ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU

VERIFIKACIJA KORISNIČKOG KOMUNIKACIONOG PROTOKOLA PRIMENOM eRM METODOLOGIJE

– Master rad –

Kandidat:

Stefanija Dačić 2012/3032

Mentor:

doc. dr Zoran Čiča

Beograd, Septembar 2015.

SADRŽAJ

1. UVOD

Elektronika je jedna od oblasti koja se veoma brzo razvija. Svakih par meseci na tržištu se pojavljuju novi čipovi. Zbog velike ekonomske isplativosti, u ovoj oblasti vlada velika konkurencija. Cilj proizvođača je da što pre na tržište izbace novi čip ne bi li se stekla prednost nad konkurencijom. Proces razvoja čipa čine dve važne faze: dizajn i verifikacija. Verifikacija čini i do 70% vremena za razvoj čipa, te se teži da se vreme posvećeno verifikaciji svede na minimum. To smanjenje se postiže tako što se ponovo koriste delovi verifikacionih okruženja koji su korišćeni na prethodnim projektima tj. ne razvijaju se od početka.

Ovaj rad opisuje razvoj verifikacione komponente (za funkcionalnu verifikaciju čipa) koja može da se više puta koristi u različitim projektima. Komponenta je napisana u *e* programskom jeziku korišćenjem programskog alata *Specman* kao kompajler i simulator. Verifikaciona komponenta u *e* jeziku ili kraće eVC (*e Verification Component*) simulira protokol između mrežnog i sloja linka podataka N2DL (*Network to Data Link*). U svakom projektu u kojem deo čipa ili ceo čip koristi ovaj protokol, može da se koristi N2DL eVC i time drastično smanji vreme razvoja samog verifikacionog okruženja.

Drugo i treće poglavlje prikazuju osnovne pristupe verifikaciji i eRM metodologije. U poglavlju 2 dat je opis tri osnovna pristupa funkcionalnoj verifikaciji koja se danas koriste. To su direktna, slučajna i softverska verifikacija. Dodatno, analizirane su dobre i loše strane navedenih pristupa. Poglavlje 3 daje pregled metodologije za višekratnu upotrebu eRM (*e Reusable Methodology*) koja u suštini predstavlja pristup slučajne verifikacije. Detaljno je objašnjeno koje sve delove standardna eVC komponenta mora da sadrži da bi bila u skladu sa eRM metodologijom i opisana je funkcija svakog dela. Dat je i prikaz kako se eVC komponente koriste i konfigurišu prilikom verifikacije čipa. Takođe je objašnjen način pisanja testova korišćenjem sekvenci, kao i virtuelna sekvenca koja se koristi kada imamo više eVC komponenti koje simuliraju različite protokole.

Poglavlje 4 sadrži specifikaciju samoga N2DL protokola. Dat je funkcionalan opis šta sve protokol treba da radi. Prikazana je i objašnjena blok šema svih strana koje sadrži protokol. Navedeni su svi signali koji sačinjavaju N2DL protokol kao i funkcija istih. Data je slika sa signalima na kojoj se vidi očekivana komunikacija između pomenutih strana protokola.

Poglavlje 5 govori o realizaciji i načinu korišćenja N2DL eVC komponente. Objašnjeno je koji programski alati se koriste pri simulacijama. Dat je opis svakog dela eVC komponente, kako statičkog tako i dinamičkog. Prikazana je arhitektura N2DL eVC komponente uz detaljan opis realizacije i funkcije njenih delova. U poglavlju 5.2. se govori o instanciranju eVC komponente, njenom konfigurisanju i povezivanju na signale jezika za opis hardvera HDL (*Hardware Description Language*), koji se kasnije mogu povezati na signale čipa. Poglavlje 5.3 opisuje test kojim se testira eVC komponenta. Dat je primer signaliziranja greške u čipu. Analizira se način i rezultati testiranja.

Na samom kraju dat je jedan globalni pregled ovog master rada sa fokusom na najbitnije detalje i zaključke iznesene u ranijim poglavljima.

NAPOMENA: Praktičan deo ovog rada je urađen u kompaniji *HDL Design House* iz Beograda koja se bavi projektovanjem i verifikacijom integrisanih kola. Kompletan kod projekta prikazanog u tezi priložen je u elektronskoj formi kao prilog uz tezu.

2. PROCES VERIFIKACIJE

Potrebe korisnika diktiraju trendove razvoja novih (proizvoda) sistema na tržištu. Sistem na čipu SoC (*System on chip*) [1], koji je po definiciji, ceo sistem smešten na jednom čipu, zbog svojih mogućnosti integracije, dimenzija i postizanja različitih funkcionalnosti, je veoma zastupljen. Cilj proizvođača je da što pre, na tržište, plasiraju novi čip. Kako bi se to postiglo, neophodno je da postupci u procesu razvoja čipa postanu brži, efikasniji i produktivniji. Dve osnovne faze u procesu razvoja čipa su: dizajn i verifikacija.

Kako raste kompleksnost i broj implementiranih funkcionalnosti SoC-a, tako se povećava i broj problema u postupku njegovog dizajna i verifikacije. Iskustvo ukazuje da se oko 70% vremena, utrošenog na celokupan razvoj čipa, utroši na utvrđivanje ispravnosti funkcionisanja dizajna, odnosno na verifikaciju. Ovakvi podaci ukazuju na potrebu da pristup verifikaciji bude produktivniji, ekonomičniji, a sam proces brži. Pri verifikovanju čipa u obzir se uzimaju ne samo njegova funkcionalnost, koja je definisana specifikacijom već i željene sistemske preformanse, kao što su parazitne kapacitivnosti i induktivnosti, proces izrade logičkog kola itd. Ovaj master rad se odnosi na funkcionalnu verifikaciju, koja podrazumeva samo funkcionalno testiranje integrisanog kola.

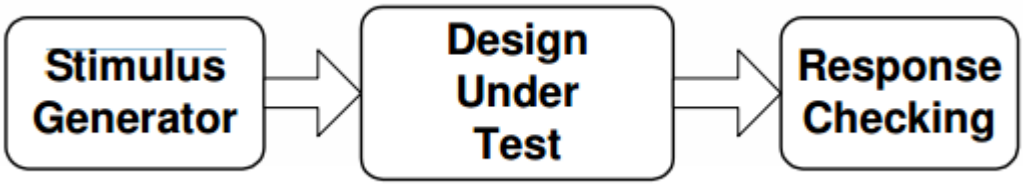
Suštinski u procesu verifikacije čipa potrebno je proizvesti ulazni stimulus za dizajn u skladu sa onim što je definisano specifikacijom i ispitati dizajn postavljanjem različitih scenarija. Ono što se postavlja kao dodatni zahtev jeste integracija različitih komponenti koje su deo čipa kao i njihovo kombinovanje, koje zbog kompleksnosti svakog od podsistema može predstavljati problem.

Postoji nekoliko pristupa za verifikaciju od kojih su najzastupljeniji [1]:

- direktna (tradicionalna) verifikacija
- slučajna (*random*) verifikacija
- softverska verifikacija

2.1. Direktna (tradicionalna) verifikacija

Direktna verifikacija odnosi se na pisanje direktnih testova sa unapred definisanim vrednostima signala. Verifikaciono okruzenje ili *testbench* ovog tipa može se predstaviti kao na slici 2.1.1.



Slika 2.1.1. Direktno verifikaciono okruženje [2]

Srednji blok sa slike DUT (*Device Under Testing*) predstavlja uređaj koji se testira. DUT može biti integrisano kolo specifične namene ASIC (*Aplication-Specific Integrated Curcit*), SoC ili neki od njegovih podmodula. Prvi blok, *Stimulus generator* ili generator pobude je generator ulaznih parametara, najčešće vrednosti ulaznih signala testiranog uređaja. Poslednji blok, *Response checking* proverava izlazne parametre iz DUT-a, odnosno vrednosti izlaznih signala u zavisnosti od ulaznih parametara. Ukoliko na izlazu ne dobijemo očekivane vrednosti, na osnovu ulazne pobude, to znači da u DUT-u postoji greška, odnosno *bug*.

Direktna verifikacija počinje detaljnim pisanjem test plana na osnovu specifikacije čipa koji se testira. Test plan se sastoji od nekoliko stotina direktnih testova i opisuje različite scenarije koje dizajneri i sistem arhitekte smatraju važnim za funkcionisanje čipa. Međutim, ovakav način pravljenja testova nosi sa sobom i velike nedostatke. Prvo, zbog velike kompleksnosti DUT-a neki važni scenariji mogu biti izostavlejni, jer ih se niko nije setio. Sa druge strane što je kompleksnost sistema veća, sve je teže napisati direktni test koji će pogoditi zadati cilj.

Svaki direktni test samo jednom pogađa određeni scenario. Međutim potrebno je ostvarivanje tih scenarija u različitim kombinacijama. To znači da je potrebno napisati testove koji pogađaju određenu oblast, a ne samo jednu „tačku“. Iako ovako napisani testovi mogu da pokriju određeni broj značajnih scenarija, kao i nepredviđene granične slučajeve odnosno corner cases, još uvek postoji velika verovatnoća da se veliki broj grešaka u DUT-u propusti. Razlog tome je to što se oblast koja se testira „gađa“ u svim pravcima. Da bi se ovaj problem prevazišao potrebno je napraviti test generator koji će lako moći da fokusira testove ka interesantnim oblastima.

Problem kod direktne verifikacije se javlja i kod provere odziva DUT-a. Ovaj problem najbolje je uočljiv pri verifikaciji SoC-a. Tada se ceo sistem posmatra kao celina, uzimajući pretpostavku da će se svaka greška, koja se desi u okviru DUT-a, manifestovati kroz neke sporedne efekte koji se mogu detektovati na izlazu tj. da se greška kroz sistem propagira. Osnovna mana ovog pristupa je pronalaženje izvora problema, radi njegovog popravljanja, jer oduzima mnogo vremena. To je ipak moguće rešiti integrisanjem *monitor*-a u *testbench*-u. *Monitor*-i bi bili priključeni na ključnim mestima unutar DUT-a i tako detektovali grešku na onim mestima na kojim su nastala [1].

2.2. Slučajna (*random*) verifikacija

Problemi koji se javljaju kod direktne verifikacije mogu se rešiti primenom slučajne verifikacije. Kao što joj i samo ime kaže, osnovni princip ove verifikacije je slučajnost, odnosno, nasumično generisanje određenih scenarija. Veoma je teško setiti se svakog mogućeg scenarija za testiranje DUT-a, pa se zato to ni ne pokušava. Ono što može da se preduzme je slučajno generisanje scenarija sa pretpostavkom da će posle dovoljnog broja tako generisanih scenarija kompletna funkcionalnost biti testirana.

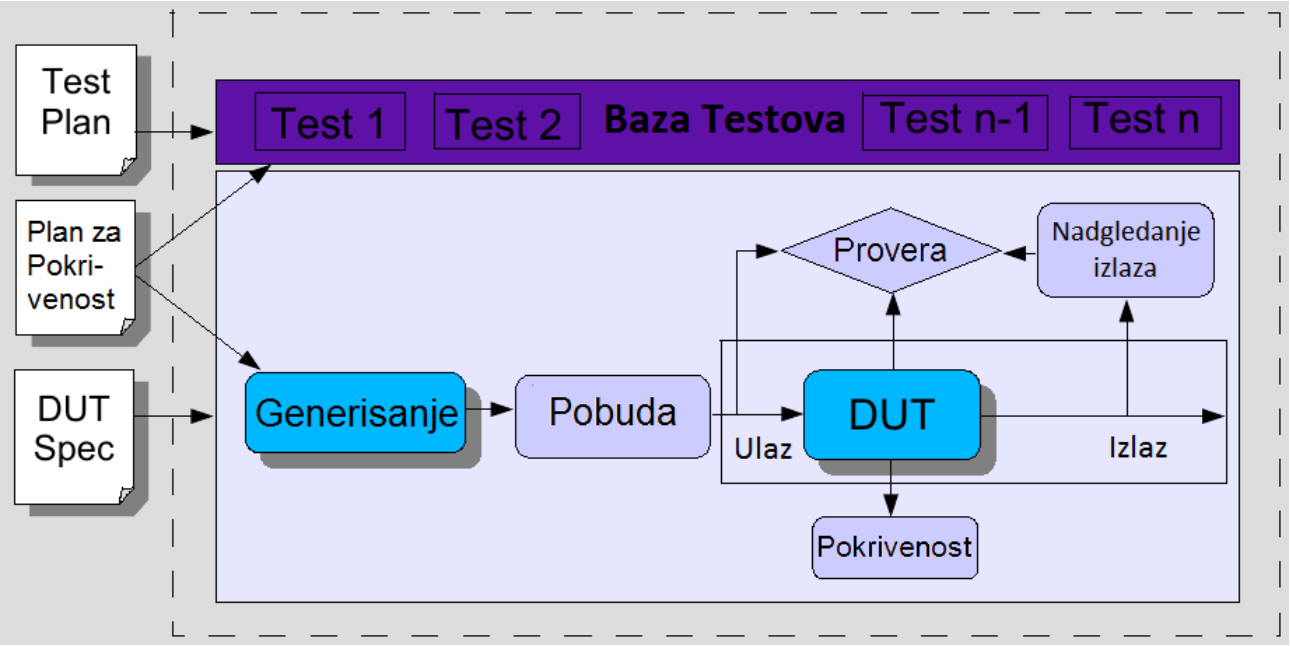
Potpuno slučajna verifikacija nema smisla jer se može desiti da zahteva više vremena i resursa od direktne verifikacije, npr. uvek se slučajno generiše jedan isti scenario. Da se tako nešto ne bi desilo neophodno je uvesti izvesna ograničenja pri njihovom generisanju.

C*onstraints* predstavljaju ograničenja, odnosno koriste se kako bi ograničila generisanje scenarija u određenu oblast ili oblasti koje su od interesa. Na primer, ako postoje dva ulazna pina u DUT i ako je prema protokolu kojim DUT komunicira logička vrednost ulaza „11“ nedozvoljena, onda će se pri generisanju ograničiti ulazni parametri, tako da mogu biti bilo koja vrednost izuzev „11“. Pored ograničenja vrednosti ulaznih parametara potrebno je vršiti proveru vrednosti kako

ulaznih, tako i izlaznih parametara, kako bi bili sigurni u ispravnost ponašanja uređaja koji se testira.

Coverage ili pokrivenost je takođe jedan od bitnih preduslova za slučajnu verifikaciju. *Coverage* procentualno predstavlja koliko je od funkcionalnosti definisanih specifikacijom obuhvaćeno slučajno generisanim scenarijima. Iako bi teoretski pokrivenost morala biti 100%, u praksi se prihvata 99%. Preostalih 1% je ostavljen za takozvane „rupe“ u *coverage*-u, *coverage holes* tj. delove funkcionalnosti koji ne mogu biti „pogođeni“ slučajnim generisanjem. Da bi se i ove funkcionalnosti pokrile, dodatno se pišu direktni testovi.

Verifikaciono okruženje slučajne verifikacije može se prikazati kao na slici 2.2.1:



Slika 2.2.1. Verifikaciono okruženje slučajne verifikacije [3]

Na osnovu specifikacije DUT-a pravi se test plan i *coverage plan*. Testovi određuju na koji način će generator generisati ulazne parametre, što se postiže pomoću *constrains*. Upravljač pobude prosleđuje te parametre kao ulazne signale testiranog uređaja. Izlazni signali se skupljaju i šalju na proveru. Tokom celog procesa *coverage* se nadzire, odnosno skuplja. Na taj način postoji uvid koliko od ukupne funkcionalnosti DUT-a je pokriveno verifikacijom.

Još jedna od velikih prednosti ovog pristupa je ta što testovi imaju autonomiju u odnosu na verifikaciono okruženje, što nije slučaj kod direktne verifikacije. Kod direktne verifikacije svaki test sa sobom je povlačio i određeno verifikaciono okruženje, što je ograničavalo mogućnost da se stari kod iskoristi u izmenjenom, odnosno, novom okruženju. Kod slučajne verifikacije cilj je razviti verifikaciono okruženje koje se lako može prilagoditi tj. koje je konfigurabilno tako da se može iznova koristiti. U tom slučaju veoma je jednostavno napisati veliki broj jednostavnih testova, koji sadrže relativno malo koda. Na ovaj način smanjuje se vreme pisanja testova, pa samim tim i vreme celog procesa verifikacije.

2.3. Softverska verifikacija

Softeverska verifikacija podrazumeva da se uređaj koji se testira, priključi na procesor koji u sebi sadrži softever za testiranje napisan u *C* ili nekom drugom asemblerskom programskom jeziku. Ovaj softver služi za testiranje uređaja. Pošto ovaj pristup nije od posebnog značaja za ovaj rad, neće mu biti posvećena veća pažnja.

Danas se u verifikaciji koristi nekoliko metodologija za razvoj okruženja. Fokus u ovom tekstu će biti na eRM (*e Reuse Methodology*). Pored nje u verifikaciji se koristi još i OVM (*Open Verification Methodology*) [4] i UVM (*Universal Verification Methodology*) [5].

3.E REUSE METODOLOGIJA

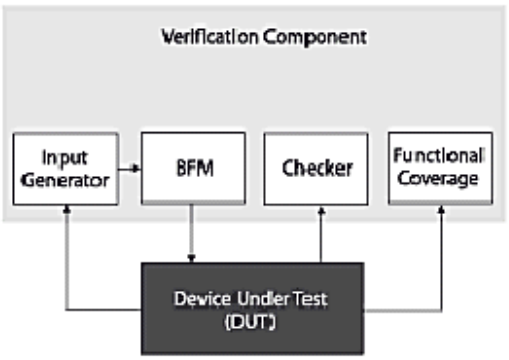
eRM (*e Reuse Methodology*) je prva metodologija koja je omogućila ponovno korišćenje postojećeg koda nekog projekta na nekom novom projektu, čime se drastično smanjilo vreme potrebno za razvoj verifikacionog okruženja. Ovakav pristup osmišljen je 2001. od strane *Verisity Design*, a realizovan je 2002. [6]. Koristi se *Verisity e* programski jezik [7] i *Specman tool* za kreiranje verifikacionih komponenti i test okruženja. *Specman Elite* obezbeđuje generisanje stimulusa i upravljanje funkcionalnim *coverage*, dok se *e* programski jezik koristi za kreiranje verifikacionog okružanja. Kod je organizovan u pakete, koji se mogu iznova koristiti. Paketi su nezavisni jedni od drugih, jednostavni su za upotrebu i lako se konfigurišu. Najsavršeniji oblik organizacije unutar ovih paketa predstavlja eVC (*e Verification Component*) komponenta.

3.1. e verifikaciona komponenta

eVC je verifikaciona komponenta napisana u *e* programskom jeziku. Ona predstavlja konfigurabilno verifikaciono okruženje i uglavnom je pisana za određen protokol ili arhitekturu kao što su AHB (*Advanced High-performance Bus*), APB (*Advanced Peripherial Bus*), AXI (*Advanced Extensible Interface*) ili USB (*Universal Serial Bus*).

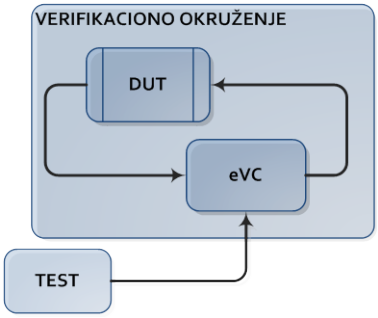
Kako je eVC komponenta već prvi put verifikovana, pri svakom sledećem korišćenju, rizik od greške je manji, a kvalitet dizajna raste. Još jedna prednost je u tome što eVC, kao komponenta opšte namene, može identifikovati granične slučajeve, uključujući i one koje dizajnerima hardvera mogu promaći. eVC su kompatibilni sa svim *Verilog* i VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) uređajima tj. svim uređajima koji su nastali korišćenjem relevantnih programskih jezika za opis hardvera.

Svaki eVC se sastoji od kompletnog skupa elemenata potrebnih za generisanje ulaznih kao i proveru izlaznih parametara i sakupljanje informacija o pokrivenosti za određeni protokol ili arhitekturu, kao što je prikazano na slici 3.1.1.



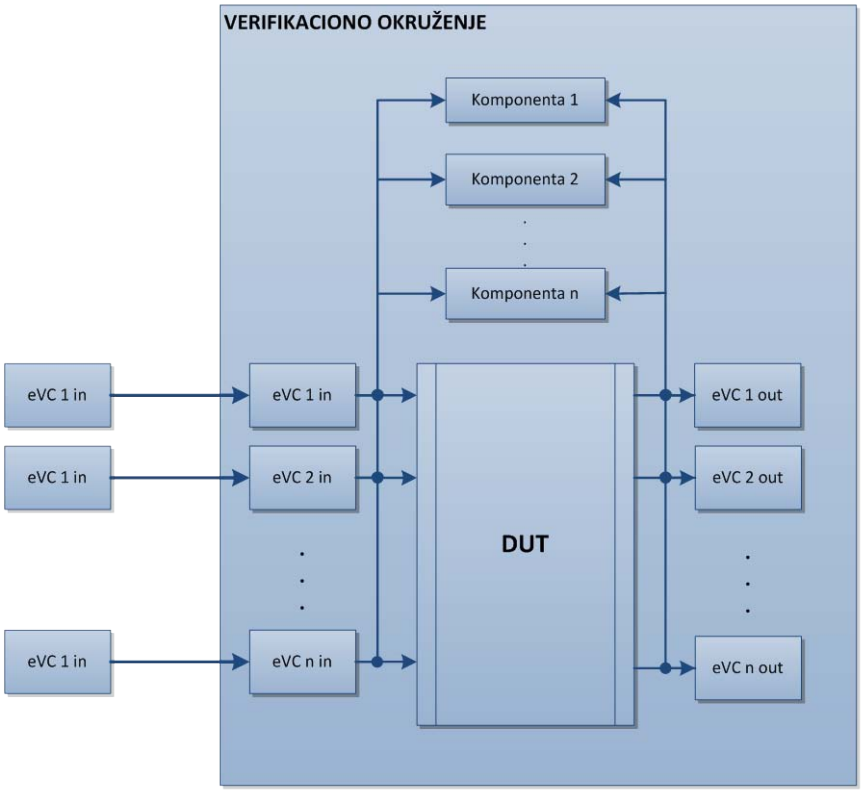
Slika 3.1.1. Prikaz verifikacione komponente [8]

eVC se koristi prilikom verifikacije DUT-a za protokol za koji je eVC predviđen. Na primer, u slučaju da DUT podržava AHB protokol, prilikom verifikacije koristi se AHB eVC. Komponenta generiše stimulus za DUT, u skladu sa protokolom AHB-a. Pomoću BFM (*Bus Functional Model*) šalje saobraćaj i povezuje se sa DUT-om. Pritom se konstantno nadzire, sakuplja i proverava odziv iz DUT-a kako bi se ustanovilo da li su rezultati očekivani, prema primenjenom AHB protokolu. Takođe se skuplja i *functional coverage* da bi se proverilo da li su svi predviđeni scenariji pokriveni bar jednom. Sam eVC može da se koristi kao kompletno verifikaciono okruženje (slika 3.1.2) ili kao deo nekog većeg verifikacionog okruženja (slika 3.1.3) u kome može da se nalazi nekoliko različitih ili istih eVC-a.



Slika 3.1.2. eVC kao kompletno verifikaciono okruženje

U složenim verifikacionim okruženjima pored eVC-a mogu se naći i određene komponente napisane u *e* jeziku kao što je prikazano na slici 3.1.3, koje su uglavnom specifične za određeni DUT.



Slika 3.1.3. eVC kao deo složenog verifikacionog okruženja

Te komponente nisu deo paketa, a nisu ni dovoljno kompleksne da bi bile eVC, pa je samim tim njihova ponovna upotreba, u drugim projektima, poprilično ograničena. Uglavnom se koriste pri verifikaciji novijih, unapređenih verzija istog DUT-a. Zbog svega navedenog, ove dodatne komponente neće biti dalje razmatrane u ovom master radu. Takođe, eVC može biti deo nekog drugog eVC-a. Primer je protokol za kontrolu prenosa podataka putem interneta TCP/IP (*Transmission Control Protocol/Internet Protocol*) eVC koji u sebi koristi *Ethernet* eVC.

Razlika između eVC-a i dobro napisanog modularnog verifikacionog okruženja je u tome da je eVC napravljen tako da može da se koristi u više različitih okruženja, sa više različitih konfiguracija. U idealnom slučaju eVC je komponenta tipa *Plug-and-play*. Ovo praktično znači da novo verifikaciono okruženje može da se napravi od različitih eVC komponenti koje inicijalno nisu bile planirane da rade zajedno. Postoji mogućnost i da se one povežu hijerarhijski, odnosno, da se od verifikacionog okruženja konstruisanog od više eVC-a, napravi jedan novi eVC koji može da se priključi i postane sastavni deo većeg i kompleksnijeg verifikacionog okruženja. Da bi se ovako nešto omogućilo, eVC, kao i verifikaciona okruženja koja potencijalno mogu biti pretvorena u eVC, treba napisati kao da su deo univerzalnog verifikacionog okruženja. Bez standarda svaki inženjer bi razvijao verifikacione komponente prema sopstvenoj logici, gde bi kao rezultat dobili komponente koje ne bi bile kompatibilne.

Ima nekoliko zahteva koje svaki eVC mora da ispunjava kako bi mogao da se ponovo koristi u različitim verifikacionim okruženjima [9].

1. Ne sme biti preklapanja između različitih eVC-a:
   - preklapanja pri nazivima komponenti;
   - ne sme postojati međusobna zavisnost pri rukovanju na istom modulu (DUT-u);
   - ne sme postojati međusobna zavisnost vezana za vremena;
   - ne sme postojati međusobna zavisnost vezana za globalna podešavanja.
2. Zajednički izgled eVC-eva, slična aktivacija, slična dokumentacija:
   - zajednički način da se instalira eVC;
   - zajednički način da se eVC popravi tj. zakrpi (*patch*);
   - zajednički način da se uoče greške u eVC-u;
   - zajednički način za rukovanje DUT greškama;
   - dobijanje eVC identifikacije;
   - zajednički način za programiranje interfejsa ka standardnim blokovima;
   - zajednički stil pisanja dokumentacije za eVC.
3. Podrška za kombinovanje više eVC-a:
   - zajednički način da se konfiguriše eVC;
   - zajednički način da se napišu testovi;
   - zajednički način da se naprave sekvence;
   - zajednički način da se radi provera;
   - zajednički način da se uradi podela protokola na slojeve (*layers*);
   - zajednički način za povezivanje pokrivenosti.

4. Podrška za modularno otklanjanje grešaka:
   - razumevanje kombinovanih ograničenja;
   - rekonstrukcija ponašanja jednog eVC-a u verifikacionom okruženju.
5. Istovetnost u implementaciji:
   - zajednička struktura podataka;
   - zajednička metodologija prilikom testiranja eVC-a;
   - zajednički način za korišćenje portova i paketa.

Pored nabrojanih postoje i drugi zahtevi koji nisu toliko interesantni za ovaj rad pa samim tim nisu ni navedeni.

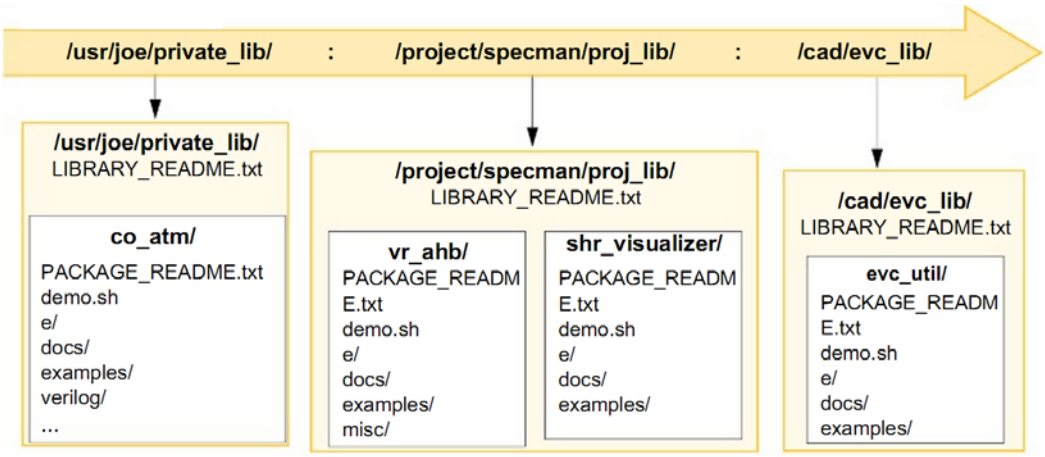
3.2. Paketi verifikacionih komponenti

Paket predstavlja strukturu direktorijuma, koji sadrži sav relevantan kod potreban za verifikovanje neke vrste komponente. Na ovaj način, odnosno „pakovanjem“ omogućava da se verifikaciono okruženje ponovo koristi. Postoji konvencija, prema kojoj se vrši imenovanje paketa, tako da se lako može utvrditi ime i verzija verifikacionog okruženja. Pored ovog, postoje i drugi standardi, kao što su precedure za instalaciju.

Svaka verifikaciona komponenta, koja će se ponovo koristiti, treba da bude spakovana u poseban direktorijum. Direktorijum se pravi posebno za svaku novu verziju komponente i sadrži kod verifikacione komponente koji je napisan u *e* programskom jeziku. Takođe, potrebno je da ima standardnu strukturu i *PACKAGE_README.txt* u standardnom formatu. Ovako zapakovani direktorijumi predstavljaju osnovnu jedinicu za isporuku verifikacionih komponenti.

Dva paketa sa istim imenom ne mogu se uneti u kompajler ili simulator, jer će javiti grešku. Preporuka je da se ime paketa formira tako što će se za prefiks imena koristiti ime kompanije, dok ostatak predstavlja ime proizvoda (npr: *amd_usb* (AMD USB eVC)).

Pored *e* fajlova paket sadrži i dokumentaciju, kao i primere kako se verifikaciona komponenta može koristiti. Paketi se instaliraju i smeštaju u biblioteke, *libraries*. Na slici 3.2.1 je dat primer nekoliko biblioteka.

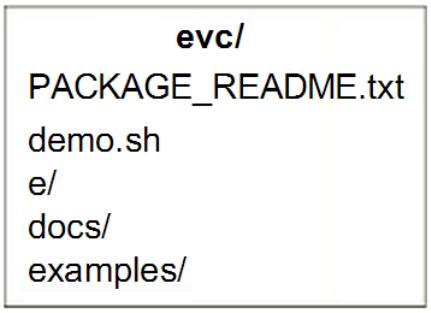


Slika 3.2.1. Primeri biblioteka [9]

Kao što se vidi, biblioteka je direktorijum koji sadrži podirektorijume, odnosno pakete koji opet imaju svoju strukturu direktorijuma i obavezno *LIBRARY_README.txt* fajl.

3.2.1. eVC paketi

Na slici 3.2.1.1 prikazana je struktura direktorijuma eVC paketa.



Slika 3.2.1.1 Prikaz strukture eVC paketa [9]

Kao što je već napomenuto, svaki direktorijum mora da sadrži *PACAKAGE_README.txt* fajl. On sadrži sve bitne informacije, kao što su ime paketa, verzija, autor, kompanija, promene u odnosu na prethodnu verziju itd.

Pored njega u okviru eVC paketa su:

1. *demo.sh*, skripta kojom se pokreće demonstraciona simulacija za eVC. U sklopu paketa je i nekoliko primera, *examples*, koje bi ova skripta trebala da pokrene.
2. *e direktorijum*, koji sadrži kompletan kod koji čini eVC. Tu se prevashodno misli na komponente eVC-a, koje su organizovane u poddirektorijumima u okviru paketa.
3. *docs direktorijum*, sadrži svu dokumentaciju vezanu za eVC (specifikacija, upustvo za korisnika...)
4. *examples direktorijum*, sadrži primere kako se koristi eVC (instancira, konfiguriše, povezuje sa DUT-om). Tu se nalaze i primeri osnovnih testova kao i primeri sekvenci.

Ova četiri direktorijuma su osnova, ali je pored njih moguće imati još neke dodatne. Kao primer može se uzeti RTL direktorijum, koji sadrži kod napisan u VHDL ili Verilog programskom jeziku.

3.3. Arhitektura eVC-a

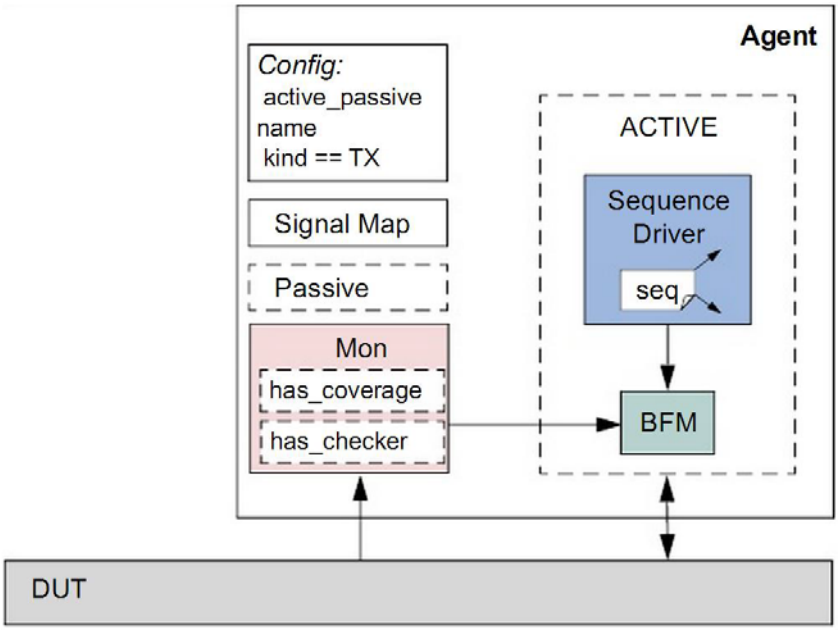
Arhitekura i dizajn eVC-a u mnogome zavisi od protokola na koji se odnosi. Ipak, bez obzira na to postoji određena istovetnost koja karakteriše svaki eVC.

Najbitniji deo svakog eVC-a je *agent*. *Agent*-i emuliraju ponašanje većih uređaja i imaju specifikacijom definisane funkcionalnosti i sve osnovne komponente. Pored toga postoje i konfiguraciona polja, koja omogućavaju dodatna podešavanja. Ona su deo standarda i sadrži ih svaki eVC.

Agent može biti aktivan (*active*) ili pasivan (*passive*). Aktivni ili TX (*transmit*) *agent*-i šalju stimulus DUT-u, na njegov ulazni port, tj. upravljaju (*drive*) određenim vrednostima ulaznih signala

za DUT. Dok, pasivni ili RX (*receive*) *agent*-i sakupljaju, odnosno nadziru saobraćaj koji šalje DUT. Osim nadziranja, pasivni *agent*-i mogu da sadrže i kontrolore (*checkers*), proveravaju pokrivenost (*coverage*) i "semafore" (*scoreboard*).

Na slici 3.3.1 prikazan je eVC sa dva *agent*-a, jednim pasivnim i drugim aktivnim povezanim na DUT.



Slika 3.3.1. Tipična striktuura eVC *agent*-a [9]

Svi *agent*-i imaju istu osnovnu arhitekturu sa dodacima koji su opcioni i zavise od specifičnosti verifikacionog okruženja. Komponente koje čine standardnu konfiguraciju su:

- *configuration* (konfiguracione jedinice) – grupa polja za konfigurisanje atributa *agent*-a i njegovo ponašanje
- *signal map* (mapa signala) – signali *hardware*, kojima *agent* mora imati pristup da bi mogao da komunicira sa DUT-om.
- *Bus Functional Model* (BFM) – komponenta koja komunicira sa DUT-om i ujedno šalje (*drive*) i monitoriše (*sample*) signale DUT-a. Na slici to je predstavljeno bidirekcionom strelicom
- *sequence driver* (upravljač sekvenci) – komponenta koja se povezuje na BFM i prosleđuje mu transakcije kada su dostupne.
- *monitor* – komponenta koja pasivno monitoriše signale DUT-a i te informacije prenosi ostalim komponentama *agent*-a. Monitor ne može da šalje saobraćaj DUT-u.

Sve navedene komponente su detaljnije objašnjene u nastavku.

Svaki *agent* u eVC-u može biti konfigurisan kao pasivan. Pasivni *agent*-i, kao što je već rečeno skupljaju informacije od DUT-a i čuvaju ih kao reference verifikacionom okruženju. Ovakav pristup omogućava uređeno monitorisanje DUT-a, sakupljanje informacija, kao i proveru tih informacija, odnosno da li se DUT ponaša u skladu sa onim što je predviđeno. Najbitnija komponenta pasivnog *agent*-a je *monitor*, tj. s obzirom da ne šalje nikakav saobraćaj komponente kao što su BFM i *sequence driver*, su nepotrebne. *Agent*-e treba tako dizajnirati da su, koliko je god

to moguće, nezavisni od specifičnosti okruženja u kome se nalaze, jer se takvi mogu lakše prilagođavati zahtevima koje korisnik eVC-a može da ima. Zbog toga su *sigmal map*-a i konfiguracija veoma bitan deo *agent*-a, pri čemu se njihova podešavanja vrše iz okružanja u kojem je taj *agent* instanciran. Moguće je i da *agent* ima pokazivač na okruženje u kome je instanciran, kroz koji će prikupiti sve potrebne informacije u vezi mape signala i konfiguracije.

3.3.1. Configuration (konfiguracija) i signal map (mapa signala)

Configuration, konfiguraciona jedninica određuje *agent*-ov interfejs sa ostatkom verifikacionog okruženja. Ona omogućava konfiguraciju *agent*-a, tj uspostavljanje određenih modova u kojima će raditi (na primer: da li će *agent* biti aktivan ili pasivan, da li će skupljanje *coverage*, biti uključeno ili isključeno) kao i druge informacije od značaja za rad *agent*-a (na primer: širina magistrale).

Signali koji povezuju eVC na DUT su tipa tekstualnog niza (*string*). Oni su definisani ili u *agent*-u ili u *signal map*-i koja se nalazi u *agent*-u. Pošto se signali prosleđuju ostalim komponentama u *agent*-u kao što su *monitor* ili BFM, postojanjem *signal map*-e kao zasebne celine unutar *agent*-a prosleđivanje se vrši jednostavnije.

3.3.2. Sequence driver (upravljač sekvenci)

Sekvence (*sequences*) su osnovni interfejs kojom inženjer koji piše testove može da upravlja verifikacionim procesom. Sva specifična upravljanja testa, što uključuje i *reset* i ubacivanje grešaka, trebalo bi da su dostupna preko sekvenci.

Sequence driver je jedinica instancirana isključivo u aktivnom *agent*-u tj. ne postoji u pasivnom. Podrazumeva se da je ova jedinica povezana na BFM. Elementi sekvenci (*items*) koji se generišu u *sequence driver* šalju se ka BFM-u svaki put kada su elementi sekvenci dostupni za slanje i kada je BFM spreman da te elemente procesuira. Sa druge strane, BFM i *monitor* stalno prosleđuju *sequence driver*-u informaciju o tome u kojem se stanju DUT nalazi, na osnovu koje *sequence driver* generiše elemente sekvenci.

3.3.3. Monitor

Monitor je zadužen za izvlačenje informacija iz signala DUT-a i prevođenje istih u smislene događaje kao i u statusne informacije. Tako definisane informacije su zatim dostupne svim ostalim komponentama *agent*-a kao i inženjeru koji piše testove. *Monitor* nikada ne sme da se oslanja na informacije dobijene od strane ostalih komponenti, kao što je BFM. Jedini izvor koji *monitor* treba da ima su signali DUT-a.

Monitor je jedinica koja je uvek instancirana (bilo da je *agent* aktivan ili pasivan). Njegova funkcionalnost zasniva se na nadgledanju signala, što je osnovni zahtev. Sve druge dodatne funkcionalnosti trebalo bi implementirati odvojeno, kao nadogradnju *monitor*-a. Ovo se odnosi na protokol kontrolore (*checkers*), "semafore" (*scoreboard*) i pokrivenost (*coverage*), koji se obično koriste u pasivnom *agent*-u.

Kako će događaji biti organizovani od strane *monitor*-a, zavisi najviše od implementiranog protokola. Za osnovne podatke, *monitor* registruje događaje tipa *item_started event* i *item_ended*

event, na primer kada je paket počeo da se šalje i kada je završio. *Monitor* takođe skuplja i podatke koji se odnosne na trenutno stanje signala i pretvara to u *current_item event*. *Monitor* skuplja sa signala elemente sekvenci i rekonstruiše ih (radi suprotno od BFM-a). Tako rekonstruisani elementi mogu biti korišćeni od strane drugih komponenti *agent*-a. Osim podataka u vidi elemenata sekvenci, *monitor* sakuplja i relevantna vremena, kao što je vreme trajanja transakcije.

Kao što je već pomenuto, *checkers* i *coverage*, prave se kao nadogradnja *monitor*-a i njih korisnik može da uključi i isključi podešavanjem polja *has_checker* i *has_coverage*.

3.3.4. Bus Functional Model

BFM služi za upravljanje svih signala iz *agent*-a ka DUT-u. To je jedinica instancirana samo u aktivnom *agent*-u. Na ovaj način pri prebacivanju aktivnog *agent*-a u pasivni, sprečavamo *agent*-a da šalje bilo kakve signale. U BFM-u se ne radi bilo kakvo generisanje elemenata sekvenci, već ih dobija od *driver*-a. Nad njima se vrše sve potrebne operacije, ne bi li se prosledili ka DUT-u, u skladu sa protokolom. Element sekvence u sebi treba da sadrži sve potrebne informacije da bi se transakcija uspešno završila.

Da bi svoj zadatak sproveo korektno, BFM mora da zna trenutno stanje DUT-a. Ovu informaciju može da dobije direktnim posmatranjem signala ili korišćenjem informacija dobijenih iz *monitor*-a.

BFM nikada ne bi trebao da javlja bilo kakve greške vezane za protokol. To je posao *checker*-a, koji je deo nadogradnje *monitor*-a.

3.3.5. Checkers (protokol kontrolori)

Checker je jedinica koja se uglavnom koristi za verifikovanje DUT-a, mada se može koristiti i za proveru ispravnosti aktivnog eVC *agent*-a. On može biti implementiran kao zasebna jedinica u *agent*-u ili kao nadogradnja *monitor*-a. Prema konvenciji, svi kontrolori u pasivnom modu su uključeni dok su u aktivnom modu isključeni.

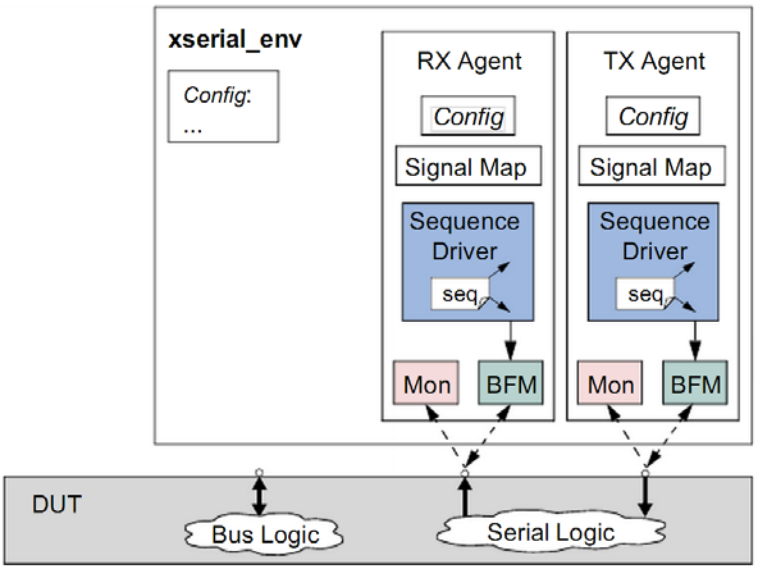
Checker funkcioniše na osnovu događaja i podataka skupljenih od strane monitora. Ukoliko je realizovan kao zasebna jedinica onda u sebi poseduje pokazivač na *monitor* koji je podešen od strane *agent*-a pri instanciranju *checker*-a.

Checker minimalno treba da proverava da li je prilikom slanja ili primanja podataka prekršeno bilo koje pravilo protokola. To uključuje i vremena uključena u transakciju.

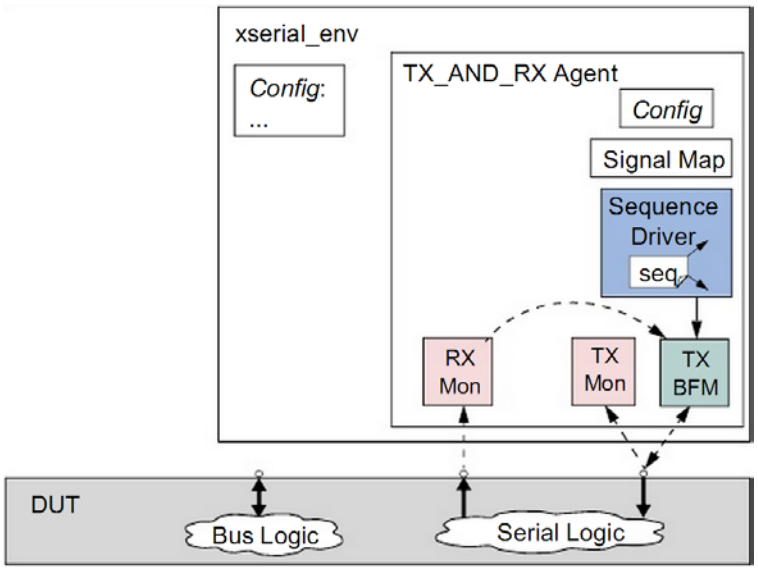
3.3.6. Coverage (pokrivenost)

Coverage se koristi da bi se verifikovao DUT, mada se može koristiti i za proveru ispravnosti aktivnog eVC *agent*-a, tako što bi se proverilo da li su pogođeni svi tipovi *sequences*. *Coverage* može biti implementrian na dva načina, ili kao posebna jedinica ili kao nadogradnja *monitor*-a. Prema konvenciji, sakupljanje *coverage* se radi u pasivnom modu, odnosno tada je uključeno *has_coverage ==TRUE*. Dok je u aktivnom modu isključeno, *has_coverage==FALSE*. *Coverage* funkcioniše na osnovu događaja i podataka sakupljenih od strane *monitor*-a. U slučaju da je realizovan kao zasebna jedinica, sadrži pokazivač na *monitor*, koji je podešen od strane *agent*-a pri instanciranju pokrivenosti.

Na slikama 3.3.6.1 i 3.3.6.2 se nalazi primer jednog eVC-a napravljenog za *XSerial* protokol.



Slika 3.3.6.1. Primer *XSerial_env* protokola sa dva agenta RX i TX [9]



Slika 3.3.6.2. Primer *XSerial_env* protokola sa jednim agentom RX_TX [9]

Kao što se sa slika vidi, DUT koji se testira ima dva interfejsa, *bus* i *serial*. Na svaki od ovih interfejsa moguće je povezati eVC, pomoću koga se ovi interfejsi testiraju. Serijski interfejs ima dva porta, RX (*receive*) i TX (*transmit*) port. Kao što se na slikama vidi, za ovaj interfejs zakačen je *xserial_env*. Na svakoj od slika prikazana je jedna od mogućih realizacija eVC-a. Na slici 3.3.6.1, eVC ima dva *agent*-a, odnosno po jedan za svaki od portova interfejsa, dok su na slici 3.3.6.2 ta dva *agent*-a realizovana kroz jedan. Ovi *agent*-i mogu da emuliraju ponašanje realnog uređaja. *Config* predstavlja grupu polja koja omogućavaju konfigurisanje *agenat*-a i kontrolu njihovog ponašanja.

Obe implementacije su validne. Koja će biti primenjena, zavisi od toga šta definiše protokol za koji je eVC napravljen, odnosno od njegovog nivoa kompleksnosti. U slučaju da je RX strana definisana kao pasivna, implementacija posebnog *agent*-a na RX portu nije potrebna. Tada nije potrebno slati bilo kakav saobraćaj, na RX port, tako da *driver* i BFM nisu potrebni. Međutim, implementacija posebnog *agent*-a na RX strani ima smisla u slučaju da je protokolom definisan

mehanizam komunikacije između RX i TX strane. To znači da je neophodno da TX prima neke pakete od strane RX, da bi se obezbedilo njegovo pravilno funkcionisanje, odnosno definisani mehanizam suviše je kompleksan i mora se realizovati kroz poseban *agent*.

3.4. Sekvence i konstruisanje test scenarija

Sequence (sekvence) omogućavaju slanje niza podataka ka DUT-u (ili niza radnji koje treba da se odrade na DUT interfejsu). Takođe, sekvence mogu da se koriste i za generisanje statičke liste podataka koja nikako nije povezana sa DUT interfejsom. Da bi malo bolje razumeli *sequence* potrebno je objasniti terminologiju tri glavna entiteta vezana za ovo poglavlje. Ti entiteti su i ranije bili pominjani u ovom rad, ali će sada biti objašnjeni u malo drugačijem svetlu.

Prvi entitet – *item* ili element sekvence – struktura koja predstavlja glavni ulaz za DUT (kao na primer: paket, transakcija, instrukcija). Tipično elementi sekvence već postoje u verifikacionom okruženju (najčešće u vidu protokol transakcija) i samo njihove male modifikacije su dovoljne da bi se koristili u sekvencama.

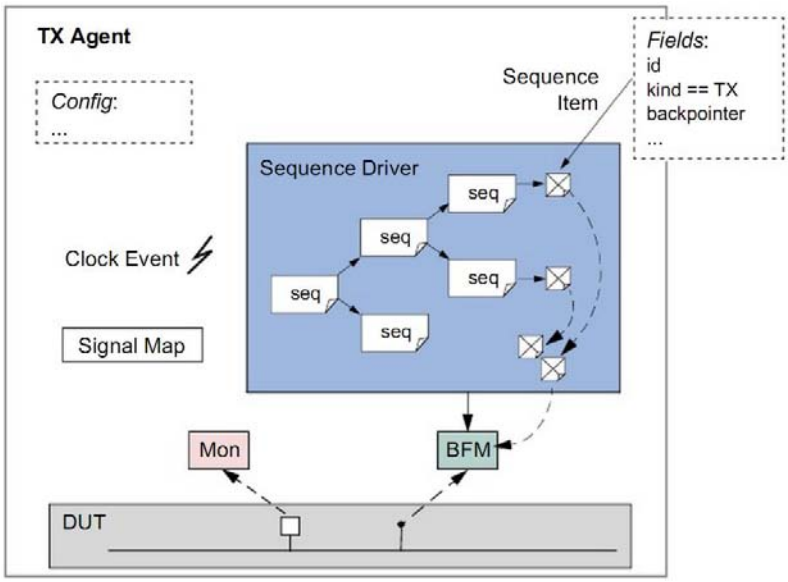
Drugi entitet – *sequence* ili sekvenca – struktura koja predstavlja niz elemenata sekvence pa samim tim označava scenarije višeg sloja apstrakcije. Realizovana je tako što se elementi generišu jedan za drugim u skladu sa nekim specificiranim pravilima. Sama struktura *sequence* u sebi sadrži predefinisana polja i metode koji neće biti detaljno razmatrani u ovome radu. Za više informacija pogledati [9]. Sekvence takođe mogu biti nadograđene od strane korisnika.

Treći entitet – *sequence driver* ili upravljač sekvenci – jedinica koja služi kao posrednik između sekvenci i verifikacionog okruženja. Generisani elementi sekvenci se prosleđuju preko sekvenci do upravljača sekvenci koji reaguje na svaki od elemenata ponaosob i tipično ih prosleđuje ka BFM-u. Ukoliko sekvence ne treba da idu ka DUT-u, upravljač ih smešta u listu. U svrhu slanja podataka na DUT signale, *driver* komunicira isključivo sa BFM-om.

Sequence driver i BFM rade u paru. Dok prvi služi sekvencama kao interfejs naviše tako da one stalno mogu da vide šta se dešava na signalima DUT-a, BFM služi kao interfejs naniže prema DUT-u, ostvarujući tako slobodu da se sekvence napišu na koji god način korisnik poželi.

Isprva deluje kao da je nepotrebno imati odvojen upravljač sekvenci i BFM. Važnost ovoga odvajanja komponenti se ogleda pri uvođenju virtuelnih sekvenci (o kojima će biti reči u poglavlju 3.4.1.).

Na slici 3.4.1 se nalazi šematski primer sekvenci. *Items* (na slici kvadratići sa oznakom X) kroz same sekvence i upravljač sekvenci bivaju prosleđene BFM-u koji ih tumači i u skladu sa protokolom komunicira sa DUT-om. Uočava se da jedna sekvenca može u sebi imati više drugih sekvenci. Te sekvence ne komuniciraju sa DUT-om direktno. Takođe, ista sekvenca može biti podsekvenca nekoliko različitih sekvenci. Sve ovo daje veliku slobodu prilikom pisanja test scenarija.



Slika 3.4.1. Šematski prikaz i njihovo prosleđivanje BFM-u [9]

Jasno je da je za konstruisanje raznih test scenarija potrebno imati napisan određen broj sekvenci. Sve sekvence se uglavnom smeštaju u jedan fajl koji se zove biblioteka sekvenci ili *sequence library*.

Koraci potrebni za konstruisanje test scenarija, odnosno testova su:

- definisati strukturu elementa sekvence,
- definisati *sequences* i *driver*,
- povezati *driver* sa okruženjem (ovde se misli na okruženje unutar eVC-a tj. *agent*-a),
- kreirati biblioteku *sequences* koja će da sadrži razne scenarije konstruisane od *sequences*,
- napisati testove na osnovu *sequence libraries*.

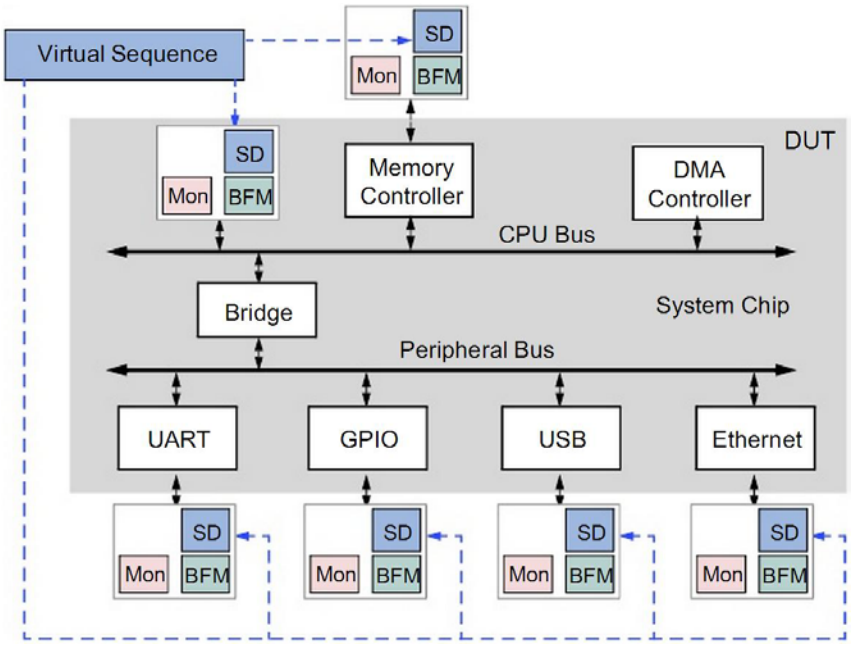
3.4.1. Virtuelne sekvence

Obične sekvence su tesno povezane sa svojim tipom (sekvence) i elementima sekvence. To znači da one ne mogu da rade sa sekvencama drugih tipova (na primer USB sekvenca ne može da bude procesuirana od strane *Ethernet* sekvence pošto su USB BFM i *Ethernet* BFM drugačije implementirani, svaki za svoj protokol). Kod složenijih verifikacionih okruženja (koja sadrže nekoliko eVC-a) potrebno je imati mehanizam koji bi omogućio pravljenje sekvenci (samim tim i test scenarija) koje u sebi sadrže sekvence iz nekoliko različitih eVC-a (protokola). Taj mehanizam je virtuelna sekvenca i virtuelni upravljač sekvenci.

Virtuelne sekvence, za razliku od običnih, nisu tesno povezane sa specifičnim sekvencama ili elementima. Virtuelne sekvence mogu da procesuiraju sekvence različitih tipova (ali ne i različitih elemenata sekvenci). Njima upravlja virutalni upravljač sekvenci, koji tipično ima pokazivač ka svakom od individualnih običnih upravljača sekvenci. Virtuelne sekvence se mogu koristiti za sinhronizovanje i slanje običnih sekvenci ka nekoliko BFM-a.

Vrituelni upravljač sekvenci odnosno *driver* nije povezan ni na jedan specifični BFM. Otuda izostaje logika i funkcionalnost BFM-a. To je razlog zašto virtuelni upravljač sekvenci može samo da upravlja sekvencama, ali ne i elementima sekvence. Nijedna od metoda (predefinisanih ili kasnije dodatih) koje imaju veze sa elementima sekvenci ne mogu da se koriste u viruelnom upravljaču sekvenci.

Na slici 3.4.1.1 je prikazan primer jednog složenog verifikacionog okruženja. Kao što se može videti na svaki od interfejsa DUT-a prikačen je po jedan odgovarajući eVC. Svaki od ovih eVC-a ima svoju biblioteku sekvenci koja je specifična za tačno određeni protokol.



Slika 3.4.1.1. Verifikacija DUT-a koji sadrži više različitih interfjsa [9]

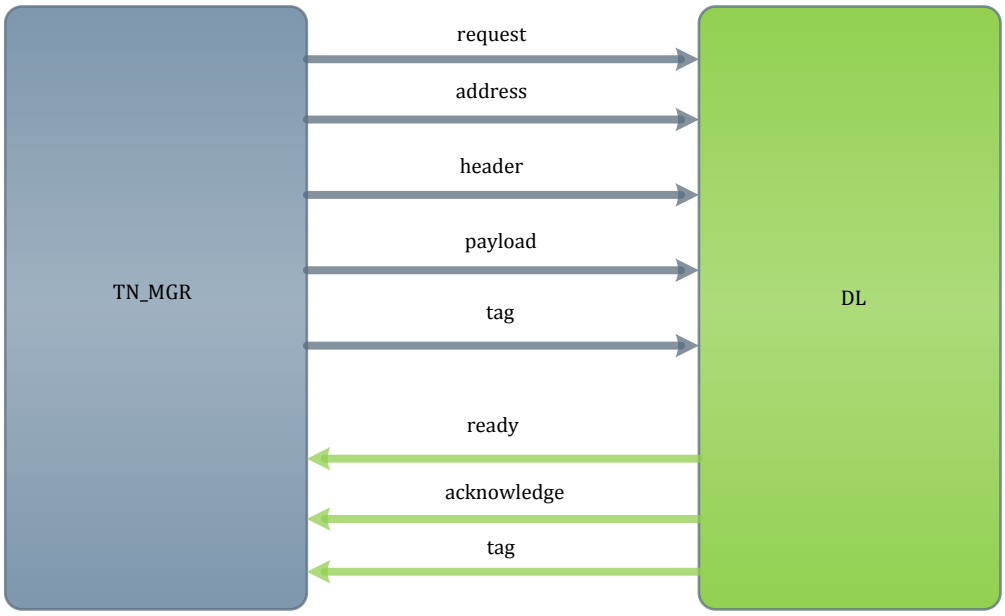
Sada dolaze do izražaja prednosti virtuelnog upravljača sekvenci i virtuelne sekvence. Naime scenarija i sekvence koje se generišu na sloju samoga čipa (*top level*) mogu da sadrže sekvence različitih protokola. Te sekvence virtuelni upravljač sekvenci „razbija" (u skladu sa pravilima definisanim u test scenariju tj. početnoj sekvenci) na pojedinačne i prosleđuje ih odgovarajućim eVC upravljačima sekvenci koji ih prosleđuje BFM-u koji ih opet šalje na DUT signale. Na ovaj način je omogućeno i paralelno upravljanje na više interfejsa DUT-a (na primer, u isto vreme može da se pošalje USB i *Ethernet* transakcija).

Za više informacija vezanih za sekvence i za njihovo kreiranje i upravljanje pogledati [9].

4. SPECIFIKACIJA N2DL INTERFEJSA

N2DL *(Network-to-Data Layer*) interfejs je interfejs koji povezuje mrežni sloj sa slojem linka podataka. On služi da bi se sa mrežnog sloja, TN_MGR (*Transport Network Manager*) izdao zahtev sloju linka podataka DL (*Data Link*), da pošalje paket mrežnog sloja N_PDU (*Network Protokol Data Unit*) ka nižem sloju bez posrednika, odnosno na *peer*. Informacija sa *peer* koja treba da bude obrađena je organizovana u transportne pakete T_PDU (*Transport Protocol Data Unit).* U suštini to su određeni podaci na određenim adresama u memoriji. Paket N_PDU sadrži informaciju sa koje adrese i koji deo paketa T_PDU treba da bude obrađen.

Postoje dve instance za ovaj protokol, jedna za TN_MGR sloj i druga za DL sloj. Na slici 4.1 je prikazana blok šema N2DL interfejsa. Ulogu *master*-a odrađuje TN_MGR instanca, dok je uloga *slave*-a rezervisana za DL instancu.



Slika 4.1. Blok šema N2DL interfejsa [10]

U tabeli 4.1 je dat opis svih signala koji sačinjavaju N2DL interfejs. Dati su detaljan opis funkcije, širina magistrale i vrsta (ulazni ili izlazni) signala. DL instanca je uzeta kao referenca. Ako se u prefiksu naziva signala koristi *_tn_* onda se signal odnosi na mrežni sloj. Ako se u prefiksu naziva signala koristi *_dl_* onda se signal odnosi na sloj linka podataka. Ukoliko nijedna od pomenute dve naznake ne postoji u nazivu signala, onda je signal zajednički i za viši sloj (*master*) i za niži sloj (*salve*).

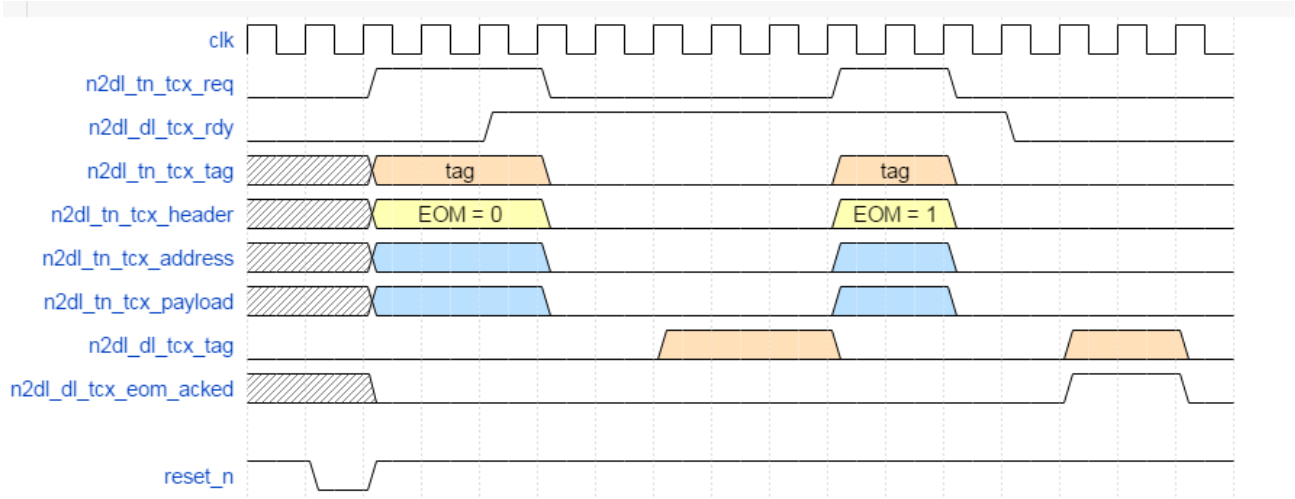
Tabela 4.1. Signali N2DL interfejsa

Naziv signala	Opis signala	Smer signala	Širina signala [bit]
Clk	Signal takta. Određuje takt interfejsa.	Ulazni	1
Reset	Glavni *reset* signal. Njegovim aktiviranjem sve vrednosti ostalih signala se vraćaju na početne vrednosti tj. na „0". Reset je aktivan na „0“.	Ulazni	1
n2dl_tn_tcx_req	Kada je ovaj signala postavljen na „1“, svi signali transportnog sloja su validni. Njega postavlja TN_MGR sloj kada zahteva od DL sloj da se pošalje paket na *peer*.	Ulazni	1
n2dl_tn_tcx_tag	Koristi ga TN_MGR sloj da bi obeležio transakciju koja zahteva poslednji segment poruke EOM (*End Of Message*) potvrdu od strane DL sloja.	Ulazni	4
n2dl_tn_tcx_header	Sadrži informaciju o zaglavlju transportnog i mrežnog signala. Ukoliko je L3s=1: *tn_tcx_header*[15] – L3s *tn_tcx_header*[14:8] – DestDeviceId_Enc *tn_tcx_header*[7] – L4s *tn_tcx_header*[6:2] – DestCPortId_Enc *tn_tcx_header*[1] – FCT *tn_tcx_header*[0] – EOM Ukoliko je L3s=0, onda će niži biti *tn_tcx_header*[14:0] imati nedefinisane vrednosti budući da se u uvom slučaju šalje zahtev za paket sa dugačkim (*long*) zaglavljem.	Ulazni	16
n2dl_tn_tcx_addr	32 – bitna adresa sadrži informaciju o lokaciji jednog dela ili celog T_PDU tela paketa (*payload*).	Ulazni	32
n2dl_tn_tcx_payload_len	Veličina tela paketa na koju pokazuje *tn_tcx_addr*. Ukoliko je L3s=1, maksimalna vrednost ovoga signala može biti 256b. Ukoliko je L3s=0 maksimalna vrednost (za pakete sa dugačkim zaglavljem) može biti 512b.	Ulazni	9
n2dl_dl_tcx_rdy	Kada je *dl_tcx_rdy* signal setovan na „1" to znači da TN_MGR sloj može da postavi svoj zahtev na interfejs i da će taj zahtev odmah biti prihvaćen. Ukoliko je signal na „0", strana koja je postavila zahtev mora da prolongira isti dok se *dl_tcx_rdy* signal ne promeni.	Izlazni	1
n2dl_dl_tcx_eom_acked	EOM potvrda. Kada je poslednji deo poruke poslat (EOM=1), TN_MGR sloj će čekati potvrdan odgovor od DL sloja da je T_PDU paket prihvaćen od strane bafera DL sloja.	Izlazni	1
n2dl_dl_tcx_tag	Služi da se označi poslednji zahtev koji je iziskivao potvrdu od DL sloja.	Izlazni	4

Komunikacija između TN_MGR i DL sloja se odvija na sledeći način:

1. TN_MGR sloj postavlja signale *n2dl_tn_tcx_tag*, *n2dl_tn_tcx_header*, *n2dl_tn_tcx_addr*, *n2dl_tn_tcx_payload*_len na željene vrednosti. Zatim setuje *n2dl_tn_tcx_req* (u istom periodu signala takta) na „1" signalizirajući DL sloju da je zahtev na signalima validan.
2. Ukoliko je signal *n2dl_dl_tcx_rdy* na „1" zahtev biva odmah prihvaćen, ukoliko nije TN_MGR sloj mora da produži slanje paketa time što drži *n2dl_tn_tcx_req* signal na visokoj vrednosti. Po prihvatanju paketa od strane DL sloja signal *n2dl_tn_tcx_req* biva oboren na "0" u sledećoj periodi signala takta.
3. Ako je EOM=1 u zaglavlju paketa od strane TN_MGR sloja (što signalizira da je ovo kraj paketa) DL sloj obara *n2dl_dl_tcx_rdy* signal u sledećoj periodi signala takta nakon obaranja *n2dl_tn_tcx_req* signala, a zatim šalje oznaku (kroz *n2dl_dl_tcx_tag* signal) i potvrdu da je primio poslednji deo paketa (u trajanju od jedne takt periode *n2dl_dl_tcx_eom_acked* signal biva postavljen na „1").
4. Ako je EOM=0 u zaglavlju paketa od strane TN_MGR sloja (što signalizira da je ovo nije kraj paketa već samo jedan njegov deo), DL sloj šalje oznaku kroz signal *n2dl_dl_tcx_tag* čime stavlja do znanja TN_MGR sloju da je opslužio ovaj deo paketa. Signal *n2dl_dl_tcx_rdy* ostaje na „1".
5. Obaranjem *n2dl_dl_tcx_rdy* signala, DL sloj govori TN_MGR sloju da trenutno obrađuje postojeći zahtev i da nove ne može da procesuira.

Na slici 4.2 se nalaze svi signali prikazani u funkciji vremena. Za više informacija vezanih za N2DL protokol pogledati [10].



Slika 4.2. Signali N2DL protokola prikazani u funkciji vremena [10]

5. N2DL eVC

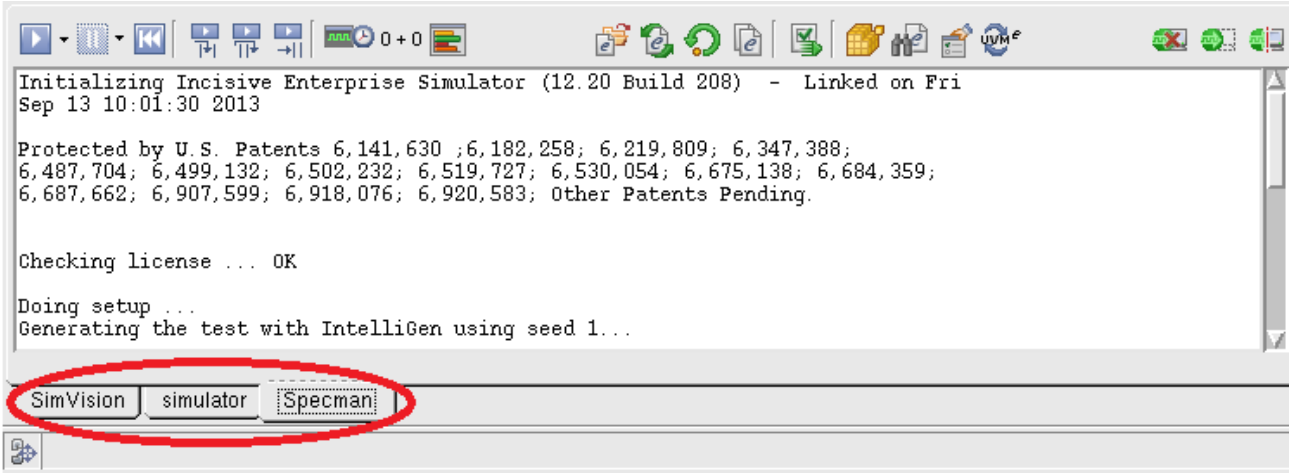
N2DL eVC je verifikaciona komponenta napisana u *e* programskom jeziku. Komponenta u zavisnosti od konfiguracije može da simulira kako mrežni deo interfejsa tako i deo interfejsa vezan za sloj linka podataka. Svaka instanca komponente može biti podešena da bude aktivna ili pasivna. O načinu konfiguracije eVC-a će biti reči u poglavlju 5.2. Sam programski jezik nije dovoljan za verifikovanje integrisanih kola. Osim njega potrebno je posedovati i određene programske alate.

e programski jezik spada u HVL (*Hardware Verification Language*) jezike, pogodan za pisanje verifikacionih okruženja, koja služe za verifikaciju dizajna integrisanih kola, napisanog u HDL (*Hardware Description Language*) jeziku. Ovakva verifikaciona okruženja odlikuje fleksibilnost i mogućnost ponovne upotrebe. Ovaj programski jezik kreiran je 1992. za *Specman tool* od strane osnivača kompanije *Verisity*. *e* je prvi HVL jezik koji je podržava *constrained-random* generator, generator stimulusa i *checkers* u jednom alatu. To omogućava pisanje celog *testbench*-a u jednom HVL jeziku, što pre pojave ovakvog koncepta nije bilo moguće. *e* programski jezik se zasniva na AOP (*Aspect-Oriented Programming*) pristupu, što omogućava da se *testbench* organizuje kroz aspekte. Tako se omogućava manipulacija određenim aspektima u klasi, dodavanjem *constraints*, funkcionalnosti i njihov *override* [13].

Specman je programski alat za automatski elektronski dizajn EDA (*Electronic Design Automation*) kompanije *Cadence* napravljen specifično za funkcionalnu verifikaciju integrisanih kola. On omogućava okruženje za rad, kompajliranje i debagovanje verifikacionih okruženja napisanih u *e* programskom jeziku. Sam *Specman* ne podržava HDL simulaciju, odnosno simulaciju dela koda napisanog, na primer, u Verilogu ili VHDL-u, kao ni kompajliranje istih. Zbog toga se *Specman* povezuje na HDL simulatore sa kojima radi u paru [11]. *Cadence Incisive* je paket programskih alata koji služe za HDL kompajliranje i simulaciju. Dva najbitnija alata iz ovoga paketa su svakako *NC Sim* i *Sim Vision*. Novije verzije *Incisive* u sebi sadrže i *Specman* [12].

NC Sim je HDL simulator koji se koristi za simuliranje DUT-a čije ponašanje je opisano HDL kodom. Programski paket *Incisive* prvo kompajlira HDL kod kojim je opisan DUT, pa zatim simulira ponašanje projektovanog uređaja korišćenjem alata *NC Sim* koji koristi iskompajliran HDL kod pri simulaciji ponašanja DUT-a. *Sim Vision* služi za grafički prikaz signala u jedinici vremna kao i za grafički prikaz šeme samog DUT-a.

Na slici 5.1 je prikazana osnovna konzola za rad sa alatima. Uokvireno crvenom bojom na slici su gore pomenuti alati integrisani u jedan zajednički interfejs.



Slika 5.1. Osnovna konzola za rad programskog paketa Incisive

5.1. Arhitektura N2DL eVC-a

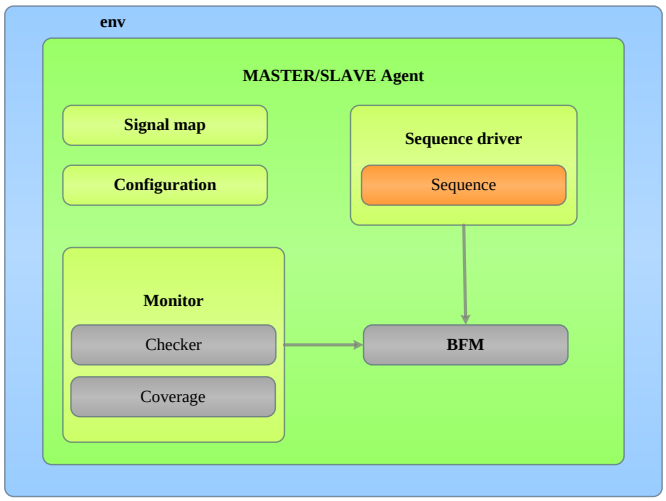
Komponente eVC-a mogu biti statičke i dinamičke. Statičke komponente su jedinice (*units*), koje se pre simulacije konfigurišu i povežu, a tokom simulacije se ne menjaju. Dinamičke komponente su strukture (*struct*), transakcije, odnosno sekvence (*sequence*) [9]. One se menjaju u toku simulacije u zavisnosti od načina na koji je test napisan. Više o ovome biće rečeno u poglavljima koja slede.

Tabela 5.1.1 sadrži spisak svih komponenti uz kratak opis. Sufiks *_u* u nazivu komponente označava jedinicu, dok sufiks *_s* označava strukturu.

Tabela 5.1.1. Komponete N2DL eVC-a

Unit/Struct	Opis
hdh_n2dl_agent_u	Ovo je osnovni tip jedinice za sve *agent*-e (viši i niži sloj) koji su povezani na magistralu.
hdh_n2dl_config_u	Konfiguraciona jedinica koji sadrži sva relevantna konfiguraciona polja vezana za eVC.
hdh_n2dl_signal_map_u	Jedinica sadrži sve signale vezane za N2DL eVC.
hdh_n2dl_monitor_u	*Monitor* jedinica skuplja informacije o svakoj transakciji koja se pojavi na signalima.
hdh_n2dl_bfm_u	Osnovna jedinica za sve BFM-e (viši i niži sloj).
hdh_n2dl_env_u	Jedinica je najviši sloj (*top level*) eVC-a.
hdh_n2dl_master_driver_u	Jedinica se koristi za upravljanje sekvencama višeg sloja.
hdh_n2dl_slave_driver_u	Jedinica se koristi za upravljanje sekvencama nižeg sloja.
hdh_n2dl_packet_s	Struktura koja sadrži sve relevantne informacije potrebne za jednu transakciju.
hdh_n2dl_master_sequence	Struktura koja predstavlja generičku sekvencu za interfejs *agent*-a višeg sloja.
hdh_n2dl_slave_sequence	Struktura koja predstavlja generičku sekvencu za interfejs *agent*-a nižeg sloja.

Na slici 5.1.1 je prikazana šema arhitekture N2DL eVC-a.



Slika 5.1.1. Blok šema arhitekture N2DL eVC-a

U pitanju je standardna eRM arhitektura sa jednim *agent*-om instanciranim u *env_u* jedinici. *Agent* se može konfigurisati tako da predstavlja i viši i niži sloj, odnosno *master* i *slave*. Sam *master* sadrži *monitor*, BFM, *signal map*, *sequence driver*, *configuration*. Kao ekstenzije *monitor*-a, u njemu su i *checkers* i *coverage*.

Uz licencu za simulator *Specman* je uključen i *evc_utils* paket koji sadrži sve komponente definisane eRM metodologijom. Svaka komponenta N2DL eVC-a nasleđuje neku komponentu iz *evc_utils* paketa. Prilikom nasleđivanja nova komponenta zadržava sva svojstva roditeljske komponente.

U daljem tekstu će svaka eVC komponenta ponaosob biti detaljno obrađena. Nakon toga biće reči i o ostalim *e* fajlovima koji sačinjavaju eVC, ali nisu komponente istog. Funkcija tih fajlova je čisto tehničke prirode tj. oni služe da bi kompajler mogao na pravi način da iskompajlira kod.

5.1.1. Komponenta n2dl_env_u

Komponenta ***n2dl_env_u*** je osnovna jedinica eVC-a, odnosno njegova statička komponenta. *env* u nazivu komponente je skraćenica za okruženje (*environment*) i odnosi se na verifikaciono okruženje eVC-a. Okruženje nasleđuje karakteristike *any_env*, koji se nalazi definisan u *evc_utils* paketu. Koristi se sledeća komanda:

```
unit n2dl_env_u like any_env {
        ...
};
```

any_env [13] je predefinsana jedinica u okviru *evc_utils* paketa, koja se koristi za osnovno definisanje svakog okruženja. Ova jedinica ima određen broj predefinisanih metoda koje se koriste pri pokretenju simulacije za definisanje ponašanja verifikacionog okruženja. Jedinica sadrži i polje (*field*) name koji je tipa *n2dl_env_name_t*. U njemu se čuva ime okruženja. Za to se koristi predefinisana metoda *short_name()*, koja vraća vrednost polja *name*.

n2dl_env_u jedinica u sebi sadrži i instance dve komponente. Prva je *logger* koji je tipa *message_logger* i instancira se na sledeći način:

logger: message_logger is instance;

keep soft logger.verbosity == LOW

Instanca *message_logger* je definisana u *evc_utils* paketu. Njena funkcija je manipulacija porukama koje se nalaze u kodu i koje će se potencijano naći u izveštaju simulacije, kao i na ekranu konzole za vreme simulacije. Ovde je važnost poruka (*verbosity*) ograničena na LOW, što znači da će se u izveštaju naći sve poruke koje imaju postavljenu važnost poruke na LOW ili viši. Viši sloj može da bude MEDIUM, HIGH ili FULL. Niži sloj je NONE. Poruke imaju ogroman značaj u traženju grešaka tj. njihovog izvora. Ova komponenta se ne nalazi u blok šemi arhitekture eVC-a, jer je svaki eVC poseduje. Ona služi isključivo za manipulaciju porukama i nema nikakve veze sa protokolima koje eVC predstavlja.

Druga komponenta je agent koji je tipa ***n2dl_agent_u*** i o njemu će biti reči u odeljku 5.1.2. Instancira se na sledeći način:

agent : n2dl_agent_u is instance;

5.1.2. Komponenta n2dl_agent_u

Komponenta ***n2dl_agent_u*** je *unit*, odnosno statička komponenta eVC-a, koja predstavlja uređaj u verifikacionom okruženju. Polja koja *agent* sadrži su:

- *name* : polje je tipa *n2dl_env_name_t* i sadrži ime instance eVC-a čiji je on deo;
- *p_env* : pokazivač na *n2dl_env_u* komponentu;
- *kind* : polje je tipa *n2dl_agent_kind_t* i može biti MASTER ili SLAVE. U ovom slučaju MASTER prestavlja transportni sloj protokola, a SLAVE sloj linka podataka. Prema pravilu *kind* je ograničen da inicijalno bude MASTER i to tako što:

  kind : n2dl_agent_kind_t;

  keep soft kind == MASTER;

- *active_passive* : polje je tipa *erm_active_passive_t*. *erm_active_passive_t* je predefinisan tip u okviru *evc_utils* paketa. Ovo polje određuje da li je agent aktivan (ACTIVE) ili pasivan (PASSIVE).

U agentu su instancirane i neke statičke komponente. Prva je *smp* koja je tipa *n2dl_signal_map_u* i predstavlja instancu jedinice signal mape o kojoj će biti reči u sledećem odeljku. Signal mapa instancira se na sledeći način:

smp : n2dl_signal_map_u is instance;

Sledeća komponenta je *monitor* koji je tipa *n2dl_monitor_u* i predstavlja instancu jedinice *monitor*-a o kome će biti reči u odeljku 5.1.4

monitor : n2dl_monitor_u is instance;

Komponenta *configuration* je instanca ***n2dl_configuration_u*** jedinice. U ovom eVC-u *config* instanca se ne koristi. Ona postoji samo da bi se uslovi eRM metodologije zadovoljili i kako

bi se u slučaju neke buduće nadogradnje mogla koristiti. Uzrok toga je da sva konfiguraciona polja koja su potrebna, već postoje definisana u okviru *agent*-a. Kod koji se koristi za instanciranje je:

config: n2dl_configuration_u is instance;

U slučaju da je eVC konfigurisan kao aktivan, u okviru *agent*-a se instanciraju *driver* i BFM komponente. U slučaju da je *master*, kod je sledeći:

```
extend ACTIVE MASTER n2dl_agent_u {
    bfm  : MASTER n2dl_bfm_u is instance;
    driver : MASTER n2dl_driver_u is instance;
    ...
};
```

Ako se koristi *slave* konfiguracija, koristi se kod:

```
extend ACTIVE SLAVE n2dl_agent_u {
    bfm  : SLAVE n2dl_bfm_u is instance;
    driver : SLAVE n2dl_driver_u is instance;
    ...
};
```

O jedinicama *driver* i BFM biće više reči u nastavku, u poglavlju 5.1.5 i 5.1.6

Pored polja i instanci drugih jedinica, u ***n2dl_agent_u*** su definisani i događaji (*events*). Događaji su deo *e* programskog jezika koji mogu da se posmatraju kao delta impulsi. Oni se aktiviraju kada se dese određene, definisane aktivnosti unutar simulacije. Na taj način stiče se pojam o vremenu u eVC-u. Oni se mogu definisati korišćenjem "*is*" *temporal expression* sintakse. Događaji koji su definisani u okviru *agent*-a opisani su u tabeli 5.1.2.1.

Tabela 5.1.2.1. Događaji definisani u *n2dl_agent_u* jedinici

Event	Opis
clock_unqualified_e	Događaj koji signalizira rastuću ivicu signal takta (*clk*).
clock_qualified_e	Događaj koji signalizira rastuću ivicu signal takta u koliko je *reset* deaktiviran
clock_unqualified_fall_e	Događaj koji signalizira opadajuću ivicu signal takta
clock_qualified_fall_e	Događaj koji signalizira opadajuću ivicu signal takta ukoliko je *reset* deaktiviran

5.1.3. Komponenta n2dl_signal_map_u

Komponenta ***n2dl_signal_map_u*** je *unit*, koja sadrži sve signale relevantne za N2DL eVC i predstavlja vezu između *e* okruženja, kao verifikacionog okruženja i HDL okruženja, kao razvojne table. ***n2dl_signal_map_u*** je takođe statička komponenta eVC-a. Pri instanciranju eVC-a u verifikaciono okruženje, ovi signali se povezuju na HDL signale, odnosno signale DUT-a i na taj način ostvaruju konekciju. O povezivanju eVC signala sa DUT signalima biće reči u poglavlju 5.2.

Svi signali, izuzev signala takta (*clk*) definisani su kao ulazno-izlazni portovi (*simple_port*) koji se vezuju za spoljni izvor. Primer *reset* signala:

reset_n : inout simple_port of bit is instance;

keep bind(reset_n, external);

Signal takta (*clk*) je podeljen na dva ulazna porta (*event_port*). Jedan je za rastuću ivicu (*clk*), dok je drugi za opadajuću ivicu (*clk_fall*). Oba porta se vezuju na jedan zajednički HDL signal takta, međutim svaki od njih posmatra svoju ivicu. Signal takta je definisan na sledeći način:

```
clk: in event_port is instance;
    keep clk.hdl_path() != "" => bind(clk, external);
    keep clk.hdl_path() == "" => bind(clk, empty);
    keep clk.edge() == rise;
clk_fall: in event_port is instance;
    keep clk.hdl_path() != "" => bind(clk, external);
    keep clk.hdl_path() == "" => bind(clk, empty);
    keep clk.edge() == fall;
```

Portovi se koriste za interakciju sa HDL kodom u simulatoru. *event_port* se koristi za transfer događaja (*event*) između dve jedinice napisane u *e*-u ili između jedinice napisane u *e* i nekog eksternog objekta. *simple_port* se koristi za transfer podataka ka eksternim ili internim objektima. Ovi portovi mogu biti *input*, *output* ili *inout* [14].

U okviru *event_ port*-a definiše se svojstvo *edge()*, kojim se omogućava sinhronizacija na rastuću ivicu (*rise*) odnosno opadajuću ivicu (*fall*), signala koji se vezuje na *clk*, odnosno *clk_fall* port. Ukoliko se iz nekog razloga port ne poveže sa odgovarajućim HDL signalom simulator će javiti grešku. Da bi se to izbeglo, ukoliko HDL signal nije definisan, *event port* se vezuje (*bind*) na "ništa" (*empty*), odnosno koristi se kod *bind(clk, empty)*.

5.1.4. Komponenta n2dl_monitor_u

n2dl_monitor_u je kao i sve prethodno definisane komponente, statička komponenta eVC-a. Njegova osnovna funkcija je nadziranje signala eVC-a i njihovo pakovanje u paket odnosno transakciju. O transakcijama će biti više reči u odeljku 5.1.8

Polja definisana u okviru ove jedinice su:

- *name* : polje je tipa *n2dl_env_name_t* i sadrži ime instance eVC-a čiji je monitor deo. Metoda *short_name()* vraća vrednost "MON",
- *p_agent* : polje predstavlja pokazivač na ***n2dl_agent_u*** komponentu, odnosno jedinicu u kojoj je *monitor* instanciran;
- *unit_vt_style* : polje je tipa *vt_style* i označava boju kojim će se poruke iz *monitor*-a ispisivati u konzoli. Metoda *short_name_style()* vraća vrednost polja *unit_vt_style*. Određena boja je plava i to na sledeći način:

  ```
  unit_vt_style : vt_style;
      keep soft unit_vt_style == BLUE;
  ```

- *has_checker* : polje je tipa *bool*. Ukoliko je postavljen na TRUE, *monitor* jedinica će tada imati uključen protokol kontrolora signala eVC-a. U poglavlju 5.1.4.1 biće opisan protokol kontroler. Prema pravilu *has_checker* polje je postavljeno na TRUE.

has_checker : bool;

keep soft has_checker == TRUE;

- *has_coverage* : polje je tipa *bool*. Ako je setovano na TRUE, *monitor* će tada imati uključeno skupljanje *coverage*. Više o tome biće reči u odeljku 5.1.4.2. Po konvenciji *has_coverage* polje ima inicijalnu vrednost TRUE i definiše se kodom:

has_coverage : bool;

keep soft has_coverage == TRUE;

Jedinica ***n2dl_monitor_u*** sadrži i dve fundamentalne metode. Prva je *packet_done()*. Argument metode je tipa *n2dl_packet_s*. Metoda se koristi za proveru primeljenih podataka unutar paketa. U slučaju N2DL eVC-a, primljeni podaci su dovoljno jednostavni da nije potrebna implementacija pomenute metode. Ona u *monitor*-u postoji samo radi forme.

Druga metoda je *collecting_packets()*. Ova metoda služi da od vrednosti koje vidi na signalima napravi paket. Metoda TCM (*Time Consuming Method*) odnosno, metoda koja ima pojma o vremenu. Ona po pokretanju ostaje u beskonačnoj petlji nadgledajući signale sve do kraja testa. Startuje se na samom početku simulacije komandom:

run() is also {

start collecting_packets();

};

Osim polja i metoda, ***n2dl_monitor_u*** jedinica sadrži i događaje, koji su opisani u tabeli 5.1.4.1, kao i *packet_collected* port metodu. Port metoda je konfigurisana kao izlazni (*out*) port i koristi se da bi se prikupljeni paket prosledio *scoreboard*-u na dodatnu proveru podataka. *method_port* nam omogućava da uspostavimo komunikaciju između *e* metoda i funkcija koje su definisane u eksternim jedinicama ili između *e* metoda koje su definisane u različitim *e* jedinicama. Port je definisan kodom:

packet_collected : out method_port of packet_t is instance;

keep bind (packet_collected, empty);

…

packet_collected$(current_packet);

Scoreboard nije deo N2DL eVC-a, pa će biti instanciran kao zasebna komponenta. Više detalja o tome će biti reči u poglavljima 5.1.7 i 5.2.

Tabela 5.1.4.1 Događaji definisani u *n2dl_monitor_u* jedinici

Event	**Opis**
packet_send_eom_0_e	Događaj koji se emituje ako je EOM bit jednak “0”.
packet_send_eom_1_e	Događaj koji se emituje ako je EOM bit jednak “1”.
packet_received_e	Događaj koji se emituje kada je paket prikupljen od strane *monitor*-a.
req_and_rdy_true_e	Događaj koji se emituje ako signali *n2dl_tn_tcx_req* i *n2dl_dl_tcx_rdy* imaju vrednost “1”.

i) Protokol kontrolori (checkers)

Checkers nisu zasebne jedinice, već ekstenzija *monitor*-a i predstavljene su kodom:

extend has_checker n2dl_monitor_u {

…

};

Njihova uloga je da kontrolišu saobraćaj na signalima, tj određene scenarije definisane protokolom. Ukoliko ono što se dešava nije u skladu sa protokolom, *checkers* signaliziraju grešku. *Checkers* su napisani tako da signaliziraju *dut_error*, tj. simulacija će se prekinuti čim se prva greška pojavi.

Kao što je već pomenuto, *checkers* se pri definisanju eVC-a mogu uključiti ili isključiti, pomoću polja *has_checker* definisanog u *monitor*-u. *Chekers* su uključeni u slučaju da se polje *has_checker* setuje na TRUE, odnosno isključeni ako se isto polje setuje na FALSE.

U okviru n2dl eVC-a definisani su sledeći *checker*-i:

- *n2dl_tn_tcx_req* mora da se obori tačno jednu periodu signal takta (*clk*) nakon što se *n2dl_dl_tcx_rdy* signal postavi, u suprotnom *checker* ce prijaviti grešku.
- u slučaju da je paket poslat i EOM = 0, *slave* mora da pošalje svoju oznaku (*tag*) *master*- u inače se prijavljuje greška.
- u koliko je paket poslat i EOM = 1 *slave* mora da pošalje svoju oznaku *master* i da postavi *n2dl_dl_tcx_eom_acked* signal, u suprotnom će prijaviti grešku.

Checkers koriste i događaje. Događaji su opisani u tabeli 5.1.4.1.1.

Tabela 5.1.4.1.1. Događaji definisani u okviru *checkers*

Events	**Opis**
tn_tcx_req_true_e	Događaj koji se emituje ukoliko *n2dl_tn_tcx_req* signal ima vrednost „1”.
tn_tcx_req_rise_e	Događaj koji se emituje na rastuću ivicu *n2dl_tn_tcx_req* signala
tn_tcx_req_fall_e	Događaj koji se emituje na opadajuću ivicu *n2dl_tn_tcx_req* signala
dl_tcx_rdy_rise_e	Događaj koji se emituje na rastuću ivicu *n2dl_dl_tcx_rdy* signala
dl_tcx_rdy_true_e	Događaj koji se emituje ukoliko *n2dl_dl_tcx_rdy* signal ima vrednost „1”
dl_tcx_eom_acked_true_e	Događaj koji se emituje ukoliko *n2dl_dl_tcx_eom_acked* signal ima vrednost „1”
dl_tcx_tag_true_e	Događaj koji se emituje ukoliko *n2dl_dl_tcx_tag* signal ima vrednost različitu od „0”

ii) Pokrivenost (coverage)

Coverage se ne definiše kao zasebna jedinica, već kao ekstenzija *monitor*-a i to na sledeći način:

extend has_coverage n2dl_monitor_u{

. . .

};

Za uključivanje, odnosno isključivanje *coverage*, pri konfigurisanju eVC-a koristi se *has_coverage* polje. Ono se koristi na identičan način kao i *has_checker* polje u slučaju *checker*-a. Pod pojmom *coverage* podrazumeva se *functional coverage* ili funkcionalna pokrivenost, i daje informaciju o tome koliko od funkcionalnosti N2DL eVC-a je pogođeno testovima. Svaki segment funkcionalnosti mora biti testiran, što znači da funkcionalna pokrivenost mora biti 100%.

Functional coverage se može realizovati na više načina. U ovom slučaju biće korišćen *group model* [15]. *Group model* podrazumeva presek stanja, u određenim vremenskim trenucima i

beleženje tako dobijenih vrednosti signala. Ovo se postiže definisanjem *coverage groups*. *Coverage group* je deo strukture i definisana je svojim imenom, odnosno *coverage keyword*. Ona sadrži opis podataka čije će se vrednosti skupljati i pod kojim uslovima, tokom simulacije [5]. Svaka grupa može da sadrži sledeće:

- *clocking event* : definisani događaj koji govori u kom vremenskom trenutku se vrši presek (*sample*)
- *coverage points* : promenljiva ili izraz koji se odnosi na podatak koji želimo da bude semplovan. Svaki *coverage group* sadrži više *coverage points*, dok svaki *coverage point* u sebi ima definisano više *bin*-ova koji predstavljaju vrednost ili opseg vrednosti parametara podatka koji bi semplovanjem trebao biti pokriven.
- *cross coverage*: kombinacija različitih *coverage points* ili nekih drugih promeniljivih gde se očekuju sve moguće kombinacije vrednosti ta dva *coverage poin*t-a
- *coverage options* : koristi se za kontrolu ponašanja *coverage groups*.

U okviru sledećih tabela 5.1.4.2.1 i 5.1.4.2.2, prikazane su dve *coverage groups*, *coverage_1* i *coverage_2*, koje se koriste u okviru *coverage model*-a definisanog za ovaj primer. Svaka od grupa ima svoje elemente pokrivenosti, odnosno *coverage points*.

Za grupu, *coverage_1*definisano je sedam različitih *covegare points*, dok je za *coverage_2* definisano četiri *coverage points*. Za svaki od *coverage points* definisan je u kodu komponente, opseg ili vrednost određenog polja za koji se očekuje da bude „pogođen“, odnosno generisan u nekoj kombinaciji paketa. Na primer, kod *address* definisana su tri različita opsega u kojima u kojima se očekuje da sve moguće vrednosti budu generisane. Ti opsezi su, SMALL od 0 do 9 b, MEDIUM od 10 do 19 b, LARGE od 20 do 31 b. Slično je i za *payload*. Za *coverage group*, kao što je *L3s* očekuje da sve moguće vrednosti za 15. bit zaglavlja budu generisane. Sve moguće vrednosti predstavljaju sve moguće kombinacije „0“ i „1“ na svim pozicijama definisanim opsegom. *Cross coverage*, *request, ready* predstavlja sve moguće kombinacije vrednosti signala *n2dl_tn_tcx_req* i *n2dl_dl_tcx_rdy*.

Tabela 5.1.4.2.1. Lista coverage points za coverage group coverage 1

Coverage group	*Coverage points*	Opis
coverage_1	*address*	Sve kombinacije adresa raspoređenih u nekoliko opsega (SMALL, MEDIUM, LARGE)
	payload	Sve kombinacije veličine tela paketa raspoređenih u nekoliko opsega (VERY_SMALL, VERY_LARGE).
	L3s	[15] bit u polju zaglavlja.
	DestDeviceID_Enc	[14:8] bit u polju zaglavlja.
	L4s	[7] bit u polju zaglavlja.
	DestCPortID_Enc	[6:2] bit u polju zaglavlja.
	FCT	[1] bit u polju zaglavlja.

Tabela 5.1.4.2.2. Lista *coverage points* za *coverage group coverage 2*

Coverage group	*Coverage points*	Opis
coverage_2	*EOM*	[0] bit u polju zaglavlja.
	Request	Sve vrednosti *n2dl_tn_tcx_req* signala.
	ready	Sve vrednosti *n2dl_dl_tcx_rdy* signala.
	cross request, ready	Unakrsna pokrivenost *request* i *ready* elemenata pokrivenosti.

5.1.5. Komponenta n2dl_master(slave)_driver_u

Komponenta ***n2dl_master_driver_u***, odnosno ***n2dl_slave_driver_u*** predstavlja *driver* i realizovana je na identičan način za *master* i *slave*. On predstavlja statičku komponentu eVC-a i funkcija mu je prosleđivanje sekvenci na BFM koji ih dalje šalje na fizičke portove, odnosno kao signale. *Driver* je kreiran zajedno sa *sequence*, korišćenjem *sequence statement* [9], koja je opisana u poglavlju 5.1.9. Kod za kreiranje *master sequence* i njenog *driver*-a:

```
sequence n2dl_master_sequence using
  item            = MASTER n2dl_packet_s,
  created_driver  = n2dl_master_driver_u
  created_kind    = n2dl_master_sequence_kind_t;
```

Isti kod se može implemetirati i za *slave*, zamenom ključne reči *master_* sa *slave_*. U okviru *n2dl_sequence.e* komponente definisani su *driver*-i i *sequence* za *master* i *slave*.

```
extend n2dl_master_driver_ {
  ...
};
```

Osnovna polja koja ova jedinica sadrži su :

- *name* : polje tipa *n2dl_env_name_t* i sadrži ime instance eVC-a čiji je *driver* deo
- *p_agent* : pokazivač na *n2dl_agent_u* komponentu u okviru koje je *driver* instanciran
- *kind* : polje tipa *n2dl_agent_kind_t* koji može biti MASTER ili SLAVE

Pored toga definisan je i događaj *reset_done event*, koji se emituje na rastuću ivicu *reset* signala, kao i TCM (*Time Consuming Method*) metod *wait_for_reset()* koja čeka da se *reset* završi. Metoda *wait_for_reset()* biće pozvana prva, na startu od strane *agent*-a. Ona omogućava korisniku da natera *agent*-a da sačeka da se *reset* završi, pre nego što počne sa slanjem paketa.

5.1.6. Komponenta n2dl_bfm_u

Komponenta *n2dl_bfm-u* je jedinica eVC-a čija je funkcija da transakcije u vidu sekvenci pošalje u skladu sa N2DL protokolom na signale eVC-a.

Polja koje ova jedinica sadrži su:

- *name* : polje tipa *n2dl_env_name_t* i sadrži ime instance eVC-a čiji je on deo. Metod *short_name()* vraća vrednost BFM
- *p_agent* : pokazivač na *n2dl_agent_u* komponentu, odnosno jedinicu u kojoj je instanciran BFM
- *kind* : polje tipa *n2dl_agent_kind_t* i može biti MASTER ili SLAVE
- *unit_vt_style* : polje tipa *vt_style* i označava boju kojom će se poruke BFM-a ispisivati u konzoli.

U zavisnosti od vrednosti polja *kind* koriste se različite ekstenzije BFM-a. Jedna MASTER i druga SLAVE.

```
extend MASTER n2dl_bfm_u {
  ...
};
extend SLAVE n2dl_bfm_u {
  ...
};
```

Master BFM se koristi za postavljanje signala mrežnog sloja u skladu sa N2DL protokolom. *Slave* BFM se koristi da upravlja signalima sloja linka podataka u skladu sa N2DL protokolom. Oba BFM-a imaju pokazivač na *driver*, odnosno svaki na svoj:

```
p_driver : n2dl_master_driver_u ;  –  pokazivač na master driver
p_driver : n2dl_slave_driver_u ;   – pokazivač na slave driver
```

Pored toga, BFM-ovi sadrže i po jednu *packet_generated* port metodu. Port metoda je konfigurisana da bude izlazni port (*out*) i koristi se da bi se skupljeni paket prosledio *scoreboard*-u na dodatnu proveru podataka. Prosleđivanje se vrši na sledeći način, korišćenjem koda:

```
packet_generated$(curr_packet);
```

gde je *curr_packet* tipa paketa koji se prosleđuje odnosno *n2dl_paket_s*. Osnovni TCM *execute_item()* jednom kada se startuje, ostaje da se vrti u beskonačnoj petlji. Prvo što ovaj *event* traži od *driver*-a jeste transakcija za slanje korišćenjem *built-in methode get_next_item()*. Pre slanja na signale, paket se šalje *scoreboard*-u, na poređenje sa paketom koji *monitor* tek treba da skupi. Zatim se metodom *drive_packet()* transakcija konvertuje u signale. Nakon što se transakcija završi, potrebno je kontrolu ponovo vratiti *driver*-u. To se postiže emitovanjem predefinisanog događaja u *driver, item_done()*. Kod izgleda:

```
execute_items()@clock is {
  while(TRUE) {
    curr_packet = p_agent.get_next_item();
    packet_generated$(curr_packet);
    driver_packet(curr_packet);
    emit p_driver.item_done;
  };
};
```

Osim pomenutih metoda oba BFM-a sadrže metodu *init_signal()* koja postavlja sve signale na početne vrednosti. Metoda se startuje na samom početku simulacije. *execute_items()* metoda se takođe startuje na početku simulacije:

```
run() is also{
  start init_signal();
  start execute_items();
};
```

Slave BFM sadrži još i *driver_rdy_signal()* TCM koji opslužuje upravljanje *n2dl_dl_tcx_rdy* signal, kao i TCM *set_tag_or_acked()* koja opslužuje postavljanje *n2dl_dl_tcx_eom_acked* i *n2dl_dl_tcx_tag* signala.

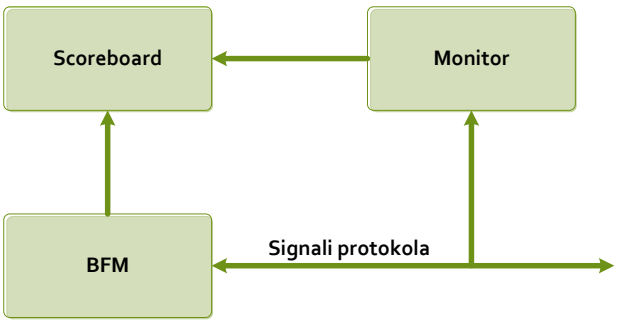
Događaji definisani u BFM-u dati su u tabeli 5.1.6.1

Tabela 5.1.6.1. Događaji BFM jedinice

Event	**Opis**
packet_started_e	Događaj koji se emituje kada BFM počne sa opsluživanjem paketa
packet_ended_e	Događaj koji se emituje kada BFM završi sa opsluživanjem paketa

5.1.7. Komponenta n2sl_scoreboard_u

Komponeta ***n2dl_scoreboard_u*** je jedinica koja nije deo eVC-a. Instancirana je u istom hijerarhijskom sloju kao i eVC, o čemu će biti više reči u poglavlju 5.2. Funkcija *scoreboard*-a u ovom slučaju je da proveri da li BFM dobro šalje transakciju, pre slanja iste na "žice" i paketa iz *monitor*-a skupljenog sa "žice" takođe poslatog putem port metode. Blok šema povezivanja *scoreboard*-a je data na slici 5.1.7.1.



Slika 5.1.7.1. Blok šema povezivanja *scoreboard*-a

Scoreboard od polja sadrži listu paketa *n2dl_packet_l*, koja je tipa *n2dl_packet_s*. U ovu listu se pomoću ulaznog (*in*) *method_port*-a, *add_n2dl_packet*, smeštaju paketi koji se od BFM-a šalju. Kada paket stigne iz *monitor*-a, poziva se ulazni (*in*) *method_port match_n2dl_packet*, koji uzima paket iz *n2dl_packet_l* liste i upoređuje ga sa pristiglim paketom iz *monitor*-a. Greška se prijavljuje u slučaju da se paketi razlikuju. Poređenje paketa vrši se pomoću predefinisane metode *deep_compare_physical()*, koja vrši poređenje fizičkih polja instanci dve strukture i vraća razliku.

Na samom kraju simulacije *n2dl_packet_l* lista treba da bude prazna. Odnosno da svaki paket BFM-a odgovara jednom od paketa *monitor*-a. Ako lista nije prazna prijavljuje se greška, kodom:

check that n2dl_packet_l.size() == 0

else dut_error("Scoreboard is not empty at the end of the test");

5.1.8. Komponenta n2dl_packet_s

Komponenta ***n2dl_packet_s*** predstavlja strukturu koja je dinamički deo eVC-a, odnosno menja se u toku simulacije. Od vrednosti polja paketa zavisi kako će BFM upravljati saobraćajem

na "žici". Polja mogu biti fizička i nefizička. Fizička polja ispred svog naziva imaju znak " % ". To su polja koja se vezuju na fizičke signale. Paket sadrži sledeća fizička polja:

- *addr* : polje predstavlja adresu, odnosno sadrži informaciju o lokaciji jednog dela ili celog tela T_PDU paketa.
- *tag_master* : polje predstavlja oznaku mrežnog sloja
- *tag_slave* : polje predstavlja oznaku sloja linka podataka
- *header* : polje predstavlja zaglavlje mrežnog nivao
- *payload_len* : polje predstavlja veličinu tela paketa

Nefizička polja nemaju svoje predstavnike u fizičkim signalima, ali se i dalje uzimaju u obzir prilikom slanja saobraćaja na signale. Paket sadrži sledeća nefizička polja:

- *delay_req_rise* : polje predstavlja broj perioda signala takta pre nego što će se *n2dl_tn_tcx_req* signal postaviti. Broj perioda ograničen je na opseg [1..10].
- *delay_rdy_rise* : polje predstavlja broj perioda signala takta pre nego što će se *n2dl_dl_tcx_rdy* signal postaviti. Broj perioda ograničen je na opseg [1..10].
- *delay_rdy_fall* : polje predstavlja broj perioda signala takta pre nego što će se *n2dl_dl_tcx_rdy* signal oboriti. Broj perioda ograničen je na opseg [1..100].
- *delay_tag_only* : polje predstavlja broj perioda signala takta pre nego što će se *n2dl_dl_tcx_tag* signal postaviti. Broj perioda ograničen je na opseg [1..10].
- *delay_tag_and_acked* : polje predstavlja broj perioda signala takta pre nego što će se *n2dl_dl_tcx_tag* i *n2dl_dl_tcx_eom_acked* signali postaviti. Broj perioda ograničen je na opseg [1..10].
- *delay_duration_tag_acked* : polje predstavlja dužinu trajanja akivnog satnja *n2dl_dl_tcx_tag* i *n2dl_dl_tcx_eom_acked* signala izraženog u broj rastućih ivica takta. Broj perioda ograničen je na opseg [1..10].

Pored navedenih paket sadrži i sledeća polja:

- *name* : polje je tipa *n2dl_env_name_t* i sadrži ime instance eVC-a čiji je on deo;
- *kind* : polje je tipa *n2dl_packet_kind_t* i ima podrazumevanu vrednost;
- *owner* : polje je tipa *n2dl_agent_kind_t* i govori o tome koji *agent* trenutno koristi paket.

Metode koje ***n2dl_packet_s*** komponenta sadrži su date u tabeli 5.1.8.1.

Tabeli 5.1.8.1. Metode u n2dl_packet_s komponenti

Method	Opis
read_eom()	Vraća vrednost EOM bita iz zaglavlja
read_l3s()	Vraća vrednost L3s bita iz zaglavlja
read_fct()	Vraća vrednost FTC bita iz zaglavlja
read_l4s()	Vraća vrednost L4s bita iz zaglavlja
read_DestDeviceID_Enc()	Vrednost DestDeviceID_Enc bita iz zaglavlja
read_DestCPortID_Enc()	Vraća vrednost DestCPortID_Enc bita iz zaglavlja.

5.1.9. Komponenta n2dl_sequence_u

Komponenta ***n2dl_sequence_u*** predstavlja strukturu koja je dinamički deo eVC-a, odnosno menja se u toku simulacije. U okviru nje definisane su sve strukture *n2dl_master_sequence* i *n2dl_slave_sequence*. *Master*, odnosno *slave*, sekvenca je ujedno i glavna (*main*) sekvenca, odnosno nema podsekvenci. Ona manipuliše elementima sekvence (*item*), u ovom slučaju to su transakcije, mada to generalno mogu biti i signali, polja ili druge sekvence. Tako formirana se preko *driver*-a prosleđuje BFM-u. U zavisnosti od testa, vrednosti *item*-a mogu konstantno da se menjaju u toku simulacije.

Da bismo definisali sekvencu, potrebno joj je definisati element sekvence (*item*), tip sekvence (*kind*) i *driver*, kroz koji se prosleđuje. To se radi na sledeći način:

```
sequence n2dl_master_sequence using
  item            = MASTER n2dl_packet_s,
  created_driver  = n2dl_master_driver_u,
  created_kind    = n2dl_master_sequence_kind_t;
```

Primer koda dat je za *master sequence*. Što se tiče *slave sequence*, definiše se na isti način s tim što se u kodu koristi ključna reč *slave*, umesto *master*. Sekvenca sadrži i polje *name* koje je tipa *n2dl_env_name_t* i sadrži ime instance eVC-a čiji je on deo.

Sequence koristi tri predefinisane metode, *pre_body(), body()* i *post_body()* [9]. Telo sekvence ili *body()* je glavna metoda i sve što sekvenca treba da radi realizovano je u okviru ove metode. Telo glavne sekvence se automatski generiše od strane *driver*-a i šalje, pozivom *run()* metode *sequence driver*-a. Metoda sekvence *body()* definiše životni ciklus *sequence*, tako da pre nego se inicira, pokrene predefinisani događaj *event started*. Po završeku *body()* metode inicira se *event ended.* Telo glavne *master* sekvence je ujedno i test scenario te će detaljnije biti objašnjen u poglavlju 5.3. Telo *slave* sekvence realizovano je kao beskonačna petlja transakcija koja će u BFM-u služiti za upravljanje odzivom na pobude sa *master* strane. Tako će na svaku *master* transakciju biti automatski odgovoreno *slave* transakcijom:

```
extend MAIN n2dl_slave_sequence {
  body()@driver.clock is only{
    while TRUE{
      do packet;
    };
  };
};
```

Vreme simulacije određeno je konfiguracijom simulatora. To vreme je često nedovoljno da se ceo test završi. Pošto je vreme testa uglavnom nepredvidivo, uveden je mehanizam *objection*. Dokle god je i jedan prigovor aktivan (*rise*), simulator neće završiti test. Zato se u *pre_body()* metodi nakon završetka *reset*-a signalizira, odnosno aktivira *objection* i to na sledeći način :

```
pre_body()@sys.any is first {
  driver.wait_for_reset();
  driver.raise_objection(TEST_DONE);
```

```
};
```

Metoda *pre_body()* omogućava nam da izvršimo bilo kakvu akciju pre iniciranje metode *body()* ili da stopiramo njeno izvršenje pozivom metode *stop()*. Po završetku *body()* metode, prigovor se deaktivira u *post_body()* metodi nakon 20 perioda *clock*-a. Metoda *post_body()* nam omogućava da izvršimo bilo koju akciju po završetku *body()* metode. U ovom slučaju koristi se sledeći kod:

```
post_body()@sys.any is also {
  start drop_obj();
};
drop_obj()driver.clock is {
  wait [20];
  driver.drop_objection(TEST_DONE);
};
```

Svaka sekvenca može da aktivira i deaktivira prigovore. Simulator će završiti test tek kada se poslednji prigovor spusti. Pošto se telo *slave* sekvence vrti u beskonačnoj petlji, a da se test ne bi zaglavio, *slave* sekvenca niti podiže niti spušta prigovore. Tako je dužina testa određena isključivo *master* sekvencom.

5.2. Iniciranje N2DL eVC-a

Testiranje eVC-a se u ovom slučaju radi bez prisustva DUT-a. Zbog toga, *Verilog* modul *top()*, sadrži sve signale opisane protokolom, ali ne i instancu DUT-a na koju bi pomenuti signali bili povezani. Razvojna tabla upravlja *reset* signalom na sledeći način:

```
initial
  begin
    reset_n =1'b1;
    …
    #23 reset_n = 1'b0;
    #46 reset_n = 1'b1;
  end
```

kao i signalom takta (*clock*) koji je realizovan korišćenjem beskonačne petlje i to kodom:

```
always
  begin: clk_gen
    #25 clk = ~clk;
  end
```

Testiranje se radi tako što se "suprostave" *master* koji predstavlja mrežni sloj i *slave* koji predstavlja sloj linka podataka, jedna naspram drugog. Prvo je potrebno napraviti dve instance eVC

okruženja u *sys* komponenti. Po jedna instanca za svaku od strana N2DL protokola. *Sys* komponenta je osnovna komponenta i predefinisana je, tako da se sve instancira u njoj [9]:

```
extend sys {
  env_maste: ENV_MASTER n2dl_env_u is instance;
  env_slave : ENV_SLAVE n2dl_env_u is instance;
};
```

Scoreboard se kao nezavisna komponenta takođe instancira u *sys* komponenti i povezuje na *monitor* i BFM mrežnog sloja, korišćenjem komande *bind()*. Koristi se sledeći kod:

```
n2dl_scb: n2dl_scoreboard_u is instance;
  keep bind (n2dl_scb.add_n2dl_packet, env_master.agent.as_a(ACTIVE MASTER
  n2dl_agent_u) .bfm.packet_generated);
  keep bind (n2dl_scb.match_n2dl_packet, env_master.agent.monitor.packet_collected);
```

Master instancu je potrebno podesiti da bude aktivna, odnosno da može da upravlja i nadzire signale protokola. To se postiže na sledeći način:

```
extend ENV_MASTER n2dl_agent_u {
  keep kind == MASTER;
  keep active_passive == ACTIVE;
};
```

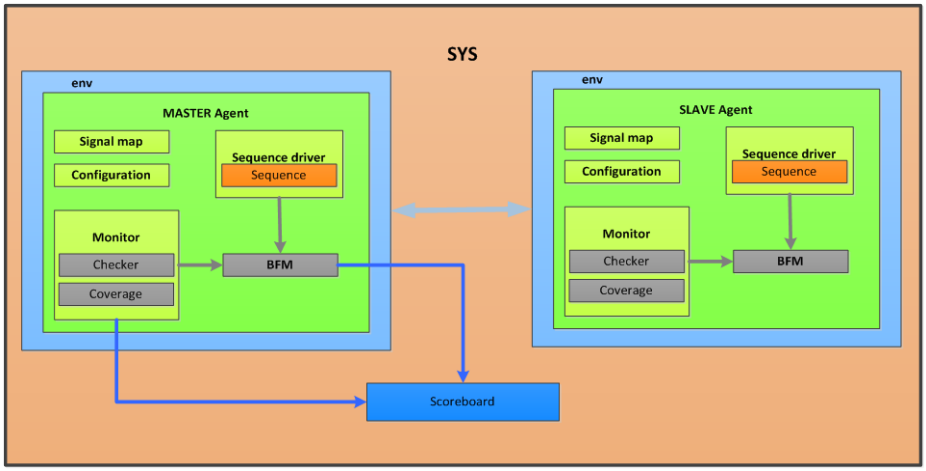
Slave instancu je takođe potrebno konfigurisati da bude aktivna, tj. da može da upravlja i nadzire signale:

```
extend ENV_SLAVE n2dl_agent_u {
  keep kind == SLAVE;
  keep active_passive == ACTIVE;
};
```

Pošto su sve kompnente instancirane, povezane i iskonfigurisane, ostaje još da se eVC signali povežu na HDL signale iz modula *top()*, razvojne table. Sledi primer za *clock* i *reset* signala:

```
extend hdh_n2dl_env_u {
        keep hdl_path() == "~/top";
};
extend hdh_n2dl_signal_map_u {
  keep clk.hdl_path() == "clk";
  keep reset_n.hdl_path() == "reset_n";
…};
```

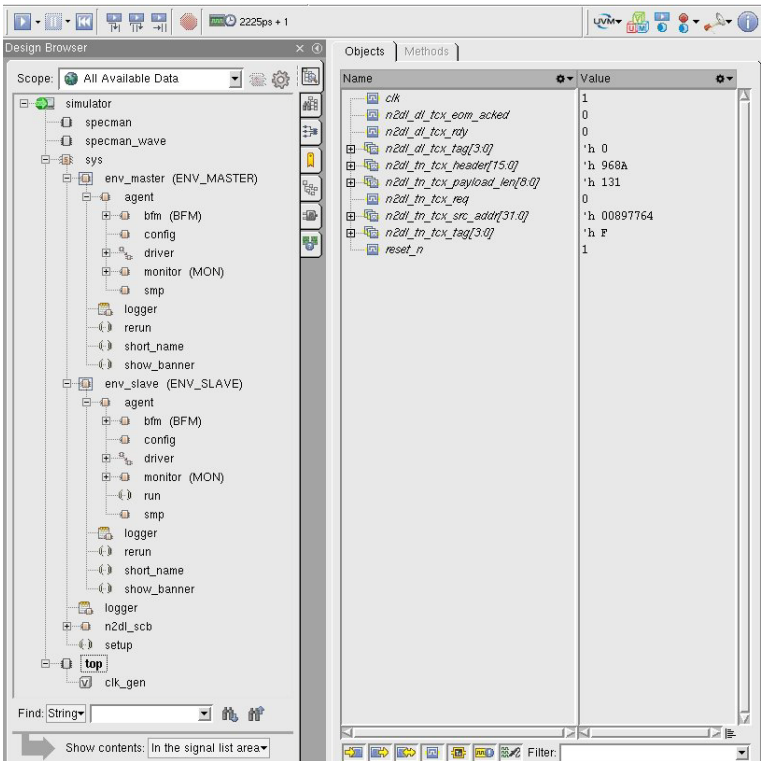
Simbol ~ označava najviši sloj HDL dizajna. On je u HDL svetu ekvivalent *sys* komponenti u *e* svetu. Pod ~ je instanciran *top()* modul. Na slici 5.2.1 je dat prikaz celokupnog verifikacionog okruženja.



Slika 5.2.1. N2DL verifikaciono test okruženje

Pre nego što se pokrene simulacija dizajna u *NC Sim*-u, dizajn je potrebno učitati (*load it*), kompajlirati (*compile*) i elaborirati (*elaborate*). Kompajliranjem koda dizajna dobija se njegova prezentacija u okviru simulatora. *Elaborate* je proces pri kom se dizajn konfiguriše, povezuju se signali i postavljaju se inicijalne vrednosti svih objekata u okviru dizajna. *NC Sim* koristi različite komande u ove svrhe, što zavisi od HDL jezika u kom je DUT, odnosno modul napisan. U ovom slučaju potrebno je učitati *Verilog* modul *top.v*. Više informacija o upotrebi *NC Sim* može se pronaći u [16]. Druga mogućnost je da se napravi *makefile* skripta preko koje će se vršiti učitavanje svih fajlova koji su deo dizajna. Nakon što se učita dizajn, učita se i kompajlira eVC okruženje koje nadalje manipuliše dizajnom.

Na slici 5.2.2 se nalazi prikaz okruženja u simulatoru. U desnoj polovini se nalaze signali protokola, dok je u levoj prikazan hijerarhijski prikaz svih *e* i HDL komponenti.



Slika 5.2.2. N2DL verifikaciono okruženje u simulatoru

Ukoliko bi DUT postojao, on bi bio instaciran pod *top() verilog* modulom i bio bi povezan na signale pomenutog modula. Sa druge strane, samo jedna eVC instanca bi bila instancirana i konfigurisana suprotno od one strane koju DUT predstavlja. Takav slučaj neće biti razmatran u ovom radu.

5.3. N2DL eVC test

Test predstavlja ekstenziju glavne *master* sekvence. Kao što je naglašeno u poglavlju 5.1.9 *slave* sekvenca se vrti u beskonačnoj petlji i njena funkcija je da automatski generiše odziv na pobudu *master* sekvence.

U testu se šalju paketi odnosno transakcije s tim ograničenjem da se polje *addr* koje predstavlja adresu paketa generiše na sledeći način:

.addr == select {

30 : [0..ipow(2,10)-1];

30 : [ipow(2,10)..ipow(2,20)-1];

40 : [ipow(2,20)..ipow(2,32)-1];

};

Šansa da se adresa generiše u opsegu od 0 do 2^{10-1} je 30%.

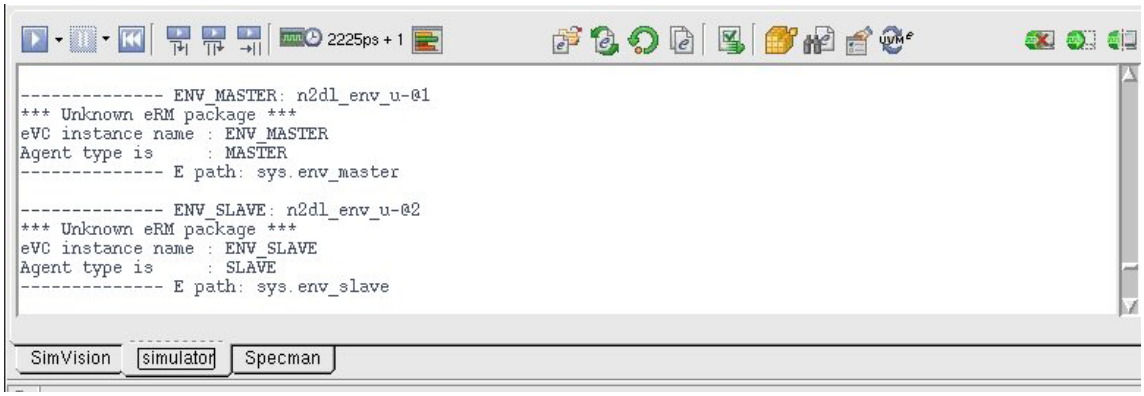
Šansa da se adresa generiše u opsegu od 2^{10} do 2^{20-1} je 30%.

Šansa da se adresa generiše u opsegu od 2^{20} do 2^{32-1} je 40%.

Ove vrednosti za generisanje adresa u različitim opsezima, nasumično su izabrane kao primer da je moguće vršiti takva podešavanja. Sva ostala polja se generišu nasumično u skladu sa svojim ograničenjima definisanim u *hdh_n2dl_packet_s.e* fajlu. Broj paketa koji će biti generisan, određuje se *for* petljom.

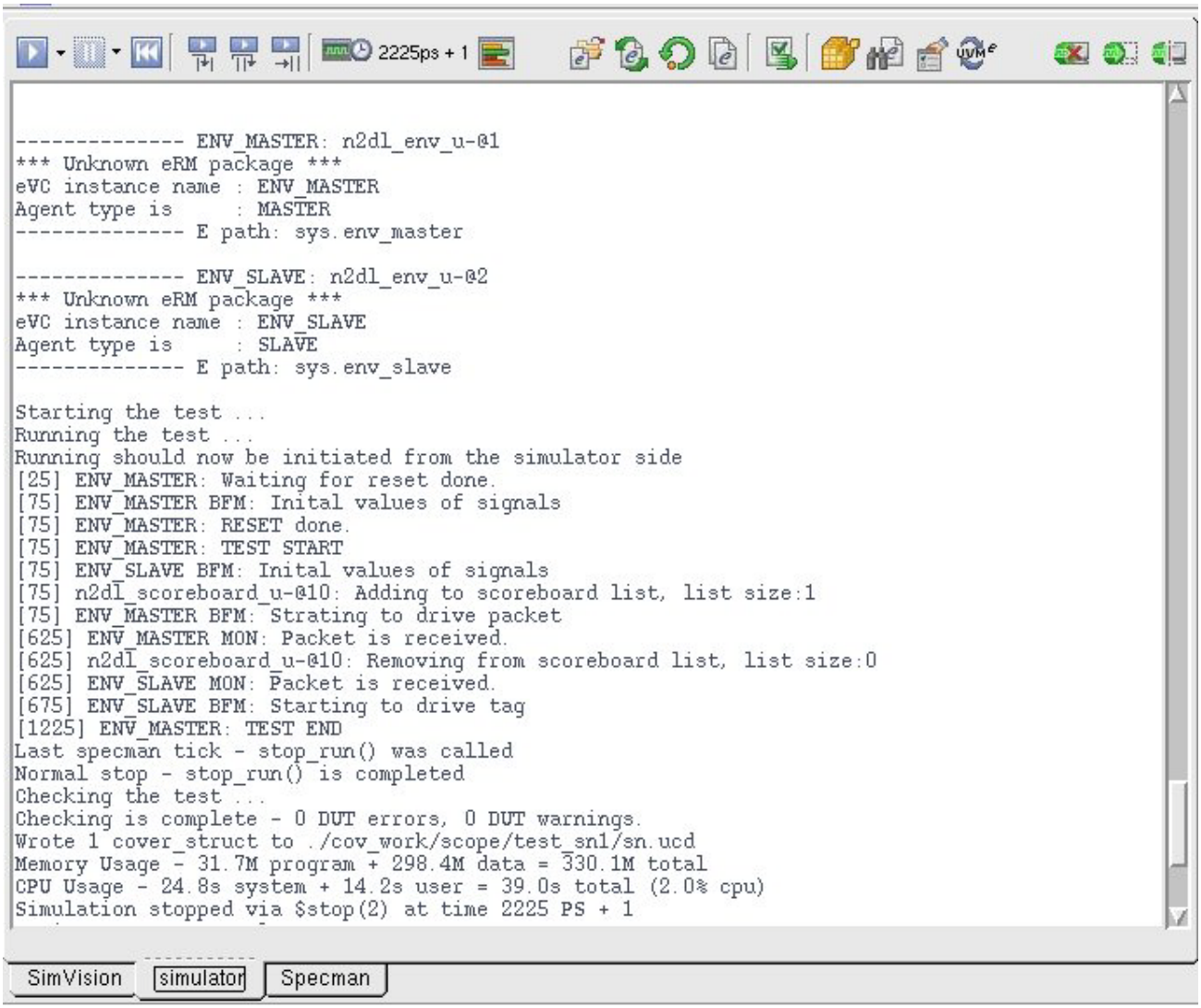
5.3.1. N2DL eVC test – rezultati

Na slici 5.3.1.1 je dat prikaz *Specman* konzole pre nego što se test pokrene. Ispisane su sve instance eVC-a koje se koriste u simulaciji kao i deo konfiguracije istih. Pokretanje simulacije se vrši klikom na taster *run* koji se nalazi u gornjem levom uglu.



Slika 5.3.1.1. Izgled *Specman* konzole pre pokretanja testa

Prvo će biti pokrenut test koji šalje na signale samo jedan paket, odnosno transakciju. Na slici 5.3.1.2 je prikazana *Specman* konzola po završetku testa.

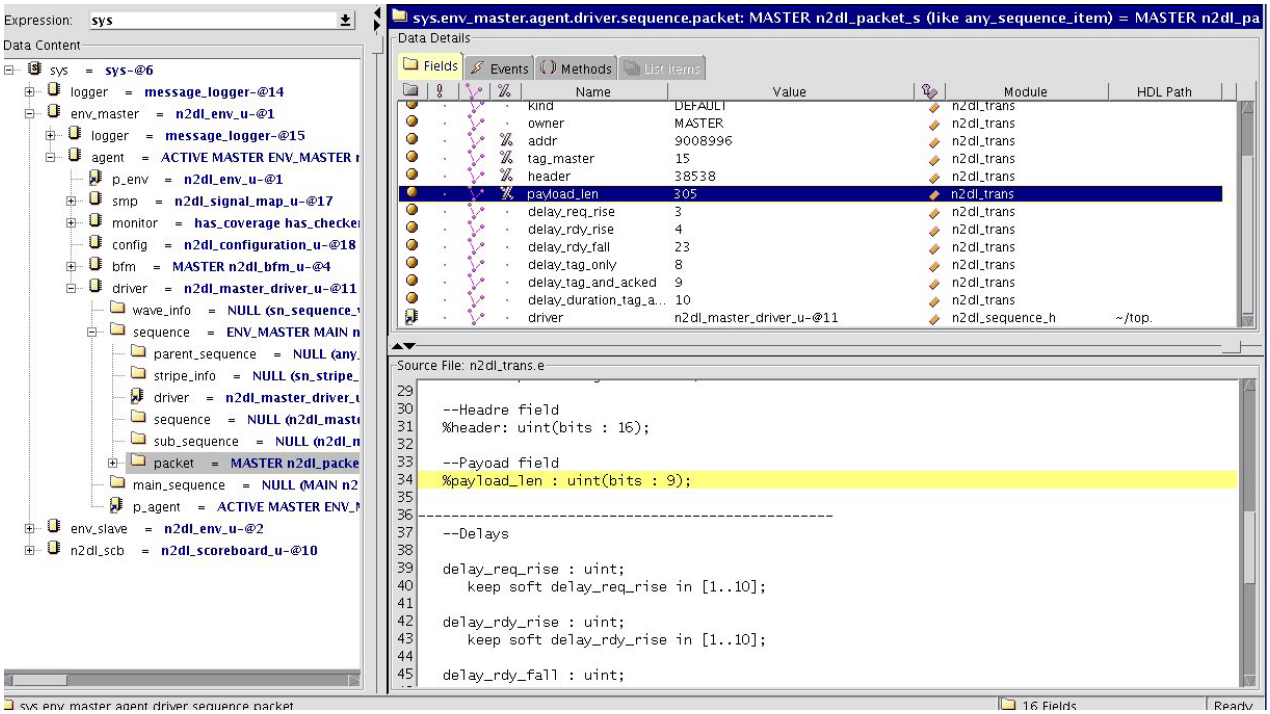


Slika 5.3.1.2. Izgled *Specman* konzole nakon pokretanja testa sa jednim paketom

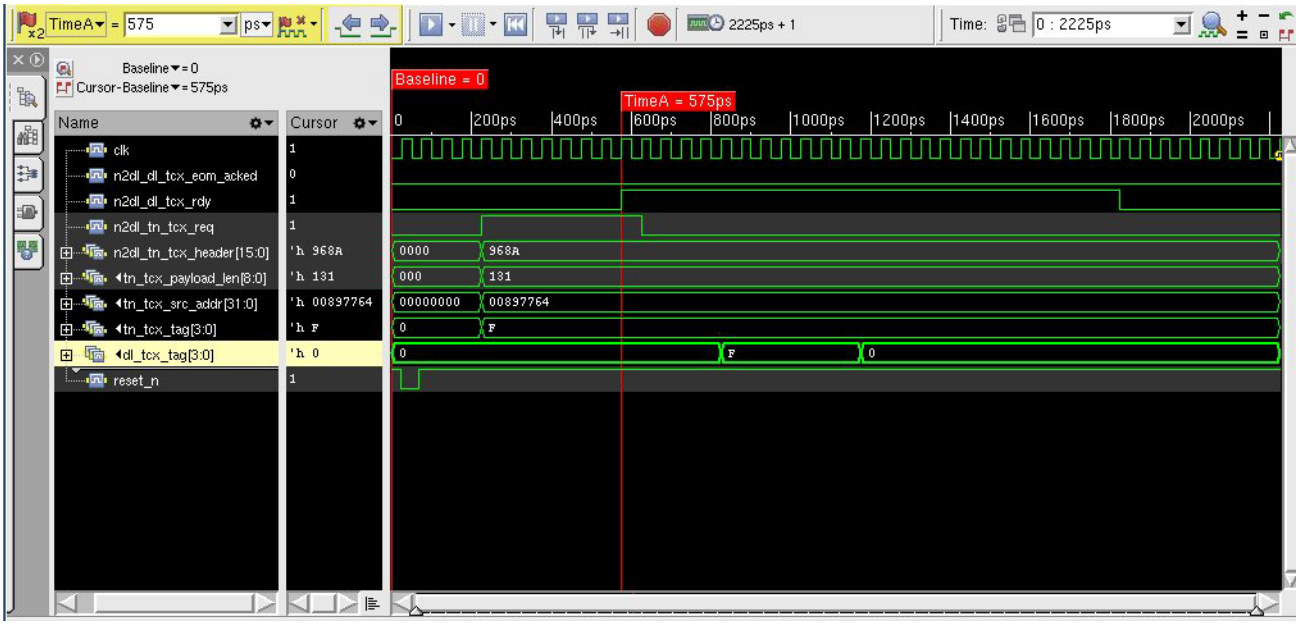
U konzoli se ispisuju poruke iz komponenti eVC-a i time se može lako pratiti tok simulacije. Prvo se čeka da se *reset* završi. Zatim test počinje. BFM mrežnog sloja šalje paket na *scoreboard* i zatim isti počinje da ga šalje na signale. *Scoreboard* stavlja paket u listu. *Monitor* mrežnog sloja skuplja paket sa "žica" i isti prosleđuje *scoreboard*-u. Ovaj zatim upoređuje paket iz *monitor*-a sa onim dobijenim iz BFM-a i skida ga sa liste. *Monitor* sloja linka podataka skuplja paket sa žica i BFM sloja linka podataka počinje da šalje svoj odziv na "žice".

Brojevi u uglastim zagradama ispred poruka predstavljaju vreme izraženo u piko sekundama. Test je uspešno završen ukoliko je broj DUT grešaka jednak nuli što je ovde i slučaj.

Generisan paket je moguće videti u posebnom prozoru alata. Izgled paketa je dat na slici 5.3.1.3, dok je na slici 5.3.1.4 prikazan izgled signala po završetku testa.



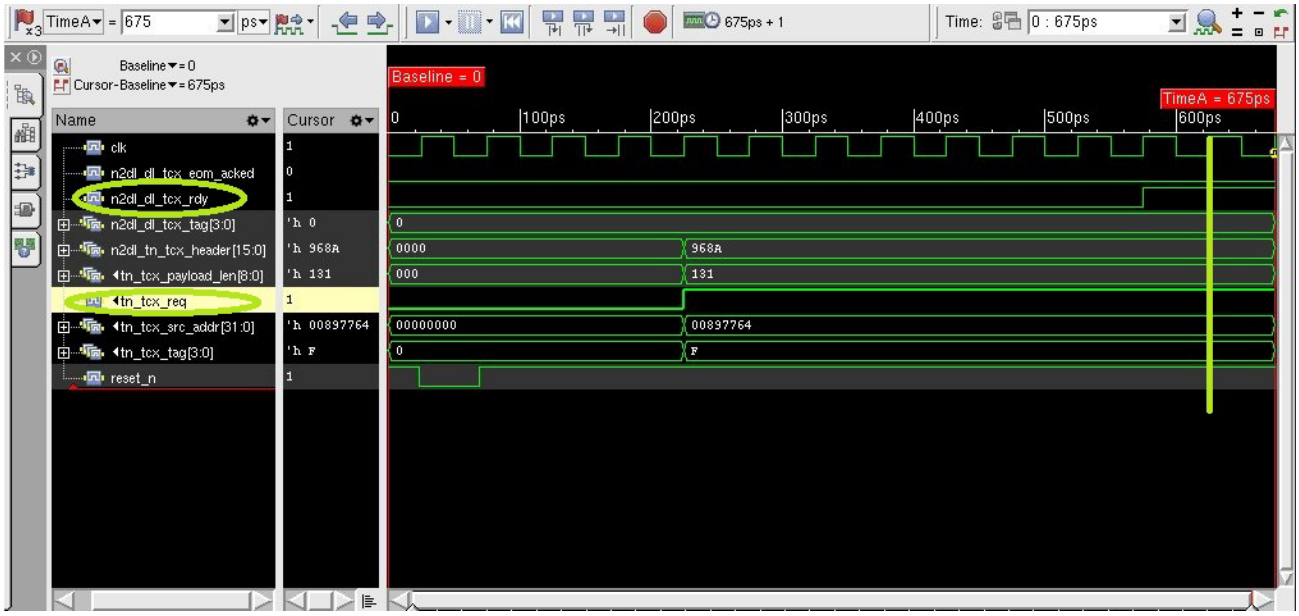
Slika 5.3.1.3. Prikaz generisanog paketa



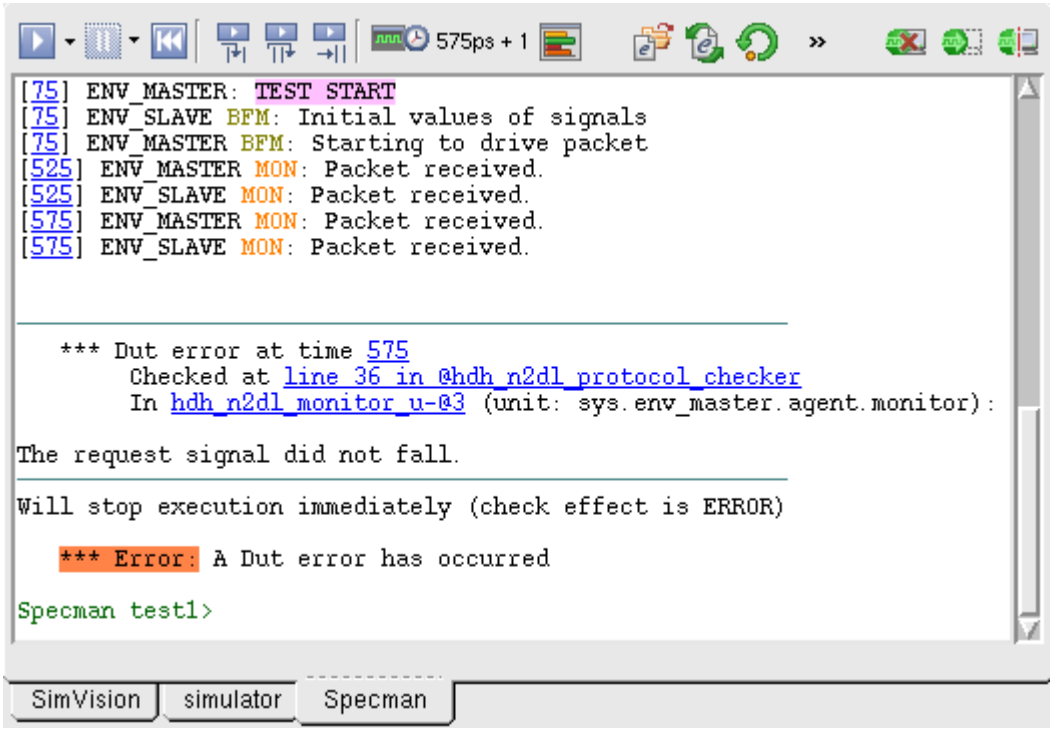
Slika 5.3.1.4. Prikaz N2DL protocol signala za jedan paket

Generisan paket biva, u skladu sa protokolom, postavljen na signale. Tako na primer, vrednost polja transakcije zaglavlja na slici 5.3.1.4 iznosi 968A i on biva postavljen od strane mrežnog sloja tek pošto se *n2dl_tn_tck_req* signal setuje (slika 5.3.1.4). S druge strane, sloj linka podataka vraća svoju oznaku nakon što je postavio *n2dl_dl_tcx_rdy* signal (slika 5.3.1.4), dok signal *n2dl_dl_tcx_eom_acked* ostaje nepromenjen budući da je vrednost EOM bita, najnižeg bita u zaglavlju, jednaka „0”.

Ukoliko dođe do greške, protokol kontrolor će odmah signalizirati grešku i zaustaviti simulaciju u trenutku nastanka greške. Na slikama 5.3.1.5 i 5.3.1.6 signal *n2dl_tn_tck_req* se ne obara tačno jedan takt period nakon što je *slave* setovao *n2dl_dl_tcx_rdy* signal.



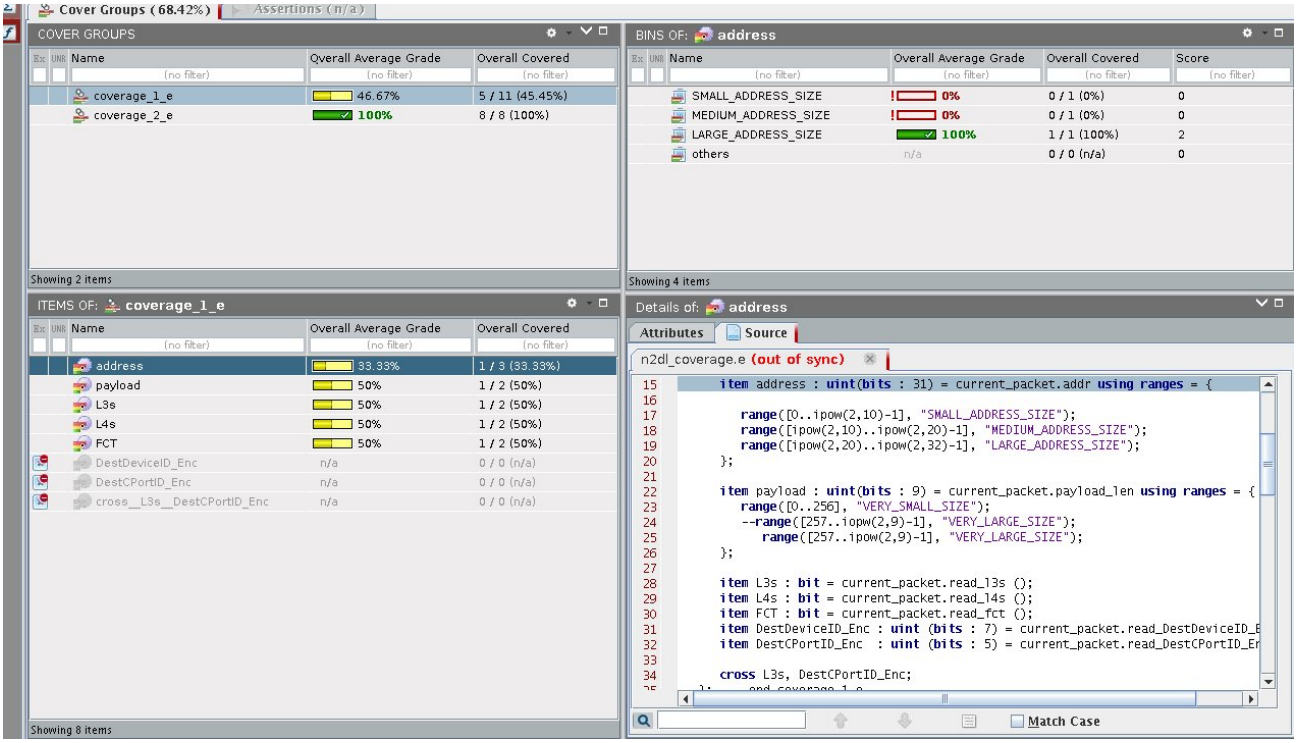
Slika 5.3.1.5. Prikaz signala zaustavljene simaulacije od strane *checker*-a



Slika 5.3.1.6. Izgled konzole u kojoj je signalizirana greška od strane *checker*-a

Zelena linija na slici 5.3.1.5 pokazuje trenutak kada je signal *n2dl_tn_tck_req* trebao da se obori. Na sledećoj rastućoj ivici signal takta kontrolor je „pao" zaustavljajući simulaciju.

Nakon uspešno završenog testa dolazi do skupljanja *coverage*. Na slici 5.3.1.7 je prikazan izgled *coverage* kada se test sastoji od jednog paketa.

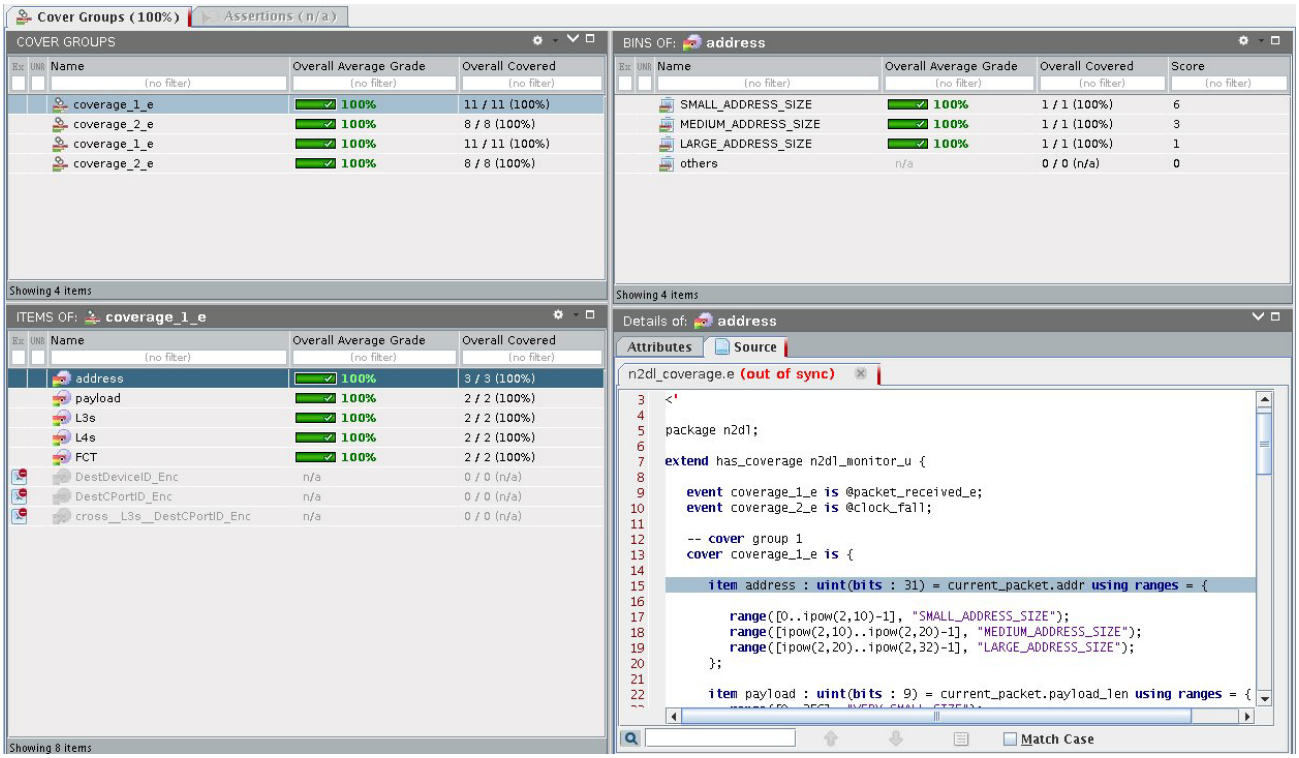


Slika 5.3.1.7. *Coverage* kada je poslat jedan paket

Na slici se vide četiri prozora, dva na desnoj i dva na levoj strani. U levom delu slike se nalaze *coverage groups*, dok se ispod njih nalaze njihovi *coverage items* markirane grupe. Desno na slici su informacije koje vrednosti *coverage items* su pogođene (zelena boja), a koje su promašene (crvena boja). U prozoru ispod nalazi se *source*, odnosno kod kojim su definisane u okviru eVC.

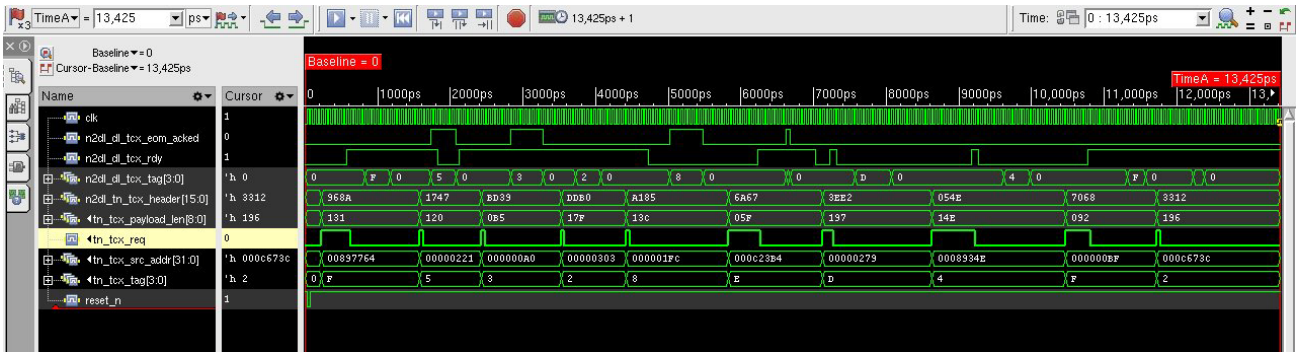
Funkcionalnost eVC-a u potpunosti je testirana kada je pokrivenost 100%. Posle testa od jednog paketa vrednost funkcionalne pokrivenosti je 46,67%. Zaključak je da jedan paket nije dovoljan da bi se testirala celokupna funkcionalnost N2DL eVC protokola.

Sada će biti posmatran test koji sadrži 100 paketa koji se šalju na signale jedan za drugim. *Coverage* nakon ovakvog testa je prikazan na slici 5.3.1.8.



Slika 5.3.1.8. *Coverage* kada je poslato sto paketa

Sada je funkcionalna pokrivenost 100% te je kompletna funkcionalnost N2DL eVC-a testirana. Na slici 5.3.1.9 je prikazan izgled protokol signala za 100 nasumično generisanih paketa. Da se primetiti da u slučaju kada je EOM bit zaglavlja jednak „1”, *slave* osim oznake setuje i *n2dl_dl_tcx_eom_acked* signal.



Slika 5.3.1.9. Prikaz dela N2DL protokol signala za sto paketa

6. ZAKLJUČAK

Slučajna verifikacija se pokazala kao najbolji pristup za verifikaciju što se tiče kvaliteta i redukovanja vremena verifikacije. eRM metodologija omogućava da se na lak i brz način pišu testovi, da se lako uočavaju i lociraju greške testiranog uređaja i da postoji visok stepen ponovne upotrebe postojećih verifikacionih komponenti. eVC komponenta je najbolji primer eRM metodologije.

N2DL eVC predstavlja verifikacionu komponentu koja simulira obe strane N2DL protokola. Sam eVC može lako da se konfiguriše da predstavlja kako mrežni sloj, tako i sloj linka podataka. Putem sekvenci testovi se veoma lako pišu. Sekvence kroz upravljač sekvenci bivaju prosleđene BFM-u koji transakcije u skladu sa protokolom šalje na signale.

eVC u sebi sadrži monitor, koji u sebi ima pokrivenost i protokol kontrolore. Da bi funkcionalna pokrivenost bila 100% test mora da sadrži više od jednog paketa. Pošto se paketi generišu slučajno, može se desiti da se isti, ili pak sličan paket, generiše više puta. Zbog toga nije moguće odrediti tačan broj paketa potrebnih da se pokrije kompletna funkcionalnost. Empirijskim putem je utvrđeno da se sa 73 paketa, u jednom slučaju, može dobiti funkcionalna pokrivenost od 100%. U ovom radu koristi se 100 paketa koji se sekvencijalno šalju, što je sasvim dovoljno. Kontrolori nadgledaju signale i ukoliko dođe do narušavanja protokola, zaustavljaju simulaciju i signaliziraju grešku, tačno na mestu gde je ona nastala, štedeći tako dragoceno vreme u procesu traženja greške.

Ukoliko postoje, na primer, dva DUT-a koja međusobno komuniciraju putem N2DL protokola, eVC je moguće konfigurisati da bude pasivan. U tom slučaju on će samo nadgledati signale protokola i signalizirati greške ukoliko postoje.

Sve to daje veliku fleksibilnost N2DL eVC-u, a to je veoma bitno kada se razvija verifikaciono okruženje, bilo da je N2DL eVC sam za sebe verifikaciono okruženje (što je u ovome master radu i slučaj), ili je deo nekog većeg verifikacionog okruženja.

Pošto je vreme novac, a 70% vremena funkcionalnog razvoja integrisanog kola se potroši na verifikaciju, onda nam uvođenje eRM metodologije i eVC-a drastično smanjuje troškove projektovanja i razvoja čipa, a to opet utiče na nižu cenu finalnog proizvoda.

Cilj ovog master rada je da se ukaže na značaj *random* verifikacije i da se napravi funkcionalan i praktičan primer verifikacione komponente. Mišljenje autora je da ova tema nije dovoljno obrađena na tehničkim fakultetima, iako ima ogroman potencijal. Na relativno jednostavnom protokolu prikazani su skoro svi aspekti ovog tipa verifikacije koji će budućem čitaocu dati solidnu osnovu kako u teorijskom, tako i u praktičnom smislu.

LITERATURA

[1] Guy Mosensoson, „Practical Approaches to SOC Verification”, Verisity Design, Inc, 2002.
[2] J. A. Abraham, „Verification of SoC Designs”, UT Austin, ECE Department, 2010.
[3] „Specman Basic Training”, interni dokument, HDL Design House, Beograd, 2009.
[4] „Open Verification Methodology User Guide” Product Version 2.0.1, Cadence, oktobar 2008.
[5] „UVM COOKBOOK ARTICLES”, Mentor Graphics Corporation, 2012. Available: https://verificationacademy.com/cookbook
[6] Wikipedia, „eRM”, 5 December 2012. Available: https://en.wikipedia.org/wiki/ERM_(e_Reuse_Methodology)
[7] „Specman e Language Reference” Product Version 9.2, Cadence, decembar 2009.
[8] http://www.design-reuse.com/articles/4617/maximizing-verification-productivity-evc-reuse-methodology-erm.html
[9] „e Reuse Methodology (eRM) Developer Manual” Version 4.3.5, Verisity Design, Inc, 2004.
[10] „Network-to-Data Link (N2DL) Interface", interni dokument, HDL Design House, Beograd
[11] Wikipedia, „Specman”, 5. februar 2014. Available: http://en.wikipedia.org/wiki/Specman
[12] Wikipedia, „NCSim”, 11. avgust 2014. Available: http://en.wikipedia.org/wiki/NCSim
[13] Sasan Iman, Sunita Joshi, "The e Hardware Verification Language", Springer, 2004.
[14] „Preliminary e Language Reference Draft“, IEEE Standard Draft, 2003.
[15] Samir Palnitkar, „Design Verification with E“, Prentice Hall PTR,2003.
[16] „Cadence NC-Verilog Simulator Tutorial“, Cadence Design Systems, Inc, September 2003