UNIVERZITET U BEOGRADU
SAOBRAĆAJNI FAKULTET

Završni rad

Simulacija šalterske službe primenom DEVS formalizma

Mentor:
prof. Vesna Radonjić Đogatović, dipl. inž.

Student:
Dušan B. Tadić
PS110243

Beograd, 2016.

Rezime

U radu je predstavljena implementacija simulacionog modela šalterske službe primenom klasične DEVS specifikacije. Model je razvijen u programskom jeziku C++ korišćenjem novih klasa definisanih C++11 standardom. Jezgro programa čini biblioteka koja sadrži definicije osnovnih DEVS modela kao i njihove prateće simulacione rutine.

Oslanjajući se na prednosti objektno-orijentisanog pristupa biblioteka je proširena tako što je implementiran niz modularnih komponenata koje nasleđuju postojeće osnovne DEVS modele. Nove komponente su predstavljene detaljno.

Realizovani model iskorišćen je za analizu kvaliteta opsluge dela poštanske šalterske službe namenjene za usluge novčanog poslovanja.

U prilogu su dati izvorni kod simulatora i modela kao i rezultati simulacije.

Ključne reči: računarska simulacija, DEVS formalizam, objektno-orijentisano programiranje, šalterska služba

Abstract

This paper presents an implementation of a simulation model of a teller service based on classic DEVS specification. Model was developed in C++ programming language with some of the new classes defined in C++11 standard. The core of the program is a library that defines basic DEVS models and their respective simulation routines.

By relying on the advantages of the object-oriented approach the initial library was extended with a series of additional modular components which inherit the existing DEVS models. New componts are presented in detail.

Developed model was used to analyze the quality of service for a section of the post office dedicated to financial services.

The source code for the simulator, the model itself and simulation results are presented in the appendix.

Keywords: computer simulation, DEVS formalism, object-oriented programming, teller service

Sadržaj

Pregled slika

Pregled algoritama

Pregled tabela

1. UVOD

Realni sistem je izvor za prikupljanje relevantnih podataka o aktivnostima koje se u njemu dešavaju, kao i za definisanje modela na osnovu koga će se vršiti analiza njegovog funkcionisanja [1]. Računarska simulacija omogućava naučnicima i istraživačima da eksperimentišu u "virtualnom" okruženju, uzdižući pritom analizu sistema i procesa na novi nivo, na način kakav u ranijim fazama naučno-istraživačkog rada nije bio moguć. Simulacijom se pronalaze specifična rešenja za probleme (za razliku od opštih rešenja koja se nalaze analitičkim metodama) koristeći uređaje (računare) za kontrolisanje eksperimenta i skraćivanje vremena neophodnog za analizu [2].

Stohastička simulacija razvijala se paralelno sa nastankom i razvojem računara. Neki od početnih pristupa simulaciji i modeliranju ove vrste bili su primarno vezani za specijalizovan hardver i programske jezike, dok su tehnike modeliranja sa čvrstom matematičkom pozadinom novija pojava [2]. Specifikacija sistema zasnovanih na događajima (*Discrete-Event System Specification* – DEVS) je jedna od takvih tehnika. Formalizam se zasniva na konceptima iz teorije sistema i dozvoljava da se predstave svi sisteme čije se ponašanje može prikazati kao sekvenca događaja.

DEVS je osmišljen za modeliranje i simulaciju dinamičkih sistema zasnovanih na događajima i pruža mogućnost specifikacije takvih sistema čija se stanja menjaju usled prijema ulaznog događaja ili usled isteka vremena trajanja stanja. Sa ciljem da se smanji kompleksnost sistema koji se analizira model je organizovan hijerarhijski. Još jedan od načina na koji se utiče na kompleksnost je skrivanjem informacija, kroz pruža modularnog interfejsa za svaki model. Formalna specifikacija kao što je ova nam pruža sredstva za matematičku manipulaciju i omogućava nezavisnost od jezika kojim će model biti implementiran.

U ovom radu razvijen je simulacioni model i izvršena je simulaciona analiza rada šalterske sale za usluge novčanog poslovanja pošte 11030 Beograd 8. Simulator i model implementirani su programskom jeziku C++[6] koristeći neke od novih klasa definisanih standardom C++11. Za razvoj simulatora takođe je iskorišćena i poznata Boost [5] biblioteka čiji elementi neretko postaju deo standarda.

Analiza rada šalterske sale predstavljene u ovom radu prvobitno je odrađena u diplomskom radu [4] u jeziku za simulaciju GPSS (General Purpose Simulation System), preciznije u GPSS/FON varijanti jezika. Korišćenjem statističkih podataka prikupljenih za realizaciju modela u jeziku GPSS u ovom radu razvijen je istovetan model primenom DEVS specifikacije.

Dugo poglavlje ovog rada bavi se predstavljanjem osnovnih pojmova vezanih za simulaciju. Razmatrane su neke od osnovnih vrsta simulacionih modela i definisani koraci od kojih se sastoji proces njihove izrade. Objašnjen je pojam računarske simulacije i posebno simulacije diskretnih događaja.

U trećem poglavlju detaljno je predstavljen DEVS formalizam. Definisani su atomski i spojeni modeli i predstavljeni algoritmi za njihovu simulaciju.

U četvrtom poglavlju opisan je sistem šalterske službe za usluge novčanog poslovanja. Date su raspodele vezane ulazni tok klijenata, izbor usluga i vreme trajanja opsluge. Predstavljene su i karakteristike sistema koje nisu direktno vezane za opslugu klijenata ali utiču na rad sistema.

U četvrtom poglavlju dat je detaljan pregled implementiranih komponenti simulatora. Predstavljene su klase atomskih i spojenih modela kao i klase njihovih simulatora i pomoćna klasa port. Dat je opis svih klasa koje nasleđuju atomski DEVS a koje su iskorićene za izradu modela. Zatim je prikazan i analiziran razvijeni model sistema i dati su rezultati simulacije.

U poslednjem poglavlju data su zaključna razmatranja i predlozi za poboljšanje modela.

2. SIMULACIJA

Simulacija se koristi u različitim oblastima i neke od svrha mogu biti rešavanje problema optimizacije, razvijanje i testiranje bezbednosnih sistema, treniranje ili obuka ljudi i slično. Da bi se neki sistem simulirao potrebno je prethodno razviti njegov model koji definiše osobine tog sistema. Model tako predstavlja sam sistem a simulacija predstavlja imitaciju procesa koji se odvijaju u njemu tokom nekog vremenskog intervala.

Sam razvoj simulacije i njena široka primena usko je povezana sa nastankom i razvojem računara [1]. Moć koju računari danas poseduju omogućava nam da veom složene realne sisteme, predstavljene odgovarajućim modelima, analiziramo za jako kratak vremenski period.

2.1. Modeliranje i model

Modeliranje je osnovni proces ljudskog uma. To je isplativo (u smislu troškova) korišćenje nečega (modela) umesto nečeg drugog (realnog sistema) sa ciljem da se dođe do određenog saznanja. Rezultat modeliranja je model [1].

Model je uprošćena i idealizovana slika realnosti. Model je apstrakcija realnog sistema, zadržava samo one osobine originala koje su bitne za izučavanje. Nivo apstrakije utiče na validnost modela tj. na uspešnost predstavljanja realnog sistema preko modela. Isuviše složeni modeli su skupi i neadekvatni dok isuviše prosti modeli ne oslikavaju posmatrani sistem na pravi način [1].

2.2. Vrste modela

Postoje nekoliko podela modela, među najpoznatijim su sledeće .

1. *Mentalni (misaoni) modeli*. Konstruiše ih ljudski um i na osnovu toga deluje. Omogućavaju komunikaciju među ljudima, planiranje aktivnosti itd.,
2. *Verbalni modeli* su direktna posledica mentalnih modela, njihov izraz u govornom jeziku. Uobičajeno se predstavljaju u pisanom obliku i spadaju u klasu neformalnih modela,
3. *Fizički modeli* predstavljaju umanjene modele realnog sistema. Ponašaju se kao njihovi originali a prave se na osnovu teorije sličnosti ili fizičkih zakona sličnosti,
4. *Matematički modeli* se javljaju ako su veze između objekata opisanematematičkim (numeričkim) relacijama. Polazi se od verbalnog modela koji se transformiše u stanje koje se može opisati matematičkim jezikom. Spadaju u klasu apstraktnih modela a primenjuju se u naučnim i inženjerskim disciplinama.

Različiti fizički modeli mogu imati iste matematičke modele pa se kaže da između njih postoji matematička analogija (analogija u ponašanju). To pruža mogućnost da se neki od fizičkih objekata jednog modela koristi za analizu drugog modela i tada se onda naziva analogni model.

5. *Konceptualni modeli* nastaju na osnovu strukture, logike rada sistema. Zovu se još i strukturni modeli pošto u grafičkom obliku ukazuju na strukturu sistema te su zgodno sredstvo za komunikaciju. Predstavljaju osnovu za izradu računarskih modela,
6. *Računarski (simulacioni) modeli* su prikaz konceptualnih modela u obliku programa za računar korišćenjem programskih jezika i usko su vezani za razvoj računarske nauke [1].

Modeli se često dele na materijalne (hemijska struktura molekula, model aviona) i simboličke (matematički, konceptualni, računarski, simulacioni) [1].

Podela modela prema opisu:

1. *Neformalni opis* modela daje osnovne pojmove o modelu i najčešće nije potpun i precizan. Zbog toga se vrši podela na:
    a) objekte – to su delovi iz kojih se sastoji model,
    b) opisne promenljive – opisuju stanje u kome se objekat nalazi u nekom vremenskom trenutku,
    c) pravila interakcije objekata – definišu se kako objekti modela koji utiču jedni na druge i na opisne promenljive u cilju promena njihovog stanja.

    Neformalni opis je dosta brz i lak te zbog toga može biti nekompletan (ne sadrži sve situacije koje mogu da nastupe), nekonzistentan (predviđanje dva ili više pravila za istu situaciju – kontradiktorne akcije), nejasan (ako nije definisan redosled akcija). Ovakve situacije se prevazilaze pravilima i konvencijama u komuniciranju zvanim formalizmi [1].

2. *Formalni opis* modela treba da obezbedi veću preciznost, potpunost u opisivanju modela. Omogućava i formalizovanje nekompletnosti, nekonzistentnosti i nejasnosti kao i usmeravanje pažnje na karakteristike objekata koje su od najvećeg značaja za istraživanje (apstrakcija) [1].

2.3. Podela i vrste simulacionih modela

Često se modelii dele prema vrsti promenljivih (deterministički i stohastički modeli) i prema načinu na koji se stanje modela menja u vremenu (diskretni i kontinualni) [1].

1. *Deterministički modeli* su modeli čije stanje možemo predvideti, tj. novo stanje je potpuno određeno prethodnim;
2. *Stohastički modeli* su modeli čije se ponašanje ne može unapred predvideti, ali se mogu predvideti verovatnoće promene stanja;
3. *Diskretni modeli* su modeli u kojima se stanje sistema menja u tačno određenim vremenskim trenucima, a te promene nazivamo događajima;
4. *Kontinualni modeli* su modeli u kojima se promenjive stanja menjaju kontinualno u vremenu. Na računarima se ne mogu izvoditi kontinualne promene veličina već se moraju aproksimirati skupom diskretnih vrednosti.

Sa druge strane simulacioni modeli se dele na četiri osnovne vrste [1]:

1. *Monte Karlo*, statička simulacija kod koje se u rešavanju problema koristi stvaranje uzoraka iz raspodela slučajnih promenljivih, a problemi mogu biti i determinističkog i stohastičkog oblika,
2. *Kontinualna simulacija (dinamička)*, se koristi za dinamičke probleme kod kojih se promenjive stanja menjaju kontinualno u vremenu,
3. *Stohastička simulacija*, se bavi modeliranjem sistema koji se mogu predstaviti skupom događaja. Događaj je diskretna promena stanja entiteta sistema. Nastupa u određenom trenutku a te promene stanja entiteta se dešavaju diskontinualno u vremenu, tj. samo u nekim trenucima. Između dva događaja stanje sistema se ne menja;
4. *Mešovita simulacija* (hibridna, kontinualno-diskretna), se uvodi da bi se modelirali oni sistemi koji sadrže procese i događaje. Procesi teku kontinualno dok događaji dovode do diskontinuiteta u ponašanju sistema.

2.4. Verifikacija i validacija modela

Validacija i verifikacija su postupci kojim ispitujemo koliko verno i precizno jedan model predstavlja realni sistem. One se konceptualno razlikuju ali se najčešće simultano sprovode, odnosno kaže se da su u dinamičkoj povratnoj sprezi [1].

Verifikacija se odnosi na proveru da li je simulacioni program (računarski kod) bez grešaka i konzistentan sa modelom (koncepcijom) [1].

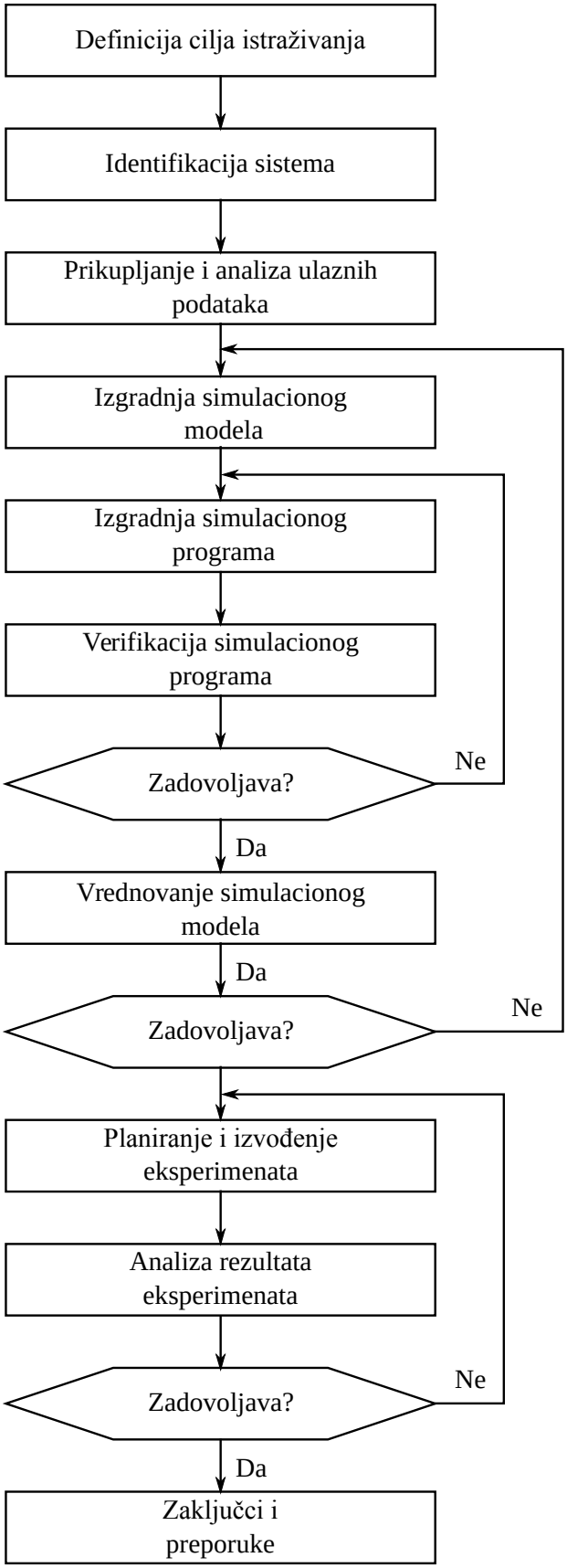
Validacija se odnosi na proveru da li je model precizna reprezentacija realnog sistema. To je interaktivna procedura kojom se poredi ponašanje modela i realnog sistema sve dok se ne dobije tačnost modela koja zadovoljava [1].

2.5. Simulacioni proces

Simulacioni proces je struktura rešavanja stvarnih problema pomoću simulacionog modeliranja. Sastoji se iz više koraka i nije strogo sekvencijalna, moguć je povratak na korake procesa (slika 1.).

Koraci simulacionih procesa [1]:

1. Definicija cilja simulacione studije,
2. Identifikacija sistema (opis komponenata, način rada, formalni prikaz sistema),
3. Prikupljanje podataka o sistemu i njihova analiza,
4. Izgradnja simulacionog modela (stvaranje konceptualnog modela koji adekvatno opisuje sistem),
5. Izgradnja simulacionog programa (izbor programskog jezika i pisanje simulacionog koda),
6. Verifikacija simulacionog programa (da li nam program verno prenosi model),
7. Validacija (vrednovanje) simulacionog modela (da li model adekvatno predstavlja realni sistem),
8. Planiranje simulacionih eksperimenata i njihovo izvođenje,
9. Analiza rezultata eksperimenata (najčešće statistička analiza),
10. Zaključci i preporuke.



Slika 1. Dijagram toka simulacionog procesa

2.6. Simulacija diskretnih događaja

Simulacija diskretnih događaja je metoda simulacionog modeliranja sistema kod kojih se diskretne promene stanja u sistemu ili njegovom okruženju događaju diskontinualno u vremenu. Koristi se uglavnom za analizu dinamičkih sistema sa stohastičkim karakteristikama. Događaj je diskretna promena stanja entiteta sistema i nastupa u određenom trenutku vremena. Sistemi sa diskretnim događajima su: banke, auto servisi, pošte, samoposluge jer se stohastički karakter ogleda u slučajnoj prirodi veličina (događaja) za opis ovakvog sistema. Prema Šanonu, opis diskretnih događaja je sledeći [1]:

-Entiteti, koji se opisuju atributima i uzajamno deluju u aktivnostima, pod određenim uslovima stvaraju događaje koji menjaju stanje sistema.

Kod modela sa diskretnim događajima, pored koncepata koji opisuju strukturu kao što su objekti, relacije između njih i njihovi atributi, uvedeni su i koncepti za opis dinamike: *događaj*, *aktivnost* i *proces* [1].

Događaj predstavlja diskretnu promenu stanja entiteta u sistemu ili njegovom okruženju. Između dva uzastopna događaja stanje sistema se ne menja. Događaj može nastupiti zbog ulaska/izlaska privremenog entiteta u/iz sistema (dolazak/odlazak klijenta) – eksterni; ili zbog promene atributa pojedinih objekata sistema (opslužilac u sistemu postaje slobodan ili zauzet) - interni. Uslovni događaji mogu nastupiti kada je ispunjen uslov njihovog nastupanja. Obično su povezani sa zauzimanjem nekog resursa. Bezuslovni događaji su oni čiji je jedini uslov da tekuće vreme simulacije bude jednako vremenu njegovog nastupanja. Obično su povezani sa oslobađanjem nekog resursa odnosno planiranim završetkom neke aktivnosti [1].

Aktivnost je skup događaja koji menjaju stanje jednog ili više entiteta. Trajanje aktivnosti može biti unapred definisano (determinističko) ili da zavisi od ispunjenja uslova pa je vreme završetka takve aktivnosti nepoznato (stohastičko) [1].

Proces je niz uzastopnih, logički povezanih događaja kroz koje prolazi neki privremeni objekat. To je hronološki uređena sekvenca događaja koja opisuje jednu pojavu od nastajanja do terminiranja. On može da obuhvati deo ili celokupan život privremenog entiteta [1].

2.6.1. Razvoj simulacije diskretnih događaja

Ključni elementi razvoja simulacije diskretnih događaja su [1]:

1) Mehanizam pomaka vremena.
2) Generisanje događaja.
3) Strategije simulacije.

Mehanizam pomaka vremena

U simulaciji diskretnih događaja koriste se dva osnovna mehanizma pomaka vremena [1]:

1. **Pomak vremena za konstantani priraštaj** - Vreme u simulacionom modelu se menja tako da se uvek dodaje konstantan priraštaj Δt. Nakon svakog pomaka vremena, odnosno ažuriranja vrednosti simulacionog sata, ispituje se da li je u prethodnom intervalu vremena trebalo da dođe do nastupanja nekih događaja. Ukoliko jeste, tada se ti događaji planiraju za kraj intervala. Nedostatak je u tome što se kod pomeranja događaja na kraj vremenskog intervala uvodi greška u simulaciji. Događaji koji nisu istovremeni u ovom se pristupu prikazuju kao istovremeni, a potom se određuje redosled njihovog izvođenja (koji se može razlikovati od stvarnog redosleda).

Smanjenjem vremenskog prirasta te se greške smanjuju, ali se povećava vreme koje se troši na izvođenje simulacije kao i porast broja vremenskih intervala u kojima nema događaja.

2. **Pomak vremena na naredni događaj** - Simulacioni sat se pomera na vreme u kom će nastupiti prvi naredni događaj (ili više njih). Simulacija se završava kada nema više događaja ili kada je zadovoljen neki unapred definisan uslov završetka simulacije. Na ovaj način se izbegava greška u vremenu izvođenja događaja a ujedno se preskaču intervali u kojima nema događaja. Ovaj princip je složeniji ali i efikasniji pa svi ključni simulacioni jezici koriste ovaj mehanizam.

Generisanje događaja

Događaj se opisuje sa više atributa, koji formiraju slog događaja. S obzirom na promenljiv broj događaja u vremenu, slogovi događaja se memorišu u listama događaja. Postoje dva pristupa generisanja događaja [1]:

1) Definisanje događaja unapred – Svi događaji su unapred poznati i definisani, a lista događaja sadrži slogove svih događaja,
2) Definisanje narednog događaja – Poznat je jedino prvi naredni događaj, a lista događaja sadrži samo jedan slog, slog poznatog događaja. Pri izvršavanju događaja, planira se i ubacuje u listu njegov naslednik.

Događaje možemo svrstati u dve kategorije u odnosu na mesto nastajanja (generisanja):

1) Eksterni događaji - su oni događaji koji ne zavise od modela i predstvljaju uticaj okoline na sistem (dolazak kupaca u samoposlugu, vozača u auto servis);
2) Interni događaji - zavise od modela i u njemu se generišu (dolazak kupaca u red na kasu).

Strategija izvođenja simulacije

Strategije su [1]:

1. **Raspoređivanje događaja** - podrazumeva da se događaji planiraju unapred i drže u listi budućih događaja, najčešće sortirani po vremenu nastupanja i prioritetu. Procedura planiranja događaja: generiše se slog događaja; dodele se vednosti njegovih atributa (vreme nastajanja, prioritet); događaj se stavlja u listu budućih događaja koja je uređena po vremenu nastupanja događaja i prioritetu.

Funkcionisanje simulatora: sa liste budućih događaja uzima se prvi; ažurira se simulacioni sat na vreme njegovog nastupanja; na osnovu tipa izabranog događaja poziva se odgovarajuća procedura koja izvršava sva ažuriranja u modelu i simulatoru; kada se izvrše svi događaji koji imaju isto vreme nastupanja, simulacioni sat se ažurura na vreme sledećeg događaja iz liste budućih događaja.

2. **Skeniranje aktivnosti** - podrazumeva da se događaji implicitno raspoređuju tako da se promena stanja izvršava preko funkcija koje se nazivaju aktivnosti. Svaka aktivnost ima uslov i akciju. Za svaki vremenski korak Δt, aktivnosti se skaniraju i traži se prva aktivnost koja ima zadovoljen uslov. Tada se izvršava odgovarajući programski segment koji specificira akciju za zadatu aktivnost. Proces skaniranja traje sve dok sve aktivnosti ne budu blokirane. Tada i samo tada se simulacioni sat ažurira na sledeći vremenski korak. Skaniranje aktivnosti je bolje od raspoređivanja događaja kada je broj aktivnosti u modelu mali, a broj događaja u okviru aktivnosti veliki, jer se štedi na računarskim operacijama koje se odnose na listu i njegovo skidanje sa liste događaja.

3. **Interakcija procesa** - predstavlja tehniku simulacije koja je nastala kombinacijom raspoređivanja događaja i skaniranja aktivnosti. Proces možemo posmatrati kao skup isključivih aktivnosti, povezanih tako da terminiranje jedne aktivnosti dozvoljava inicijalizaciju neke druge aktivnosti iz skupa aktivnosti procesa. Glavni problem je sinhronizacija procesa jer je model sistema skup paralelnih procesa od kojih neki mogu biti uzajamno isključivi. Da do ovoga ne bi došlo uvode se dve naredbe WAIT i DELAY i to i u uslovnom i u bezuslovnom kontekstu. WAIT naredba u uslovnom kontekstu ima oblik: WAIT UNTIL <uslov> – ako je uslov zadovoljen, proces koji čeka na taj uslov nastavlja svoj tok, a u suprotnom biće blokiran. Tada se ispituje koji od ostalih procesa može da se aktivira i on se izvršava. DELAY naredbom se odlaže nastavak procesa za određeni broj vremenskih jedinica. U tom slučaju, ispituje se koji od ostalih procesa može da ostvari svoj tok. U bezuslovnom kontekstu naredba DELAY označava se i sa ADVANCE <broj vremenskih jedinica>.

3. DEVS FORMALIZAM

Discrete-Event System Specification (DEVS) formalizam je tehnika matematičkog modeliranja izvedena iz teorije sistema koja omogućava konstruisanje hirerarhijski organizovanih, modularnih modela. Zbog hijerarhijske prirode DEVS-a moguće je spojiti nekoliko jednostavnih modularnih modela u jedan veliki model, što nam omogućava izradu veoma kompleksinh sistema. DEVS teorija pruža i rigoroznu metodologiju za prikazivanje modela i omogućava apstraktan način razmišljanja o sistemima i procesima nezavisan od mehanike same simulacije.

Realni sistemi modelirani primenom DEVS specifikacije mogu da se opišu kao skup atomskih i spojenih komponenti.

Atomski model definiše se kao [2]:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (1)$$

pri čemu je:

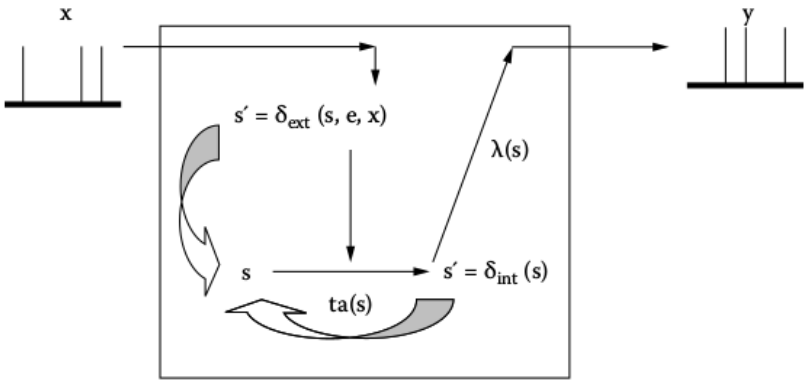
$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$	skup događaja na ulazu koji se sastoji od setova ulaznih portova (p, port) i njihovog sadržaja (v, value),
$Y = \{(p, v) \mid p \in OPorts, v \in Y_p\}$	skup događaja na izlazu koji se sastoji od setova izlaznih portova i njihovog sadržaja,
S	skup sekvencijalnih stanja,
$\delta_{int}: S \to S$	funkcija interne promene stanja,
$\delta_{ext}: Q \times S \to S$	funkcija eksterne promene stanja, ($Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ je celokupni skup stanja, e je vreme proteklo od poslednje promene stanja),
$\lambda: S \longrightarrow Y$	funkcija izlaza,
$ta: R_0^+ \cup \{\infty\}$	vreme trajanja trenutnog stanja atomskog modela.

U svakom trenutku DEVS model se nalazi u nekom stanju $s \in S$. U odsustvu eksternih pojava, model ostaje istom stanju određeni vremenski period koji je definisan funkcijom $ta(s)$. Po isteku vremena $ta(s)$ model prosleđuje vrednost $\lambda(s)$ kroz port $y \in Y$ i prelazi u novo stanje definisano funkcijom $\delta_{int}(s)$.

Promena koja nastaje istekom vremena $ta(s)$ se naziva internom tranzicijom, dok ako do promene stanja dođe usled prijema eksternog događaja pojavu nazivamo eksterna tranzicija. U slučaju eksterne tranzicije novo stanje modela definiše funkcija $\delta_{ext}(s, e, x)$, gde je s trenutno stanje modela, e je proteklo vreme od poslednje tranzicije i $x \in X$ je eksterni događaj koji je izazvao tranziciju.

Funkcija *ta (time advance)* može da vrati bilo koju realnu vrednost u intervalu od 0 do ∞. Stanje za koje $ta(s) = 0$ naziva se tranzitno (nepermanentno) stanje koje prouzrokuje

trenutnu internu promenu stanja. Nasuprot tome stanje za koje je $ta(s) = \infty$ naziva se pasivno stanje u kojem sistem ostaje sve do prijema nekog eksternog događaja.



Slika 2. Prikaz atomskog DEVS modela

Spojeni DEVS model sastoji se od nekoliko atomskih ili kompozitnih "podmodela" i definiše se kao [2]:

$$CM = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, Select \rangle \quad (2)$$

pri čemu su:

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$	skup događaja na ulazu koji se sastoji od setova ulaznih portova (p, port) i njihovog sadržaja (v, value),
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$	skup događaja na izlazu koji se sastoji od setova izlaznih portova i njihovog sadržaja,
D	skup indeksa svih modela,
$\{M_i \mid \forall i \in D\}$	atomski DEVS model,
$\{I_i \mid \forall i \in D \cup \{CM\}\}$	skup indeksa modela na koje utiče model i, $I_i \subseteq D \cup \{CM\}, i \notin I_i$,
$\{Z_{ij} \mid \forall i \in D \cup \{CM\}, \forall j \in I_i\}$	skup funkcija translacije izlaza sa modela i na model j,f
$\{Select: 2^D \rightarrow D\}$	funkcija sinhronizacije (rešava situacije kada se dva ili više događaja realizuju u isto vreme).

Spojeni blokovi grupišu nekoliko DEVS modula u kompozitni model koji se kao takav, usled principa *zatvaranja*, može smatrati novim DEVS modelom. Princip zatvaranja garantuje da povezivanje nekoliko instanca klase kao rezultat daju model iste klase što omogućava stvaranje hijerarhijske strukture.[2]

Specifikacije date u (1) i (2) služe samo za opis modela. Da bi se nad modelom opisanim specifikacijama izvršila simulaciona analiza neophodno je realizovati algoritme za simulaciju atomskog DEVS modela i koordinaciju spojenog DEVS modela kao i algoritam korenog DEVS simulatora kojim se modeli iniciraju. Jedan od načina implementacije pomenutih algoritama dat je u produžetku [3] (algoritmi 1, 2, 3).

Algoritam 1. Algoritam simulacije atomskog DEVS-a

```
DEVS-simulator
variables:
    parent  //parent coordinator
    tl      //time of last event
    tn      //time of next event
    DEVS    //associated model with total state (s, e)
    y       //current output value of the associated model
when receive i-message (i, t) at time t
    tl = t – e
    tn = tl + ta(s)
when receive *-message (*, t) at time t
    if t != tn then
        error: bad synchronization
    y = λ(s)
    send y-message (y, t) to parent coordinator
    s = δint(s)
    tl = t
    tn = tl + ta(s)
when receive x-message (x, t) at time t with input value x
    if not (tl ≤ t ≤ tn) then
        error: bad synchronization
    e = t – tl
    s = δext(s, e ,x)
    tl = t
    tn = tl + ta(s)
end DEVS-simulator
```

Algoritam 2. Glavna simulaciona petlja

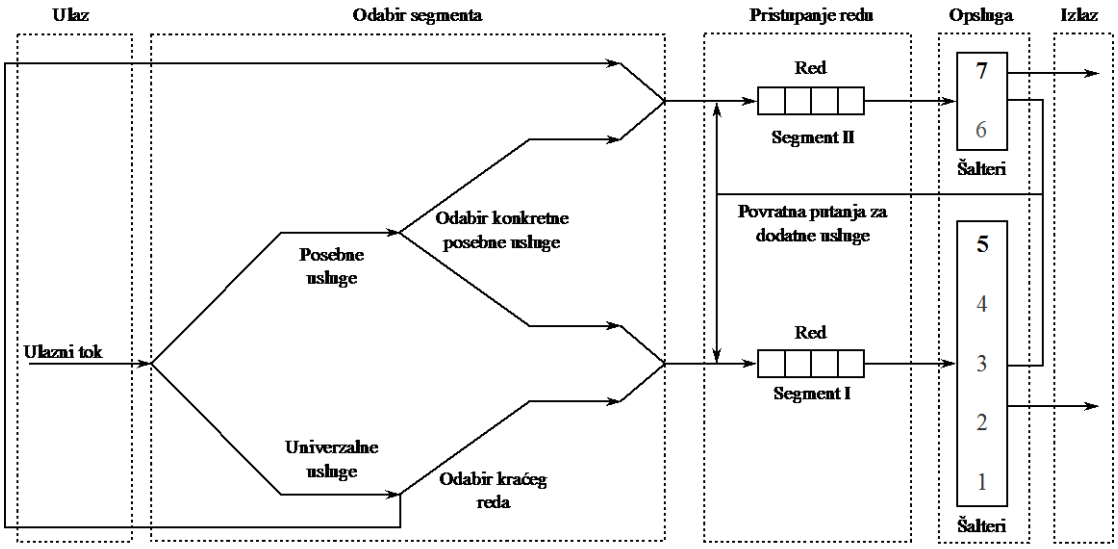
```
DEVS-root-coordinator
variables:
    t
    child
t = t0
send initialization message (i, t) to child
t = tn of its child
loop
    send (*, t) message to child
    t = tn of its child
until end of simulation
end DEVS-root-coordinator
```

Algoritam 3. Algoritam simulacije spojenog DEVS-a

```
DEVS-coordinator
variables:
    DEVN = (X, Y, D, {Md}, {Id}, {Zi,d}, Select) //the associated network
    parent                          //parent coordinator
    tl                              //time of last event
    tn                              //time of next event
    event-list                      //list of elements (d, tnd) sorted by tnd and Select
    d*                              //selected imminent child
when receive i-message (i, t) at time t
    for-each d in D do
        send i-message (i, t) to child d
        sort event-list according to tnd and Select
when receive *-message (*, t) at time t
    if t != tn then
        error: bad synchronization
    d* = first(event-list)
    send *-message (*, t) to d*
    sort event-list according to tnd and Select
    tl = t
    tn = min {tnd | d ϵ D}
when receive x-message (x, t) at time t with external input x
    if not (tl ≤ t ≤ tn) then
        error: bad synchronization
       //consult external input coupling to get children influenced by input
    receivers = {r | r ϵ D, N ϵ Ir, ZN,r(x) ≠ Φ}
    for-each r in receivers
        send x-messages (xr, t) with input value xr = ZN,r(x) to r
        sort event-list according to tnd and Select
        tl = t
        tn = min {tnd | d ϵ D}
when receive y-message (yd, t) with output yd* from d*
    //check external couplng to see if there is and ext. output event
    if d* ϵ IN & Zd*,N(yd*) ≠ Φ then
        send y-message (yN, t) with value yN = Zd*,N(yd*) to parent
    //check internal coupling to get influenced children
    receivers = {r | r ϵ D, d* ϵ Ir, Zd*,r(yd*) ≠ Φ}
    for-each r in receivers
        send x-messages (xr, t) with input value xr = Zd*,r (yd*) to r
end DEVS-coordinator
```

4. ŠALTERSKA SLUŽBA

U sali za novčano poslovanje (slika 3), dolazni tok klijenata se grana na dva toka koja vode u odvojene redove čekanja. Grananje je uslovljeno postojanjem šaltera u dva prostorno odvojena segmenta, kao i postojanjem specijalizovanih šaltera (šalteri 5 i 7, označeni **bold** fontom na slici 3) za pojedine usluge.



Slika 3. Šematski prikaz kretanja klijenata kroz sistem

4.1 Vrste usluga

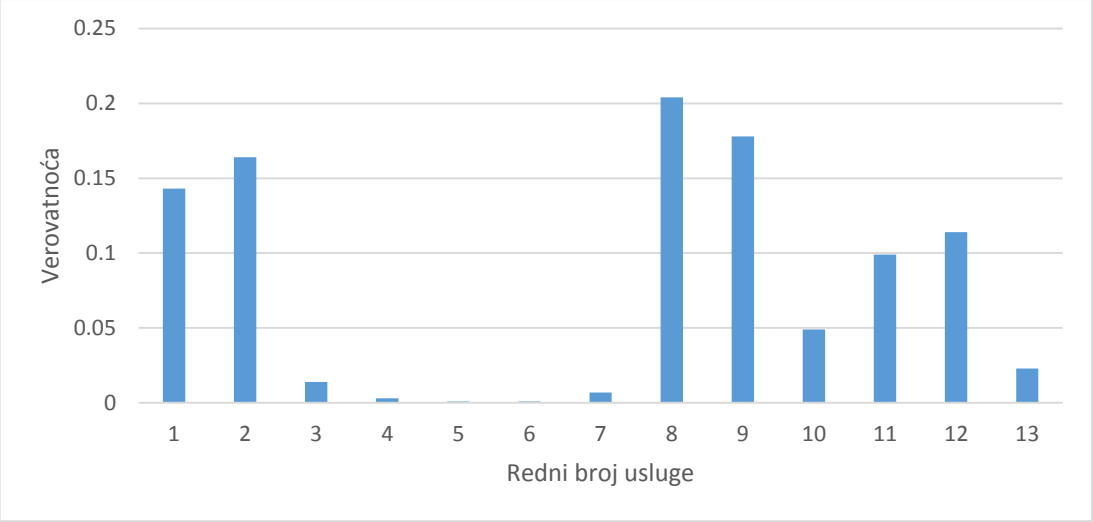
Korisnicima su dostupne sledeće vrste usluga:

1. prodaja vrednosnica,
2. internet i telefonske dopune,
3. pakovanje paketa,
4. usluge vezane za internet naloge i telefonske brojeve,
5. "Diners" usluge,
6. "Western Union" usluge,
7. obavljanje menjačkih poslova,
8. pojedinačne uplate i uputničke usluge,
9. grupna usluga,
10. transakcija sa tekućih računa klijenata (bez provere stanja),
11. transakcije sa tekućih računa klijenata (sa proverom stanja),
12. provera stanja na tekućim računima,
13. štednja

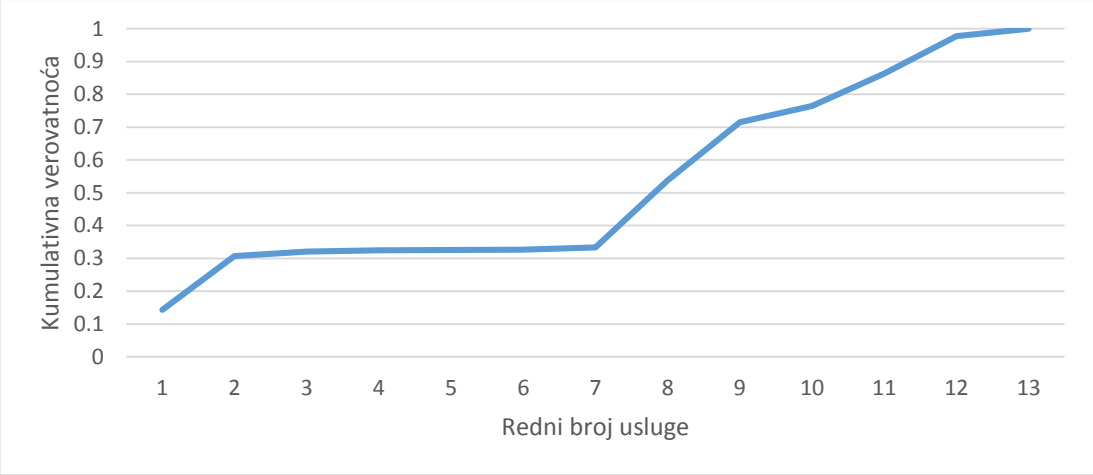
Statistički podaci vezani za usluge prikazani su u tabeli 1. Na slikama 4 i 5 respektivno prikazane su verovatnoća i kumulativna raspodela verovatnoća izbora usluga od strane korisnika.

Tabela 1. Verovatnoća izbora usluge i raspodele vremena opsluge.

Vrsta usluge	Verovatnoća odabira usluge	Kumulativna verovatnoća	Srednje vreme opsluge [s]	Raspodela
1.	0.143	0.143	13.70	Eksponencijalna
2.	0.164	0.307	20.00	Eksponencijalna
3.	0.014	0.321	600.00	Uniformna
4.	0.003	0.324	204.70	Normalna (σ = 32.3)
5.	0.001	0.325	204.70	Normalna (σ = 32.3)
6.	0.001	0.326	204.70	Normalna (σ = 32.3)
7.	0.007	0.333	204.70	Normalna (σ = 32.3)
8.	0.204	0.537	63.10	Erlangova II reda
9.	0.178	0.715	117.40	Erlangova II reda
10.	0.049	0.764	64.60	Erlangova II reda
11.	0.099	0.863	79.30	Erlangova II reda
12.	0.114	0.977	24.00	Eksponencijalna
13.	0.023	1.00	278.60	Erlangova II reda



Slika 4. Verovatnoća izbora usluge



Slika 5. Kumulativna verovatnoća izbora usluge

Usluge sa aspekta pokrivenosti na kanalima opsluge možemo svrtstati u dve grupe:

- usluge dostupne na svim kanalima opsluge (uključujući i specijalizovane) - univerzalne usluge,
- usluge dostupne isključivo na specijalizovanim kanalima opsluge - posebne usluge.

Usluge obeležene u tabeli 1. rednim brojevima od 8 - 13 spadaju u grupu univerzalno dostupnih usluga. Ostale usluge (obeležene rednim brojevima 1 - 7) spadaju u grupu posebnih i njihovo opsluživanje se vrši na nekom od specijalizovanih šaltera od kojih se po jedan nalazi u svakom segmentu.

Specijalizovani šalter prvog segmenta nudi tri posebne usluge, a to su: pružanje usluga korisnicima "Diners" kartica, usluge koje posredstvom pošte nudi kompanija "Western Union" i obavljanje menjačkih poslova. Sve tri navedene specijalizovane usluge su istog prioriteta opsluge ali većeg od usluga iz domena univerzalnih. Na specijalizovani šalteru drugog segmenta nude se preostale četiri posebne usluge: prodaja poštanskih vrednosnica, usluge vezane za internet i telefonske dopune, usluga pakovanja paketa i usluge vezane za internet naloge i telefonske brojeve. Prodaja poštanskih vrednosnica i internet i telefonske dopune imaju veći prioritet u odnosu na preostale dve posebne usluge. Kao i na specijalizovanom šalteru drugog segmenta sve posebne usluge su većeg prioriteta od univerzalnih. Sve usluge iz domena univerzalnih su istog prioritea opsluge na svim šalterima.

Usled takve organizacije rada klijenti se opredeljuju za segment u koji odlaze na osnovu dva faktora:

- odabrane usluge - u slučaju da je prva usluga za koju se klijent opredelio iz domena posebnih,
- dužine reda čekanja u segmentu - ako se klijent odlučio za neku od usluga iz domena univerzalnih.

Pojedini klijenti se opredeljuju i za veći broj usluga i to po sledećem pravilu:

- svi klijenti koji su kupili vrednosnice se zatim opredeljuju za neku drugu uslugu i odlaze u odgovarajući segment prema nekom od gore navedenih pravila;
- polovina klijenata koji su proveravali stanje tekućeg računa se opredeljuje za dodatne usluge. Četrdeset procenata od njih zatim zahteva transakciju sa tekućeg računa, dok preostalih šestdeset zahteva grupnu uslugu;
- svi klijenti koji su izabrali uslugu zamene deviza se opredeljuju za bilo koju drugu uslugu, pri čemu ako se nova usluga pruža na šalteru na kojem su trenutno odmah nastavljaju sa opslugom na istom šalteru. U suprotnom odlaze red čekanja u drugom segmentu.

4.2 Opsluga klijenata

Radno vreme pošte je od sedam do devetnaest časova i organizovano je u dve smene. Vreme između nailazaka klijenata je eksponencijalno raspodeljeno sa srednjim vremenom datim u tabeli 2.

Tabela 2. Vreme između nailazaka klijenata.

Smena	Srednje vreme [s]
prepodnevna	26
popodnevna	29

Na šalterima u toku rada dolazi do prekida opsluge klijenata i to usled:

- odlazaka radnika na pauzu,
- odlazaka do blagajne i
- popisa stanja šaltera.

Pauze se organizuju u trajanju od dvadeset minuta na svim šalterima izuzev specijalizovanog šaltera u drugom segmentu (šaltera 7) gde pauza traje petnaest minuta (usled visoke učestanosti potraživanja posebnih usluga sa tog šaltera).

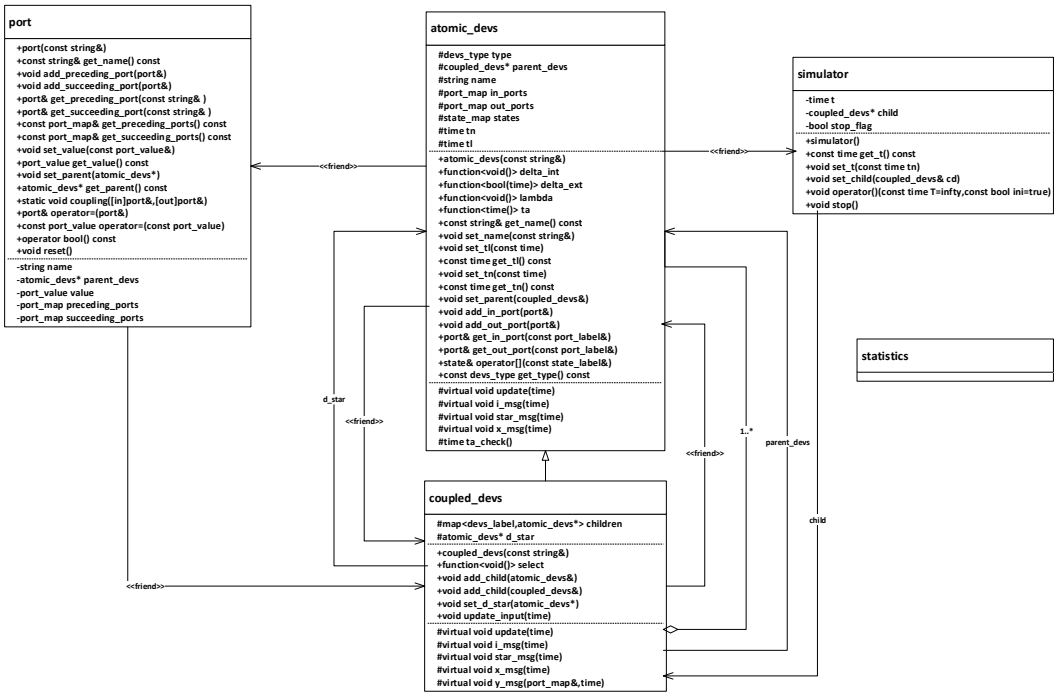
Vreme potrebno za odlazak radnika na blagajnu je 10 minuta na svim šalterima izuzev šaltera broj 7. gde je to vreme 200 sekundi jer su na ovom šalteru uplate ređe od isplata.

Popis stanja šaltera na kraju smene se vrši samo na kraju prve smene na šalterima koji nude samo univerzalne usluge (šalteri 1 - 4 i šalter 6), a na kraju obe smene na specijalizovanim šalterima (šalteri 5 i 7) jer radnici imaju veliki broj zaduživanja različitih vrednosnica, telefonskih kartica i dopuna, opreme za pakovanje i sl. Vreme potrebno da se izvrši popis je 10 minuta.

Specijalizovani šalter prvog segmenta se zatvara deset minuta pred kraj radnog vremena pošte, tako da klijenti koji dođu u sistem posle tog trenutka i opredele se za neku od posebnih usluga sa tog šaltera napuštaju sistem bez opsluge. U sistem se ne primaju novi klijenti pet minuta pred kraj radnog vremena.

5. SIMULACIONI MODEL I REZULTATI

Na osnovu specifikacija datih u poglavlju 3. u programskom jeziku C++ implementirane su klase za atomske modele (*atomic_devs*), spojene modele (*coupled_devs*), klasa koja omogućava povezivanje modela (*port*) kao i koreni simulator (*simulator*) [3]. Klase *atomic_devs* i *coupled_devs* služe za opis i simulaciju formalizama atomskih i spojenih modela. Klasa *simulator* realizuje algoritam korenog simulator, održava vremensku bazu i kontroliše simulaciju inicijalizacijom i izvršavanjem podmodela spojenog modela dodeljnenog objektu ove klase. Klasa *port* implementira način prenosa signala iz jednog u drugi model. Objekat ove klase sadrži vrednost koja se prenosi putem porta, kao i informaciju o portu modela koji je vezan za ovaj port. Na slici 6. prikazan je UML hijerarhija klasa simulatora DEVS specifikacije [3].



Slika 6. UML hijerarhija klasa simulatora DEVS specifikacije.

Veliki deo implementacije *atomic_devs* klase je isti kao i klase *coupled_devs* tako da je za implementaciju klase spojenog modela iskorišćen princip nasleđivanja. Na taj način obezbeđen je polimorfizam podmodela spojenog modela. Svaki objekat *atomic_devs* i *coupled_devs* klase poseduje ime, liste ulaznih i izlaznih portova, stanja, vreme poslednje promene stanja i vreme naredne promene stanja. Klase takođe sadrže metode *delta_int*, *delta_ext*, *lambda* i *ta* koji odgovaraju sledećim elementima strukture atomskog modela $\delta_{int}, \delta_{ext}, \lambda, ta$, respektivno. Ovim članovima neophodno je dodeliti odgovarajue funkcije u zavisnsoti od vrste delatnosti koju obavlja dati model, što će biti detaljnije razrađeno u nastavku. Klasa *coupled_devs* sadrži listu podmodela koji su sadržani u

spojenom modelu, kao i trenutno aktivni podmodel (podmodel koji se trenutno izvršava). Klasa sadrži i metodu s*elect* koja odgovara funkciji *Select* specifikacije (2).

Virtualne funkcije koje se nalaze u klasama *atomic_devs* i *coupled_devs*: *i_msg*, *star_msg*, *x_msg* i *y_msg* su funckije koje proističu iz algoritma za simulaciju atomskog i spojenog modela. Funkcija *i_msg* vrši inicijalizaciju podmodela, *star_msg* vraća izaz roditeljskom modeu (važi samo za atomski model) i vrši unutrašnju promenu stanja podmodela, *x_msg* se izvršava u situaciji pojave spoljnog događaja, dok funckija *y_msg* prosleđuje izlaz iz atomskog modela preko roditeljskog spojenog modela drugom atomskom modelu[3].

Klasa *statistics* služi za prikupljanje statistike simulacionog modela.

5.1 Komponente modela

Atomski modeli (dalje u radu blokovi) koji oponašaju neke specifične procese u sistemu nasleđuju klasu *atomic_devs* radi kompatibilnosti sa simulatorom a metodama *delta_int*, *delta_ext*, *lambda* i *ta* kao i svim pomoćnim metodama i promenljivima dodeljuju se odgovarajuće funkcije u zavisnosti od procesa koji se simulira. Za simulaciju šalterske službe implementirane su sledeće klase: *entity*, *generate*, *assign*, *buffer*, *input_switch*, *select_queue*, *queue*, *server* i *transducer*. Osim navedenih implementirane su i klasa za generisanje pseudoslučajnih brojeva (*devs_supplement::rng*) sa različitim statističkim raspodelama i klasa za štampanje statistika nakon simulacije (*devs_supplement::print_statistics*).

Klasa entity

Klasa *entity* definiše elemente sistema koji se kreću kroz blokove modela (entitete). Entiteti sadrže mapu karakteristika uz pomoć koje je moguće razlikovati dva entiteta i pojedinačno uticati na njihov tok kroz model kao i zadržavanje u određenim delovima sistema. Za implementaciju klase iskorišćena je poznata C++ boost biblioteka, konkretno njena klasa `boost::any` jer karakteristike mogu biti različitog oblika (prirodni ili realni brojevi, *true* ili *false*, slova, reči itd.). Karakteristike se skladište u `std::map<std::string, boost:any>` *STL* kontejneru koji mapira ime karakteristike na njenu vrednost.

Klasa generate

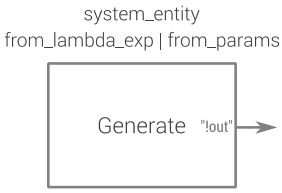
Objekat klase *generate* (slika 7) predstavlja ulaz u sistem i služi za generisanje entiteta. Svaki generisani entitet po nastanku dobija tri karaktersitike, a to su:

- vreme generisanja (“GENERATED_AT”),
- vrednost zastavice za sistemske entitete (“SYS_FLAG”),
- prioritet (“PRIORITY”, 0 osim ako drugačije nije navedeno).

Postojanje sistemske zastavice omogućava blokovima da na osnovu njene vrednosti vrše opslugu na poseban način za svaki tip entiteta ako je to neophodno.

Entiteti se generišu na dva načina a to su:

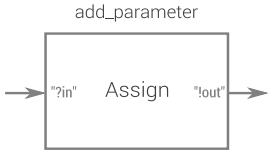
- navođenjem vremena generisanja prvog i svakog narednog entiteta (konstantan interval),
- pomoću *lambda* izraza kojim je moguće definisati kompleksnija pravila.



Slika 7. Grafički prikaz objekta klase generate

Klasa assign

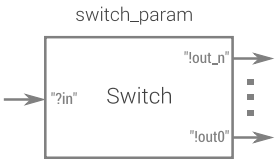
Objekat klase *assign* (slika 8) koristi se za dodeljivanje karakteristika entitetima. Po dolasku u blok entitetima se u mapu karakteristika upisuje predviđeni broj (jedan ili više) parametara čije se vrednosti određuju po nekom pravilu. Pravila se definišu koristeći lambda izraze i mogu da dodele neograničen broj parametara.



Slika 8. Grafički prikaz objekta klase assign

Klasa input_switch

Objekat klase *input_switch* (slika 9) se koristi za usmeravanje entiteta kroz model. Blok ovog tipa poseduje jedan ulazni i nekoliko (2 ili više) izlaznih portova. Nakon inicijalizacije ovog bloka potrebno mu je saopstiti ime parametra na osnovu kojeg će se entitet proslediti na odgovarajući izlazni port (pretpostavka je da je navedeni parametar celobrojna vrednost u intervalu od 0 do [broj izlaznih portova] – 1).



Slika 9. Grafički prikaz objekta klase input_switch

Klase buffer i queue

Objekat klase buffer sadrži po jedan ulazni, izlazni i signalni "*?pull*" port. Koristi se za privremeno čuvanje entiteta u slučajevima kada entiteti iz nekoliko različitih blokova pokušavaju da u isto vreme pređu u naredni blok. Po nailasku entiteta u blok oni se smeštaju u interni FIFO (First In, First Out) red i čekaju trenutak kada mogu da nastave u sledeći blok. Usled toga što postoje blokovi koji mogu da obrađuju samo jedan entitet u datom trenutku, blok ovog tipa čeka na signal od narednog bloka da bi prosledio sledeći entitet iz reda. Blok prima signal preko signalnog porta *"?pull*". Ovaj blok bi postao

redundantan ukoliko bi se svi blokovi koji su funkcionalno sposobni za to (svi blokovi navedeni u radu do ovog trenutka) implementirali sa internim redom umesto samo jednim slotom za opslugu entiteta. Potreba za time se nije ukazala do samog kraja izrade modela pa će ova funkcionalnost morati da bude implementirana retroaktivno.

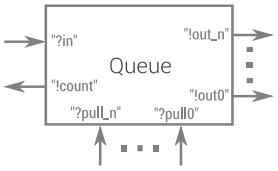
Objekat klase *queue* (slika 10) je implementiran na veoma sličan način kao i blok buffer. Funkcija mu je da privremeno zadrži entitete dok se ne oslobode adekvatni kanali opsluge (serveri) i da sakuplja statističke podatke. Blokovi ovog tipa imaju jedan ulazni, jedan signalni port "*!count*" kojim emituju svoju dužinu i jedan ili više parova izlaznih i signalnih "*?pull*" portova. Entiteti ulaze u blok queue i čekaju na signal da je njihov izabrani kanal opsluge (parametar "DEDICATED_SERVER", pretpostavlja se da je vrednost parametra celobrojna vrednost od 0 do [broj izlaznih portva] - 1) oslobođen, nakon čega se prosleđuju na naredni blok. Usled toga što redovi čekanja u sistemu koji se modelira u ovom radu koriste komplikovanu disciplinu opsluživanja (kombinaciju PRIORITY i FIFO discipline), sa ciljem da se model uprosti, blokovi queue su implementirani tako da prate disciplinu opsluživanja na način na koji je definisana u realnom sistemu. Algoritam izbora narednog entiteta iz reda za opslugu dat je u produžetku (algoritam 4).

Algoritam 4. Algoritam izbora narednog klijenta za opslugu

```
if queue not empty
//System entities are used to temporarily close down the server block
//They have the highest level of priority
    for-each e in queue
        if e.SYS_FLAG & e.DEDICATED_SERVER == free_server
           send e
           done
    tmax_priority = 0
    priority_e
//Check for clients that are waiting for specialized service and find the
//one with the highest priority level
    for-each e in queue
        if e.priority > tmax_priority & e.DEDICATED_SERVER == free_server
            tmax_priority = e.priority
            priority_e = e
    if priority_e exists
        send e
        done
//Finally if none of the above apply send first eligable entity
    for-each e in queue
        if e.DEDICATED_SERVER == -1 //-1 indicates no preference on server
            send e
            done
```

Iz priloženog algoritma se vidi da je funkcija bloka kompozitna i da može biti podeljen na nekoliko blokova. Jedan od načina bi na primer bio kombinacija blokova *input_switch* i uprošćenih varijanti ovog bloka koji bi vršili opslugu po nekoj od osnovnih disciplina.

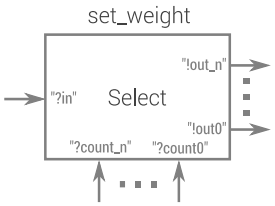
Sistemski entiteti u modelu se koriste da predstave odlazak ranika na pauzu ili do blagajne i vršenje popisa stanja na šalterima. Impementacija na takav način izvršena je usled potrebe da se sinhronizuje rad bloka queue i bloka server o čemu će biti reči dalje u radu. Sistemske entitete blok ne uzima u razmatranje prilikom sakupljanja statističkih podataka.



Slika 10. Grafički prikaz objekata klase queue

Klasa select_queue

Objekat klase *select_queue* (slika 11) koristi se za usmeravanje entiteta na minimalno opterećenu putanju. Sastoji se od jednog ulaznog u dva ili više parova izlaznih i ulaznih signalnih „*?count*“ portova. Određivanje izlazne putanje entiteta odvija se na osnovu vrednosti koje su prijavljene ovom bloku preko „*?count*“ portova i težinskih faktora za svaki izlaz koji se saopštavaju bloku nakon inicijalizacije.



Slika 11. Grafički prikaz objekata select_queue

Klasa server

Objekat klase *server* (slika 12) koristi se za zadržavanje entiteta određeni vremenski period i sakupljanje statističkih podataka o tome. Blok se sastoji od po jednog ulaznog, izaznog i jednog izlaznog signalnog porta “*!pull*”. Vreme zadržavanja entiteta u ovom bloku zavisi od vrednosti parametra “*SERVICE_TIME*”. Po iskteku tog vremena entitet se prosleđuje na izlazni port, a preko signalnog porta se obaveštava povezani blok da je server slobodan. Blok je implementiran sa jednim slotom za opslugu entiteta i bez funkcije “preče” opsluge (opsluga entiteta se vrši neprekidno za vreme njenog trajanja). Sistemski entiteti u modelu zbog toga čekaju pred ovim blokom kao i ne sistemski jer se pretpostavlja da radnik neće ići na pauzu, do blagajne ili vršiti popis dok opslužuje klijenta. Blok server moguće je i ugasiti u predviđenom trenutku tako da više ne obaveštava o svom statusu preko signalnog porta.

Slika 12. Grafički prikaz objekata server

Klasa transducer

Objekat klase *transducer* (slika 13) služi za modeliranje izlaza iz sistema i oslobađanje resursa alociranih za inicijalizaciju entiteta. Blok poseduje samo jedan ulazni port.

Slika 13. Grafički prikaz objekata transducer

5.2 Model šalterske službe

Model koji odgovara sistemu šalterske sale za usluge novačnog poslovanja pošte 11030 u prethodnom poglavlju šematski je prikazan na slici 14.

Klijenti koji nailaze u sistem generišu se u bloku Generate odakle se prosleđuju u blok Assign gde im se dodeljuju karakteristike vezane za opslugu.

Karakteristike koji se upisuju su

- SERVICE_TYPE – vrsta usluge,
- SERVICE_TIME – vreme trajanja opsluge,
- PATH_PICK – izbor putanje do odgovarajućeg segmenta,
- DEDICATED_SERVER – redni broj šaltera,
- PRIORITY – prioritet opsluge i
- 2NDS_0123 – zastavica koja pokazuje da li druga izabrana usluga bila neka sa specijalizovanog šaltera drugog segmenta.

U bloku Switch se na osnovu vrednosti parametra "PATH_PICK" određuje putanja entiteta i to: "0" za enitete koji momentalno napuštaju sistem (entiteti koji su došli u sistem i zahtevaju uslugu sa zatvorengo šaltera ili su došli u sistem pred sam kraj opsluge kada se novi klijenti ne puštaju), "1" za entitete koji su izabrali neku od specijalizovanih usluga drugog segmenta, "2" za entitete koji su izabrali neku od univerzalnih usluga, "3" za entitete koji su izabrali neku od specijalizovanih usluga prvog segmenta. Entiteti koji su izabrali univerzalne usluge šalju se u blok Select odakle se usmeravaju u kraći red.

Po zadatom algoritmu (algoritam 4) entiteti se iz reda prosleđuju na opslugu, nakon čega se filtriraju sistemski entiteti i prosleđuju na izlaz. Ne sistemski entiteti prolaze do bloka Assign gde im se dodeljuje parametar „*GOTO_START*" na osnovu prethodno odabrane usluge usled kojeg se pojedini entiteti vraćaju na ponovnu opslugu.

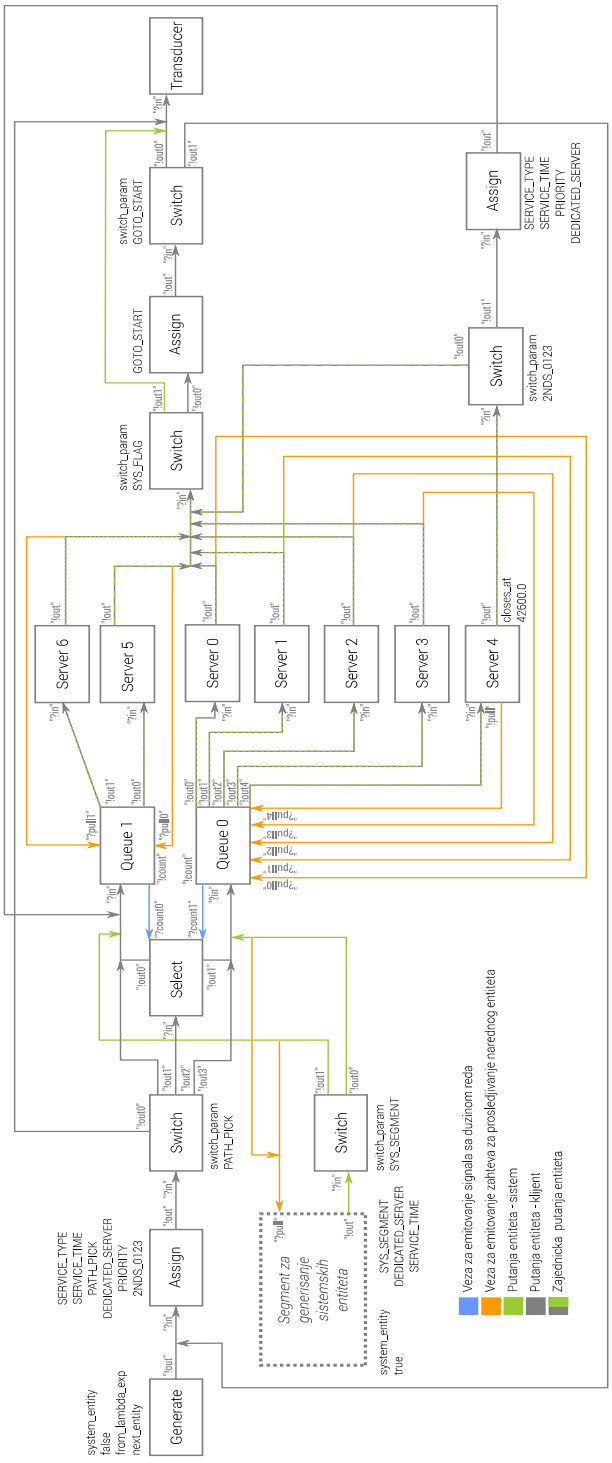
Sa specijalizovanog šaltera prvog segmenta (Server 4 na slici 14) klijenti se na osnovu parametra „*2NDS_0123*" šalju na dodatnu opslugu na specijalizovanom šalteru drugog segmenta ili nastavljaju u naredni blok i izlaz iz sistema.

Segment za generisanje sistemskih entiteta (slika 15) je spojeni model i sastoji se od blokova Generate i Assign i jednog Buffer bloka.

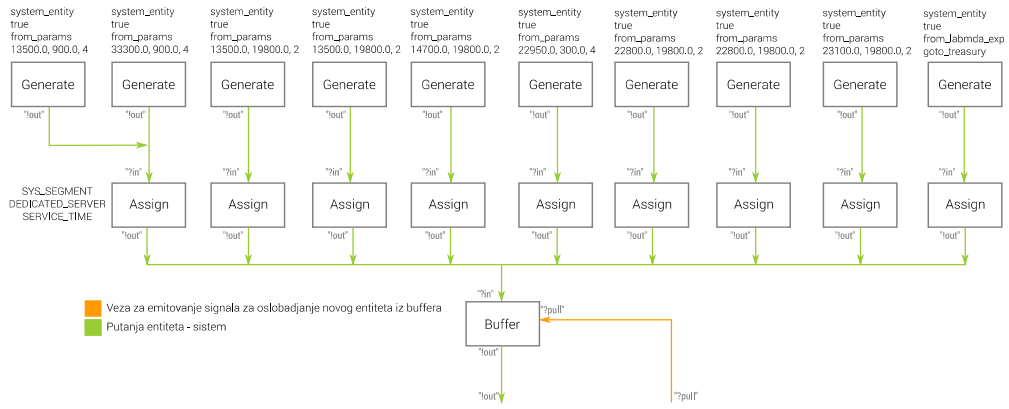
Sistemskim entitetima se u blokovima Assign dodeljuju sledeći parametri:

- SYS_SEGMENT – redni broj segmenta,
- DEDICATED_SERVER – redni broj servera koji se zauzima,
- SERVICE_TIME – vreme opsluge.

Po izlasku iz povezanog bloka entiteti se prosleđuju ka odgovarajućim segmentima i čekaju na opslugu.



Slika 14. Model šalterske sale za usluge novčanog poslovanja pošte 11030



Slika 15. Segment za generisanje sistemskih entiteta

5.3 Rezultat simulacije

Statistike sistema koje su prikupljene su sledeće:

Statistike redova čekanja:

- broj prispelih entiteta (n),
- preostali broj entiteta u redu (Δn),
- maksimalna dužina reda (d_{max}),
- prosečno vreme provedeno u redu ($\bar{t}_r$),
- Srednja dužina reda ($\bar{d}$),
- broj nultih pristupa (n_0)
- procenat nultih pristupa (P_0),
- prosečno vreme provedeno u redu (isključujući nulte pristupe) ($\bar{t}_r(n - n_0)$).

Statistike kanala opsluge:

- broj opsluženih entiteta (n)
- srednje vreme opsluge ($\bar{t}_{ops}$)
- iskorišćenost (η)

Statistički podaci kanala opsluge dobijeni jednom realizacijom simulacionog programa prikazani su u tabeli 3 (podaci za specijalizovane šaltere u oba segmenta su osenčeni).

Tabela 3. Statistike kanala opsluge.

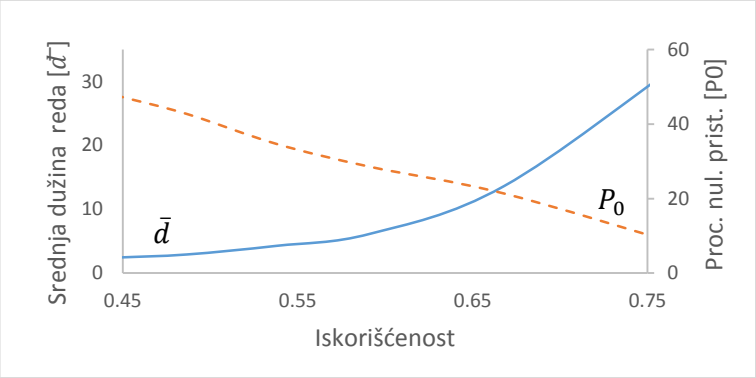
	Prvi segment					Drugi segment	
	Serv. 0	Serv. 1	Serv. 2	Serv. 3	Serv. 4	Serv. 5	Serv. 6
Broj opsluženih klijenata	252	227	182	130	102	130	582
Prosečno vreme opsluge [s]	85.51	81.68	83.99	97.14	110.68	53.98	70.04
Iskorišenost	0.54	0.47	0.39	0.35	0.25	0.30	0.81
	= 0.4					= 0.605	

U tabeli 4 prikazane su statistike redova čeanja u oba segmenta

Tabela 4. Statistike redova čekanja.

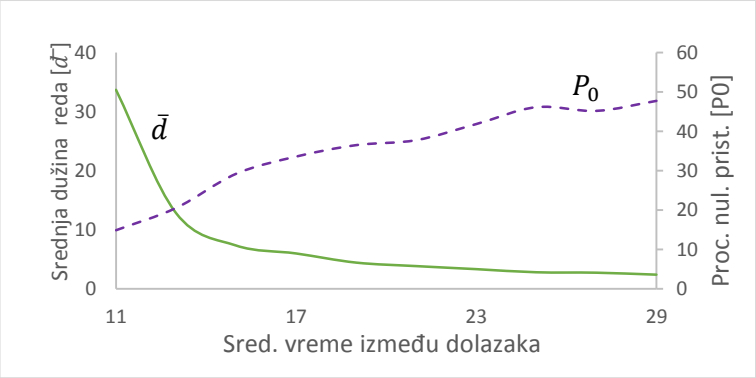
	Red prvog segmenta	Red drugog segmenta
Broj prispelih klijenata	981	815
Preostali broj klijenata	1	2
Maksimalna dužina reda	9	20
Prosečno vreme čekanja [s]	28.46	175.61
Srednja dužina reda	0.692	3.34
Broj nultih pristupa	694	212
Procenat nultih pristupa	70.74	26.01
Prosečno vreme čekanja (bez nultih pristupa) [s]	97.52	237.56

Na slici 16 prikazane su dužina redova čekanja i procenat nultih pristupa u funkciji iskorišćenosti kanala opsluge za obe smene.

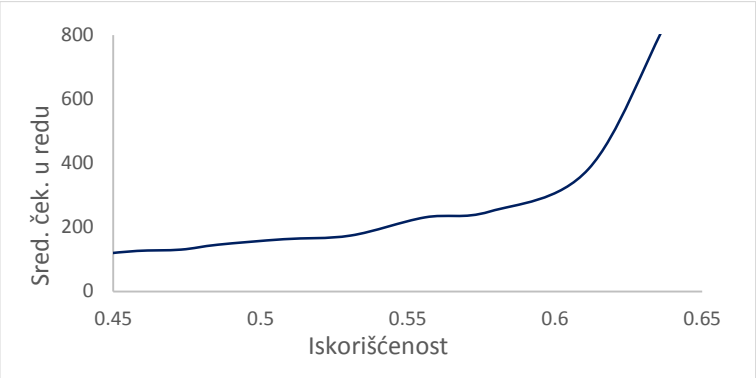


Slika 16. Srednja dužina redova i procenat nultih pristupa u funkciji iskorišćenosti

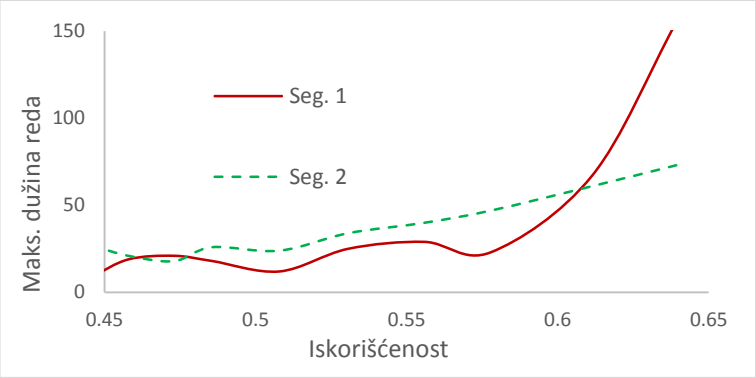
Uveaćanjem intenziteta ualznog toka klijenata za popodnevnu smenu dobijeni su sledeći rezultati: dužina redova i procenat nultih pristupa u funkciji srednjeg vremena između dolazaka (slika 17), srednje vreme čekanja u funkciji iskorišćenosti (slika 18) i maksimalne dužine redova u funkciji iskorišćenosti (slika 19).



Slika 17. Srednja dužina redova i procenat nultih pristupa



Slika 18. Srednje vreme čekanja u funkciji iskorišćenosti



Slika 19. Maksimalne dužine redova u funkciji iskorišćenosti

Da bismo analizirali opslugu u posmatranom sistemu neophodno je prvo definisati kriterijume kvaliteta. Za uslužne sisteme poput ovog u upotrebi su sledeća tri kriterijuma [4]:

1. Opsluga korisnika je kvalitetna ako je verovatnoća čekanja korisnika u redu manja od usvojene vrednosti ε (npr. 0.3, 0.4).
2. Opsluga korisnika je kvalitetna ako je srednja dužina reda čekanja manja od kapaciteta sistema, tj. od broja kanala opsluge.
3. Opsluga korisnika je kvalitetna ako je srednje vreme provedeno u redu manje od srednjeg vremena opsluge. (apsolutni kriterijum kvaliteta opsluge)

Prema rezultatima dobijem simulacionom analizom i gore navedenim kriterijumima dolazi se do zaključka da šalterska služba, kakva je definisana u ovom radu, u celini ne posluje kvalitetno po apsolutnom kriterijumu kvaliteta opsluge. Po prvom i drugom kriterijumu ne posluje dobro u drugom segmentu.

6. ZAKLJUČAK

DEVS je formalizam za modeliranje i simulaciju dinamičkih sistema zasnovanih na događajima za koji su karakteristični modularnost i hijerarhija. Mogućnost hijerarhijske kompozicije modela složenih sistema spajanjem modularnih komponenata je veoma značajna paradigma koja se se sve više koristi u simulaciji.

U radu je prikazana implementacija modela šalterske službe u programskom jeziku C++ primenom klasične DEVS specifikacije. Osnovu programa predstavlja biblioteka sa implementacijom DEVS modela i glavne simulacione petlje. Koristeći prednosti objektno-orijentisanog pristupa implementiran je niz komponenata koje predstavljaju različite procese u sistemu nasleđivanjem osnovnih DEVS modela. Tokom izrade pojedinih komponenata modela, a sa ciljem da se isti uprosti, napravljeni su određeni kompromisi po pitanju njihove implementacije. Tu se pre svega misli na blokove čija funkcija nije nedeljiva, što se najbolje vidi na primeru bloka *queue*, i blokova koji su nastali sa ciljem da nadoknade nedostatke nekih drugih blokova, kao što je slučaj kod bloka buffer. Adekvatnim izmenama u postojećem programu i implementacijom dodatnih komponenata (redova sa različitim disciplinama opsluživanja, servera koji obrađuju nekoliko entiteta od jednom itd.) dobio bi se set modula opšte namene, kojima bi bilo moguće razviti modele najrazličitijih realnih sistema.

Implementirani model iskorišćen za simulacionu analizu rada šalterske službe pošte 11030 Beograd. Dobijeni podaci potvrđuju rezultate dobijene simulacionom analizom modela razvijenom u jeziku GPSS iz rada [4].

LITERATURA

[1] Radenković B., Stanojević M., Marković A., Računarska simlacija, Beograd, Srbija, Saobraćajni fakultet (2009).

[2] Gabriel A. Wainer, Discrete-Event Modeling and Simulation, A practitioner's Approach, 2009.

[3] Đogatović M., Stanojević M., Simuacija sistema masnovnog opsluživanja korišćenjem klasične DEVS specifikacije. SYM-OP-IS 2014.

[4] Samardžić I., Simulaciona analiza kvaliteta opsluživanja šalterske službe jedinicke poštanske mreže 11030 Beograd 8, Diplomski rad, 2004.

[5] Boost C++ Libraries, www.boost.org

[6] B. Stroustrup, Programming: Principles and Practice Using C++, Second Edition, Addison-Wesley Publishing Company

PRILOG 1. Izvorni kod simulatora

```
#ifndef DEVS_HPP
#define DEVS_HPP

#include <iostream>
#include <functional>
#include <stdexcept>
#include <map>
#include <vector>
#include <utility>
#include <string>
#include <limits>
#include <cmath>
#include <boost/any.hpp>

namespace devs {
        typedef double time;
        typedef std::string str;
        typedef boost::any state;
        typedef boost::any port_value;
        typedef str port_label;
        typedef str state_label;
        typedef str devs_label;
        typedef std::map<state_label, state> state_map;
        /*
         * Vrste blokova
         */
        enum devs_type {
            atomic,
            coupled,
            //esched STUB
        };

        const boost::any empty_value = boost::any();
        const time infinity = std::numeric_limits<time>::infinity();
        const time eps = std::numeric_limits<time>::epsilon();

        template <typename T = time>
        T cast(const boost::any& v) {
            return boost::any_cast<T>(v);
        }

        class bad_synchronization : public std::domain_error {
        public:
            bad_synchronization():domain_error("Bad synchronization\n") {}
        };

        class negative_time : public std::domain_error {
        public:
            negative_time():domain_error("Negative time\n") {}
        };

        class port;
        class atomic_devs;
        class coupled_devs;
        //class es_devs; STUB
        class simulator;
        typedef std::map<port_label, port*> port_map;

         class port {
            str name;
            atomic_devs* parent_devs;
            port_value value;
            port_map preceding_ports, succeeding_ports;


         public:
            port(const str name):name(name),parent_devs(nullptr) {}
            const str& get_name() const { return name; }
            void add_preceding_port(port& p) {
                preceding_ports[p.get_name()] = &p;
            }
            void add_succeeding_port(port& p) {
                succeeding_ports[p.get_name()] = &p;
            }
```

```
    port& get_preceding_port(const str& n) {
        return *preceding_ports[n];
    }
    port& get_succeeding_port(const str& n) {
        return *succeeding_ports[n];
    }
    const port_map get_preceding_ports() const {
        return preceding_ports;
    }
    const port_map get_succeeding_ports() const {
        return succeeding_ports;
    }
    void set_value(const port_value& v) { value = v; }
    port_value get_value() { return value; }
    void set_parent(atomic_devs* p) { parent_devs = p; }
    atomic_devs* get_parent() const { return parent_devs; }
    static void coupling(port& in, port& out) {
        in.add_preceding_port(out);
        out.add_succeeding_port(in);
    }
    port& operator=(port& po) {
        add_preceding_port(po);
        po.add_succeeding_port(*this);
        return po;
    }
    const port_value operator=(const port_value v) {
        set_value(v);
        return v;
    }
    operator bool() const { return !value.empty(); }
    void reset() { value = empty_value; }
};

class atomic_devs {
    friend class coupled_devs;
    //friend class es_devs; STUB
    friend class simulator;
protected:
    devs_type type;
    coupled_devs* parent_devs;
    str name;
    port_map in_ports, out_ports;
    state_map states;
    time tn, tl;

    virtual void update(time t);
    virtual void i_msg(time t);
    virtual void star_msg(time t);
    virtual void x_msg(time t);

    time ta_check() {
        time t = ta();
        if (t < 0)
            throw negative_time();
        return t;
    }
 public:
 atomic_devs(const str& name):parent_devs(nullptr), name(name),
                                        tl(0.0),type(atomic) {}
    std::function<void()> delta_int;
    std::function<bool(time)> delta_ext;
    std::function<void()> lambda;
    std::function<time()> ta;

    const str& get_name() const { return name; }
    void set_name(const str& n) { name = n; }
    void set_tl(const time t) { tl = t; }
    const time get_tl() const { return tl; }
    void set_tn(const time t) { tn = t; }
    const time get_tn() const { return tn; }
    void set_parent(coupled_devs& p) { parent_devs = &p; }
    void add_in_port(port& in) {
        in.set_parent(this);
        in_ports[in.get_name()] = &in;
    }
    void add_out_port(port& out) {
        out.set_parent(this);
        out_ports[out.get_name()] = &out;
```

```
    }
    port& get_in_port(const port_label& n) {
        return *in_ports[n];
    }
    port& get_out_port(const port_label& n) {
        return *out_ports[n];
    }
    state& operator[](const state_label& n) {
        return states[n];
    }
    const devs_type get_type() const { return type; }
};

class coupled_devs : public atomic_devs {
    friend class atomic_devs;
    friend class simulator;
protected:
    std::map<devs_label, atomic_devs*> children;
    atomic_devs* d_star;

    static bool compare(const std::pair<devs_label, atomic_devs*>& a,
                        const std::pair<devs_label, atomic_devs*>& b) {
        return a.second->tn < b.second->tn;
    }

    virtual void update(time t);
    virtual void i_msg(time t);
    virtual void star_msg(time t);
    virtual void x_msg(time t);
    virtual void y_msg(port_map& y, time t);
public:
    coupled_devs(const str& name):atomic_devs(name) {
        type = coupled;
    }

    std::function<void()> select; //STUB

    void add_child(atomic_devs& ad) {
        children[ad.get_name()] = &ad;
        ad.set_parent(*this);
    }


 void add_child(coupled_devs& cd) {
        children[cd.get_name()] = &cd;
        cd.set_parent(*this);
    }
    void set_d_star(atomic_devs* ad) {
        d_star = ad;
    }
    void update_input(time t) {
        x_msg(t);
    }
};
class simulator {
    time t;
    coupled_devs* child;
    bool stop_flag;
public:
    simulator():child(nullptr),t(0.0),stop_flag(false) {}
    const time get_t() const { return t; }
    void set_t(const time tn) { t = tn; }
    void set_child(coupled_devs& cd) { child = &cd; }
    void operator()(const time T=infinity, const bool ini=true) {
        if (ini) {
            child->i_msg(t);
            t = child->get_tn();
        }
        while (!stop_flag && t < T) {
            child->star_msg(t);
            t = child->get_tn();
        }
    }
    void stop() { stop_flag = true; }
};
template<typename UID>
class statistics {
    int num, curr_num, max_num;
```

```
            int nowait_num;
            time total_time;
            time area;
            time time_instant;
            static time transition_time;
            std::map<UID, time> idtime;
        public:
            statistics():num(0), curr_num(0), max_num(0), nowait_num(0),
                         total_time(0.0), area(0.0), time_instant(0) {}
            void start(const UID& id, time ti) {
                if (ti > transition_time) {
                    num++;
                    area += (curr_num++) * (ti - time_instant);
                    time_instant = ti;
                    max_num = (curr_num > max_num) ? curr_num : max_num;
                    idtime[id] = ti;
                }
            }
           void end(const UID& id, time ti) {
                if (idtime.find(id) != idtime.end()) {
                    time dt = ti - idtime[id];
                    if (abs(dt) < eps)
                            nowait_num++;
                    total_time += dt;
                    area += (curr_num--) * (ti - time_instant);
                    time_instant = ti;
                    idtime.erase(id);
                }
            }
            inline void finalize(time ti) {
                area += curr_num * (ti - time_instant);
            }
            void update(time te) {
                total_time +=te;
            }
            static void set_transition_time(time);
            inline int get_num() const { return num; }
            inline int get_curr_num() const { return curr_num; }
            inline int get_max_num() const { return max_num; }
            inline int get_nowait_num() const { return nowait_num; }
            inline int get_total_time() const { return total_time; }
            inline time get_mean_time() {
                return total_time / (num - curr_num);
            }
            inline time get_mean_time_nowait() {
                return total_time / (num - curr_num - nowait_num);
            }
            inline double get_mean_num(time sim_time) {
                return area / sim_time;
            }
            inline double get_util(time sim_time, int num_of_servers=1) {
                return get_mean_num(sim_time) / num_of_servers;
            }
            inline double get_nowait_contrib() {
                return (100.0 * nowait_num) / num;
            }
        };
        template <typename UID>                                  //UPDATE
        time statistics<UID>::transition_time = 0.0;
        template <typename UID>
        void statistics<UID>::set_transition_time(time  tt) {
            transition_time = tt;
        }
}

#endif   /* DEVS_HPP */
```

```
#include "devs.hpp"

namespace devs {
    void atomic_devs::update(time t) {
        tl = t;
        tn = tl + ta_check();
    }
    void atomic_devs::i_msg(time t) {
        update(0.0);
    }
    void atomic_devs::star_msg(time t) {
        if (abs(t - tn) > eps)
            throw bad_synchronization();
        lambda();
        if (parent_devs)
            parent_devs->y_msg(out_ports, t);
        delta_int();
        update(t);
    }
    void atomic_devs::x_msg(time t) {
        if (t < tl || t > tn)
            throw bad_synchronization();
        if (delta_ext(t - tl))
            update(t);
    }
    void coupled_devs::update(time t) {
        d_star = std::min_element(children.begin(), children.end(),
                                          &coupled_devs::compare)->second;
        tl = t;
        tn = d_star->tn;
    }
    void coupled_devs::i_msg(time t) {
        for (auto d:children)
            d.second->i_msg(t);
        update(t);
    }
    void coupled_devs::star_msg(time t) {
        if (abs(t - tn) > eps)
            throw bad_synchronization();
        if (select)
            select();
        d_star->star_msg(t);
        update(t);
    }
    void coupled_devs::x_msg(time t) {
        if (t < tl || t > tn)
            throw bad_synchronization();
        for (auto p:in_ports) {
            port* pp = p.second;
            if (*pp) {
                for (auto sp:pp->get_succeeding_ports()) {
                    port* spp = sp.second;
                    spp->set_value(pp->get_value());
                    spp->get_parent()->x_msg(t);
                    spp->reset();
                }
                pp->reset();
            }
        }
        update(t);
    }
    void coupled_devs::y_msg(port_map& y,time t) {
        if(t<tl||t>tn)
            throw bad_synchronization();
            for(auto p:y) {
              port* pp = p.second;
              if(*pp) {
                for(auto sp:pp->get_succeeding_ports()) {
                    port* spp = sp.second;
                    if(spp->get_parent()==this) {
                        if(parent_devs) {
                             for(auto sspp:spp->get_succeeding_ports()) {
                                port* ssppp = sspp.second;
                                ssppp->set_value(pp->get_value());
                                ssppp->get_parent()->x_msg(t);
                                ssppp->reset();
                            }
                        }
```

```
                    }
                    else {
                        spp->set_value(pp->get_value());
                        spp->get_parent()->x_msg(t);
                        spp->reset();
                    } } pp->reset(); } } } }

/*
 * File:   devs_assign.hpp
 * Author: Dušan Tadić
 *
 * Definicija bloka za dodelu parametara entitetima
 *
 * Created on July 21, 2015, 4:47 PM
 */
#ifndef DEVS_ASSIGN_HPP
#define  DEVS_ASSIGN_HPP

#include "devs.hpp"
#include "devs_utility.hpp"
#include <vector>
namespace devs_block {
    using namespace devs_supplement;
    typedef std::function<void(entity*)> definition;
    typedef std::vector<definition> definition_list;
    class assign : public devs::atomic_devs {
        definition_list list; //Lista pravila po kojima se entitetima upisuju parametri
        devs::port* in;       //Ulazni port
        devs::port* out;      //Izlazni port
        entity* inter_buff;   //Entitet koji je trenutno u bloku
        enum state {
            ASSIGN_IDLE,
            ASSIGN_ASSIGNING
        };
    public:
        assign(const std::string&);
        void param_lambda(definition);
        ~assign();
    };
}
#endif   /* DEVS_ASSIGN_HPP *
#include "devs_assign.hpp"
namespace devs_block {
    assign::assign(const std::string& name):devs::atomic_devs(name) {
        //Inicijalizacija portova
        in = new devs::port("?in");
        out = new devs::port("!out");

        delta_int = [&]() {
            current_state = ASSIGN_IDLE;
        };

        delta_ext = [&](devs::time t) -> bool {
            current_state = ASSIGN_ASSIGNING;
            //Vadjenje entiteta iz ulaznog porta
            inter_buff = ent_cast(in->get_value());
            //Dodeljivanje parametara po definicijama
            for (auto d:list) {
                d(inter_buff);
            }
            return true;
        };
        lambda = [&]() {
            if (get_state() == ASSIGN_ASSIGNING) {
                //Prebacivanje entiteta iz internog buffera u izlazni port
                out->set_value(inter_buff);
                inter_buff = nullptr;
            }
        };
        ta = [&]() -> devs::time {
            if (get_state() == ASSIGN_ASSIGNING) {
                return 0; //Dodeljivanje parametara je trenutno
            }
            else {
                return devs::infinity;
            }
        };
        add_in_port(*in);
```

```
        add_out_port(*out);
        current_state = ASSIGN_IDLE;
    }
    //Dodavanje definicije u listu
    void assign::param_lambda(definition defin) {
        list.push_back(defin);
    }
    assign::~assign() {
        delete in;
        delete out;
    }
}
/*
 * File:   devs_buffer.hpp
 * Author: DuÅ¡an TadiÄ‡
 *
 * Definicija buffer bloka
 * Uproscena varijanta devs_block::queue bloka.
 *
 * Created on July 20, 2015, 1:28 PM
 */
#ifndef BUFFER_HPP
#define  BUFFER_HPP
#include "devs.hpp"
#include "devs_utility.hpp"
#include <deque>
namespace devs_block {
    using namespace devs_supplement;
    class buffer : public devs::atomic_devs {
        bool blocking;           //Status povezanog bloka
        devs::port* in;          //Ulazni port
        devs::port* pull;        //Port za signal da se prosledi sledeci entitet
        devs::port* out;         //Izlazni port
        std::deque<entity*> buff;//Buffer
        enum state {
            BUFFER_IDLE,
            BUFFER_SEND_TO
        };
    public:
        buffer(const std::string&);
        ~buffer();
    };
}
#endif   /* BUFFER_HPP */
#include "devs_buffer.hpp"
namespace devs_block {
    buffer::buffer(const std::string& name):devs::atomic_devs(name) {
        //Inicijalizacija portova
        in = new devs::port("?in");
        out = new devs::port("!out");
        pull = new devs::port("?pull");
        blocking = false;
        delta_int = [&]() {
            current_state = BUFFER_IDLE;
        };
        delta_ext = [&](devs::time e) -> bool {
            if (*in) {//Dogadjaj na ulaznom portu
                  //Vadjenje entiteta iz ulaznog porta
                   entity* ent = ent_cast(in->get_value());
                   buff.push_back(ent);
                if (!blocking) {                    //Ako je prolaz slobodan
                    current_state = BUFFER_SEND_TO; //Predji u stanje za slanje
                    return true;
                }
            }
            else {
                if (*pull) {                //Dogadjaj na pull portu
                    blocking = false;
                    current_state = BUFFER_SEND_TO; // Predji u stanje za slanje
                    return true;
                }
            }

        };
         lambda = [&] () {
             if (get_state() == BUFFER_SEND_TO) {
                if (!buff.empty()) {
                    entity* send_entity = buff.front();
```

```
                    buff.pop_front();                   //Izvadi ga iz buffera
                    out->set_value(send_entity);        //Prosledi ga u naredni blok
                    blocking = true;  //Blokiraj izlaz dok entitet ne prodje
                }
            }
        };
        ta = [&]() -> devs::time {
            if (get_state() == BUFFER_SEND_TO) {
                return 0;
            }
            else {
                return devs::infinity;
            }
        };
        add_in_port(*in);
        add_in_port(*pull);
        add_out_port(*out);
        current_state = BUFFER_IDLE;
    }
    buffer::~buffer() {
        delete in;
        delete out;
        delete pull;
    }
}

/*
 * File:   devs_generate.hpp
 * Author: Dušan Tadić
 *
 *  Definicija bloka za generisanje entiteta
 *
 * Created on July 20, 2015, 11:48 PM
 */

#ifndef DEVS_GENERATE_HPP
#define  DEVS_GENERATE_HPP

#include "devs.hpp"
#include "devs_utility.hpp"

namespace devs_block {
    using namespace devs_supplement;
    class generate : public devs::atomic_devs {
        devs::simulator* sim;
//Racunanje vremena sledeceg generisanja
        std::function<devs::time(devs::simulator*)> generate_next;
        enum state {
            GENERATE_GENERATING
        };
        enum type {
            FROM_LAMBDA_EXP,
            FROM_PARAMS
        };
        type gen_type;
        devs::port* out; //Izlazni port
        int count;       //Broj generisanih entieta
        bool sys;        //Da li su entiteti sistemski
        bool done;       //Da li je generator zavrsio sa generisanjem
        entity* inter_buff; //Entitet koji je trenutno u bloku
        devs::time start;   //Vreme generisanja prvog entiteta
        devs::time next;    //Vreme generisanja narednog entiteta
        int n;              //Broj entiteta koji treba da se generisu
    public:
        generate(const std::string&, devs::simulator*);
        //Postavljanje funkcije koja vraca vreme generisanja novog entiteta
        void from_lambda_exp(std::function<devs::time(devs::simulator*)>);
        void from_params(devs::time, devs::time, int);
        //Odredjivanje da li su entiteti sistemski
        void system_entity(bool);
        ~generate();
    };
}


#endif   /* DEVS_GENERATE_HPP */
```

```
#include "devs_generate.hpp"

namespace devs_block {
     generate::generate(const std::string& name, devs::simulator* s):
                                                                    devs::atomic_devs(name) {
        //Inicijalizacija portova
        out = new devs::port("!out");
        sim = s;

        delta_int = [&]() {};
        delta_ext = [&](devs::time e) -> bool {};
        lambda = [&]() {
            if (get_state() == GENERATE_GENERATING) {
                inter_buff = new entity;                //Generisanje novog entiteta
                if (sys) {
                    inter_buff->set("SYS_FLAG", 1);     //Postavljanje zastavice za
                }                                       //sistemske entitete
                else {
                    inter_buff->set("SYS_FLAG", 0);
                }
                //Cuvanje vremena generisanja entiteta
                inter_buff->set("GENERATED_AT", sim->get_t());
                inter_buff->set("PRIORITY", 0);                //Default prioritet
                //inter_buff->set("GENERATED_BY", this->name); //DEBUG
                out->set_value(inter_buff);      //Prosledjivanje entiteta
                inter_buff = nullptr;
                count++;                      //Uvecavanje broja generisanih entiteta
                if (count == n && n != 0) {     //Ako su generisani svi entiteti
                            next = devs::infinity;
  }
            }
        };
        ta = [&]() -> devs::time {
            if (gen_type == FROM_LAMBDA_EXP) {
                return generate_next(sim);
            }
            else {
                if (start != -1.0) {    //Ako vec nije prosledjena
                    devs::time t;       //Prosledi pocetnu vrednost
                    t = start;
                    start  = -1.0;
                    return t;
                }
                else {        //Prosledi vreme generisanja sledeceg entiteta
                    return next;
                }
            }
        };
        sys = false;
        count = 0;
        start = 0.0;
        next = 0.0;
        n = 0;
        add_out_port(*out);
        current_state = GENERATE_GENERATING;
    }

    void generate::from_lambda_exp(std::function<devs::time(devs::simulator*)> r) {
        gen_type = FROM_LAMBDA_EXP;
        generate_next = r;
    }
    void generate::from_params(devs::time start_time, devs::time next_time, int n_of_entities)  {
        gen_type = FROM_PARAMS;
        start = start_time;
        next = next_time;
        n = n_of_entities;
    }
    void generate::system_entity(bool s) {
        sys = s;
    }
    generate::~generate() {
        delete out;
    }
}
```

```
/*
 * File:   devs_input_switch.hpp
 * Author: Dušan Tadić
 *
 * Definicija bloka za odabir izlaza
 *
 * Created on July 21, 2015, 10:31 PM
 */
#ifndef DEVS_INPUT_SWITCH_HPP
#define  DEVS_INPUT_SWITCH_HPP
#include "devs.hpp"
#include "devs_utility.hpp"
namespace devs_block {
    using namespace devs_supplement;
    class input_switch : public devs::atomic_devs {
        std::string val;  //Ime parametra kojim se odredjuje izlazni port
        int n;            //Broj izlaza
        enum state {
            SWITCH_IDLE,
            SWITCH_SWITCHING
        };
        devs::port* in;     //Ulazni port
        devs::port** out;   //Izlazni port
        entity* inter_buff; //Entitet koji je trenutno u bloku
    public:
        input_switch(const std::string&, int);
        void switch_param(const std::string&); //Iz kog parametra se čita izlazni port
        ~input_switch();
    };
}

#endif   /* DEVS_INPUT_SWITCH_HPP */
#include "devs_input_switch.hpp"
namespace devs_block {
    input_switch::input_switch(const std::string& name, int n):devs::atomic_devs(name),n(n) {
        //Inicijalizacija portova
        in = new devs::port("?in");
        out = new devs::port*[n];
        for (int i = 0; i < n; ++i) {
            out[i] = new devs::port("!out" + std::to_string(i));
        }
        delta_int = [&]() {
            current_state = SWITCH_IDLE;
        };
        delta_ext = [&](devs::time) -> bool {
            inter_buff = ent_cast(in->get_value()); //Vadjenje entiteta iz porta
            current_state = SWITCH_SWITCHING;
            return true;
        };
        lambda = [&]() {
            if (get_state() == SWITCH_SWITCHING) {
                //Postavljanje entiteta u izlazni port
                out[inter_buff->get(val)]->set_value(inter_buff);
                inter_buff = nullptr;
            }
        };
        ta = [&]() -> devs::time {
            if (get_state() == SWITCH_SWITCHING) {
                return 0;
            }
            else {
                return devs::infinity;
            }
        };
        add_in_port(*in);
        for (int i = 0; i < n; ++i) {
            add_out_port(*out[i]);
        }
        current_state = SWITCH_IDLE;
    }
    //Ime parametra u kojem stoji broj izlaza
    void input_switch::switch_param(const std::string& tag) {
         val = tag;
    }
    input_switch::~input_switch() {
        delete in;
        for (int i = 0; i < n; ++i) delete out[i];
        delete[] out; } }
```

```
/*
 * File:   devs_queue.hpp
 * Author: Dušan Tadić
 *
 * Definicija bloka za red cekanja
 *
 * Created on July 20, 2015, 1:28 PM
 */

#ifndef QUEUE_HPP
#define  QUEUE_HPP

#include "devs.hpp"
#include "devs_utility.hpp"
#include <deque>

namespace devs_block {
    using namespace devs_supplement;
    class queue : public devs::atomic_devs {
        devs::simulator* sim;   //Pointer na simulator
        bool* server_status;    //Status povezanih servera
        devs::port* in;         //Ulazni port
        devs::port** pull;      //Port za signal da se server oslobodio
        devs::port** out;       //Izlazni port
        devs::port* count;      //Port za emitovanje duzine reda
        std::deque<entity*> que;//Red cekanja
        int where_to;           //U koji server se salje entitet
        int n;                  //Broj povezanih servera
        enum state {
            QUEUE_IDLE,
            QUEUE_RESIZE,
            QUEUE_SEND_TO
        };
    public:
        devs::statistics<int> stats;
        queue(const std::string&, devs::simulator*, int);
        ~queue();
    };
}
#endif   /* QUEUE_HPP */
#include "devs_queue.hpp"
namespace devs_block {
    queue::queue(const std::string& name,
                    devs::simulator* sim,
                int n = 1):devs::atomic_devs(name),sim(sim),n(n) {
        //Inicijalizacija portova
        in = new devs::port("?in");
        out = new devs::port*[n];
        pull = new devs::port*[n];
        count = new devs::port("!count");
        server_status = new bool[n](); //() - Zero init
        for (int i = 0; i < n; ++i) {
            out[i] = new devs::port("!out" + std::to_string(i));
            pull[i] = new devs::port("?pull" + std::to_string(i));
        }
        delta_int = [&, sim]() {
            current_state = QUEUE_IDLE;
        };
        delta_ext = [&, sim, n](devs::time e) -> bool {
            if (*in) {//Dogadjaj na ulaznom portu
                //Vadjenje entiteta iz ulaznog porta
                entity* ent = ent_cast(in->get_value());
                if (ent->get("SYS_FLAG") == 0) {     //Ako entitet nije sistemski
                    stats.start(ent->id, sim->get_t());   //Loguj ulaz
                    que.push_back(ent);      //Dodavanje entiteta na kraj reda
                }
                else {                    //Ako je entitet sistemski
                    que.push_front(ent);  //Dodaj ga na pocetak reda
                }
                for (int i = 0; i < n; ++i) {
                    if (!server_status[i]) {       //Ako ima slobodnih servera
                            current_state = QUEUE_SEND_TO;
                            where_to = i;    //Zapamti koji server se oslobodio
                            return true;
                    }
                }
                current_state = QUEUE_RESIZE;
                return true; }
```

```
else {
        for (int i = 0; i < n; ++i) {
            if (*pull[i]) {
                server_status[i] = false;
                current_state = QUEUE_SEND_TO; // Predji u stanje za slanje
                where_to = i;
                return true;
            }
        }
    }
};
 lambda = [&, sim] () {
    if (get_state() == QUEUE_SEND_TO) {
        if (!que.empty()) {
            //Entitet koji se salje na opslugu
            entity* send_entity = nullptr;
            /*
             * Da li u redu ceka neki sistemski entitet na oslobodjeni server
             */
            for (auto e:que) {
                if (e->get("SYS_FLAG") == 1 &&
                          e->get("DEDICATED_SERVER") == where_to) {
                    send_entity = e;
                    break;
                }
            }
     /*
      * Ako nema sistemskih entiteta naci entitet
      * sa najvisim prioritetom koji ceka na oslobodjeni server
          */
            if (send_entity == nullptr) {
                int priority_max = 0;
                entity* temp_max = nullptr;
                for (auto e:que) {
                    if (e->get("PRIORITY") > priority_max &&
                       e->get("DEDICATED_SERVER") == where_to) {
                        priority_max = e->get("PRIORITY");
                        temp_max = e;
                    }
                }
                if (temp_max != nullptr) {
                    send_entity = temp_max;
                }
            }
       /*
        * Ako nema entiteta koji cekaju ovaj oslobodjeni server naci entitet
        * koji nema preferencu na server
        */
            if (send_entity == nullptr) {
                for (auto e:que) {
                    if (e->get("DEDICATED_SERVER") == -1) {
                        send_entity = e;
                        break;
                    }
                }
            }
            if (send_entity != nullptr) { //Postoji entitet
                      que.erase(std::find(que.begin(), que.end(), send_entity));
                if (send_entity->get("SYS_FLAG") == 0) {
                    stats.end(send_entity->id, sim->get_t()); //loguj izlaz
                }
                //Prosledjivanje entiteta u izlazni port
                out[where_to]->set_value(send_entity);
                      //Prosledjivanje duzine reda u signalni port
                count->set_value(que.size());
                      //Upisivanje statusa servera na tom portu
                server_status[where_to] = true;
            }
        }
    }
    if (get_state() == QUEUE_RESIZE) {
        count->set_value(que.size());    //Duzina reda se povecala
    }
};
ta = [&]() -> devs::time {
    if (get_state() == QUEUE_SEND_TO) {
        return 0;
    }
```

```
            else if (get_state() == QUEUE_RESIZE) {
                return 0;
            }
            else {
                return devs::infinity;
            }
        };

        add_in_port(*in);
        add_out_port(*count);
        for (int i = 0; i < n; ++i) {
            add_in_port(*pull[i]);
            add_out_port(*out[i]);
        }
        current_state = QUEUE_IDLE;
    }
    queue::~queue() {
        for (int i = 0; i < n; ++i) {
            delete out[i];
            delete pull[i];
        }
        delete in;
        delete[] server_status;
        delete count;
        delete[] out;
        delete[] pull;
    }
}


/*
 * File:   devs_select.hpp
 * Author: Dušan Tadić
 *
 * Definicija bloka za izbor kraceg reda cekanja
 *
 * Created on July 23, 2015, 3:34 AM
 */

#ifndef DEVS_SELECT_HPP
#define  DEVS_SELECT_HPP

#include "devs.hpp"
#include "devs_utility.hpp"
namespace devs_block {
    using namespace devs_supplement;
    class select_queue : public devs::atomic_devs {
        int n;              //Broj povezanih redova cekanja
        devs::port* in;     //Ulazni port
        devs::port** out;   //Izlazni portovi
        devs::port** count; //Portovi za prijem signala o promenjenoj duzini reda
        std::size_t* queue_sizes; //Niz duzina redova
        int selected;             //Izabrani najkraci red
        entity* inter_buff;       //Trenutni entitet u bloku
        std::vector<int> weight;  //Tezinski faktor (broj servera po redu cekanja)
        enum state {
            SELECT_IDLE,
            SELECT_SELECTING
        };
    public:
        select_queue(const std::string&, int);
        void set_weights(std::vector<int>); //Postavljanje tezinskih faktora
        virtual ~select_queue();
    };
}


#endif   /* DEVS_SELECT_HPP
```

```
#include "devs_select.hpp"

namespace devs_block {
    select_queue::select_queue(const std::string& name, int num):devs::atomic_devs(name),
                                                                  n(num), weight(num,1) {
        //Inicijalizacija portova
        in = new devs::port("?in");
        out = new devs::port*[n];
        count = new devs::port*[n];
        queue_sizes = new std::size_t[n]();
        for (int i = 0; i < n; ++i) {
            out[i] = new devs::port("!out" + std::to_string(i));
            count[i] = new devs::port("?count" + std::to_string(i));
        }
        delta_int = [&]() {
            current_state = SELECT_IDLE;
        };
        delta_ext = [&](devs::time e) -> bool {
            if (*in) {//Dogadjaj na ulaznom portu
                entity* ent = ent_cast(in->get_value()); //Vadjenje entiteta
                inter_buff = ent;
                current_state = SELECT_SELECTING; // izbor kraceg reda
                double min = 1.0 * queue_sizes[0] / weight[0];
                int indx = 0;
                for (int i = 1; i < n; ++i) {
                    if (1.0 * queue_sizes[i] / weight[i] < min) {
                        min = queue_sizes[i];
                        indx = i;
                    }
                }
                selected = indx;    //Cuvanje rednog broja najkraceg reda cekanja
                return true;
            }
            else {
                for (int i = 0; i < n; ++i) {
                    if (*count[i]) {//Signal za promenu duzine reda
                      //Cuvanje nove duzine reda
                            queue_sizes[i] = devs::cast<std::size_t>(count[i]->get_value());
                    }
                }
                return false;
            }
        };
        lambda = [&]() {
            if (get_state() == SELECT_SELECTING) {
                out[selected]->set_value(inter_buff); //Prosledjivanje entiteta
                inter_buff = nullptr;
            }
        };
        ta = [&]() -> devs::time {
            if (get_state() == SELECT_SELECTING) { return 0; }
            else { return devs::infinity; }
        };
        add_in_port(*in);
        for (int i = 0; i < n; ++i) {
            add_out_port(*out[i]);
            add_in_port(*count[i]);
        }
        current_state = SELECT_IDLE;
    }
    //Postavljanje tezinskih faktora(broj servera po redu cekanja)
    void select_queue::set_weights(std::vector<int> w) {
        weight = w;
    }
    select_queue::~select_queue() {
        for (int i = 0; i < n; ++i) {
            delete out[i];
            delete count[i];
        }
        delete in;
        delete[] queue_sizes;
        delete[] out;
        delete[] count;
    }
}
```

```
/*
 * File:   devs_server.hpp
 * Author: Dušan Tadić PS110243
 *
 *  Definicija bloka server
 *
 * Created on July 20, 2015, 9:54 PM
 */

#ifndef DEVS_SERVER_HPP
#define  DEVS_SERVER_HPP

#include "devs.hpp"
#include "devs_utility.hpp"

namespace devs_block {
    using namespace devs_supplement;
    class server : public devs::atomic_devs {
        entity* inter_buff;       //Entitet koji je trenutno u bloku
        enum state {
            SERVER_IDLE,
            SERVER_SERVING
        };
        devs::port* in;    //Ulazni port
        devs::port* out;   //Izlazni port
        //Implementirano zbog servera 5*
        devs::port* pull;  //Signal za oslobodjen server
        devs::simulator* sim; //Simulator
        devs::time closing_time; //Kraj radnog vremena
        std::vector<int> st;     //Lista pruzenih usluga DEBUG
    public:
        devs::statistics<int> stats;
        server(const std::string&, devs::simulator*);
        void closes_at(devs::time);
        ~server();
    };
}
#endif   /* DEVS_SERVER_HPP */
#include "devs_server.hpp"

namespace devs_block {
    server::server(const std::string& name, devs::simulator* sim):devs::atomic_devs(name),
                                                                  sim(sim), inter_buff(nullptr),st(13,0) {
        in = new devs::port("?in");
        out = new devs::port("!out");
        pull = new devs::port("!pull");
        delta_int = [&, sim]() {
            current_state = SERVER_IDLE;
        };
        delta_ext = [&, sim](devs::time t) -> bool {
            entity* ent = ent_cast(in->get_value());
            current_state = SERVER_SERVING;           //Promena stanja u opsluzivanje
            inter_buff = ent;                         //Postavljanje entiteta u blok
            if (inter_buff->get("SYS_FLAG") == 0)
                stats.start(ent->id, sim->get_t());
            }
            return true;
        };
        lambda = [&, sim]() {
            if (get_state() == SERVER_SERVING) {
                out->set_value(inter_buff);   //Prosledi entitet u naredni blok
                if (inter_buff->get("SYS_FLAG") == 0) {
                    stats.end(inter_buff->id, sim->get_t());
                }
                if (sim->get_t() < closing_time || closing_time == 0.0) {
                    pull->set_value(true);
                }
                inter_buff = nullptr;
            }
        };

        ta = [&]() -> devs::time {
            if(get_state() == SERVER_SERVING) {
                return inter_buff->get<devs::time>("SERVICE_TIME");
            }
            else {
                return devs::infinity;
            }
```

```
        };
        add_in_port(*in);
        add_out_port(*out);
        add_out_port(*pull);
        closing_time = 0.0;
        current_state = SERVER_IDLE;
    }
    void server::closes_at(devs::time t) {
        closing_time = t;
    }
    server::~server() {
        for (int i = 0; i < st.size(); ++i) {
            std::cout << st[i] << " ";
        }
        std::cout << std::endl << std::endl;*/
        delete in;
        delete out;
    }
}

/*
 * File:   devs_transducer.hpp
 * Author: Dušan Tadić
 *
 *  Definicija bloka za unistavanje entiteta
 *
 * Created on July 21, 2015, 11:51 AM
 */

#ifndef DEVS_TRANSDUCER_HPP
#define  DEVS_TRANSDUCER_HPP

#include "devs.hpp"
#include "devs_utility.hpp"

namespace devs_block {
    using namespace devs_supplement;
    class transducer : public devs::atomic_devs {
        enum state {
            TRANSDUCER_WAIT
        };
        devs::port* in; //Ulazni port
    public:
        transducer(const std::string&);
        ~transducer();
    };
}
#endif   /* DEVS_TRANSDUCER_HPP */

#include "devs_transducer.hpp"

namespace devs_block {
    transducer::transducer(const std::string& name):devs::atomic_devs(name) {
        //Inicijalizacija porta
        in = new devs::port("?in");

        delta_int = [&]() {};

        delta_ext = [&](devs::time t) -> bool {
            entity* ent = devs::cast<entity*>(in->get_value());
            delete ent;
            return false;
        };
        lambda = [&]() {};

        ta = [&]() -> devs::time { return devs::infinity; };

        add_in_port(*in);
        current_state = TRANSDUCER_WAIT;
    }
    transducer::~transducer() {
        delete in;
    }
}
```

```
/*
 * File:   devs_utility.hpp
 * Author: Dušan Tadić
 *
 * Dodaci devs API-u
 *
 * Created on July 20, 2015, 1:30 PM
 */

#ifndef DEVS_UTILITY_HPP
#define  DEVS_UTILITY_HPP

#include <map>
#include <string>
#include <boost/any.hpp>

#include "devs.hpp"

namespace devs_supplement {
    #define current_state (*this)["phase"]
    #define ent_cast(e) devs::cast<entity*>(e)
    #define get_state() devs::cast<state>(current_state)

    typedef unsigned long ulong;
    typedef boost::any parameter_value;
    typedef std::map<std::string, parameter_value> parameter_map;

    class entity {
        static ulong eid;                   //Brojac za jedinstveni ID
        parameter_map parameters;           //Mapa parametara i njihovih oznaka
    public:
        ulong id;                           //Jedinstveni ID
        entity();
        template<typename T=int>            //Vracanje parametra
        T get(const std::string& tag) {
            if (parameters.find(tag) !=  parameters.end()) {
                return boost::any_cast<T>(parameters[tag]);
            }
        }
        entity(const entity&);
        void set(const std::string&, parameter_value);
        bool has_param(const std::string&);
    };
}
#endif   /* DEVS_UTILITY_HPP */


#include "devs_utility.hpp"

namespace devs_supplement {
    ulong entity::eid = 0; //Inicijalizacija brojaca identifikacionog broja entiteta
    entity::entity():id(eid++) {}
    entity::entity(const entity& rhs):id(eid++) { parameters = rhs.parameters; }
    void entity::set(const std::string& tag, parameter_value value)  {
         parameters[tag]=value;
    }
    bool entity::has_param(const std::string& tag) {
        if (parameters.find(tag) != parameters.end()) { return true; }
        else { return false; }
    }
}
```

```
/*
 * File:   devs_print.hpp
 * Author: Dušan Tadić PS110243
 *
 * Created on July 22, 2015, 2:09 PM
 */

#ifndef DEVS_PRINT_HPP
#define  DEVS_PRINT_HPP

#include "devs.hpp"
#include "devs_server.hpp"
#include "devs_queue.hpp"

namespace devs_supplement {
    class print_statistics {
          std::vector<devs_block::server*> servers;//Lista servera
          std::vector<devs_block::queue*> queues;  //Lista redova
        devs::time sim_time;
    public:
        print_statistics(devs::time st):sim_time(st) {}
        void add_server(devs_block::server* s) {    //Dodavanje servera na listu
            servers.push_back(s);
        }
        void add_queue(devs_block::queue* q) { //Dodavanje redova cekanja na listu
            queues.push_back(q);
        }
        void print() {
            for (auto q:queues) {
                q->stats.finalize(sim_time);
                std::cout << q->get_name() << std::endl;
                std::cout << "---------------------------------" << std::endl;
                std::cout << "Number of entries: " << q->stats.get_num() <<
                                                                        std::endl;
                std::cout << "Current number: " << q->stats.get_curr_num() <<
                                                                        std::endl;
                std::cout << "Maximum number: " << q->stats.get_max_num() <<
                                                                        std::endl;
              std::cout << "Mean waiting time: " << q->stats.get_mean_time() <<
                                                                        std::endl;
             std::cout << "Average queue length: " << q->stats.get_mean_num(sim_time) <<
                             std::endl;
                std::cout << "Zero entries: " << q->stats.get_nowait_num() << std::endl;
                std::cout << "Percent zeroes: " << q->stats.get_nowait_contrib() << std::endl;
                std::cout << "Mean waiting time without zero entries: " <<
                              q->stats.get_mean_time_nowait() << std::endl;
                std::cout << "---------------------------------" << std::endl;
            }
            for (auto s:servers) {
                s->stats.finalize(sim_time);
                std::cout << s->get_name() << std::endl;
                std::cout << "--------------------------------" << std::endl;
                std::cout << "Number of entries: " << s->stats.get_num() << std::endl;
                std::cout << "Server state: " << ((s->stats.get_curr_num()==1)?"busy":"idle")
    << std::endl;
                std::cout << "Mean service time: " << s->stats.get_mean_time() << std::endl;
                std::cout << "Utilization: " << s->stats.get_mean_num(sim_time) << std::endl;
                std::cout << "---------------------------------" << std::endl;
            }
        }
    };
}



#endif   /* DEVS_PRINT_HPP */
```

```
/*
 * File:   devs_random.hpp
 * Author: Dušan Tadić
 * Definicija klase rng i njenih naslednika
 * Created on July 20, 2015, 10:20 PM
 */
#ifndef DEVS_RANDOM_HPP
#define  DEVS_RANDOM_HPP
#include <random>
namespace devs_supplement {
    //Apstraktni rng
    class rng {
    protected:
        std::minstd_rand0 gen;
    public:
        rng(int);
        virtual double generate();
        virtual ~rng();
    };
    class exponential : public rng {
        std::exponential_distribution<double> e;
    public:
        exponential(int, double);
        void set(double);
        double generate();
        ~exponential();
    };
    class uniform : public rng {
        std::uniform_real_distribution<double> u;
    public:
        uniform(int);
        uniform(int, double, double);
        double generate();
        ~uniform();
    };
    class normal : public rng {
        std::normal_distribution<double> n;
    public:
        normal(int, double, double);
        double generate();
        ~normal();
    };
    class erlang : public rng {
        int red;
        std::exponential_distribution<double> e;
    public:
        erlang(int, double, int);
        double generate();
        void set(double, int);
        ~erlang();
    };
    class empirical : public uniform {
        std::vector<double> data;
    public:
        empirical(int);
        double generate();
        void set(std::vector<double>);
        ~empirical();
    };
}
#endif   /* DEVS_RANDOM_HPP */
```

```
#include "devs_random.hpp"

namespace devs_supplement {
    rng::rng(int seed) { gen.seed(seed); }
    double rng::generate() {}
    rng::~rng() {}
    exponential::exponential(int seed, double sv):rng(seed) {
        std::exponential_distribution<double>::param_type params(1.0/sv);
        e.param(params);
    }
    void exponential::set(double sv) {
        std::exponential_distribution<double>::param_type params(1.0/sv);
        e.param(params);
    }
    double exponential::generate() { return e(gen); }
    exponential::~exponential() {}
    uniform::uniform(int seed):rng(seed) {
        std::uniform_real_distribution<double>::param_type params(0.0, 1.0);
        u.param(params);
    }
    uniform::uniform(int seed, double lb, double ub):rng(seed) {
        std::uniform_real_distribution<double>::param_type params(lb, ub);
        u.param(params);
    }
    double uniform::generate() { return u(gen); }
    uniform::~uniform() {}
    normal::normal(int seed, double mean, double stddev):rng(seed) {
        std::normal_distribution<double>::param_type params(mean, stddev);
        n.param(params);
    }
    double normal::generate() { return n(gen); }
    normal::~normal() {}
    erlang::erlang(int seed, double sv, int r):rng(seed), red(r) {
        std::exponential_distribution<double>::param_type params((1.0/(sv/r)));
        e.param(params);
    }
    double erlang::generate() {
        double result = 0.0;
        for (int i = 0; i < red; ++i) {
            result += e(gen);
        }
        return result;
    }
    void erlang::set(double sv, int r) {
        std::exponential_distribution<double>::param_type params((1.0/(sv/r)));
        e.param(params);
    }
    erlang::~erlang() {}
    empirical::empirical(int seed):uniform(seed) {}
    double empirical::generate() {
        double value = uniform::generate();
        for (int i = 0; i < data.size(); ++i) {
            if (value < data[i]) {
                return i;
            }
        }
    }
    void empirical::set(std::vector<double> d) { data = d; }
    empirical::~empirical() {}
}
```

```
/*
 * File:   devs_statistics.hpp
 * Author: Dušan Tadić
 *
 * Statistika preuzeta iz diplomskog rada
 * Simulaciona analiza kvaliteta opsluživanja šalterske službe jedinice
 * poštanske mreže 11030 Beograd 8, Irena Samardžić, 2004.
 *
 * Created on July 20, 2015, 11:43 PM
 */
#ifndef DEVS_STATISTICS_HPP
#define  DEVS_STATISTICS_HPP
/*
 * Trajanje simulacije
 */
#define SIM_TIME    43200.0
//Intenzitet ulaznog toka u dve smene
#define IN_STREAM_AM 26.0
#define IN_STREAM_PM 29.0
//Kumulativna verovatnoća izbora usluge
#define S1  0.143    //Vrednosnice
#define S2  0.307    //Internet i telefonske dopune
#define S3  0.321    //Pakovanje i pecacenje paketa
#define S4  0.324    //Internet nalozi, telefonski brojevi
#define S5  0.325    //"Diners" usluge
#define S6  0.326    //"Western Union" usluge
#define S7  0.333    //Zamena deviza
#define S8  0.537    //Jedna uplata i uputnicke usluge
#define S9  0.715    //Grupne usluge
#define S10 0.764    //Transakcije sa TR bez provere stanja
#define S11 0.863    //Transakcije sa TR sa proverom stanja
#define S12 0.977    //Provera stanja
#define S13 1.0      //Stednja
//Parametri raspodela vremena opsluge
#define S1_LMBD     13.7
#define S2_LMBD     20.0
#define S3_LB       120.0
#define S3_UB       1080.0
#define S4_MEAN     204.7
#define S4_SDEV     32.3
#define S5_MEAN     204.7
#define S5_SDEV     32.3
#define S6_MEAN     204.7
#define S6_SDEV     32.3
#define S7_MEAN     204.7
#define S7_SDEV     32.3
#define S8_LMBD     63.1
#define S8_O        2
#define S9_LMBD     117.4
#define S9_O        2
#define S10_LMBD    64.6
#define S10_O       2
#define S11_LMBD    79.3
#define S11_O       2
#define S12_LMBD    24.0
#define S13_LMBD    278.6
#define S13_O       2

#endif   /* DEVS_STATISTICS_HPP */
```

```
/*
 * File:   main.cpp
 * Author: Dušan Tadić PS110243
 *
 * Definicija modela
 *
 * Created on July 19, 2015, 9:07 PM
 */

#include "devs.hpp"
#include "devs_utility.hpp"
#include "devs_random.hpp"
#include "devs_statistics.hpp"
#include "devs_generate.hpp"
#include "devs_queue.hpp"
#include "devs_server.hpp"
#include "devs_transducer.hpp"
#include "devs_assign.hpp"
#include "devs_input_switch.hpp"
#include "devs_print.hpp"
#include "devs_select.hpp"
#include "devs_buffer.hpp"

using namespace devs_supplement;

int main(int argc, char** argv) {
    uniform emp(1);
    empirical service_type(3);
    uniform unif(5, S3_LB, S3_UB);
    exponential expo1(7, S1_LMBD);
    exponential expo2(9, S2_LMBD);
    exponential expo3(11, S12_LMBD);
    erlang  erl1(13, S8_LMBD, S8_O);
    erlang  erl2(15, S9_LMBD, S9_O);
    erlang  erl3(17, S10_LMBD, S10_O);
    erlang  erl4(19, S11_LMBD, S11_O);
    erlang  erl5(21, S13_LMBD, S13_O);
    normal  norm1(23, S4_MEAN, S4_SDEV);
    normal  norm2(25, S5_MEAN, S5_SDEV);
    normal  norm3(27, S6_MEAN, S6_SDEV);
    normal  norm4(29, S7_MEAN, S7_SDEV);
    std::vector<rng*> rngs = {&expo1, &expo2, &unif, &norm1, &norm2, &norm3, &norm4,
                              &erl1, &erl2, &erl3, &erl4, &expo3, &erl5 };
    std::vector<double> s = {S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13};
    service_type.set(s);
    devs::simulator sim;
    /*
     * Definisanje blokova modela
     */
    //Generator klijenata
    devs_block::generate g_clients("clients", &sim);
    //Generator pauza u prepodnevnoj smeni za servere 0 - 3
    devs_block::generate g_pause_am_0123("pause_am_0123", &sim);
    //Generator pauza u poslepodnevnoj smeni za servere 0 - 3
    devs_block::generate g_pause_pm_0123("pause_pm_0123", &sim);
    //Generator pauza za server 4
    devs_block::generate g_pause_4("pause_4", &sim);
    //Generator pauza za server 5
    devs_block::generate g_pause_5("pause_5", &sim);
    //Generator pauza za server 6
    devs_block::generate g_pause_6("pause_6", &sim);
    //Generator odlazaka na blagajnu u obe smene za sve servere
    devs_block::generate g_treasury("treasury", &sim);
    //Generator popisa za servere 0 - 3
    devs_block::generate g_inventory_0123("inventory_0123", &sim);
    //Generator popisa za server 4
    devs_block::generate g_inventory_4("inventory_4", &sim);
    //Generator popisa za server 5
    devs_block::generate g_inventory_5("inventory_5", &sim);
    //Generator popisa za server 6
    devs_block::generate g_inventory_6("inventory_6", &sim);
    //Parametri pauza  na serverima 0 - 3
    devs_block::assign pparam_0123("pparam_0123");
    //Parametri pauza za server 4
    devs_block::assign pparam_4("pparam_4");
    //Parametri pauza za server 5
    devs_block::assign pparam_5("pparam_5");
    //Parametri pauza za server 6
```

```
devs_block::assign pparam_6("pparam_6");
//Parametri odlaska na blagajnu za sve servere
devs_block::assign tparam("tparam");
//Parametri popisa na serverima 0 - 3
devs_block::assign iparam_0123("iparam_0123");
//Parametri popisa na serveru 4
devs_block::assign iparam_4("iparam_4");
//Parametri popisa na serveru 5
devs_block::assign iparam_5("iparam_5");
//Parametri popisa na serveru 6
devs_block::assign iparam_6("iparam_6");
//Buffer za sistemske entitete
devs_block::buffer sys_buffer("sys_buffer");

//Dodeljivanje parametara vezanih za opslugu
devs_block::assign service_param0("service_param0");
//Dodeljivanje parametara vezanih za oplsugu klijenata
//koji posle usluge 6 biraju uslugu od 0 - 3
devs_block::assign service_param1("service_param1");
//Izbor dodatnih usluga u zavinosti od prve usluge
devs_block::assign back_to_start("back_to_start");

//Izbor putanje kroz sistem u zavinosti od izabrane usluge i vremena u toku simulacije
devs_block::input_switch choose_path("choose_path", 4);
//Izbor putanje sistemskih entiteta u zavinosti od segmenta u koji si posli
devs_block::input_switch system_segment("system_segment", 2);
//Izbor putanje entiteta u zavisnosti od toga da li su sistemski ili nisu
devs_block::input_switch sys_entity("sys_entity", 2);
//Izbor putanje entiteta u zavisnosti od parametara za dodatne usluge
devs_block::input_switch to_start("to_start", 2);
//Izbor putanje entiteta u zavisnosti od parametara za drugu uslugu posle usluge 6
devs_block::input_switch to_server5("to_server5", 2);

//Izbor kraceg reda
devs_block::select_queue select_queue("select_queue", 2);
//Sistemski segment 0
devs_block::queue queue_0("queue_0", &sim, 5);
devs_block::server server_0("server_0", &sim);
devs_block::server server_1("server_1", &sim);
devs_block::server server_2("server_2", &sim);
devs_block::server server_3("server_3", &sim);
devs_block::server server_4("server_4", &sim);


//Sistemski segment 1
devs_block::queue queue_1("queue_1", &sim, 2);
devs_block::server server_5("server_5", &sim);
devs_block::server server_6("server_6", &sim);
//Izlaz iz sistema
devs_block::transducer terminate("terminate");

/*
 * Izbor usluge
 */
std::function<void(entity*)> service_params0 = [&](entity* ent) {
    int service_num = service_type.generate();     //Generiši tip usluge
    if (ent->has_param("SERVICE_TYPE")) {          //Ako je ovo klijentu druga usluga
        int st = ent->get("SERVICE_TYPE");
        if (st == 0) {                             //Ako je prosla usluga bila 0
            while (service_num == st) {            //Izaberi bilo koju drugu
                service_num = service_type.generate();
                ent->set("HAD_0", 1);
            }
        }
        else {                                     //Ako je usluga bila 11
            double s = emp.generate();
            if (s <= 0.4) {                        //40% bira uslugu 10
                service_num = 10;
            }
            else {                                 //60% bira uslugu 9
                service_num = 9;
            }
        }
    }
    //Dodeljivanje vrste usluge i vremena opsluge
    ent->set("SERVICE_TYPE", service_num);
    ent->set("SERVICE_TIME", rngs[service_num]->generate());
    //Pretpostavka da klijent nije izabrao uslugu 6
```

```
    ent->set("2NDS_0123", 0);
    //U suprotnom
    if (service_num == 6) {                    //Ako je klijent izabrao uslugu 6
        //Onda odmah bira jos jednu razlicitu uslugu
        service_num = service_type.generate();
        while (service_num == 6) {
            service_num = service_type.generate();
        }
        if (service_num > 3) { //Ako se usluga pruza na istom salteru, opsluzuje se odmah
            ent->set("SERVICE_TIME", ent->get<devs::time>("SERVICE_TIME") +
 rngs[service_num]->generate());
        }
        else {                              //U suprotnom prosledjuje se na Server 5
            ent->set("2NDS_0123", 1);
            ent->set("2NDS", service_num);      //Upisati id druge usluge
        }
    }
};
/*
 * Postavljanje parametara za drugu uslugu(0, 1, 2, 3) posle usluge 6
 */
std::function<void(entity*)> service_params1 = [&](entity* ent) {
    ent->set("DEDICATED_SERVER", 0);
    if (ent->get("2NDS") < 2) {
        ent->set("PRIORITY", 2);
    }
    else {
        ent->set("PRIORITY", 1);
    }
    ent->set("SERVICE_TYPE", ent->get("2NDS"));
    ent->set("SERVICE_TIME", rngs[ent->get("2NDS")]->generate());
};
/*
 * Funkcija za izbor putanje u zavinsoti od izabrane usluge i vremena u toku simulacije
 */
std::function<void(entity*)> path_pick = [&](entity* ent) {
    int st = ent->get("SERVICE_TYPE");                          //Vrati tip usluge
    if (st < 4) {                                               //Ako je usluga od [0 - 3]
        if (ent->get<devs::time>("GENERATED_AT") <= 42600.0) { //A Server 5 jos uvek radi
            ent->set("PATH_PICK", 1);                           //Izaberi izlaz 3[Queue 0]
            ent->set("DEDICATED_SERVER", 0);     //Posveceni server je server 0[Server 5]
            if (st < 2) {                        //Ako je usluga 0 ili 1
                ent->set("PRIORITY", 2);         //Prioritet opsluge je 1
            }
            else {                          //U suprotnom prioritet je 2
                ent->set("PRIORITY", 1);
            }
        }
        else {                           //Ako je server zatvoren posalji ga na izlaz
            ent->set("PATH_PICK", 0);
        }
    }
    else if (3 < st && st < 7) {              //Ako je usluga od [4 - 6]
        ent->set("PATH_PICK", 3);             //Izaberi izlaz 1[Queue 1]
        ent->set("DEDICATED_SERVER", 4);      //Posveceni server je server 4[Server 4]
        ent->set("PRIORITY", 1);              //Prioritet opsluge je 1
    }
    else {
        ent->set("PATH_PICK", 2);             //U suprotnom izaberi izlaz 2[Select]
        ent->set("DEDICATED_SERVER", -1);     //Ne postoji posveceni server
        if (ent->has_param("HAD_0")) {
            if (ent->get("HAD_0") == 1) {
                ent->set("PATH_PICK", 1);
            }
        }
    }
};
/*
 * Funkcija za izbor putanje u zavisnosti od prethodno odabrane usluge
 */
std::function<void(entity*)> goto_start = [&](entity* ent) {
    static uniform returning(1);
    int st = ent->get("SERVICE_TYPE");        //Vrati tip usluge
    if (st == 0) {        //Posle usluge 0 klijent uvek potrazuje dodatne usluge
            ent->set("GOTO_START", 1);
    }
    else if (st == 11) {//U 50% slucajeva posle usluge 11 klijent potrazuje dodatne usluge
        if (returning.generate() < 0.50)
```

```
            ent->set("GOTO_START", 0);
        else
            ent->set("GOTO_START", 1);
    }
    else {
        ent->set("GOTO_START", 0);
    }
};
/*
 * Parametri pauza na serverima 0 - 3
 */
std::function<void(entity*)> pause0123 = [&](entity* ent) {
    static int counter;
    ent->set("SYS_SEGMENT", 0);
    ent->set("DEDICATED_SERVER", counter);
    ent->set("SERVICE_TIME", 1200.0);
    counter++;
    if (counter == 4) {
        counter = 0;
    }
};
/*
 * Parametri pauza  na serveru 4
 */
std::function<void(entity*)> pause4 = [&](entity* ent) {
    ent->set("SYS_SEGMENT", 0);
    ent->set("DEDICATED_SERVER", 4);
    ent->set("SERVICE_TIME", 1200.0);
//Entiteti izbegavaju poseban segment za dodatne usluge na serveru 4
    ent->set("2NDS_0123", 0);
};
/*
 * Parametri pauza na serveru 5
 */
std::function<void(entity*)> pause5 = [&](entity* ent) {
    ent->set("SYS_SEGMENT", 1);
    ent->set("DEDICATED_SERVER", 0);
    ent->set("SERVICE_TIME", 900.0);
};
/*
 * Parametri pauza na serveru 6
 */
std::function<void(entity*)> pause6 = [&](entity* ent) {
    ent->set("SYS_SEGMENT", 1);
    ent->set("DEDICATED_SERVER", 1);
    ent->set("SERVICE_TIME", 1200.0);
};


/*
 * Parametri odlaska na blagajnu za sve servere
 */
std::function<void(entity*)> treasury_all = [&](entity* ent) {
    static int counter = 0;
//Entiteti izbegavaju poseban segment za dodatne usluge na serveru 4
    ent->set("2NDS_0123", 0);
    if (counter == 7) { counter = 0; }  //Ima 7 odlazaka na blagajnu u svakoj smeni
    if (counter < 5) {     //Prva 4 odlaska na blagajnu se desavaju u segmentu nula
        ent->set("SYS_SEGMENT", 0);
        ent->set("DEDICATED_SERVER", counter);
        ent->set("SERVICE_TIME", 300.0);
    }
    else {                          //Druga dva u segmentu 1
        ent->set("SYS_SEGMENT", 1);
        ent->set("DEDICATED_SERVER", counter - 5);
        if (counter == 5) {
    //Odlasci na blagajnu na serveru 5 traju krace nego na ostalim serverima
            ent->set("SERVICE_TIME",200.0);            }
        else {
            ent->set("SERVICE_TIME", 300.0);
        }
    }
    counter++;
};
```

```
   /*
    * Parametri popisa na serverima 0 - 3
    */
   std::function<void(entity*)> inventory0123 = [&](entity* ent) {
       static int counter = 0;
       ent->set("SYS_SEGMENT", 0);
       ent->set("DEDICATED_SERVER", counter);
       ent->set("SERVICE_TIME", 600.0);
       counter++;
   };
   /*
    * Parametri popisa na serveru 4
    */
   std::function<void(entity*)> inventory4 = [&](entity* ent) {
       ent->set("SYS_SEGMENT", 0);
       ent->set("DEDICATED_SERVER", 4);
//Entiteti izbegavaju poseban segment za dodatne usluge na serveru 4
       ent->set("SERVICE_TIME", 600.0);
       ent->set("2NDS_0123", 0);
   };
   /*
    * Parametri popisa na serveru 5
    */
   std::function<void(entity*)> inventory5 = [&](entity* ent) {
       ent->set("SYS_SEGMENT", 1);
       ent->set("DEDICATED_SERVER", 0);
       ent->set("SERVICE_TIME", 600.0);
   };
   /*
    * Parametri popisa na serveru 6
    */
   std::function<void(entity*)> inventory6 = [&](entity* ent) {
       ent->set("SYS_SEGMENT", 1);
       ent->set("DEDICATED_SERVER", 1);
       ent->set("SERVICE_TIME", 600.0);
   };
   /*
    * Funkcije za odredjivanje trenutka generisanja novog entiteta
    */
   std::function<devs::time(devs::simulator*)> next_entity = [&](devs::simulator* s) {
       static bool pm_init = false;
       static exponential in_stream(7, IN_STREAM_AM);
       if (s->get_t() > 25200.0 && s->get_t() <= 42900.0 && !pm_init){ //Popodnevna smena
           in_stream.set(IN_STREAM_PM);
           pm_init = true;
       }
       if (s->get_t() > 42900.0) {  //Prekid generisanja u 5 minuta do kraja radnog vremena
           return devs::infinity;
       }
       return in_stream.generate();
   };
   /*
    * Funkcija za odredjivanje trenutka odlaska na blagajnu
    */
   std::function<devs::time(devs::simulator*)> goto_treasury = [&](devs::simulator* s) {
       static std::deque<devs::time> treasury_visits;
       static bool init = false;
       uniform uni(27, 3600.0, 21600.0);
       if (!init) {
   //Generisanje 7 trenutaka odlazaka na blagajnu po smenama
           for (int i = 0; i < 7; ++i) {
               double t = uni.generate();
               treasury_visits.push_back(t);
               treasury_visits.push_back(t + 19800);
           }
           /*
            * Generator koristi razliku u vremenu izmedju generisanja a
            * ne apsolutno vreme u toku simulacije
            */
           std::sort(treasury_visits.begin(), treasury_visits.end());
    //Sortiranje trenutaka generisanja sa rastucim poretkom
           for (int i = treasury_visits.size() - 1; i > 1; --i) {
       //Racunanje rastojanja izmedju trenutaka generisanja
               treasury_visits[i] -= treasury_visits[i - 1];
           }
           init = true;
       }
       if (!treasury_visits.empty()) {
```

```
        //Ako nisu generisani svi odlasci na blagajnu vrati vreme narednog odlaska
        double t = treasury_visits.front();
        treasury_visits.pop_front();
        return t;
    }
    else {
        return devs::infinity;
    }
};


/*
 * Postavljanje osobina pojedinacnih blokova
 */
g_clients.from_lambda_exp(next_entity);
g_clients.system_entity(false);
g_pause_am_0123.from_params(13500.0, 900.0, 4);
g_pause_am_0123.system_entity(true);
g_pause_pm_0123.from_params(33300.0, 900.0, 4);
g_pause_pm_0123.system_entity(true);
g_pause_4.from_params(13500.0, 19800.0, 2);
g_pause_4.system_entity(true);
g_pause_5.from_params(13500.0, 19800.0, 2);
g_pause_5.system_entity(true);
g_pause_6.from_params(14700.0, 19800.0, 2);
g_pause_6.system_entity(true);
g_treasury.from_lambda_exp(goto_treasury);
g_treasury.system_entity(true);
g_inventory_0123.from_params(22950.0, 300.0, 4);
g_inventory_0123.system_entity(true);
g_inventory_4.from_params(22800.0, 19800.0, 2);
g_inventory_4.system_entity(true);
g_inventory_5.from_params(22800.0, 19800.0, 2);
g_inventory_5.system_entity(true);
g_inventory_6.from_params(23100.0, 19800.0, 2);
g_inventory_6.system_entity(true);
pparam_0123.param_lambda(pause0123);
pparam_4.param_lambda(pause4);
pparam_5.param_lambda(pause5);
pparam_6.param_lambda(pause6);
tparam.param_lambda(treasury_all);
iparam_0123.param_lambda(inventory0123);
iparam_4.param_lambda(inventory4);
iparam_5.param_lambda(inventory5);
iparam_6.param_lambda(inventory6);
service_param0.param_lambda(service_params0);
service_param0.param_lambda(path_pick);
service_param1.param_lambda(service_params1);
back_to_start.param_lambda(goto_start);
server_4.closes_at(42600.0);
choose_path.switch_param("PATH_PICK");
to_server5.switch_param("2NDS_0123");
sys_entity.switch_param("SYS_FLAG");
to_start.switch_param("GOTO_START");
system_segment.switch_param("SYS_SEGMENT");
select_queue.set_weights(std::vector<int> {2, 5});
/*
 * Postavljanje prelaznog perioda i inicijalizacija objekta za stampanje statistike
 */
devs::statistics<int>::set_transition_time(3600.0);
print_statistics ps(SIM_TIME);
ps.add_server(&server_0);
ps.add_server(&server_1);
ps.add_server(&server_2);
ps.add_server(&server_3);
ps.add_server(&server_4);
ps.add_server(&server_5);
ps.add_server(&server_6);
ps.add_queue(&queue_0);
ps.add_queue(&queue_1);


/*
 * Povezivanje blokova
 */
devs::port::coupling(service_param0.get_in_port("?in"), g_clients.get_out_port("!out"));
devs::port::coupling(choose_path.get_in_port("?in"), service_param0.get_out_port("!out"));
devs::port::coupling(terminate.get_in_port("?in"), choose_path.get_out_port("!out0"));
```

```
devs::port::coupling(queue_1.get_in_port("?in"), choose_path.get_out_port("!out1"));
devs::port::coupling(select_queue.get_in_port("?in"), choose_path.get_out_port("!out2"));
devs::port::coupling(queue_0.get_in_port("?in"), choose_path.get_out_port("!out3"));
devs::port::coupling(queue_1.get_in_port("?in"), select_queue.get_out_port("!out0"));
devs::port::coupling(queue_0.get_in_port("?in"), select_queue.get_out_port("!out1"));
devs::port::coupling(select_queue.get_in_port("?count0"), queue_1.get_out_port("!count"));
devs::port::coupling(select_queue.get_in_port("?count1"), queue_0.get_out_port("!count"));
devs::port::coupling(queue_0.get_in_port("?pull0"), server_0.get_out_port("!out"));
devs::port::coupling(queue_0.get_in_port("?pull1"), server_1.get_out_port("!out"));
devs::port::coupling(queue_0.get_in_port("?pull2"), server_2.get_out_port("!out"));
devs::port::coupling(queue_0.get_in_port("?pull3"), server_3.get_out_port("!out"));
devs::port::coupling(queue_0.get_in_port("?pull4"), server_4.get_out_port("!pull"));
devs::port::coupling(server_0.get_in_port("?in"), queue_0.get_out_port("!out0"));
devs::port::coupling(server_1.get_in_port("?in"), queue_0.get_out_port("!out1"));
devs::port::coupling(server_2.get_in_port("?in"), queue_0.get_out_port("!out2"));
devs::port::coupling(server_3.get_in_port("?in"), queue_0.get_out_port("!out3"));
devs::port::coupling(server_4.get_in_port("?in"), queue_0.get_out_port("!out4"));
devs::port::coupling(queue_1.get_in_port("?pull0"), server_5.get_out_port("!out"));
devs::port::coupling(queue_1.get_in_port("?pull1"), server_6.get_out_port("!out"));
devs::port::coupling(server_5.get_in_port("?in"), queue_1.get_out_port("!out0"));
devs::port::coupling(server_6.get_in_port("?in"), queue_1.get_out_port("!out1"));
devs::port::coupling(sys_entity.get_in_port("?in"), server_0.get_out_port("!out"));
devs::port::coupling(sys_entity.get_in_port("?in"), server_1.get_out_port("!out"));
devs::port::coupling(sys_entity.get_in_port("?in"), server_2.get_out_port("!out"));
devs::port::coupling(sys_entity.get_in_port("?in"), server_3.get_out_port("!out"));
devs::port::coupling(sys_entity.get_in_port("?in"), server_5.get_out_port("!out"));
devs::port::coupling(sys_entity.get_in_port("?in"), server_6.get_out_port("!out"));
devs::port::coupling(to_server5.get_in_port("?in"), server_4.get_out_port("!out"));
devs::port::coupling(sys_entity.get_in_port("?in"), to_server5.get_out_port("!out0"));
devs::port::coupling(service_param1.get_in_port("?in"), to_server5.get_out_port("!out1"));
devs::port::coupling(queue_1.get_in_port("?in"), service_param1.get_out_port("!out"));
devs::port::coupling(back_to_start.get_in_port("?in"), sys_entity.get_out_port("!out0"));
devs::port::coupling(terminate.get_in_port("?in"), sys_entity.get_out_port("!out1"));
devs::port::coupling(to_start.get_in_port("?in"), back_to_start.get_out_port("!out"));
devs::port::coupling(terminate.get_in_port("?in"), to_start.get_out_port("!out0"));
devs::port::coupling(service_param0.get_in_port("?in"), to_start.get_out_port("!out1"));
//Segment za generisanje i inicijalizaciju sistemskih entiteta
devs::port bundle_out("!out");
devs::port bundle_pull("?pull");
devs::coupled_devs generator_bundle("generator_bundle");
generator_bundle.add_out_port(bundle_out);
generator_bundle.add_in_port(bundle_pull);
devs::port::coupling(pparam_0123.get_in_port("?in"), g_pause_am_0123.get_out_port("!out"));
devs::port::coupling(pparam_0123.get_in_port("?in"), g_pause_pm_0123.get_out_port("!out"));
devs::port::coupling(pparam_4.get_in_port("?in"), g_pause_4.get_out_port("!out"));
devs::port::coupling(pparam_5.get_in_port("?in"), g_pause_5.get_out_port("!out"));
devs::port::coupling(pparam_6.get_in_port("?in"), g_pause_6.get_out_port("!out"));
devs::port::coupling(tparam.get_in_port("?in"), g_treasury.get_out_port("!out"));
devs::port::coupling(iparam_0123.get_in_port("?in"), g_inventory_0123.get_out_port("!out"));
devs::port::coupling(iparam_4.get_in_port("?in"), g_inventory_4.get_out_port("!out"));
devs::port::coupling(iparam_5.get_in_port("?in"), g_inventory_5.get_out_port("!out"));
devs::port::coupling(iparam_6.get_in_port("?in"), g_inventory_6.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), pparam_0123.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), pparam_4.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), pparam_5.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), pparam_6.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), tparam.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), iparam_0123.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), iparam_4.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), iparam_5.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?in"), iparam_6.get_out_port("!out"));
devs::port::coupling(sys_buffer.get_in_port("?pull"),_bundle.get_in_port("?pull"));
devs::port::coupling(generator_bundle.get_out_port("!out"), sys_buffer.get_out_port("!out"));
devs::port::coupling(system_segment.get_in_port("?in"), generator_bundle.get_out_port("!out"));
devs::port::coupling(generator_bundle.get_in_port("?pull"), system_segment.get_out_port("!out0"));
devs::port::coupling(generator_bundle.get_in_port("?pull"), system_segment.get_out_port("!out1"));
devs::port::coupling(queue_0.get_in_port("?in"), system_segment.get_out_port("!out0"));
devs::port::coupling(queue_1.get_in_port("?in"), system_segment.get_out_port("!out1"));
generator_bundle.add_child(g_pause_am_0123);
generator_bundle.add_child(g_pause_pm_0123);
generator_bundle.add_child(g_pause_4);
generator_bundle.add_child(g_pause_5);
generator_bundle.add_child(g_pause_6);
generator_bundle.add_child(g_treasury);
generator_bundle.add_child(g_inventory_0123);
generator_bundle.add_child(g_inventory_4);
generator_bundle.add_child(g_inventory_5);
generator_bundle.add_child(g_inventory_6);
```

```
    generator_bundle.add_child(pparam_0123);
    generator_bundle.add_child(pparam_4);
    generator_bundle.add_child(pparam_5);
    generator_bundle.add_child(pparam_6);
    generator_bundle.add_child(tparam);
    generator_bundle.add_child(iparam_0123);
    generator_bundle.add_child(iparam_4);
    generator_bundle.add_child(iparam_5);
    generator_bundle.add_child(iparam_6);
    generator_bundle.add_child(sys_buffer);
    /*
     * Model sistema
     */
    devs::coupled_devs sys("model");
    sys.add_child(g_clients);
    sys.add_child(generator_bundle);
    sys.add_child(service_param0);
    sys.add_child(service_param1);
    sys.add_child(back_to_start);
    sys.add_child(choose_path);
    sys.add_child(system_segment);
    sys.add_child(sys_entity);
    sys.add_child(to_start);
    sys.add_child(to_server5);
    sys.add_child(select_queue);
    sys.add_child(queue_0);
    sys.add_child(queue_1);
    sys.add_child(server_0);
    sys.add_child(server_1);
    sys.add_child(server_2);
    sys.add_child(server_3);
    sys.add_child(server_4);
    sys.add_child(server_5);
    sys.add_child(server_6);
    sys.add_child(terminate);
    /*
     * Pocetak simulacije
     */
    sim.set_child(sys);
    sim(SIM_TIME);
    ps.print();
    return 0;
}
```

PRILOG 2. Rezultati dobijeni jednim izvršenjem programa

```
---------------------------------
server_0
---------------------------------
Number of entries: 268
Server state: idle
Mean service time: 88.5883
Utilization: 0.549575
---------------------------------
server_1
---------------------------------
Number of entries: 234
Server state: idle
Mean service time: 86.8801
Utilization: 0.470601
---------------------------------
server_2
---------------------------------
Number of entries: 199
Server state: idle
Mean service time: 85.2286
Utilization: 0.392604
---------------------------------
server_3
---------------------------------
Number of entries: 174
Server state: idle
Mean service time: 87.0985
Utilization: 0.350813
---------------------------------
server_4
---------------------------------
Number of entries: 105
Server state: idle
Mean service time: 101.451
Utilization: 0.246581
---------------------------------
server_5
---------------------------------
Number of entries: 644
Server state: idle
Mean service time: 54.0477
Utilization: 0.805711
---------------------------------
server_6
---------------------------------
Number of entries: 169
Server state: idle
Mean service time: 77.0507
Utilization: 0.301425
---------------------------------
```

```
---------------------------------
queue_0
---------------------------------
Number of entries: 981
Current number: 1
Maximum number: 9
Mean waiting time: 28.4584
Average queue length: 0.692577
Zero entries: 694
Percent zeroes: 70.7441
Mean waiting time without zero entries: 97.515
---------------------------------
queue_1
---------------------------------
Number of entries: 815
Current number: 2
Maximum number: 20
Mean waiting time: 175.612
Average queue length: 3.3375
Zero entries: 212
Percent zeroes: 26.0123
Mean waiting time without zero entries: 237.559
---------------------------------
```