Formalna verifikacija softvera metodom provjere modela

Lešić, Klara

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Klara Lešić

FORMALNA VERIFIKACIJA SOFTVERA METODOM PROVJERE MODELA

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, studeni 2019.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U današnje vrijeme, verifikacija i validacija ključne su faze softverskog procesa[1]. S povećanjem ljudske ovisnosti o sve složenijim računalnim i softverskim sustavima, raste i potreba za tehnikama i alatima za procjenu funkcionalnih svojstava tih sustava. Jedna takva tehnika je *provjera modela* (eng. *model checking*).

Provjera modela jedna je od najraširenijih i najučinkovitijih metoda formalne verifikacije softvera. Kod provjere modela, softver koji želimo verificirati najprije prikazujemo odgovarajućim modelom, npr. konačnim automatom ili logičkim izrazom. Zatim provjeravamo da taj model ima neko poželjno svojstvo. Provjera se obično svodi na sistematsku provjeru svih mogućnosti, npr. prolazak svim putovima kroz konačni automat ili generiranje svih kombinacija vrijednosti varijabli u logičkom izrazu. Rezultat provjere je ili potvrda da svojstvo vrijedi ili protuprimjer koji pokazuje da svojstvo ne vrijedi.[2] Ovu provjeru provodi posebni softverski alat, tzv. *model checker*.

U ovom radu izložena je teorija o provjeri modela, a u praktičnom dijelu rada predstavljena je verifikacija provjerom modela na primjerima nekih poznatih matematičkih i računalnih problema, koristeći model checker NuSMV.

[1] Skup aktivnosti i pripadnih rezultata čiji je cilj razvoj softvera.

[2] Opis metode provjere modela preuzet je iz [5].

Poglavlje 1

Verifikacija softvera

Verifikacija softvera jedna je od osnovnih aktivnosti unutar softverskog procesa. Nju čini skup aktivnosti kojima se nastoji utvrditi ima li promatrani dizajn ili produkt određeno svojstvo. Svojstva koja se procjenjuju uglavnom proizlaze iz specifikacije sustava. Specifikacija propisuje što sustav treba raditi, a što ne, i time čini temelj za sve aktivnosti verifikacije. Sustav se smatra ispravnim ako on zadovoljava sva svojstva dobivena specifikacijom, a ukoliko postoji svojstvo iz specifikacije koje on ne ispunjava, sustav je potrebno modificirati. Dakle, ispravnost sustava uvijek se promatra relativno u donosu na specifikaciju i nije apsolutno svojstvo sustava.

Ovaj rad bavi se tehnikom verifikacije zvanom provjera modela, koja počiva na formalnoj specifikaciji sustava. Prije predstavljanja ove tehnike, recimo prvo nešto o formalnim metodama verifikacije u idućem odjeljku.

1.1 Formalna verifikacija

Prilikom razvoja softvera, često je potrebno više vremena i pažnje posvetiti verifikaciji nego samoj konstrukciji. Traže se tehnike koje će pojednostaviti i olakšati verifikaciju, istodobno povećavajući širinu područja koje će njome biti obuhvaćeno. Zbog svoje statičke prirode, formalne metode nude mogućnost da se verifikacija integrira već u ranoj fazi oblikovanja softvera. To rezultira ranijim otkrivanjem pogrešaka, čime se ukupni troškovi i trajanje verifikacije mogu znatno smanjiti.

Formalne metode možemo promatrati kao „primijenjenu matematiku za modeliranje i analizu softvera“. Njihova svrha je utvrditi ispravnost sustava oslanjajući se na matematičku strogost. Radi se o tehnici statičke verifikacije koja se svodi na matematičko dokazivanje da je program konzistentan sa svojom specifikacijom.

Dokaz se provodi tako da se polazni uvjeti iz specifikacije i same naredbe iz programa uzmu kao činjenice, pa se iz njih izvode konačni uvjeti koji po specifikaciji moraju biti zadovoljeni nakon izvršenja programa.[1] Ova vrsta verifikacije obično se primjenjuje samo na manje dijelove softvera, i to na one koji zahtijevaju izuzetno visoku razinu pouzdanosti i sigurnosti. U svrhu ubrzanja ove zahtjevne i dugotrajne metode, koriste se alati koji omogućuju automatsko provođenje dokaza.

Ideja formalne verifikacije pojavila se prije više od 70 godina, a intenzivnije je zaživjela tijekom 70-ih godina prošlog stoljeća, kad su se njome počela baviti velika imena računarstva kao što su Dijkstra, Hoare i Wirth. Iako u to vrijeme ova metoda nije zaživjela u praksi nego je korištena samo u manjim akademskim primjerima, u posljednja dva desetljeća istraživanja formalnih metoda dovela su do razvoja nekih vrlo obećavajućih tehnika verifikacije koje olakšavaju ranu detekciju pogrešaka. Te tehnike popraćene su moćnim softverskim alatima za automatizaciju raznih koraka verifikacije.

Smatra se da je formalna verifikacija tehnika koja zbog svog strogog nedvosmislenog matematičkog pristupa vodi do pouzdanijeg i sigurnijeg softvera. Ipak, to nije nužno tako. Naime, postoji mogućnost da specifikacija sustava ne odgovara u potpunosti stvarnim zahtjevima korisnika. Osim toga, sam dokaz korektnosti sustava može sadržavati pogreške budući da je dug i složen. Još jedna potencijalna manjkavost tog pristupa je mogućnost da će korisnik koristiti softver na način drugačiji od onoga koji je uzet kao pretpostavka dokaza.

Verifikacija temeljena na modelima[2] počiva na modelima koji opisuju moguće ponašanje sustava na matematički precizan i nedvosmislen način. Praksa je pokazala da, prije bilo kojeg oblika verifikacije, već ispravno modeliranje sustava često dovodi do otkrivanja nepotpunosti, nejasnoća i nedosljednosti u specifikaciji sustava. Takvi problemi obično bivaju otkriveni tek u puno kasnijoj fazi razvoja softvera. Modelima sustava pridružuju se algoritmi koji sistematski istražuju sva stanja modela sustava. Time se dobiva temelj za cijeli spektar različitih tehnika verifikacije – od iscrpnog istraživanja (provjera modela), do eksperimenata s manjim skupom scenarija u modelu (simulacija), ili testiranja u stvarnosti.

1.2 Provjera modela

Najistaknutija tehnika formalne verifikacije je *provjera modela*. To je automatizirana tehnika koja, za dani model s konačnim brojem stanja i za dano formalno svojstvo, sustavno

[1] Preuzeto iz [5]

[2] eng. *model-based verification*

provjerava zadovoljava li taj model (u određenom stanju) to svojstvo. Da bi se ta provjera mogla provesti algoritamski, potrebno je i softver koji želimo verificirati i traženo svojstvo formulirati nekim preciznim matematičkim jezikom, tj. prikazati u obliku nekog apstraktnog matematičkog modela. Najčešće je to konačni automat ili logički izraz.

Ovu provjeru provodi softverski alat koji se zove *model checker*. On sistematski ispituje sva moguća stanja sustava, primjerice, prolaskom po svim putovima konačnog automata ili generiranjem svih kombinacija vrijednosti varijabli logičkog izraza. Na ovaj način može se provjeriti zadovoljava li zaista dani model sustava određeno svojstvo. Ako je naišao na stanje sustava koje narušava traženo svojstvo, model checker generira protuprimjer koji pokazuje da svojstvo ne vrijedi. Taj protuprimjer opisuje put izvršavanja od početnog stanja sustava do stanja koje narušava to svojstvo. Inače, model checker daje potvrdu da svojstvo vrijedi.

Neke od prednosti metode provjere modela:

- ima širok raspon primjena, primjerice, može se primjenjivati za verifikaciju ugrađenih sustava, softvera i hardverskih sklopova;
- pruža mogućnost *parcijalne* verifikacije, tj. svojstva je moguće ispitivati zasebno, što omogućuje fokusiranje na neka temeljna svojstva, te stoga nije potrebna potpuna specifikacija;
- u slučaju da je traženo svojstvo narušeno, pruža dodatne informacije koje pomažu pri debuggiranju;
- ne zahtijeva veliku interakciju korisnika;
- lako se integrira u postojeći razvojni ciklus.

Mane metode provjere modela:

- ispituje ispravnost *modela sustava*, a ne stvarnog sustava – svaki dobiveni rezultat je točan onoliko koliko je točan model sustava;
- često ne može izraziti cjelokupne zahtjeve na sustav, nego provjerava samo neke pojedinačne zahtjeve;
- za sustave s beskonačnim brojem stanja, provjera modela općenito nije efektivno izračunljiva;
- zahtijeva visoku razinu stručnosti;
- pri njenoj primjeni često se javlja tzv. *state-space explosion* problem – broj stanja potrebnih za precizno modeliranje sustava lako može prekoračiti količinu dostupne računalne memorije.

Poglavlje 2

Modeliranje sustava

Preduvjet za provođenje provjere modela je stvaranje modela koji odgovara programu kojeg treba verificirati. Model opisuje ponašanje sustava na precizan i jednoznačan način. Priprema ispravnog modela od presudne je važnosti - svaka verifikacija koja koristi tehnike temeljene na modelu je onoliko dobra koliko je dobar model sustava. Ručno modeliranje sustava može zahtijevati previše vremena i može rezultirati modelom koji ne odgovara u potpunosti programu. Zbog toga je bolje ako se model oblikuje automatski iz izvornog teksta programa. Ovu mogućnost pružaju neki alati poput Java Pathfinder[1].

U ovom poglavlju predstavljena je standardna klasa modela za reprezentaciju softverskih sustava - *tranzicijski sustavi*[2].

2.1 Tranzicijski sustavi

Tranzicijski sustavi često su korišteni modeli u računarstvu za opisivanje ponašanja sustava. To su usmjereni grafovi u kojima vrhovi predstavljaju *stanja*, a lukovi *prijelaze* (*tranzicije*) sustava. Stanje opisuje neke informacije o ponašanju sustava u određenom trenutku, dok prijelazi propisuju kako sustav može napredovati od jednog stanja do drugog.

Slijedi formalna definicija tranzicijskog sustava.

[1] alat za verifikaciju Java programa, razvijen od strane NASA-a. Više o Java Pathfinder-u može se pronaći na `https://ti.arc.nasa.gov/tech/rse/vandv/jpf/`

[2] eng. *transition systems*

Definicija 2.1.1. *Tranzicijski sustav TS je šestorka* $(S, Act, \rightarrow, I, AP, L)$*, gdje je*

- S *skup čije elemente zovemo stanja,*
- Act *skup čije elemente zovemo radnje,*
- $\rightarrow \subseteq S \times Act \times S$ *relacija prijelaza,*
- $I \subseteq S$ *skup čije elemente zovemo početna stanja,*
- AP *skup atomarnih propozicija*[3]*,*
- $L\colon S \rightarrow 2^{AP}$ *funkcija označavanja*[4]*.*

Kažemo da je TS konačan ako su skupovi S*,* Act *i* AP *konačni.*

U literaturi se javljaju razni tipovi tranzicijskih sustava. Ovdje koristimo tranzicijske sustave kod kojih su prijelazima pridružene *radnje*, a stanjima *atomarne propozicije*. Atomarne propozicije intuitivno izražavaju neke jednostavne činjenice o stanjima sustava i koriste se za formaliziranje temporalnih svojstava.

Ponašanje tranzicijskog sustava (u daljnjem tekstu koristit ćemo pokratu TS) možemo opisati na sljedeći način: inicijalno se TS nalazi u nekom početnom stanju $s_0 \in I$ i dalje prelazi iz stanja u stanje na način zadan relacijom prijelaza $\rightarrow$. Preciznije, ako je s trenutno stanje, onda se na nedeterministički način bira prijelaz $s \xrightarrow{\alpha} s'$ [5] koji će idući biti napravljen. Zatim se taj izbor idućeg prijelaza na isti način nastavlja u stanju s', itd. Ova procedura prestaje kad se TS nađe u stanju iz kojeg nema daljnjih prijelaza. Slično kao prijelazi, i početno stanje se bira nedeterministički u slučaju da skup I sadrži više od jednog stanja.

Funkcija označavanja L svakom stanju $s \in S$ pridružuje skup $L(s) \in 2^{AP}$. Intuitivno, elementi skupa $L(s)$ su upravo one atomarne propozicije $a \in AP$ koje su istinite u stanju s.

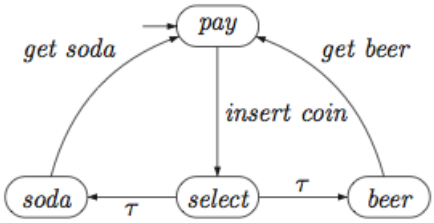
Slika 2.1 prikazuje jedan primjer TS-a, TS za pojednostavljeni automat za piće koji može izbaciti sok ili pivo. Stanja su prikazana ovalima unutar kojih su upisana imena stanja, a prijelazi lukovima označenim imenima radnji. Početno stanje označeno je "ulaznom" strelicom koja nema svog izvora.[6]

[3] eng. *atomic propositions*, tvrdnja koja mora biti ili istinita ili lažna. Neki primjeri atomarnih propozicija: "x je veći od 0", "x je jednak 100", za dani cijeli broj x, "5 je prost broj", itd.

[4] 2^{AP} je oznaka za partitivni skup skupa AP.

[5] pokrata za $(s, \alpha, s') \in \rightarrow$.

[6] Slike 2.1 i 2.2 preuzete su iz [1].



Slika 2.1: Tranzicijski sustav za jednostavni automat za piće.

- $S = \{pay, select, soda, beer\}$;
- $I = \{pay\}$;
- $Act = \{insert\_coin, get\_soda, get\_beer, \tau\}$;
- Neki primjeri prijelaza: $pay \xrightarrow{insert\_coin} select$, $beer \xrightarrow{get\_beer} pay$:
- Atomarne propozicije ovise o svojstvima koja se procjenjuju. Najjednostavniji primjer pridruživanja atomarnih propozicija stanjima je taj da svakom stanju pridružimo njegovo vlasitito ime kao atomarnu propoziciju:

  $$L(s) = \{s\}, \quad \text{za svako stanje } s.$$

  Drugi primjer definiranja skupa AP i funkcije označavanja je:

  $$AP = \{paid, drink\};$$
  $$L(pay) = \emptyset, \quad L(soda) = L(beer) = \{paid, drink\}, \quad L(select) = \{paid\}.$$

Definicija 2.1.2. *Neka je $TS = (S, Act, \rightarrow, I, AP, L)$ tranzicijski sustav. Za $s \in S$ i $\alpha \in Act$ skup neposrednih α-sljedbenika od s definiramo sa:*

$$Post(s, \alpha) = \left\{ s' \in S \;\middle|\; s \xrightarrow{\alpha} s' \right\}.$$

Skup svih sljedbenika od s označavamo s $Post(s)$, tj.

$$Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha).$$

Skup neposrednih α-prethodnika od s definiramo sa:

$$Pre(s, \alpha) = \left\{ s' \in S \;\middle|\; s' \xrightarrow{\alpha} s \right\}.$$

Skup svih prethodnika od s označavamo s $Pre(s)$, tj.

$$Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha).$$

Definicija 2.1.3. *Za stanje s tranzicijskog sustava kažemo da je završno ako je $Post(s) = \emptyset$.*

2.1.1 Izvršenja tranzicijskog sustava

Ponašanje TS-a, koje je dosad bilo opisano na intuitivnoj razini, sada ćemo formalizirati koristeći pojam *izvršenja*[7]. Različita izvršenja TS-a rezultat su nedeterminizma. Stoga pojam izvršenja koristimo za opisivanje mogućeg ponašanja TS-a.

Definicija 2.1.4. *Neka je $TS = (S, Act, \rightarrow, I, AP, L)$ tranzicijski sustav.*
Konačan fragment izvršenja[8] *ϱ TS-a je alternirajući niz stanja i radnji koji počinje i završava stanjem:*

$$\varrho = s_0 \alpha_1 s_1 \alpha_2 ... s_{n-1} \alpha_n s_n, \quad \text{takav da } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ za svaki } 0 \leq i < n,$$

gdje je $n \geq 0$ duljina fragmenta izvršenja ϱ.
Besonačan fragment izvršenja ρ TS-a je beskonačni alternirajući niz stanja i radnji:

$$\rho = s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 ..., \quad \text{takav da } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ za svaki } 0 \leq i.$$

Definicija 2.1.5. *Maksimalni fragment izvršenja je ili konačni fragment izvršenja koji završava u završnom stanju, ili beskonačni fragment izvršenja.*
Početni fragment izvršenja je fragment izvršenja koji započinje u početnom stanju.

Definicija 2.1.6. *Izvršenje tranzicijskog sustava TS je početni maksimalni fragment izvršenja.*

Definicija 2.1.7. *Neka je $TS = (S, Act, \rightarrow, I, AP, L)$ tranzicijski sustav.*
Za $s \in S$ kažemo da je dostupno stanje TS-a ako postoji konačni početni fragment izvršenja

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_n} s_n = s.$$

Skup svih dostupnih stanja TS-a označavamo s Reach(TS).

[7] eng. *execution*
[8] eng. *execution fragment*

$$\rho_1 = pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} \dots$$

$$\rho_2 = select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} beer \xrightarrow{bget} \dots$$

$$\varrho = pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda \ .$$

Slika 2.2: Neki primjeri fragmenata izvršenja TS-a sa slike 2.1

2.2 Data-dependent sustavi

Ovaj odjeljak govori o vrsti jednostavnih sekvencijalnih računalnih programa, *data-dependent sustavima*[9]. U takvim programima stanja na neki način služe za "pohranu varijabli" (primjerice, nekih kontrolnih varijabli, poput programskog brojača), a promjene stanja predstavljaju "promjene varijabli".

U slučaju data-dependent sustava, radnje koje se izvršavaju uglavnom su rezultat uvjetnog grananja. Npr.

$$\textbf{if } x\%2 = 1 \quad \textbf{then } x := x + 1 \quad \textbf{else } x := 2x \textbf{ fi}.$$

Za reprezentaciju kontrole toka programa koristimo tzv. *programski graf*[10]. Programski graf nad skupom varijabli je graf čiji su lukovi, osim radnjama, označeni uvjetima na varijable. Slijedi formalna definicija.

Definicija 2.2.1. *Programski graf PG nad skupom varijabli Var*[11] *je šestorka (Loc, Act, Effect,* $\hookrightarrow$*,* Loc_0*,* g_0*), gdje je*

- *Loc skup čije elemente zovemo lokacije,*
- *Act skup čije elemente zovemo radnje,*
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ *posljedična funkcija,*
- $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ *uvjetna relacija prijelaza*[12]*,*
- $Loc_0 \subseteq$ *Loc skup čije elemente zovemo početne lokacije,*
- $g_0 \in Cond(Var)$ *početni uvjet.*

[9] eng. *data-dependent systems*

[10] eng. *program graph*

[11] Za skup varijabli *Var* s $Eval(Var)$ označavamo skup svih evaluacija varijabli iz *Var*, a s $Cond(Var)$ skup svih uvjeta na varijable iz *Var*.

[12] Kao pokratu za $(l, g, \alpha, l') \in \hookrightarrow$ koristit ćemo $l \overset{g:\alpha}{\hookrightarrow} l'$.

Svaki programski graf (u daljnjem tekstu kraće pišemo PG) može se interpretirati kao TS.

Definicija 2.2.2. *Tranzicijski sustav TS(PG) programskog grafa PG = (Loc, Act, Effect,* $\hookrightarrow$*,* Loc_0*,* g_0*) nad skupom varijabli Var je šestorka (S, Act,* $\rightarrow$*, I, AP, L), gdje je*

- $S = Loc \times Eval(Var)$,
- $\rightarrow \subseteq S \times Act \times S$ *definirana na sljedeći način:*

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \ \wedge \ \eta \models g}{\langle l, \eta \rangle \overset{\alpha}{\rightarrow} \langle l', Effect(\alpha, \eta) \rangle},$$

- $I = \{\langle l, \eta \rangle \mid l \in Loc_0, \ \eta \models g_0\}$,
- $AP = Loc \cup Cond(Var)$,
- $L(\langle l, \eta \rangle) = \{l\} \cup \{g \in Cond(Var) \mid \eta \models g\}$.

Notacija $\frac{premisa}{konkluzija}$ može se čitati ovako: "ako vrijedi premisa, onda vrijedi konkluzija". Takve "ako ..., onda ..." tvrdnje zovu se *pravila izvoda*[13]. Ako je premisa tautologija, može biti izostavljena (kao i horizontalna linija).

2.3 Modeliranje paralelnih programa

Dosad smo vidjeli kako pomoću tranzicijskih sustava možemo modelirati jednostavnije sekvencijalne računalne sustave. No u stvarnosti, sustavi su uglavnom paralelni, a ne sekvencijalni.

U ovom odjeljku predstavit ćemo tri mehanizma za modeliranje paralelnih sustava: korištenje *operatora preplitanja*[14], komunikacija programa preko *zajedničkih varijabli*[15], te korištenje *operatora rukovanja*[16].

U prvom modelu, sustav je zapravo sastavljen od (djelomično) neovisnih komponenti. Radnje međusobno neovisnih komponenti su "prepletene" i spojene u jedan "globalni" sustav. Dakle, u ovom modelu paralelnost je predstavljena preplitanjem, tj. nedeterminističkim izborom među aktivnostima nekoliko procesa koji rade istovremeno.

[13] eng. *inference rules*
[14] eng. *interleaving operator*
[15] eng. *shared variables*
[16] eng. *handshaking*

Definicija 2.3.1. *Neka su* $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$*, i=1,2, tranzicijski sustavi. Tada definiramo tranzicijski sustav* $TS_1 ||| TS_2$ *s*

$$TS_1 ||| TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L),$$

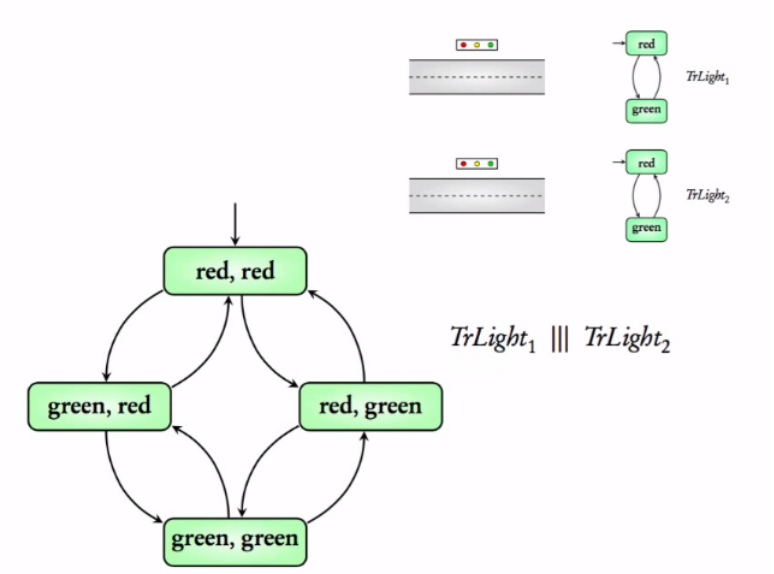
gdje je relacija prijelaza $\rightarrow$ *definirana sljedećim pravilima:*

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2\rangle} \quad i \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1, s_2'\rangle},$$

a funkcija označavanja s:

$$L(\langle s_1, s_2\rangle) = L(s_1) \cup L(s_2).$$

Slika 2.3[17] pokazuje primjer djelovanja operatora preplitanja. Dva međusobno neovisna semafora predstavljena su tranzicijskim sustavima $TrLight_1$ i $TrLight_2$. Tranzicijski sustav za paralelnu kompoziciju oba semafora dobivena je preplitanjem $TrLight_1$ i $TrLight_2$.



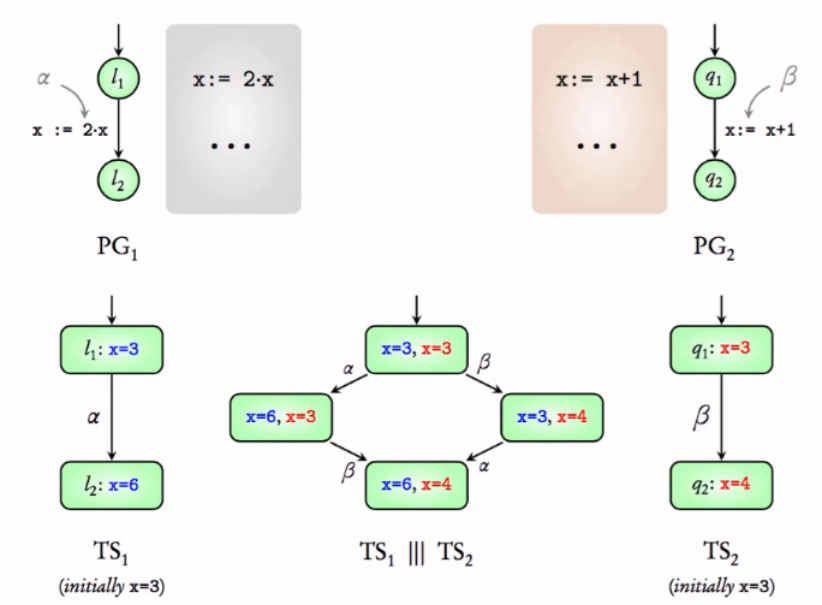
Slika 2.3: Operator preplitanja na primjeru sustava dva neovisna semafora.

Razmotrimo sada sustav sastavljen programa koji nisu neovisni, nego imaju nešto zajedničko - *dijeljene varijable*[18].

[17] preuzeta s kanala `https://www.youtube.com/channel/UCUXDMaaobCO1He1HBiFZnPQ/featured`

[18] eng. *shared variables*

U slučaju programa s dijeljenim varijablama, operator preplitanja koji smo definirali za nezavisne programe neće davati dobre rezultate. Primjer koji to pokazuje predstavljen je idućom slikom.[19] Tranzicijski sustav koji dobivamo preplitanjem TS_1 i TS_2 ima završno stanje gdje je $x = 6$ i $x = 4$, što je očito nemoguće.



Za modeliranje paralelnih programa sa zajedničkim varijablama moramo definirati operator preplitanja na razini programskih grafova.

Definicija 2.3.2. *Neka su* $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$, *i=1,2, programski grafovi nad skupovima varijabli* Var_i. *Tada definiramo programski graf* $PG_1|||PG_2$ *nad skupom* $Var_1 \cup Var_2$ *s*

$$PG_1|||PG_2 = (Loc_1 \times Loc_2, Act_1 \cup Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2}),$$

gdje je relacija $\hookrightarrow$ *definirana sljedećim pravilima:*

$$\frac{l_1 \overset{g:\alpha}{\hookrightarrow}_1 l_1'}{\langle l_1, l_2\rangle \overset{g:\langle\alpha,1\rangle}{\longrightarrow} \langle l_1', l_2\rangle} \quad i \quad \frac{l_2 \overset{g:\alpha}{\hookrightarrow}_2 l_2'}{\langle l_1, l_2\rangle \overset{g:\langle\alpha,2\rangle}{\longrightarrow} \langle l_1, l_2'\rangle},$$

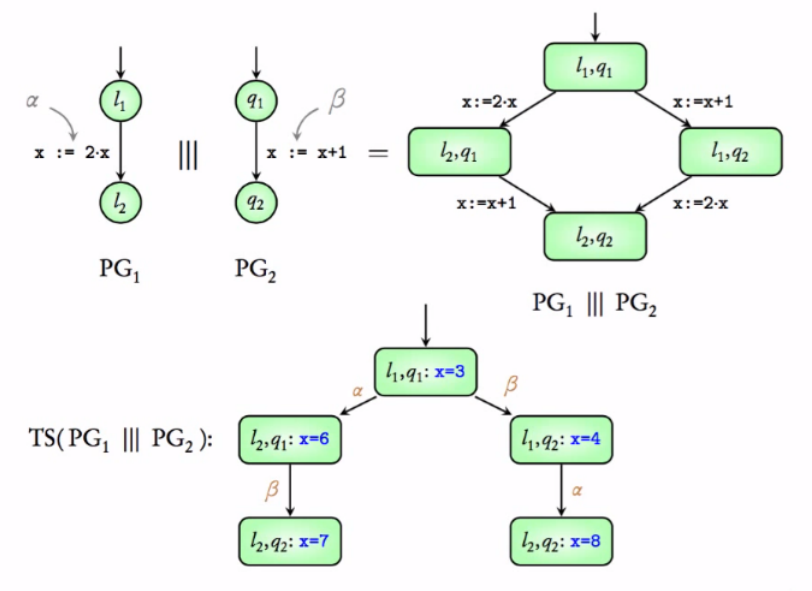
a posljedična funkcija s:

$$Effect(\alpha, \eta)(v) = \begin{cases} Effect_i(\langle\alpha, i\rangle, \eta|_{Var_i})(v), & ako\ v \in Var_i \\ \eta(v), & inače \end{cases},$$

pri čemu $\eta|_{Var_i}$ *označava restrikciju funkcije* η *na* Var_i.

[19] preuzeta s kanala `https://www.youtube.com/channel/UCUXDMaaobCO1He1HBiFZnPQ/featured`

Primjer preplitanja programskih grafova prikazan je slikom 2.4.[20]



Slika 2.4: Operator prepletanja na dva programska grafa.

Treći mehanizam za modeliranje paralelnih sustava, i to onih koji dijele neke zajedničke radnje, je tzv. *rukovanje*. U ovom slučaju, paralelni procesi rade sinkrono. Drugim riječima, procesi djeluju interaktivno samo ukoliko svaki od njih sudjeluje u toj interakciji u isto vrijeme (odatle dolazi naziv "rukovanje"). Sve radnje koje nisu dijeljene među procesima su neovisne i stoga se mogu izvršavati samostalno na način kako je opisano kod operatora prepletanja.

Radi opsega ovog rada, operator rukovanja nećemo detaljnije obrađivati. Više o rukovanju može se pronaći u [1].

[20] preuzeta s kanala `https://www.youtube.com/channel/UCUXDMaaobCO1He1HBiFZnPQ/featured`

Poglavlje 3

Svojstva

Kako bismo mogli provesti verifikaciju, modelu sustava kojeg ispitujemo potrebno je pridružiti svojstva koja on mora zadovoljavati. Ta svojstva treba formalizirati i predstaviti ih matematičkim jezikom, slično kao što smo modelu pridijelili matematičko značenje u obliku stanja i prijelaza tranzicijskog sustava.

Ovo poglavlje prvo daje formalnu definiciju svojstva i opisuje *linearno-vremensko ponašanje*[1], a zatim predstavlja neke važne klase svojstava, te algoritme koji se koriste za automatsku provjeru tih svojstava.

Za analizu svojstava, uglavnom ćemo koristiti pristup koji se temelji na stanjima sustava. Taj pristup isključuje radnje i uzima u obzir samo oznake nizova stanja. Oznake prijelaza sustava potrebne su samo za modeliranje komunikacije među procesima, pa nam stoga nisu potrebne u ovom poglavlju.

3.1 Formalizacija pojma svojstva

U definicijama koje slijede, neka je $TS = (S, Act, \rightarrow, I, AP, L)$ tranzicijski sustav.

Definicija 3.1.1. *Graf stanja TS-a, u oznaci* $G(TS)$*, je graf* (V, E) *s vrhovima* $V = S$ *i bridovima* $E = \{(s, s') \in S \times S \,|\, s' \in Post(s)\}$.

[1] eng. *linear-time behaviour*

Definicija 3.1.2. *Konačan fragment puta $\hat{\pi}$ TS-a je konačan niz stanja $s_0 s_1 ... s_n$ takav da*

$$s_i \in Post(s_{i-1}), \quad za\ sve\ 0 < i \leq n,$$

gdje je $n \geq 0$.
Beskonačan fragment puta π je beskonačan niz stanja $s_0 s_1 s_2 ...$ takav da

$$s_i \in Post(s_{i-1}), \quad za\ sve\ i > 0.$$

Definicija 3.1.3. *Maksimalni fragment puta je ili konačan fragment puta koji završava u završnom stanju, ili beskonačni fragment puta.*
Za fragment puta $s_0 s_1 ...$ kažemo da je početni ako započinje u početnom stanju, tj. ako je $s_0 \in I$.

Označimo s $Paths(s)$ skup svih maksimalnih fragmenata puta π takvih da je $first(\pi) = s$ [2], a s $Paths_{fin}(s)$ skup svih konačnih fragmenata puta $\hat{\pi}$ takvih da je $first(\hat{\pi}) = s$.

Definicija 3.1.4. *Put tranzicijskog sustava TS-a je početni, maksimalni fragment puta.*

Skup svih puteva TS-a označimo s $Paths(TS)$, a skup svih konačnih početnih fragmenata puta s $Paths_{fin}(TS)$.

Izvršenja predstavljena u odjeljku 2.1.1 su alternirajući nizovi stanja i radnji. Ovdje se želimo fokusirati samo na stanja koja su posjećena tijekom izvšenja. Preciznije, želimo se fokusirati na atomarne propozicije tih stanja. Zato, umjesto da izvšenje opisujemo u obliku $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$, promatramo nizove oblika $L(s_0)L(s_1)L(s_2)...$ koji bilježe atomarne propozicije koje su istinite tokom izvršenja. Takve nizove zovemo *tragovima*[3].

Radi jednostavnosti, bez smanjenja općenitosti u nastavku pretpostavljamo da TS nema završnih stanja[4]. U tom slučaju, svi tragovi su beskonačne riječi.

---

[2] first(π) je oznaka za početno stanje puta π

[3] eng. *traces*

[4] Svaki TS TS_1 može se proširiti do ekvivalentnog TS-a TS_2 bez završnih stanja tako da se za svako završno stanje od TS_1 doda novo stanje s_{stop}, prijelaz $s \rightarrow s_{stop}$ i stanju s_{stop} se doda beskonačna petlja $s_{stop} \rightarrow s_{stop}$.

Definicija 3.1.5. *Neka je* $TS = (S, Act, \rightarrow, I, AP, L)$ *tranzicijski sustav bez završnih stanja. Trag beskonačnog fragmenta puta* $\pi = s_0 s_1 s_2 ...$ *definiramo kao*

$$trace(\pi) = L(s_0)L(s_1)L(s_2)...$$

Trag konačnog fragmenta puta $\hat{\pi} = s_0 s_1 ... s_n$ *je*

$$trace(\hat{\pi}) = L(s_0)L(s_1)...L(s_n).$$

Dakle, trag fragmenta puta je izvedena konačna ili beskonačna riječ nad alfabetom 2^{AP}. Preciznije, to je niz skupova atomarnih propozicija koje su istinite u stanjima koja čine fragment puta.

Uvodimo sljedeće oznake:

$$trace(\Pi) = \{\ trace(\pi) \mid \pi \in \Pi\ \}, \text{ za neki skup putova } \Pi;$$

$$Traces(s) = trace(Paths(s)) \quad and \quad Traces(TS) = \bigcup_{s \in I} Traces(s);$$

$$Traces_{fin}(s) = trace(Paths_{fin}(s)) \quad and \quad Traces_{fin}(TS) = \bigcup_{s \in I} Traces_{fin}(s).$$

Linearno-vremenska svojstva[5] određuju željeno ponašanje sustava koji verificiramo. Sada ćemo dati formalnu definiciju tih svojstava. Kasnije, u poglavlju 5, predstavit ćemo i logički formalizam za specificiranje linearno-vremenskih svojstava.

U nastavku pretpostavljamo da imamo fiksni skup atomarnih propozicija AP. Uvodimo oznaku $(2^{AP})^{\omega}$ za skup beskonačnih riječi nad skupom 2^{AP}. Osim operatora L^{ω} za beskonačnu konkatenaciju riječi iz L, gdje je L proizvoljan skup riječi, uvodimo i operatore L^* (tzv. *Kleenejeva zvijezda*) i L^+ (*Kleenejev plus*) koje ćemo također koristiti u nastavku rada. Oni se definiraju na sljedeći način:

$$L^* := \bigcup_{i \in \mathbb{N}_0} L^i,$$

$$L^+ := \bigcup_{i \in \mathbb{N}_+} L^i, \quad \text{tj.} \quad L^+ := L^* \setminus \{\varepsilon\}.^6$$

Pritom je L^i oznaka za konkatenaciju skupa L sa samim sobom i puta. Preciznije, L^i se definira rekurzivno na sljedeći način:

$$L^0 := \{\varepsilon\},$$

$$L^1 := L,$$

$$L^{i+1} := \{\ wv \mid w \in L^i \text{ i } v \in L\ \}, \text{ za svaki } i \geq 1.$$

[5] eng. *linear-time properties*, koristit ćemo pokratu LT svojstvo

[6] S ε označavamo praznu riječ.

Definicija 3.1.6. *Linearno-vremensko svojstvo (LT svojstvo) nad skupom atomarnih propozicija AP je podskup skupa* $(2^{AP})^{\omega}$.

Sada se postavlja pitanje "kada kažemo da tranzicijski sustav zadovoljava svojstvo"? Odgovor je dan idućom definicijom.

Definicija 3.1.7. *Neka je P neko LT svojstvo nad skupom AP, te neka je* $TS = (S, Act, \rightarrow, AP, L)$ *tranzicijski sustav bez završnih stanja. Tada kažemo da TS zadovoljava P, u oznaci* $TS \models P$, *ako vrijedi*

$$Traces(TS) \subseteq P.$$

Kažemo da stanje $s \in S$ *zadovoljava P, u oznaci* $s \models P$, *ako je* $Traces(s) \subseteq P$.

Dakle, TS zadovoljava LT svojstvo P ako su svi njegovi tragovi sadržani u P, a stanje zadovoljava P ako se svi tragovi koji počinju u tom stanju nalaze u P.

U nastavku ovog poglavlja govorimo o nekim često korištenim klasama svojstava.

3.2 Invarijante

Invarijanta[7] je LT svojstvo koje za dani uvjet Φ zahtijeva da je Φ ispunjen u svim dostupnim stanjima.

Definicija 3.2.1. *LT svojstvo* P_{inv} *nad skupom AP zovemo invarijanta ako postoji formula propozicionalne logike*[8] Φ *nad AP takva da*

$$P_{inv} = \{A_0A_1A_2... \in (2^{AP})^{\omega} \mid \forall j \geq 0 : A_j \models \Phi\}.$$

Formulu Φ *zovemo uvjet invarijante svojstva* P_{inv}.

[7] eng. *invariant*

[8] *Formula propozicionalne logike* nad *AP* definira se induktivno:

1. *true* je formula.
2. Svaka atomarna propozicija $a \in AP$ je formula.
3. Ako su Φ_1, Φ_2 i Φ formule, onda su $(\neg\Phi)$ i $(\Phi_1 \wedge \Phi_2)$ također formule.
4. Ništa osim gore navedenog nije formula.

Uočimo da je

$$\begin{aligned} TS \models P_{inv} \quad & akko \quad trace(\pi) \in P_{inv}, \text{za sve puteve } \pi \text{ u } TS \\ & akko \quad L(s) \models \Phi, \text{za svako stanje } s \text{ koje pripada nekom putu od } TS \\ & akko \quad L(s) \models \Phi, \text{za svako stanje } s \in Reach(TS). \end{aligned}$$

Dakle, pojam invarijante možemo opisati ovako: uvjet Φ mora biti ispunjen u svim početnim stanjima i zadovoljenost uvjeta Φ je konstantna na svim dostupnim fragmentima zadanog TS-a.

Sada ćemo dati algoritam koji odgovara na pitanje "zadovoljava li zadani TS invarijantu?". Budući da je provjera invarijante za propozicionalnu formulu Φ zapravo provjera istinitosti formule Φ u svakom dostupnom stanju, malom izmjenom standardnih algoritama za prolazak grafom poput *DFS-a (pretraživanja u dubinu)* ili *BFS-a (pretraživanja u širinu)* dobit ćemo traženi algoritam, pod uvjetom da je zadani TS konačan.

Algoritam koji ćemo predstaviti vraća odgovor na pitanje zadovoljava li TS invarijantu. Uz to, daje nam i protuprimjer u slučaju da invarijanta nije zadovoljena. U trenutku kad algoritam naiđe na stanje s_n koje narušrava uvjet invarijante Φ, u stogu kojeg koristimo kao "spremnik" za stanja koja je još potrebno ispitati nalaze se stanja koja, čitajući od dna prema vrhu, čine traženi početni fragment puta - protuprimjer.

Algoritam 1 Provjera invarijante modificiranim DFS algoritmom

Ulaz: konačni tranzicijski sustav TS i propozicionalna formula Φ
Izlaz: "da" ako $TS \models$ "*uvijek* Φ", inače "ne" uz protuprimjer

---

set of states $R := \emptyset$; (* skup posjećenih dostupnih stanja *)
stack of states $U := \varepsilon$; (* prazni stog *)
bool $b := true$; (* flag - sva stanja u R zadovoljavaju Φ *)

while $(I \setminus R \neq \emptyset \wedge b)$ **do**
  let $s \in I \setminus R$; (* odaberi proizvoljno početno stanje koje nije u R *)
  visit(s); (* napravi DFS za svako neposjećeno početno stanje *)
end while
if b **then**
  return("da") (* $TS \models$ "*uvijek* Φ" *)
else
  return("ne", reverse(U)) (* protuprimjer se dobiva iz sadržaja stoga *)
end if

---

procedure visit (state s)
  $push(s, U)$; (* stavi s na vrh stoga *)
  $R := R \cup \{s\}$; (* označi s kao posjećeno (dostupno) *)
  repeat
    $s' := top(U)$;
    if $Post(s') \subseteq R$ **then**
      $pop(U)$;
      $b := b \wedge (s' \models \Phi)$; (* provjeri istinitost od Φ u s' *)
    else
      let $s'' \in Post(s') \setminus R$;
      $push(s'', U)$;
      $R := R \cup \{s''\}$; (* stanje s'' je novo posjećeno (dostupno) stanje *)
    end if
  until $((U = \varepsilon) \vee \neg b)$
endproc

3.3 Svojstva sigurnosti

Ovaj odjeljak bavi se *svojstvom sigurnosti*[9], koje je generalizacija invarijante. Svojstvo sigurnosti koristimo za provjeru da se "nešto loše neće nikad dogoditi".

U prethodnom odjeljku vidjeli smo da invarijante možemo promatrati kao svojstva stanja i provjeravamo ih uzimajući u obzir samo dostupna stanja. Međutim, za neka svojstva sigurnosti nije dovoljno provjeravati samo dostupna stanja. Primjerice, ako promatramo sustav bankomata, prirodan zahtjev je da novac može biti isplaćen tek nakon što je unesen ispravan PIN. Ovo svojstvo nije invarijanta, budući da se ne radi o svojstvu stanja. Ovdje je riječ o svojstvu sigurnosti - svako beskonačno izvršenje koje narušava zahtjev ima konačan *"loši prefiks"*; u slučaju baknomata, to bi npr. bio scenarij gdje se novac izdaje bez ispravnog unosa PIN-a. Dakle, svojstvo sigurnosti sadrži sve riječi koje ne počinju lošim prefiksom.

Definicija 3.3.1. *LT svojstvo P_{safe} nad skupom AP zovemo svojstvo sigurnosti ako za svaku riječ $\sigma \in (2^{AP})^\omega \setminus P_{safe}$ postoji konačan prefiks $\hat{\sigma}$ od σ takav da je*

$$P_{safe} \cap \left\{ \sigma' \in (2^{AP})^\omega \mid \hat{\sigma} \text{ je konačan prefiks od } \sigma' \right\} = \emptyset.$$

Svaku takvu konačnu riječ $\hat{\sigma}$ zovemo loš prefiks za P_{safe}.

Minimalni loš prefiks za P_{safe} je loš prefiks $\hat{\sigma}$ za P_{safe} takav da ne postoji prefiks od $\hat{\sigma}$ koji je također loš prefiks za P_{safe}. Drugim riječima, minimalni loši prefiksi su loši prefiksi minimalne duljine.

Skup svih loših prefiksa za P_{safe} označava se s $BadPref(P_{safe})$, a skup svih minimalnih loših prefiksa s $MinBadPref(P_{safe})$.

Uočimo da je svaka invarijanta svojstvo sigurnosti; za propozicionalnu formulu Φ nad skupom AP i njezinu invarijantu P_{inv}, sve konačne riječi oblika

$$A_0 A_1 ... A_n \in (2^{AP})^+,$$

gdje $A_0 \models \Phi, ..., A_{n-1} \models \Phi$ i $A_n \not\models \Phi$, čine minimalni loš prefiks za P_{inv}.

S druge strane, postoje svojstva sigurnosti koja nisu invarijante. To možemo vidjeti na primjeru pojednostavljenog automata za piće. Prirodan zahtjev na automat za piće je

"Broj ubacivanja novca u automat je uvijek barem jednak broju isporučenih napitaka."

[9] eng. *safety property*

Koristeći skup propozicija $pay, drink$ i odgovarajuću jednostavnu funkciju označavanja, ovo svojstvo može se formalizirati skupom beskonačnih riječi $A_0A_1A_2...$ takvih da za svaki $i \geq 0$ vrijedi

$$\left|\{\ 0 \leq j \leq i \mid pay \in A_j\ \}\right| \quad \geq \quad \left|\{\ 0 \leq j \leq i \mid drink \in A_j\ \}\right|.$$

Loši prefiksi za ovo svojstvo sigurnosti su npr.

$$\emptyset\ \{pay\}\ \{drink\}\ \{drink\} \quad \text{i}$$
$$\emptyset\ \{pay\}\ \{drink\}\ \emptyset\ \{pay\}\ \{drink\}\ \{drink\}.$$

3.4 Svojstva napretka

Neformalno govoreći, svojstva sigurnosti osiguravala su da se "nešto loše nikad neće dogoditi". Algoritam lako može ispuniti svojstvo sigurnosti tako da jednostavno ne poduzima nikakve korake i time nikad ni ne dolazi u situaciju da se nešto "loše" dogodi. Budući da uglavnom nije cilj da sustav samo miruje, svojstva sigurnosti dopunjuju se svojstvima koja zahtijevaju nekakav napredak. Takva svojstva zovu se *svojstva napretka*[10]. Intuitivno, ona govore da će se "nešto dobro kad-tad dogoditi".

Za razliku od svojstava sigurnosti koja bivaju narušena u konačnom vremenu, svojstva napretka narušavaju se u beskonačnom vremenu. Tipičan primjer ove klase svojstava je zahtjev da se određeni događaj javlja beskonačno mnogo puta.

Definicija 3.4.1. *LT svojstvo P_{live} nad skupom AP zovemo svojstvom napretka ako je*

$$pref(P_{live}) = (2^{AP})^*,$$

gdje je $pref(P_{live})$ skup svih konačnih prefiksa od P_{live}.

Dakle, svojstvo napretka je LT svojstvo P takvo da se svaka konačna riječ može proširiti do beskonačne riječi koja zadovoljava P. Drugim riječima, P je svojstvo napretka ako i samo ako za svaku konačnu riječ $w \in (2^{AP})^*$ postoji beskonačna riječ $\sigma \in (2^{AP})^\omega$ tako da je $w\sigma \in P$.

[10] eng. *liveness property* ili *progress property*

3.5 Pravednost

U ovom odjeljku govorimo o *pravednosti*[11] sustava. Pravednost isključuje beskonačno ponašanje sustava koje se smatra nerealnim i često je nužna za postizanje svojstva napretka.

Neka su $P_1, ..., P_N$ procesi, te neka je *Server* proces koji bi trebao pružati usluge tim procesima. Ponašanje sustava opisano je na sljedeći način. *Server* provjerava zahtjeve počevši od P_1, zatim P_2, i tako dalje, te poslužuje najprije onaj proces koji traži uslugu na kojeg je prvo naišao. Zatim, nakon što je poslužio taj proces, ponavlja istu proceduru odabira, ponovno krećući od P_1.

Pretpostavimo sada da P_1 stalno iznova traži uslugu. U tom slučaju ova strategija rezultirat će time da *Server* poslužuje samo P_1, a ostali procesi koji traže uslugu čekaju beskonačno dugo da bi im usluga bila pružena. Ovdje govorimo o *nepravednoj strategiji*. *Pravedna strategija* podrazumijeva da će *Server* kad-tad odgovoriti na zahtjev bilo kojeg procesa.

Specijalna vrsta pravednosti je *pravednost procesa*. Ona osigurava pravedno raspoređivanje izvršavanja procesa. Pravednost procesa bit će prikazana primjerom međusobnog isključivanja kojim ćemo se baviti u odjeljku 3.6.

Općenito govoreći, pravednost je potrebna za dokazivanje svojstva napretka ili drugih svojstava koja kažu da će se ”nešto dobro dogoditi u budućnosti”. Ovo je od presudne važnosti ako se promatrani sustav na neki način ponaša nedeterministički. Pravednost tada jamči da rezultat tog nedeterminizma neće biti konstantno ignoriranje neke mogućnosti.

Razlikujemo tri vrste pravednosti: bezuvjetnu, jaku i slabu pravednost.

Bezuvjetna pravednost znači da je aktivnost (npr. ulazak procesa u kritični odsječak ili posluživanje procesa koji je zatražio uslugu) omogućena beskonačno mnogo puta, uvijek, bez ograničenja nekim uvjetom pod kojim se to može dogoditi. Primjer takve pravednosti je ”Svaki proces dolazi na red beskonačno mnogo puta.”

Jaka pravednost znači da je aktivnost moguća, ali ne nužno uvijek. Npr. mogu postojati neki konačni periodi tokom kojih aktivnost neće biti moguća. Dakle, aktivnost ne mora biti uvijek omogućena, no mora biti omogućena beskonačno mnogo puta. Npr. ”Svaki proces koji je omogućen beskonačno mnogo puta dolazi na red beskonačno mnogo puta.”

Slaba pravednost znači da ako je aktivnost neprekidno omogućena (dakle, niti u jednom periodu ne smije biti onemogućena), onda ona mora biti izvršena beskonačno mnogo puta.

[11] eng. *fairness*

Primjer slabe pravednosti je "Svaki proces koji je neprekidno omogućen od nekog trenutka nadalje dolazi na red beskonačno mnogo puta."

Definicija 3.5.1. *Za tranzicijski sustav $TS = (S, Act, \rightarrow, AP, L)$ bez završnih stanja, skup $A \subseteq Act$ i beskonačan fragment izvršenja $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} ...$ TS-a kažemo:*

- *ρ je bezuvjetno A-pravedan ako vrijedi:*

$$\text{postoji beskonačno mnogo indeksa } j \text{ takvih da } \alpha_j \in A;$$

- *ρ je jako A-pravedan ako vrijedi:*

$$\text{ako postoji beskonačno mnogo indeksa } j \text{ takvih da } Act(s_j) \cap A \neq \emptyset,$$
$$\text{onda postoji beskonačno mnogo indeksa } j \text{ takvih da } \alpha_j \in A;$$

- *ρ je slabo A-pravedan ako vrijedi:*

$$\text{ako za gotovo sve}^{12} \text{ indekse } j \text{ vrijedi } Act(s_j) \cap A \neq \emptyset,$$
$$\text{onda postoji beskonačno mnogo indeksa } j \text{ takvih da } \alpha_j \in A.$$

Za stanje s, s $Act(s)$ označavamo skup svih radnji koje je moguće poduzeti iz stanja s, tj.

$$Act(s) = \{\alpha \in Act \mid \exists s' \in S : \ s \xrightarrow{\alpha} s'\}.$$

3.6 Međusobno isključivanje

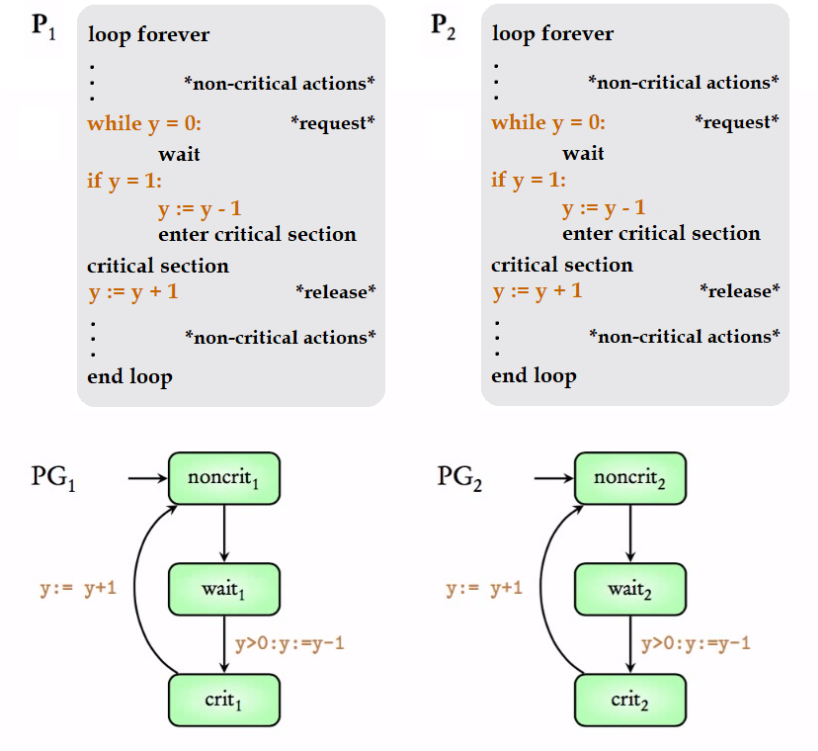
Sada ćemo prikazati dosad opisana svojstva na primjeru poznatog problema *međusobnog isključivanja*[13]. Međusobno isključivanje je mehanizam kontrole kod paralelnih programa s dijeljenim resursima. On osigurava da dva procesa ne obavljaju konfliktne radnje u isto vrijeme, tj. da samo jedan proces smije koristiti neki dijeljeni resurs u jednom trenutku. Rad procesa s resursom predstavlja njegovu takozvanu *kritičnu sekciju*. Dakle, dva procesa se ne smiju istovremeno nalaziti u svojim kritičnim sekcijama.

Promatramo dva procesa, P_1 i P_2, koji su reprezentirani programskim grafovima PG_1 i PG_2, respektivno. P_1 i P_2 dijele zajedničku kontrolnu bool varijablu y. $y = 0$ govori da

[12] za sve osim njih konačno mnogo

[13] eng. *mutual exclusion*

se neki od procesa trenutno nalazi u kritičnoj sekciji. Ako je kritična sekcija "slobodna", $y = 1$.



Slika 3.1: Međusobno isključivanje

Kad neki od procesa želi ući u kritičnu sekciju, on provjeri vrijednost varijable y. Ako je $y > 0$, kritična sekcija je slobodna i proces ulazi u nju, postavljajući pritom $y := y - 1$. Prilikom izlaska iz kritične sekcije, proces ponovno postavlja y na 1. Ako je u trenutku kad je proces zatražio ulazak u kritičnu sekciju vrijednost varijable y jednaka 0, on mora čekati da drugi proces izađe iz kritične sekcije i postavi $y = 1$. Ponašanje ovih procesa i pripadni programski grafovi prikazani su na slici 3.1, a TS koji reprezentira paralelni rad tih procesa na slici 3.2.[14] Napomenimo samo da na slici 3.2 n označava *noncrit*, w *wait*, a c *crit*.

Za specificiranje *svojstva međusobnog isključivanja*, da se u svakom trenutku najviše jedan proces nalazi u kritičnoj sekciji, dovoljno je promatrati samo atomarne propozicije $crit_1$ i $crit_2$. Svojstvo međusobnog isključivanja može se formalizirati LT svojstvom

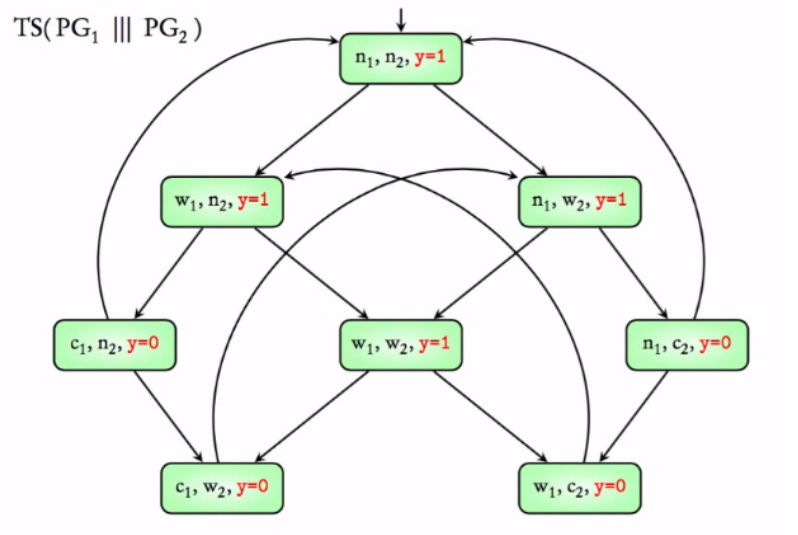
P_{mutex} = skup beskonačnih riječi $A_0A_1A_2...$ takvih da $\{crit_1, crit_2\} \not\subseteq A_i$, za svaki $i \geq 0$.

[14] preuzeto s `https://www.youtube.com/channel/UCUXDMaaobC01He1HBiFZnPQ/featured`

Primjerice, beskonačne riječi

$$\{crit_1\}\,\{crit_2\}\,\{crit_1\}\,\{crit_2\}\,\{crit_1\}\,\{crit_2\}\,..., \quad \text{i}$$
$$\{crit_1\}\,\{crit_1\}\,\{crit_1\}\,\{crit_1\}\,\{crit_1\}\,\{crit_1\}\,..., \quad \text{i}$$
$$\emptyset\,\emptyset\,\emptyset\,\emptyset\,\emptyset\,\emptyset\,...$$

su elementi skupa P_{mutex}. Svojstvo P_{mutex} je svojstvo sigurnosti. Štoviše, P_{mutex} je invarijanta.



Slika 3.2: Međusobno isključivanje - reprezentacija tranzicijskim sustavom

Svojstvo međusobnog isključivanja je temeljno pravilo međusobnog isključivanja, ali ne i jedino koje je važno. Algoritam koji nikad ne dopušta procesu da uđe u kritičnu sekciju zadovoljava ovo svojstvo, ali obično takvo ponašanje nije poželjno. Zato se uz problem međusobnog isključivanja veže i svojstvo koje zahtijeva da procesu koji želi ući u kritičnu sekciju to zaista prije ili kasnije bude i omogućeno. Ovo svojstvo sprječava scenarij gdje proces beskonačno dugo čeka, a formalno ga možemo izraziti LT svojstvom

$P_{finwait}$ = skup beskonačnih riječi $A_0A_1A_2...$ takvih da za svaki indeks j vrijedi:

$$wait_i \in A_j \;\Rightarrow\; \exists k > j :\; crit_i \in A_k \;\text{ za svaki } i \in \{1,2\}.$$

U ovom slučaju, za skup atomarnih propozicija uzeli smo $AP = \{wait_1, crit_1, wait_2, crit_2\}$. Dakle, svojstvo $P_{finwait}$ osigurava da će vrijeme čekanja pojedinog procesa na ulazak u kritičnu sekciju uvijek biti konačno.

Razmotrimo sada invarijantu

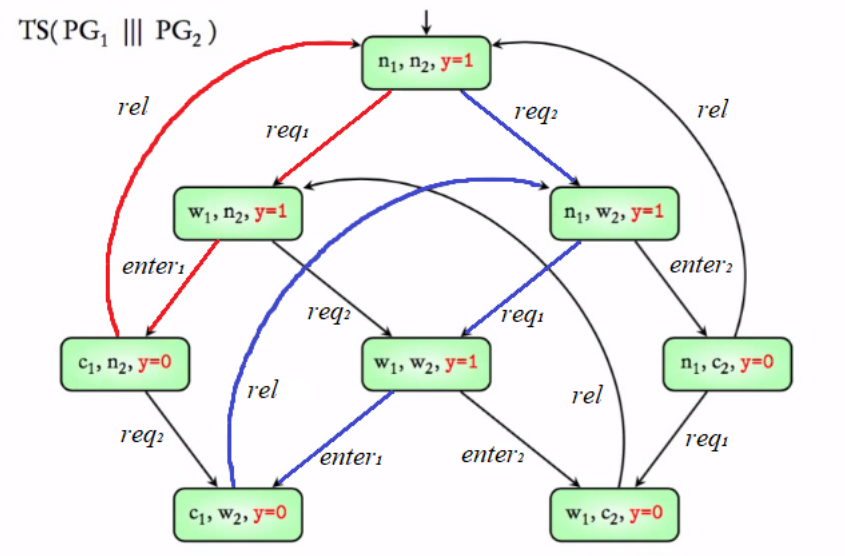
$$P_{nostarve} = \text{ skup beskonačnih riječi } A_0A_1A_2... \text{ takvih da vrijedi:}$$

$$(\forall k \geq 0 : \exists j \geq k : wait_i \in A_j) \Rightarrow (\forall k \geq 0 : \exists j \geq k : crit_i \in A_j) \text{ za svaki } i \in \{1, 2\}.$$

Svojstvo $P_{nostarve}$ izražava da ako je proces zatražio ulazak u kritičnu sekciju beskonačno mnogo puta, onda on i ulazi u kritičnu sekciju beskonačno mnogo puta. Algoritam međusobnog isključivanja kojeg smo mi u ovom odjeljku opisali ne zadovoljava ovo svojstvo. Naime, sa slike 3.2 vidimo da je

$$\emptyset\,(\{wait_2\}\,\{wait_1, wait_2\}\,\{crit_1, wait_2\})^{\omega}\ ^{15}$$

mogući trag tranzicijskog sustava, no ta riječ ne pripada svojstvu $P_{nostarve}$. Ovaj trag predstavlja izvršenje u kojem samo prvi proces ulazi u kritičnu sekciju beskonačno mnogo puta, dok drugi proces čeka beskonačno dugo na ulazak u kritičnu sekciju. Svojstvo $P_{nostarve}$ je svojstvo napretka i zovemo ga *odsutstvo izgladnjivanja*[16]. Postoje neki drugi algoritmi međusobnog isključivanja koji zadovoljavaju odsutstvo izgladnjivanja, npr. Pethersonov algoritam[17].



Slika 3.3: Međusobno isključivanje - reprezentacija tranzicijskim sustavom s označenim prijelazima

[15] ω označava da se niz iz zagrade ponavlja beskonačno mnogo puta

[16] eng. *starvation freedom*

[17] Više o Pethersonovom algoritmu može se pronaći u [1].

Promotrimo sada TS sa slike 3.3. Prijelaze smo označili radnjama $rel, req_i, enter_i$, za $i = 1, 2$. U fragmentu izvršenja

$$\langle n_1, n_2, y = 1\rangle \xrightarrow{req_1} \langle w_1, n_2, y = 1\rangle \xrightarrow{enter_1} \langle c_1, n_2, y = 0\rangle \xrightarrow{rel} \langle n_1, n_2, y = 1\rangle \xrightarrow{req_1} ...$$

jedino prvi proces dolazi na red za ulazak u kritičnu sekciju. Za skup radnji $A = \{enter_2\}$, ovaj fragment izvršenja nije bezuvjetno A-pravedan, no jest jako A-pravedan. Naime, nije posjećeno nijedno stanje u kojem je moguće izvršiti radnju $enter_2$, pa je stoga uvjet jake pravednosti trivijalno zadovoljen. Na slici je ovaj fragment izvršenja istaknut crvenim strelicama. Pogledajmo sada fragment izvršenja označen plavom bojom:

$$\langle n_1, n_2, y = 1\rangle \xrightarrow{req_2} \langle n_1, w_2, y = 1\rangle \xrightarrow{req_1} \langle w_1, w_2, y = 1\rangle \xrightarrow{enter_1}$$
$$\langle c_1, w_2, y = 0\rangle \xrightarrow{rel} \langle n_1, w_2, y = 1\rangle \xrightarrow{req_1} ...$$

U ovom slučaju proces P_2 traži ulazak u kritičnu sekciju, ali zauvijek ostaje zanemaren. Ovaj fragment izvršenja nije niti jako A-pravedan: iako je radnju $enter_2$ moguće izvršiti beskonačno mnogo puta (svaki put iz stanja $\langle w_1, w_2, y = 1\rangle$ ili $\langle n_1, w_2, y = 1\rangle$), ona nikada nije izvršena. Ipak, on je slabo A-pravedan, budući da radnja $enter_2$ nije stalno moguća (u stanju $\langle c_1, w_2, y = 0\rangle$).

Poglavlje 4

Regularna i ω-regularna svojstva

Glavni cilj ovog poglavlja je upoznati algoritme koje koriste model checkeri i koji daju odgovor na pitanje zadovoljava li model sustava neko svojstvo. Da bismo to mogli učiniti, prvo je potrebno predstaviti koncept automata.

Prvo ćemo promatrati algoritam za provjeru klase *regularnih svojstava* sigurnosti. Za to će nam biti potrebni pojmovi konačnog automata, jezika automata i sinkroniziranog produkta. Zatim ćemo taj algoritam generalizirati na veću klasu LT svojstava, tzv. *ω-regularnih svojstava*. U ovu klasu svojstava ubrajaju se regularna svojstva sigurnosti, ali i mnoga druga važna svojstva kao što su razna svojstva napretka. Radi reprezentacije ω-regularnih svojstava, upoznat ćemo se s pojmom Büchijevog automata, inačicom konačnog automata koja prihvaća beskonačne riječi (za razliku od konačnog automata, koji prihvaća konačne riječi).

4.1 Automati na konačnim riječima

Definicija 4.1.1. *Nedeterministički konačni automat (NKA) $\mathcal{A}$ je petorka $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, gdje je*

- *Q konačan skup stanja,*
- *Σ konačan alfabet,*
- *$\delta : Q \times \Sigma \to 2^Q$ funkcija prijelaza*[1]*,*
- *$Q_0 \subseteq Q$ skup početnih stanja,*
- *$F \subseteq Q$ skup prihvatljivih (prihvaćajućih) stanja.*

[1] 2^Q je oznaka za partitivni skup skupa Q.

Veličina NKA-a $\mathcal{A}$, u oznaci $|\mathcal{A}|$, je broj stanja i prijelaza u $\mathcal{A}$:

$$|\mathcal{A}| = |Q| + \sum_{q \in Q} \sum_{A \in \Sigma} |\delta(q, A)|.$$

Σ definira skup simbola na kojima je automat definiran. Skup Q_0 (može biti i prazan, kao i skup F) određuje skup stanja u kojima automat može početi s radom. Funkcija prijelaza može se poistovijetiti s relacijom $\rightarrow \subseteq Q \times \Sigma \times Q$ zadanom s

$$q \xrightarrow{A} q' \iff q' \in \delta(q, A).$$

Zato se često koristi relacijska notacija za δ. Intuitivno, $q \xrightarrow{A} q'$ označava da automat može prijeći iz stanja q u stanje q' čitajući simbol A.

Opišimo ponašanje NKA na intuitivnoj razini. Automat kreće iz nekog stanja iz skupa Q_0 i dana mu je neka ulazna riječ $w \in \Sigma^*$. Automat čita ovu riječ znak po znak slijeva nadesno. Nakon što pročita jedan znak ulazne riječi, automat mijenja stanje s obzirom na relaciju prijelaza δ. Preciznije, ako je u stanju q pročitan simbol A, automat nedeterministički bira jedno od mogućih prijelaza $q \xrightarrow{A} q'$ (tj. jedno stanje q' takvo da $q' \in \delta(q, A)$) i prelazi u stanje q' gdje potom čita idući znak ulazne riječi. NKA ne može napraviti prijelaz ako njegovo trenutno stanje q nema "izlazni" prijelaz označen trenutno pročitanim simbolom A. U tom slučaju (ako je $\delta(q, A) = \emptyset$) automat ne može postići daljnji napredak i kažemo da je ulazna riječ *odbijena*. U slučaju da je cijela ulazna riječ pročitana, kažemo da automat staje[2]. Ako se automat pritom nalazi u nekom od prihvatljivih stanja, kažemo da automat *prihvaća* riječ, a inače da automat *odbija* riječ.

Definicija 4.1.2. *Neka je $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ NKA i $w = A_1...A_n \in \Sigma^*$ konačna riječ. Izvršenje za w u $\mathcal{A}$ je konačan niz stanja $q_0, q_1, ..., q_n \in Q$ takav da vrijede sljedeće dvije tvrdnje:*

- $q_0 \in Q_0$,
- $q_i \xrightarrow{A_{i+1}} q_{i+1}$, *za svaki* $0 \leq i < n$.

Kažemo da je izvršenje $q_0 q_1 ... q_n$ prihvatljivo ako dodatno vrijedi i:

- $q_n \in F$.

[2] eng. *halts*

Kažemo da $\mathcal{A}$ prihvaća riječ w ako postoji prihvatljivo izvršenje za w. Prihvatljivi jezik automata $\mathcal{A}$, označen s $\mathcal{L}(\mathcal{A})$, je skup svih konačnih riječi iz Σ^* koje $\mathcal{A}$ prihvaća, tj.

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \textit{postoji prihvatljivo izvršenje za } w \textit{ u } \mathcal{A}\}.$$

Definicija 4.1.3. *Neka su $\mathcal{A}$ i $\mathcal{A}'$ dva NKA s istim alfabetom. Kažemo da su $\mathcal{A}$ i $\mathcal{A}'$ ekvivalentni ako je $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

Glavni problem u teoriji automata je za dani NKA $\mathcal{A}$ odrediti je li njegov prihvatljivi jezik prazan, tj. je li $\mathcal{L}(\mathcal{A}) = \emptyset$. Taj problem poznat je pod nazivom *problem praznine*[3]. Iz definicije 4.1.2 slijedi da je $\mathcal{L}(\mathcal{A})$ neprazan ako i samo ako postoji barem jedno izvršenje koje završava u nekom prihvatljivom stanju. Drugim riječima, $\mathcal{L}(\mathcal{A})$ je neprazan ako i samo ako postoji prihvatljivo stanje $q \in F$ koje je dostupno iz početnog stanja $q_0 \in Q_0$. Ovo se može provjeriti algoritmom pretraživanja u dubinu ili algoritmom pretraživanja u širinu. Za stanje $q \in Q$, uvodimo oznaku $Reach(q)$ za skup svih stanja koja su dostupna putem proizvoljnog izvršenja koje počinje u stanju q.

Definicija 4.1.4. *Za NKA $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, Q_{0,i}, F_i)$, $i = 1, 2$, sinkrozinirani produkt automata $\mathcal{A}_1$ i $\mathcal{A}_2$ definiramo s*

$$\mathcal{A}_1 \otimes \mathcal{A}_2 = (Q_1 \times Q_2, \Sigma, \delta, Q_{0,1} \times Q_{0,2}, F_1 \times F_2),$$

gdje je δ definirana kao

$$\frac{q_1 \xrightarrow{A}_1 q_1' \wedge q_2 \xrightarrow{A}_2 q_2'}{(q_1, q_2) \xrightarrow{A} (q_1', q_2')}.$$

Pokazuje se da se riječ koju prihvaća automat $\mathcal{A}_1 \otimes \mathcal{A}_2$ nalazi u presjeku jezika $\mathcal{L}(\mathcal{A}_1)$ i $\mathcal{L}(\mathcal{A}_2)$, tj. da je

$$\mathcal{L}(\mathcal{A}_1 \otimes \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2).$$

Definicija 4.1.5. *Neka je $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ NKA. Za $\mathcal{A}$ kažemo da je deterministički konačni automat (DKA) ako je*

$$|Q_0| \leq 1 \quad i \quad |\delta(q, A)| \leq 1, \textit{ za sva stanja } q \in Q \textit{ i sve simbole } A \in \Sigma.$$

Kažemo da je DKA potpun ako je $|Q_0| = 1$ i $|\delta(q, A)| = 1$, za sve $q \in Q$ i sve $A \in \Sigma$.

[3] eng. *emptiness problem*

Dakle, NKA je deterministički ako ima najviše jedno početno stanje i za svaki simbol A sljedbenik svakog stanja q je ili jedinstveno definirano ili nedefinirano. U slučaju potpunog DKA, svako stanje q ima jedinstvenog sljedbenika, pa time i jedinstveno izvršenje za svaku ulaznu riječ.

Svaki DKA može se pretvoriti u ekvivalentan potpuni DKA dodavanjem neprihvatljivog stanja q_{self} i beskonačne petlje koju čini prijelaz $q_{self} \xrightarrow{A} q_{self}$, za bilo koji simbol $A \in \Sigma$. Iz svakog stanja $q \neq q_{self}$ postoji prijelaz u stanje q_{self} za bilo koji pročitani simbol $A \in \Sigma$ za koji q nema A-sljedbenika u zadanom nepotpunom DKA. Prihvatljivi jezik potpunog DKA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$[4] zadan je s

$$\mathcal{L}(\mathcal{A}) = \{\ w \in \Sigma^* \mid \delta^*(q_0, w) \in F\ \}^5.$$

Budući da potpuni DKA $\mathcal{A}$ ima točno jedno izvršenje za svaku ulaznu riječ, jednostavno je dobiti komplementarni potpuni DKA $\overline{\mathcal{A}}$; ako je $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ potpuni DKA, onda je $\overline{\mathcal{A}} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ potpuni DKA za kojeg vrijedi $\mathcal{L}(\overline{\mathcal{A}}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. Kad je riječ o NKA ili nepotpunom DKA, ova tehnika neće dati automat za komplementarni jezik.

Za svaki NKA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ moguće je konstruirati ekvivalentan potpuni DKA $\mathcal{A}_{det}$. Ta metoda poznata je pod nazivom *konstrukcija partitivnog skupa*[6]. Nju u ovom radu nećemo opisivati, no detaljnije o njoj može se pročitati u [1] i [6].

4.1.1 Opisivanje regularnog svojstva sigurnosti automatom

Sada ćemo pokazati kako provjeriti je li zadovoljeno *regularno* svojstvo sigurnosti koristeći NKA. Regularna svojstva sigurnosti čine važnu klasu svojstava sigurnosti, a njezina najbitnija karakteristika je činjenica da skup svih loših prefiksa nekog svojstva iz te klase čini *regularan jezik*. Za jezik $\mathcal{L}$ kažemo da je regularan ako postoji konačan automat $\mathcal{A}$ takav da vrijedi $\mathcal{L}(\mathcal{A}) = \mathcal{L}$.

Definicija 4.1.6. *Za svojstvo sigurnosti P_{safe} nad skupom AP kažemo da je regularno ako skup njegovih loših prefiksa čini regularni jezik nad skupom 2^{AP}.*

[4] Potpuni DKA često se piše u obliku $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, gdje q_0 označava jedinstveno početno stanje, a δ funkciju prijelaza $\delta : Q \times \Sigma \to Q$.

[5] Proširenu funkciju prijelaza δ^* potpunog DKA možemo promatrati kao funkciju $\delta^* : Q \times \Sigma^* \to Q$, koja danom stanju q i konačnoj riječi w pridružuje jedinstveno stanje $p = \delta^*(q, w)$ u koje je automat došao iz stanja q čitajući ulaznu riječ w.

[6] eng. *powerset construction*

Svaka invarijanta je regularno svojstvo sigurnosti. Naime, ako je Φ uvjet invarijante koji treba biti zadovoljen u svim dostupnim stanjima, onda se jezik svih loših prefiksa sastoji od riječi $A_0A_1...A_n$ takvih da $A_i \not\models \Phi$ za neki $0 \leq i \leq n$. Takvi jezici su regularni, budući da ih možemo okarakterizirati (neformalno) regularnom notacijom $\Phi^*(\neg\Phi)true^*$, gdje Φ predstavlja skup svih $A \subseteq AP$ takvih da $A \models \Phi$, $\neg\Phi$ skup svih $A \subseteq AP$ takvih da $A \not\models \Phi$, a *true* skup svih podskupova A skupa AP.

Neka je P_{safe} regularno svojstvo sigurnosti nad skupom atomarnih propozicija AP te neka je $\mathcal{A}$ NKA koji prepoznaje (minimalne) loše prefikse svojstva P_{safe}[7]. Pretpostavimo da $\varepsilon \notin \mathcal{L}(\mathcal{A})$. Nadalje, neka je TS konačni tranzicijski sustav bez završnih stanja s pripadnim skupom atomarnih propozicija AP. Cilj nam je predstaviti algoritam koji provjerava zadovoljava li TS regularno svojstvo sigurnosti P_{safe}, tj. vrijedi li $TS \models P_{safe}$. Za početak, definirajmo produkt tranzicijskog sustava i NKA.

Definicija 4.1.7. *Neka je $TS = (S, Act, \rightarrow, I, AP, L)$ tranzicijski sustav bez završnih stanja, te neka je $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ NKA s pripadnim alfabetom $\Sigma = 2^{AP}$ i vrijedi $Q_0 \cap F = \emptyset$. Produkt tranzicijskog sustava TS i NKA $\mathcal{A}$ je tranzicijski sustav $TS \otimes \mathcal{A}$ definiran na sljedeći način:*

$$TS \otimes \mathcal{A} = (S', Act, \rightarrow', I', AP', L'),$$

gdje je

- $S' = S \times Q$,
- $\rightarrow'$ *je najmanja relacija definirana pravilom* $\dfrac{s \xrightarrow{\alpha} t \wedge q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}{}' \langle t, p\rangle}$,
- $I' = \{\langle s_0, q\rangle \mid s_0 \in I \wedge \exists q_0 \in Q_0 : q_0 \xrightarrow{L(s_0)} q\}$,
- $AP' = Q$,
- $L' : S \times Q \rightarrow 2^Q$ *je zadana s* $L'(\langle s, q\rangle) = \{q\}$.

Neka su tranzicijski sustav TS i NKA $\mathcal{A}$ definirani kao ranije. Neka je $P_{inv(\mathcal{A})}$ invarijanta nad skupom $AP' = 2^Q$ koja je definirana formulom propozicionalne logike

$$\bigwedge_{q \in F} \neg q.$$

U nastavku ćemo kao pokratu za ovu formulu pisati $\neg F$.

[7]Može se pokazati da je ekvivalentno prihvaća li $\mathcal{A}$ sve loše prefikse ili sve minimalne loše prefikse svojstva P_{safe}.

Teorem 4.1.1. *Za tranzicijski sustav TS nad skupom AP, NKA $\mathcal{A}$ s alfabetom 2^{AP} kao ranije i regularno svojstvo sigurnosti P_{safe} nad skupom AP, tako da je $\mathcal{L}(\mathcal{A})$ jednak skupu (minimalnih) loših prefiksa svojstva P_{safe}, sljedeće tvrdnje su ekvivalentne:*

1. $TS \models P_{safe}$,
2. $Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \emptyset$,
3. $TS \otimes \mathcal{A} \models P_{inv(\mathcal{A})}$.

Formalni dokaz ovog teorema nećemo provoditi, može ga se pronaći u [1]. Teorem zapravo kaže da je za provjeru zadovoljava li tranzicijski sustav TS regularno svojstvo sigurnosti P_{safe} dovoljno provjeriti je li istina da niti jedno stanje $\langle s, q\rangle$ produkta $TS \otimes \mathcal{A}$ nije dostupno, pri čemu je q prihvatljivo stanje automata $\mathcal{A}$. Ova invarijanta "*prihvatljivo stanje od $\mathcal{A}$ nikada nije posjećeno*" (formalno zadana formulom $\neg F$) može se provjeriti korištenjem algoritma 1. Prisjetimo se, u slučaju da invarijanta nije zadovoljena, algoritam 1 nam daje protuprimjer. Taj protuprimjer zapravo je konačan fragment puta $\langle s_0, q_1\rangle\langle s_1, q_2\rangle...\langle s_n, q_{n+1}\rangle$ tranzicijskog sustava $TS \otimes \mathcal{A}$ koji vodi do prihvatljivog stanja od $\mathcal{A}$. Time dobivamo konačan početni fragment puta $s_0 s_1 ... s_n$ TS-a, čiji trag $trace(s_0 s_1 ... s_n) \in (2^{AP})^*$ prihvaća automat $\mathcal{A}$ (budući da ima prihvatljivo izvršenje $q_0 q_1 ... q_{n+1}$). Iz ovoga sada slijedi da je $trace(s_0 s_1 ... s_n)$ loš prefiks za P_{safe}. Dakle, $s_0 s_1 ... s_n$ pokazuje da svojstvo P_{safe} nije zadovoljeno, budući da $trace(\pi) \notin P_{safe}$ za sve puteve π TS-a koji počinju prefiksom $s_0 s_1 ... s_n$.

Korolar 4.1.2. *Neka su $TS, \mathcal{A}$ i P_{safe} kao u teoremu 4.1.1. Tada za svaki početni fragment puta $\langle s_0, q_1\rangle...\langle s_n, q_{n+1}\rangle$ produkta $TS \otimes \mathcal{A}$ vrijedi:*

$$ako\ q_1, ..., q_n \notin F\ i\ q_{n+1} \in F,\ onda\ trace(s_0 s_1 ... s_n) \in \mathcal{L}(\mathcal{A}).$$

Kao rezultat, za provjeru regularnog svojstva sigurnosti dobivamo idući algoritam.

Algoritam 2 Algoritam provjere modela za regularno svojstvo sigurnosti

Ulaz: konačni tranzicijski sustav TS i regularno svojstvo sigurnosti P_{safe}
Izlaz: "true" ako $TS \models P_{safe}$, inače "false" uz protuprimjer za P_{safe}

Neka je NKA $\mathcal{A}$ (sa skupom prihvatljivih stanja F) takav da je $\mathcal{L}(\mathcal{A})$ skup svih (minimalnih) loših prefiksa za P_{safe};
Konstruiraj produkt $TS \otimes \mathcal{A}$;
Provjeri invarijantu $P_{inv(\mathcal{A})}$ s formulom $\neg F = \bigwedge_{q \in F} \neg q$ na produktu $TS \otimes \mathcal{A}$.

if $TS \otimes \mathcal{A} \models P_{inv(\mathcal{A})}$ **then**
  return true
else
  Odredi početni fragment puta $\langle s_0, q_1 \rangle ... \langle s_n, q_{n+1} \rangle$ produkta $TS \otimes \mathcal{A}$ takav da je $q_{n+1} \in F$;
  return (false, $s_0 s_1 ... s_n$)
end if

4.2 Automati na beskonačnim riječima

Dosad smo opisali automate koji prihvaćaju konačne riječi, tj. nizove simbola konačne duljine, i time dobili temelj za provjeru regularnih svojstava sigurnosti. Sada želimo to generalizirati kako bismo dobili mehanizam za provjeru šire klase LT svojstava. Ona obuhvaća regularna svojstva sigurnosti, ali i razna svojstva napretka i mnoga druga važna svojstva. U tu svrhu, prvo ćemo predstaviti *ω-regularne* izraze i jezike te definirati *nedeterministički Büchijev automat*, vrstu konačnog automata koja prihvaća beskonačne riječi.

Beskonačne riječi nad alfabetom Σ su beskonačni nizovi $A_0 A_1 A_2 ...$ simbola $A_i \in \Sigma$. Sa Σ^ω označavat ćemo skup svih beskonačnih riječi nad Σ. Svaki podskup skupa Σ^ω zovemo jezik beskonačnih riječi ili ω-jezik. U nastavku, pojam jezika koristit ćemo za bilo koji podskup skupa $\Sigma^* \cup \Sigma^\omega$.

Za jezik $\mathcal{L} \subseteq \Sigma^*$, neka je $\mathcal{L}^\omega$ skup riječi iz $\Sigma^* \cup \Sigma^\omega$ nastalih konkatenacijom (proizvoljnih) riječi iz $\mathcal{L}$, tj.

$$\mathcal{L}^\omega = \{w_1 w_2 w_3 ... \mid w_i \in \mathcal{L}, i \geq 1\}.$$

$\mathcal{L}^\omega$ je ω-jezik, pod uvjetom da je $\mathcal{L} \subseteq \Sigma^+$, tj. da $\mathcal{L}$ ne sadrži praznu riječ ε.

U definiciji 4.2.2, koristimo operator konkatenacije $\mathcal{L}_1.\mathcal{L}_2$ koji "spaja" jezik konačnih riječi $\mathcal{L}_1$ s jezikom beskonačnih riječi $\mathcal{L}_2$. On je definiran s

$$\mathcal{L}_1.\mathcal{L}_2 = \{w\sigma \mid w \in \mathcal{L}_1, \sigma \in \mathcal{L}_2\}.$$

Također, prije definicije 4.2.2 trebamo formalno definirati pojam regularnog izraza.

Definicija 4.2.1. *Kažemo da je* R *regularni izraz ako je* R *nešto od sljedećeg:*

- *a, za neki simbol a alfabeta* Σ*;*
- ε*;*
- $\emptyset$*;*
- $(R_1 \cup R_2)$*, gdje su* R_1 *i* R_2 *regularni izrazi;*
- (R_1R_2)*, gdje su* R_1 *i* R_2 *regularni izrazi;*
- R_1^**, gdje je* R_1 *regularni izraz.*

Vrijednost regularnog izraza je jezik. Drugim riječima, jedan regularni izraz opisuje točno jedan jezik. Za regularni izraz R, s $\mathcal{L}(R)$ označavamo jezik kojeg on opisuje. Primjerice, za alfabet $\Sigma = \{0, 1\}$, vrijednost regularnog izraza $R = 1(0 \cup 1)^*$ je jezik svih riječi koje započinju s 1, tj. vrijedi $\mathcal{L}(R) = \{w \in \Sigma^* \mid w$ započinje s $1\}$.

Definicija 4.2.2. *ω-regularan izraz* G *nad alfabetom* Σ *je izraz oblika*

$$G = E_1F_1^\omega + ... + E_nF_n^\omega,$$

gdje je $n \geq 1$ *i* $E_1, ..., E_n, F_1, ..., F_n$ *su regularni izrazi nad* Σ *takvi da* $\varepsilon \notin \mathcal{L}(F_i)$*, za sve* $1 \leq i \leq n$*.*

Semantika ω-regularnog izraza G *je jezik beskonačnih rječi*

$$\mathcal{L}_\omega(G) = \mathcal{L}(E_1).\mathcal{L}(F_1)^\omega \cup ... \cup \mathcal{L}(E_n).\mathcal{L}(F_n)^\omega,$$

gdje $\mathcal{L}(E) \subseteq \Sigma^*$ *označava jezik (konačnih) riječi dobiven regularnim izrazom* E*.*

Dva ω-regularna izraza G *i* H *su ekvivalentna ako je* $\mathcal{L}_\omega(G) = \mathcal{L}_\omega(H)$*. Pišemo* $G \equiv H$*.*

Definicija 4.2.3. *Za jezik* $\mathcal{L} \subseteq \Sigma^\omega$ *kažemo da je ω-regularan ako je* $\mathcal{L} = \mathcal{L}_\omega(G)$ *za neki ω-regularan izraz* G *nad* Σ*.*

ω-regularni jezici zatvoreni su na uniju, presjek i komplement.[8]

Definicija 4.2.4. *LT svojstvo P nad skupom AP je ω-regularno ako je P ω-regularan jezik nad alfabetom 2^{AP}.*

Svaka invarijanta nad proizvoljnim skupom atomarnih propozicija AP je ω-regularno svojstvo, budući da se može opisati ω-regularnim izrazom Φ^ω, gdje je Φ uvjet te invarijante koji se može poistovjetiti s regularnim izrazom $\sum_{\substack{A \subseteq AP \\ A \models \Phi}} A$.

Nadalje, svako regularno svojstvo sigurnosti P_{safe} je ω-regularno svojstvo. To proizlazi iz činjenice da je njegov komplementarni jezik

$$(2^{AP})^\omega \setminus P_{safe} = \underbrace{BadPref(P_{safe})}_{\text{regularan jezik}} .(2^{AP})^\omega$$

ω-regularan jezik. To slijedi iz zatvorenosti ω-regularnih jezika na konkatenaciju.

Sada ćemo definirati nedeterministički Büchijev automat (NBA) – vrstu automata koja radi s beskonačnim riječima. Sintaksa NBA-a je ista kao sintaksa NKA-a, no te dvije vrste automata razlikuju se po semantici. Naime, prihvatljivi jezik NKA-a $\mathcal{A}$ je jezik konačnih riječi ($\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$), dok je prihvatljivi jezik NBA-a $\mathcal{A}$, kojeg označavamo s $\mathcal{L}_\omega(\mathcal{A})$, ω-jezik ($\mathcal{L}_\omega(\mathcal{A}) \subseteq \Sigma^\omega$).

Definicija 4.2.5. *Nedeterministički Büchijev automat (NBA) $\mathcal{A}$ je petorka $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, gdje su $Q, \Sigma, \delta, Q_0, F$ oznake iste kao kod NKA (definicija 4.1.1).*

Izvršenje za (beskonačnu) riječ $\sigma = A_0A_1A_2... \in \Sigma^\omega$ je beskonačan niz stanja $q_0, q_1, q_2, ... \in Q$ takav da vrijede sljedeće dvije tvrdnje:

- $q_0 \in Q_0$,
- $q_i \xrightarrow{A_i} q_{i+1}$, *za svaki* $i \geq 0$.

Kažemo da je izvršenje $q_0q_1q_2...$ prihvatljivo ako još dodatno vrijedi i:

- $q_i \in F$, *za beskonačno mnogo indeksa* $i \in \mathbb{N}$.

Kažemo da $\mathcal{A}$ prihvaća beskonačnu riječ σ ako postoji prihvatljivo izvršenje za σ. Prihvatljivi jezik NBA-a $\mathcal{A}$ je

$$\mathcal{L}_\omega(\mathcal{A}) = \{\sigma \in \Sigma^\omega \mid \textit{postoji prihvatljivo izvršenje za } \sigma \textit{ u } \mathcal{A}\}.$$

Veličina NBA-a $\mathcal{A}$, u oznaci $|\mathcal{A}|$, definira se kao broj stanja i prijelaza u $\mathcal{A}$.

[8] Argumentaciju ove tvrdnje moguće je pronaći u [1].

Intuitivno, kriterij prihvaćanja za NBA je taj da neko od njegovih prihvatljivih stanja bude posjećeno beskonačno mnogo puta. Dakle, prihvatljivi jezik NBA-a sastoji se od svih beskonačnih riječi za koje postoji izvršenje u kojem je neko prihvatljivo stanje posjećeno beskonačno mnogo puta. Uočimo da definicija NBA-a dopušta da skup F bude prazan. U tom slučaju očito nijedno izvršenje nije prihvatljivo. Ako je $Q_0 = \emptyset$, također ne postoji prihvatljivo izvršenje, budući da ne postoji izvršenje niti za jednu riječ.

Funkcija prijelaza može se poistovijetiti s relacijom $\rightarrow \subseteq Q \times \Sigma \times Q$ zadanom s

$$q \xrightarrow{A} p \iff p \in \delta(q, A).$$

Svako izvršenje za beskonačnu riječ $\sigma \in \Sigma^\omega$ je beskonačno.

Kasnije u ovom poglavlju koristit ćemo NBA za provjeru ω-regularnih svojstava, slično kao što smo koristili NKA za provjeru regularnih svojstava sigurnosti u odjeljku 4.1.1. U ovom slučaju, koristit ćemo alfabet $\Sigma = 2^{AP}$.

Pokažimo sada vezu između NBA i regularnih svojstava sigurnosti. Neka je P_{safe} regularno svojstvo sigurnosti nad skupom AP, te neka je $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ NKA koji prepoznaje jezik svih loših prefiksa svojstva P_{safe}. Za svako prihvatljivo stanje $q_F \in F$ možemo smatrati da mu je dodijeljena petlja $q_F \xrightarrow{A} q_F$, za svaki simbol $A \subseteq AP$.[9]

Ako interpretiramo $\mathcal{A}$ kao NBA, on prihvaća točno one beskonačne riječi $\sigma \in (2^{AP})^\omega$ koje narušavaju svojstvo P_{safe}. Drugačije rečeno,

$$\mathcal{L}_\omega(\mathcal{A}) = (2^{AP})^\omega \setminus P_{safe}.$$

Uočimo, $\mathcal{A}$ prihvaća sve loše prefikse, ne samo minimalne! Ako je $\mathcal{A}$ potpuni DKA, onda NBA $\overline{\mathcal{A}} = (Q, 2^{AP}, \delta, Q_0, Q \setminus F)$ prihvaća jezik $\mathcal{L}_\omega(\overline{\mathcal{A}}) = P_{safe}$.

Teorem 4.2.1. *Klasa jezika koje prihvaća neki NBA odgovara klasi ω-regularnih jezika.*

Ideja dokaza prethodnog teorema je pokazati da za svaki ω-regularni jezik postoji neki NBA koji ga prepoznaje, te da je jezik $\mathcal{L}_\omega(\mathcal{A})$ kojeg NBA $\mathcal{A}$ prepoznaje ω-regularan. Dokaz nećemo provoditi, kao ni za ostale nadolazeće tvrdnje. Sve formalne dokaze moguće je pronaći u [1].

Kao što je ranije već spomenuto, temeljni problem za svaku vrstu automata je problem praznine (pitanje je li prihvatljivi jezik danog automata prazan). Problem praznine za NBA

[9] Ovo možemo pretpostaviti, budući da je svako proširenje lošeg prefiksa također loš prefiks jer u sebi sadrži isti "loši događaj" koji narušava svojstvo.

može se riješiti pomoću algoritama koji pretražuju sva dostupna stanja u grafu te provjeravaju pripadaju li ona nekom ciklusu grafa. Jedan takav algoritam obradit ćemo u odjeljku 4.2.1.

Definicija 4.2.6. *Neka su $\mathcal{A}$ i $\mathcal{A}'$ dva NBA s istim alfabetom. Kažemo da su $\mathcal{A}$ i $\mathcal{A}'$ ekvivalentni ako je $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}')$. Pišemo $\mathcal{A} \equiv \mathcal{A}'$.*

Definicija 4.2.7. *Neka je $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ NBA. Za $\mathcal{A}$ kažemo da je deterministički Büchijev automat (DBA) ako je*

$$|Q_0| \leq 1 \quad i \quad |\delta(q, A)| \leq 1, \text{ za sva stanja } q \in Q \text{ i sve simbole } A \in \Sigma.$$

Kažemo da je DBA potpun ako je $|Q_0| = 1$ i $|\delta(q, A)| = 1$, za sve $q \in Q$ i sve $A \in \Sigma$.

Ponašanje DBA-a za zadanu ulaznu riječ očito je determinističko: ili će u nekom trenu "zapeti" u nekom stanju pokušavajući obraditi trenutno pročitani simbol ili postoji jedinstveno beskonačno izvršenje za danu ulaznu riječ. Ako je DBA potpun, prvi slučaj se nikada neće dogoditi, nego će za svaku ulaznu riječ iz Σ^ω postojati jedinstveno izvršenje.

Kao i kod potpunog DKA, funkcija prijelaza δ potpunog DBA može se proširiti do potpune funkcije $\delta^* : Q \times \Sigma^* \to Q$ na sljedeći način. Definiramo

$$\delta^*(q, \varepsilon) = q, \quad \delta^*(q, A) = \delta(q, A), \quad \delta^*(q, A_1A_2...A_n) = \delta^*(\delta(q, A_1), A_2...A_n).$$

Tada, za svaku beskonačnu riječ $\sigma = A_0A_1A_2... \in \Sigma^\omega$, izvršenje $q_0q_1q_2...$ od $\mathcal{A}$ za ulaznu riječ σ zadano je s

$$q_{i+1} = \delta^*(q_0, A_0...A_i), \text{ za svaki } i \geq 0,$$

pri čemu je q_0 jedinstveno početno stanje od $\mathcal{A}$. Prihvatljivi jezik potpunog DBA-a $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ je

$$\mathcal{L}_\omega(\mathcal{A}) = \{A_0A_1A_2... \in \Sigma^\omega \mid \delta^*(q_0, A_0...A_i) \in F \text{ za beskonačno mnogo indeksa i }\}.$$

Bitna razlika između konačnih automata i Büchijevih automata je ta da, za razliku od konačnih automata u čijem slučaju je klasa jezika prihvatljivih za neki NKA jednaka klasi jezika prihvatljivih za neki DKA, u slučaju Büchijevih automata NBA-i opisuju veću klasu jezika nego DBA-i. To pokazuje sljedeći teorem: postoji ω-regularan jezik prihvatljiv za neki NBA takav da ne postoji DBA koji ga prihvaća. To je jezik zadan izrazom $(\mathrm{A} + \mathrm{B})^* B^\omega$.

Teorem 4.2.2. *Ne postoji DBA $\mathcal{A}$ takav da je $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega((\mathrm{A} + \mathrm{B})^*\mathrm{B}^\omega)$.*

4.2.1 Provjera ω-regularnih svojstava

Sada ćemo vidjeti kako se pristup temeljen na automatima za provjeru regularnih svojstava sigurnosti može poopćiti za provjeru ω-regularnih svojstava. Cilj nam je za zadani tranzicijski sustav $TS = (S, Act, \rightarrow, I, AP, L)$ i ω-regularno svojstvo P algoritamski provjeriti vrijedi li $TS \models P$. Slično kao za regularna svojstva sigurnosti, algoritam za verifikaciju trebao bi pokazivati da $TS \not\models P$ generirajući protuprimjer - put π tranzicijskog sustava TS takav da $trace(\pi) \notin P$. Ako takav put ne postoji, TS zadovoljava svojstvo P.

U tu svrhu, pretpostavimo da je skup "loših tragova" reprezentiran NBA-om $\mathcal{A}$, tj. $\mathcal{A}$ prepoznaje jezik $\overline{P} = (2^{AP})^\omega \setminus P$. Dakle, $\mathcal{L}_\omega(\mathcal{A}) = \overline{P}$. Tada imamo:

$$\begin{aligned} & TS \not\models P \\ \iff & Traces(TS) \not\subseteq P \\ \iff & Traces(TS) \cap \left((2^{AP})^\omega \setminus P\right) \neq \emptyset \\ \iff & Traces(TS) \cap \overline{P} \neq \emptyset \\ \iff & Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \emptyset. \end{aligned}$$

Dakle, problem se svodi na provjeru vrijedi li $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \emptyset$. Ovo možemo provjeriti na sličan način kao kod provjere regularnih svojstava sigurnosti. Konstruiramo produkt $TS \otimes \mathcal{A}$ koji predstavlja kombinaciju puteva TS-a i izvršenja automata $\mathcal{A}$, a zatim provjerimo postoji li u tom produktu put u kojem je neko prihvatljivo stanje od $\mathcal{A}$ posjećeno beskonačno mnogo puta. Ako postoji, algoritam za provjeru daje nam protuprimjer koji pokazuje da $TS \not\models P$. Ako takav put ne postoji, tj. ako svako prihvatljivo stanje može biti posjećeno najviše konačno mnogo puta za sve puteve u produktu, onda su sva izvršenja neprihvatljiva, pa slijedi da je $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \emptyset$, a onda i $TS \models P$.

U nastavku ćemo detaljnije obraditi ovaj postupak. No prije toga definirat ćemo *svojstvo perzistencije*[10]. To je LT svojstvo pomoću kojeg ćemo formalno moći izraziti da je neko prihvatljivo stanje posjećeno najviše konačno mnogo puta. Zatim ćemo iskazati teorem koji tvrdi da se provjera ω-regularnih svojstava može svesti na provjeru svojstva perzistencije.

Definicija 4.2.8. *Svojstvo perzistencije nad skupom AP je LT svojstvo $P_{pers} \subseteq (2^{AP})^\omega$ "Φ vrijedi uvijek počevši od nekog trena", za neku formulu propozicionalne logike Φ nad AP. Formalno,*

$$P_{pers} = \left\{A_0 A_1 A_2 ... \in (2^{AP})^\omega \mid \exists i \geq 0 : \forall j \geq i : A_j \models \Phi\right\}.$$

Formulu Φ zovemo uvjetom perzistencije svojstva P_{pers}.

[10] eng. *persistence property*

Definicija 4.2.9. *Neka je $TS = (S, Act, \rightarrow, I, AP, L)$ tranzicijski sustav bez završnih stanja, te neka je $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ NBA takav da $\delta(q, A) \neq \emptyset$, za svako stanje q i svaki simbol $A \in 2^{AP}$. Produkt tranzicijskog sustava TS i NBA $\mathcal{A}$ je tranzicijski sustav $TS \otimes \mathcal{A}$ definiran na sljedeći način:*

$$TS \otimes \mathcal{A} = (S', Act, \rightarrow', I', AP', L'),$$

gdje je

- $S' = S \times Q$,
- $\rightarrow'$ *je najmanja relacija definirana pravilom* $\dfrac{s \xrightarrow{\alpha} t \wedge q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}{}' \langle t, p\rangle}$,
- $I' = \{\langle s_0, q\rangle \mid s_0 \in I \wedge \exists q_0 \in Q_0 : q_0 \xrightarrow{L(s_0)} q\}$,
- $AP' = Q$,
- $L' : S \times Q \rightarrow 2^Q$ *je zadana s* $L'(\langle s, q\rangle) = \{q\}$.

Nadalje, s $P_{pers(\mathcal{A})}$ označavamo svojstvo perzistencije nad skupom $AP' = Q$, zadano s

"od nekog trena zauvijek $\neg F$",

gdje $\neg F$ označava propozicionalnu formulu $\bigwedge_{q \in F} \neg q$ nad $AP' = Q$.

Teorem 4.2.3. *Neka je TS konačan tranzicijski sustav nad AP bez završnih stanja te neka je P ω-regularno svojstvo nad AP. Nadalje, neka je $\mathcal{A}$ NBA s alfabetom 2^{AP} takav da $\delta(q, A) \neq \emptyset$, za svako stanje q i svaki simbol $A \in 2^{AP}$, te takav da je $\mathcal{L}_\omega(\mathcal{A}) = (2^{AP})^\omega \setminus P$. Tada su sljedeće tvrdnje ekvivalentne:*

1. $TS \models P$,
2. $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \emptyset$,
3. $TS \otimes \mathcal{A} \models P_{pers(\mathcal{A})}$.

Sada treba vidjeti kako za zadani tranzicijski sustav TS i svojstvo perzistencije P_{pers} ustanoviti vrijedi li $TS \not\models P_{pers}$. Neka je Φ propozicionalna formula koja određuje uvjet perzistencije koji mora vrijediti "od nekog trena zauvijek". Pretpostavimo da je s stanje TS-a koje je dostupno iz početnog stanja od TS i za koje $s \not\models \Phi$ (formula Φ nije istinita u stanju s). Jer je s dostupno, postoji početni fragment puta u TS koji završava u s. Ako s

pripada nekom ciklusu, onda se na ovaj fragment puta može nastavljati beskonačni put dobiven beskonačnim "kruženjem" po tom ciklusu. Tako dobivamo put u TS-u koji posjećuje stanje s beskonačno mnogo puta. No tada slijedi $TS \not\models P_{pers}$.

Teorem 4.2.4. *Neka je TS konačan tranzicijski sustav bez završnih stanja nad AP, neka je Φ formula propozicionalne logike nad AP, te neka je P_{pers} svojstvo perzistencije "od nekog trena zauvijek Φ". Tada su sljedeće tvrdnje ekvivalentne:*

1. $TS \not\models P_{pers}$,
2. *postoji dostupno stanje s takvo da $s \not\models \Phi$ i s pripada ciklusu.*

Objasnimo kako pokazati da $TS \not\models P_{pers}$. Neka je $\hat{\pi} = u_0u_1u_2...u_k$ put u grafu izvedenom iz tranzicijskog sustava TS, $G(TS)$, takav da je $k > 0$ i $s = u_0 = u_k$. Pretpostavimo da $s \not\models \Phi$. Dakle, $\hat{\pi}$ je ciklus u $G(TS)$ koji sadrži stanje koje narušava Φ. Neka je $s_0s_1s_2...s_n$ početni fragment puta u TS-u za koji vrijedi $s_n = s_0$. Konkatenacijom tog početnog fragmenta puta i "razmotanog" ciklusa dobivamo put

$$\pi = s_0s_1s_2... \underbrace{s_n}_{=s} u_1u_2... \underbrace{u_k}_{=s} u_1u_2... \underbrace{u_k}_{=s} ...$$

u TS-u. Budući da je stanje $s \not\models \Phi$ u putu π posjećeno beskonačno mnogo puta, π ne zadovoljava "od nekog trena zauvijek Φ".

Prema teoremu 4.2.4, da bismo provjerili zadovoljenost svojstva perzistencije dovoljno je provjeriti postoji li dostupni ciklus koji sadrži neko stanje koje narušava Φ. Jedan od načina kako to možemo napraviti je pomoću tzv. *ugniježđenog DFS algoritma.*

Prisjetimo se, na razini grafova, za konačan usmjeren graf G i vrh grafa v moguće je provjeriti pripada li v ciklusu. Potrebno je iz vrha v započeti s pretraživanjem u dubinu, te za svaki posjećeni vrh w provjeriti postoji li brid od w do v. Ako postoji, pronašli smo ciklus koji počinje u v, slijedi put u grafu do vrha w (taj put odgovara trenutnom sadržaju stoga kojeg koristimo u DFS algoritmu), te završava bridom iz w do početnog vrha v. U suprotnom, v ne pripada ciklusu. Takve bridove koji zatvaraju ciklus zovemo *backword bridovima*. Slična tehnika može se primijeniti za provjeru postoji li ciklus u grafu G.

Osnovna ideja ugniježđenog DFS algoritma je provesti dva pretraživanja u dubinu na isprepleten način; "vanjski" DFS traži sva dostupna stanja gdje je $\neg\Phi$, dok "unutrašnji" DFS traži backward bridove koji vode do nekog od tih $\neg\Phi$-stanja. Unutrašnji DFS je ugniježđen u vanjski na sljedeći način: nakon što je vanjski DFS pronašao sva dostupna $\neg\Phi$-stanja, za svako takvo stanje s on započinje postupak koji provodi još jedan, unutarnji, DFS na tom stanju. Taj postupak sastoji se u tome da se posjećuju sva stanja s' koja su

dostupna iz s i koja još dosad nisu bila posjećena. Ako nije pronađen nijedan backward brid iz nekog takvog stanja s' do s, onda vanjski DFS nastavlja isti postupak sa sljedećim dostupnim $\neg\Phi$-stanjem, koje još dosad nije bilo obrađeno.

Algoritam 4 prikazuje pseudokod za vanjski DFS (reachable_cycle), koji poziva unutrašnji DFS (cycle_check) prikazan algoritmom 3. U slučaju da TS ne zadovoljava svojstvo perzistencije, ugniježđeni DFS algoritam nam daje protuprimjer: nakon što pronađe ciklus koji sadrži $\neg\Phi$-stanje s, put do s lako se odredi pomoću stogova. Stog U vanjskog DFS-a sadrži fragment puta od početnog stanja s_0 do s (u obrnutom redoslijedu), dok stog V unutrašnjeg DFS-a sadrži ciklus os s do s (u obrnutom redoslijedu). Konkatenacijom ovih fragmenata puta dobivamo traženi protuprimjer.

Algoritam 3 Unutrašnji DFS - otkrivanje ciklusa

Ulaz: konačni tranzicijski sustav TS i stanje s iz TS takvo da $s \not\models \Phi$
Izlaz: "true" ako s pripada ciklusu u TS, inače "false"

```
procedure cycle_check (state s)
  bool cycle_found := false;                          (* još nije pronađen ciklus *)
  push(s, V);                                         (* stavi s na vrh stoga *)
  T := T ∪ {s};                                       (* označi s kao posjećeno *)
  repeat
    s' := top(V);                                     (* uzmi element s vrha stoga *)
    if S ∈ Post(s') then
      cycle_found := true;          (* ako je stanje s sljedbenik stanja s', našli smo ciklus*)
      push(s, V);                                     (* stavi s na stog *)
    else
      if Post(s') \ T ≠ ∅ then
        let s'' ∈ Post(s') \ T;
        push(s'', V);               (* stavi dosad neposjećenog sljedbenika od s' na stog *)
        T := T ∪ {s''};                   (* i označi ga kao posjećenog (dostupnog) *)
      else
        pop(V);
      end if
    end if
  until ((V = ε) ∨ cycle_found)                       (* stani kad je stog prazan ili je nađen ciklus *)
  return cycle_found
endproc
```

Algoritam 4 Provjera svojstva perzistencije ugniježđenim DFS algoritmom

Ulaz: tranzicijski sustav TS bez završnih stanja i propozicionalna formula Φ
Izlaz: "da" ako $TS \models$ "od nekog trena zauvijek" Φ, inače "ne" uz protuprimjer

```
set of states R := ∅;                          (* skup posjećenih stanja u vanjskom DFS *)
stack of states U := ε;                        (* prazni stog za vanjski DFS *)
set of states T := ∅;                          (* skup posjećenih stanja u unutrašnjem DFS *)
stack of states V := ε;                        (* prazni stog za unutrašnji DFS *)
bool cycle_found := false;

  while (I \ R ≠ ∅ ∧ ¬cycle_found) do
    let s ∈ I \ R;                             (* istraži dostupna početna stanja *)
    reachable_cycle(s);                        (* vanjski DFS *)
  end while
  if ¬cycle_found then
    return("da")                               (* TS ⊨ "od nekog trena zauvijek Φ" *)
  else
    return("ne", reverse(V.U))                 (* protuprimjer se dobiva iz sadržaja stoga *)
  end if
```

```
procedure reachable_cycle (state s)
  push(s, U);                                  (* stavi s na vrh stoga *)
  R := R ∪ {s};                                (* označi s kao posjećeno *)
  repeat
    s' := top(U);
    if Post(s') \ R ≠ ∅ then
      let s'' ∈ Post(s') \ R;
      push(s'', U);                            (* stavi na stog sljedbenika od s' koji nije posjećen *)
      R := R ∪ {s''};                          (* označi s'' kao posjećeno *)
    else
      pop(U);                                  (* vanjski DFS za s' je dovršen *)
      if s' ⊭ Φ then
        cycle_found := cycle_check(s');        (* nastavi s unutrašnjim DFS u stanju s' *)
      end if
    end if
  until ((U = ε) ∨ cycle_found)                (* stani kad je stog vanjskog DFS prazan )
                                               ( ili je nađen ciklus *)
endproc
```

Na kraju ovog poglavlja ukratko ćemo predstaviti još jednu vrstu automata koja je dobivena malom modifikacijom NBA-a - *generalizirani nedeterministički Büchijev automat* (GNBA). Razlika između NBA i GNBA je u tome što uvjet prihvaćanja za GNBA zahtijeva da više skupova $F_1, ..., F_k$, tzv. *prihvatljivih skupova*, bude posjećeno beskonačno mnogo puta (tj. da neko od stanja iz svakog skupa $F_i \subseteq Q, 1 \leq i \leq k$, bude posjećeno beskonačno mnogo puta), dok se u slučaju NBA zahtijevalo da bar jedno od završnih stanja $s \in F$ bude posjećeno beskonačno mnogo puta. Slijedi formalna definicija GNBA, a zatim neki važni rezultati za GNBA.

Definicija 4.2.10. *Generalizirani nedeterministički Büchijev automat (GNBA) je petorka $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, gdje su Q, Σ, δ, Q_0 definirani isto kao kod NBA, a $\mathcal{F}$ je podskup od 2^Q (može biti i prazan) čije elemente $F \in \mathcal{F}$ nazivamo prihvatljivim skupovima.*

Izvršenje u $\mathcal{G}$ za beskonačnu riječ $A_0A_1... \in \Sigma^\omega$ je beskonačan niz stanja $q_0q_1q_2... \in Q^\omega$ takav da je

$$q_0 \in Q_0 \quad i \quad q_{i+1} \in \delta(q_i, A_i), \text{ za svaki } i \geq 0.$$

Za beskonačno izvršenje $q_0q_1q_2...$ kažemo da je prihvatljivo ako

$$\forall F \in \mathcal{F} : \Big(\text{postoji beskonačno mnogo indeksa } j \in \mathbb{N} : q_j \in F \Big).$$

Prihvatljivi jezik GNBA-a $\mathcal{G}$ je

$$\mathcal{L}_\omega(\mathcal{G}) = \{\sigma \in \Sigma^\omega \mid \text{postoji prihvatljivo izvršenje za } \sigma \text{ u } \mathcal{G}\}.$$

Kažemo da su GNBA $\mathcal{G}_1$ i GNBA $\mathcal{G}_2$ ekvivalentni ako je $\mathcal{L}_\omega(\mathcal{G}_1) = \mathcal{L}_\omega(\mathcal{G}_2)$.

Veličina GNBA-a $\mathcal{G}$, u oznaci $|\mathcal{G}|$, jednaka je broju stanja i prijelaza u $\mathcal{G}$.

Teorem 4.2.5. *Za svaki GNBA $\mathcal{G}$ postoji NBA $\mathcal{A}$ takav da je $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A})$.*

Korolar 4.2.6. *Klasa jezika koji su prihvatljivi za neki GNBA odgovara klasi ω-regularnih jezika.*

Poglavlje 5

Linearna temporalna logika

U ovom poglavlju bavimo se *linearnom temporalnom logikom*[1] (*LTL*), najčešćim i najpogodnijim logičkim formalizmom za opisivanje LT svojstava. Razni alati za provjeru modela koriste LTL kao jezik za specificiranje svojstava. Među njima je i model checker NuSMV, koji će se koristiti u praktičnom dijelu rada, a bit će predstavljen u poglavlju 6.

5.1 Sintaksa i semantika

Temeljne gradivne jedinice LTL formule su atomarne propozicije – oznake stanja $a \in AP$, zatim logički veznici – konjunkcija $\wedge$ i negacija $\neg$, te dva osnovna temporalna oblika – "*sljedeći*" $\bigcirc$ i "*sve dok*" **U**.

Definicija 5.1.1. *LTL formula nad skupom atomarnih propozicija AP ($a \in AP$) je bilo koji izraz generiran sljedećom gramatikom:*

$$\varphi := true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \boldsymbol{U} \varphi_2 \ .$$

Koristeći veznike $\wedge$ i $\neg$, obuhvaćene su sve LTL formule. Ostali logički veznici poput disjunkcije $\vee$, implikacije $\rightarrow$ i ekvivalencije $\leftrightarrow$ mogu se izvesti iz konjunkcije i negacije

[1] eng. *linear temporal logic*. Pridjev "temporalna" u nazivu LTL odnosi se na specificiranje relativnog poretka događaja, iako intuitivno upućuje na vezu s ponašanjem sustava u realnom vremenu. Npr. "Poruka je primljena nakon što je poslana." ili "Auto se zaustavlja kada vozač pritisne kočnicu." Dakle, nikada se ne odnosi na preciziranje točnog vremena događaja.

na sljedeći način:

$$\varphi_1 \vee \varphi_2 \stackrel{def}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$$
$$\varphi_1 \rightarrow \varphi_2 \stackrel{def}{=} \neg\varphi_1 \vee \varphi_2$$
$$\varphi_1 \leftrightarrow \varphi_2 \stackrel{def}{=} (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$$

Redoslijed prioriteta LTL operatora je sljedeći:

- Unarni operatori imaju veći prioritet nego binarni.
- $\neg$ i $\bigcirc$ imaju jednak prioritet.
- **U** ima veći prioritet nego $\wedge, \vee, \rightarrow$.

Zagrade se izostavljaju kad god nisu nužne, primjerice, pišemo

- $\neg\varphi_1$ **U** $\bigcirc\varphi_2$ umjesto $(\neg\varphi_1)$ **U** $(\bigcirc\varphi_2)$.

Operator **U** je desno asociran, npr.

- φ_1 **U** φ_2 **U** φ_3 zapravo znači φ_1 **U** $(\varphi_2$ **U** $\varphi_3)$.

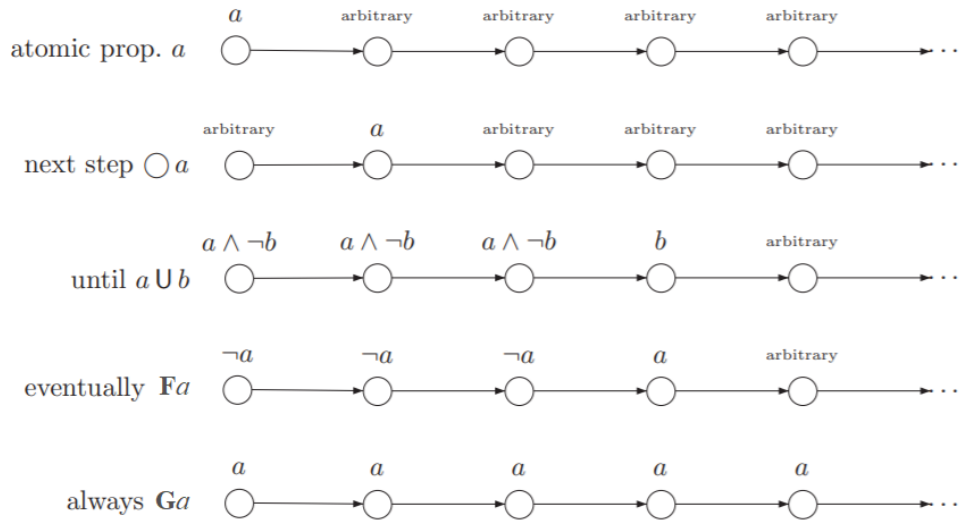
Pomoću operatora **U** moguće je izvesti i temporalne oblike **F** – "sada ili nekada u budućnosti" i **G** – "uvijek":

$$\mathbf{F}\varphi \stackrel{def}{=} true\mathbf{U}\varphi$$
$$\mathbf{G}\varphi \stackrel{def}{=} \neg\mathbf{F}\neg\varphi$$

Dakle, $\mathbf{F}\varphi$ znači da će φ prije ili kasnije biti istinita. $\mathbf{G}\varphi$ je istinita ako i samo ako nije točno da će u nekom trenu vrijediti $\neg\varphi$, drugim riječima, φ vrijedi sada i zauvijek. Kombiniranjem temporalnih oblika **F** i **G** dobivamo nove korisne temporalne oblike kao što su **GF** – "beskonačno mnogo puta u budućnosti" i **FG** – "od nekog trena zauvijek".

Slika 5.1 ilustrira intuitivno značenje temporalnih oblika na jednostavnom primjeru gdje su oblici primijenjeni na atomarne propozicije iz skupa $AP = \{a, b\}$. Na lijevoj strani slike navedene su neke LTL formule, dok su s desne strane prikazani nizovi stanja (tj. putevi).[2]

[2]Slika preuzeta iz [1]



Slika 5.1: Intuitivno značenje temporalnih oblika

Semantika LTL formule φ definirana je preko jezika $Words(\varphi)$ koji sadrži sve beskonačne riječi nad alfabetom 2^{AP} koje zadovoljavaju φ. Drugim riječima, svakoj LTL formuli φ možemo pridružiti jedinstveni skup $Words(\varphi) \subseteq (2^{AP})^{\omega}$. Budući da smo LT svojstvo nad skupom AP definirali kao podskup skupa $(2^{AP})^{\omega}$ (definicija 3.1.6), slijedi da svakoj LTL formuli možemo pridružiti jedinstveno LT svojstvo nad AP.

Definicija 5.1.2. *Neka je φ LTL formula nad AP. LT svojstvo inducirano formulom φ je skup*

$$Words(\varphi) = \left\{\sigma \in (2^{AP})^{\omega} \mid \sigma \models \varphi\right\},$$

gdje je relacija "zadovoljava" $\models \subseteq (2^{AP})^{\omega} \times LTL$ najmanja relacija takva da za svaku beskonačnu riječ $\sigma = A_0A_1A_2...$ vrijede sljedeća svojstva:

$\sigma \models true$
$\sigma \models a$ *akko* $a \in A_0$ (*npr.* $A_0 \models a$)
$\sigma \models \varphi_1 \wedge \varphi_2$ *akko* $\sigma \models \varphi_1$ *i* $\sigma \models \varphi_2$
$\sigma \models \neg\varphi$ *akko* $\sigma \not\models \varphi$
$\sigma \models \bigcirc\varphi$ *akko* $\sigma[1...] = A_1A_2A_3... \models \varphi$
$\sigma \models \varphi_1 \boldsymbol{U} \varphi_2$ *akko* $\exists j \geq 0 : \sigma[1...] \models \varphi_2$ *i* $\sigma[i...] \models \varphi_1$, *za svaki* $0 \leq i < j$

Za riječ $\sigma = A_0A_1A_2... \in (2^{AP})^{\omega}$, $\sigma[j...] = A_jA_{j+1}A_{j+2}...$ je sufiks riječi σ koji počinje u $(j+1)$-om simbolu A_j.

Za izvedene operatore **F** i **G**, te njihove kombinacije semantika je sljedeća:

$$\sigma \models \mathbf{F}\varphi \quad \text{akko} \quad \exists j \geq 0 : \sigma[j...] \models \varphi$$
$$\sigma \models \mathbf{G}\varphi \quad \text{akko} \quad \forall j \geq 0 : \sigma[j...] \models \varphi$$

$$\sigma \models \mathbf{GF}\varphi \quad \text{akko} \quad \overset{\infty}{\exists} j : \sigma[j...] \models \varphi$$
$$\sigma \models \mathbf{FG}\varphi \quad \text{akko} \quad \overset{\infty}{\forall} j : \sigma[j...] \models \varphi$$

Ovdje koristimo pokrate $\overset{\infty}{\exists} j$ i $\overset{\infty}{\forall} j$:
$\overset{\infty}{\exists} j$ znači $(\forall i \geq 0)(\exists j \geq i)$, tj. "za beskonačno mnogo $j \in \mathbb{N}$",
$\overset{\infty}{\forall} j$ znači $(\exists i \geq 0)(\forall j \geq i)$, tj. "za gotovo sve $j \in \mathbb{N}$" (sve osim njih konačno mnogo).

U definiciji 5.1.2 semantiku LTL formule interpretirali smo preko skupa riječi. Sada ćemo definirati semantiku LTL formule s obzirom na tranzicijski sustav. Po definiciji 3.1.7, stanje s zadovoljava formulu φ ako svi putevi koji počinju u s zadovoljavaju φ. Tranzicijski sustav TS zadovoljava φ ako TS zadovoljava LT svojstvo $Words(\varphi)$, tj. ako svi putevi u TS-u koji počinju u nekom početnom stanju TS-a zadovoljavaju φ.

Definicija 5.1.3. *Neka je $TS = (S, Act, \to, I, AP, L)$ tranzicijski sustav bez završnih stanja, te neka je φ LTL formula nad AP. Tada relaciju "zadovoljava", $\models$, definiramo na sljedeći način:*

za beskonačan fragment puta π u TS-u,

$$\pi \models \varphi \quad akko \quad trace(\pi) \models \varphi;$$

za stanje $s \in S$,

$$s \models \varphi \quad akko \quad (\forall \pi \in Paths(s) : \pi \models \varphi;$$

za tranzicijski sustav TS,

$$TS \models \varphi \quad akko \quad Traces(TS) \subseteq Words(\varphi).$$

Iz ove i prethodnih definicija proizlazi sljedeći niz ekvivalencija:

$$\begin{aligned} TS \models \varphi \quad &\overset{\text{def. 5.1.3}}{\Longleftrightarrow} \quad Traces(TS) \subseteq Words(\varphi) \\ &\overset{\text{def. 3.1.7}}{\Longleftrightarrow} \quad TS \models Words(\varphi) \\ &\overset{\text{def. 5.1.2}}{\Longleftrightarrow} \quad \pi \models \varphi, \text{ za sve puteve } \pi \in Paths(TS) \\ &\overset{\text{def. 5.1.3}}{\Longleftrightarrow} \quad s_0 \models \varphi, \text{ za sva početna stanja } s_0 \in I \text{ } TS\text{-a}. \end{aligned}$$

5.2 Algoritmi za LTL provjeru modela

U ovom odjeljku dan je algoritam koji za konačni tranzicijski sustav TS bez završnih stanja i LTL formulu φ, koja formalizira zahtjev na TS, ispituje vrijedi li $TS \models \varphi$. Ako ne vrijedi, taj algoritam nam na neki način treba dati protuprimjer – konačan prefiks beskonačnog puta u TS za koji formula φ ne vrijedi.

Algoritam provjere modela predstavljen u ovom odjeljku temelji se na činjenici da se svaka LTL formula φ može prikazati nedeterminističkim Büchijevim automatom. Osnovna ideja je pokušati pokazati $TS \not\models \varphi$ tražeći put π u TS-u takav da $\pi \models \neg\varphi$. Ako je takav put pronađen, prefiks od π čini protuprimjer. U suprotnom, zaključuje se da $TS \models \varphi$.

$$
\begin{aligned}
& TS \models \varphi \\
\iff\ & Traces(TS) \subseteq Words(\varphi) \\
\iff\ & Traces(TS) \cap ((2^{AP})^{\omega} \setminus Words(\varphi)) = \emptyset \\
\iff\ & Traces(TS) \cap Words(\neg\varphi) = \emptyset \\
\iff\ & Traces(TS) \cap \mathcal{L}_{\omega}(\mathcal{A}) = \emptyset, \text{ za NBA } \mathcal{A} \text{ takav da je } \mathcal{L}_{\omega}(\mathcal{A}) = Words(\neg\varphi)
\end{aligned}
$$

Dakle, problem provjere svojstva φ na tranzicijskom sustavu TS svodi se na provjeru je li presjek $Traces(TS) \cap \mathcal{L}_{\omega}(\mathcal{A})$ prazan skup, pri čemu je prvo potrebno konastuirati NBA $\mathcal{A}$ za negaciju formule φ koja opisuje "nepoželjno ponašanje".

Algoritam 5 LTL provjera modela temeljena na automatima

Ulaz: konačni tranzicijski sustav TS nad AP i LTL formula φ nad AP
Izlaz: "da" ako $TS \models \varphi$, inače "ne" uz protuprimjer

---

Konstruiraj NBA $\mathcal{A}_{\neg\varphi}$ takav da je $\mathcal{L}_{\omega}(\mathcal{A}_{\neg\varphi}) = Words(\neg\varphi)$;
Konstruiraj produkt $TS \otimes \mathcal{A}_{\neg\varphi}$;

if postoji put π u $TS \otimes \mathcal{A}$ koji zadovoljava uvjet prihvaćanja za $\mathcal{A}$ **then**
  return "ne" i loši prefiks od π kao protuprimjer
else
  return "da"
end if

Ključni korak za postupak provjere modela prikazan algoritmom 5 je konstrukcija NBA $\mathcal{A}$ za LTL formulu $\neg\varphi$ tako da vrijedi $\mathcal{L}_\omega(\mathcal{A}) = Words(\neg\varphi)$. U nastavku će biti opisano kako se to može napraviti algoritamski. Prvo je potrebno konstruirati generalizirani NBA $\mathcal{G}$ (definiran u 4.2.10) za formulu $\neg\varphi$, koji se zatim transformira u ekvivalentni NBA (on postoji po teoremu 4.2.5).

Po definiciji 5.1.2, za zadanu LTL formulu φ postoji jedinstveni jezik $Words(\varphi) \subseteq (2^{AP})^\omega$ koji joj je pridružen. Time dobivamo da je pripadni alfabet GNBA-a za formulu φ $\Sigma = 2^{AP}$. GNBA $\mathcal{G}$ čija je familija prihvatljivih skupova $\mathcal{F}$ jednočlana možemo jednostavno smatrati NBA-om, a ako je $\mathcal{F} = \emptyset$, jezik $\mathcal{L}_\omega(\mathcal{G})$ čine sve beskonačne riječi za koje postoji beskonačno izvršenje u $\mathcal{G}$, pa $\mathcal{G}$ možemo smatrati NBA-om čija su sva stanja prihvatljiva.

Neka je φ LTL formula nad skupom AP za koju treba konstruirati GNBA $\mathcal{G}_\varphi$ nad 2^{AP} tako da je $\mathcal{L}_\omega(\mathcal{G}_\varphi) = Words(\varphi)$. Pretpostavimo da φ sadrži samo operatore $\wedge, \neg, \bigcirc$ i $\mathbf{U}$ (izvedeni operatori $\vee, \rightarrow, \mathbf{F}, \mathbf{G}$ izraženi su pomoću osnovnih operatora), te da $\varphi \neq true$ (jer je slučaj $\varphi = true$ trivijalan). Nadalje, neka je $\sigma = A_0A_1A_2... \in Words(\varphi)$.

Proširimo skupove $A_i \subseteq AP$ potformulama ψ od φ i njihovim negacijama tako da dobijemo beskonačnu riječ $\overline{\sigma} = B_0B_1B_2...$ za koju vrijedi:

$$\psi \in B_i \iff \underbrace{A_iA_{i+1}A_{i+2}...}_{\sigma^i} \models \psi.$$

Dakle, B_i je podskup skupa koji se sastoji od svih potformula od φ i njihovih negacija. Taj skup zovemo *zatvorenje* formule φ.

Definicija 5.2.1. *Zatvorenje LTL formule φ je skup closure(φ) kojeg čine sve potformule ψ od φ i njihove negacije $\neg\psi$ (pri čemu je $\neg\neg\psi \equiv \psi$).*

GNBA $\mathcal{G}_\varphi$ konstruiramo tako da skupovi B_i čine njegova stanja. Takva konstrukcija osigurava da je beskonačna riječ $\overline{\sigma} = B_0B_1B_2...$ izvršenje za riječ $\sigma = A_0A_1A_2...$ u $\mathcal{G}_\varphi$. Uvjeti prihvaćanja za $\mathcal{G}_\varphi$ određeni su na sljedeći način:

$$\text{izvršenje } \overline{\sigma} \text{ je prihvatljivo} \iff \sigma \models \varphi.$$

Dakle, značenje logičkih operatora potrebno je opisati u terminima stanja, prijelaza i prihvatljivih skupova GNBA-a $\mathcal{G}_\varphi$. Cijeli postupak konstruiranja GNBA za LTL formulu temeljito je opisan u dokazu teorema 5.2.1 kojeg ovdje nećemo provoditi, no može se pronaći u [1].

Teorem 5.2.1. *Za svaku LTL formulu φ nad skupom AP postoji GNBA $\mathcal{G}_\varphi$ nad alfabetom 2^{AP} takav da vrijedi*

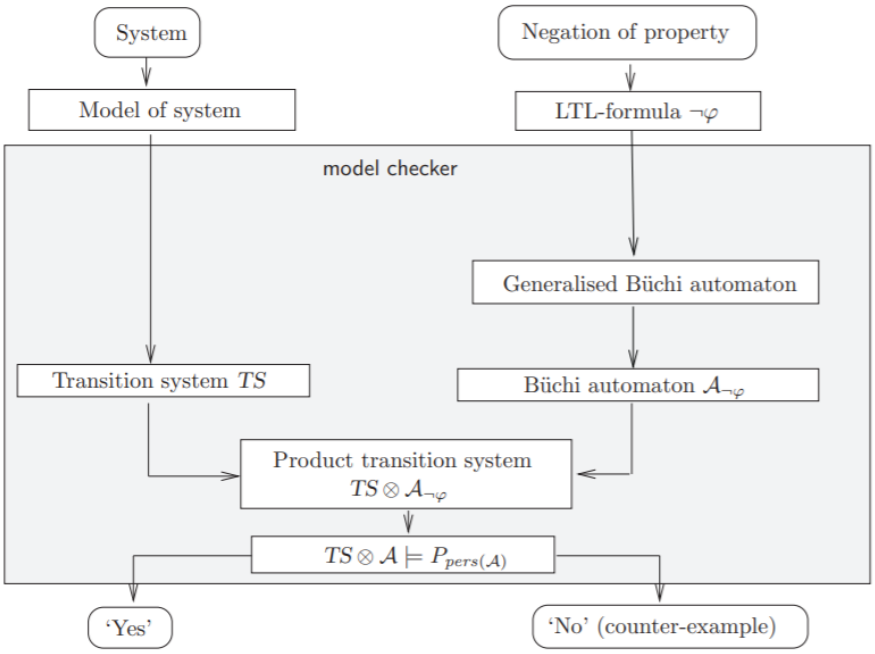
$$Words(\varphi) = \mathcal{L}_\omega(\mathcal{G}_\varphi).$$

Nakon što je za formulu φ konstruiran GNBA $\mathcal{G}_\varphi$, iz $\mathcal{G}_\varphi$ možemo dobiti njemu ekvivalentan NBA, po teoremu 4.2.5. Taj rezutat možemo objediniti u sljedeći teorem.

Teorem 5.2.2. *Za svaku LTL formulu φ nad skupom AP postoji NBA $\mathcal{A}_\varphi$ takav da vrijedi*

$$Words(\varphi) = \mathcal{L}_\omega(\mathcal{A}_\varphi).$$

Čitav postupak LTL provjere modela temeljen na automatima koji je opisan algoritmom 5 može se ilustrirati slikom 5.2.[3]



Slika 5.2: Pregled LTL provjere modela

[3] preuzeto iz [1]

Poglavlje 6

Model checker NuSMV

Praktični dio rada bit će izveden pomoću model checkera NuSMV. NuSMV je softverski alat koji je nastao kao proširenje simboličkog model checkera SMV[1]. Namijenjen je pouzdanoj verifikaciji većih dizajna, kao pomoćni alat za druge alate za verifikaciju, te kao alat za istraživanje formalnih tehnika verifikacije.

Razvoj NuSMV-a započeo je 1998. godine kad su na njemu udruženo počeli raditi ITC-IRST (Institut kulture u Trentu, Italija) i CMU (Sveučilište Carnegie Mellon u Pittsburghu, SAD), a prva verzija izašla je 1999. godine. NuSMV je besplatan i dostupan preko Interneta[2]. Nema grafičko sučelje, pa se interakcija s korisnikom vrši preko tekstualnog sučelja.

NuSMV može vršiti analizu specifikacije svojstava izražene pomoću LTL ili CTL[3]. Za modeliranje, NuSMV koristi svoj interni ulazni jezik. Automati se opisuju tekstualno na način koji je orijentiran opisivanju relacije "mogućeg idućeg stanja" između stanja. Jezik NuSMV-a bit će vidljiv na primjerima izvedenim u sklopu praktičnog dijela rada, koji su opisani u idućim odjeljcima.

U prethodnom poglavlju opisali smo algoritam za LTL provjeru modela; za zadanu formulu φ i tranzicijski sustav TS koji predstavlja model razmatranog sustava provjerava se vrijedi li $TS \models \varphi$ tako da se prvo konstruira NBA $\mathcal{A}_{\neg\varphi}$, a onda se promatra produkt $TS \otimes \mathcal{A}_{\neg\varphi}$ i provjerava postoji li put u tom produktu koji zadovoljava uvjet prihvaćanja za $\mathcal{A}_{\neg}\varphi$. Provjeru egzistencije takvog puta moguće je implementirati u terminima CTL provjere modela. Upravo tom tehnikom služi se NuSMV. U CTL-u, osim temporalnih operatora $\bigcirc$, **U**, **F** i **B**, postoje i kvantifikatori **A** i **E**, koji znače "*svi putovi*", odnosno "*postoji put*", respektivno. Tako, u terminima CTL-a, produkt $TS \otimes \mathcal{A}_{\neg\varphi}$ odgovara sustavu na kojemu NuSMV treba provesti provjeru modela, a formulu koju treba provjeriti možemo

[1] Više o SMV-u i ostalim često korištenim model checkerima može se naći u [2].

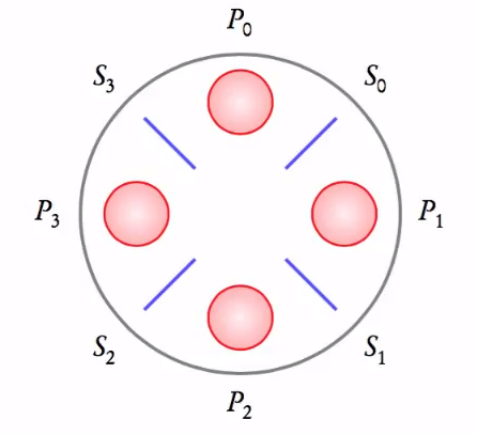
[2] Dostupan je na stranici `http://nusmv.fbk.eu/`.

[3] *Computation Tree Logic*, druga često korištena logika u provjeri modela za formaliziranje svojstava preko stabala izračunavanja (stanja organizirana u razgranatu strukturu).

jednostavno izraziti kao CTL formulu **EG***true*. Prema tome, pitanje na koje treba dati odgovor glasi: postoji li u produktu put? Detaljnije o tome kako je LTL provjera modela implementirana u NuSMV-u može se pročitati u [4] i [3].

6.1 Problem filozofa koji ručaju

Problem filozofa koji ručaju[4] jedan je od najistaknutijih problema na području paralelnih sustava. Problem glasi ovako: četiri filozofa sjede za okruglim stolom, svaki pred sobom ima tanjur s rižom pokraj kojeg se sa svake strane nalazi štapić za jelo. Svaki filozof nalazi se u jednoj od dvije faze: fazi razmišljanja ili fazi jedenja. Da bi mogao početi jesti, filozof mora uzeti dva štapića, jedan s lijeve i jedan s desne strane njegovog tanjura. No između dva susjedna filozofa nalazi se samo po jedan štapić. Dakle, u svakom trenutku samo jedan od bilo koja dva susjedna filozofa može jesti. Potrebno je napraviti raspored tako da niti jedan filozof *ne gladuje*, tj. da svaki od filozofa jede beskonačno mnogo puta. Dakle, svaki filozof treba zauvijek izmjenjivati faze razmišljanja i jedenja, ne znajući pritom kada ikoji od ostalih filozofa želi jesti ili razmišljati. Problem je ilustriran slikom 6.1.[5]



Slika 6.1: Problem filozofa koji ručaju za okruglim stolom

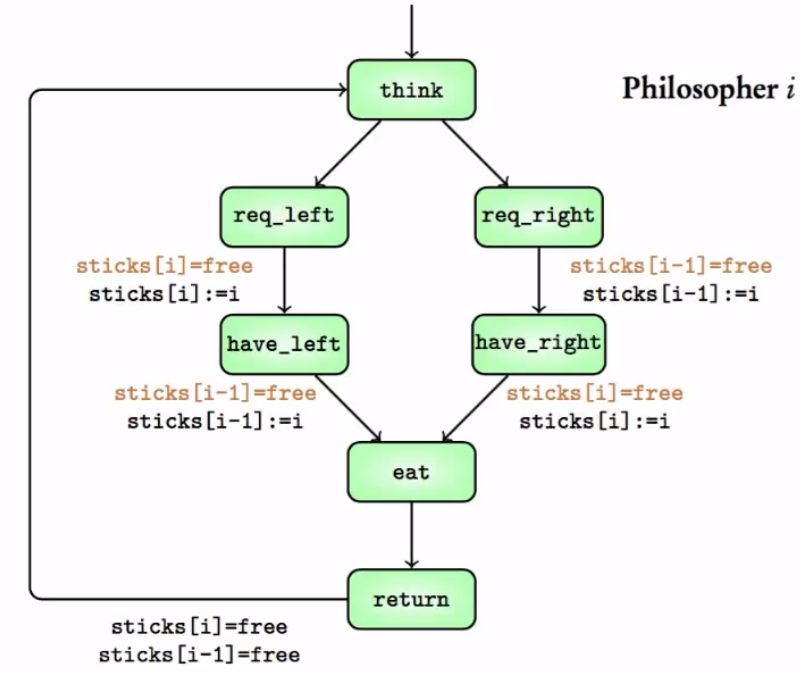
Ovaj problem osmišljen je kako bi prikazao izbjegavanje tzv. *deadlock* scenarija, situacije da sustav dođe u stanje iz kojeg je nemoguće postići ikakav daljnji napredak. Tipičan deadlock scenarij je slučaj da komponente sustava međusobno čekaju jedna na drugu da naprave neki korak. Odsustvo deadlocka je, uz svojstva opisana u poglavlju 3, također jedno od LT svojstava često provjeravanih metodom provjere modela.

[4] eng. *Dining Philosophers problem*, postavio ga je Dijkstra 1965. godine.

[5] preuzeto s `https://www.youtube.com/channel/UCUXDMaaobC01He1HBiFZnPQ/featured`

Na primjeru filozofa, deadlock scenarij može se pojaviti kad svaki od filozofa posjeduje samo po jedan štapić. Željeni rezultat je redoslijed za filozofe tako da čitav sustav nikada ne dođe u situaciju deadlocka – uvijek barem jedan filozof može jesti i razmišljati beskonačno mnogo puta.

Prvo dajemo očito rješenje problema, koje može dovesti do deadlock scenarija. Označimo s P_0, P_0, P_2, P_3 filozofe, a sa S_0, S_1, S_2, S_3 štapiće za jelo. Filozof P_i može jesti ako su mu dostupni štapići $S_{(i-1)mod4}$ i S_{imod4}. Model za filozofa P_i prikazan je programskim grafom na slici 6.2.[6] Lijevi štapić za i-tog filozofa označen je sa $sticks[i]$, a desni sa $sticks[i-1]$. Ako P_i želi jesti, odlazi u stanje req_left, te ako je lijevi štapić slobodan ($sticks[i] = free$), može uzeti štapić (postaviti $sticks[i] := i$) i prijeći u stanje $have\_left$. Nakon toga, ako je i desni štapić slobodan ($sticks[i-1] = free$), može uzeti i desni štapić ($sticks[i-1] := i$) i početi jesti, tj. prijeći u stanje eat. Kad završi s jelom, treba odložiti oba štapića ($sticks[i] := free$ i $sticks[i-1] := free$) te se vratiti u stanje $think$. Analogno ako prvo zatraži desni štapić.



Slika 6.2: Programski graf za filozofa P_i

[6] preuzeto s `https://www.youtube.com/channel/UCUXDMaaobC01He1HBiFZnPQ/featured`

Ovako modeliran sustav dolazi u deadlock situaciju npr. ako svi filozofi ”istovremeno” uzmu lijevi štapić, što će biti demonstrirano NuSMV programom. Dakle, iz početnog stanja $\langle think, think, think, think \rangle$ gdje svi filozofi razmišljaju, sustav dolazi u stanje $\langle have\_left, have\_left, have\_left, have\_left \rangle$, gdje je svaki filozof uzeo lijevi štapić, no ne može započeti s jelom jer zauvijek čeka na desni štapić.

U jeziku NuSMV-a ovaj problem možemo opisati na sljedeći način:

```
MODULE philosopher(i, left, right)

VAR
   location: {think, req_right, req_left, have_right, have_left, eat, return};

ASSIGN
   init(location) := think;
   next(location) := case
                        location=think : {req_left, req_right};
                        location=req_left & left=free : have_left;
                        location=have_left & right=free : eat;
                        location=req_right & right=free : have_right;
                        location=have_right & left=free : eat;
                        location=eat : {eat, return};
                        location=return : think;
                        TRUE: location;
                    esac;
   next(left) := case
                    location=req_left & left=free : i;
                    location=return : free;
                    location=have_right & left=free : i;
                    TRUE : left;
                esac;
   next(right) := case
                    location=req_right & right=free : i;
                    location=return : free;
                    location=have_left & right=free : i;
                    TRUE : right;
                esac;

MODULE main

VAR
   sticks: array 0 .. 3 of {free, 0, 1, 2, 3};
   phil0: process philosopher(0, sticks[0], sticks[3]);
   phil1: process philosopher(1, sticks[1], sticks[0]);
   phil2: process philosopher(2, sticks[2], sticks[1]);
   phil3: process philosopher(3, sticks[3], sticks[2]);

ASSIGN
   init(sticks[0]) := free;
   init(sticks[1]) := free;
   init(sticks[2]) := free;
   init(sticks[3]) := free;
```

Pokretanje i odabir početnog stanja, te naredba za početak simulacije rada opisanog sus-

tava prikazani su idućom slikom:

```
λ NuSMV -int
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

NuSMV > read_model -i C:\Users\eleskla\Desktop\NuSMV\Philosophers_deadlock.smv
NuSMV > go
WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > pick_state -i

**************  AVAILABLE STATES  *************

  ================= State =================
  0) -------------------------
  sticks[0] = free
  sticks[1] = free
  sticks[2] = free
  sticks[3] = free
  phil0.location = think
  phil1.location = think
  phil2.location = think
  phil3.location = think


There's only one available state. Press Return to Proceed.

Chosen state is: 0
NuSMV > simulate -i -k 15
********  Simulation Starting From State 1.1   ********
```

Odabirom stanja na način da filozofi `phil3,phil2,phil1,phil0` redom odlaze u stanje `req_left`, a zatim istim redom svi odlaze u stanje `have_left` dolazimo u deadlock

situaciju – koje god stanje iduće odaberemo, sustav zauvijek ostaje u stanju gdje je

`phil0.location=phil1.location=phil2.location=phil3.location=have_left`, `sticks[0]=0, sticks[1]=1, sticks[2]=2, sticks[3]=3.`

```
********** AVAILABLE STATES ******

  ================ State ==============
  sticks[0] = 0
  sticks[1] = 1
  sticks[2] = 2
  sticks[3] = 3
  phil0.location = have_left
  phil1.location = have_left
  phil2.location = have_left
  phil3.location = have_left

    This state is reachable through:
    0) -------------------------
    _process_selector_ = phil0
    running = FALSE
    phil3.running = FALSE
    phil2.running = FALSE
    phil1.running = FALSE
    phil0.running = TRUE

    1) -------------------------
    _process_selector_ = main
    running = TRUE
    phil0.running = FALSE

    2) -------------------------
    _process_selector_ = phil2
    running = FALSE
    phil2.running = TRUE

    3) -------------------------
    _process_selector_ = phil1
    phil2.running = FALSE
    phil1.running = TRUE

    4) -------------------------
    _process_selector_ = phil3
    phil3.running = TRUE
    phil1.running = FALSE


Choose a state from the above (0-4): 2

Chosen state is: 2
```

```
********** AVAILABLE STATES ******

  ================ State ==============
  sticks[0] = 0
  sticks[1] = 1
  sticks[2] = 2
  sticks[3] = 3
  phil0.location = have_left
  phil1.location = have_left
  phil2.location = have_left
  phil3.location = have_left

    This state is reachable through:
    0) -------------------------
    _process_selector_ = phil0
    running = FALSE
    phil3.running = FALSE
    phil2.running = FALSE
    phil1.running = FALSE
    phil0.running = TRUE

    1) -------------------------
    _process_selector_ = main
    running = TRUE
    phil0.running = FALSE

    2) -------------------------
    _process_selector_ = phil2
    running = FALSE
    phil2.running = TRUE

    3) -------------------------
    _process_selector_ = phil1
    phil2.running = FALSE
    phil1.running = TRUE

    4) -------------------------
    _process_selector_ = phil3
    phil3.running = TRUE
    phil1.running = FALSE


Choose a state from the above (0-4): 4

Chosen state is: 4
```

```
********** AVAILABLE STATES ********

  ================ State ==============
  sticks[0] = 0
  sticks[1] = 1
  sticks[2] = 2
  sticks[3] = 3
  phil0.location = have_left
  phil1.location = have_left
  phil2.location = have_left
  phil3.location = have_left

    This state is reachable through:
    0) -------------------------
    _process_selector_ = phil0
    running = FALSE
    phil3.running = FALSE
    phil2.running = FALSE
    phil1.running = FALSE
    phil0.running = TRUE

    1) -------------------------
    _process_selector_ = main
    running = TRUE
    phil0.running = FALSE

    2) -------------------------
    _process_selector_ = phil2
    running = FALSE
    phil2.running = TRUE

    3) -------------------------
    _process_selector_ = phil1
    phil2.running = FALSE
    phil1.running = TRUE

    4) -------------------------
    _process_selector_ = phil3
    phil3.running = TRUE
    phil1.running = FALSE


Choose a state from the above (0-4):
```

Radi boljeg razumijevanja prethodnog ispisa, napomenimo da NuSMV prikazuje stanja tako da, ako vrijednost pojedine varijable u nekom stanju nije ispisana, ona je ista kao u prethodno ispisanom stanju.

Drugo rješenje za problem filozofa kojim se može izbjeći deadlock izgleda ovako. Imamo iste oznake P_0, P_0, P_2, P_3 za filozofe, te S_0, S_1, S_2, S_3 za štapiće za jelo. i-tom štapiću odgovara i-ti element polja *sticks*, čiji je sadržaj indeks j filozofa P_j koji trenutno

ima pristup štapiću S_i. Na početku, to polje izgleda ovako:

$$sticks: [\ 0 \mid 2 \mid 2 \mid 0\ ],$$

tj. štapić S_0 dostupan je filozofu P_0, S_1 filozofu P_2, S_2 također filozofu P_2, a S_3 opet filozofu P_0. Dakle, na početku rada sustava, P_0 i P_2 mogu jesti budući da oba imaju pristup i svom lijevom i svom desnom štapiću. Nakon što završe s jelom, oni prepuštaju štapiće svojim susjedima tako da se sadržaj polja *stick* promijeni na sljedeći način:

$$sticks: [\ 1 \mid 1 \mid 3 \mid 3\ ].$$

Kad su i oni gotovi s jelom, ponovno prepuštaju štapiće filozofima P_0 i P_2, tj. polje je opet oblika

$$sticks: [\ 0 \mid 2 \mid 2 \mid 0\ ],$$

nakon toga opet

$$sticks: [\ 1 \mid 1 \mid 3 \mid 3\ ],$$

itd. Na ovaj način zauvijek naizmjenično mogu jesti u paru P_0 i P_2, te P_1 i P_3. Time je izbjegnut deadlock scenarij. I ovaj model bit će demonstriran u NuSMV-u.

Postavlja se pitanje "Koje svojstvo treba provjeriti da bi se detektirala prisutnost deadlocka u sustavu?". Drugim riječima, koju LTL formulu treba zadati model checkeru NuSMV da bi on otkrio mogućnost deadlock scenarija? Kad je riječ o problemu filozofa, treba provjeriti je li ispunjeno svojstvo da svaki od filozofa može jesti beskonačno mnogo puta u budućnosti. Izraženo LTL formulom:

$$(\mathbf{GF}(phil0.location = eat)) \ \wedge \ (\mathbf{GF}(phil1.location = eat)) \wedge$$
$$(\mathbf{GF}(phil2.location = eat)) \ \wedge \ (\mathbf{GF}(phil3.location = eat))$$

Pripadni NuSMV kod za drugi model za filozofe:

```
MODULE philosopher(i, left, right)

VAR
   location: {think, req_right, req_left, have_right, have_left, eat, return};

ASSIGN
   init(location) := think;
   next(location) := case
                        location=think : {think, req_left, req_right};
                        location=req_left & left=i : have_left;
                        location=have_left & right=i : eat;
                        location=req_right & right=i : have_right;
                        location=have_right & left=i : eat;
                        location=eat : {eat, return};
                        location=return : think;
                        TRUE: location;
                    esac;
```

```
    next(left) := case
                    location=return & i<3 : i + 1;
                    location=return & i=3 : 0;
                    TRUE : left;
                esac;
    next(right) := case
                    location=return & i>0 : i - 1;
                    location=return & i=0 : 3;
                    TRUE : right;
                esac;

FAIRNESS running
FAIRNESS !(location=eat)
FAIRNESS !(location=think)


MODULE main

VAR
    sticks: array 0 .. 3 of {free, 0, 1, 2, 3};
    phil0: process philosopher(0, sticks[0], sticks[3]);
    phil1: process philosopher(1, sticks[1], sticks[0]);
    phil2: process philosopher(2, sticks[2], sticks[1]);
    phil3: process philosopher(3, sticks[3], sticks[2]);

ASSIGN
    init(sticks[0]) := 0;
    init(sticks[1]) := 2;
    init(sticks[2]) := 2;
    init(sticks[3]) := 0;

LTLSPEC
    (G F (phil0.location=eat)) & (G F (phil1.location=eat))
    & (G F (phil2.location=eat)) & (G F (phil3.location=eat))
```

Naredbom `FAIRNESS running` unutar bloka `MODULE philosopher` ogradili smo se od nepravednog odabira procesa, gdje niti jedan od procesa za filozofe nije pokrenut, nego se uvijek odabire proces `main` i time niti jedan proces za filozofe nije u mogućnosti napredovati. Naredbe `FAIRNESS !(location=eat)` i `FAIRNESS !(location=think)` osiguravaju da niti jedan od filozofa ne ostane zauvijek u stanju `eat`, odnosno u stanju `think` (što bi inače bilo moguće, budući da svaki od njih može nedeterministički odabrati hoće li ostati u stanju `eat` ili prijeći u stanje `return`, ukoliko se nalazi u stanju `eat`, odnosno hoće li ostati u stanju `think` ili prijeći u neko od stanja `req_left`, `req_right`, ukoliko se nalazi u stanju `think`). U bloku `LTLSPEC` nalazi se LTL formula koju ispitujemo.

Pokretanjem NuSMV programa za ovaj model i zadavanjem naredbe `check_ltlspec`, NuSMV automatski provjerava zadovoljava li zadani model sustava zadanu LTL formulu. NuSMV vraća ”true”, što potvrđuje da je sustav modeliran na ovaj način oslobođen deadlock scenarija.

```
NuSMV > read_model -i C:\Users\eleskla\Desktop\NuSMV\Philosophers.smv
NuSMV > go
WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > check_ltlspec
-- specification ((( G ( F phil0.location = eat) &  G ( F phil1.location = eat)) &
  G ( F phil2.location = eat)) &  G ( F phil3.location = eat))  is true
```

6.2 Problem vuka, koze i kupusa

Razmotrimo sada sljedeći problem. Potrebno je prevesti čovjeka (*f - ferryman*), kozu (*g - goat*), vuka (*w - wolf*) i kupus (*c - cabbage*) preko rijeke s lijeve obale na desnu obalu. Prijevoz se vrši čamcem kojim isključivo upravlja čovjek. Osim čovjeka, u čamac stane još najviše jedan putnik – vuk, koza ili kupus. Dakle, čamac može ploviti od jedne obale do druge tako da je u njemu ili samo čovjek ili čovjek i jedan od putnika. Iz očitih razloga, niti u jednom trenutku se na istoj obali bez čovjekovog nadzora ne smiju nalaziti kupus i koza, odnosno koza i vuk. Pitanje je postoji li način da čovjek čamcem preveze sve putnike s lijeve na desnu obalu.

Za čovjeka, kozu, vuka i kupus deklariramo bool varijable $man, goat, wolf$ i $cabbage$. Za svaku od tih varijabli vrijedi: ako joj je pridružena vrijednost $false$, to znači da se jedinka koju ona predstavlja nalazi na početnoj (lijevoj) obali, a inače, da se nalazi na ciljnoj (desnoj) obali. Zatim deklariraom varijablu *carry* koja opisuje tko je, uz čovjeka, trenutni putnik u čamcu. Prema tome, varijabla *carry* može poprimiti neku od vrijednosti iz skupa $\{g, w, c, 0\}$ (poprima vrijednost g kad se prevozi koza, w kad se prevozi vuk, c kad se prevozi kupus, a 0 kad se čovjek vozi sam u čamcu). Da bi čovjek mogao prevesti nekog putnika, oni se moraju nalaziti na istoj obali. Drugim riječima, čovjek može prevesti onog putnika koji ima istu bool vrijednost kao i on. Osim ovog ograničenja, izbor putnika je nedeterministički.

Cilj je pronaći put u tranzicijskom sustavu koji opisuje ponašanje ovog sustava koji zadovoljava sljedeću LTL formulu:

$$\varphi = (\,(\,goat = cabbage \;\vee\; goat = wolf\,) \;\rightarrow\; man = goat\,)\;\mathbf{U}\;(\,man \wedge cabbage \wedge goat \wedge wolf\,).$$

To znači, ako su koza i kupus na istoj obali ili su koza i vuk na istoj obali, onda čovjek mora biti na istoj obali kao i koza, sve dok svih četvero nisu na desnoj obali.

NuSMV provjerava svojstvo φ po svim putovima pripadnog TS-a. Naravno, neće svi putovi zadovoljavati formulu φ, nego je cilj dobiti putove koji ju zadovoljavaju. To se postiže tako da se provjeri formula $\neg\varphi$. Protuprimjer koji NuSMV izgenerira je put koji narušava $\neg\varphi$, a to je upravo onaj put koji zadovoljava φ.

Problem vuka, koze i kupusa u NuSMV jeziku zapisuje se na sljedeći način:

```
MODULE main

VAR
   man: boolean;
   goat: boolean;
   wolf: boolean;
   cabbage: boolean;
   carry: {g, w, c, 0};

ASSIGN
   init(man) := FALSE;
   init(goat) := FALSE;
   init(wolf) := FALSE;
   init(cabbage) := FALSE;
   init(carry) := 0;

   next(man) := {TRUE, FALSE};
   next(carry) := case
                      man=goat : g;
                      TRUE : carry;
                  esac union
                  case
                      man=wolf : w;
                      TRUE : carry;
                  esac union
                  case
                      man=cabbage : c;
                      TRUE : carry;
                  esac union 0;
   next(goat) :=case
                    next(carry)=g : next(man);
                    TRUE : goat;
               esac;
   next(wolf) := case
                    next(carry)=w : next(man);
                    TRUE : wolf;
               esac;
   next(cabbage) := case
                      next(carry)=c : next(man);
                      TRUE : cabbage;
                  esac;

LTLSPEC
   !( (( goat=cabbage | goat=wolf ) -> man=goat) U (man & cabbage & goat & wolf) )
```

Kao što je ranije spomenuto, da bismo dobili rješenje, trebamo ispitati negaciju tražene formule, pa se u skladu s time zadaje formula unutar `LTLSPEC` bloka. Pokretanjem ovako

napisanog koda uz naredbu `check_ltlspec`, NuSMV nam generira protuprimjer koji je rješenje problema:

```
NuSMV > read_model -i C:\Users\eleskla\Desktop\NuSMV\Puzzle.smv
NuSMV > go
NuSMV > check_ltlspec
-- specification !(((goat = cabbage | goat = wolf) -> man = goat)
 U (((man & cabbage) & goat) & wolf))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    man = FALSE
    goat = FALSE
    wolf = FALSE
    cabbage = FALSE
    carry = 0
  -> State: 1.2 <-
    man = TRUE
    goat = TRUE
    carry = g
  -> State: 1.3 <-
    man = FALSE
    carry = 0
  -> State: 1.4 <-
    man = TRUE
    cabbage = TRUE
    carry = c
  -> State: 1.5 <-
    man = FALSE
    goat = FALSE
    carry = g
  -> State: 1.6 <-
    man = TRUE
    wolf = TRUE
    carry = w
  -> State: 1.7 <-
    man = FALSE
    carry = 0
  -> State: 1.8 <-
    man = TRUE
    goat = TRUE
    carry = g
  -> State: 1.9 <-
    man = FALSE
    cabbage = FALSE
    carry = c
  -> State: 1.10 <-
    man = TRUE
    carry = 0
  -> State: 1.11 <-
    man = FALSE
    wolf = FALSE
    carry = w
  -> State: 1.12 <-
    man = TRUE
    carry = 0
  -> State: 1.13 <-
    man = FALSE
    goat = FALSE
    carry = g
  -> State: 1.14 <-
    carry = 0
```

Dakle, rješenje koje je NuSMV našao je sljedeće:

1. na početku su čovjek, koza, vuk i kupus na lijevoj obali;
2. čovjek prevozi kozu na desnu obalu;
3. čovjek se vraća sam na lijevu obalu;
4. čovjek prevozi kupus na desnu obalu;
5. čovjek vraća kozu na lijevu obalu;

6. čovjek prevozi vuka na desnu obalu;
7. čovjek se vraća sam na lijevu obalu;
8. čovjek prevozi kozu na desnu obalu.

Sada su čovjek, koza, vuk i kupus na desnoj obali (`State: 1.8`). Protuprimjer kojeg je generirao NuSMV se nastavlja jer on zapravo prikazuje beskonačan put, no za rješenje problema vuka, koze i kupusa potreban je samo konačan prefiks tog puta.

Problem vuka, koze i kupusa jedan je od problema iz klase *problema planiranja*. To je klasa problema s područja umjetne inteligencije u kojima je za skup stanja potrebno naći put do ciljnog stanja tako da pritom budu zadovoljena određena svojstva. Dakle, LTL provjera modela može se koristiti i za rješavanje problema planiranja.

Bibliografija

[1] Christel Baier i Joost Pieter Katoen, *Principles of model checking*, The MIT press, Cambridge MA, USA, 2008.

[2] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci i Philippe Schnoebelen, *Systems and software verification: model-checking techniques and tools*, Springer Science & Business Media, 2013.

[3] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia i Marco Roveri, *NuSMV: a new symbolic model checker*, International Journal on Software Tools for Technology Transfer **2** (2000), br. 4, 410–425.

[4] Michael Huth i Mark Ryan, *Logic in Computer Science: Modelling and reasoning about systems*, Cambridge university press, 2004.

[5] Robert Manger, *Softversko inženjerstvo*, Element doo, Zagreb, 2016.

[6] Michael Sipser et al., *Introduction to the Theory of Computation*, sv. 2, Thomson Course Technology Boston, 2006.

Sažetak

Ovaj rad obrađuje teoriju o provjeri modela, jednoj od metoda formalne verifikacije softvera. Opisan je način modeliranja promatranog sustava u vidu tranzicijskih sustava, te način specificiranja svojstava pomoću formula linearne temporalne logike. Glavni cilj bio je predstaviti algoritam koji provjerava zadovoljava li tranzicijski sustav kojim je opisan promatrani softver LTL formulu koja opisuje željeno svojstvo. U tu svrhu, prvo je bilo potrebno uvesti pojmove nedeterminističkog Büchijevog automata (NBA), produkta tranzicijskog sustava i NBA, te svojstva perzistencije. Nadalje, opisana su neka važna svojstva koja se često ispituju provjerom modela. Na kraju je na primjerima dva poznata problema demonstriran rad model checkera NuSMV-a.

Summary

This thesis addresses the theory of model checking, one of the formal software verification methods. It describes a way of modeling a software under consideration in terms of transition systems, and a way to specify properties using linear temporal logic formulae. The main goal was to present the algorithm that checks whether the transition system describing the software that needs to be verified satisfies the LTL formula that specifies a desired property. For this purpose, it was necessary to introduce the concepts of the nondeterministic Büchi automata (NBA), product of transition system and NBA, and persistence property. Furthermore, some important properties, that are often checked by model checking method, are described. Finally, on examples of two well known problems, it is demonstrated how model checker NuSMV operates.

Životopis

Rođena sam 8. kolovoza 1994. godine u Slavonskom Brodu. Osnovnu školu i opću gimnaziju završila sam u Županji, gdje sam maturirala 2013. godine. U ak. god. 2013./2014. upisala sam preddiplomski sveučilišni studij matematike na Prirodoslovno-matematičkom fakultetu Sveučilišta u Zagrebu, koji sam kao prvostupnica završila 2017. godine. Na istome sam fakultetu ak. god. 2017./2018. upisala diplomski sveučilišni studij Računarstvo i matematika. Tijekom preddiplomskog i diplomskog studija povremeno sam davala individualne instrukcije iz matematike učenicima osnovne i srednje škole. Od travnja 2019. godine zaposlena sam u Ericsson Nikola Tesla d.d. na odjelu kontinuirane integracije.