UNIVERZITET „DŽEMAL BIJEDIĆ“ MOSTAR

FAKULTET INFORMACIJSKIH TEHNOLOGIJA

SEMINARSKI RAD

Formalna specifikacija i verifikacija modela Pitersenovog algoritma

- formalne metode -

Mentor:

Prof. Dr. Nina Bijedić

Student:

Mursel Musabašić, IB190138

Goražde, 2020

Contents

1. Uvod

U informacijskim tehnologijama, općenito u kompjuterskim naukama, često se susreće termin apstrakcija. Termin apstrakcija, koja potiče od latinske riječi abstractio, znači izdvajanje ili izvlačenje. U slikarstvu umjetnici koriste apstrakciju kako bi nešto prikazali upečatljivije, npr: osobu, životinju, i sl., u odnosu na ostale nacrtane detalje na slici. U filozofiji to je misaoni proces koji uključuje razmišljanje o određenim elementima, a gdje se zanemaruju njihove osobine. U softverskom inženjeringu i kompjuterskim naukama apstrakcija je tehnika kojom se uređuju složenosti jednog ili više kompleksnih sistema.

U objektno-orijentiranom programiranju, programeri koriste mehanizam naslijeđivanja kod kreiranja apstraktne klase kao zajedničkog interfejsa za više objekata koji su implementirani na različite načine, ali u osnovi imaju isto značenje. Jednostavno rečeno proces uklanjanja ne-relevantnih detalja se naziva apstrakcija.

Apstrakcija je veoma važan dio inženjeringa, jer pruža bolje shvatanje kompleksnih sistema. Sisteme je potrebno dobro razumjeti, kako bi bili dobro implementirani. Za razumijevanje i implementaciju sistema u kompjuterskim naukama i softverskom inženjeringu koriste se formalne metode. Formalne metode predstavljaju tehnike koje su bazirane na matematici, a služe za specificiranje, implementaciju i verifikaciju softverskih i hardverskih sistema. Postoje brojne formalne metode koje se koriste za specifikaciju modela, dizajn sistema i softversku verifikaciju.

U sklopu seminarskog rada koristi se TLC model checking koji se oslanja na PlusCal algoritamski jezik i TLA^+ formalno-specifikacijski jezik za modeliranje, dizajn, dokumentovanje i verifikaciju sistema.

TLC model checking će se koristiti kako bi se dokazala osobina Petersenovog algoritma da su dva procesa u sinhronizaciji procesa međusobno isključivi tj. da nije moguća situacija da oba procesa koriste određeni isti resurs u jednom trenutku.

2. Koncepti formalnih metoda

Implementacija softverskog projekta započinje neformalno tj. diskusijom, dok završetak implementacije se smatra formalnim u smislu softverske aplikacije. U konačnici koriste se formalne metode u smislu matematičkih testova da se verifikuje ispravnost rada aplikacije.

Formalne metode se baziraju na slijedećim konceptima:

a. Specifikacija modela
b. Formalno dokazivanje
c. Verifikacija modela (model checking)
d. Apstrakciji

2.1. Formalna specifikacija modela

Formalna specifikacija modela je formalna iz razloga što posjeduje sintaksu, semantiku i vokabular. Obzirom da postoji potreba za formalnom semantikom to znači da jezici specifikacije ne mogu biti prirodni jezici, već oni koji su bazirani na matematici.

Formalnim specifikacijama vršimo opis osobina i ponašanja jednog ili više sistema, a zajedno sa dobro definisanom jezičnom semantikom, formalne specifikacije možemo koristiti kako bi potvrdili i verifikovali da ti sistemi rade uredno.

Postoje slijedeći pristupi i tehnike formalnih specifikacija:

- Tehnika algebarskog pristupa - gdje se na sistem gleda shodno njegovim aktivnostima i vezama između tih aktivnosti.
- Pristup orijentiran na model – gdje se na sistem gleda kao na model stanja („state model“) koji je konstruisan koristeći matematičke cjeline kao što su skupovi i funkcije.

Kao što se koristi TLA$^+$ formalno specifikacijski jezik u ovom seminarskom radu, tako postoje i drugi jezici gdje su neki bazirani na algebarskom pristupu (Lotos, OBJ, Larch) ili pristupu orijentiranom prema modelu (Z, VDM, CSP, B, Event-B).

2.2. Formalno dokazivanje

Formalno dokazivanje se koristi pri validaciji određene osobine sistema. U procesu dokazivanja konstruiše se čitav niz koraka gdje svaki korak sadrži određeni set pravila koja treba da budu ispoštovana. Na Internetu postoje brojni asistenti u obliku aplikacija kojima je moguće uraditi potpuno automatsko testiranje tj. dokazivanje određenih osobina sistema. Neki od tih alata mogu potpuno automatski raditi, dok drugi rade polu-automatski i zahtjevaju određenu korisničku interakciju (npr: zadavanje uslova u obliku lambda izraza). Svi ovi programi koriste već navedene formalno-specifikacijske jezike.

2.3. Verifikacija modela (Model checking)

Verifikacija modela predstavlja jednu od najzastupljenijih formalnih metoda gdje se model stanja opisuje nekim od formalnih jezika. Sa ovom metodom provjeravamo ispravnost modela shodno njegovoj formalnoj specifikaciji.

U osnovi proces formalne specifikacije možemo opisati kao niz koraka gdje bi prvi korak bio izrada formalnog modela sistema (software/hardware, dio software/hardware, i sl.). Nakon toga model je potrebno formalno opisati koristeći formalnu specifikaciju. Formalnu specifikaciju provjeravamo koristeći neku od formalnih metoda, u ovom slučaju model checking.

Potrebno je napomenuti da postoje ograničenja navedene formalne metode u smislu konačnog broja stanja modela i težina kreiranja modela po uzoru na ljudsko biće.

3. TLA$^+$ formalno specifikacijski jezik

Leslie Lamport je američki matematičar, programer, dobitnik Turingove nagrade, tvorac sistema za pripremu dokumenata (LaTeX) i Paxos algoritma, te ujedno i tvorac TLA$^+$ jezika. TLA$^+$ (u nastavku teksta TLA+) je formalno-specifikacijski jezik koji je originalno kreiran za dizajniranje modela sistema i algoritama, a nakon toga da se isti verificiraju da ne postoje određeni bug-ovi (greške). U osnovi TLA+ predstavlja ekvivalent blueprint-u. TLA+ je akronim od „temporal logic of actions“. Baziran je na Zermelo-Fraenkel teoriji skupova i temporalnoj logici (klasična logika sa operatorima kojima možemo izražavati koncept vremena). TLA+ jezik je jezik za modeliranje softvera iznad kôda („above the code“) i hardvera iznad nivoa integralnih kola („above the circuit level“.).

Termini „above the code“ i „above the circuit level“ predstavljaju jedan način pristupa modeliranju u smislu da prvo definišemo blueprint našeg sistema, tj. da kreiramo specifikaciju navedenih modela prije njegove implementacije.

TLA+ omogućava softverskim inženjerima i sistemskim analitičarima da se formalno i praktično obrazloži kôd koji namjeravaju napisati. Pomoću TLA+ jezika i njegovog alter ego algoritamskog jezika PlusCal moguće je opisati samo dio ciljanog sistema, te verificirati da taj dio radi uredno. Pored tih jezika TLA+ posjeduje svoj IDE sa integrisanim alatima kao što je TLC model checker i TLA Proof manager – TLA+ Toolbox.

U odnosu na druge programske jezike, TLA+ se ne kompajlira niti interpretira, već se koristi model checker (TLC) kako bi provjerili svako moguće ponašanje naše specifikacije. Kada je u pitanju dokazivanje teorema onda možemo da koristimo TLAPS dokazni sistem[1] koji dolazi kao zasebna aplikacija koja se u pozadini oslanja na softverske dokazivače teorema kroz koje prolazi redoslijedom, i to prvo koristi SMT (tj. CVC3), pa onda Zenon, te na kraju Isabelle. Po instaliranju TLAPS je moguće pokrenuti direktno iz TLA+ Toolbox-a.

TLA+ formalno-specifikacijski jezik sadrži operatore, izraze, skupove/nizove, logiku, funkcije i rekurzivne funkcije, n-torke (tuples) i strukture (hashes). [2]

PlusCal algoritamski jezik dr. Lamport je kreirao iz razloga zato što je semantički gledano bliži programerima i softverskim inženjerima za razliku od TLA+ formalnog jezika koji se oslanja na matematičke izraze. Većina primjera (osim naučno-istraživačkih radova) na

[1] https://tla.msr-inria.inria.fr/tlaps/content/Home.html (trenutna verzija 1.4.3 od juna 2015. god.)
[2] https://lamport.azurewebsites.net/tla/summary-standalone.pdf (25.1.2020. god.)

Internetu su pisani u PlusCal-u, dok TLA+ Toolbox nudi mogućnost konverzije u formalni jezik. Međutim, bitno je napomenuti da PlusCal izraz može biti bilo koji izraz iz TLA+, što znači da može biti bilo šta izraženo iz teorije skupova i predikatne logike.[3]

4. TLA+ u opisu digitalnih sistema

Izvršenje („operation execution“) na određenom sistemu predstavlja određeni niz koraka. Šta je u osnovi korak?

TLA+ opisuje taj korak kao promjenu stanja, tj. korak je promjena jednog stanja u slijedeće stanje. Imajući to u vidu može se konstatovati da izvršenje predstavlja sekvencu stanja. Moderna nauka modelira sisteme sukladno promjenama koje se dešavaju u vremenskim intervalima, tj. obično se te promjene dešavaju u kontinuitetu. Npr.: svi operativni sistemi su event-based operativni sistemi tj. operativni koji se zasnivaju na određenim događajima koji se dešavaju/izvršavaju u kontinuitetu.

Ako bi posmatrali sekvencu stanja kao na niz ponašanja („behaviors“) tada bi se ta ponašanja mogla opisati pomoću:

- Programskih jezika
- Turingove mašine
- Pomoću automata
- Ili opisnim jezicima za hardver (VHDL ili Verilog)

TLA+ na izvršenja posmatra kao na „state machines“ tj. automate stanja. Svaki automat stanja je struktuiran pomoću njegovih inicijelnih vrijednosti/stanja, te koje stanje slijedi nakon njegovog prethodnog stanja. TLA+ na stanje gleda kao na dodijelu vrijednosti određenoj promjenljivoj (varijabli). Sa tog gledišta automati stanja se opisuju na slijedeći način:

- koje su njegove varijable,
- koje su inicijelne vrijednosti tih varijabli,
- koja je veza između vrijednosti varijabli trenutnog stanja i vrijednosti varijabli stanja koje slijedi.

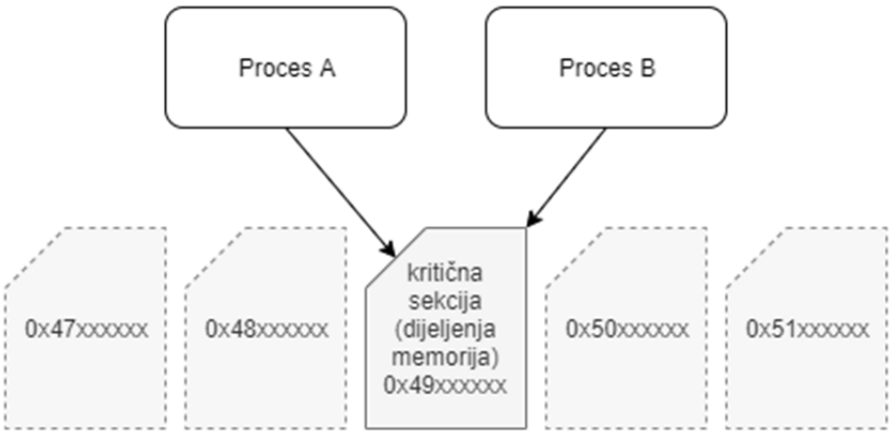
Kako bi bolje upoznali ovaj proces definiranja modela u nastavku seminarskog rada pogledati ćemo kako TLC model checking radi kada je potrebno dokazati određena osobina nekog modela, konkretno u ovom slučaju da Petersenov algoritam posjeduje osobinu međusobnog isključivanja procesa.

[3] https://lamport.azurewebsites.net/tla/c-manual.pdf (strana 75)

5. Sinhronizacija procesa

Prije nego krenemo sa opisom Petersenovog algoritma potrebno je da se ukratko podsjetimo problema sinhronizacije procesa. Kada se govori o procesima i problemu sinhronizacije u informacijskim tehnologijama, pri tome se misli na procese koji se izvršavaju unutar operativnog sistema. Procesi se kategorišu u dvije grupe, i to na nezavisne sa jedne strane i kooperativne procese sa druge. Nezavisni procesi su takvi procesi koji rade nezavisno od drugih procesa i ne dijele memoriju sa drugim procesima.

U osnovi sinhronizacija kooperativnih procesa predstavlja mehanizam kojim se minimizira narušavanje integriteta podataka u određenom dijelu memorije, a kojoj pristupaju dva ili više procesa, i to na način da se postigne uzajamni dogovor u kojem trenutku i kada će određeni proces izvršiti pristup dijeljenom resursu.



Slika 1. Istovremeni pristup memoriji procesa A i B[4]

Na slici 1. prikazan je paralelni pristup jednom dijeljenom resursu, u istom trenutku, gdje postoji mogućnost da jednoj varijabli pristupaju dva kooperativna procesa.

// Izvršenje instrukcije unutar Procesa A

x = 1 + k;

// Izvršenje instrukcije unutar Procesa B

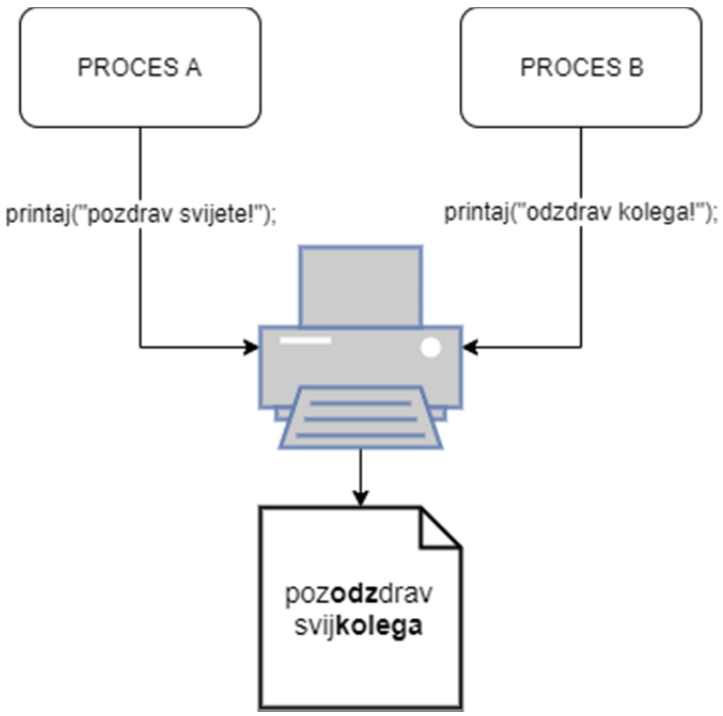
k = k + 1;

[4] Diagram kreiran koristeći online alat – Free Diagram Editor (https://www.diagrameditor.com/)

Kod prikazanog pristupa memoriji, gdje oba procesa mogu u jednom trenutku pristupiti određenoj istoj varijabli, može doći do preplitanja procesa i nedetermisanog rezultata varijable nad kojom se vrše operacije pisanja ili čitanja. Ovaj problem se naziva – **stanje trke**.

Ako pogledamo prethodne dvije linije koda ili bolje rečeno instrukcije možemo uvidjeti da proces A (potrošač) čita vrijednost varijable **k**, dok proces proces B (proizvođač) mijenja vrijednost varijable **k**. Ako ne postoji uzajamni dogovor između ta dva procesa nije moguće razlučiti koji od procesa je prvi pristupio sekciji i koja se operacija zadnja desila.

Dijelu memorije kojoj pristupaju ovi procesi se naziva **kritična sekcija** („critical section“). Kritična sekcija može biti dijeljena varijabla, dijeljenja datoteka, ili neki drugi dijeljeni resur, a koju dijele dva ili više procesa.



Slika 2. Nedostatak mehanizma sinhronizacije procesa – nedeterminisan rezultat

Postoje tri osnovna problema kritične sekcije kod kooperativnih (zavisnih) procesa:

- međusobna isključivost (mutual exclusivity)
- napredak (progress)
- ograničeno čekanje (bounded waiting)

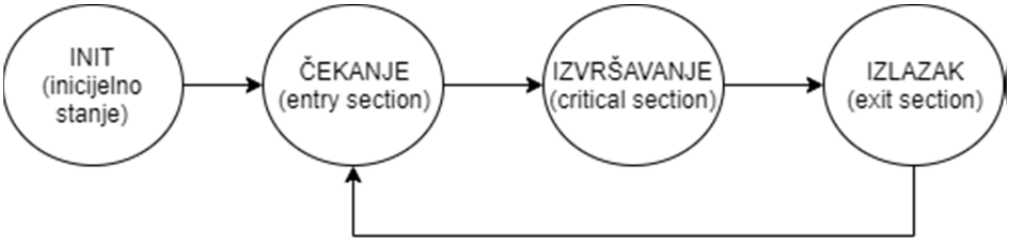
Na slici 3. prikazan je generalni opis interakcije procesa sa kritičnom sekcijom.

```
do
{
    [ULAZNA SEKCIJA]

        [KRITIČNA SEKCIJA]

    [IZLAZNA SEKCIJA]

        [OSTATAK KÔDA]

} while (true);
```

Slika 3. Generalni prikaz interakcije procesa sa sekcijom

Na slici 3. sekcije koje su najzanimljivije jesu ulazna sekcija i izlazna sekcija. Ove sekcije predstavljaju kontrolne sekcije u kojima se izvršavaju nedjeljive (atomic) operacije.

Ulazna sekcija u ovoj strukturi predstavlja sekciju gdje otpočinje inicijacija procesa da želi da pristupi kritičnoj sekciji. Proces daje signal operativnom sistemu da želi da pristupi kritičnoj sekciji, te operativni sistem odlučuje shodno situaciji koja se u tom trenutku odvija može li proces pristupiti kritičnoj sekciji. Pored inicijacije postoji i provjera da li trenutno drugi proces isto tako želi da pristupi kritičnoj sekciji. Na ovom mjestu proces koji pristupa kritičnoj sekciji radi zaključavanje (lock) dijeljenog resursa (varijabla, datoteka i sl.), te time postiže automatski međusobno isključivanje, jer u tom trenutku nije moguće pristupiti navedenom resursu. Izlazna sekcija sadrži instrukciju procesa koji, nakon što je napustio kritičnu sekciju, daje signal operativnom sistemu da napušta kritičnu sekciju, te na taj način je ustupa drugom procesu. Pored toga proces odključava taj resurs na korištenje drugim procesima.



Slika 4. Dijagram stanja modela procesa [5]

[5] Diagram kreiran koristeći online alat – Free Diagram Editor (https://www.diagrameditor.com/)

5.1. Međusobna isključivost (mutual exclusion)

Sa gledišta teorije vjerovatnoće se kaže da događaji E_1, E_2, ... , E_n su međusobno isključivi tako da se kod pojave jednog događaja neće pojaviti drugi događaj u isto vrijeme. To se može formalno iskazati formulom $P(A \cup B) = P(A) + P(B) - P(A \cap B)$, gdje je $P(A \cap B) = 0$. Shodno tome $P(A \cup B) = P(A) + P(B)$, tj. vjerovatnoća događaja koji čine njihovu uniju jeste jednaka sumi njihovih pojedinačnih vjerovatnoća. Drugačije rečeno ili će se dogoditi događaj A ili događaj B. Kao što je ranije navedeno samo jedan proces u jednom trenutku može pristupiti kritičnoj sekciji. U osnovi cilj međusobne isključivosti jeste da izbjegne problem stanja trke procesa koji pokušavaju da istovremeno pristupe određenom djeljenom resursu. U određenim situacijama o iznimne važnosti je da se jednom procesu dozvoli pristup djeljenom resursu („kritičnoj sekciji“), jer od tog procesa zavisi kakav će biti krajnji ishod npr. neke računske operacije. U neka od poznatijih rješenja za problem međusobne isključivosti spadaju: softverski pristup (Petersonov ili Dekkerov algoritam), hardverska podrška (Test & Set operacije), semafori, monitori, i sl.

5.2. Napredak (progress)

U slučaju kada postoji više procesa na čekanju i kada jedan od tih procesa želi da pristupi kritičnoj sekciji onda to mora i da uradi. Izbor da proces pristupi kritičnoj sekciji ne treba odlagati beskonačno. Drugim riječima dijeljeni resur će uvjek biti zauzet i neće se dogoditi zastoj (deadlock). Ovim se postiže da napredak ili progres uvjek bude zagarantovan na nivou cijelog sistema („deadlock freedom“ – oslobađanje od smrtnog zagrljaja“).

5.3. Ograničeno čekanje (bounded waiting)

Ograničeno čekanje definiše vremenski period unutar kojeg se mora omogućiti procesu koji čeka da u jednom trenutku pristupi kritičnoj sekciji. Isto tako procesu nije dozvoljeno da ostane neograničeno dugo u kritičnoj sekciji. Sa ovim se postiže tzv. oslobađanje od izgladnjivanja („starvation freedom“). Međutim do „izgljadnjivanja“ procesa dovodi prethodno navedeni zastoj („deadlock“) gdje bi proces zauvjek ostao u petlji čekanja na pristup kritičnoj sekciji.

6. Pitersenov algoritam u programskom jeziku

Petersenov algoritam je klasičan primjer algoritma za riješavanje problema međusobne isključivosti u sinhronizaciji procesa. Petersenov algoritam je softverski bazirano rješenje za problem kritične sekcije. Poznato je da algoritam u nekim slučajevima ne radi na modernom hardveru. Algoritam je baziran na dva procesa koji međusobno dijele dvije varijable koje su ključne za njegovo pravilno funkcionisanje: **flag[i]** i **turn.** Varijabla flag[i] predstavlja namjeru/želju procesa da pristupi kritičnoj sekciji, dok turn ukazuje čiji je red da pristupi kritičnoj sekciji.

```
int turn;
bool flag[2];

do {

    flag[i] = true;  // i predstavlja naš proces
    turn = j;   // j predstavlja drugi proces

    while(flag[j] && turn == j);    // spin

    // < KRITIČNA SEKCIJA >

    flag[i] = false;

    // ostatak koda

}while(true);
```

Slika 5. Struktura procesa P(i) u Petersenovom algoritmu

Neka proces (kako je navedeno u kôdu gore ***i*** je reprezentacija našeg procesa) ima namjeru da pristupi kritičnoj sekciji, a kako bi to uradio postavlja se vrijednost varijable *flag[i]* na *true*. U slijedećem koraku vrijednost varijable *turn* se postavlja na *j* (***j*** reprezentira drugi zavisni proces). Na ovom koraku pruža se mogućnost drugom procesu (proces **j**), ako čeka na pristup kritičnoj sekciji, da mu se da mogućnost da to i uradi. Ovo je temeljna razlika u odnosu na druge algoritme, gdje se ispoljava korektnost („*fairness*“) procesa *i* da omogući procesu *j* da pristupi kritičnoj sekciji, a kako proces *i*, ne bi pristupio svojoj sekciji po drugi put. Ako proces *j* pristupi svojoj sekciji, onda će proces *i* ostati u svojoj petlji na čekanju na svoj red, te obratno pristupiti će svojoj kritičnoj sekciji, izvršiti operacije koje treba da izvrši, i po izlasku iz kritične sekcije postaviti će vrijednost varijable *flag[i]* na *false*. Ovim će dati signal procesu *j* da može da pristupi kritičnoj sekciji. Nakon toga proces *j* izlazi iz svoje kritične sekcije i postavlja varijablu *flag[j]* na *false*, te time obavještava proces *i* da može pristupiti kritičnoj sekciji. Nakon toga sve se dešava ponovo isto kako je opisano u *do – while(true)* petlji.

Na slijedećem primjeru promatramo dva procesa A i B. Inicijelna vrijednost za niz *flag[2]* je **flag[0] = flag[1] = false**, dok varijabla **turn** nema svoju inicijelnu vrijednost.

```
// PROCES A
do
{
    flag[0] = true;
    turn = 1;

    while(flag[1] && turn == 1); // spin

    // kritična sekcija

    flag[0] = false;    // izlazna sekcija

    // ostatak koda

}while(true);
```

```
// PROCES B
do
{
    flag[1] = true;
    turn = 0;

    while(flag[0] && turn == 0); // spin

    // kritična sekcija

    flag[1] = false;    // izlazna sekcija

    // ostatak koda

}while(true);
```

Slika 6. Pitersenov algoritam za dva procesa

Na slici 6. prikazan je kôd algoritma za dva procesa. Postupak algoritma je slijedeći:

1. Neka proces A prvi izrazi namjeru da pristupi kritičnoj sekciji tako što postavi vrijednost varijable *flag[0]* na *true*.
2. Nakon toga postavlja varijablu *turn* na 1 gdje daje mogućnost drugom procesu, ako čeka, da pristupi kritičnoj sekciji („fairness“).
3. Proces A vrši provjeru uslova *while(flag[1] && turn==1);* petlje gdje je vrijednost *flag[1]=false* (proces B nije još izrazio namjeru da pristupi kritičnoj sekciji), te iako je *turn = 1*, kompletan izraz daje vrijednost *false.*
4. Pošto uslov nije zadovoljen proces A ulazi u kritičnu sekciju.
5. U međuvremenu, proces B biva zakazan („scheduled“), dok proces A biva isključen („preempted“). Na ovom mjestu dolazi do promjene konteksta („context switching“).
6. Proces B postavlja vrijednost *flag[1]* na *true* i daje do znanja da ima namjeru da pristupi kritičnoj sekciji.
7. Postavlja se vrijednost *turn* na *0*, te time, kao što je navedeno u 2. koraku, provjerava da li proces A čeka na ulaz u kritičnu sekciju.
8. Obzirom da obje varijable zadovoljavaju uslov: *flag[0]* je *true* i *turn == 0* izvršenje procesa B će ostati u svojoj *while* petlji.
9. Nakon što proces A izađe iz svoje kritične sekcije postavlja vrijednost zastavice *flag[0]* na false.
10. Odmah nakon toga proces B izlazi iz svoje *while* petlje i ulazi u kritičnu sekciju.
11. Proces B će po izlasku iz kritične sekcije postaviti zastavicu *flag[1]* na *false*, te time će se zaokružiti jedan ciklus algoritma.

Iz ovog postupka možemo zaključiti slijedeće:

- I da se desi da oba procesa postave svoje zastavice na *true*, varijabla *turn* će da odredi na koga je red da pristupi kritičnoj sekciji što zadovoljava osobinu međusobne isključivosti.
- Ako proces A ima namjeru (*flag[0]=true)* da pristupi kritičnoj sekciji, te vrijednost varijable *turn=0*, proces B će ostati u *while* petlji na čekanju, dok će proces A ući u kritičnu sekciju, i obratno. Ovim se zadovoljava osobina progresa tj. napretka procesa.
- Pošto je procesa A ušao u kritičnu sekciju, po izlasku iz sekcije postaviti će vrijednost zastavice na *false (flag[0]=false)*, te time će dozvoliti procesu B, ako je spreman, da pristupi kritičnoj sekciji. Ovim će se spriječiti procesu A da ponovo pristupi kritičnoj sekciji dva puta uzastopno, jer je vrijednost varijable *turn* postavio na 1. Ovim je zadovoljena osobina ograničenog čekanja.

Iako Petersenov algoritam u osnovi donosi rješenje za neke od problema sinhronizacije procesa, ipak ima i svojih nedostataka koji se ogledaju u spin locku (čekanje u petlji) gdje proces koji se izvršava u petlji ne radi ništa korisno i samo bez razloga troši procesorsko vrijeme. Kao rješenje ovog problema nameće se korištenje nekih od API funkcija koja su usko vezane za upravljanje procesima ili nitima (Windows OS: SwitchToThread()[6] ili Linux: sched_yield()[7]). Pored toga dokazano je da algoritam „pati“ od optimizacija koje noviji procesi koriste u svrhu poboljšanja performansi aplikacija, ali koje isto tako dovode do „out-of-order“ izvršavanja operacija. Kako bi se riješio ovaj problem potrebno je da pored korištenja navedenih API funkcija vezanih za upravljanje procesima ili nitima, koriste i funkcije u domeni memorijskih barijera („memory barrier“) ili kako se drugačije nazivaju memorijske ograde („memory fences“). Windows OS aplikacije mogu koristiti MemoryBarrier()[8] funkciju, dok za Linux aplikacije funkcija dolazi u sklopu GCC-a - __sync_synchronize()[9].

[6] https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-switchtothread (Windows Dev Center API reference)
[7] http://man7.org/linux/man-pages/man2/sched_yield.2.html (man7.org)
[8] https://docs.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-memorybarrier (Windows Dev Center)
[9] https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html (GCC 4.1.2 - Atomic builtins)

7. Pitersenov algoritam u formalnom jeziku

Kako je TLA+ formalni jezik baziran na teoriji skupova i matematičkoj logici, autor je kreirao PlusCal algoritamski jezik koji je po svojoj sintaksi bliži programerima i softverskim inženjerima. U nastavku dat je PlusCal kôd Pitersenovog algoritma:

```
---------------------------------------- MODULE peterson ----------------------------------------
Not(i) == IF i=0 THEN 1 ELSE 0
(***************************************************************************
--algorithm PetersonAlgorithm {
 variables flag = [i \in {0,1} |-> FALSE], turn = 0;
 process (ProcName \in {0,1}) {
  waiting: while (TRUE) {
     wantToEnter: flag[self]:=TRUE;
     checkForOtherProcess: turn:=Not(self);
     otherWantsToEnter: if (flag[Not(self)]) { goto otherProcessTurn } else { goto
criticalSection };
     otherProcessTurn: if (turn = Not(self)) { goto otherWantsToEnter } else { goto
criticalSection };
     criticalSection: skip; \* kao da smo pristupili kriticnoj sekciji
     exitingSection: flag[self]:=FALSE;
   }
  };
 }
}
 ***************************************************************************)
==========================================================================
```

Svaka nova specifikacija koja se kreira putem TLA+ Toolbox-a sadrži dvije obavezne linije: Prvu liniju gdje navodite naziv modula („MODULE <naziv modula>“) i zadnju liniju koja završava sa znakovima jednakosti u nizu (===). Naziv modula mora odgovarati nazivu datoteke na disku. Između ove dvije linije dolazi implementacija ili specifikacija modela u PlusCal ili TLA+ jeziku. Implementacija algoritma, kao što je navedeno gore, ne zahtjeva neko objašnjenje. Ono što je bitno napomenuti, a može se primjetiti iz navedenog kôda, su ključne riječi: *--algorithm* i *process*. Kako je PlusCal algoritamski jezik, onda je shodno tome i prilagođen za opis algoritama, te upravo zbog toga sadrži ključne riječi *algorithm* ili *process*. Svaki PlusCal kôd se mora nalaziti unutar bloka komentara koji počinje sa *(**** i završava sa ****)*. Opis algoritma počinje sa *--algorithm <naziv algoritma>*, gdje naziv algoritma može biti proizvoljan i ne zavisi od naziva .tla datoteke.

variables flag = [i \in {0,1} |-> FALSE], turn \in {0,1}; - na ovom mjestu deklarišemo varijable flag i turn. Za flag kažemo da je indeksirani niz gdje je *i* element koji pripada skupu

{0, 1} takav da je flag[i] = FALSE. U ovom slučaju elementima flag[0] i flag[1] se dodijeljuje vrijednost FALSE. Varijabli *turn* je dodjeljena vrijednost 0.

process (ProcName \in {0,1}) { - na ovom mjestu koristimo ključnu riječ *process* kao generalizaciju procesa u algoritmu, a navedeni izraz definiše ProcName kao indeks u skupu elemenata 0 i 1 gdje su elementi navedenog skupa reprezentacija procesa (proces 0 i 1). U TLA-u svaka vrijednost predstavlja skup. Čak 42 ili „abc“. Međutim njegova semantika nije u mogućnosti da kaže koji su njihovi elementi tako da $42 \in \text{“}abc\text{“}$ nije validan izraz i prouzrokovati će grešku. PlusCal koristi tekstualne oznake („labels“) kako bi opisao algoritamske korake. U slučaju da se labele izostave, PlusCal prevodilac će ih dodati u procesu prevoda/konverzije u TLA+ formalni jezik. Potrebno je napomenuti da pri tome rezultat opisanih koraka algoritma može biti dosta manji, te postoji mogućnost da propustimo određene bitne korake, iako će verifikacija broja mogućih stanja teći vrlo brzo. U navedenom kodu možemo primjetiti labele kojima smo dali simbolične nazive kako bi mogli razlikovati sve moguće korake u algoritmu (waiting, wantToEnter, checkForOtherProcess, otherWantsToEnter, otherProcessTurn, criticalSection, exitingSection). Verifikacija algoritma teče određenom kontrolnom putanjom, a ta putanja, kretanje sa jedne linije algo koda na drugu (korak), definisana je labelama. Korak u toj putanji predstavlja kontrolnu putanju koja počinje od početka labele i završava sa labelom. PlusCal algoritamski jezik je po svojoj semantici i sintaksi više bliži imperativnim jezicima kao što je C/C++ te pruža podršku za neke dodatne izraze kao što su *macro, procedure* i *define.* Mnogi izrazi koji se koriste u PlusCal kodu posuđeni su iz TLA formalnog jezika.

Za verifikaciju modela se koristi TLC model checker koji će u procesu verifikacije pronaći sva moguća stanja modela. Ova stanja modela moguće je prikazati vizuelno putem GraphViz alata, a dijagram stanja će se kreirati po završetku verifikacije i biti će zapisan na disk u PDF formatu. Pored toga ne postoji izravan način debagiranja specifikacije, već je to moguće uraditi koristeći *print* metodu iz TLC imenskog prostora (koristiti na vrhu modula izraz EXTEND TLC). Pored *print* metode moguće je zadati i *command-line* parametar kroz TLA+ IDE (*-dump stanja.txt*) u koji će TLC model checker pohraniti sva moguća stanja. Bitno je napomenuti da ova datoteka može sadržati jako puno linija teksta, te se može koristiti ako se zna šta se tačno traži. Naziv datoteke je proizvoljan.

Po završetku implementacije algoritma prvo je potrebno uraditi konverziju iz algoritamskog koda u formalni jezik. Potom se kreira model koji reprezentira formalnu specifikaciju algoritma, te se nad tim modelom vrši verifikacija.

U nastavku je data formalna specifikacija koja je nastala konverzijom iz PlusCal koda u TLA formalni jezik posredstvom korištenja opcije *Translate PlusCal Algorithm* iz *File* menija:

```
\* BEGIN TRANSLATION

VARIABLES flag, turn, pc
vars == << flag, turn, pc >>
ProcSet == ({0,1})

Init == (* Global variables *)
        /\ flag = [i \in {0,1} |-> FALSE]
        /\ turn = 0
        /\ pc = [self \in ProcSet |-> "waiting"]

waiting(self) == /\ pc[self] = "waiting"
                 /\ pc' = [pc EXCEPT ![self] = "wantToEnter"]
                 /\ UNCHANGED << flag, turn >>

wantToEnter(self) == /\ pc[self] = "wantToEnter"
                     /\ flag' = [flag EXCEPT ![self] = TRUE]
                     /\ pc' = [pc EXCEPT ![self] = "checkForOtherProcess"]
                     /\ turn' = turn

checkForOtherProcess(self) == /\ pc[self] = "checkForOtherProcess"
                              /\ turn' = Not(self)
                              /\ pc' = [pc EXCEPT ![self] = "otherWantsToEnter"]
                              /\ flag' = flag

otherWantsToEnter(self) == /\ pc[self] = "otherWantsToEnter"
                           /\ IF flag[Not(self)]
                                 THEN /\ pc' = [pc EXCEPT ![self] = "otherProcessTurn"]
                                 ELSE /\ pc' = [pc EXCEPT ![self] = "criticalSection"]
                           /\ UNCHANGED << flag, turn >>

otherProcessTurn(self) == /\ pc[self] = "otherProcessTurn"
                          /\ IF turn = Not(self)
                                THEN /\ pc' = [pc EXCEPT ![self] = "otherWantsToEnter"]
                                ELSE /\ pc' = [pc EXCEPT ![self] = "criticalSection"]
                          /\ UNCHANGED << flag, turn >>

criticalSection(self) == /\ pc[self] = "criticalSection"
                         /\ TRUE
                         /\ pc' = [pc EXCEPT ![self] = "exitingSection"]
                         /\ UNCHANGED << flag, turn >>

exitingSection(self) == /\ pc[self] = "exitingSection"
                        /\ flag' = [flag EXCEPT ![self] = FALSE]
                        /\ pc' = [pc EXCEPT ![self] = "waiting"]
                        /\ turn' = turn
```

```
ProcName(self) == waiting(self) \/ wantToEnter(self)
                  \/ checkForOtherProcess(self)
                  \/ otherWantsToEnter(self) \/ otherProcessTurn(self)
                  \/ criticalSection(self) \/ exitingSection(self)

Next == (\E self \in {0,1}: ProcName(self))

Spec == Init /\ [][Next]_vars

\* END TRANSLATION
```

Specifikacija napisana u TLA formalnom jeziku se mora nalaziti između dva komentara: BEGIN i END TRANSLATION. Odmah možemo primjetiti ključnu riječ VARIABLES kojom definišemo već poznate varijable koje će se koristiti u opisu specifikacije. Pored varijabli *flag* i *turn*, TLA je automatski kreirao i varijablu *pc*, iako je nismo definisali u PlusCal-u. Varijabla *pc* je kontrolna varijabla kojoj se dodijeljuje vrijednost labele, a koju TLC provjerava kako bi znao koji slijedeći iskaz da koristi za verifikaciju (kontrolna putanja).
vars == << *flag, turn, pc* >> - na ovom mjestu kreiramo *n*-torku (tuple) kako bi grupisali sve definisane varijable na jedno mjesto. Kao što ćemo vidjeti ovo je, tako reći, kratica koja se kasnije koristi kod finalne temporalne formule *Spec*. Prvo na ovom mjestu možemo primjetiti određene operatore kao što je == koji odgovara simbolu ≜ (jednakost po definiciji). TLA+ formal koristi << i >> kao ASCII reprezentaciju simbola uglastih zagrada ⟨ i ⟩ za obilježavanje uređenih parova u *n*-torci.
ProcSet == ({0,1}) – ProcSet predstavlja skup od dva elementa 0 i 1 koji predstavljaju dva procesa u algoritmu.
Init == ... – predstavlja formulu inicijelnog stanja algoritma gdje flag[0] i flag[1] su jednaki FALSE (niti jedan proces nema želju da pristupi kritičnoj sekciji), turn = 0 (neka je red na proces 0 da pristupi kritičnoj sekciji), te *pc* = „waiting“ označava da oba procesa ulaze u petlju ([self \in ProcSet |-> "waiting"]). Varijabla *pc* je ujedno smjernica za TLC model checker koja navodi koje slijedeće stanje je potrebno provjeriti. Svako slijedeće stanje, kao i inicijelno stanje, u TLA-u se posmatra kao formula.
waiting(self) == ... – predstavlja slijedeće stanje tj. korak u algoritmu, ali kako je navedeno posmatra se kao formula. Primjetimo da postoje dvije varijable *pc* i *pc'*. Sve varijable u TLA+ koje nisu „prim“ su varijable trenutnog stanja koje se provjerava, dok „prim“ varijable predstavljaju varijable slijedećeg mogućeg stanja. U konkretnom slučaju dodjeljuje se „waiting“ vrijednost varijablu pc[self] kao znak TLC gdje se trenutno vrši provjera, a varijabli

pc' se dodjeljuje vrijednost „wantToEnter“ koja govori TLC Model Checker-u koje slijedeće stanje ili korak je potrebno verifikovati. Zbog toga i postoji koncept „Next-state“ relacija gdje se uporeduju vrijednosti varijabli trenutnog stanja u odnosu na slijedeće stanje. U nastavku specifikacije Pitersenovog algoritma date su formule koje nose simbolične nazive koraka koji se izvršavaju unutar algoritma. Svi ti koraci tj. formule se svode na jednu „Next“ formulu koja je kulminacija svih prethodnih formula ili koraka koje je potrebno da TLC Model Checker verifikuje. Dokle god „Next“ formula nije istinita, TLC Model Checker će vršiti verifikaciju i potrebnu izmjenu vrijednosti u svrhu pronalaska svih *mogućih* stanja formalne specifikacije. Formalna specifikacija algoritma je iskazana samo kroz jednu formulu – temporalnu formulu *Spec*. Za temporalnu formulu *Spec* kažemo da predikatna formula *Init* (inicijelno stanje) mora biti istinita ***i*** da ***svako*** slijedeće stanje („Next“) je istinito ili da kod varijabla nije bilo promjena. Vrlo bitno je napomenuti da TLA+ ne poznaje tipove podataka, tako da po translaciji iz PlusCal-a u TLA+ potrebno je provjeriti sve dozvoljene rangove vrijednosti za navedene varijable (*„Type-correctness“)*. Ta provjera se radi tako da se definiše invarijanta koju će TLC provjeravati kod verifikacije svakog moguće stanja. Formula invarijante za Pitersenov algoritam bi izgledala ovako:

```
TypeOK == /\ flag \in [{0,1} -> BOOLEAN]
        /\ turn \in {0,1}
        /\ pc \in [{0,1} -> {"waiting", "wantToEnter", "checkForOtherProcess",
"otherWantsToEnter", "otherProcessTurn", "criticalSection", "exitingSection" }]
```

TypeOk je tu da kaže koje vrijednosti se očekuju za prethodno deklarisane varijable.

Centralna osobina algoritama za sinhronizaciju procesa jeste međusobna isključivost, te invarijanta koju ćemo koristiti kao verifikaciju te osobine jeste:

```
MedjusobnaIskljucivostInv == ~(pc[0] = "criticalSection" /\ pc[1] = "criticalSection")
```

Invarijanta je istinita ako izraz sa desne strane nije istinit tj. ako niti jedan od procesa neće u bilo kojem trenutku pristupiti istovremeno krtičnoj sekciji. Pored ove verifikacije moguće je i provjeriti osobinu napretka algoritma i to isto pomoći invarijante:

```
StarvationFreedomInv == \A i \in {0,1} : []<>(pc[i] = "criticalSection")
```

Invarijanta je istinita, ako i samo ako, definisani procesi (0,1) će beskonačno često (□◇) pristupati kritičnoj sekciji, te zbog prisutnosti dualnih modalnih operatora predstavlja isto tako temporalnu formulu.

Zaključak

Kada se sve sagleda vjerovatno će mnogi zaključiti da je TLA težak za učenje ili primjenu, da su matematički izrazi teški za razumjeti, ali ipak se zna da i implementacija pouzdanih sistema je vjerovatno još teža.

Pisanje formalnih specifikacija pomaže u dizajnu sistema. Sisteme je potrebno precizno opisati, jer u tom procesu se obično otkriju brojni problemi koje ponekad namjerno zanemarimo. Ove probleme je lakše ispraviti u fazi dizajna, nego kada se otpočne sa implementacijom.

Formalne specifikacije pružaju jasan i koncizan način izgradnje dizajna. Što je još bitnije pomaže inženjerima da bolje razume sistem koji žele da implementiraju.

Kako je matematika univerzalni jezik, TLA+, kao formalno-specifikacijski jezik, pruža mogućnost kombinovanja matematičkog izražavanja u svrhu pronalska grešaka u dizajnu sistema, te njegovom testiranju.

Literatura

- https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_Synchronization.html (Process Synchronization, 2.1.2020)
- https://www.geeksforgeeks.org/introduction-of-process-synchronization/ (Introduction of Process Synchronization, 2.1.2020)
- https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/ (Peterson's Algorithm in Process Synchronization, 7.1.2020)
- https://cgi.csc.liv.ac.uk/~coopes/comp201/handouts/SE_L11.pdf (Software Engineering COMP 201, Lecture 11 – Formal Specifications, 8.1.2020)
- http://www.ams.org/notices/200811/200811FullIssue.pdf (Notices of the American Mathematical Society, Volume 55, number 11, December 2008, 8.1.2020)
- https://groups.google.com/forum/#!forum/tlaplus (TLA+ Google grupa)
- https://learntla.com/tla/ (Learn TLA+ - Hillel Wayne)
- https://tla.msr-inria.inria.fr/tlaps/content/Home.html(The **TLA**+ **P**roof **S**ystem (TLAPS)
- https://lamport.azurewebsites.net/pubs/euclid.pdf (Euclid Writes an Algorithm: A Fairytale, Leslie Lamport, 6.6.2011)
- https://lamport.azurewebsites.net/pubs/pubs.html (My Writings, Leslie Lamport, 15.10.2019)
- https://lamport.azurewebsites.net/pubs/peterson-theorem.pdf (On a "Theorem" of Peterson, Leslie Lamport, 31.10.1984)
- https://lamport.azurewebsites.net/tla/c-manual.pdf (A PlusCal User's Manual, C-Syntax∗ Version 1.8, Leslie Lamport, 31 August 2018)