Simboličko izvršavanje (eng. *symbolic execution*) je tehnika statičke analize programa koja se koristi radi sistematskog istraživanja mogućih izvršavanja nekog softverskog sistema. Podrazumeva praćenje simboličkih vrednosti umesto konkretnih vrednosti ulaza i prati evaluaciju izraza kao formule nad simbolima, stvarajući tzv. putanje izvršavanja (eng. *execution paths*).

Kroz program se prate simbolička stanja izvršavanja umesto konkretnih stanja. Kada god se naiđe na uslove i grananja gde postoji mogućnost daljeg izvršavanja programa na više načina, prate se sve moguće nove putanje, daodavanjem u nove putanje izvršavanja odgovarajuće različite uslove nad simboličkim vrednostima. Deo softverskog alata koji sprovodi samo simboličko izvršavanje se naziva analizator: održava simbolička stanja i uslove putanje, grana putanje na uslovima, i proverava zadovoljivost uslova. Na primer, za uslov if $(x > 5)$, analizator generiše putanju koji odgovara istinitosti i lažnosti uslova, zajedno sa tekućim uslovima putanje (eng. *path conditions*) — logičkim formulama koje moraju važiti da bi se ta putanja mogala izvršiti. Na ovaj način, simboličko izvršavanje obuhvata izvršavanja više putanja programa, potencijalno sve, i otkriva greške koje zavise od određenih kombinacija ulaznih vrednosti.

Simboličko izvršavanje ima za cilj automatsko generisanje test primera i automatsko pronalaženje grešaka u programu. Iako se čini da bi se ovakvim pristupom mogle generisati sve moguće putanje kroz program to se u praksi ne dešava. Postoje različiti problemi koji se javljaju u ovom kontekstu, a neki od njih su problemi sa memorijom i okruženjem, potencijalno eksponencijalan rast broja putanja, nedostupnost izvornog koda i slično.

Proveravanje modela

Proveravanje modela (eng. *model checking*) predstavlja metod za automatsku verifikaciju ispunjenosti formalno definisanih svojstava nad modelom sistema. Proces podrazumeva izgradnju preciznog formalnog modela sistema, kao i specifikaciju svojstava čiju ispunjenost treba proveriti. Model se najčešće zasniva na matematičkoj logici, teoriji formalnih jezika i teoriji grafova, te se opisuje strogo formalno — najčešće u obliku konačnog automata, sistema prelaza ili Kripkeove strukture. Svojstva se izražavaju u nekom formalnom logičkom jeziku, najčešće u okviru temporalnih logika, kao što su linearna temporalna logika (eng. *Linear Temporal Logic*,

LTL) i logika stabla izračunavanja (eng. *Computation Tree Logic*, CTL).[1]

Proveravač modela sistematski pretražuje prostor stanja (eng. *state space*) i ispituje da li zadato svojstvo važi za sva moguća izvršavanja. U slučaju da se pronađe niz stanja koji krši posmatrano svojstvo, generiše se kontra-primer (eng. *counterexample*) — konkretna sekvenca prelaza koja ilustruje grešku u sistemu. Ova metoda nalazi primenu kako u analizi softverskih i hardverskih sistema, tako i u modeliranju konkurentnih sistema, komunikacionih protokola i drugih kritičnih oblasti.

3.2 Apstraktna interpretacija

Apstraktna interpretacija je tehnika koja se koristi u verifikaciji softvera radi analize programa i otkrivanja grešaka. Ona omogućava statičku analizu programa na visokom nivou apstrakcije, što znači da se ne uzimaju u obzir sve pojedinosti izvornog koda, već se vrši aproksimacija ponašanja programa. Apstraktno posmatranje softvera je intuitivno i blisko ljudima.

Intuicija u apstraktnoj interpretaciji

Ideja apstrakcije datira još od samih početaka ljudskog mišljenja. U situacijama koje obiluju detaljima, često dolazi do generalizacije kroz izostavljanje manje relevantnih informacija. U nastavku razmatramo nekoliko primera generalizacije, tj. apstrakcije.

Primer 1. Ljudskom mozgu je svojstveno ne samo da spontano vrši aproksimaciju i apstrakciju opaženih pojava, već i da prepoznaje aproksimacije i apstrakcije u spoljnjem svetu. Na slici 3.1 jasno se može uočiti da su prikazane dve osobe, uprkos tome što su predstavljene isključivo pomoću stilizovanih obrisa. Štaviše, ovakve ilustracije često omogućavaju bogatiju interpretaciju: moguće je naslutiti međusobni odnos prikazanih figura, njihovo emocionalno stanje, pa čak i pretpostaviti njihove fizičke karakteristike — pol, telesnu građu, izraženost pojedinih crta lica (npr. brade, jagodica, nosa) i drugo. Ovakva analiza vizuelnih elemenata doprinosi formiranju celovite predstave o subjektima prikazanim na slici. Na osnovu konkretne ilustracije, može se zaključiti da se radi o dve zabrinute osobe, verovatno muškog i ženskog pola, koje su u bliskom međusobnom odnosu.

[1] **LTL** opisuje ponašanje duž linearnih nizova stanja (izvršavanja), dok **CTL** omogućava izražavanje svojstava nad stablima izvršavanja, u kojima se razmatraju svi mogući putevi koji proizlaze iz datog stanja.

Slika 3.1: Apstraktna slika dvoje ljudi

Primer 2. Ukoliko se na nekom događaju pojavilo 1156 ljudi, može se reći da je prisustvovalo više od 1000 osoba. U tom slučaju dolazi do gubitka informacije o preostalih 156 učesnika; međutim, takav gubitak često nije od suštinskog značaja. Time se vrši aproksimacija stvarnog broja prisutnih.

Međutim, ukoliko bi se za isti događaj navelo da je prisustvovalo više od 100 osoba, aproksimacija bi, iako tehnički tačna, bila neinformativna, jer ne pruža realističan uvid u veličinu događaja i time gubi svoju svrhu. Nasuprot tome, tvrdnja da je događaju prisustvovalo više od 1200 osoba predstavlja aproksimaciju koja je bliža stvarnoj vrednosti, ali nije tačna. Ovi primeri ilustruju značaj balansa između tačnosti i informativnosti prilikom vršenja aproksimacije.

Primer 3. U određenim kontekstima, planeta Zemlja se aproksimira kao sfera, iako je poznato da njen stvarni oblik odgovara elipsoidu, odnosno geoidu (slika 3.2). Ova aproksimacija se koristi u astronomskim i geografskim istraživanjima, kao i u obrazovne svrhe, kada pojednostavljenje ne dovodi do značajnog gubitka u tačnosti informacija.

Na primer, u proračunima kretanja satelita ili prostiranja satelitskih signala, Zemlja se često modeluje kao sfera. Takva aproksimacija pojednostavljuje matematičke

modele i obezbeđuje dovoljan stepen preciznosti za određene astronomske primene.

Međutim, u situacijama koje zahtevaju visoku preciznost — kao što su merenje nadmorskih visina ili izrada detaljnih topografskih mapa — ovakvo pojednostavljenje nije prihvatljivo. Pretpostavka da je Zemlja sfera implicira da su sve tačke na njenoj površini jednako udaljene od središta, što ne odgovara stvarnosti i dovodi do značajnog gubitka informacija.

Slika 3.2: Zemlja predstavljena kao geoid, elipsoid i sfera

Bitno je proceniti koja aproksimacija sa sobom ne nosi gubitak bitnih informacija.

Primer 4. Neka je u proizvoljnom programu data promenljiva x za koju važi da može sadržati vrednosti prvih nekoliko celih pozitivnih brojeva.

Precizna aproksimacija podrazumeva dodeljivanje promenljivoj x intervala vrednosti, na primer od 1 do 10. Takav interval se potom prati tokom izvršavanja programa. Ovakav pristup omogućava detaljnije sagledavanje mogućih vrednosti promenljive i olakšava detekciju grešaka koje bi grublje aproksimacije mogle da propuste. Međutim, ova vrsta apstrakcije može da bude računarski zahtevna, naročito u kontekstu većih programa sa velikim brojem promenljivih, jer zahteva više memorijskih i procesorskih resursa radi održavanja i propagacije intervalskih informacija.

Manje precizan pristup podrazumeva viši nivo apstrakcije — umesto praćenja konkretnih intervala, vrednost promenljive x se apstrahuje na osnovu znaka. U konkretnom slučaju, to znači da se x posmatra samo kao pozitivna vrednost, bez informacija o tačnom numeričkom opsegu. Ova apstrakcija je znatno efikasnija u pogledu vremena izvršavanja i potrošnje memorije, jer izostavlja detaljno praćenje vrednosti.

Ipak, zbog svoje grublje prirode, ovakav pristup može prevideti greške koje bi bile detektovane preciznijom analizom vrednosnih intervala.

Ovo su bili samo neki od primera koji ilustruju blisku povezanost između apstrakcije i ljudskog načina razmišljanja. Takva povezanost proističe delimično iz stečenog iskustva, a delimično iz same ljudske prirode. U prvom primeru prikazana je aproksimacija u umetnosti, u drugom aproksimacija u svakodnevnom govoru, u trećem aproksimacija u naučnim kontekstima, dok četvrti primer osvetljava ulogu aproksimacije u programiranju.

Shodno tome, može se uočiti da se aproksimacija primenjuje u širokom spektru konteksta. Jedan od značajnih doprinosa u ovoj oblasti potiče iz 1977. godine, kada je francuski informatičar Patrik Kuso (fr. *Patrick Cousot*), zajedno sa suprugom Radijom (fr. *Radhia Cousot*), formulisao osnovne ideje o primeni apstrakcije u verifikaciji softvera. Rad kojim su uveli pojam apstraktne interpretacije je *„Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”* [5].

Osnovna ideja apstraktne interpretacije zasniva se na analizi programa putem apstrakcije domena koja predstavlja aproksimaciju vrednosti promenljivih, stanja programa ili putanja izvršavanja. Takva aproksimacija treba da omogući efikasnu statičku analizu i detekciju širokog spektra grešaka, uključujući deljenje nulom, korišćenje neinicijalizovanih promenljivih i pristup elementima izvan granica niza.

Apstraktna interpretacija se često primenjuje u verifikaciji sigurnosnih svojstava (eng. *safety property*)[2]. Ovaj pristup omogućava analizu programa bez njegovog stvarnog izvršavanja, čime se postiže ušteda vremena i računskih resursa u poređenju sa dinamičkim tehnikama.

Za sprovođenje apstraktne interpretacije neophodno je definisati apstraktni domen koji opisuje aproksimaciju vrednosti i operacija nad tim vrednostima. Ovaj domen može biti jednostavan, poput domena celobrojnih intervala, ili složeniji, poput domena skupova (detaljnije u nastavku teksta).

[2]Sigurnosno svojstvo označava da se određena negativna pojava nikada neće desiti.

Ključni izazov u primeni apstraktne interpretacije jeste postizanje ravnoteže između preciznosti i skalabilnosti. Preciznija apstrakcija može obezbediti tačnije rezultate, ali uz značajno veće računske troškove, dok grublja apstrakcija može biti brža i zahtevati manje resursa, ali uz povećan rizik od generisanja lažnih upozorenja (detaljnije u poglavlju 3.4).

Osnovni pojmovi

Radi potpunijeg razumevanja apstraktne interpretacije, u nastavku će biti objašnjeni ključni pojmovi i koncepti: problem zaustavljanja (eng. *halting problem*), semantika programskih jezika (uključujući apstraktnu semantiku), domen i apstraktni domen.

Problem zaustavljanja

Problem zaustavljanja predstavlja jedan od fundamentalnih pojmova u informatici i odnosi se na pitanje da li će se određeni program, za dati ulaz, zaustaviti ili će se izvršavati beskonačno. Ovaj problem spada u klasu problema odlučivanja.

U teoriji izračunljivosti, problem odlučivanja definiše se kao problem u kojem, na osnovu unapred definisanog formalnog sistema, na postavljeno pitanje treba dati odgovor u obliku *„da”* ili *„ne”*. Na primer, problem odlučivanja može biti pitanje: *„da li x deli y?”*, za unapred zadate brojeve x i y.

Neformalno, halting problem glasi:

Za proizvoljni računarski program i ulaz, da li program završava svoj rad?

Problem zaustavljanja je prvi formalno definisao Alan Tjuring (eng. *Alan Turing*) 1936. godine u radu *„On Computable Numbers, with an Application to the Entscheidungsproblem”* [11], u kojem je dokazao da je ovaj problem neodlučiv. Neodlučivost u ovom kontekstu znači da ne postoji opšti algoritam koji može odrediti, za svaki mogući par programa i ulaza, da li će se izvršavanje programa završiti ili trajati beskonačno.

Ovo fundamentalno ograničenje ima dalekosežne posledice u informatici, jer implicira da nije moguće, u svim slučajevima, automatski odrediti da li će se izvršavanje nekog programa zaustaviti, kao i da li će se izvršavanje nekog dela programa

zaustaviti. To implicira da nije moguće automatski odrediti da li je neka naredba u programu dostižna, pa samim tim ni da li može da dovede do greške u izvršavanju. Kao rezultat toga, sva složenija pitanja o konkretnoj semantici programa su neodlučiva: nije moguće napisati program koji bi mogao odgovoriti na bilo koje pitanje o izvršavanjima bilo kog programa.

Posledica neodlučivosti problema zaustavljanja jeste nemogućnost izrade programa koji bi, potpuno automatski, u konačnom vremenu i uz korišćenje konačnih resursa, mogao tačno da utvrdi ispravnost proizvoljnog programa. Zbog toga se u procesu automatizovane verifikacije često žrtvuje potpuna preciznost. Budući da je postavljeno fundamentalno ograničenje u pogledu utvrđivanja da li neko svojstvo programa važi, neophodno je pribegavati aproksimacijama. Pri razvoju automatskih alata za analizu programa, nužno je praviti kompromis između preciznosti i efikasnosti.

Semantika programskih jezika i apstraktna semantika

Da bi razumeli šta neki program radi gledajući njegov kôd neophodno je dobro poznavanje programskog jezika u kom je program napisan. Nekada kôd na prvi pogled deluje dobro, ali se program *čudno* ponaša i ne daje očekivane rezultate. Da bi se što lakše i brže zaključio uzrok ovakvog ponašanja i otklonio problem bitno je razumeti semantiku programskog jezika.

Semantika programskog jezika govori šta se dešava kada se program izvršava, odnosno određuje značenje tog jezika. Konkretna semantika programa opisuje sve moguće načine na koje program može biti izvršen u svim mogućim okruženjima.

Na osnovu sematike programskog jezika i koda programa može se nedvosmisleno zaključiti šta program radi.

Apstraktna semantika je tehnika u okviru semantike programskih jezika koja pravi aproksimaciju konkretne semantike programskog jezika i koristi se za analizu programa bez stvarnog izvršavanja. Ona se oslanja na apstrakcije kako bi pojednostavila analizu programa, fokusirajući se na bitne aspekte programa dok

zanemaruje manje bitne. Na taj način, apstraktna semantika omogućava analizu programa bez potrebe za svim detaljima konkretnih izvršavanja.

Za svaki program postoji skup mogućih ulaznih vrednosi i program se za različite vrednosti izvršava na različite načine. U programu se mogu naći pozivi funkcija, grananja, petlje, uslovna izvršavanja i slično, a šta će se od navedenog izvršiti, kada i na koji način zavisi od toga kako je program napisan i koje su ulazne vrednosti.

Definicija 1. Stanje programa, prelaz iz stanja u stanje, dostižno stanje
Stanje programa, označeno sa s, predstavlja potpunu sliku sistema u jednom trenutku. Ono obuhvata tekuće vrednosti svih promenljivih koje se javljaju u programu. Skup svih mogućih stanja sistema označava se sa S.
Prelaz iz stanja u stanje, označen sa $s_i \rightarrow s_{i+1}$, opisuje promenu stanja sistema kao rezultat izvršavanja jedne programske instrukcije. Prelaz je dozvoljen ako postoji validna instrukcija u stanju s_i koja vodi u stanje s_{i+1}.
Dostižno stanje je svako stanje $s_n \in S$ za koje postoji putanja prelaza iz početnog stanja s_{start}. Drugim rečima, postoji sekvenca dozvoljenih prelaza:

$$[s_{start} \rightarrow s_1 \rightarrow \cdots \rightarrow s_n]$$

Razmatrajmo jednostavan programski jezik koji podržava učitavanje ulaznih vrednosti, ispisivanje vrednosti, naredbe dodele, uslovnog skoka i naredbu *halt* koja označava završetak rada programa. Program predstavlja konačan niz sastavljen od ovih naredbi. Koji je sledeći korak u programu određuje se na osnovu brojača koji označavamo sa *pc* (skraćeno od engleskog *program counter*).

Neka su izrazi koji se pojavljuju u programu izrazi nad celim brojevima. Neka je $\rho : Var \rightarrow \mathbb{Z} \cup \{\perp\}$ preslikavanje koje promenljivama iz skupa Var dodeljuje celobrojne vrednosti. Ovo preslikavanje može promenljivoj dodeliti nedefinisanu vrednost, koja će biti označena sa $\perp$, u slučaju da je ona rezultat deljenja nulom ili ako joj još uvek nije dodeljena vrednost. Trenutno stanje programa je uređeni par (pc, ρ) dok je novo stanje programa (pc', ρ').

Za potrebe objašnjenja veze između konkretne i apstraktne semantike dovoljno je da definišemo samo semantiku izraza dodele i izraza sabiranja. Semantika izraza

k za preslikavanje ρ, u oznaci $[\![k]\!]_\rho$, se definiše kao vrednost preslikavanja ρ u tački k, odnosno $[\![k]\!]_\rho = \rho(k)$. Semantika izraza $E_1 + E_2$ za preslikavanje ρ, u oznaci $[\![E_1 + E_2]\!]_\rho$, se definiše rekurzivno kao zbir vrednosti podizraza E_1 i E_2, odnosno $[\![E_1 + E_2]\!]_\rho = [\![E_1]\!]_\rho + [\![E_2]\!]_\rho$.

Konkretna semantika naredbi se definiše kao preslikavanje trenutnog stanja programa u novo stanje programa prema sledećim pravilima:

1. Za trenutnu vrednost brojača uzima se ulaz `input k`, onda je $pc' = pc + 1$, a ρ' se dobija od ρ promenom $k \mapsto v$, za uneto $v \in \mathbb{Z}$.

2. Za trenutnu vrednost brojača izršava se ispisivanje vrednosti `print E`, onda je $pc' = pc + 1$, $\rho' = \rho$.

3. Za trenutnu vrednost brojača izršava se dodela $k := E$, onda je $pc' = pc + 1$, a ρ' se dobija od ρ promenom $k \mapsto [\![E]\!]_\rho$.

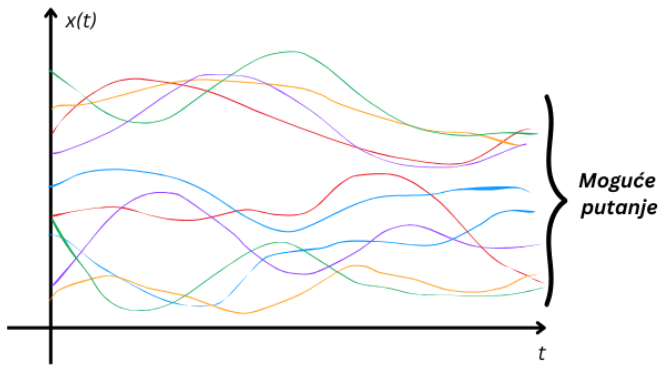
4. Za trenutnu vrednost brojača izršava se uslovni skok `if E goto n`, onda je $pc' = n$ ako je $[\![E]\!]_\rho \neq 0$, inače je $pc' = pc + 1$, $\rho' = \rho$.

5. Za trenutnu vrednost brojača izršava se naredba `halt`, onda je $pc' = n$, gde je n van domena programa, a $\rho' = \rho$.

Početno stanje programa je $s_{start} = (0, \rho)$, gde je $\rho : k \mapsto \perp$ za svako k, jer na početku programa nijedna promenljiva nema definisanu vrednost.

Izvršavanje programa se naziva *putanjom programa* koja predstavlja niz stanja $s = (pc, \rho)$, pri čemu postoje različite putanje u zavisnosti od ulaza korisnika. Semantika programa je skup svih mogućih putanja, za sve moguće ulaze.

Moguće je grafički predstaviti razne putanje kojima se kreće program. Izvršavanje programa može da se predstaviti kroz vreme kao promena vektora *x(t)*, koji predstavlja vrednosti ulaznih podataka, stanja i izlaznih promenljivih programa. Krive opisuju kako se ove vrednosti menjaju u zavisnosti od vremena t, demonstrirajući različita izvršavanja programa u različitim okruženjima i u različitim scenarijima. Ako se izvršavanje predstavi krivom koja pokazuje promenu vektora *x(t)* kroz vreme t, konkretna semantika se može predstaviti skupom takvih krivih (primer prikazan na slici 3.3).



Slika 3.3: Primer konkretne semantike predstavljene skupom krivih

Primer 5. Potrebno je odrediti maksimum dva cela broja a i b, koji će biti označen sa max. Ulazni podaci su a i b, svako i-to stanje će biti označeno sa s_i pri čemu je početno stanje s_{start} a krajnje s_{finish}. Dijagram stanja za određivanje maksimuma dva broja dat je na slici 3.4 a kôd je prikazan u listingu 3.1. Najpre se program nalazi u početnom stanju gde su ulazne vrednosti a i b još uvek nepoznate jer ih tek treba uneti, takođe izlazna vrednost max je nepoznata tj. imaju vrednost $\perp$:

$$x(t_0) = [\perp, \perp, s_{start}, \perp]$$

Zatim se unose ulazne vrednosti. U stanju s_1 vektor $x(t)$ ima vrednosti za a i b a vrednost max je idaje nepoznata:

$$x(t_1) = [a, b, s_1, \perp]$$

U sledećem koraku se porede vrednosti a i b i na osnovu rezultata poređenja se bira dalji put izvršavanja. Postoje dve moguće putanje izvršavanja programa:

- Putanja kada je $a > b$ – u ovom slučaju promenljiva max dobija vrednost a, a vektor *x(t)* prelazi u stanje:

  $$x(t_2) = [a, b, s_2, a]$$

  Primeri vrednosti koje vode ovom putanjom su $a = 10, b = 5$.

- Putanja kada je $a \leq b$ – u ovom slučaju promenljiva max dobija vrednost b, a vektor *x(t)* prelazi u stanje:

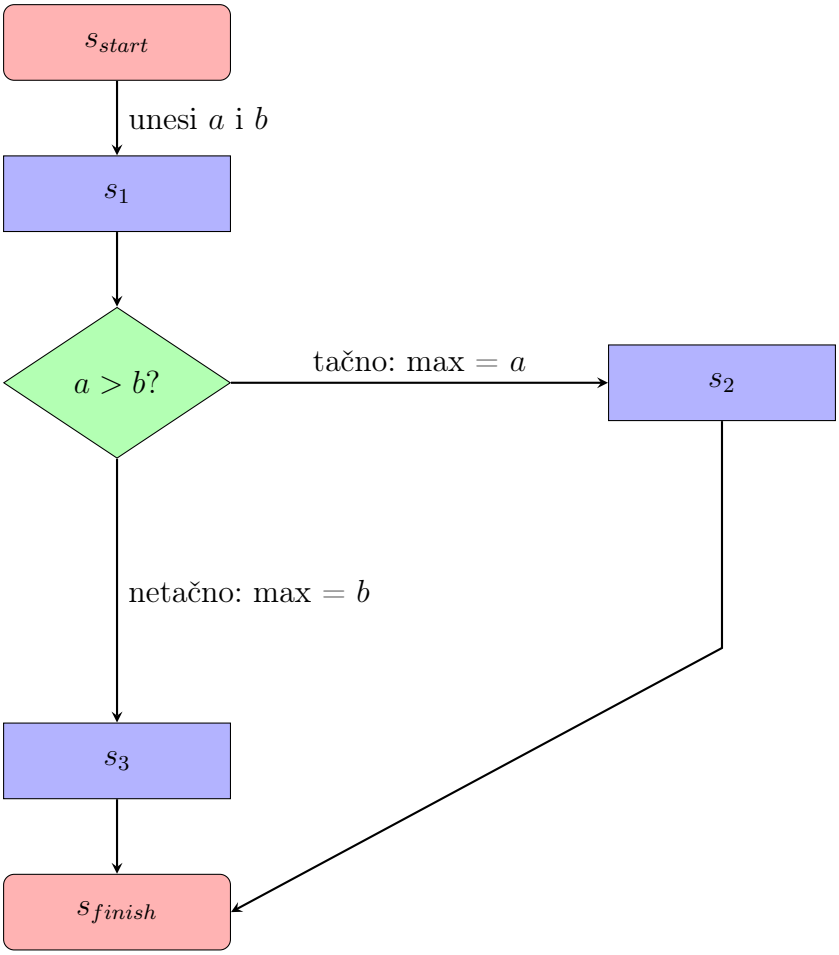
  $$x(t_2) = [a, b, s_3, b]$$

Primeri vrednosti koje vode ovom putanjom su $a = 5, b = 10$.

Nakon ovog koraka, bez obzira na putanju, program dospeva u krajnje stanje:

$$x(t_3) = [a, b, s_{finish}, max]$$

Sem ulaznih i izlaznih promenljivijh, u ovom primeru nema dodatnih promenljivih koje se menjaju tokom izvršavanja, pa se stanje programa ne menja po drugim komponentama.



Slika 3.4: Dijagram stanja za određivanje maksimuma dva broja

```
a, b = map(int, input("Unesite a i b (odvojene razmakom): ").split())

if a > b:
    max = a
else:
```

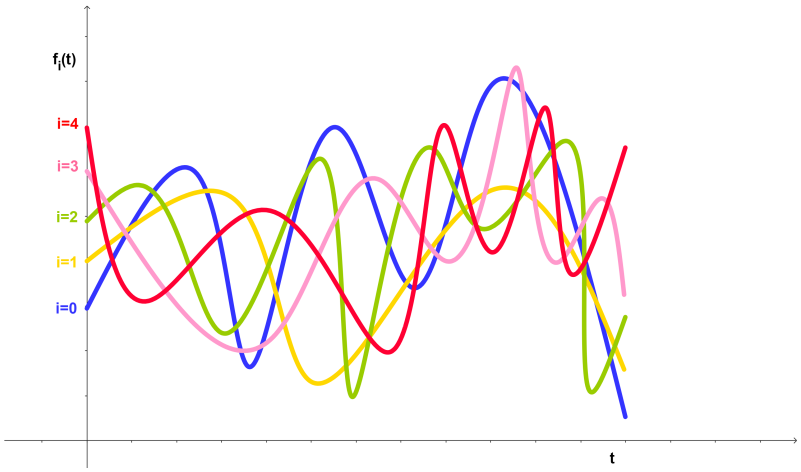
```
    max = b

print("Maksimum je:", max)
```

Listing 3.1: Kôd za određivanje maksimuma dva broja u jeziku Python

> Konkretna semantika predstavlja beskonačan matematički objekat koji nije izračunljiv: nemoguće je kreirati program koji bi mogao da predstavi i izračuna sva moguća izvršavanja za bilo koji program u svim mogućim uslovima.

Kao rezultat toga, većina netrivijalnih pitanja o konkretnoj semantici programa su neodlučiva: nije moguće napisati program koji bi mogao odgovoriti na bilo koje pitanje o izvršavanjima bilo kog programa. Za pravljenje alata koji bi automatski radio verifikaciju softvera pored neodlučivosti, prepreku predstavlja i kompleksnost izračunavanja. Za prevazilaženje ove prepreke koriste se razne aproksimacije. Kako se konkretna semantika predstavlja grafički kao skup krivih, podsetimo se kroz sledeći primer kako se aproksimira skup funkcija.

Primer 6. Neka su grafički prikazane funkcije f_0, f_1, f_2, f_3, f_4. Ovih pet funkcija se može aproksimirati na različite načine

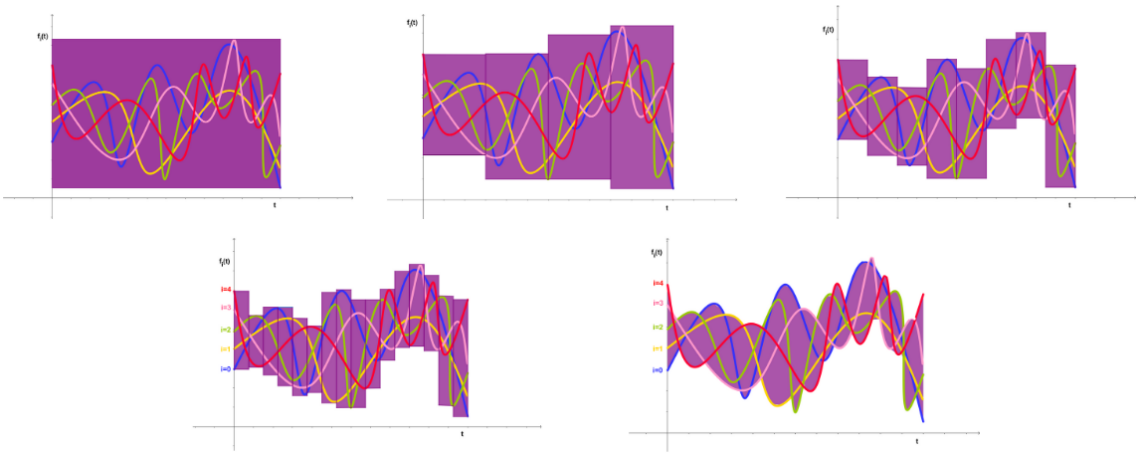


Slika 3.5: Skup funkcija

Primećuje se da je svaka od posmatranih funkcija ograničena, te je i njihov skup takođe ograničen. Ukoliko se odredi minimum m kao najmanja vrednost koja se dostiže u skupu funkcija i maksimum M kao najveća vrednost koja se dostiže

u istom skupu, može se reći da je taj skup funkcija apstrahovan pravougaonikom oblika $t \times [m, M]$.

Budući da je ovakvo rešenje grubo, može se pristupiti njegovom profinjenju podelom domena funkcija na manje segmente, pri čemu se na svaki segment primenjuje isti postupak. Na taj način dobija se sve finija aproksimacija funkcija, koja konvergira ka tačnom pokrivaču skupa funkcija, što je ilustrovano na slici 3.6. Pokrivač funkcija je nadskup skupa funkcija koji predstavlja njihovu aproksimaciju. U ilustracijama se, radi lakšeg praćenja, označava ljubičastom površinom.

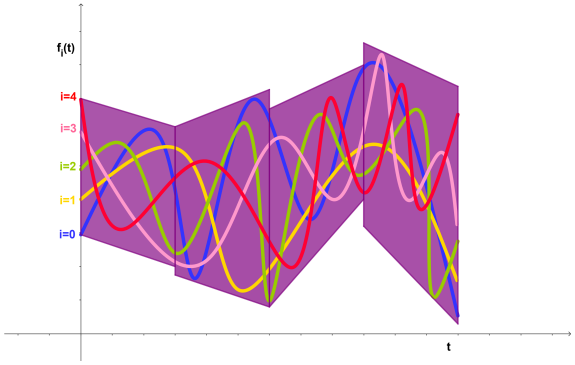


Slika 3.6: Aproksimacije skupa funkcija

Ovakva aproksimacija može delovati dovoljno dobra, međutim, u kontekstu apstraktne interpretacije to ne mora biti slučaj. Budući da je semantika programa opis načina na koji se program ponaša, očekivano je da postoje stanja koja su nepoželjna, odnosno stanja u kojima se program ne sme naći. Ta stanja nazivaju se *zabranjenim zonama* (detaljnije u odeljku 3.4). U grafičkoj interpretaciji, zabranjena zona može se nalaziti bilo gde unutar prikaza grafikona funkcija. Zbog toga aproksimacija opisana u prethodnom primeru nije uvek pogodna. U određenim situacijama biće potrebno iz pokrivača izostaviti određene delove (praviti *rupe*) kako bi se izbegle zabranjene zone.

Takođe, aproksimacija se ne mora vršiti isključivo pomoću pravougaonika. U prikazanom postupku domen funkcija deli se na manje segmente, a odsečak skupa funkcija koji pripada segmentu aproksimira se pravougaonikom. Na taj način postiže se postepena konvergencija ka tačnom pokrivaču funkcija (poslednji prikaz na slici 3.6).

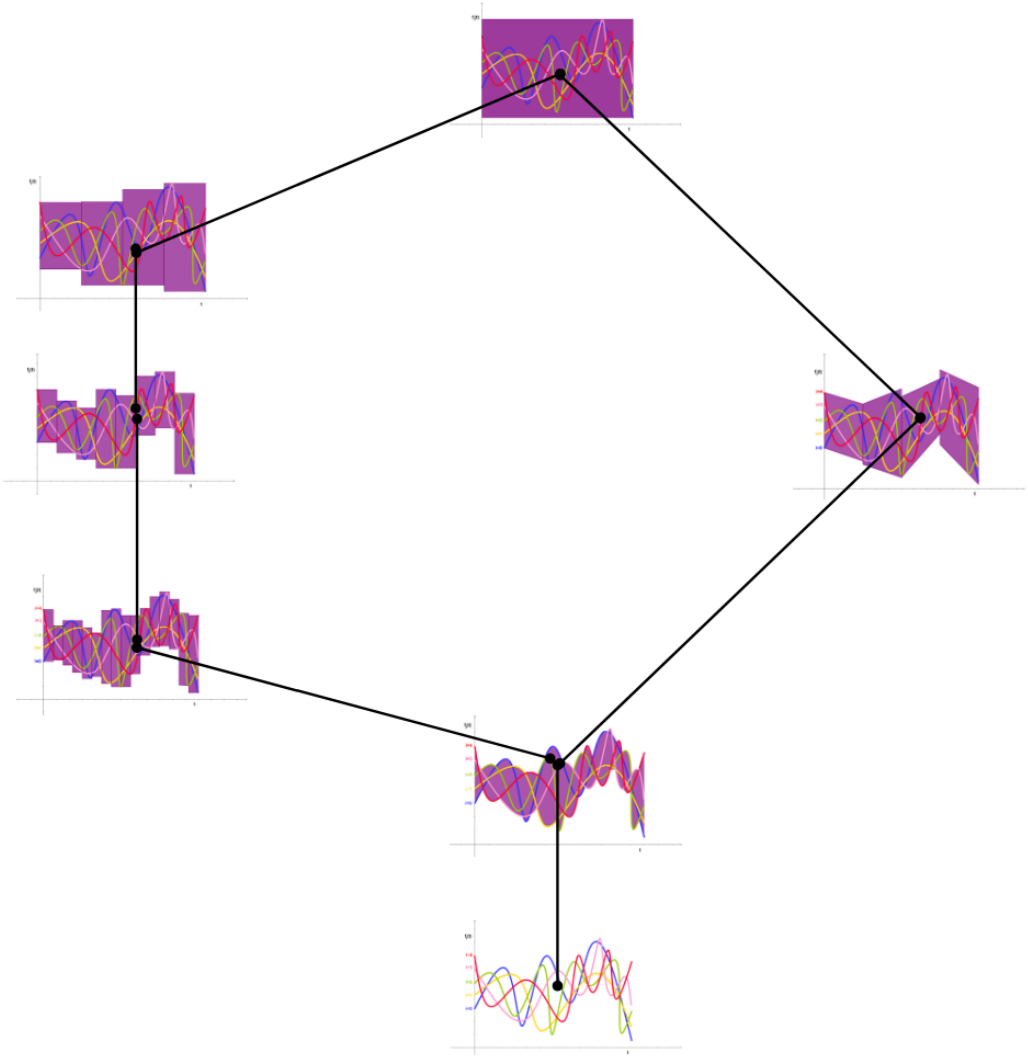
Na slici 3.7 dat je jedan primer koji ne koristi pristup pravougaonika, a daje preciznije profinjenje u mnogo manjem broju koraka. Odnos ova dva pristupa dat je grafički na slici 3.8. Slika se može posmatrati kao stablo na čijem je vrhu najjednostavnija aproksimacija od koje se kreće. Leva grana predstavlja put aproksimacije pravougaonicima. Desna grana predstavlja put aproksimacije proizvoljnim oblicima. Leva strana ima više čvorova, i što je čvor niži daje finiju aproksimaciju. Na kraju se spušta do tačnog pokrivača skupa funkcija. Desna strana ima samo jedan čvor na putu od vrha stabla do tačnog pokrivača i on predstavlja aproksimaciju skupa funkcija četvorouglovima. Spuštanje do tačnog pokrivača dobija se u znato manje koraka što ukazuje na to će se često brže doći do fine aproksimacije koristeći različite geometrijske oblike. Najniža grana je grana koja spaja poslednji unutrašnji čvor (tačan pokrivač) sa listom (zadati skup funkcija). Spuštanje kroz stablo odozgo na dole može se posmatrati kao proces konvergencije ka zadatom skupu funkcija.



Slika 3.7: Apstrakcija korišćenjem različitih geometrijskih oblika

Domen i apstraktni domen

Domen jednog programa je specifično područje problema koji program rešava. U kontekstu razvoja softvera, domen se odnosi na opseg vrednosti koje se očekuju kao ulaz u program. Razumevanje domena programa je ključno za dizajn i implementaciju, jer pomaže programerima da stvore softverska rešenja koja su prilagođena specifičnim potrebama. Dobro poznavanje domena je neophodno i za verifikaciju kako bi se program testirao za što veći broj mogućih ulaza, čime se smanjuje verovatnoća za eventualne propuste u funkcionisanju programa.



Slika 3.8: Hijerarhija apstrakcija

Apstraktni domen

Veliki programi imaju veliki domen i veliki skup svih mogućih izvršavanja. Zarad apstraktne interpretacije, prvo treba pronaći odgovarajući apstraktni domen, a zatim apstraktnu semantiku programa.

Za svako svojstvo koje se dokaže u apstraktnom okruženju, ono mora važiti i u konkretnim okvirima. Odnoso, ako se apstraktnom interpretacijom tvrdi da

je svojstvo tačno, onda je ono sigurno tačno u konkretnom programu.

Za početak je potrebno shvatiti kako se traži apstraktni domen. Potrebno je napraviti širu sliku domena i uočiti koja su to najbitnija svojstva koja treba posmatrati. U odnosu na svojstva koja se posmatraju, mogu se praviti različiti apstraktni domeni.

Semantika svakog programa može se opisati konkretnim domenom D_c i relacijama nad ovim domenom. Za svaki konkretni domen postoji i njegova apstrakcija — apstraktni domen D_a. Apstraktni domen je opštiji od konkretnog, u sebi mora sadržati sve vrednosti konkretnog domena i može se posmatrati kao opis vrednosti konkretnog domena. U nastavku su dati primeri koji će pomoći u sticanju intuicije o apstraktnom domenu.

Primer 7. Neka je dat jedan ceo broj. Konkretan domen D_c je skup celih brojeva. Ukoliko je potrebno odrediti znak ovog broja i nekih aritmetičkih izraza u kojima on učestvuje, za njegov apstraktni domen D_a se može uzeti skup vrednosti znakova celih brojeva. Time se dolazi do sledeće apstrakcije:

- $a_0 = \{0\}$ Skup a_0 sadrži samo nulu.
- $a_+ = \{\mathrm{n} \mid \mathrm{n} > 0\}$ Skup a_+ je skup koji sadrži sve pozivine brojeve.
- $a_- = \{\mathrm{n} \mid \mathrm{n} < 0\}$ Skup a_- je skup koji sadrži sve negativne brojeve.

Dakle, nije bitno kolika je apsolutna vrednost zadatog broja, ni od čega je on veći ili manji, već samo kog je znaka zadati broj. To svojstvo je pokriveno skupovima a_+, a_- i a_0. Unijom ova tri skupa su obuhvaćeni svi brojevi. Ovakva apstrakcija je dovoljna za određivanje znaka jednog zadatog celog broja ali se može primeniti i na određivanje znaka rezultata nekih binarnih operacija. Na primer, ako se posmatra operacija množenja ova apstrakcija je dovoljna, slično je i za deljenje.

Objašnjenje: nula pomnožena sa brojem proizvoljnog znaka je nula, ako dva činioca imaju isti znak proizvod je pozitivan, inače je negativan (prikazano u tabeli 3.2). Ista tabela važi i za deljenje pri čemu treba voditi računa da deljenje nulom nije definisano.

U okviru *Jupyter* sveske napisan je kôd koji implementira određivanje znaka prilikom množenja dva broja preko apstraktnog domena. Zatim je dat kôd koji testira taj apstrktni domen tako što provera vrednosti iz tabele 3.2.

$\times$	a_0	a_+	a_-
a_0	a_0	a_0	a_0
a_+	a_0	a_+	a_-
a_-	a_0	a_-	a_+

Tabela 3.2: Tabela za operaciju množenja

Ova apstrakcija nije dovoljna za sve binarne operacije. Ako se određuje znak zbira pozitivnog i negativnog broja ova apstrakcija ne daje odgovor. Zbir će biti istog znaka kao sabirak sa većom apsolutnom vrednošću što znači da može biti pozitivan, negativan ili nula ako su apsolutne vrednosti sabiraka jednake. Slična je situacija i sa oduzimanjem. Određivanje znaka za sabiranje je prikazano u tabeli 3.3.

$+$	a_0	a_+	a_-
a_0	a_0	a_+	a_-
a_+	a_+	a_+	?
a_-	a_-	?	a_-

Tabela 3.3: Tabela za operaciju sabiranja

Česta je situacija da je apstraktni domen dosta jednostavan i da ne može dati odgovore na sva pitanja.

U tabeli su polja koja imaju vrednost ? polja koja predstavljaju gubitak informacije o znaku rezltata prilikom sabiranja dva broja. Potrebno je izmeniti postojeću apstrakciju tako da se reši problem gubitka informacija prilikom sabiranja i oduzimanja. Rešenje je da se apstraktni domen proširi tako da obuhvata sve brojeve:

$$a_0 = \{0\}$$
$$a_+ = \{\mathrm{n} \mid \mathrm{n} > 0\}$$
$$a_- = \{\mathrm{n} \mid \mathrm{n} < 0\}$$
$$a = \{\mathrm{n}\}$$

Skupovi a_0, a_+ i a_- imaju isto značenje kao i u prethodnoj apstrakciji. Novi skup koji je dodat u apstrakciju je skup a koji sadrži sve brojeve. On se koristi u situacijama kada se za rezultat binarne operacije ne može pouzdano tvrditi kog je znaka, odnosno rezultat može biti pozitivan, negativan ili nula u zavisnosti od znaka

i apsolutnih vrednosti operanada. Ovakvim proširanjem apstrakcije su uključeni slučajevi koji su pre proširenja bili nedefinisani, odnosno u tabeli označeni sa ?. Nova tabela za sabirenje je tabela 3.4.

$+$	a_0	a_+	a_-	a
a_0	a_0	a_+	a_-	a
a_+	a_+	a_+	a	a
a_-	a_-	a	a_-	a
a	a	a	a	a

Tabela 3.4: Proširena tabela za operaciju sabiranja

Primećuje se da a zapravo predstavlja gubitak informacije, odnosno situaciju u kojoj ne znamo ništa o znaku rezultata.

Primer 8. Neka je dat jedan prirodan broj. Potrebno je odrediti parnost tog broja. Konkretna implementacija ovog problema mogla bi se svesti na deljenje datog broja sa dva i proveravanje ostatka pri deljenju. Apstraktna interpretacija bi ovaj problem svela na proveravanje poslednje cifre datog broja. Konkretan domen ovog problema je skup prirodnih brojeva, dok je apstraktni domen skup cifara {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Primećuje se da je apstraktni domen znatno manji.

Ova aproksimacija se može posmatrati kao preslikavanje koje svaki prirodan broj slika u svoju poslednju cifru. U ovom slučaju dolazi do gubitka ostalih informacija o broju, ali ako se ispituje samo parnost broja te informacije nisu od značaja. Ono što se postiže ovakvom aproksimacijom jeste efikasnost (posmatranje poslednje cifre je u opštem slučaju efikasnije od deljenja sa dva). Ova apstrakcija se može primeniti na binarne operacije sabiranja, oduzimanja i množenja. Ako znamo parnost operanada znamo i parnost rezultata za ove tri binarne operacije.

Objašnjenje: zbir dva parna broja je paran broj; zbir dva neparna broja je paran broj; zbir parnog i neparnog broja je neparan broj. Slično se može uočiti pravilo i za operacije oduzimanja i množenja.

Operacija deljenja se razikuje i na nju se ne može primeniti ova apstrakcija. Deljenje je suprotna operacija od množenja. Množenjem dva parna broja dobija se paran broj (primer $2 * 4 = 8$) ali se i množenjem parnog i neparnog broja takođe dobija paran broj (primer $2 * 5 = 10$). Kako imam više situacija koje daju paran proizvod, ne može se utvrditi koje su parnosti činioci od kojih je taj proizvod nastao. Ovakve situacije se prenose na operaciju deljenja i zato ona nije uvek definisana u smislu određivanja parnosti broja. Za dat paran deljenik i delilac ne može se sa

sigurnošću tvditi da li je količnik paran ili neparan. Ovako izgleda primer koji se koristio pri množenju kada se prebaci u deljenje:
$8/2 = 4$ - parno, $10/2 = 5$ - neparno
Drugi problem koji se javlja kod deljenja je što količnik dva prirodna (ili cela) broja ne mora biti prirodan (ceo) broj. Ovaj problem se rešava na sličan način kao u prethonom primeru — uvođenjem apstrakcije koje obuhvata sve brojeve.

Primer 9. Neka je dat jedan prirodan broj iz intervala [0, 1 000 000 000]. Potrebno je ispitati da li je dati broj deljiv sa 3. Konkretana interpretacija bi ovaj zadatak rešila tako što bi proveravala ostatak pri deljenju broja sa 3. Apstraktna interpretacija daje drugačije rešenje — proverava da li je zbir cifara datog broja deljiv sa tri[3]. Ova aproksimacija se može posmatrati kao preslikavanje koje prirodan broj iz intervala [0, 1 000 000 000] slika u zbir svojih cifara. Konkretni domen je skup prirodnih brojeva iz navedenog intervala, dok je apstraktni domen skup brojeva iz intervala [0, 81] što je znatno manji domen.

3.3 Uređenje skupova i Galoaove veze

Apstraktni domen je nadskup konkretnog domena. To znači da apstraktni domen mora sadržati sve ulazne vrednosti programa, a često sadrži i više od toga. Slično važi i za semantiku programa — apstraktna semnatika je nadskup konkretne semantike. Da bi se razumeo odnos apstraktne i konkretne semantike programa koriste se skupvi i vrši se poređenje sukpova. Da bi bilo razumljivo kako se skupovi porede potrebno je uvesti pojam *uređenja* na skupu.

Definicija 1. Parcijalno uređen skup
Parcijalno uređen skup je uređen par $(L, \leq)$, gde je L neprazan skup, a $\leq$ binarna relacija definisana na tom skupu, za koju važi:

- refleksivnost: $\forall a \in L, a \leq a$,
- antisimetričnost: $\forall a, b \in L, a \leq b \wedge b \leq a \Rightarrow a = b$,
- tranzitivnost: $\forall a, b, c \in L, a \leq b \wedge b \leq c \Rightarrow a \leq c$.

[3]Broj je deljiv sa 3 ako i samo ako mu je zbir cifara deljiv sa 3.

Realcija uređenja ne mora biti potpuna, odnosno mogu postojati dva elementa a i b za koje ne mora da važi nijedna od naredne dve relacije: $a \leq b, b \leq a$.

Definicija 2. Gornja granica, donja granica, najmanja gornja granica (supremum), najveća donja granica (infimum)
Element $s \in L$ je *gornja granica* skupa $X \subseteq L$ ako je $\forall x \in X, x \leq s$.
Element $i \in L$ je *donja granica* skupa $X \subseteq L$ ako je $\forall x \in X, i \leq x$.
Element $s \in L$ je *najmanja gornja granica (supremum)* skupa $X \subseteq L$ ako je s gornja granica skupa X i za svaku gornju granicu $s^{'} \in L$ važi $s \leq s^{'}$.
Element $i \in L$ je *najveća donja granica (infimum)* skupa $X \subseteq L$ ako je i donja granica skupa X i za svaku donju granicu $i^{'} \in L$ važi $i^{'} \leq i$.
Najmanja gornja granica za bilo koja dva elementa $a, b \in L$ se označava sa $a \vee b$ ($\vee$ se čita kao *disjunkcija*, *najmanja gornja granica*, *supremum*, ili *join*) i to je najmanji element u L koji je veći ili jednak i od a i od b:

$$a \vee b = \text{najmanja gornja granica od } a \text{ i } b.$$

Najveća donja granica za bilo koja dva elementa $a, b \in L$ se označava sa $a \wedge b$ ($\wedge$ se čita kao *konjunkcija*, *najveća donja granica*, *infimum*, ili *meet*) i to je najveći element u L koji je manji ili jednak i od a i od b:

$$a \wedge b = \text{najveća donja granica od } a \text{ i } b.$$

Definicija 3. Rešetka
Rešetka je parcijalno uređen skup $(L, \leq)$ takav da za svaka dva elementa $a, b \in L$ postoji najmanja gornja granica $a \vee b$ i najveća donja granica $a \wedge b$. Rešetka se označava sa $(L, \leq, \vee, \wedge)$. Binarne operacije $\vee$ i $\wedge$ na skupu L zadovoljavaju sledeće aksiome [1]:

- idempotentnost

$$a \vee a = a$$
$$a \wedge a = a$$

- komutativnost

$$a \vee b = b \vee a$$
$$a \wedge b = b \wedge a$$

- asocijativnost

$$(a \vee b) \vee c = a \vee (b \vee c)$$
$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$

- apsorpcija

$$a \vee (a \wedge b) = a$$
$$a \wedge (a \vee b) = a$$

Prema definiciji rešetke, za svaki dvočlani podskup $X \subset L$ postoji najmanja gornja granica i najveća donja granica. Zaključuje se indukcijom da onda i za svaki konačan podskup postoji najmanja gornja granica i najveća donja granica. Međutim, to ne mora biti slučaj za beskonačne podskupove.

Definicija 4. Potpuna rešetka
Potpuna rešetka je parcijalno uređen skup $(L, \leq)$ takav da svaki podskup $X \subseteq L$ ima najmanju gornju granicu $\bigvee X$ i najveću donju granicu $\bigwedge X$. Specijalno, skup L tada ima najveći element $\top = \bigvee L$ i najmanji element $\bot = \bigwedge L$. Rešetku koja ima najveći i najmanji element nazivamo *ograničenom*. Potpuna rešetka se označava sa $(L, \leq, \bot, \top, \vee, \wedge)$.

Uvođenje relacije $\leq$ je bitno jer se njome postiže preciznost prilikom biranja apstrakcije. Ako je $a \leq b$, onda je a precizniji od b. Na primer, interval $[a, b]$ je precizniji od intervala $[a', b']$ ako je sadržan u njemu, tj. ako je $a' \leq a$ i $b \leq b'$, i tada pišemo $[a, b] \leq [a', b']$.

Primer 1. U skupu celih brojeva možemo posmatrati rešetku intervala

$$\{\emptyset\} \cup \{[a, b] \mid a \leq b, a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}\}$$

Radi jednostavnosti za neograničene intervale $(-\infty, b]$, $[a, +\infty)$ i $(-\infty, +\infty)$ koristimo oznake $[a, b]$. U ovom primeru važi sledeće:

- $\leq = \subseteq$ - operacija poređenja je operacija podskupa
- $\vee = \cup$ - najmanje gornje ograničenje se dobija operacijom unije
- $\wedge = \cap$ - najveće donje ograničenje se dobija operacijom preseka
- $\perp = \emptyset$ - najmanji element je prazan skup jer je sadržan u svakom intervalu
- $\top = (-\infty, +\infty)$ - najveći element je element koji sadrži sve druge intervale

Ova rešetka je potpuna.

Verifikacija programa apstraktnom interpretacijom može se posmatrati kao preslikavanje između dva uređena skupa. Konekretni skup (domen preslikavanja) je skup stanja i putanja programa. Apstraktni skup (kodomen preslikavanja) je apstrakcija programa. Nad svakim od ova dva skupa postoje relacije uređenja koje se mogu interpretirati kao *nivoi znanja*: u konkretnom skupu veći skup znači više mogućih stanja i putanja; u apstraktnom skupu veći elementi predstavljaju grublje (manje precizne) apstrakcije.

Definicija 5. Sakupljajuća semantika
Sakupljajuća semantika je semantika koja računa skup mogućih putanja. Formalno se može definisati kao

$$Tr = \{s_1 \to \cdots \to s_n \mid s_1 \text{ je početno stanje}, s_i \to s_{i+1} \text{ je dostižno}\}.$$

Skup dostižnih stanja je $Sr = \{s \mid s_1 \to \cdots \to s \in Tr\}$. Skupovi svih putanja i stanja su uređenje sa relacijom inkluzije ($\subseteq$) i čine potpunu rešetku.

Potrebna je veza kojom će biti spojeni original i apstrakcija. To se postiže korišćenjem Galoaovih veza.

Definicija 6. Funkcija apstrakcije i funkcija konkretizacije
Funkcija apstrakcije α preslikava skup konkretnih objekata u njegovu apstraktnu interpretaciju.

Funkcija konkretizacije γ preslikava apstraktni skup objekata u konkretne.

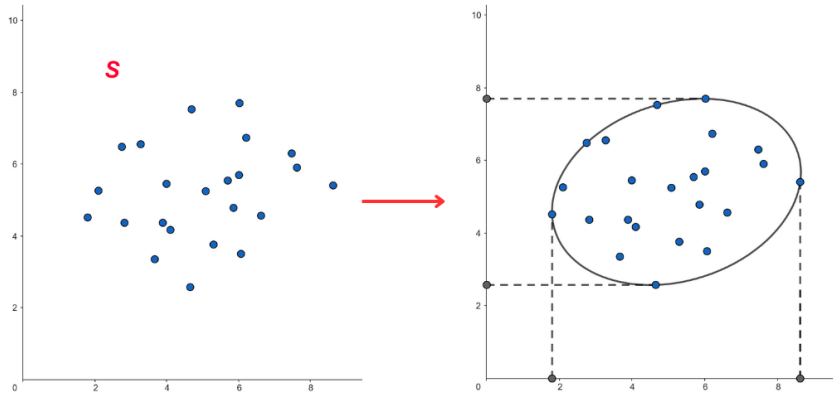
Može se primetiti da se primenom apstrakcije pa konkretizacije gubi na preciznosti, tj. važi:

$$S \subseteq \gamma(\alpha(S)).$$

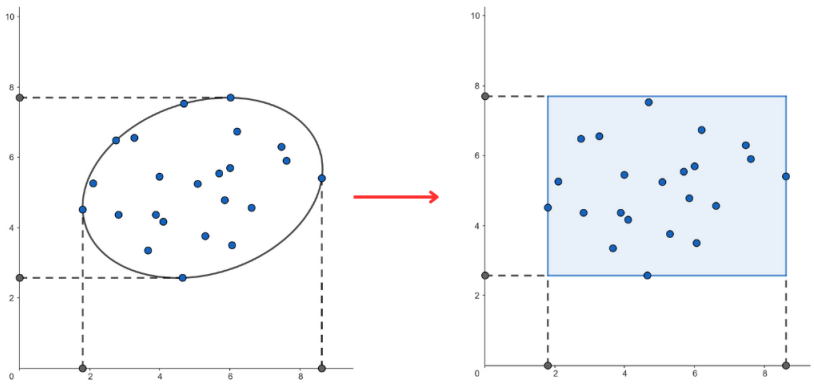
Primer 2. Dat je skup konkretnih objekata S koji se preslikava u apstraktni skup funkcijom apstrakcije α, prikazano na slici 3.9.
Na slici 3.10 dat je apstraktni skup $\alpha(S)$ koji se funkcijom konkretizacije γ slika u pravugaonik čija dužina odgovara vrednostima koje uzima x iz intervala $[1.8, 8.5]$ a visina vrednostima koje uzima y iz intervala $[2.6, 7.7]$.
Ako se uporedi skup konkretnih objekata S od kog je počelo preslikavanje i slika dobijena funkcijom konkretizacije $\gamma(\alpha(S))$ uočava se da je zaista:

$$S \subseteq \gamma(\alpha(S))$$

.



Slika 3.9: Funkcija apstrakcije



Slika 3.10: Funkcija konkretizacije

Definicija 7. Galoaova veza
Neka su $(L_1, \leq_1)$ i $(L_2, \leq_2)$ dva parcijalno uređena skupa, i neka su date monotone funkcije[4] α i γ pri čemu važi $\alpha : L_1 \to L_2$, $\gamma : L_2 \to L_1$. Par (α, γ) definiše *Galoaovu vezu* između L_1 i L_2 ako i samo ako važi:

$$(\forall x \in L_1, \forall y \in L_2) \quad x \leq_1 \gamma(y) \Longleftrightarrow \alpha(x) \leq_2 y$$

Galoaova veza između L_1 i L_2 sa definisanim funkcijama α i γ označava se sa $(L_1, \leq_1) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} (L_2, \leq_2)$.

Parcijalno uređen skup $(L_1, \leq_1)$ je konkretan, a parcijalno uređen skup $(L_2, \leq_2)$ je apstraktan. Preslikavanje α je apstrakcija, a preslikavanje γ je konkretizacija.

Definicija 8. Ekstenzivna funkcija
Neka je $f : L \to L$ funkcija na parcijalno uređenom skupu $(L, \leq)$. Kažemo da je f *ekstenzivna* ako za svako $x \in L$ važi:

$$x \leq f(x).$$

[4]Monotona funkcija podrazumeva monotono rastuću funkciju osim ako nije drugačije naglašeno

Definicija 9. Kontraktivna funkcija
Neka je $f : L \to L$ funkcija na parcijalno uređenom skupu $(L, \leq)$. Kažemo da je f *kontraktivna* ako za svako $x \in L$ važi:

$$f(x) \leq x.$$

Definicija Galoaove veze zahteva da, za svaku funkciju $\alpha : L_1 \to L_2$ i $\gamma : L_2 \to L_1$, važi:

$$(\forall x \in L_1, \forall y \in L_2) \quad x \leq_1 \gamma(y) \Longleftrightarrow \alpha(x) \leq_2 y$$

Ova formulacija uključuje proveru za sve moguće parove $(x, y) \in L_1 \times L_2$, što može biti nepraktično, naročito kada se posmatraju beskonačni ili veliki skupovi. Dodatno, prisustvo logičke ekvivalencije komplikuje proveru u praksi, jer zahteva razmatranje oba smera implikacije.

Međutim, postoji korisna karakterizacija koja značajno olakšava proveru da li dve funkcije formiraju Galoaovu vezu:

Teorema 9. Neka su $(L_1, \leq_1)$ i $(L_2, \leq_2)$ parcijalno uređeni skupovi i neka su date funkcije $\alpha : L_1 \to L_2$, $\gamma : L_2 \to L_1$. Par (α, γ) definiše Galoaovu vezu između $(L_1, \leq_1)$ i $(L_2, \leq_2)$ [2] ako i samo ako su

- α i γ monotone,
- $\gamma \circ \alpha$ je ekstenzitivna, tj. $\forall x \in L_1, x \leq_1 (\gamma \circ \alpha)(x)$ i
- $\alpha \circ \gamma$ je kontraktivna, tj. $\forall y \in L_2, (\alpha \circ \gamma)(y) \leq_2 y$.

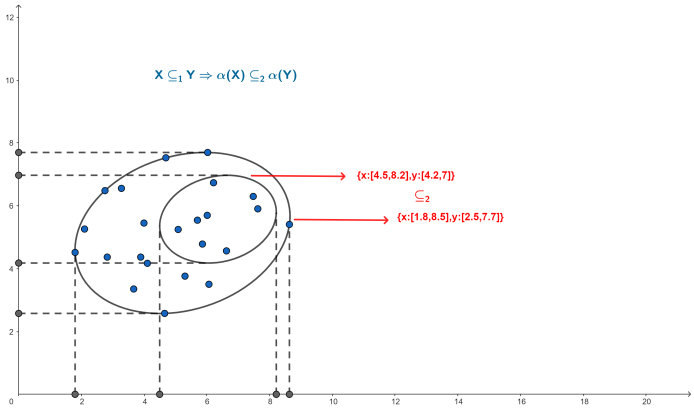
Na osnovu ove teoreme, umesto proveravanja uslova za sve parove (x, y), može se proveriti ponašanje kompozicija $\gamma \circ \alpha$ i $\alpha \circ \gamma$ nad pojedinačnim elementima. Drugim rečima, dovoljna je provera ponašanja funkcija na svakom elementu posebno, bez potrebe za povezivanjem različitih domena funkcija.

U praksi, kada su α i γ već konstruisane kao monotone funkcije (što je često slučaj kod interpretacije intervala ili drugih apstrakcija), provera da su $\gamma \circ \alpha$ ekstenzitivna i $\alpha \circ \gamma$ kontraktivna može se izvršiti jednostavno evaluacijom kompozicija i upoređivanjem rezultata sa početnim vrednostima. To čini ovu teoremu posebno

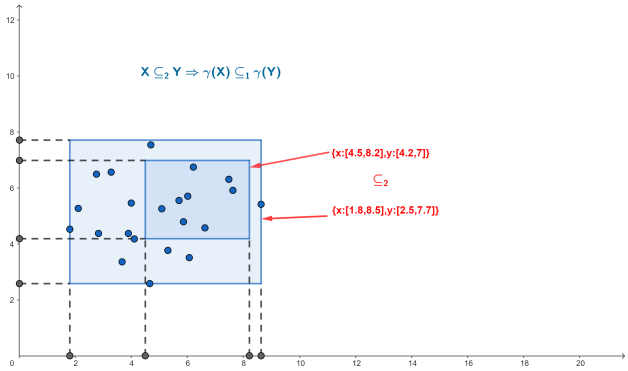
pogodnom za mehaničku proveru u kontekstu apstraktne interpretacije i automatizovane analize softvera.

Grafički prikazi monotonosti fukncije apstrakcije dat je na slici 3.11 - ako je X podskup skupa Y u konkretnom okruženju tada je slika skupa X podskup slike skupa Y u apstraktnom okruženju tj. važi: $X \subseteq_1 Y \implies \alpha(X) \subseteq_2 \alpha(Y)$.

Grafički prikazi monotonosti fukncije konkretizacije dat je na slici 3.12 - ako je X podskup skupa Y u apstraktnom okruženju tada je slika skupa X podskup slike skupa Y u konkretnom okruženju tj. važi: $X \subseteq_2 Y \implies \gamma(X) \subseteq_1 \gamma(Y)$.



Slika 3.11: Grafički prikaz svojstva monotonosti funkcije apstrakcije



Slika 3.12: Grafički prikaz svojstva monotonosti funkcije konkretizacije

Osobine $\alpha(x) \leq_2 y$ i njoj dualna osobina $x \leq_1 \gamma(y)$ znače da je y dobra aproksimacija konkretnog svojstva x. Pri tome je $\alpha(x)$ najpreciznija aproksimacija $x \in L_1$ u L_2. Najpreciznija znači da je to je najmanji (po $\leq_2$) element u L_2 koji zadovolja-

va uslov aproksimacije, odnosno od svih elemenata u L_2 koji su dovoljno veliki da *pokrivaju* x, u smislu Galoaove veze, nema nijednog koji je strogo manji (precizniji) od $\alpha(x)$. Element $\gamma(y)$ je najneprecizni element iz L_1 koji se može korektno aproksimirati elementom $y \in L_2$. Najnepreciziniji znači da obuhvata sve moguće konkretne vrednosti koje odgovaraju toj apstraktnoj vrednosti y.

Intuitivno, α zadržava što više informacija pri prelasku iz konkretnog u apstraktni domen dok γ vraća najveći skup konkretnih vrednosti koji odgovara datoj apstraktnoj vrednosti.

Primer 3. Parcijalno uređeni skup podskupova skupa celih brojeva sa uređenjem inkluzije $(P(\mathbb{Z}), \subseteq)$ može se apstrahovati na osnovu znakova elemenata. U ovom slučaju, skup koji je konkretan je $P(\mathbb{Z})$, a skup koji je apstraktni je $P(\{-, 0, +\})$ [9]. Uređenje nad apstraktnim skupom je takođe inkluzija $(P(\{-, 0, +\}), \subseteq)$. Galoaove veze su date sa:

$$
\begin{aligned}
\alpha(\emptyset) &= \emptyset, \\
\alpha(\{0\}) &= \{0\}, \\
\alpha(S) &= \{+\}, \text{ ako } S \text{ sadrži samo pozitivne brojeve,} \\
\alpha(S) &= \{-\}, \text{ ako } S \text{ sadrži samo negativne brojeve,} \\
\alpha(S) &= \{0, +\}, \text{ ako } S \text{ sadrži samo nulu i pozitivne brojeve,} \\
\alpha(S) &= \{-, 0\}, \text{ ako } S \text{ sadrži samo nulu i negativne brojeve,} \\
\alpha(S) &= \{-, +\}, \text{ ako } S \text{ sadrži i pozitivne i negativne brojeve, a ne sadrži nulu,} \\
\alpha(S) &= \{-, 0, +\}, \text{ ako } S \text{ sadrži nulu i pozitivne i negativne brojeve.}
\end{aligned}
$$

$$
\begin{aligned}
\gamma(\emptyset) &= \emptyset, \\
\gamma(\{0\}) &= \{0\}, \\
\gamma(\{+\}) &= (0, +\infty), \\
\gamma(\{-\}) &= (-\infty, 0), \\
\gamma(\{0, +\}) &= [0, +\infty), \\
\gamma(\{-, 0\}) &= (-\infty, 0], \\
\gamma(\{-, +\}) &= (-\infty, 0) \ \cup \ (0, +\infty), \\
\gamma(\{-, 0, +\}) &= \mathbb{Z}.
\end{aligned}
$$

Primer 4. Parijalno uređeni skup podskupova skupa celih brojeva sa uređenjem inkluzije $(P(\mathbb{Z}), \subseteq)$ možemo apstrahovati u zatvorene intervale. U ovom slučaju skup koji je konkretan je $P(\mathbb{Z})$, a skup koji je apstraktni je potpuna rešetka intervala opisana u primeru 1. Galoaove veze su date sa:

$\alpha(S) = [\min S, \max S]$

(ako S nema najmanji, tj. najveći element, onda je $\min S = -\infty$, odnosno $\max S = +\infty$),

$\gamma([a, b]) = \{z \in \mathbb{Z} \mid a \leq z \leq b\}$

(ako je $a = -\infty$ nema donje granice, ako je $b = +\infty$ nema gornje granice)

3.4 Zabranjene zone i lažna upozorenja

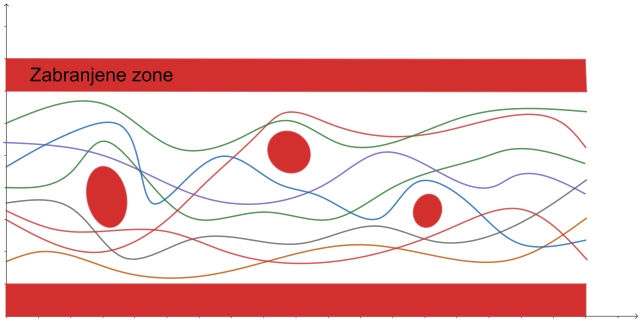
U kontekstu verifikacije, zabranjene zone (eng. *forbidden zones*) predstavljaju skupove stanja koja program ne sme da dostigne. U nastavku se u ilustracijama zabranjene zone označavaju crvenom bojom a apstraktna semantika zelenom. Ako program dospe u zabranjenu zonu znači da nije prošao verifikaciju odnosno da postoji problem. Neki od problema mogu biti neinicijalizovane promenljive, deljenje nulom, stanja koja ne zadovoljavaju specifikaciju programa (npr. negativni saldo na bankovnom računu) itd.

Sigurnosna svojstva programa izražavaju da ne postoji izvršavanje programa koje može da dovede do stanja greške. Na slici 3.13 dat je primer kako se sigurnoso svojstvo može grafički prikazati. Krivama je prikazan skup svih mogućih putanja programa. Ispravno izvršavanje programa podrazumeva da nijedno njegovo stanje ne ulazi u zabranjenu zonu. U ovom grafičkom prikazu to znači da bez obzira kojim putem se program izvršava, program ostaje van zabranjenih zona. Time se formalno ispunjava definicija sigurnosnog svojstva – ne postoji izvršavanje koje dovodi do stanja greške.

Posledica aproksimacije svih mogućih izvršavanja je da se razmatraju i neka nepostojeća izvršavanja. Takva izvršavanja, u zavisnosti od aproksimacije, mogu da vode do greške, odnosno dobija se lažno upozorenje o grešci koja se nikada neće desiti.

Lažna upozorenja odgovaraju slučajevima kada apstraktna semantika preseca zabranjenu zonu dok je konkretna semantika ne preseca. Dakle, signalizira se greška

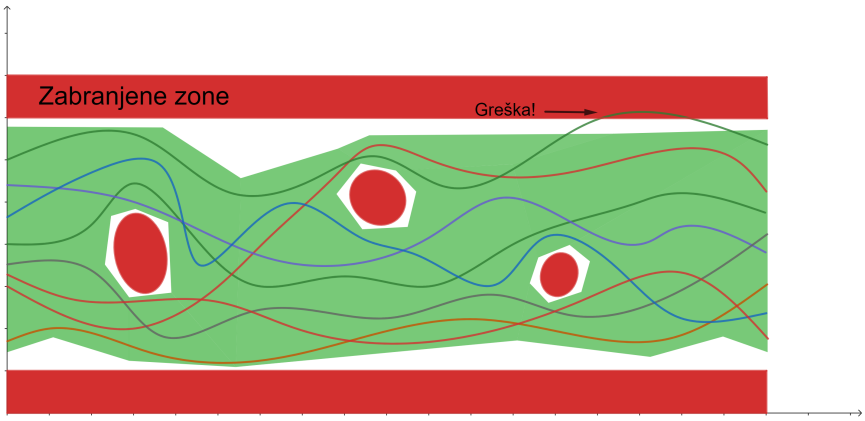
do koje se ne može stvarno doći. To zadaje izazov da apstrakcija pokrije sva moguća izvršavanja a da pritom ne bude prevelika kako bi se izbegla lažna upozorenja.



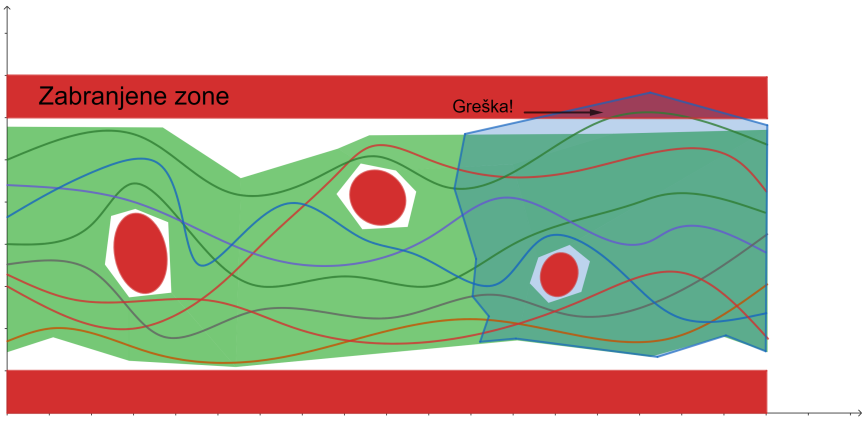
Slika 3.13: Grafički primer: Sigurnosno svojstvo

U nastavku su data potrebna svojstva svake apstraktne semantike i njihovi grafički prikazi [3]:

- nijedna eventualna greška se ne sme zaboraviti (na slikama 3.14 i 3.15 prikazan je primer apstrakcije koja ne obuhvata izvršavanje koje dovodi do greške i apstrakcija koja to ispravlja)

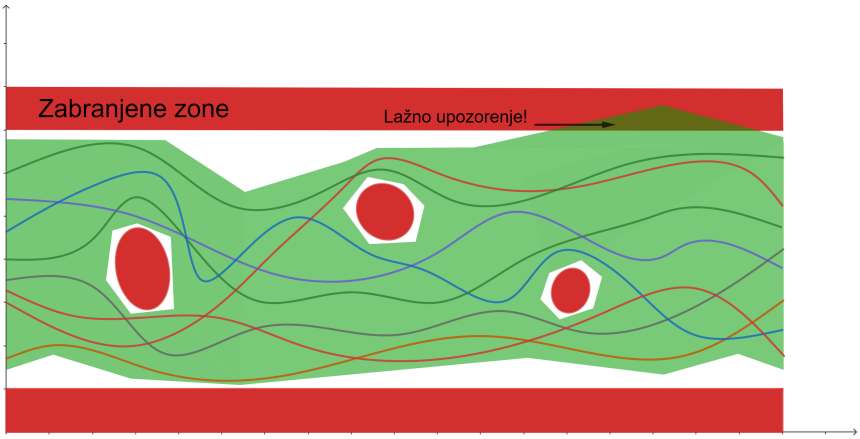


Slika 3.14: Grafički primer: Pogrešna apstrakcija



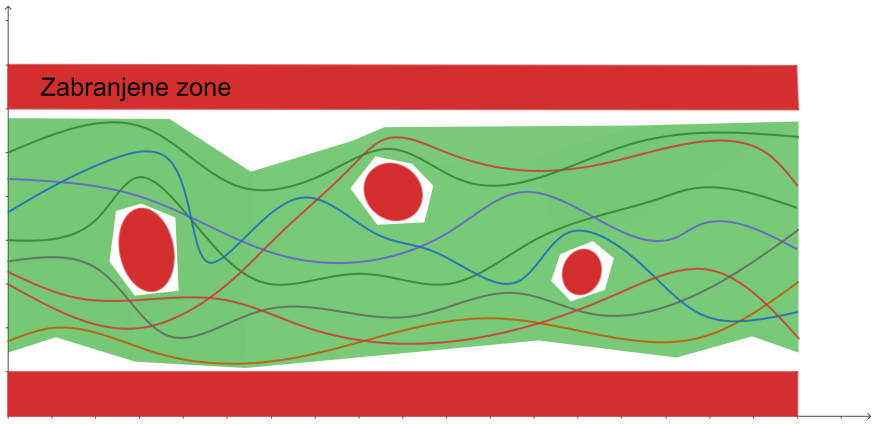
Slika 3.15: Grafički primer: Ispravljena apstrakcija primera sa slike 3.14

- preciznost (da bi se izbegla lažna upozorenja teži se što preciznijoj apstrakciji - primer loše preciznosti koja uzrokuje lažno upozorenje dat je na slici 3.16, dok je primer precizne apstrakcije dat na slici 3.17)



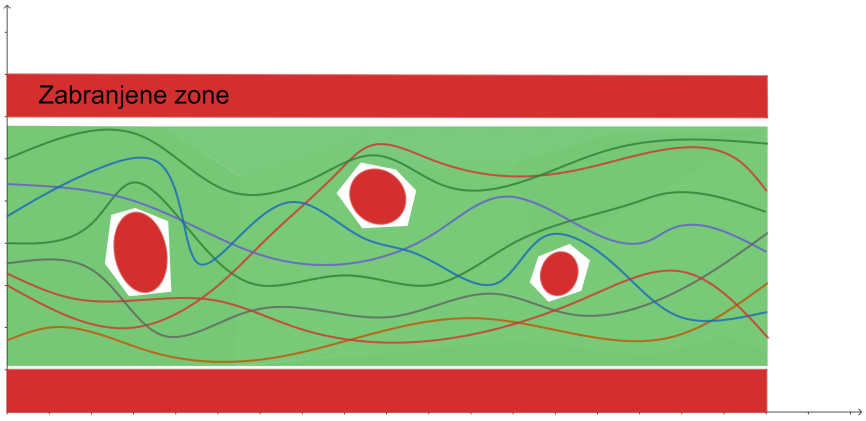
Slika 3.16: Grafički primer: Nepreciznost koja uzrokuje lažna upozorenja

- korektnost/saglasnost (asptraktne semantike su nadskup konkretne semantike - apstrakcija mora sadržati sve moguće putanje programa; u grafičkom smislu to znači da zelene površine moraju sadržati sve krive)
- jednostavnost (asptraktne semantike trebaju biti barem dovoljno jednostavne da mogu da se predstave u okviru mašine - primer najgrublje apstrakcije koja



Slika 3.17: Grafički primer: Precizna semantika

je precizna i dovoljno apstraktna dat je na slici 3.18 — uzimanje najgrublje apstrakcije omogućava jednostavniju implementaciju)



Slika 3.18: Grafički primer: Najgrublja ispravna apstraktna semantika

Glava 4

Određivanje opsega promenljivih

U okviru ovog rada implementirani su *Python* programi za rad sa 16-obitnim celim brojevima. Implementirani programi koriste tehniku apstraktne interpretacije kako bi se aproksimirale moguće vrednosti koje izrazi u programu mogu poprimiti tokom izvršavanja. Kao apsrtaktni domeni razmatraju se intervali vrednosti promenljivih.

Osnovna ideja se zasniva na praćenju dodela vrednosti i izračunavanju izraza, pridružujući nove intervale promenljivama, prema pravilima propagacije kroz aritmetičke operacije. Za dobijanje tačnih opsega izraza, neophodno je definisati pravila sabiranja, oduzimanja, množenja i deljenja nad intervalima (tabela 4.1).

Na primer, prilikom sabiranja promenljivih a i b, koje su iz intervala $[a_1, a_2]$ i $[b_1, b_2]$ redom, dobija se zbir koji pripada intervalu $[a_1 + b_1, a_2 + b_2]$, dok za množenje važi pravilo koje uzima minimum i maksimum među svim mogućim proizvodima krajeva intervala. Korišćenjem ovih pravila moguće je konstruisati apstraktni domen kroz koji se propagiraju intervali i za znatno složenije izraze.

Operacija	**Ulazni intervali**	**Rezultujući interval**
sabiranje $(+)$	$[a_1, a_2] + [b_1, b_2]$	$[a_1 + b_1,\ a_2 + b_2]$
oduzimanje $(-)$	$[a_1, a_2] - [b_1, b_2]$	$[a_1 - b_2,\ a_2 - b_1]$
množenje $(\cdot)$	$[a_1, a_2] \cdot [b_1, b_2]$	$[\min(P),\ \max(P)]$, gde je $P = \{a_i b_i \mid i, j \in \{1, 2\}\}$
deljenje $(\div)$	$[a_1, a_2] \div [b_1, b_2]$	definisano samo ako $0 \notin [b_1, b_2]$; koristi se $[\min(Q),\ \max(Q)]$ gde je $Q = \left\{\frac{a_i}{b_j} \mid i, j \in \{1, 2\}\right\}$
negacija $(-)$	$-[a_1, a_2]$	$[-a_2,\ -a_1]$

Tabela 4.1: Pravila za operacije nad intervalima

Ovakav pristup se može proširiti i na izraze koji sadrže više promenljivih i ugnežđene aritmetičke operacije. U tim slučajevima se redom, po prioritetu operacija, primenjuju pravila za operacije nad intervalima, a rezultati prethodnih koraka se koriste kao ulazi za sledeće.

U realnim aplikacijama, ova tehnika se pokazuje kao veoma korisna u raznim fazama verifikacije softverskih sistema koji moraju garantovati odsustvo grešaka izazvanih prekoračenjem, deljenja nulom ili izlaska van očekivanog opsega vrednosti. Primenom ovakve analize mogu se automatski detektovati granice ulaznih podataka koje vode ka nepoželjnim stanjima programa i sugerisati korekcije u kodu.

Implementirana rešenja je moguće interaktivno pokrenuti u *Jupyter* svesci čiji se kôd nalazi na linku:
https://github.com/kleutMilica/MasterRad/blob/main/Master.ipynb.

4.1 Apstrakcija intervalima za proveru prekoračenja

Programi se izvršavaju na mašinama sa ograničenim memorijskim resursima, koje ne mogu da podrže beskonačan skup vrednosti, za razliku od odgovarajućeg matematičkog modela. Prekoračenje nastaje kada rezultat izračunavanja nekog izraza premaši opseg vrednosti podržan za dati tip podataka. Takva situacija može dovesti do neočekivanih i nelogičnih rezultata izvršavanja programa. Programeri nekad zanemaruju činjenicu da su vrednosti u programu ograničene, što može da rezultuje propustima u rukovanju mogućim prekoračenjima. Zadatak verifikacije jeste da sistematski otkrije i spreči ovakve greške, kako bi se obezbedila pouzdanost softverskog sistema.

Prekoračenje u aritmetičkim izrazima

Kada u programu dođe do prekoračenja dozvoljene numeričke vrednosti za određeni tip podataka, ponašanje zavisi od programskog jezika i izvršnog okruženja. U jezicima kao što su `C` i `C++`, ovakve situacije često ne izazivaju grešku, već rezultiraju pogrešnim ishodom zbog takozvanog *wraparound* ponašanja. Do toga dolazi kada se rezultat, predstavljen kao niz bitova, ne može u potpunosti smestiti u predviđeni memorijski prostor, pa računar upisuje samo deo bitova za koji ima mesta. Takva binarna reprezentacija odgovara potpuno drugačijoj vrednosti od očekivane.

Nasuprot tome, savremeniji jezici i razvojni alati omogućavaju detekciju prekoračenja i generisanje odgovarajućih poruka o grešci, poput „`OverflowError: math range error`" ili „`ArithmeticException: integer overflow`". Iako ovakva poruka jasno ukazuje na uzrok problema, njen nastanak u fazi izvršavanja — naročito kod krajnjeg korisnika, a ne tokom testiranja — može predstavljati ozbiljan rizik po pouzdanost sistema.

Zapis celobrojnih vrednosti i primeri prekoračenja

Označeni brojevi se zapisuju u potpunom komplemetu — prvi je bit znaka (0 označava pozitivnu vrednost, 1 označava negativnu vrednost), a ostali bitovi služe za računanje vrednosti broja. Pozitivni brojevi se predstavljaju kao zbir stepena dvojke: $a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$ gde svaki od koeficienata a_i ima vrednost 0 ili 1 i to je vrednost bita na i-tom mestu u zapisu broja. Jasan je i drugi smer koji čita binaran zapis — zanemari se bit znaka (prvi bit) i zbirom stepena broja dva dolazimo do odgovarajuće dekadne vrednosti.

Da bi se dobio binaran zapis negativnog broja prvo se posmatra binarna reprezentacija apsolutne vrednosti tog broja. Zatim se svaki bit invertuje i na kraju se na rezultat doda 1 na mestu najmanje težine.

Primer 1. Ako se koristi 16-obitna celobrojna vrednost, prvi bit je bit znaka a ostalih 15 bitova služe za zapis vrednosti broja. To znači da najveći broj koji se može zapisati ima sledeći niz bitova 0111 1111 1111 1111.

Vrednost ovog broja u dekadnom sistemu je

$$2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 32767$$

Dakle, najveća pozitivna vrednost koja se može zapisati kao 16-obitna označena celobrojna vrednost je +32767.

Primer 2. Odredimo zapis broja -1. Apsolutna vrednost od -1 je 1, a binaran zapis broja 1 je 0000 0000 0000 00001. Kada se invertuje svaki bit dobija se 1111 1111 1111 1110. Kada tu vrednost binarno saberemo sa 1 dobija se 1111 1111 1111 1111.

Primer 3. Binarni zapis 1000 0000 0000 0000 na bitu znaka ima 1 što znači da se radi o negativnom broju. Odredimo koji je to dekadni broj koristeći potpuni komplemet. Kada se od binarno zapisanog broja 1000 0000 0000 0000 oduzme jedinica

dobija se 0111 1111 1111 1111. Kada se invertuje svaki bit dobija se niz bitova 1000 0000 0000 0000 koji dekadno ima vrednost $2^{15} = 32768$, i to je apsolutna vrednost traženog dekadnog broja. Dakle, zapis 1000 0000 0000 0000 je binarni zapis broja -32768. To je najmanja označena celobrojna vrednost koja se može zapisati pomoću 16 bitova.

Ukoliko se za zapis označenih celobrojnih vrednosti u računaru koristi 16 bitova, onda je raspon vrednosti koje se mogu zapisati od -32768 do 32767.

Primer 4. Saberimo dve pozitivne vrednosti: $20000 + 20000$ u okviru 16-obitnog označenog celobrojnog tipa. Najpre ćemo prikazati kako se broj 20000 zapisuje binarno. Broj 20000 možemo predstaviti kao zbir stepena dvojke:

$$20000 = 16384 + 2048 + 1024 + 512 + 32 = 2^{14} + 2^{11} + 2^{10} + 2^{9} + 2^{5}$$

Zato je binarni zapis 0100 1110 0010 0000.

Saberimo sada dve takve vrednosti:

$$\begin{array}{rl} 0100\ 1110\ 0010\ 0000 & (20000) \\ +\,0100\ 1110\ 0010\ 0000 & (20000) \\ \hline 1001\ 1100\ 0100\ 0000 & (\text{rezultat}) \end{array}$$

Rezultat počinje cifrom 1 što znači da je negativan.

Da bismo dobili dekadnu vrednost rezultata binarnog sabiranja najpre oduzimamo jedinicu čime se dobija vrednost 1001 1100 0011 1111. Invertujemo sve bitove i dobijamo: 0110 0011 1100 0000. Dekadno to predstavlja broj 25536. Pošto je broj negativan, konačan rezultat je: -25536.

Iako je $20000 + 20000 = 40000$, rezultat sabiranja u 16-obitnom sistemu je -25536, što je posledica prekoračenja (*overflow*), jer broj 40000 premašuje maksimalnu moguću vrednost za 16-obitni označeni celobrojni tip, tj. 32767.

Primer 5. Razmotrimo rezultat sledećeg izraza: $100 \cdot 10 + 31768$ u okviru 16-obitnog označenog celobrojnog tipa. Najpre ćemo izračunati rezultat izraza:

$$100 \cdot 10 + 31768 = 1000 + 31768 = 32768$$

Broj 32768 možemo predstaviti kao zbir stepena dvojke:

$$32768 = 2^{15}$$

Zato bi binarni zapis broja 32768 trebao biti jednak:

1000 0000 0000 0000

Međutim u 16-obitnom označenom celobrojnom tipu se koristi najnižih 15 bitova za vrednost broja, a najveći bit je bit znaka pa ovakva binarna reprezentacija ne predstavlja broj 32768. U primeru 3 je pokazano da je dekadna vrednost ovog binarnog zapisa -32768.

Dakle, iako je:

$$100 \cdot 10 + 31768 = 32768$$

u okviru 16-obitnog označenog celobrojnog tipa rezultat je -32768 što je posledica prekoračenja (*overflow*).

Apstraktna interpretacija je korisna za rano otkrivanje mogućih prekoračenja. Korišćenjem predefinisanih opsega vrednosti moguće je unapred proceniti da li će operacije nad promenljivim dovesti do grešaka.

Problem prekoračenja se javlja kada rezultat aritmetičke operacije nad numeričkim vrednostima premaši granice koje mogu biti predstavljene u okviru određenog tipa podatka u memoriji računara. U tabeli 4.2 prikazani su opsezi vrednosti za označene cele brojeve zapisane sa 16, 32 i 64 bita. U ovom radu problem se posmatra na primeru 16-obitnih označenih celih brojeva, ali slično važi i za primere brojeva koji se zapisuju sa više bitova.

Broj bitova	**Opseg (eksponencijalni oblik)**	**Opseg**
16	$[-2^{15},\ 2^{15}-1]$	$[-32768,\ 32767]$
32	$[-2^{31},\ 2^{31}-1]$	$[-2147483648,\ 2147483647]$
64	$[-2^{63},\ 2^{63}-1]$	$[-9223372036854775808,\ 9223372036854775807]$

Tabela 4.2: Opsezi vrednosti za označene celobrojne tipove zapisane sa 16, 32 i 64 bita

Zadatak je da se primenom apstraktne interpretacije detektuje i spreči ulazak u zabranjenu zonu tj. u oblasti izvan dozvoljenog intervala. Time se omogućava for-

malna verifikacija ispravnosti programa u pogledu aritmetičkog prekoračenja. Implementacija provere opsega dozvoljenih vrednosti promenljivih je značajna jer se može primeniti u gotovo svim softverskim rešenjima.

Primer 6. Prekoračenje pri množenju i sabiranju pozitivnih vrednosti testira scenario u kojem se dve pozitivne vrednosti množe, a zatim im se dodaje treća pozitivna vrednost. Ovaj eksperiment koristi unapred definisane vrednosti promenljivih $a = 100$, $b = 10$ i $c = 31768$, sa ciljem da se pokaže kako dolazi do prekoračenja kada rezultat premaši gornju granicu opsega dozvoljenog za 16-obitne celobrojne vrednosti (32767). Na ovaj način se simulira scenario kada se vrednosti koje prelaze dozvoljeni opseg interpretiraju kao potpuno drugačije binarne reprezentacije u memoriji.

Jednostavan program koji koristi tri 16-obitne celobrojne vrednosti od kojih se dve množe a zatim im se dodaje treća dat je u okviru listinga 4.1. Ovaj program demonstrira prekoračenje opisano u primeru 5. Koristi se *int16* za simulaciju 16-obitnog celog broja iz biblioteke *numpy*.

```
1   import numpy as np
2   def calculate(a, b, c):
3
4       a = np.int16(a)
5       b = np.int16(b)
6       c = np.int16(c)
7
8       result = a * b + c
9       return result
10
11  def main():
12      a = 100
13      b = 10
14      c = 31768
15
16      result = calculate(a, b, c)
17      print("Rezultat:", result)
18
19  if __name__ == "__main__":
20      main()
```

Listing 4.1: Primer prekoračenja u jeziku Python

Pokretanjem ovog programa dobija se izlaz:

Rezultat: -32768
Prvi indikator da je došlo do prekoračenja je negativan rezultat s obzirom da je rađeno množenje i sabiranje tri pozitivna broja.

Napomena: Rezultat usled prekoračenja ne mora obavezno biti negativan. Posmatrajmo isti izraz ali za drugačije ulazne vrednosti. Neka je $a = -30000$, $b = 2$ i $c = 5000$. Tada izraz ima vrednost -55000 dok je izlaz prethodnog programa za ove ulazne vrednosti 10536.

Primer 7. Provera prekoračenja sa zadatim domenom vrednosti uključuje korišćenje apstraktnog domena za vrednosti promenljivih a, b i c kako bi se proverilo da li može doći do prekoračenja pri odabiru vrednosti iz definisanih opsega. Apstraktni domen treba da odredi očekivane opsege vrednosti za promenljive, a program zatim proverava da li se njihovo množenje i sabiranje nalazi u bezbednom opsegu za 16-obitne celobrojne vrednosti.

Pretpostavimo da vrednost promenljive a pripada intervalu [1, 100], a vrednost promenljive b intervalu [1, 10]. Potrebno je odrediti dozvoljene vrednosti za pozitivnu celobrojnu promenljivu c za izraz $a \cdot b + c$.

Ukoliko nema informacije u kom se intervalu vrednost promenljive c kreće, može se uzeti što veći proizvoljan pozitivan interval, a zatim se pozvati funkcija za proveru izabranog apstraktnog intevarala — to je funkcija `check_overflow` koja je prikazana u listingu 4.2.

Ukoliko je uzet prevelik opseg vrednosti ispisuje se upozoravajuća poruka da može doći do prekoračenja. U kodu je uzet opseg $[1, 31768]$ koji će izazvati prekoračenje. Preporuka je isprobati u interaktivnom alatu i druge domene za c.

```
class AbstractDomain:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def check_overflow(self):
        max_a = max(self.a)
        max_b = max(self.b)
        max_c = max(self.c)

        product = max_a * max_b

        if product > 32767:
```

```
15            return True
16
17        if product + max_c > 32767:
18            return True
19
20        return False
21
22 def main():
23     a_domain = range(1, 101) # [1, 100]
24     b_domain = range(1, 11)  # [1, 10]
25     c_domain = range(1, 32768) # [1, 32767]
26
27     abstract_domain = AbstractDomain(a_domain, b_domain, c_domain)
28
29     if abstract_domain.check_overflow():
30         print("Upozorenje: Vrednosti mogu uzrokovati prekoracenje za 16-obitne
       cele brojeve!")
31     else:
32         print("Vrednosti su bezbedne za 16-obitne cele brojeve.")
33
34 if __name__ == "__main__":
35     main()
```

Listing 4.2: Primer provere izabranog apstraktnog intervala

Za ovako izabran opseg vrednosti, izlaz programa je:
Upozorenje: Vrednosti mogu uzrokovati prekoracenje za 16-obitne cele brojeve!

Apstraktni domen može biti pojednostavaljen ali to može predstavljati problem jer jednostavniji domen daje manje pojedinosti te može odgovoriti na manje pitanja i sa manjom preciznošću. U prethodnom primeru se pretpostavljalo koje vrednosti mogu uzeti promenljive a i b, i pretpostavljalo se da je c pozitivna vrednost. Ukoliko nemamo informaciju o opsezima za vrednosti promenljivih, moramo da pretpostavimo da sve tri vrednosti mogu biti bilo koji celi brojevi. Uvek je pogodnija prva apstrakcija — zadavanje konkretnih i što užih opsega za što veći broj promenljivih ali ako nema informacija o opsezima vrednosti mora se uzeti veći stepen apstrakcije.

Ako bi u primeru 7 bili izmenjeni domeni za a, b i c tako da za sve tri promenljive domen bude $[-32768, 32767]$ dobio bi se najširi opseg promenljivih koji zasigurno dovodi do prekoračenja. Kako promenljive mogu uzimati i negativne vrednosti najveća vrednosti izraza više nije $max_a \cdot max_b + max_c$ već se dobiija na osnovu tabele

4.1. Za date konkretne opesege, najveća vrednost izraza se izračunava na ovaj način: $min_a * min_b + max_c$.

Primer 8. Nasumična provera vrednosti sa normalnom raspodelom koristi normalnu (Gausovu) raspodelu za odabir vrednosti promenljivih a i b iz opsega $[-32768, 32767]$, kako bi se simulirale vrednosti koje su verovatno blizu nule. Na osnovu izabranih vrednosti a i b, računa se dozvoljeni opseg za promenljivu c. Ovi rezultati pokazuju fleksibilnost metode koja omogućava testiranje različitih scenarija na velikom skupu ulaznih podataka, uz generisanje različitih rezultata u svakom pokretanju.

Kôd prikazan u listingu 4.3 služi za generisanje različitih ulaznih vrednosti za promenljive a i b kako bi se za promenljivu c odredio interval kome sme da pripada tako da vrednost izraza $a \cdot b + c$ ne izađe iz granica dozvoljenih za 16-obitne cele brojeve.

Najpre se zadaje najširi opseg vrednosti za sve promenljive. Pošto se primeri prekoračenja u ovom radu razmatraju na 16-obitnim celim brojevima za sve tri promenljive opseg će biti $[-32768, 32767]$. Promenljive a i b se uzimaju kao slučajni brojevi iz tog intervala pri čemu se koristi normalna (Gausova) raspodela verovatnoće sa očekivanjem 0 (to znači da je veća verovatnoća da je slučajno izabrani broj blizu nule) i standardnom devijacijom 300 (standardnom devijacijom se meri koliko su podaci raspršeni oko očekivanja). Normalna raspodela je simetrična u odnosu na očekivanje, odnosno verovatnoća opada podjednako i sa leve i sa desne strane u odnosu na nulu i time se formira zvonast oblik krive, pri čemu sam oblik zavisi od standardne devijacije — što je devijacija veća, to je raspodela *šira* i *ravnija*; što je manja, raspodela je *uža* i *oštrija*.

Za vrednosti očekivanja i standardne devijacije zadate u ovom primeru većina podataka (oko 68%) biće u intervalu $[-300, +300]$, dok će oko 95% podataka biti u intervalu $[-600, +600]$, a gotovo svi podaci (99.7%) upadaju u interval $[-1200, +1200]$. Na slici 4.1 prikazana je normalna raspodela kada je očekivanje 0 i standardna devijacija 300.

Ovo je korisno koristiti kada nema informacija o tačnom opsegu vrednosti ali se može pretpostaviti da će se vrednost naći često u okolini neke tačke. U ovom kodu se normalna raspodela koristi iz demonstrativnog razloga kako bi se brže došlo do vrednosti koje zadovoljavaju uslove za uspešno izvršavanje operacija nad 16-obitnim celim brojevima. Dozvoljene vrednosti za c se računaju na osnovu izabranih a i b. Ovo znači da će se pri svakom pokretanju dobiti drugačije vrednosti promenljivih.

Na ovaj način se porogram može testirati na velikom skupu ulaznih vrednosi.

```
import numpy as np

class AbstractDomain:
    def __init__(self, a_range, b_range, c_range):
        self.a = int(np.random.normal(0, 300)) #normalna raspodela
        self.b = int(np.random.normal(0, 300)) #normalna raspodela
        self.c = c_range

    def check_valid_range(self, value):
        return -32768 <= value <= 32767

    def check_overflow(self):
        if not (self.check_valid_range(self.a) and
                self.check_valid_range(self.b)):
            print(f"Vrednosti a: {self.a}, b: {self.b}")
            raise ValueError("a i b moraju biti u opsegu 16-bitnog int")

        product = self.a * self.b
        if product > 32767 or product < -32768:
            print(f"Vrednosti a: {self.a}, b: {self.b}")
            return "Prekoracenje je moguce sa ovim vrednostima a i b"

        max_c_pos = 32767
        min_c_neg = -32768

        if product >= 0:
            max_c_pos = max_c_pos - product
            min_c_neg = min_c_neg

        else:
            max_c_pos = max_c_pos
            min_c_neg = min_c_neg - product

        print(f"Vrednosti a: {self.a}, b: {self.b}, c: [{min_c_neg},
        {max_c_pos}]")

        if max_c_pos < min(self.c) or max(self.c) < min_c_neg:
            return "Vrednosti za c mogu uzrokovati prekoracenje za
                    16-obitni int"
        return "Vrednosti su bezbedne za 16-obitni integer"

```

```python
def main():
    #Domeni za a, b i c
    a_domain = range(-32768, 32768)
    b_domain = range(-32768, 32768)
    c_domain = range(-32768, 32768)

    abstract_domain = AbstractDomain(a_domain, b_domain, c_domain)

    print(abstract_domain.check_overflow())

if __name__ == "__main__":
    main()
```

Listing 4.3: Primer proizvoljnih vrednosti promenljivih

Izlaz jednog od pokretanja kada dolazi do prekoračenja:

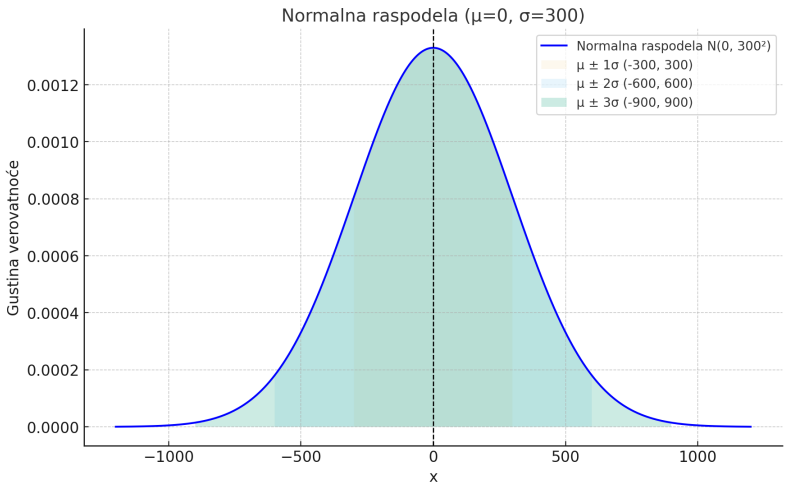
Vrednosti a: -13401, b: 31280

Prekoracenje je moguce sa ovim vrednostima a i b

Izlaz jednog od pokretanja kada ne dolazi do prekoračenja:

Vrednosti a: 137, b: 44, c: [-32768, 26739]

Vrednosti su bezbedne za 16-obitni integer



Slika 4.1: Normalna raspodela sa očekivanjem 0 i standardnom devijacijom 300

4.2 Apstrakcija intervalima za verifikaciju bankarskog softvera

Softver koji se koristi u bankarskom sektoru ima ključnu ulogu u obradi transakcija, obračunu kamata, izračunavanju rata i proveri kreditnih limita. Ovakvi sistemi moraju biti izuzetno pouzdani, jer čak i mala greška u implementaciji može dovesti do značajnih posledica – od nepravilnog obračuna troškova za korisnike, preko ozbiljnih finansijskih gubitaka za institucije, pa sve do narušavanja poverenja u čitav bankarski sistem.

Verifikacija i formalna provera ispravnosti bankarskog softvera predstavljaju važan deo procesa razvoja. Cilj je da se za program utvrdi njegova ispravnost ili da se već u ranoj fazi otkriju i isprave greške koje u realnom okruženju mogu dovesti do:

- pogrešno odobrenih ili odbijenih transakcija,
- neispravnog obračuna kamata i naknada,
- kršenja ugovorenih uslova između banke i klijenata,
- potencijalnih pravnih i reputacionih problema za instituciju.

U nastavku su dati primeri jednostavnog programa koji računa iznos za otplatu (dug) pri plaćanju kreditnom karticom. Prvobitno nije postojala nakanda za ovaj tip transakcije i program je radio ispravno, što je opisano u prvom primeru. Zatim se uvodi nakada za plaćanje kreditnom karticom. Drugi primer analizira scenario u kom je došlo do promene obračuna ukupnog iznosa za otplatu, uzimajući u obzir naknadu, dok je program ostao nepromenjen i detektuje propust posmatrajući apstraktne domene.

Primer 1. Neka je dat program koji računa iznos za otplatu na kreditnoj kartici za vrednosti u stranoj valuti. *Limit na kreditnoj kartici* je ograničen od strane banke u zavisnosti od primanja klijenta i najčešće je u visini jedne do dve plate klijenta. Za potrebe primera uzećemo limit po kreditnoj kartici od 76700 dinara. *Raspoloživo stanje* na kreditnoj kartici je limit umanjen za trenutno dugovanje po kreditnoj kartici. To znači da je raspoloživo stanje u opsegu $[0, 76700]$.

Plaćanje kreditnom karticom je moguće u stranoj valuti i ukupan iznos za otplatu u dinarima se dobija formulom:

$$iznos_za_otplatu = iznos \cdot kurs$$

Kurs za konverziju je ograničen od strane Narodne banke. U ovom primeru posmatra se plaćenje u eurima gde se za kurs uzima kao ograničenje interval $[115, 118]$.

Program proverava da li je vrednost iznosa za otplatu manja ili jednaka od raspoloživog stanja na kartici. Program napisan na ovaj način će da odbaci sve transakcije čiji je iznos u dinarima veći od raspoloživog stanja na kreditnoj kartici — što je ispravno.

Primer 2. Posmatrajmo isti program kao u prethodnom primeru. Neka je u bankarski proizvod kreditnih kartica izmenjen tako da se za svaku transakciju kreditnom karticom naplaćuje naknada. Naknada za plaćanje kreditnom karticom je propisana od strane banke koja izdaje kreditnu karticu ali je ograničena maksimalna naknada od strane Narodne banke i u primeru iznosi 5%.

Izmene u poslovnim pravilima se u praksi često dešavaju, a izmene ove vrste obično idu uz izvestan *grace period*[1]. Ako program ne bude izmenjen u skladu sa izmenom samog proizvoda doći će do neželjenog ponašanja, tj. do prihvatanja transakcija koje bi trebalo da ne budu odobrene. To se može otkriti posmatranjem apstraktnih domena.

Novi ukupni iznos za otplatu u dinarima prilikom plaćanja u evrima dobija se formulom:

$$iznos_za_otplatu = iznos \cdot kurs + naknada$$

pri čemu se naknada računa po formuli:

$$naknada = procenat_naknade \cdot iznos \cdot kurs$$

Ulazne vrednosti pripadaju sledećim intervalima:

$$\begin{aligned} \text{iznos} &\in [0, \infty] \\ \text{kurs} &\in [115, 118] \\ \text{procenat_naknade} &\in [0, 0.05] \\ \text{raspoloživo_stanje} &\in [0, 76700] \end{aligned}$$

Domeni izraza za transakcije koje treba da budu odobrene se računaju na sledeći način. Na osnovu programa, važi:

$$iznos \cdot kurs \leq \text{raspoloživo_stanje} \in [0, 76700]$$

[1] *Grace period* (eng.) označava odloženi period tokom kog obaveza, poput otplate ili obračuna kamate, privremeno ne stupa na snagu. U prevodu se koristi izraz „period počeka“.

S druge strane, ako se uzme u obzir naknada, važi:

$$iznos \cdot kurs \cdot (1 + procenat_naknade) \leq \text{raspoloživo_stanje} \in [0, 76700]$$

odnosno:

$$iznos \cdot kurs \in [0, 76700]/[1, 1.05] = [0, 73047]$$

Verifikacija proverava da li je svaki iznos transakcije u granicama raspoloživog stanja, tj. poredi intervale[2] za izraz $iznos{\cdot}kurs$ u programu i za izraz $iznos{\cdot}kurs$ kada se uzme u obzir naknada. Nejednakost nije zadovoljena jer interval $[0, 76700]$ nije manji od intervala $[0, 73047]$, odnosno program dozvoljava da vrednosti $iznos \cdot kurs$ budu veće nego što je to prihvatljivo.

Ukoliko se napisan program ne izmeni to dovodi do toga da će se neke transakcije pogrešno prihvatati iako prelaze stvarni limit. Time se jasno uočava da program ne zadovoljava sigurnosno svojstvo — ne odbacuje sve nedozvoljene transakcije koje bi zapravo trebalo da budu odbačene.

Provera deljenja nulom

Deljenje je *bezbedno* ako u svakoj instrukciji oblika $q = u/v$, imenilac v nije jednak nuli. U apstrakciji intervalima to se proverava tako što za domen imenioca $I(v)$ zahtevamo da važi $0 \notin I(v)$.

Napomena: Imenilac v može biti izraz. U tom slučaju deljenje je bezbedno ako nula ne pripada domenu izraza.

Primer 3. Za imenilac $v = x - 1$ bezbednost proveravamo uslovom:

$$0 \notin I(x-1) \iff 1 \notin I(x) \quad \text{(dodavanje 1 na obe strane).}$$

Ako je $x \in [a, b] \subseteq \mathbb{Z}$ deljenje je bezbedno za uslove:

$$b \leq 0 \quad \text{ili} \quad a \geq 2.$$

Dakle, ako je npr. $[a, b] = [5, 12]$ ili $[a, b] = [-7, 0]$, deljenje je bezbedno; ako je $[a, b] = [-3, 5]$, interval sadrži vrednost 1 te postoji opasnost deljenja nulom.

Opšti obrazac (pomeranje za konstantu):

[2]Intervali se porede tako sto se porede odgovarajući krajevi intervala međusobno

Za imenilac $x + c$ važi:

$$0 \notin I(x + c) \iff -c \notin I(x),$$

pa je za celobrojne intervale dovoljno da važi $b \leq -c - 1$ ili $a \geq -c + 1$.

Međutim, računanje domena promenljivih koje učestvuju u izrazu nije neophodno za proveru da li postoji deljenje nulom. Ono je potrebno samo za računanje dozvoljenih intervala za ulazne promenljive. Naime, za proveru da li izraz može da dovede do deljenja nulom dovoljno je utvrditi domen samog izraza koji predstavlja imenilac i proveriti da nula nije u tom domenu. Na primer, za deljenje gde je imenilac $a + b$ pri čemu $a \in [-1, 100]$ i $b \in [5, 50]$ važi da je $a + b \in [4, 150]$ pa je deljenje bezbedno. S druge strane, za isti imenilac i intervale $a \in [-10, 100]$ i $b \in [5, 50]$ važi da je $a + b \in [-5, 150]$ i deljenje nije bezbedno.

Česte greške koje vode deljenju nulom su neprovereni brojači, neinicijalizovane ili resetovane vrednosti na 0, ulazi iz spoljašnjeg sistema za koje nedostaje preduslov.

Primer 4. U sektoru za servisiranje kredita računa se iznos mesečne obaveze:

$$\texttt{mesecna_obaveza} = \frac{\texttt{preostali_dug}}{\texttt{preostali_meseci}}.$$

Najveći iznos kredita koji banka nudi je 50 000 000 dinara. Najduži period otplate kredita je 30 godina, odnosno 360 meseci. Poslovno pravilo garantuje da preostali meseci moraju da budu veći od nule, jer se ovaj kôd izvršava samo za aktivne kredite kod kojih važi `preostali_meseci` ≥ 1. Intervali neposredno pre deljenja su:

$$I(preostali_dug) = [0,\ 50\,000\,000], \quad I(preostali_meseci) = [1,\ 360].$$

Pošto važi $0 \notin [1, 360]$, deljenje je bezbedno ($0 \notin I(v)$).

Ovo je primer dobro napisanog koda, gde je mesečna obaveza računa samo pod uslovom da je broj preostalih meseci veći od 1. Imenilac ima interval bez nule, zato je uslov $0 \notin I(v)$ zadovoljen i program prolazi verifikaciju. Međutim, ukoliko poslovno pravilo koje garantuje da se izračunavanje vrši samo za aktivne kredite ne bi bilo implementirano, važilo bi da je $I(preostali_meseci) = [0,\ 360]$ i deljenje nulom bi bilo moguće.

Posmatrajmo situaciju u kojoj postoji opasnost od nebezbednog deljenja i uočavanje problema posmatranjem intervala.

Primer 5. U dnevnom bankarskom izveštaju računa se prosečan iznos po transakciji:

$$\texttt{prosek} = \frac{\texttt{ukupno_isplaceno}}{\texttt{broj_transakcija}}.$$

Tokom prazničnog režima, ili u nekim vanrednim situacijama, može se desiti da tog dana ne bude transakcija. Gornje granice za ukupno isplaćen iznos i broj transakcija su vrednosti koje je definisala Narodna banka na osnovu dosadašnjih statistika, na primer, neka je gornja granica za ukupno isplaćen iznos 2 000 000 000, a gornja granica za broj transakcija 200 000.

Intervali neposredno pre deljenja su:

$$I(\texttt{ukupno_isplaceno}) = [0,\ 2\,000\,000\,000], \quad I(\texttt{broj_transakcija}) = [0,\ 200\,000].$$

Pošto važi $0 \in [0, 200\,000]$, može se desiti greška uzrokovana deljenjem sa nulom.

Ova program ne prolazi verifikaciju te je potrebno otkloniti problem. Jedno od jednostavnih rešenja je uvođenje grananja — ako je broj transakcija 0, tada je i prosek nula inače se računa prosek po navedenoj formuli.

Primer 6. Prosečno ponderisana kamatna stopa na određenu kategoriju proizvoda (kredita ili depozita) predstavlja prosečnu kamatna stopu za tu vrstu proizvoda, pri čemu se ne radi o prostom proseku kamatnih stopa, već se uzima u obzir i efekat iznosa kredita ili depozita na koju se određena kamatna stopa primenjuje. Odnosno, u obračunu prosečne kamatne stope daje veći značaj onim kamatnim stopama koje se primenjuju na veći iznos kredita ili depozita.

Prosečno ponderisane kamatne stope (u nastavku PPKS) računaju se na kratkoročne i dugoročne kredite ili depozite prema sledećoj formuli:

$$PPKS \ = \ \frac{\sum_i (kamatna_stopa_i \times iznos_kredita_i)}{\sum_i iznos_kredita_i} \times 100.$$

Program koji računa i ispisuje PPKS za konkretne primere iznosa i kamatnih stopa kredita dat je u listingu 4.4. Imenilac je zbir iznosa kredita. Pretpostavimo da je gornja granica za ukupno isplaćen iznos 2 000 000 000 i neka je ukupan broj kredita 100 000. Kako su iznosi kredita uvek nenegativni brojevi, domen imenioca je $[0, 100\,000 \times 2\,000\,000\,000] = [0, 200\,000\,000\,000\,000]$. Nula pripada domenu pa apstraktna interpretacija na osnovu domena registruje da postoji opasnost deljenja sa nulom — prijavljuje se nebezbedno stanje.

U praksi zbir iznosa kredita može biti nula u sledećim situacijama:

- Prazan skup nakon filtriranja — ukoliko se računa PPKS za zadati datum može se desiti da nema aktivnih kredita.
- Svi iznosi su nula — krediti su zatvoreni u posmatranom preseku.

Ukoliko se u kodu pre računanja proverava da je ukupan iznos strogo pozitivan, do ovog stanja se nikada neće doći i u tom slučaju apstraktna interpretacija daje lažno upozorenje (Listing 4.4).

```
# Lista kredita: (iznos, kamatna_stopa)
krediti = [
    (100000, 0.05),
    (200000, 0.045),
    (150000, 0.052),
]

ukupan_iznos = sum(iznos for iznos, _ in krediti)
if ukupan_iznos == 0:
    print("Nema aktivnih kredita - ne moze se izracunati ponderisana stopa.")
else:
    ponderisana = sum(iznos * stopa for iznos, stopa in krediti) / ukupan_iznos *
     100
    print(f"Ponderisana kamatna stopa: {ponderisana:.3f}%")
```

Listing 4.4: Ponderisana kamatna stopa uz zaštitu od deljenja nulom

Glava 5

Otkrivanje trke za resursima

Postoje različiti načini organizacije izvršavanja programa. U konkurentnim programima se više procesa ili niti, koji imaju zajednički cilj, izvršava u istom vremenskom periodu. Konkurentno programiranje podrazumeva logičku podelu posla, tj. primenu tehnika i metoda koje omogućavaju da se više procesa ili niti izvršava naizmenično ili istovremeno u okviru jednog programa. Procesi su međusobno nezavisni i koriste sopstveni adresni prostor, dok su niti tzv. *laki procesi*, koji dele isti adresni prostor unutar procesa. Jedan od najčešćih načina ostvarivanja konkurentnosti jeste upravo korišćenje niti [7].

Paralelno izvršavanje je specijalni slučaj konkurentnog izvršavanja koje podrazumeva postojanje fizičke, tj. hardverske podrške za podelu posla. Kod paralelnog izvršavanja, zadatak se zaista istovremeno izvršava na više procesora ili jezgara, pri čemu procesi mogu da komuniciraju putem zajedničke memorije.

5.1 Trka za resursima i sinhronizacija

Ideja deljenja posla radi bržeg izvršavanja deluje intuitivno, ali u praksi konkurentno programiranje često nailazi na različite prepreke. Osnovni problem koji se javlja je *trka za resursima*. Trka za resursima dešava se kada dve ili više niti pokušavaju istovremeno da pristupe i modifikuju iste podatke. To može dovesti do nepredvidivih i nekonzistentnih rezultata.

Primer 1. Razmotrimo scenario prodaje ulaznica putem internet aplikacije. Pretpostavimo da je u sistemu ostala još samo jedna ulaznica za određeni događaj. Dva korisnika istovremeno iniciraju zahtev za kupovinu poslednje ulaznice.

Aplikacija koristi zajedničku promenljivu koja čuva broj raspoloživih ulaznica. Oba procesa (koja predstavljaju korisničke zahteve) u isto vreme pristupaju toj promenljivoj i očitavaju vrednost 1. Prvi proces smanjuje vrednost na 0 i dodeljuje ulaznicu korisniku. Međutim, drugi proces, koji je prethodno očitao istu vrednost 1, takođe pokušava da smanji broj ulaznica i da izvrši rezervaciju.

Rezultat je stanje u kojem aplikacija beleži prodaju dve ulaznice, iako je u sistemu realno postojala samo jedna. Ovakav ishod je posledica *nedeterminističkog međusobnog preplitanja operacija čitanja i pisanja nad deljenim resursom*, što je klasičan primer trke za resursima.

Nekontrolisanje trke za resursima u praksi dovodi do narušavanja konzistentnosti podataka i ozbiljnih grešaka u poslovnoj logici sistema.

Kao što je prikazano u prethodnom primeru, trke za resursima nastaju kada različite niti *modifikuju* zajedničku promenljivu. Međutim, u situacijama kada niti nad promenljivom vrše samo *operacije čitanja*, nema potrebe uvoditi kontrolne mehanizme. U tim slučajevima konkurentno izvršavanje ne narušava konzistentnost podataka, jer sama vrednost promenljive ostaje nepromenjena.

Primer 2. Veb servis *Google Weather* prikazuje trenutnu vrednost temperature za izabranu lokaciju. Više niti istovremeno pristupa deljenoj promenljivoj koja čuva poslednju preuzetu vrednost iz pozadinskog servisa; niti ovu promenljivu isključivo čitaju. Ažuriranje ove promenljive obavlja se periodično u posebnoj, odvojenoj proceduri i van ovog toka obrade zahteva.

Pošto niti ne menjaju vrednost promenljive, već je samo čitaju, u ovom slučaju ne postoji opasnost od trke za resursima. Svi korisnici dobijaju konzistentne informacije.

Za sprečavanje trke za resursima korisi se *sinhronizacija*. Pre uvođenja mehanizama sinhronizacije neophodno je identifikovati promenljive nad kojima se mogu dogoditi trke za resursima, jer upravo one predstavljaju kritične tačke u programu.

Definicija 1. Kritična sekcija
Kritična sekcija (eng. *critical section* — u nastavku teksta CS) obuhvata operacije koje pristupaju deljenim resursima (npr. promenljivama, fajlovima, ure-

đajima). Ako bi više niti istovremeno pristupalo i menjalo deljene resurse, došlo bi do nekonzistentnosti podataka ili grešaka.

Definicija 2. Sinhronizacija
Sinhronizacija predstavlja skup tehnika i mehanizama koji omogućavaju pravilno i predvidivo izvršavanje više niti ili procesa nad zajedničkim podacima.

Kako bi se sprečili problemi koji nastaju usled konkurentnog pristupa deljenim resursima, koriste se različiti mehanizmi sinhronizacije.

Definicija 3. Ekskluzivni pristup
Ekskluzivni pristup označava mehanizam kojim se obezbeđuje da u datom trenutku samo jedna nit ili proces može pristupiti kritičnom resursu ili izvršavati kritični deo koda. Na taj način se sprečava međusobno preplitanje operacija čitanja i pisanja nad deljenim podacima, čime se eliminiše mogućnost pojave trke za resursima.

U praksi se ekskluzivni pristup ostvaruje pomoću različitih konstrukata sinhronizacije, među kojima se izdvajaju:

- **Katanac (eng. *lock*)** - najjednostavniji mehanizam sinhronizacije kojim se osigurava da samo jedna nit može pristupiti kritičnom resursu ili izvršavati kritični deo koda dok je katanac zaključan. Ostale niti moraju sačekati da se katanac oslobodi.
- **Muteks (eng. *mutex*)** - poseban oblik katanca koji dodatno garantuje da samo nit koja je zaključala muteks može da ga otključa. Time se povećava bezbednost i sprečavaju nekonzistentna stanja.
- **Semafor (eng. *semaphore*)** - najprimitivniji mehanizam sinhronizacije koji omogućava kontrolu pristupa nad ograničenim brojem resursa. Za razliku od muteksa i katanca, semafor može dozvoliti da više niti istovremeno pristupi resursu, ali njihov broj ne može preći unapred definisanu granicu.

Uloga sinhronizacije je da spreči pojavu trke za resursima pri čemu može uzrokovati druge probleme konkurentnog programiranja ukoliko se ne uradi na ispravan način:

Mrtve petlje nastaju kada dve ili više niti čekaju jedna na drugu da oslobode resurse, stvarajući cikličnu zavisnost koja sprečava nastavak izvršavanja. Na primer, ako nit A drži resurs R1 i čeka na resurs R2 koji drži nit B, dok nit B čeka na resurs R1, obe niti će biti blokirane.

Žive petlje su slične mrtvim petljama, ali niti ne stoje u mestu već nastavljaju sa radom, konstantno menjajući svoje stanje bez postizanja korisnog rezultata. To može dovesti do stanja gde niti stalno pokušavaju da reše međusobne zavisnosti, ali bez uspeha.

Izgladnjivanje se pojavljuje kada jedna ili više niti ne dobijaju pristup potrebnim resursima jer ih druge niti stalno preuzimaju. Ovo može dovesti do toga da određene niti nikada ne budu izvršene ili da izvršenje bude značajno odloženo.

Razumevanje svih ovih mehanizama od suštinskog je značaja za pravljanje konkurentnih programa, kako bi se obezbedilo konzistentno stanje sistema. U narednom podpoglavlju 5.2 biće prikazano kako se situacije koje dovode do trke za resursima detektuju putem apstraktne interpretacije, tj. analizom pristupa deljenim promenljivama tokom konkurentnog izvršavanja. Izbor odgovarajućeg alata za posmatranje ovakvih problema predstavlja ključan korak, jer može značajno olakšati njihovu analizu, razumevanje i rešavanje.

Primer 3. Da bi se sprečila trka za resursima, koristi se mehanizam ekskluzivnog pristupa, često implementiran pomoću zaključavanja. Na taj način se obezbeđuje da u svakom trenutku samo jedna nit ima pravo pristupa kritičnom delu koda. Listing 5.1 u programskom jeziku *Python* ilustruje ovu tehniku:

```
import threading

x = 0  # zajednicka promenljiva
lock = threading.Lock()

def increment():
    global x
    for _ in range(100000):
        with lock:   # samo jedna nit u kriticnom delu
            x += 1
```

```
11
12 t1 = threading.Thread(target=increment)
13 t2 = threading.Thread(target=increment)
14
15 t1.start()
16 t2.start()
17 t1.join()
18 t2.join()
19
20 print("Konacna vrednost:", x)
```

Listing 5.1: Korišćenje mehanizma `Lock` u Python-u

U prikazanom kodu koristi se biblioteka `threading` za rad sa nitima u *Python*-u. Naredba `lock = threading.Lock()` kreira katanac kojim se obezbeđuje ekskluzivan pristup kritičnom delu koda, dok konstrukcija `with lock:` garantuje da u svakom trenutku samo jedna nit može izvršavati naredbu `x += 1`. Linije 12 i 13 kreiraju dve niti koje pozivaju istu funkciju `increment`, a poziv metode `t1.join()` (analogno i za `t2.join()`) osigurava da se glavna nit programa ne nastavi sa radom dok pomoćne niti ne završe svoje izvršavanje.

U ovom primeru, zahvaljujući primeni zaključavanja, niti ne mogu istovremeno pristupiti i modifikovati promenljivu x. Konačna vrednost je uvek ispravna i jednaka 200000, jer je eliminisana mogućnost trke za resursima.

5.2 Primeri verifikacije jednostavnih konkurentnih programa

Implementacija apstraktne interpretacije za konkurentne programe je korisna za detekciju problema u konkurentnom programiranju. Primeri koji će biti prikazani demonstriraju kako se apstraktna interpretacija može koristiti za identifikaciju potencijalnih trka za resursima. Analiza prati pristupe deljenim promenljivama, kao i pristupe kritičnim delovima koda.

U nastavku je dat kôd koji implementira analizator. Uvođenjem analizatora postiže se apstrakcija koja omogućava praćenje pristipa promenljivama iz različitih niti, prepoznajući da li pristupi mogu dovesti do trke za resursima. Za svaku nit se zadaje lista torki koje sadrže naziv promenljive kojoj nit pristupa (`var`), vrstu pristupa (pristup radi čitanja — `READ` ili pristup radi pisanja — `WRITE`) i ime kri-

tične sekcije (`cs_name`, to je zapravo ime ključa, pri čemu ukoliko promenljiva nije u kritičnoj sekciji - odnosno ne zaključava se, ima oznaku `None`):

$$(\texttt{var},\ \texttt{READ/WRITE},\ \texttt{cs_name} \in \{\text{ime CS},\ \texttt{None}\}).$$

Za tako zadate niti pokreće se analizator. On prvo konstruiše rečnike sa podrazumevanim vrednostima (eng. *defaultdict*)[1]:

- rečnik `R` — svaku promenljivu slika u skup niti koje je čitaju
- rečnik `W` — svaku promenljivu slika u skup niti koje je koriste za pisanje
- rečnik `CS` — svaku kombinaciju $(promenljiva, nit)$ slika u skup imena kritičnih sekcija u kojima nit pristupa toj promenljivoj
- rečnik `UNPROT` — beleži nezaštićene pristupe[2]; svaku kombinaciju $(promenljiva, nit)$ slika u istinitnosne vrednosoti $true$ ili $false$ koje su odgovor na pitanje — *da li postoji nezaštićen pristup promenljivoj?*

Detekcija trke se dešava kada u listi niti koje pristupaju istoj promenljivoj postoje različite operacije nad promenljivoj odnosno postoji bar jedna `WRITE` nit. Ukoliko postoji bar jedna takva nit postoji i potencijalna trka za resursima. Da li je zaista došlo do trke proveravamo bezbednosnim mehanizmima definisanim rečnikom `UNPORT`. Potrebno je da je promenljiva zaštićena prilikom pristupa niti koje su u potencijalnoj trci za resurse, ali i da su joj niti pristupile u okviru istoimene kritične sekcije.

Napomena: Nema pokretanja niti; radi se isključivo nad apstraktnim opisom niti.

```
1  from collections import defaultdict
2
3  READ, WRITE = "READ", "WRITE"
4
5  def analyze_static(threads):
6      """
7      threads: dict -> thread_name : list of (var, READ/WRITE, cs_name or None)
8      Vraca sortiranu listu promenljivih nad kojima postoji potencijalna trka.
9      Strogi kriterijum 'nema trke':
10       - nijedna od dve niti nema nezasticene pristupe toj promenljivoj
```

[1] **defaultdict** - struktura podataka u *Python*-u koja sadrži parove ključ-vrednost pri čemu se ključu dodeljuje podrazumevana vrednost u slučaju kada ne postoji eksplicitna dodela

[2] evidentira da li je neka nit pristupala promenljivoj van CS

```
11        - svaka nit koristi TACNO jedan lock za tu promenljivu
12        - oba lock imena su identicna (ista zajednicka CS)
13    """
14    R = defaultdict(set)                        # var -> set(threads that READ)
15    W = defaultdict(set)                        # var -> set(threads that WRITE)
16    CS = defaultdict(lambda: defaultdict(set)) # var -> thread -> set(cs_names)
17    UNPROT = defaultdict(lambda: defaultdict(bool))  # var -> thread -> bool (ima
       li nezasticen pristup)
18
19    # Popunjavanje recnika po promenljivoj i niti
20    for t, ops in threads.items():
21        for (var, op, cs_name) in ops:
22            (R if op == READ else W)[var].add(t)
23            if cs_name is None:
24                UNPROT[var][t] = True
25            else:
26                CS[var][t].add(cs_name)
27
28    races = set()
29    for var in set(R) | set(W):
30        ts = list(R[var] | W[var])  # sve niti koje pristupaju var
31        for i in range(len(ts)):
32            for j in range(i + 1, len(ts)):
33                a, b = ts[i], ts[j]
34                # bar jedna nit pise?
35                writes = (a in W[var]) or (b in W[var])
36                if not writes:
37                    continue
38                # bezbednosne provere pristupa promenljivoj
39                same_cs = (
40                    not UNPROT[var][a] and
41                    not UNPROT[var][b] and
42                    len(CS[var][a]) == 1 and
43                    len(CS[var][b]) == 1 and
44                    CS[var][a] == CS[var][b]
45                )
46                if not same_cs:
47                    races.add(var)
48
49    return sorted(races)
```

Listing 5.2: Statička detekcija trke za resursima

Opisana apstrakcija je dovoljno dobra da može da potvrdi da trke za resursima nema, odnosno da može da detektuje sve slučajeve trke za resursima i ukaže na nedostatak sinhronizacije. Međutim, moguća je i pojava lažnih upozorenja.

Praćenje *write* pristupa bez sinhronizacije i sa sinhronizacijom

Apstraktna interpretacija beleži pristupe svake niti svakoj promenljivoj. Kandidat za potencijalnu trku je par niti od kojih bar jedna vrši upis nad istom promenljivom. Ukoliko postoje kandidati za trku proverava se da li su u okviru njih definisana pravila za bezbedno korišćenje promenljive, tačnije da li je urađena adekvatna sinhronizacija.

Primer 1. Neka su date dve niti koje pristupaju deljenim promenljivama x i y. Nad promenljivom x obe niti vrše operaciju upisa (*write*) pri čemu ne koriste sinhronizaciju. Nad promenljivom y takođe vrše operaciju upisa (*write*) pri čemu koriste sinhronizaciju. Svaka od niti je tada određena listom torki (`var`, `READ/WRITE`, `cs_name`), prikazano u listingu 5.3.

```
threads = {
    "T1": [("x", WRITE, None), ("y", WRITE, "CS_y")],
    "T2": [("x", WRITE, None), ("y", WRITE, "CS_y")],
}
print(analyze_static(threads))  # Ocekivano: ['x']
```

Listing 5.3: Dve niti upisuju u x (bez sinhronizacije) i y (sa sinhronizacijom)

Zaključak: Postoji potencijalna trka nad `x` jer različite niti vrše *write* bez odgovarajuće sinhronizacije.

Apstrakcija opisana primerom 1 se može primeniti u mnogim programima. Jedan primer iz realne upotrebe može biti jednostavna igrica čija je analiza data u nastavku, sa fokususom na *write-write* situaciju (analogno se radi analiza za *read-write* slučaj).

Primer 2. Neka je data igrica koju istovremeno igraju dva igrača u paru. Cilj igrice je da igrači sarađuju kako bi skupili u zajedničku kasu 100 novčiča izbegavajući prepreke. Sem zajedničke kase, igrači imaju i zajednički broj života. Kada neki od igrača udari u prepreku broj zajedničkih života se smanjuje za jedan. Neka je promenljiva x iz prethodnog primera stanje zajedničke kase. Neka je promenljiva y

iz prethodnog primera broj preostalih života. Posledica trke nad promenjivom x je nekonzistentno stanje kase. U situaciji kada oba igrača istovremeno osvoje po jedan novčič, umesto da se stanje kase uveća za dva, uvećava se za jedan. Ovakvo ponašanje programa usporava igračima put do pobede. Broj života se ispravno umanjuje, te igrači *lako* gube život a *teško* skupljaju novčiće.

Druga nebezbedna situacija je kada jedna nit povećava stanje kase sa 99 na 100 novčića, dok druga u tom trenutku proverava da li je zadovoljen uslov `x < TARGET_COINS` i čitajući vrednost 99, umesto 100, rešava da nastavi izvršavanje.

Pseudokod igrice prikazan je u listingu 5.4.

```
1  global TARGET_COINS := 100
2  global INITIAL_LIVES := 5
3
4  global x := 0                 // zajednicka kasa
5  global y := INITIAL_LIVES     // zajednicki broj zivota
6  global CS_y                   // zajednicki kljuc (lock) za y
7
8  procedure Player():
9      while x < TARGET_COINS and y > 0 do
10         if player collect coin
11             // skupljanje novcica (moguca trka: WRITE(x) bez CS)
12             x := x + 1
13         else
14             // udarac u prepreku: WRITE(y) pod CS_y (bezbedno)
15             with CS_y do
16                 if y > 0 then
17                     y := y - 1
18                 end if
19             end with
20         end if
21     end while
22
23 procedure Player1():
24     Player()
25
26 procedure Player2():
27     Player()
28
```

```
run Player1, Player2 in parallel  // dva igraca istovremeno igraju
```

Listing 5.4: Pseudokod igrice

Da bi analizatorom mogla da se detektuje trka za resursima potrebno je da se program apstrahuje, tačnije da se posmatranjem programa, niti predstave kao liste odgovarajučih torki. Koriste se sledeći koraci kako bi se došlo do apstrakcije:

1. **Identifikacija niti:** iz pseudokoda se vidi da postoje dve niti `Player1` i `Player2` (dva igrača).

2. **Identifikacija deljenih promenljivih:** `x` predstavlja zajedničku kasu, a `y` zajednički broj života; obe promenljive dele oba igrača.

3. **Klasifikacija pristupa:** oba igrača rade *WRITE* nad `x` i *WRITE* nad `y` (slično važi za čitanja uslova petlje `x < TARGET_COINS` i `y > 0` gde se evidentiraju *READ* pristupi).

4. **Praćenje sinhronizacije:**

   - Upisi u `x`: dešavaju se *van* kritične sekcije ⇒ `cs_name = None`.
   - Umanjenja `y`: dešavaju se *unutar* iste zajedničke kritične sekcije ⇒ `cs_name = ,,CS_y''`.

5. **Formiranje apstraktnog plana** (po niti: lista trojki (`var`, `READ/WRITE`, `cs_name`)):

```
threads = {
    "Player1": [("x", WRITE, None), ("y", WRITE, "CS_y")],
    "Player2": [("x", WRITE, None), ("y", WRITE, "CS_y")],
}
```

Listing 5.5: Apstraktni plan *write* pristupa za igricu

6. **Analiza:** prosleđivanje plana statičkom analizatoru (`analyze_static(threads)`).

   Očekivano: detektuje se potencijalna trka nad `x`.

Ukoliko bi se razmatralno više od dve niti, na primer n niti, važilo bi slično: od ukupnih n niti neke će vršiti naredbu pisanja i njih nazivamo *pisači* a broj takvih niti označavamo sa w; niti koje vrše naredbu čitanja nazivamo *čitači* i broj takvih niti

n	**w**	**r**	Obrazac pristupa	Trka bez CS	Trka sa CS
≥ 1	0	n	svi *read* (nema upisa)	Ne	Ne
≥ 1	1	0	jedan pisač, ostale ne pristupaju	Ne	Ne
≥ 2	1	≥ 1	*read–write* preplitanje	Da	Ne
≥ 2	≥ 2	≥ 0	*write–write* preplitanje	Da	Ne
≥ 2	0	≥ 2	*read–read*	Ne	Ne

Tabela 5.1: Uslovi za potencijalnu trku nad jednom deljenom promenljivom za n niti

označavamo sa r. Važi da je ukupan broj niti zbir pisača i čitača: $n = w + r$. U tabeli 5.2 je detaljno prikazano ponašanje pristupa n niti jednoj deljenoj promenljivoj.

Ukoliko postoje kandidati za trku proverava se da li su u okviru njih definisana pravila za bezbedno korišćenje promenljive, tačnije da li je urađena adekvatna sinhronizacija. To podrazumeva:

- promenljiva je obavezno pod ključem ako joj neka nit pristupa u tom trenutku,
- uvek se koristi isti ključ kada jedna ista nit koristi jednu istu promenljivu[3],
- koristi se zajednicki ključ za obe niti.

Primer 3. Neka su date dve niti koje pristupaju deljenim promenljivama x i y. Nad obe promenljive niti vrše operaciju upisa (*write*) pri čemu koriste sinhronizaciju. Svaka od niti je tada odredjena listom torki (`var`, `READ/WRITE`, `cs_name`), prikazano u listingu 5.6.

```
threads = {
    "T1": [("x", WRITE, "CS_x"), ("y", WRITE, "CS_y")],
    "T2": [("x", WRITE, "CS_x"), ("y", WRITE, "CS_y")],
}
print(analyze_static(threads))  # Ocekivano: []
```

Listing 5.6: Dve niti upisuju u x i y (sa sinhronizacijom)

Zaključak: Svi pristupi nad `x` iz obe niti dešavaju se unutar iste kritične sekcije (`CS_x`); svi pristupi nad `y` iz obe niti dešavaju se unutar iste kritične sekcije (`CS_y`) pa analizator ne prijavljuje trku.

[3]Ako ista nit ponekad pristupa promenljivoj x pod ključem $K1$, a ponekad pod $K2$ (ili bez ključa), ne postoji garantovana, jedinstvena kritična sekcija koja obuhvata sve pristupe posmatrane niti toj promenljivoj. U trenutku kada je druga nit u sekciji $K1$, prva može biti u sekciji $K2$ pa je međusobno isključivanje prekinuto i trka je moguća.

Programi kod kojih je sinhronizacija odrađena na ovaj način su bezbedni i prolaze verifikaciju.

Primer 4. Ako bi implementacija igrice analizirane u primeru 2 bila napisana uz ovakvu sinhronizaciju, preciznije ukloliko bi niti zaključavale promenljivu koja čuva stanje zajedničke kase u trenutku kada igrač osvoji novčić, program bi bio ispravan.

Lažno upozorenje

Lažno upozorenje usled apstraktne interpretacije trke za resurisima prikazuje nedostatke ove metode kroz scenario u kom se dobija informacija o potencijalnoj trci za resursima iako se ona nikada neće desiti. Do ovoga dolazi jer ista apstrakcija može da odgovara različitim programima, odnosno i programu koji ima i programu koji nema trku za resursima.

Primer 5. Neka je dat program koji koristi dve niti od kojih jedna vrši pisanje a druga čitanje nad istom promenljivom, pri čemu ne koriste sinhronizaciju. Apstrakcija za takav program opisana je listingom 5.7. Za prikazanu apstrakciju analizator prijavljuje trku za resursima.

```
threads = {
    "T_writer": [("x", WRITE, None)],
    "T_reader": [("x", READ,  None)],
}
print(analyze_static(threads))  # Ocekivano: ['x']  #ima trke
```

Listing 5.7: Niti pišu i čitaju nad istom promenljivom

Navedeni primer je apstrakcija koja odgovara velikom broju programa. Za programe čije niti istovremeno pristupaju zajedničkom resursu ovakva apstrakcija detektuje nedostatak sinhronizacije. Jedan takav program opisan je primerom 6. Za programe koji koriste niti koje, iako ne koriste sinhronizaciju putem kriticne sekcije ipak ne mogu da pristupe istovremeno zajedničkom resursu, ova apstrakcija daje lažno upozorenje. Ovakvo ponašanje programa dato je u primeru 7.

Primer 6. Posmatrajmo sistem koji održava deljenu promenljivu `kursna_lista`, koja sadrži važeće kurseve preuzete od Narodne banke. Nit T_{writer} periodično preuzima novu kursnu listu sa servisa Narodne banke i upisuje je u promenljivu `kursna_lista`, bez postavljanja katanca prilikom upisa. Nit T_{reader} za svaki korisnički zahtev čita iz

`kursna_lista` odgovarajuće kurseve (npr. EUR/RSD) radi preračuna cena. Posledica izostanka sinhronizacije je čitanje zastarele vrednosti u koju T_{writer} upravo upisuje novu tabelu kurseva. U ovom slučaju analizator prijavljuje trku nad promenljivom `kursna_lista`, što je ispravno.

Primer 7. Posmatrajmo princip rada *Loader-Server* obrazca koji se često primenjuje u mikroservisima, veb servisima, demonskim procesima (eng. *deamon process*)[4] itd. Jedna nit opslužuje potrebe *Loader*-a (učitava, validira i objavljuje konfiguraciju) i kada ova nit završi sa radom šalje `READY` signal drugoj niti. Druga nit opslužuje korisnike i ima ulogu servera. *Server* ima dvostepenu inicijalizaciju:

- **Faza A (pre-konfiguracije, paralelna):** alokacija potrebnih resursa, pokretanje sistema koji su potrebni za praćenje rada servera, priprema keš objekata/struktura, učitavanje modula.
- **Faza B (posle `READY` signala):** vezivanje na port iz konfiguracije, učitavanje TLS (engl. *Transport Layer Security*) materijala (privatni ključevi, sertifikati i slično), podešavanje veličina (kao što je velična keša i broj niti (engl. *thread-pool*) koji će biti korišćen), uspostavljanje konekcija sa bazom podataka, registracija ruta i politika, i nakon svega toga otvaranje za saobraćaj.

Server može da obavi prvi deo inicijalizacije dok *Loader* čita/parsira konfiguraciju. Ovakvom paralelizacijom se postiže brzina, a čekanjem na učitavanje konfiguracije se postiže bezbednost.

Uloga *Loader*-a je da obezbedi jedinstveno, konzistentno početno stanje sistema; bez toga bi *Server* mogao da startuje sa delimičnom ili zastarelom konfiguracijom, što vodi nedeterminističkom ponašanju i bezbednosnim rizicima. Zato čekanje uspostavlja jasno pravilo: čitanje konfiguracije dolazi tek nakon uspešnog učitavanja konfiguracije.

U obe niti pristupa se promenljivoj koja čuva informacije o konfiguraciji. *Loader* nit piše u tu promenljivu a zatim je *Server* nit čita. Analizator vidi samo `WRITE`/`READ` iz različitih niti bez sinhronizacije korišćenjem kritične sekcije i zato prijavljuje trku iako je zapravo nema.

[4]Mehanizam pokretanja procesa u pozadini se obično koristi za pokretanje procesa koji rade jako dugo, bez ikakve interakcije sa korisnikom i nazivaju se demonski procesi.

Glava 6

Zaključak

U ovom radu predstavljena je apstraktna interpretacija kao efikasan metod za statičku analizu i verifikaciju softverskih sistema, sa posebnim naglaskom na razumevanje njenih osnovnih principa kroz elektronsku lekciju razvijenu u okruženju *Jupyter* sveske. Apstraktna interpretacija predstavlja teorijski okvir za formalno aproksimiranje ponašanja programa, pri čemu se umesto konkretnih vrednosti koristi pojednostavljena, apstraktna reprezentacija stanja izvršavanja. Na taj način omogućena je analiza svih mogućih izvršavanja programa bez potrebe za njegovim pokretanjem, što je čini pogodnom za rano otkrivanje grešaka. Na teorijskom nivou, obrađeni su ključni pojmovi kao što su rešetke, Galoaove veze, apstraktni domeni i tehnike za detekciju grešaka, uključujući prekoračenja opsega i trke za podacima. Implementacija primera u jeziku *Python* demonstrirala je kako se apstraktna interpretacija može koristiti za identifikaciju potencijalnih grešaka i povećanje pouzdanosti programa.

Elektronska lekcija je osmišljena tako da studentima omogući intuitivno razumevanje teorijskih koncepata statičke analize kroz interaktivne primere. Poseban fokus stavljen je na zadatke koji uključuju rad sa celobrojnim promenljivama i analizu konkurentnog ponašanja programa, čime se omogućava direktna primena teorije na konkretne problemske situacije. Prikazani primeri potvrđuju da se ovom tehnikom mogu pravovremeno identifikovati potencijalne opasnosti već u ranim fazama razvoja softvera.

Apstraktna interpretacija se pokazala kao moćan i fleksibilan alat, sposoban da analizira kako jednostavne izraze, tako i kompleksne programske strukture i obrasce ponašanja. Njena primenljivost u oblastima poput automobilske i avio-industrije, medicinskih uređaja i sigurnosno osetljivih sistema potvrđuje njen značaj u savre-

menom razvoju softvera. Daljim razvojem interaktivnih edukativnih materijala zasnovanih na formalnim metodama moguće je dodatno osnažiti njihovu upotrebu kako u nastavi, tako i u profesionalnoj praksi razvoja softverskih rešenja.

Literatura

[1] Tin Adlešić. *Prošireni modeli teorije skupova*. https://zir.nsk.hr/islandora/object/pmf%3A5097/datastream/PDF/view. Sveučilište u Zagrebu, Prirodoslovno–matematički fakultet, Matematički odsjek. 2018.

[2] Bruno Blanchet. *Introduction to Abstract Interpretation*. https://bblanche.gitlabpages.inria.fr/absint.pdf. Tutorial, Paris. 2002.

[3] Patrick Cousot. *A Tutorial on Abstract Interpretation*. https://homepage.cs.uiowa.edu/~tinelli/classes/seminar/Cousot--A%20Tutorial%20on%20AI.pdf. Online tutorial. 2005.

[4] Patrick Cousot. *Abstract Interpretation Webpage*. https://www.di.ens.fr/~cousot/AI/. Collection of materials on abstract interpretation. 2008.

[5] Patrick Cousot i Radhia Cousot. „Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". U: *Conference Record of the Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1977), str. 238–252.

[6] Milan Čugurović. *Apstraktna interpretacija – Uvod*. https://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/08_abstraktna_interpretacija/28_Milan_Cugurovic_ApstraktnaInterpretacijaUvod.pdf. Seminarski rad u okviru kursa Verifikacija softvera, Matematički fakultet, Univerzitet u Beogradu. 2018.

[7] Milena Vujošević Janičić. *Programske paradigme, Konkurentno programiranje - Predavanja*. https://www.programskijezici.matf.bg.ac.rs/dpj/2021/predavanja/konkurentno_programiranje.pdf. Univerzitet u Beogradu, Matematički fakultet, Katedra za računarstvo i informatiku. 2021.

[8] Milena Vujošević Janičić. *Verifikacija softvera - Predavanja.* https://www.verifikacijasoftvera.matf.bg.ac.rs/vs//verifikacija_softvera.pdf. Univerzitet u Beogradu, Matematički fakultet, Katedra za računarstvo i informatiku. 2025.

[9] Xavier Rival. „Abstract interpretation — semantics and applications to verification". U: (2016).

[10] Dimitrije Špadijer. *Apstraktna interpretacija – teorijske osnove apstraktne interpretacije. Primeri.* https://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/08_abstraktna_interpretacija/29_DimitrijeSpadijer_ApstraktnaInterpretacija.pdf. Seminarski rad u okviru kursa Verifikacija softvera, Matematički fakultet, Univerzitet u Beogradu. 2018.

[11] Alan Turing. „On Computable Numbers, with an Application to the Entscheidungsproblem". U: *Proceedings of the London Mathematical Society* 2.42 (1936), str. 230–265.

Biografija autora

Milica Kleut rođena u Čačku 18. novembra 1998. godine. Nakon završene Gimnazije u Čačku upisuje Matematički fakultet Univerziteta u Beogradu 2017. godine na smeru Matematika, modulu Računarstvo i informatika i isti završava 2022. godine. Nakon završenih osnovnih upisuje master studije na istom smeru. Tokom master studija zapošljava se u Rajfajzen banci (nem. *Raiffeisen bank*) kao inženjer podataka (eng. *data engineer*) u okviru odeljenja za skladištenje podataka (eng. *Data warehouse - DWH*). Tu svakodnevno koristi stečena znanja o bazama podataka, *SQL*-u i *PL/SQL*-u. Brzo postaje vešta sa novim alatima kao što je *ODI - Oracle data integrator*, *PowerBi*, *Toad*, *Cognos*. Pored formalnog obrazovanja, Milica voli da čita naučne radove i časopise i stiče znanja na internet kursevima iz raznih naučnih oblasti. Posebno interesantne grane su joj bezbednost i zaštita podataka, optimizacija, validacija i verifikacija softvera. U slobodno vreme se bavi amaterski slikarstvom, voli muziku, sport, putovanja i prirodu.