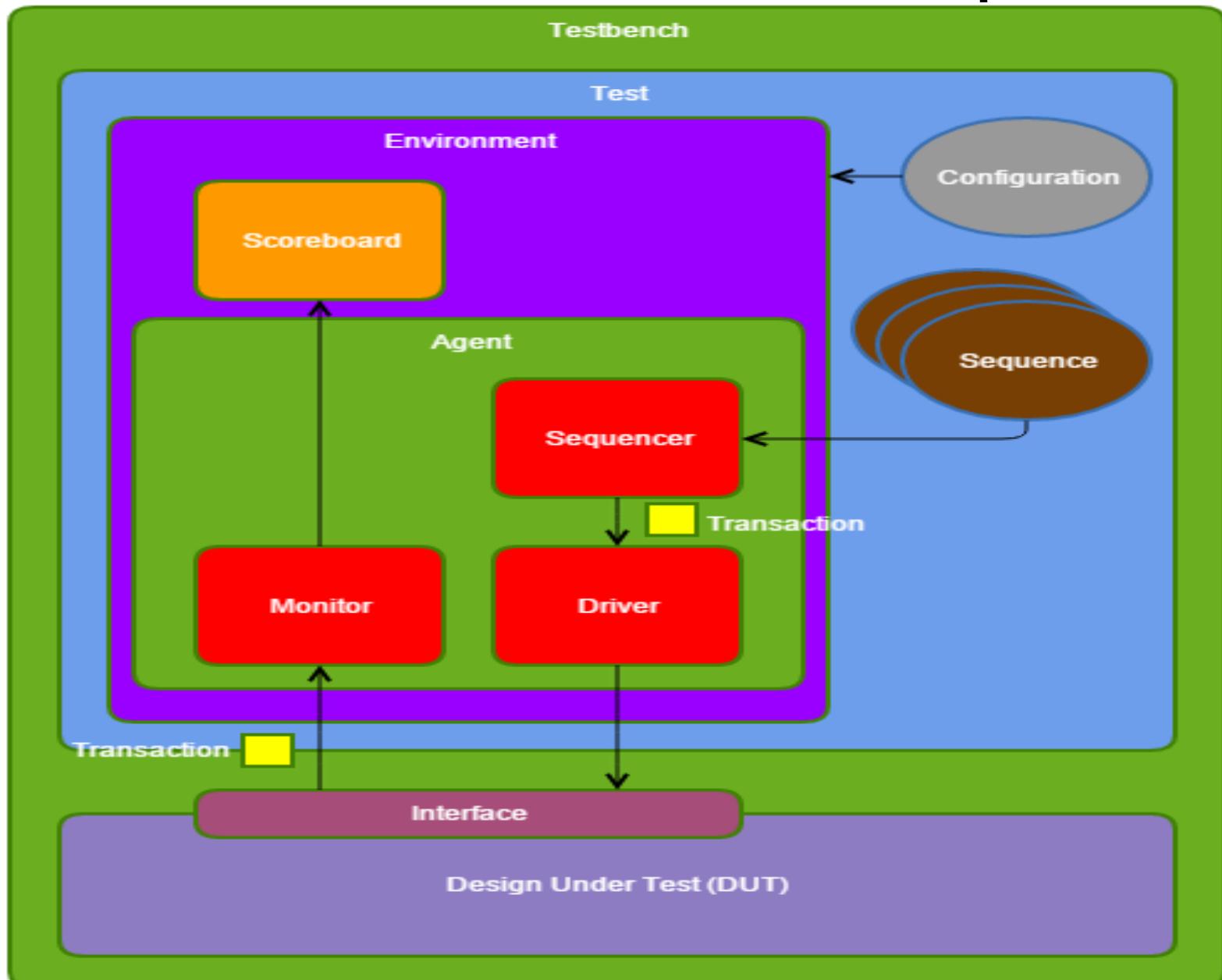


UVM

Osnovni koncepti
sa hijerarhijom klasa

UVM osnovni koncept



Osnovna ideja

- Osnovna ideja UVM-a obuhvata modularno razdvajanje sistemski bitnih scenarija ispitivanja od signalnog nivoa DUT-a.
 - Ukoliko imamo prožimanje oba ova segmenta sistemskog i signalnog, tada teško možemo modularno proširivati, nasleđivati, kapsulirati...
 - Ukoliko uspemo da razdvojimo signalni nivo od sistemskog, na sistemskom planu nam se otvara široka perspektiva prednosti OOP-a

Blokovi koji sačinjavaju osnovni koncept

- Testbench
- Test
- Environment
- Scoreboard
- Agent
- Sequencer
- Sequence
- Driver
- Monitor

UVM Testbench

- UVM Testbench tipično instancira Modul koji se testira (DUT) i UVM test klasu, takođe konfiguriše veze između njih.
- Bitno je naglasiti da je UVM test dinamičkiinstanciran u vreme izvršavanja (run time), dozvoljavajući da se jednom kompajlirani UVM Testbench pokrene više puta i izvrši različite testove.

UVM Test

- UVM test je UVM komponenta najvišeg nivoa u okviru UVM Testbencha.
- UVM test obično realizuje tri najbitnije funkcije:
 - Instancira okruženje najvišeg nivoa,
 - konfiguriše okruženje (preko fabrike-factory ili konfiguracione baze podataka),
 - priprema stimulus pozivanjem UVM sekvenci preko okruženje za DUT.
- Tipično, postoji jedan osnovni UVM test sa UVM okruženjem i drugim osnovnim stavkama. Potom, drugi pojedinačni testovi će proširiti ovaj osnovni test i konfigurisati okruženje drugačije ili odabrati različite sekvene za pokretanje.

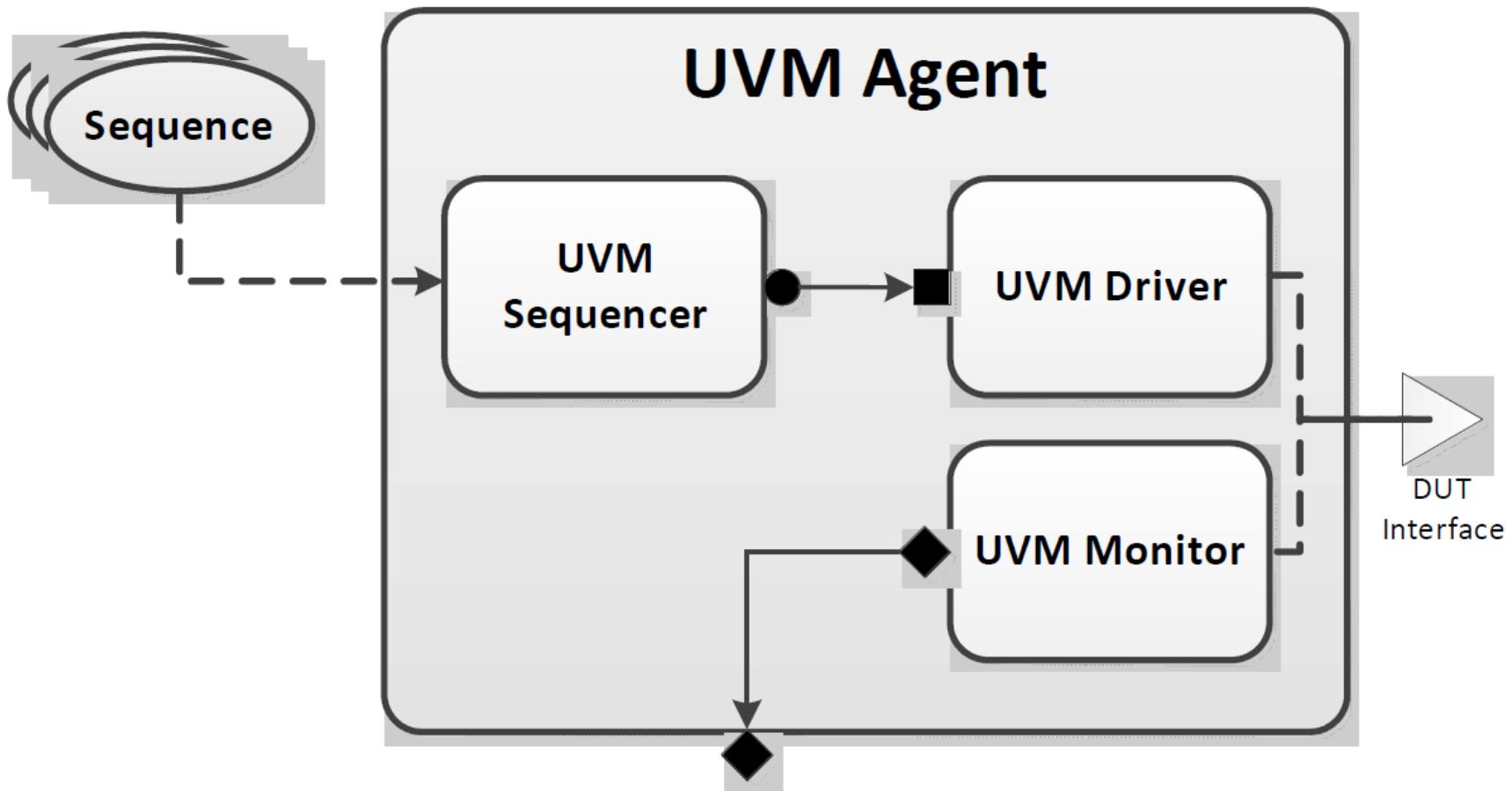
UVM Environment

- UVM Environment (okruženje) je hijerarhijska komponenta koja grupiše ostale verifikacione komponente.
- Tipične komponente koje se obično instanciraju unutar UVM okruženja su UVM Agenti, UVM Scoreboards, ili čak i druga UVM okruženja.
- UVM okruženje najvišeg nivoa enkapsulira sve komponente verifikacije koje interreaguju sa DUT-om.
 - Na primer: U tipičnom sistemu na čipu (SoC) u UVM okruženju, nalazi se jedno UVM okruženje po IP-ju (npr. PCIe okruženje, USB okruženje, okruženje kontrolera memorije, itd.).
 - Ponekad su ta IP okruženja grupisana zajedno u klaster okruženja (npr., IO okruženje, procesorsko okruženje, itd.)

UVM Scoreboard

- Glavna funkcija UVM Scoreboard-a je da proveri ponašanje određenog DUT-a.
 - Semafor za UVM obično prima transakcije koje prenose ulaze i izlaze DUT-a kroz UVM Agent portove za analizu,
 - izvršava ulazne transakcije kroz namenski referentni model (poznatiji kao prediktor) za generisanje očekivanih transakcija, a zatim upoređuje očekivane izlaze u odnosu na stvarni izlaz.
 - Postoje različite metodologije za implementaciju Scoreboarda, referentnog modela, i kako komunicirati između semafora i ostatka testbenča.

UVM Agent



UVM Agent

- UVM Agent je hijerarhijska komponenta koja grupiše druge verifikacione komponente koje su vezane za specifični DUT interfejs.
- Tipični UVM agent uključuje
 - UVM Sekvencer koji upravlja tokom stimulusa,
 - UVM Driver koji dati stimulus pretvara u signalnu stimulaciju na DUT interfejsu
 - UVM Monitor koji nadgleda DUT interfejs.
- UVM Agenti mogu sadržati i druge komponente, kao što su Coverage Collectori, Protocol checkers itd.

UVM Sequencer

- UVM sekvencer služi kao arbitar za kontrolu toka transakcija odnosno (Sequence item-a) iz višestrukih sekvenci stimulusa.
- Konkretnije, UVM sekvencer kontroliše protok UVM sekvenci transakcija generisanih od jedna ili više UVM sekvenci.

UVM Sequence

- UVM sekvenca je objekat koji sadrži definisani algoritam za generisanje stimulusa.
- UVM Sekvence nisu deo hijerarhije komponenti.
- UVM sekvence mogu da se kreiraju za jednu transakciju.
- UVM sekvence mogu da se funkcionišu hijerarhijski sa jednom izvornom sekvencom, iz koje se formiraju izvedene sekvence.
- Tipično UVM sekvenca je vezana za UVM sekvencer. Višestruka UVM sekvenca može biti vezana za isti UVM sekvencer.

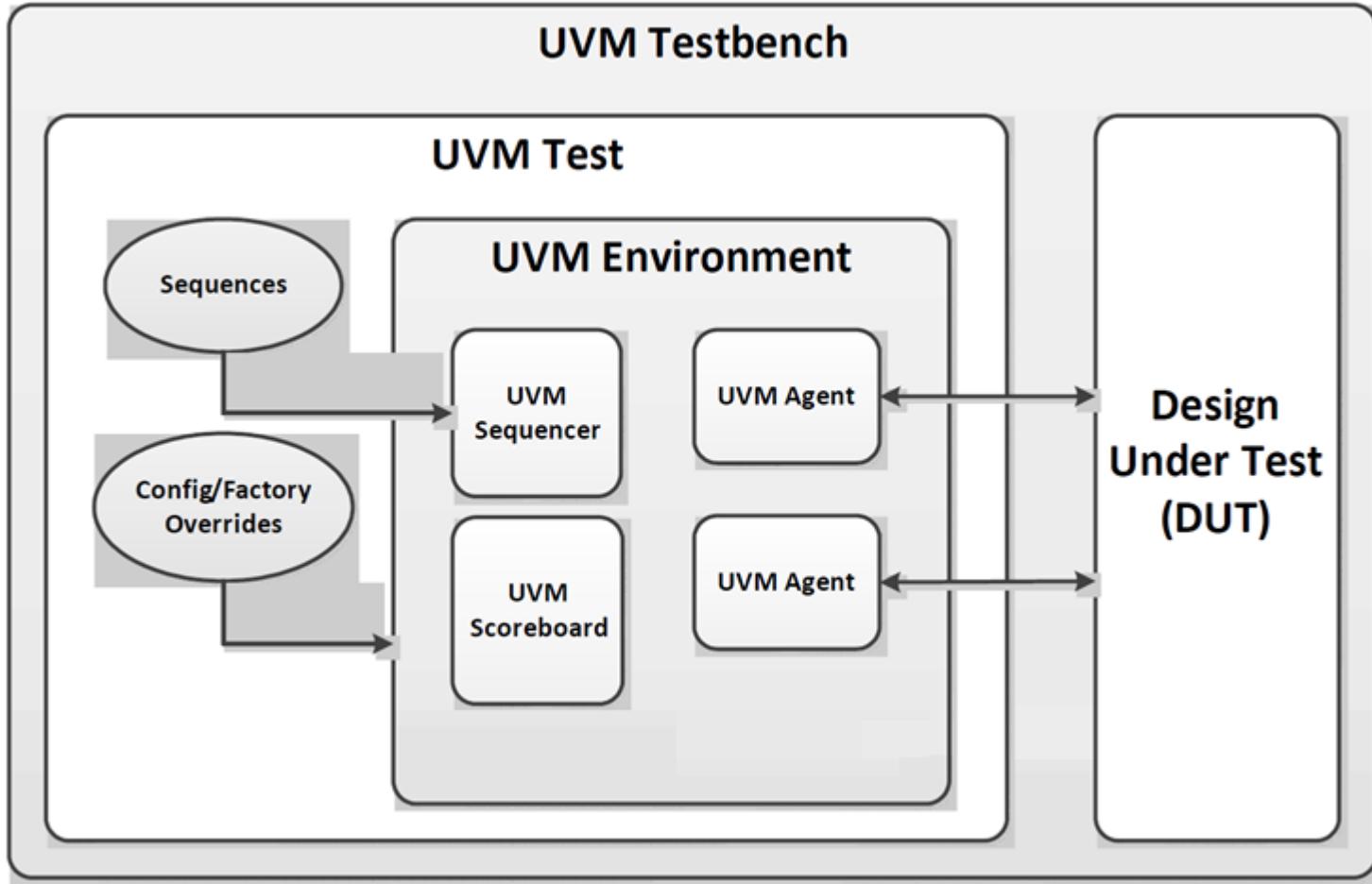
UVM Driver

- UVM Driver prima pojedinačne transakcione UVM sekvence od UVM sekvencera i iz njih formira signalne telegramme na DUT interfejsu.
- UVM drajver povezuje nivo apstrakcije sa signalnim nivoom.
- Pretvara stimulus na nivou transakcija u stimulus na nivou signala, odnosno pinova čipa.
- Poseduje TLM port za primanje transakcija iz Sekvencera.

UVM Monitor

- UVM Monitor sempluje DUT interfejs i preuzete informacije pretvara u transakcije koje šalje ostatku UVM Testbencha za dalju analizu.
- Sistemski, slično driveru, ali u suprotnom smeru, UVM Monitor od signalnih aktivnosti na pin nivou DUT-a formira transakcije.
- Dakle i UVM Monitor povezuje signalni nivo sa nivoom aprstrakcije.
- UVM Monitor obično ima pristup DUT interfejsu, dok sa druge strane ima takozvani TLM analysis port za emitovanje kreirane transakcije.
- UVM Monitor može interno da izvrši neke obrade nad formiranim transakcijama (kao što je pokrivenost, provera, logovanje itd.) ili to može da se prenese na namenske komponente povezane na njegov analysis port .

Cilj je UVM Testbench



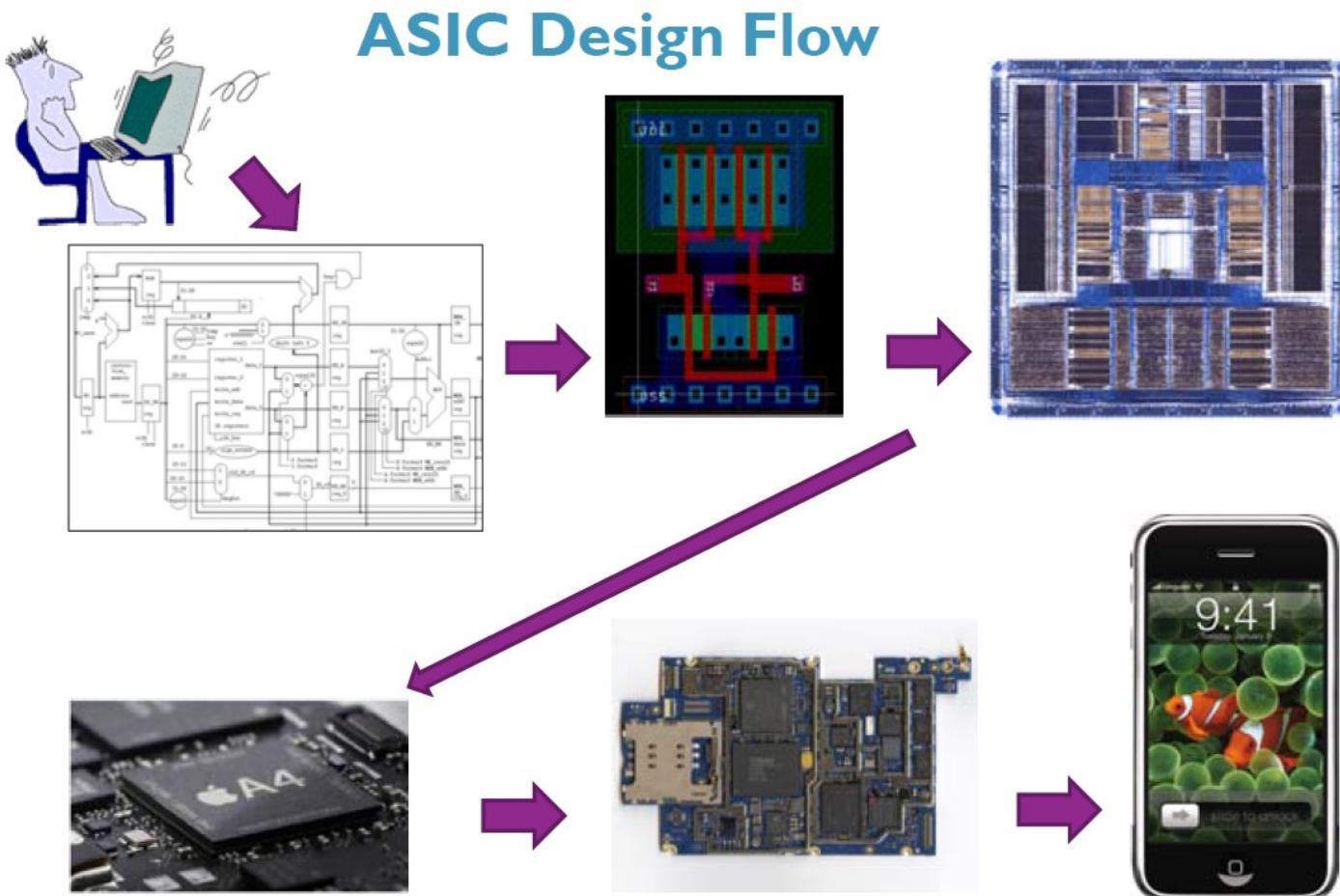
Zbog čega se insistira na šablonu?

- Ovakav UVM specifični šablon pokriva danas dominantne složene sisteme u čipu SOC sa velikim brojem komunikacionih portova različitog tipa
- Uspešno razdvaja apstraktni sistemski od niskog signalnog nivoa i uspeva da rukuje sa oba.
- Uspešno savladava zahtevana proširenja sistema, komunikacionih portova, scenarija upotrebe, velika ponovna upotrebljivost (reusability).
- Omogućava potpune promene scenarija ispitivanja u toku izvršavanja simulacije, bez potrebe za ponovnim kompajliranjem.

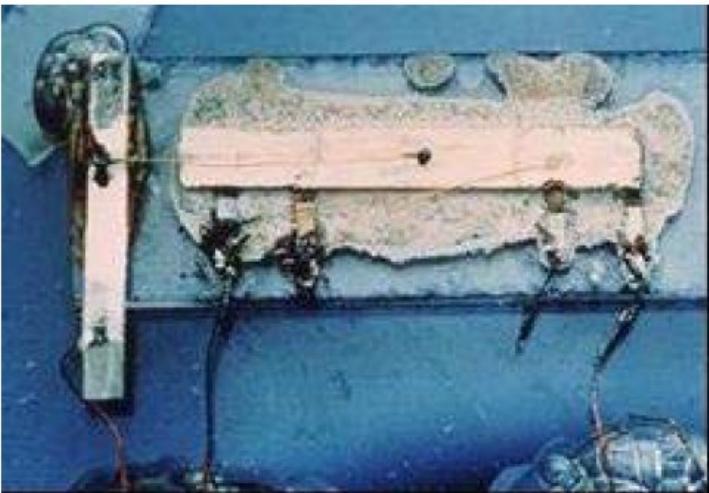
Zašto uopšte idemo u OOP pravcu u RTL verifikaciji?

- Tradicionalni VHDL / Verilog testbenčevi jesu dovoljno dobri da verifikuju jednostavan dizajn kao što je 8-bitni ALU. Čak i za nešto komplikasnije projekte sa jednostavnijim komunikacionim spregama, kao što su I2C, SPI, UART...
- Problem se javlja kada se verifikuju složeni projekti, kao što su Ethernet switchevi sa stotinama portova, ili procesori sa više nivoa keš memorije.
- Tada treba implementirati složene prediktore, opsežne alate za pokrivanje i snažne generatore stimulusa.
- Potrebno ih je skalirati od testa jedinice do testa sistema, i potreban je veliki tim inženjera koji će raditi zajedno na ovakvim projektima.
- Konačno, sve ovo se mora ponovo koristiti, jer je previše posla pisanje novih testenčeva od nule za svaki projekat.

Tok ASIC projekta



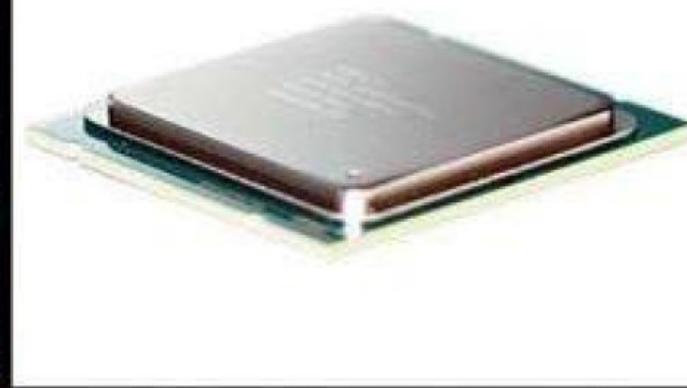
Integrисана кола некад i sad...



First Integrated Circuit (1958)

10 mm x 1.5 mm

1 transistor



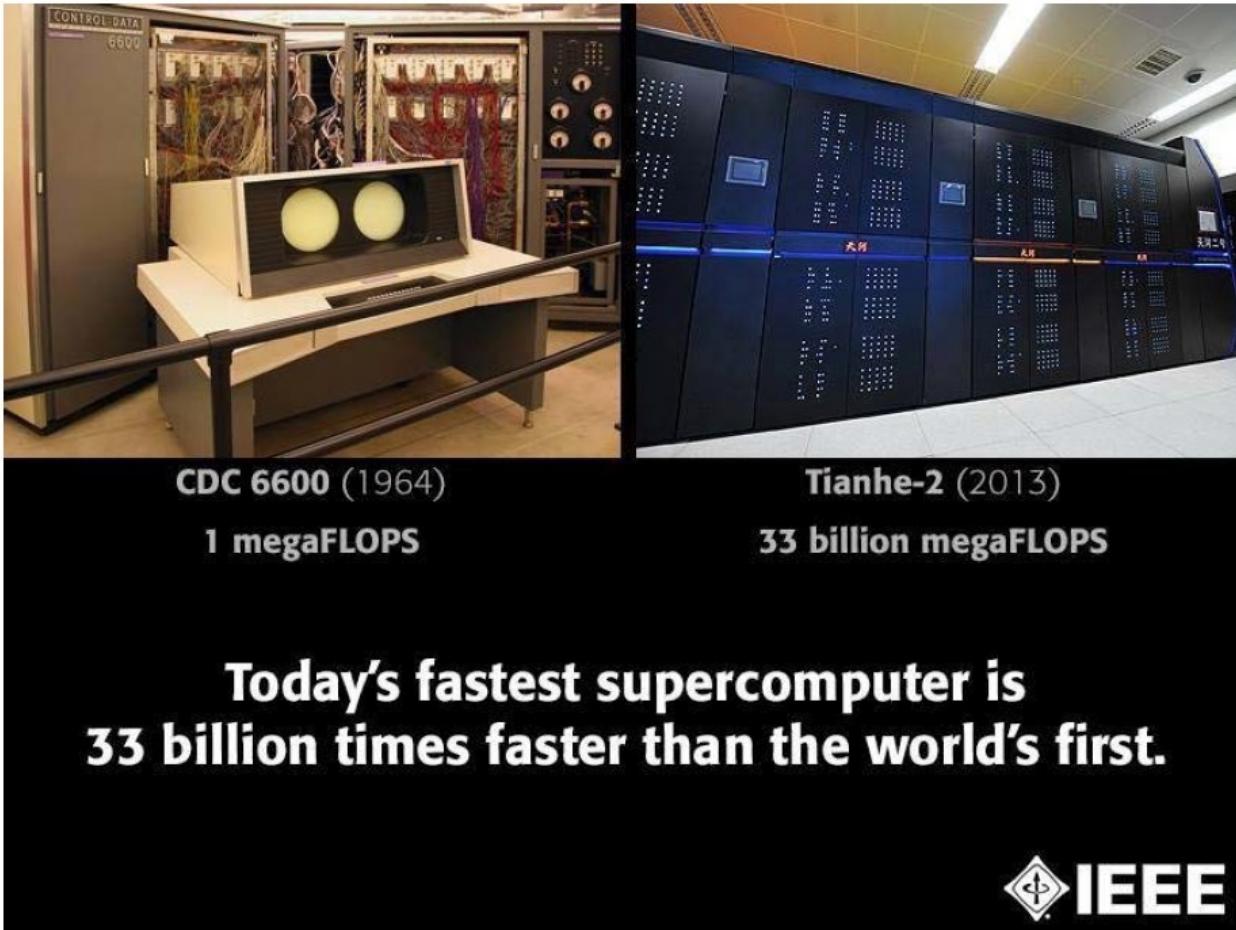
Best Commercial CPU (2014)

541 mm² die size

4.31 billion transistors

Over 56 years, innovation has increased the performance of a chip 4 billion times.

Računari nekad i sad



Nekada davno...

**\$3398
10MB**

**THE HARD DISK
YOU'VE BEEN
WAITING FOR**

**10 Megabyte Hard Disk
\$3,495***

COMPUTER COMPONENTS

More software
is included with the system than ever before, including CP/M, BASIC or automatic
monitoring, 16-bit graphics, 16-bit sound, and much more. Software packages include:
The
Ultimate Data System, BASIC, CP/M, and many others.

© 1981 Computer Components Inc. All rights reserved. No part of this publication may be reproduced without written permission from the publisher.

the 10-Megabyte Computer System

**Only
\$5995
COMPLETE**

New From IMSAI®

- 10-Megabyte Hard Disk
- 5 1/4" Dual-Density Floppy Disk Back-up
- 8-Bit Microprocessor
(Optional 16-bit Microprocessor)
- Memory-Mapped Video Display Board
- Disk Controller
- Standard 64K RAM
(Optional 256K RAM)
- 10-Slot S-100 Motherboard
- 28-Amp Power Supply
- 12" Monitor
- Standard Intelligent 62-Key ASCII Keyboard (Optional Intelligent 86-Key ASCII Extended Keyboard)
- 132-Column Dot-Matrix Printer
- CP/M® Operating System

*You Read It Right ...
All for \$5995!*

IMSAI® ...Thinking ahead for the 80's

415/635-7615

Computer Division of the Fischer-Freitas Corporation
910 81st Avenue, Bldg. 14 • Oakland, CA 94621

*CP/M is a trademark of Digital Research. Imsai is a trademark of the Fischer-Freitas Corporation

Džepni računari

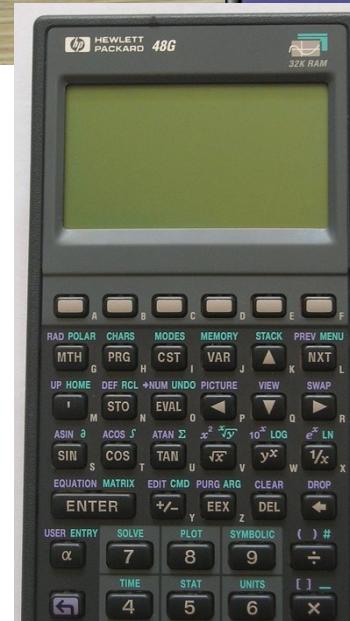
SCRIBOLA

Die kleinste sichtbare schreibende
Addier- u. Subtrahier-Maschine

* Gewicht nur 2,3 kg *

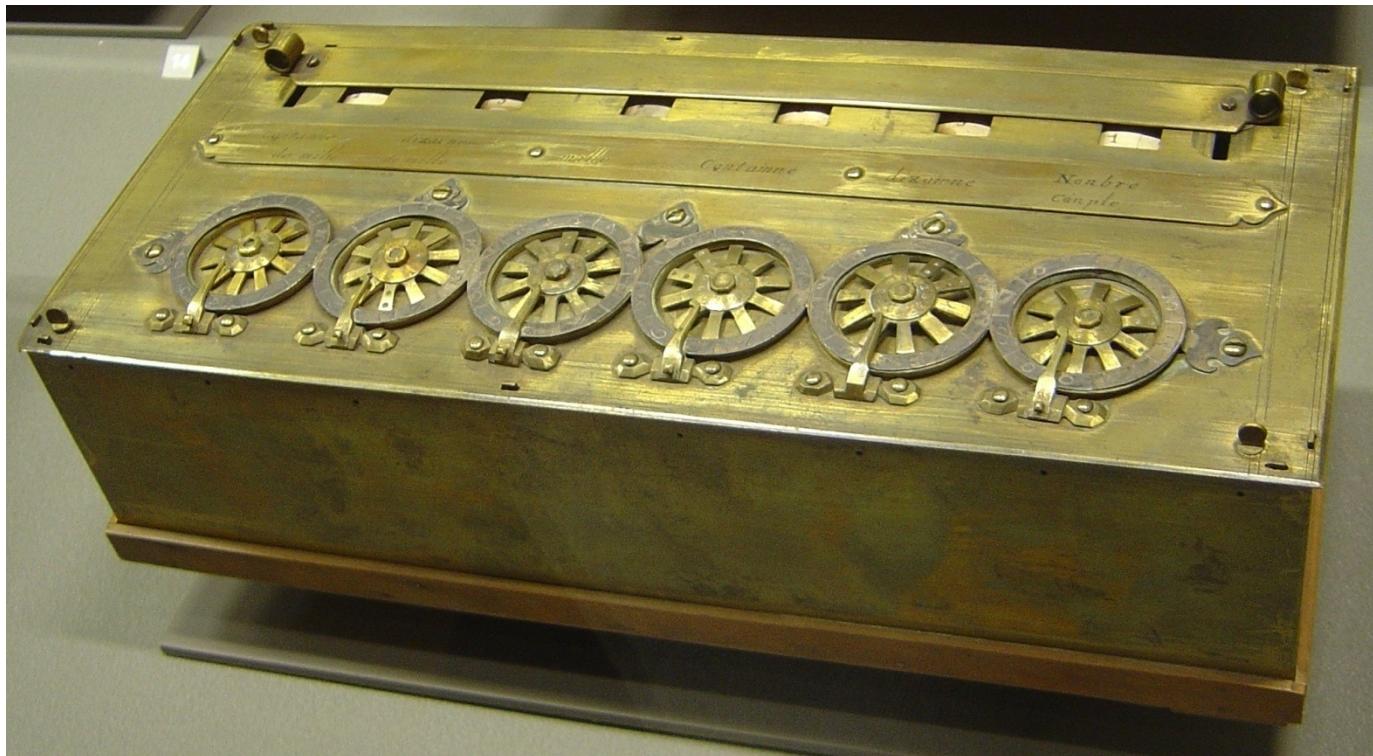
* Ganze Breite nur 7 cm *
Ganze Länge nur 31 cm *

Ruthardt & Co.
G. m. b. H., Stuttgart



A još ranije...

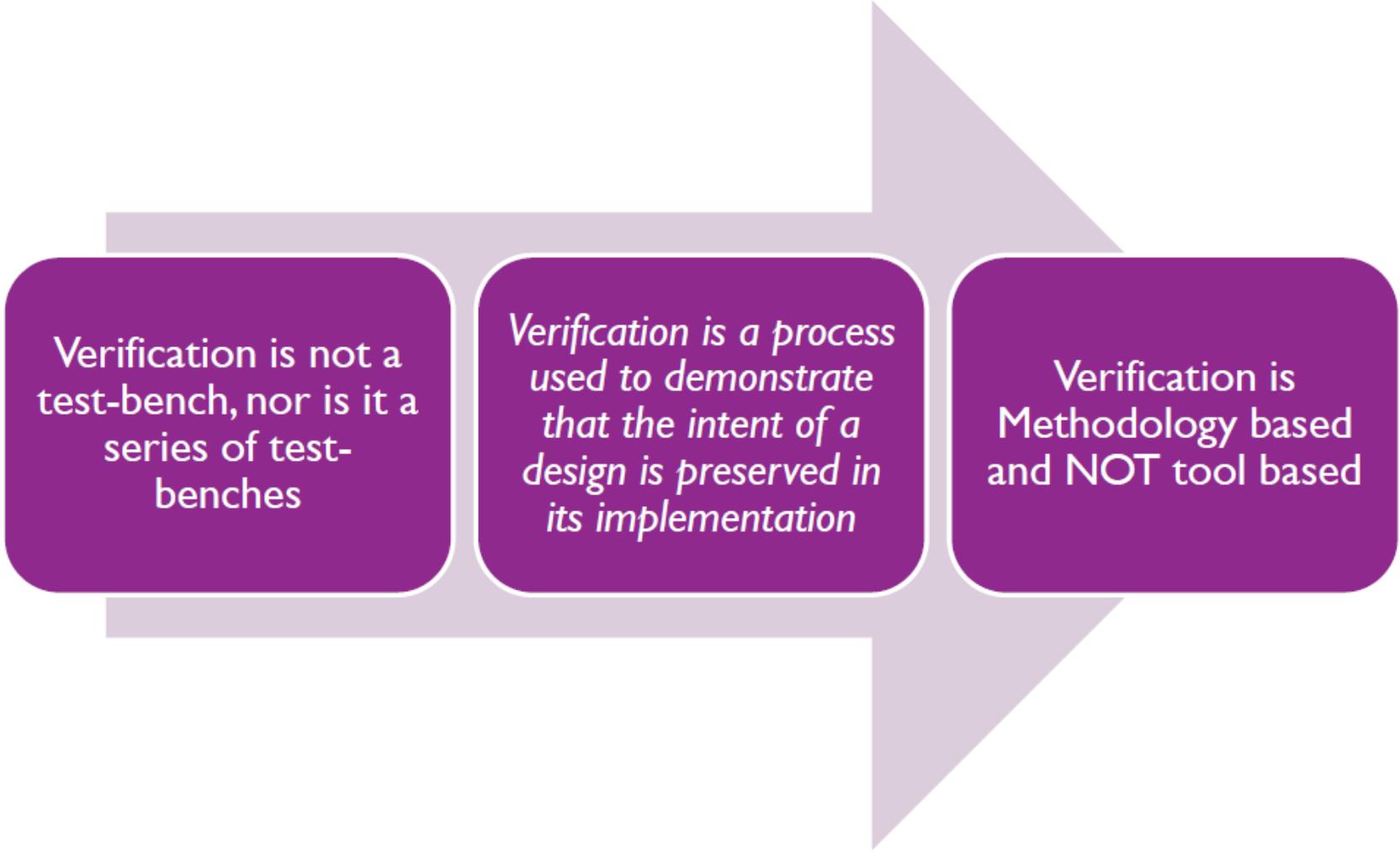
1644 Pascalina



Gde smo sad u CMOS tehnologiji

- 180 nm (analog, RF, Digital) vrlo stabilan proces dominira u bezbednosno kritičnim aplikacijama i dalje.
- 65 nm (digital)
- 40 nm
- 28 nm
- 16 nm
- 10 nm
- 7 nm
- 5 nm
- 1 nm (očekuje se da će tu biti tačka sa polu provodničkim CMOS minijaturizacijom)
- ?

Šta je uopšte verifikacija čipa

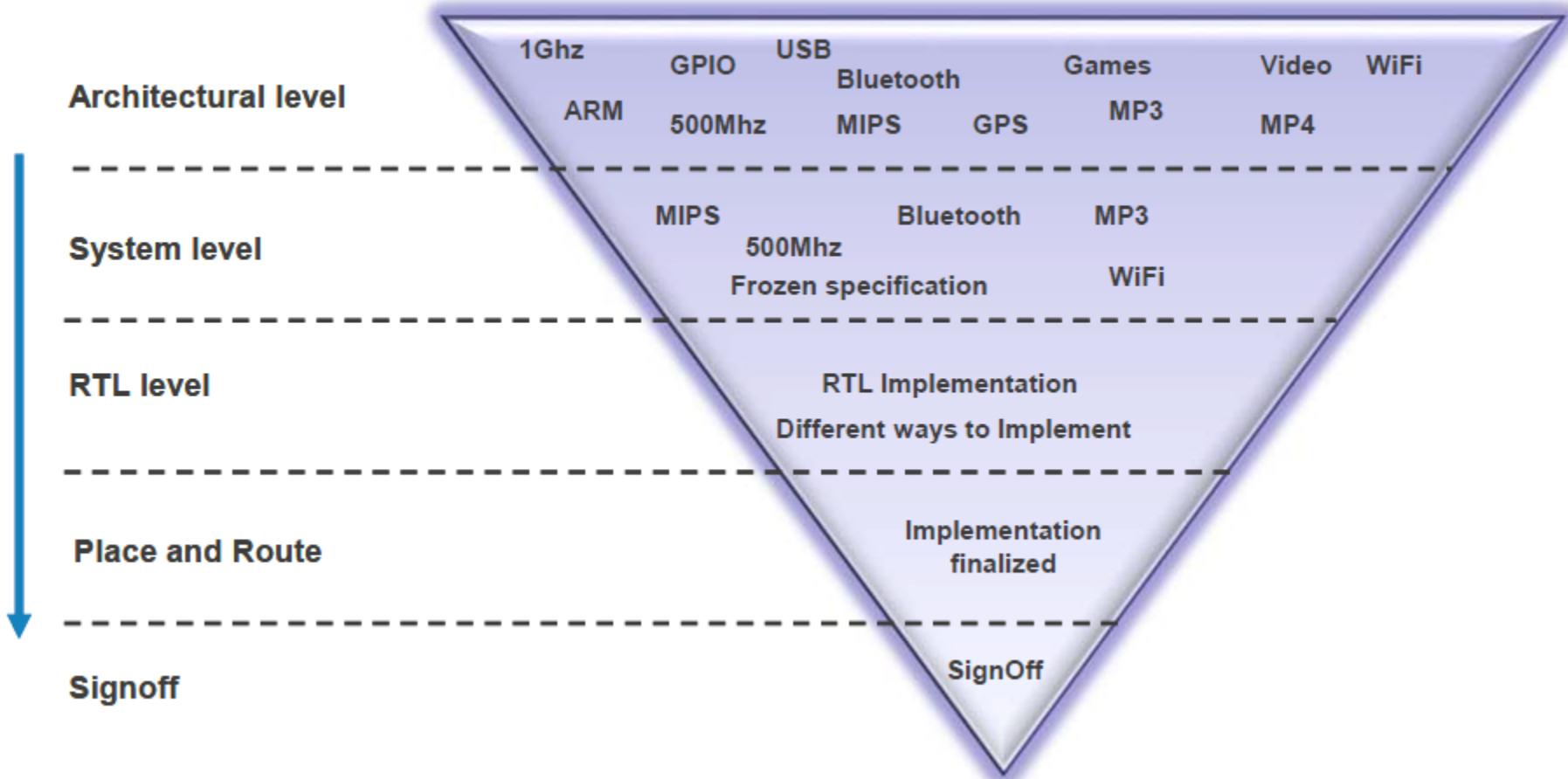


Verification is not a test-bench, nor is it a series of test-benches

Verification is a process used to demonstrate that the intent of a design is preserved in its implementation

Verification is Methodology based and NOT tool based

ASIC dizajn



- Vreme razvoja se skraćuje
- Specifikacija se zamrzava
- Funkcionalna verifikacija more da krene što pre, svako kašnjenje uvećava rizike

Nivoi verifikacije

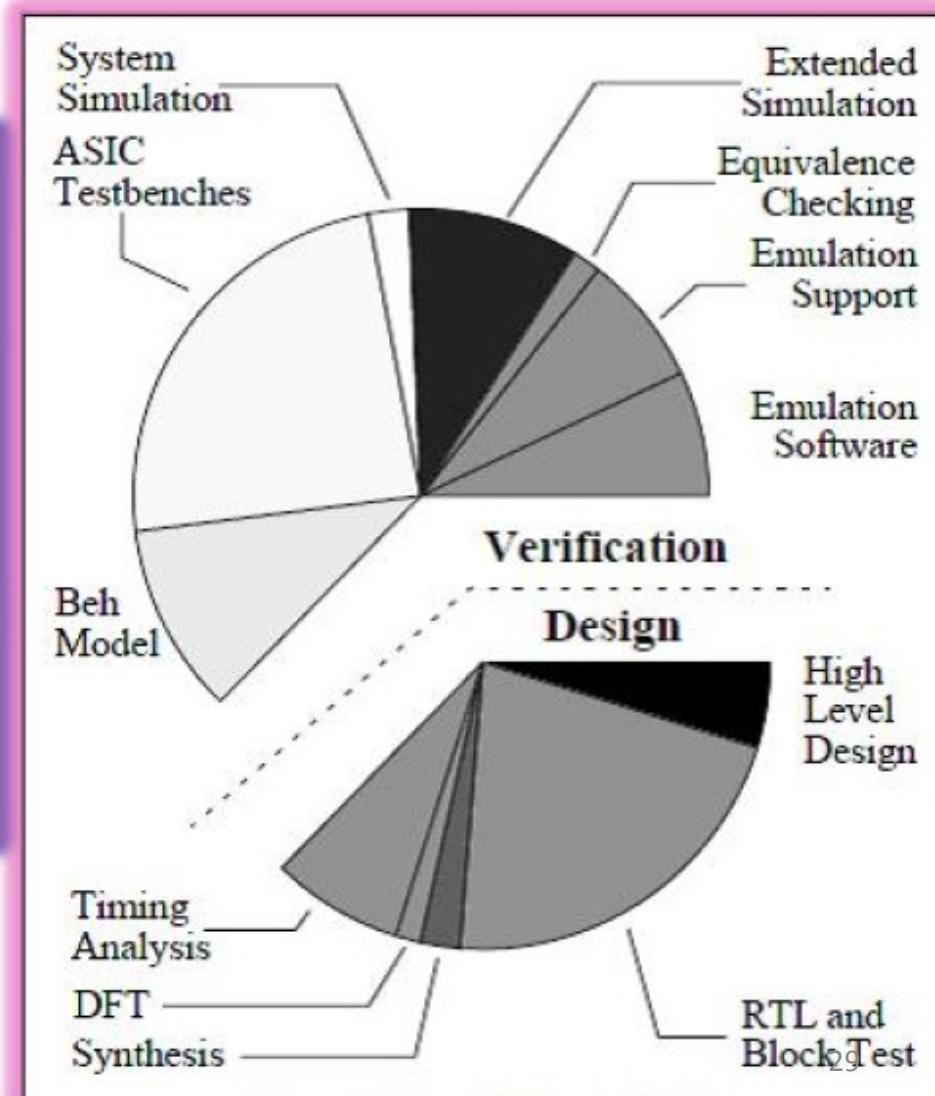
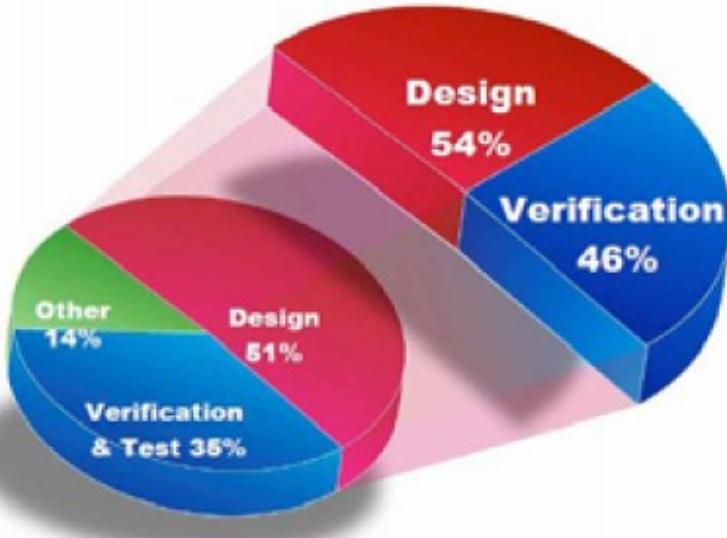
- Nivoi verifikacije
 - Black Box verifikacija
 - White Box verifikacija
 - Grey Box verifikacija
- ASIC Bagovi na različitim nivoima
 - Bag u arhitekturi
 - Bag u specifikaciji
 - Bag u RTL dizajnu/implementaciji (vhdl/verilog)
 - Synthesis/Constraining/timing/optimizations bag
 - Fizički dizajn (pogrešno označenje, labeliranje, ...)
 - Fizička verifikacija (DRC)
- Verifikacija za ASIC
 - Funkcionalna verifikacija
 - Timing verifikacija
 - Formal verifikacija
 - Fizička verifikacija

Verifikacione aktivnosti nose 70% posla

- Statistički analizirano, funkcionalna verifikacija često pokriva 70% aktivnosti u čip dizajnu
- Analize prezentovane po konferencijama kažu sledeće (*design-reuse, eetimes*)
 - Sistemski dizajn 13%
 - Logički dizajn 20%
 - Funkcionalna verifikacija 22%
 - Sinteza 8%
 - IC place and Route 13%
 - IC fizička verifikacija 11%
 - Analogni/Mixed Signal dizajn 13%
- Gde je razlika između 22% i 70%?

Verifikacija je protkana kroz sve ostale aktivnosti

Design Engineers Are Becoming Verification Engineers



Zašto funkcionalna verifikacija?

- Kvalitet ima sve veći značaj! Treba izgraditi poverenje da bi se ostalo u poslu
- Verifikacija je jedinstvena za svaki projekat
- Greške u RTL dizajnu
 - Ako se greške aktiviraju, sistem može početi da se ponaša nepredvidivo
 - Neće sve greške prouzrokovati kritično ponašanje sistema
 - Greška u mrtvom kodu nikada neće napraviti problem
 - Nisu sve greške rezultat greške dizajnera
 - Greška može da leži unutar specifikacije
 - Sporna komunikacija između dizajnerskih timova može lako da rezultuje greškom
- TestBenč emulira okruženje u kom će dizajn “živeti”
- Verifikuje da je RTL Implementiran u skladu sa specifikacijom
- Funkcionalna verifikacija podrazumeva detekciju grešaka i njihovo ispravljanja pre isporuke korisnicima

Nivoi funkcionalne verifikacije

- Unutar modula/bloka
 - Greške se lako pronalaze
 - Željena funkcionalnost se svodi na aritmetičku, logičku, komunikacionu
 - Nezavisno (uglavnom se pronalaze testiranjem nad datim blokom)
- Greške između blokova (projektovali različiti inženjeri)
 - Za prvog inženjera, drugi dizajn je strani kod!
 - Nema poznavanja implementacije, već samo specifikacije
 - Teže za debagovanje
- Testiranje ingerisanog sistema
 - Svi podblokovi već nezavisno testirani su integrисани
 - Top-level test
 - Veoma teško debagovanje
- Značajno je emulirati realno radno okruženje koliko god je moguće
 - Treba emulirati nepredvidivo ponašanje korisnika / okruženja
 - Isti nivo nepredvidivosti treba preneti u verifikaciju (tu nam pomaže randomizacija)
 - Randomizacija se uključuje u testove

Verifikacioni plan

- Pre početka pisanja testova treba znati
 - Po kom planu će se verifikovati DUT ?
 - Koja će biti struktura testbenča?
 - Svaki testbenč ima driver, receiver i monitore!
- Svaki projekat sadrži greške! Treba ih svesti na minimum pre izrade integrisanog kola (tape-out)
- Uvek treba precizno voditi evidenciju o nađenim bagovima
 - Ispravka jednog baga može kreirati nove
- Treba proveriti da li dizajn tačno reprezentuje specifikaciju
- Treba odrediti verifikacione granice datog integrisanog kola
- Jedinstvena specifikacija – različite interpretacije
 - Za implementaciju RTL dizajner
 - Za Test-benč verifikacioni inženjer

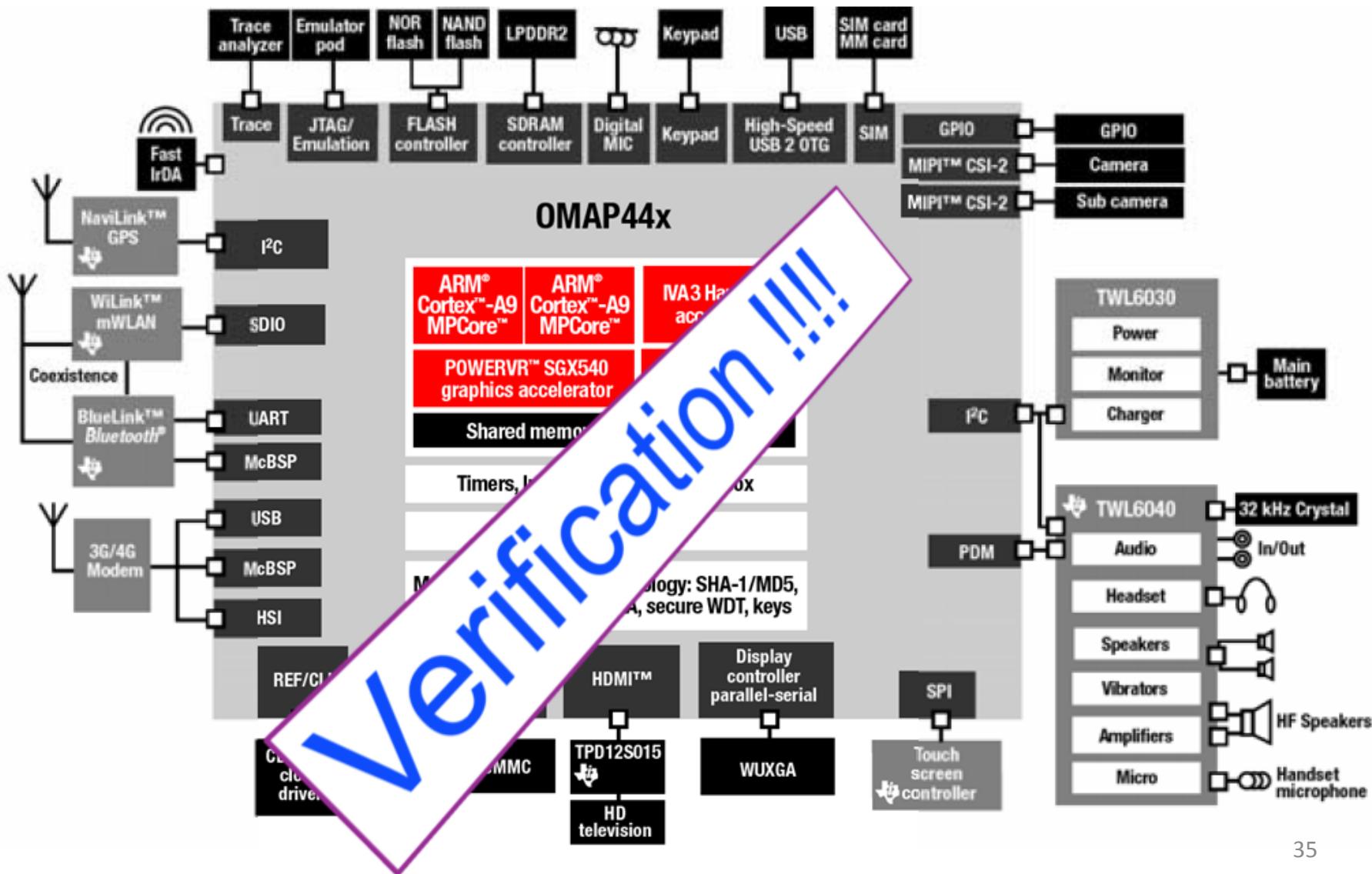
Pristup problemu verifikacije

- Mešati više verifikacionih scenarija
- Potrebno je pratiti progres verifikacije korišćenjem scoreboard-a
 - Što viši rezultat u scoreboard-u, bolji rezultati
- Za verifikaciju na sistemskom nivou
 - Treba raditi reporting na svim podnivoima
 - Na top nivou, pojedini testovi mogu da ispisuju svoj status/progres na ekranu
 - Ovo usporava simulaciju, ali olakšava debagovanje
- Hardverski akceleratori ili emulatori
 - Vrlo blisko verifikaciji realnih ASIC integrisanih kola
 - Napredne mogućnosti debagovanja
 - Znatno brže od softverskih simulatora
 - Izuzetno su skupi, u praksi firme kao Cadence, ProDesign i slične rentiraju svoje HW akceleratore po vrlo visokim cenama...
- Koristi snagu softvera (C, C++) po potrebi

Pristup problemu verifikacije

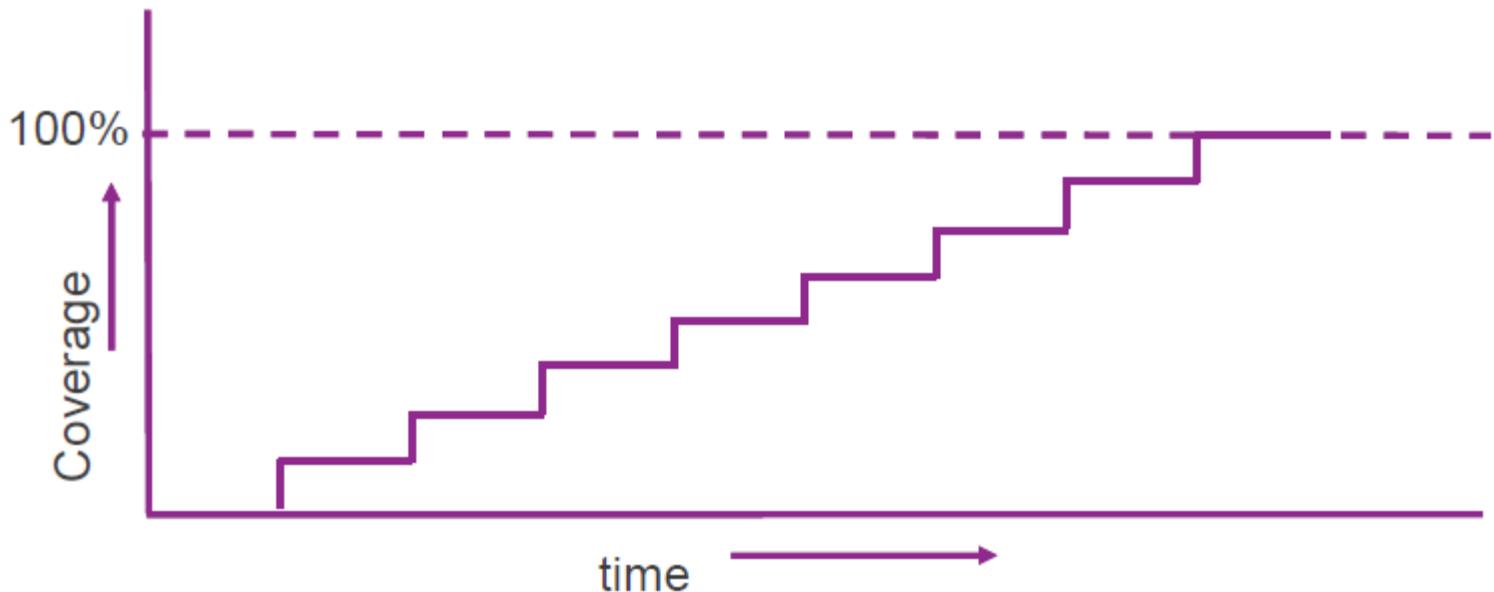
- Testbenč
 - Generisati stimulus
 - Proslediti stimulus ka DUT-u na način kako će DUT razumeti
 - Pratiti reakciju DUT-a
 - Verifikovati tačnost
 - Ciklično ponavljati gornji proces, sve dok se svi testovi ne kompletiraju po planu
- Direktni testovi
- Slučajni - Random testovi
- Ograničeni - Constrained Random testovi
- Pokrivenost
- Scoreboard
- Testbenč po nivoima!!! (ponovno upotrebljiv)

Primer složenog sistema u integrisanom kolu



Direktno testiranje

- To je tradicionalni ili klasični način testiranja
- Test jedan po jedan!
 - Ide se na sledeći nakon uspešno završenog testa
- Za mala “ASIC” kola ovaj vid testiranja je prihvatljiv!
- Metoda je fiksno vezana za kolo koje se testira, nema fleksibilnosti



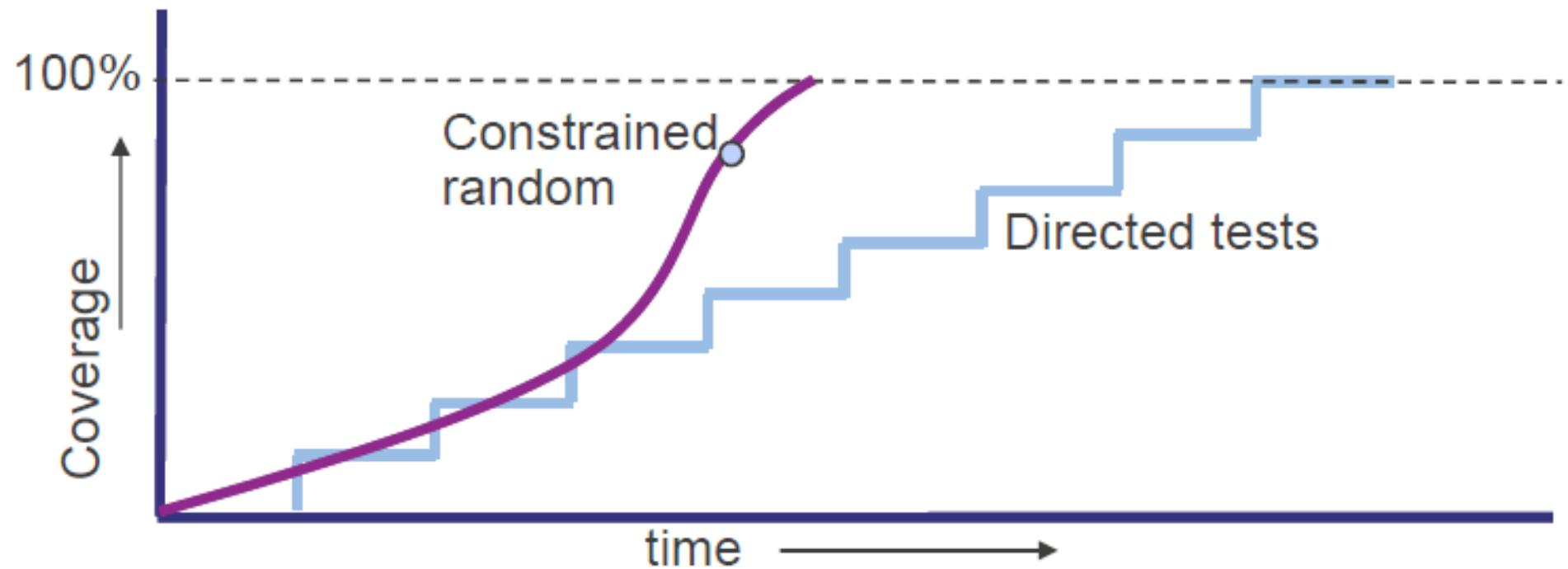
Slučajno – Random testiranje

- Dobro je koristiti slučajne testove
- Mogu da dođu do tačaka o kojima korisnik nikada nije rezmišljao
- Mogu da dođu i do tačaka do kojih nema potrebe doći
 - Vreme utrošeno na ovakve neograničene slučajne testove nije od koristi!
- Ako postoji bag, on mora biti ponovljiv
 - u suprotnom, bag se ne može pratiti
- u najgorem slučaju može se promašiti cilj verifikacije
- Korisnik gubi kontrolu nad procesom

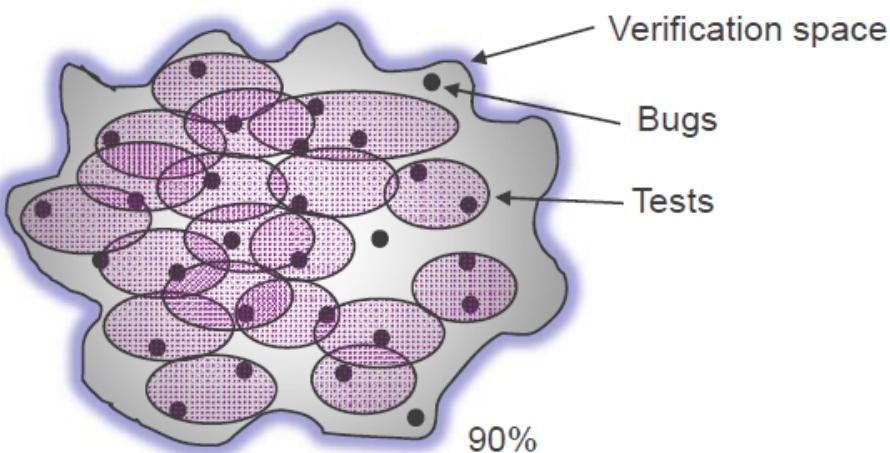
Ograničeno slučajno testiranje

- Slučajnost mora biti ponovljiva (Pseudo-slučajno)
- Ograničavanje slučajnosti, postavljanje u određene granice je veoma korisno
 - Štedi vreme
 - Usmereno testiranje
 - 100% pokrivenost se može postići u kraćem vremenu
- Svi testovi se mogu reprodukovati/ponoviti
- Moguće je pokriti sve granične slučajeve
 - Scoreboard sa ograničenim slučajnim testiranjem predstavlja najbolju opciju
- Manji podskup direktnih testova može se pridodati ograničenom slučajnom testiranju

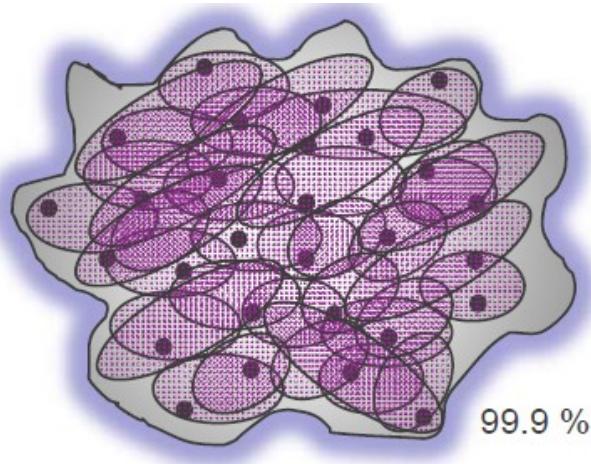
Ograničeno slučajno / direktno testiranje



Testiranje i pokrivenost



- Direktni testovi daju relativno nisku pokrivenost
- Sa niskom pokrivenošću, teško je prekinuti testiranje



- Sa slučajnim testovima (uz ograničenja i deo direktnih testova), pokrivaju se granični slučajevi.
- Dobija se visok nivo pokrivenosti

Konvencionalni testbenč

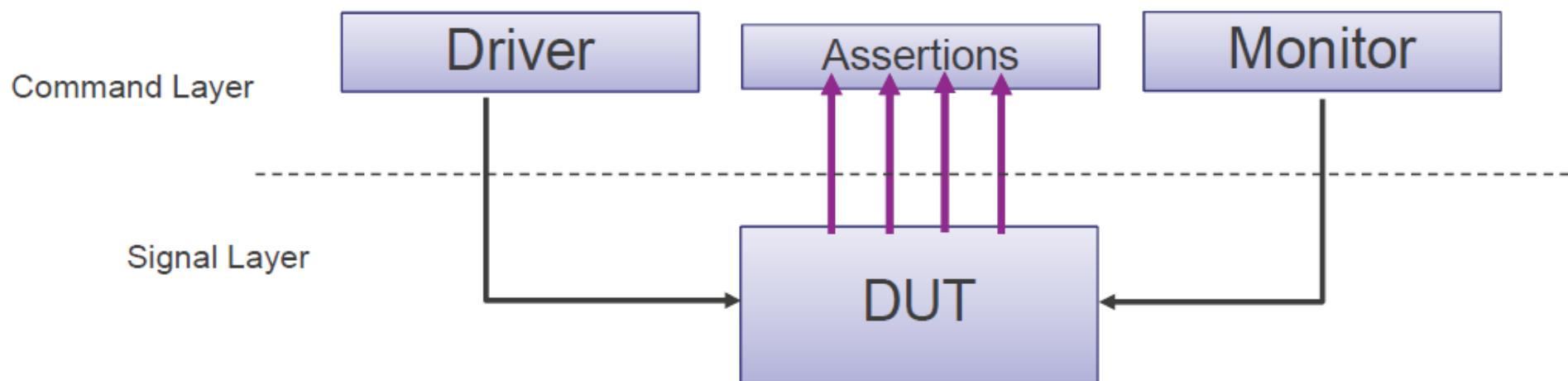
- Konvencionalni testbenč je mahom direktn
- Tipično jedan testbenč instancira DUT
 - Drajveri i monitori se nalaze u testbenč modulu
 - Uspeh ili neuspeh je raportiran od strane testbenča.
- Čitanje ili pisanje po magistrali (na primer: AMBA,OCP) se realizuje funkcijom ili taskom koju je pisao korisnik
- Takt slobodno osciluje, korisnik definiše reset na startu
- Verovatnoća pojave bagova unutar testbenča je relativno velika
- Dodavanje novih testova, modifikacije postojećih se realizuju nad istim fajlom.
- Potreban je testbenč koji će ispitati naš testbenč...?
 - Ko će nam čuvati čuvare?
- Malo šta od testbenča se može upotrebiti višekratno!
- Nakon nekog vremena, čak ni autor tesbenča neće prepoznati šta je radio i kako!
 - Vrlo teško za preuzimanje od strane drugih inženjera!

Testbenč po nivoima

- Umetnički / zanatski pristup testbenču
 - Očuvati jednostavnost
 - Formirati ga u nivoima
 - Logička podela po nivoima
 - Podela odgovornosti po verifikacionim blokovima
 - Koristiti prethodno testirane biblioteke komponenti po potrebi
- Manje grešaka, dobra struktuiranost, moguća višekratna upotreba
- Buduće modifikacije svode se samo na intervencije nad pojedinim blokovima
 - Ostatak se ne menja!
 - Direktni testovi nad tim blokovima takođe ne menjaju osnovni testbenč.
- Razumljivi za verifikacione inženjere
- Lako se debuguju i prate
 - Sve dok je blokovski podeljena verifikaciona odgovornost i funkcionalnost !

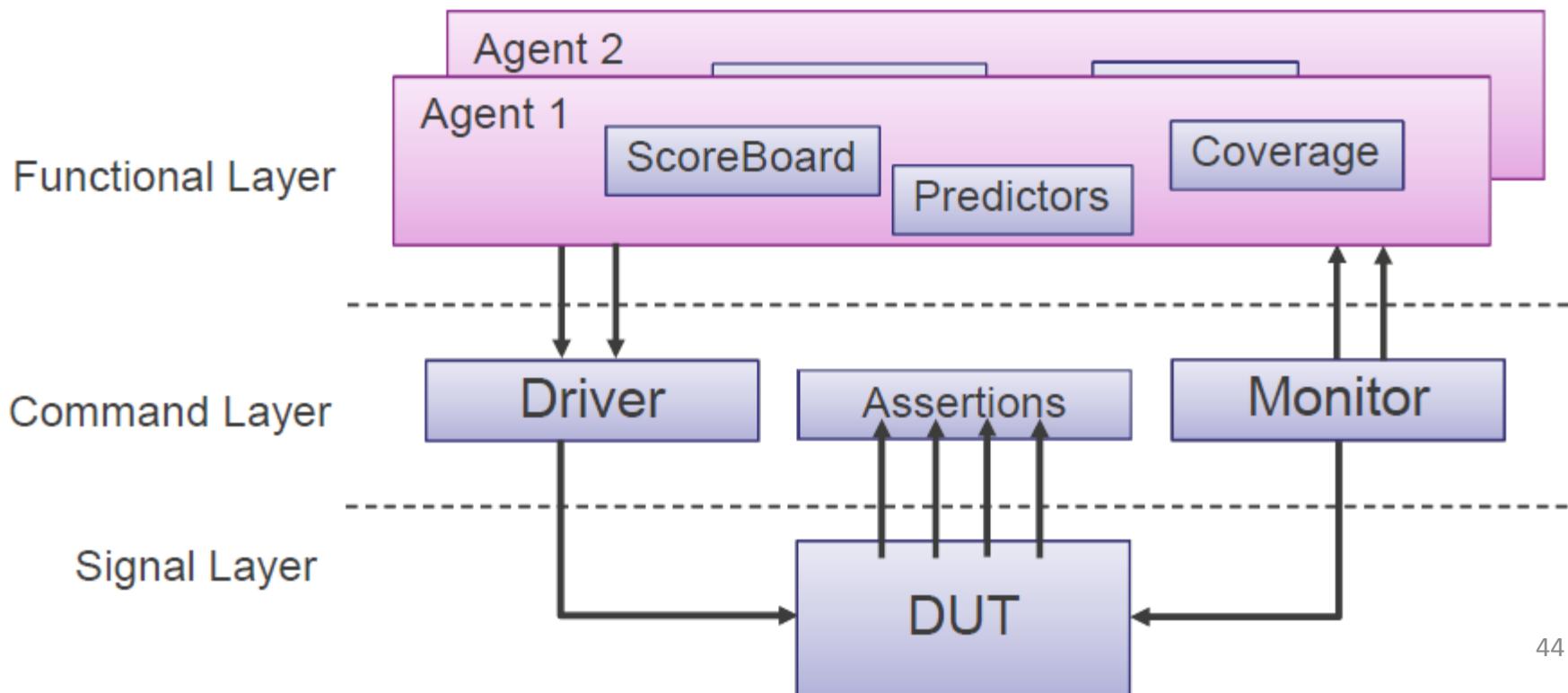
Testbenč po nivoima

- Zajedničke akcije unutar testbenča mogu se grupisati
 - Komandni nivo (r/w , bus r/w, ...)
 - Signalni nivo (konekcije ka DUT)



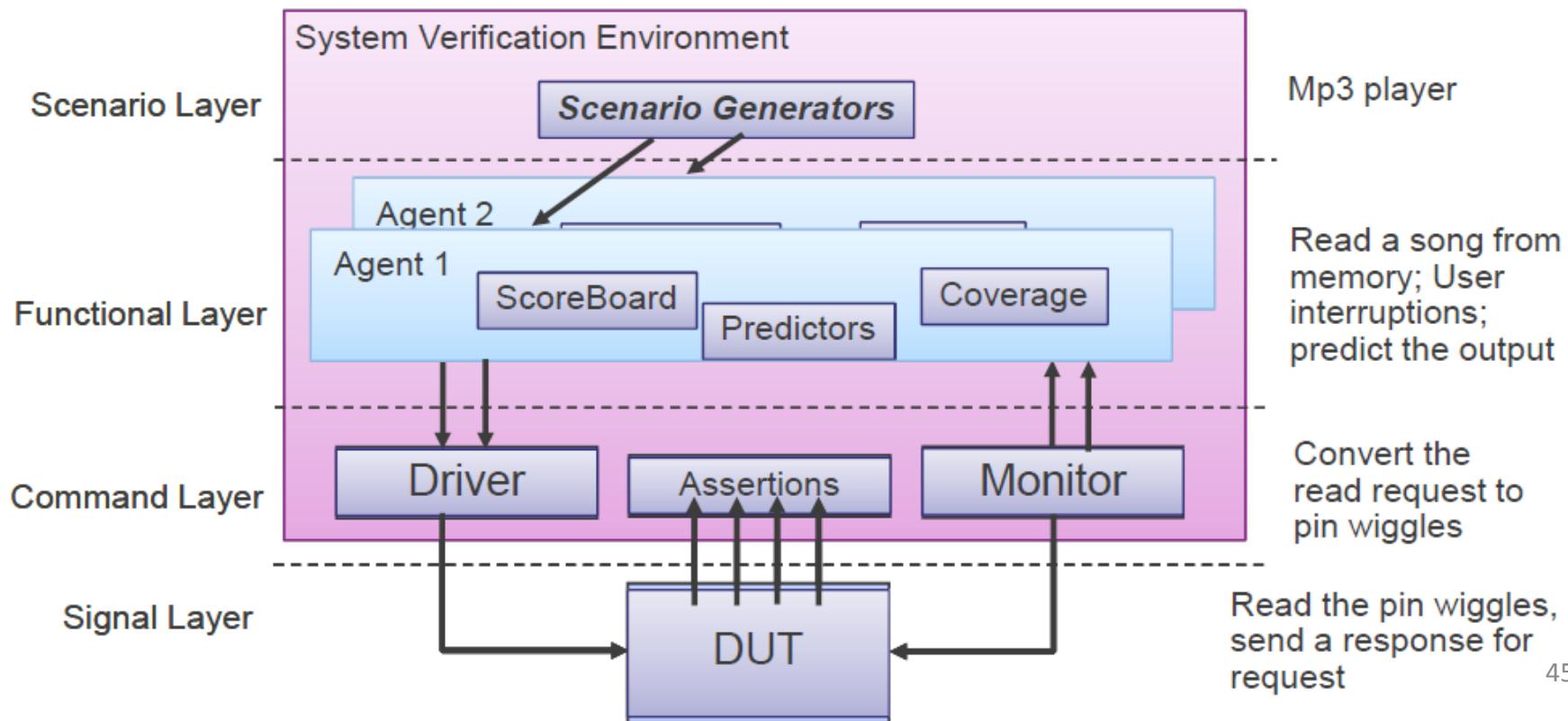
Testbenč po nivoima

- Funkcionalni nivo
- Hrani komandni nivo
- Svaki agent je odgovoran za spoju specifičnu namenu
 - On sadrži implementaciju testa



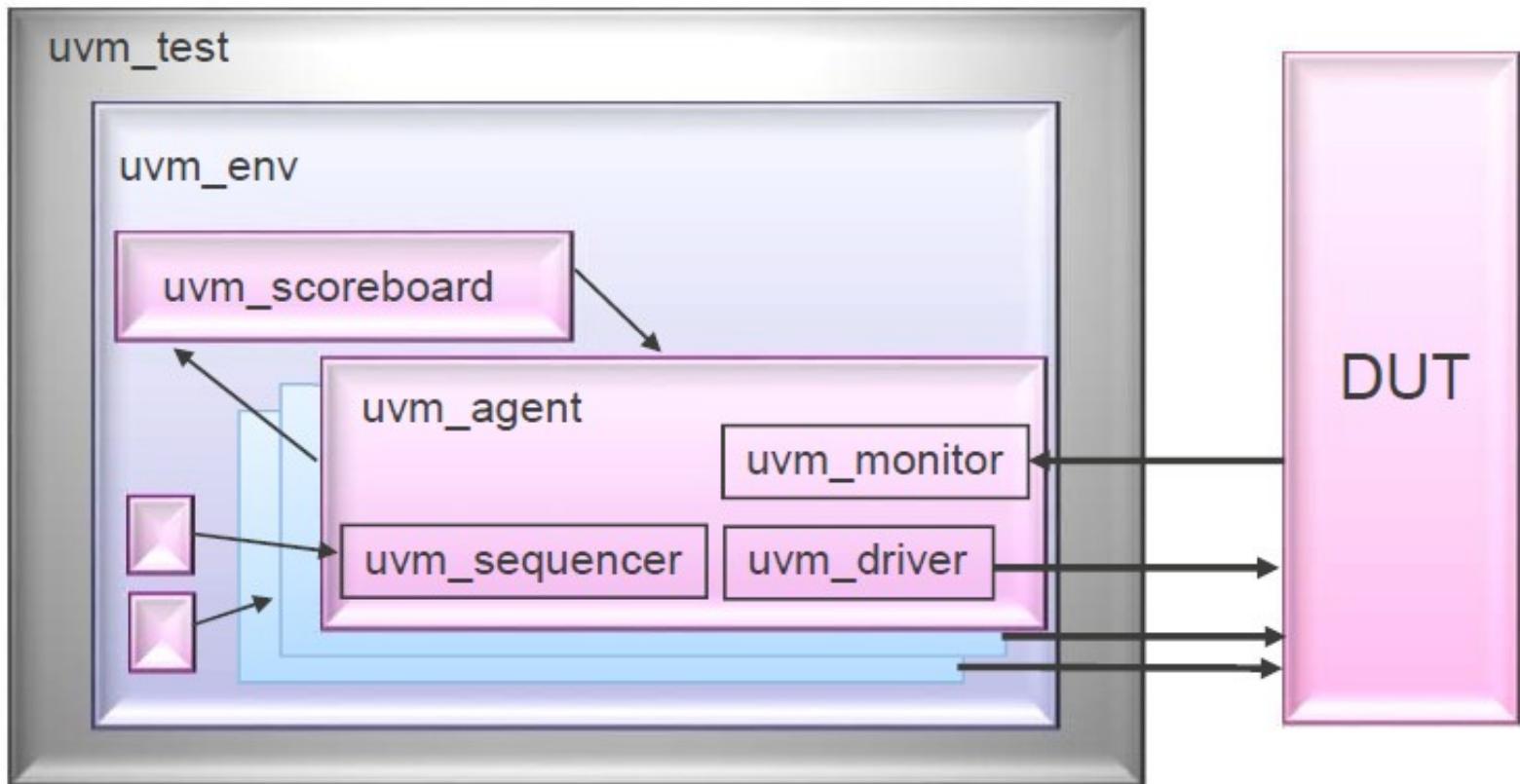
Testbenč po nivoima

- Nivo scenarija
- prime: testiranje video/mp3 plejera, GPS, USB na platformi mobilnog uređaja
- Svaki scenario sa svojim agentima na apstraktnom nivou
- Definiše se funkcionalnost koja se testira



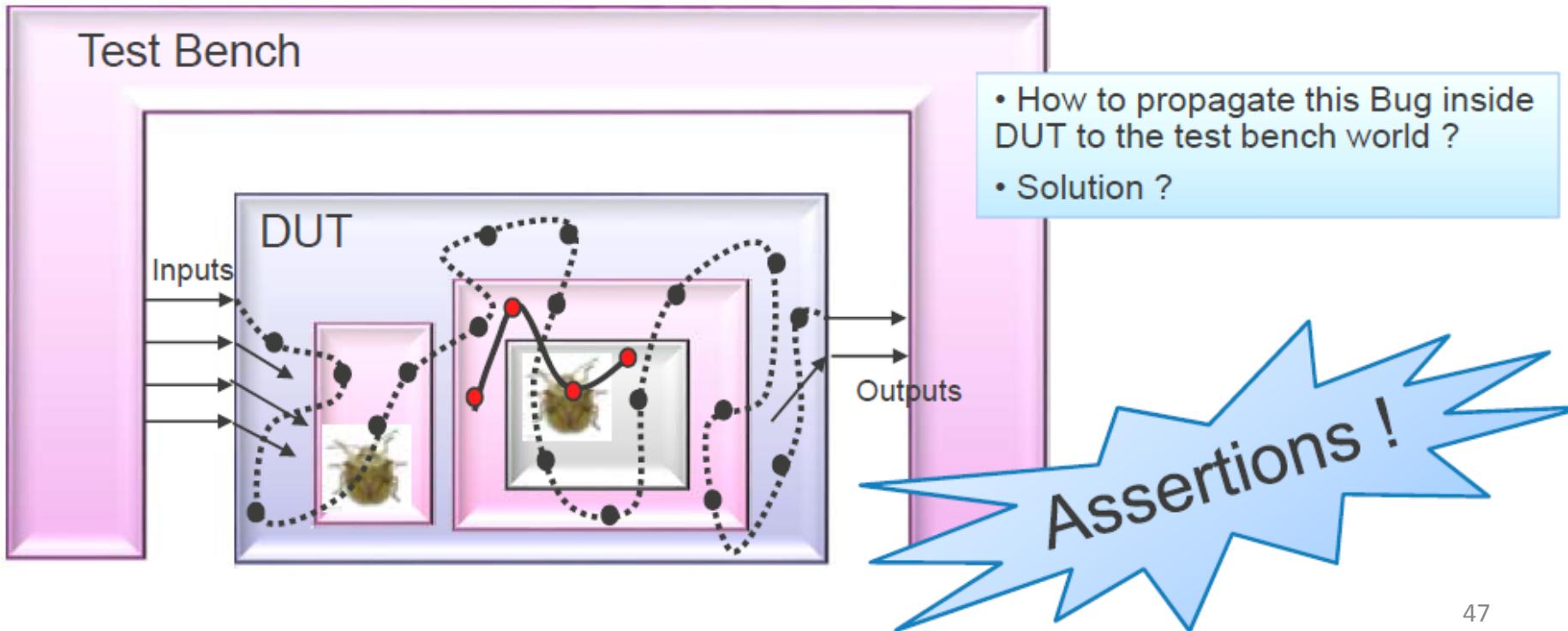
Plan

- Koji je plan?
- Uloga SystemVeriloga u svemu ovome?
- UVM omotač, SystemVerilog iznutra !
- Verifikacija bazirana na transakciji



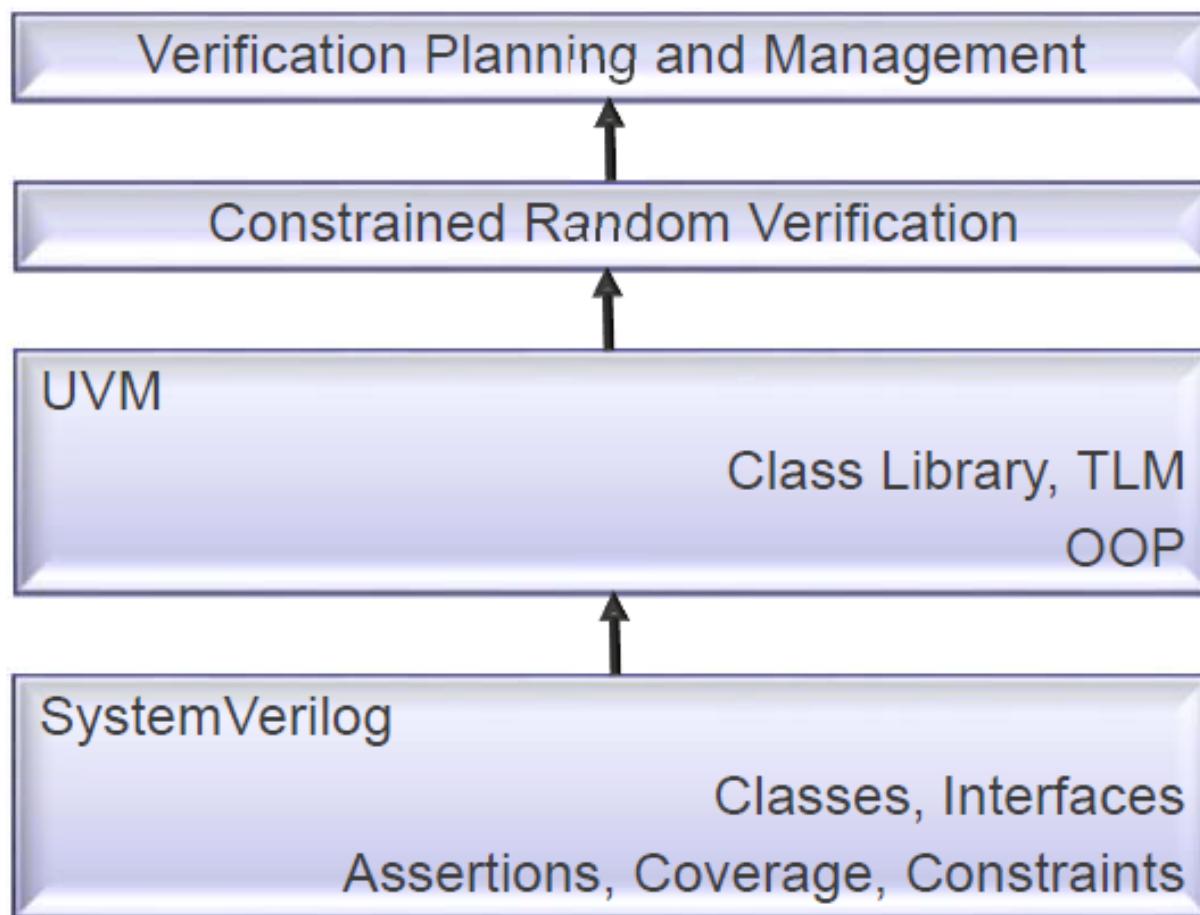
Konvencionalna verifikacija

- Proistup od ozdo ka gore
 - Verifikacija podblokova prvenstveno, zatim gore korak po korak
- Top-level verifikacija podrazumeva
 - Da su podmoduli verifikovani. Znači očišćeni od bagova?
- A šta ako ipak nisu?



Verifikacija bazirana na SV

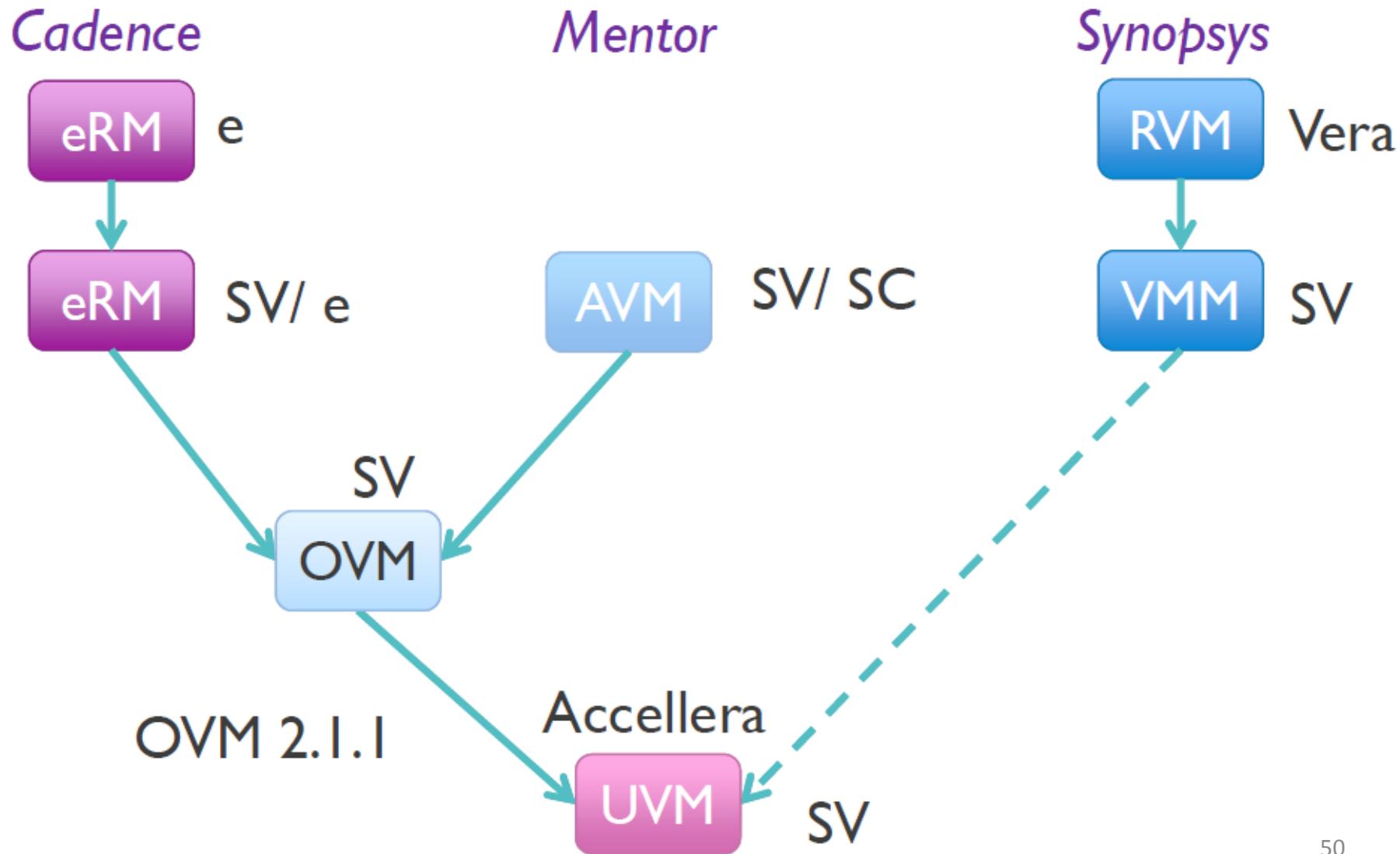
- Kako se pripremiti za SV baziranu Verifikaciju?



SV, UVM

- SystemVerilog je IEEE standard još od 2005
 - Podržan od strane većine EDA (Electronic Design Automation) prodavaca
- Mentor je podržavao svoj AVM (Advanced verification Methodology)
 - TLM baziran, modularan, upravljan pokrivenošću
- Cadence je podržavao svoj URM (Universal Reuse Methodology)
- Mentor + Cadence , OVM (Open Verification Methodology)
 - Otvoreni kod
- Synopsys, VMM (Verification Methodology Manual)

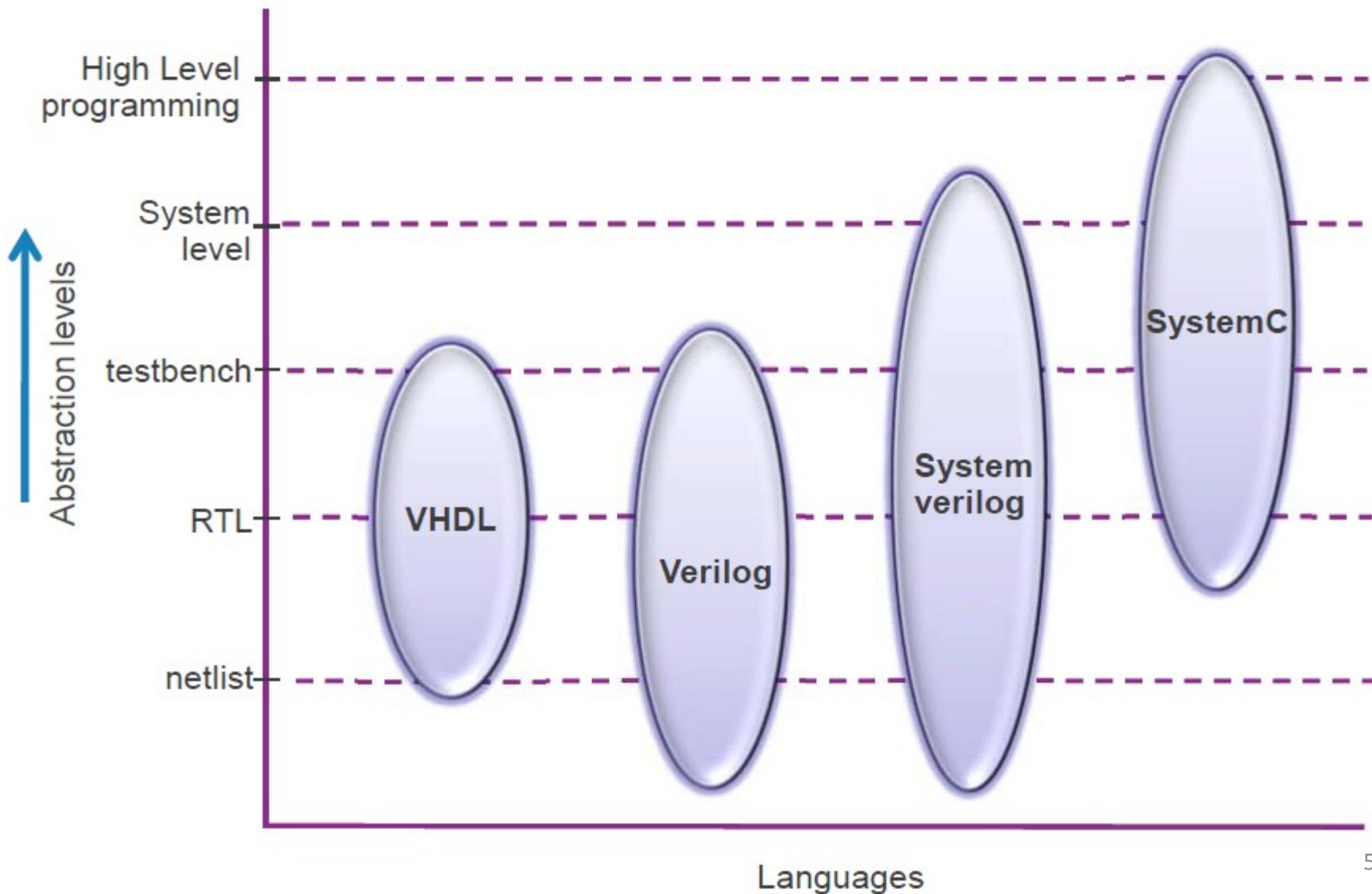
UVM porodično stablo



SV posledice

- SV sam za sebe možda nije rešenje
- Ali nudi široko polje otvorenih verifikacionih biblioteka
- UVM, VMM, AVM, ...
- Puno toga mora da se uči
- Veoma kompleksan sa puno mogućnosti
- Svako može da kreira sopstveni projektni pristup (Otvoren)
- Delom nas uvodi u softverski svet
- Da li hardverski svet to voli?
- SV obuhvata 5 jezika u jednom:
 - Sintetizibilni podskup
 - Assertions
 - Ograničenja - Constraints
 - Pokrivenost - Coverage
 - OOP

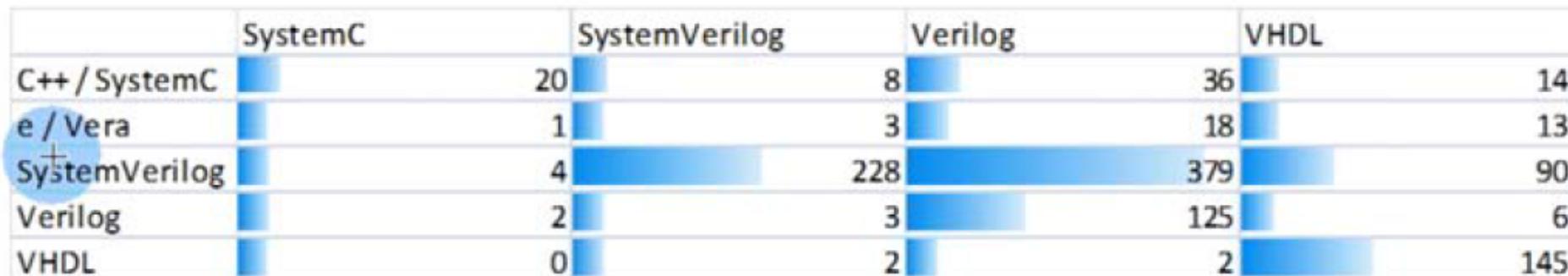
Nivoi apstrakcije HDL/HVL



Rasprostranjenost jezika

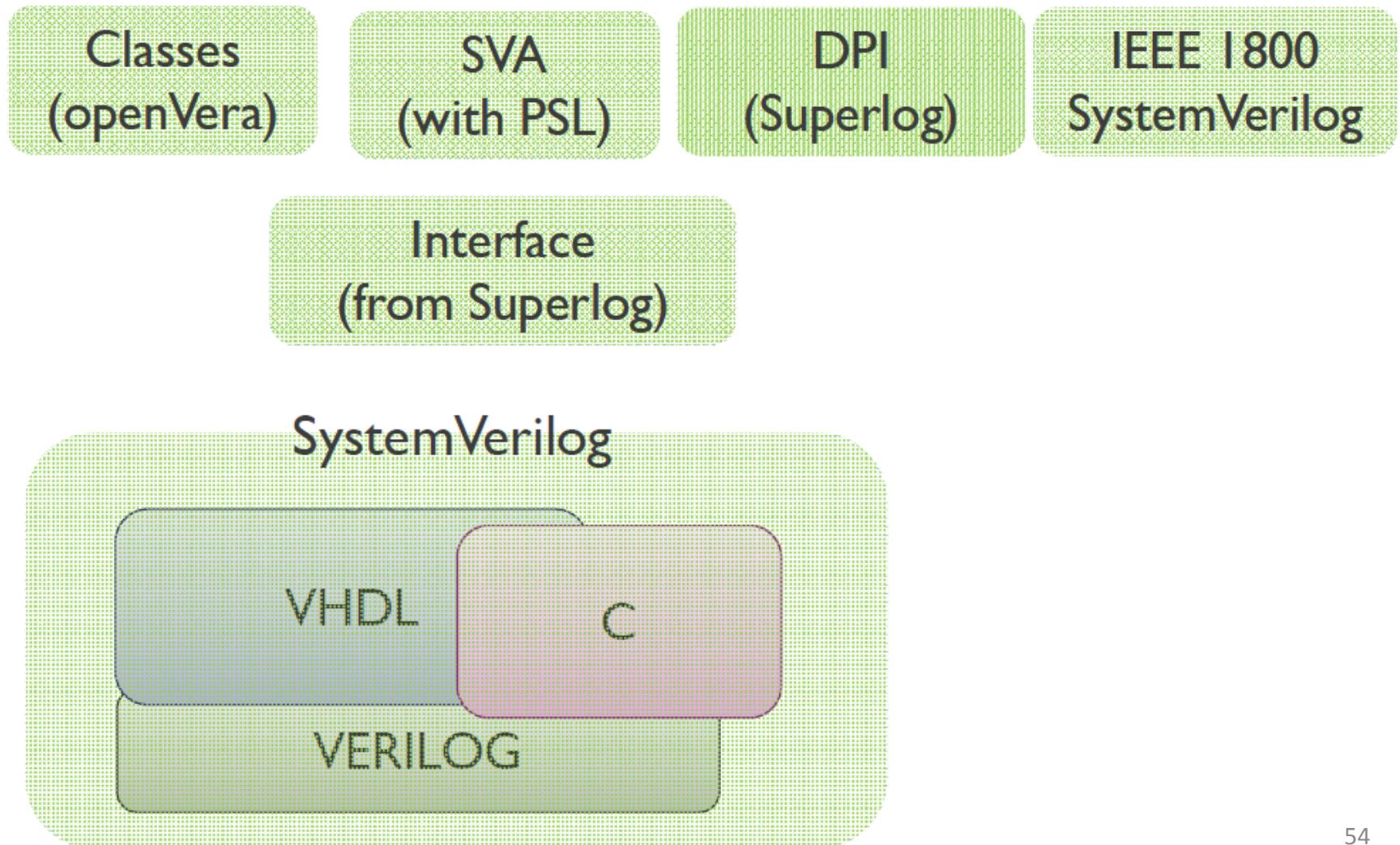
	ASIC	FPGA
C++ / SystemC	58	11
e/Vera	34	0
SystemVerilog	588	89
Verilog	88	42
VHDL	50	100

RTL

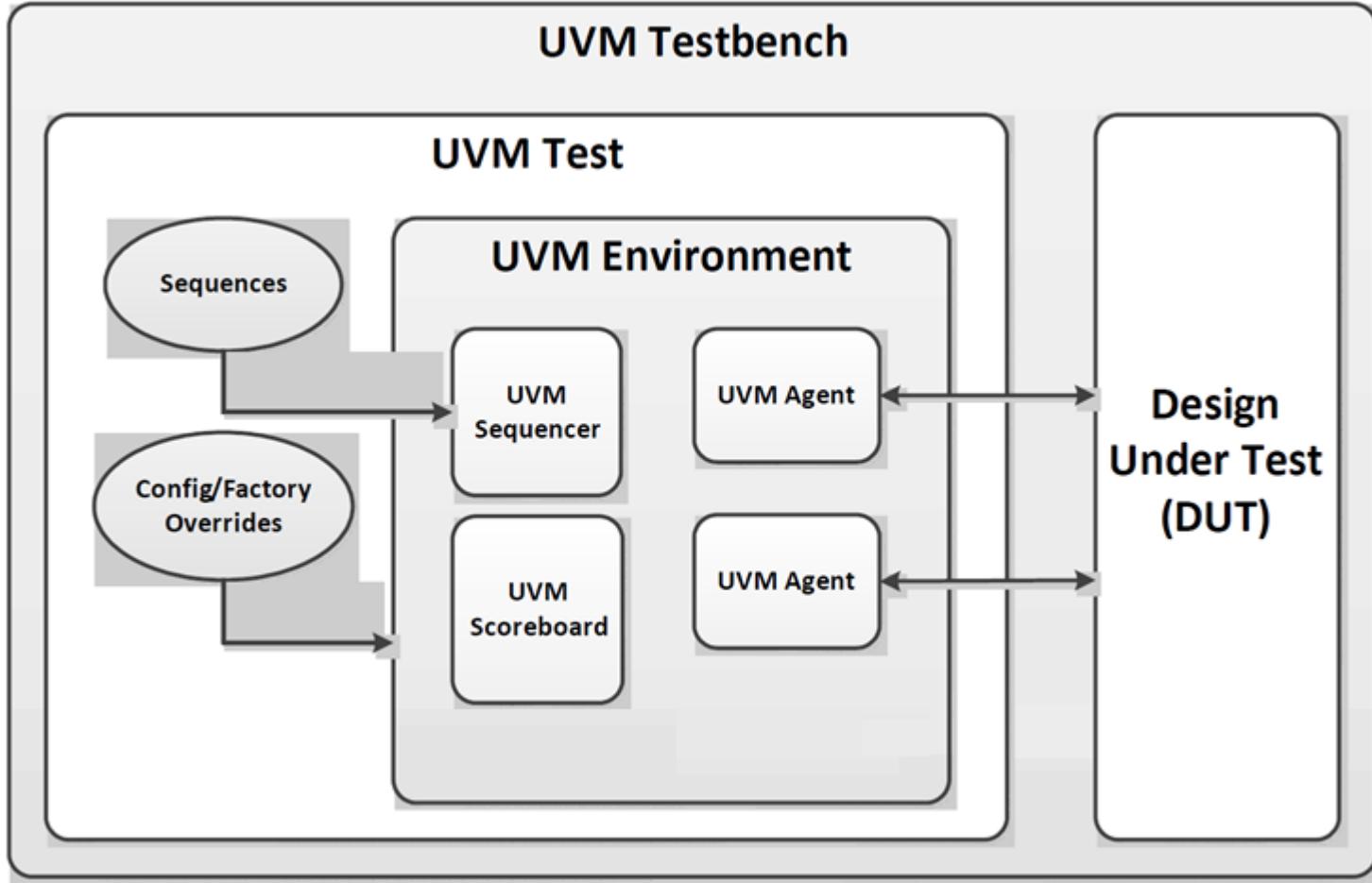


Verification

Evolucija jezika



Povratak na UVM Testbench



UVM Simulacione faze

- Verifikacija u okviru UVM šablonu je podeljena na faze
 - Kreiranje testbenča, konfigurisanje, izvršenje, provera, itd.
 - Setup aktivnosti tokom faze kreiranja
 - Generisanje stimulusa tokom faze izvrešenja
- Komponente izvršavaju taskove samo u specifičnim fazama
 - Scoreboard proverava rezultate tokom faze provere
- Faze se izvršavaju po definisanom redosledu
 - Naredna faza nemože da započne, sve dok se aktuelna faza ne okonča
- Skup standardnih predefinisanih faza omogućava laku integraciju velikog broja komponenti

UVM Simulacione faze

Build_phase
Connect_phase
end_of_elaboration_phase
Start_of_simulation_phase
run_phase
Extract_phase
Check_phase
Report_phase
Final_phase

- Kreira top nivo testbenča
- Povezuje environment topologiju
- Post elaboracija
- Konfigurisanje verifikacionih komponenti
- Izvršavanje testova
- Prikupljanje podataka o finalnom statusu DUT-a
- Procesiranje i provera simulacionih rezultata
- Analiza i raspodjeljivanje simulacionih rezultata
- Završetak i zatvaranje fajlova

Sve faze osim run_phase() se izvršavaju u nultom vremenu kao funkcije

Faza izvršenja Run faza

- Faza izvršenje je predviđena da obezbedi izvršavanje zadataka UVM komponenti (UVC)
- UVC može da generiše različiti stimulus u različitim fazama
- UVC može da skoči nazad na prethodnu run-time fazu
 - Kada je reset aktiviran tokom osnovne faze
- UVM komponente (UVC) se mogu grupisati u domene
 - UVC sa zajedničkim resetom
- Treba koristiti `run_phase()` za definisanje stimulusa na interfejsima UVC-a
- `run ()` metod je fork-ovan, tako da se redosled pokretanja svih komponenti ne može sa sigurnošću znati!

Build faza

- Build_phase se koristi za kreiranje pod komponenti
 - umesto kreiranja hijerarhije korišćenjem konstruktora
- Za korišćenje build faze, jednostavno treba definisati funkciju build_phase unutar svoje uvm_component-e
 - Ta funkcija biće automatski izvršena tokom build faze naše simulacije
- Podkomponente kreirane u build funkciji su implicitno kreirane
 - build_phase funkcije podkomponenti se rekursivno pozivaju (automatski)
- Hijerarhija komponenti se gradi od vrha nadole
 - Samo je build_phase od vrha nadole, sve ostale faze su odozdo na gore

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // create component
endfunction
```

Connect faza

- Connect faza se koristi za kreiranje referenci i TLM konekcija između komponenti
 - Ova faza ima striktno tu namenu i povezivanje treba raditi striktno u njoj, a ne u build fazi ili konstruktoru
- Za korišćenje connect faze jednostavno treba definisati funkciju `connect_phase` unutar svoje `uvm_component`-e
 - Ona će se automatski izvršiti tokom simulacione `connect_phase`
- Connect funkcije se izvršavaju nakon što je celokupna topologija izgrađena, build-ovana
 - Treba izbegavati race conditions koje mogu da uslede zavisno od redosleda izgradnje build-ovanja podkomponenti

```
function void connect_phase(uvm_phase phase);
    result_monitor_h.ap.connect(scoreboard_h.analysis_export);
    command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
    command_monitor_h.ap.connect(coverage_h.analysis_export); endfunction : connect_phase
```

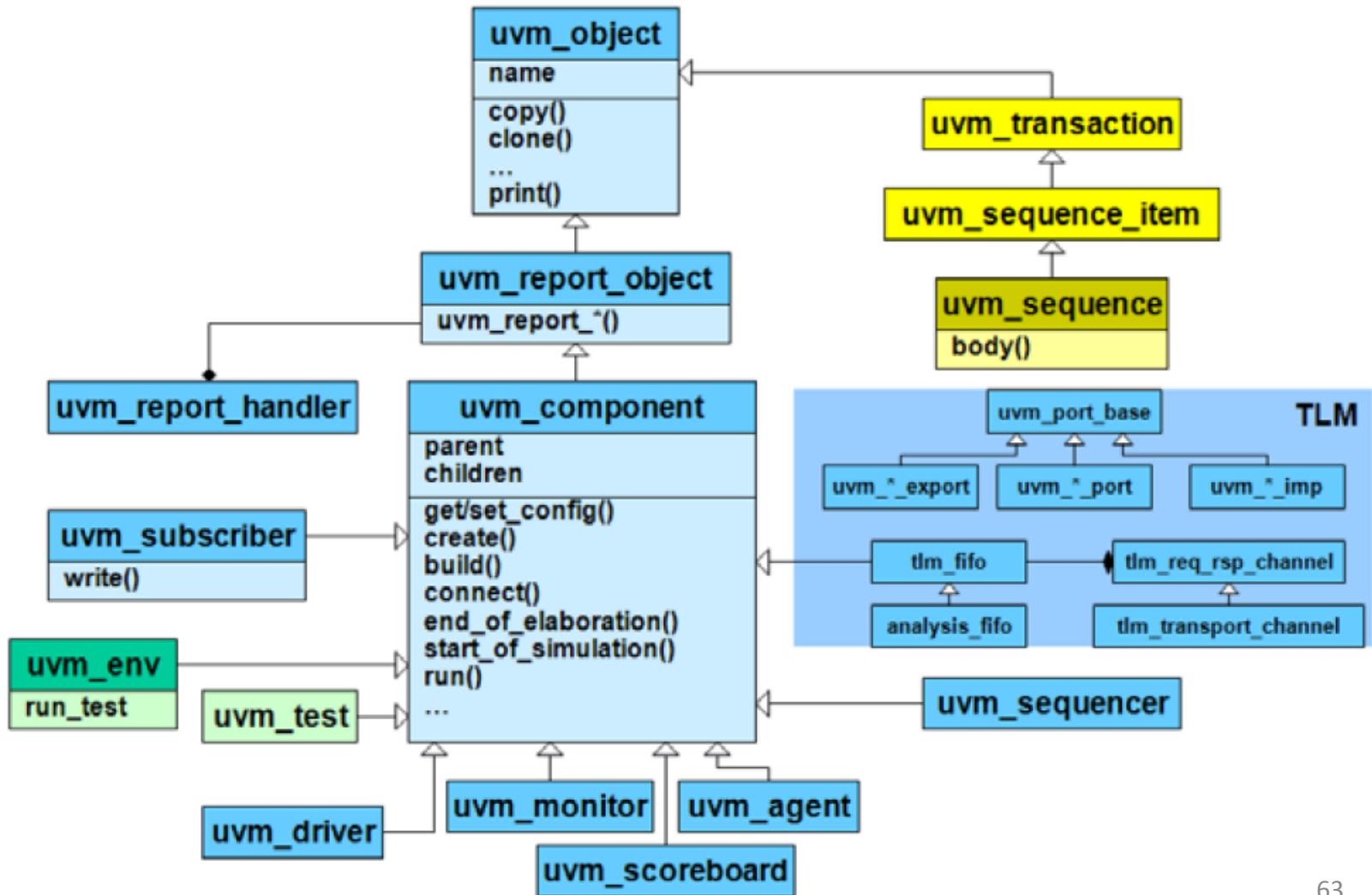
Kako se pokreću faze?

- Pozivom globalnog taska **run_test()** sa nivoa modula faze se startuju
- Izvršavanje
 - Funkcije build faze se izvršavaju odozgo na dole
 - Taskovi run faze se izvršavaju u paraleli
 - Sve ostale faze se izvršavaju odozdo na gore
 - Na nivou svih komponenti unutar simulacije faze se izvršavaju jedna za drugom, nema preklapanja
- Timing
 - Run faze svih komponenti su taskovi koji se događaju u vremenu, troše simulaciono vreme
 - Sve ostale faze su definisane kao funkcije i izvršavaju se u nultom vremenu, ne troše simulaciono vreme
- Za korišćenje faza potrebno je definisati phase metod i on će se izvršiti tokom istoimene simulacione faze

Izvršavanje faza



UVM hijerarhija klasa



Modelovanje nivoa transakcije (TLM Transaction Level Modeling)

- Jedan od bitnijih elemenata za povećanje produktivnosti verifikacije je razmišljanje o problemu na nivou apstrakcije.
- Prilikom verifikacije DUT-a koji upravlja informacionim paketima koji se promeću napred i nazad, ili obrađuju instrukcije, ili obavljaju druge aktivnosti, mora se kreirati verifikaciono okruženje koje podržava odgovarajući nivo apstrakcije.
- Dok je stvarni interfejs za DUT na kraju predstavljen signalnim nivoom, iskustvo je pokazalo da je neophodno upravljati većinom zadataka verifikacije, kao što je generisanje stimulusa i prikupljanje podataka o pokrivenosti, na nivou transakcija, što je prirodan način razmišljanja inženjera o aktivnosti sistema.

TLM

- UVM obezbeđuje skup komunikacionih interfejsa na nivou transakcije i kanala koji se mogu koristiti za njihovo povezivanje. Upotreba TLM interfejsa izoluje svaku komponentu od promena u drugim komponentama u okruženju.
- U kombinaciji sa faznom, fleksibilnom infrastrukturom u UVM-u, TLM promoviše ponovnu upotrebu (reuse) tako što dozvoljava bilo kojoj komponenti da se zameni drugom, sve dok ima isti interfejs.
- Ovaj koncept takođe omogućava da se UVM okruženja za verifikaciju poklapaju sa modelom DUT-a na nivou transakcija.
- Na kraju je neophodno da se model nivoa transakcije zameni nivoom kompatibilnih komponenti koje konvertuju aktivnosti na nivou transakcije u aktivnosti na nivou pinova na DUT-u.

Transakcija

- UVM transakcija je objekt klase koji uključuje sve informacije koje su potrebne za modelovanje osnovne komunikacija između dve komponente.
- U najosnovnijem primeru, jednostavna transakcija recimo Bus protokola sadržala bi sledeće informacije:

```
class simple_trans extends uvm_transaction;
  `uvm_object_utils(simple_trans)

  rand data_t data;
  rand addr_t addr;
  rand enum {WRITE,READ} kind;
  constraint c1 { addr < 16'h2000; }

  ...
endclass
```

Transakcije

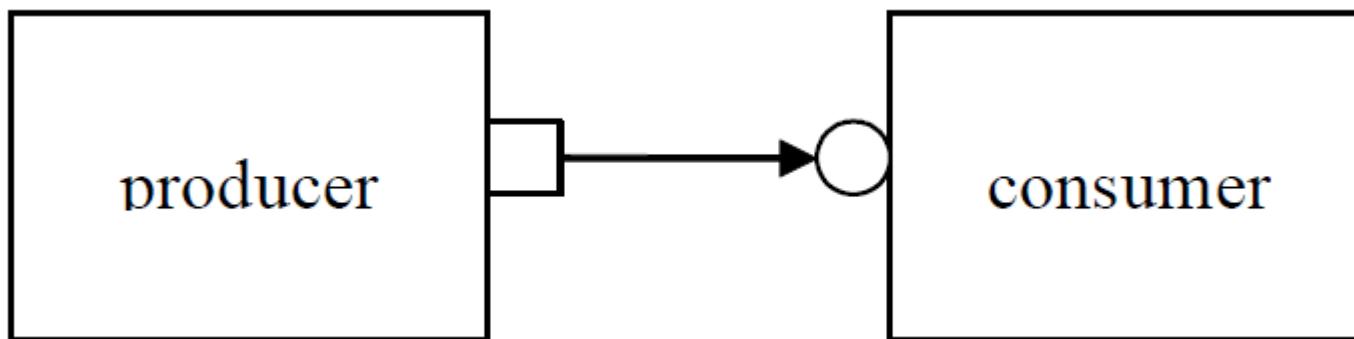
- Objekt transakcije uključuju varijable, ograničenja i metode potrebne za generisanje i radi na transakciji.
- Često je potrebno više od ovih osnovnih informacija da bi se u potpunosti definisala transakcija na Bus-u. Na primer, transakcija `simple_trans` se može proširiti sa više informacija, kao što su broj stanja čekanja pred slanje, veličina prenosa... Transakcija bi se takođe mogla proširiti i na dodatna ograničenja.
- Takođe je moguće definisati transakcije višeg nivoa koje uključuju određeni broj transakcija nižeg nivoa.
- Transakcije se stoga mogu sastaviti, dekomponovati, proširiti, postaviti slojevito i manipulisati modelom bilo koje komunikacije na bilo kom nivou apstrakcije.

Komunikacija na nivou transakcije

- Interfejsi na nivou transakcija definišu skup metoda koje koriste objekte transakcija kao argumente.
- TLM *port* definiše skup metoda koje će se koristiti za određeni metod prenosa, dok TLM *export* obezbeđuje implementaciju tih metoda.
- Povezivanje *port*-a sa *export*-om omogućava da se komunikacija izvrši kada se pozove metoda porta.

Osnovna TL komunikacija

- Najosnovnija operacija na nivou transakcija omogućava jednoj komponenti da prosledi transakciju drugoj komponenti.



- Kvadrat označen na producer objektu port, dok i krug označen na consumeru označava export. Export u ovom kontekstu možemo shvatiti kao externi port koji je povezan na to mesto.

```

class producer extends uvm_component;
  uvm_blocking_put_port #(simple_trans) put_port; // 1 parameter
  function new( string name, uvm_component parent);
    put_port = new("put_port", this);
    ...
  endfunction
  virtual task run();
    simple_trans t;
    for(int i = 0; i < N; i++) begin
      // Generate t.
      put_port.put(t);
    end
  endtask

```

- U ovakvoj implementaciji direktne veze producer-a i consumer-a task put() je definisan na strani consumera.

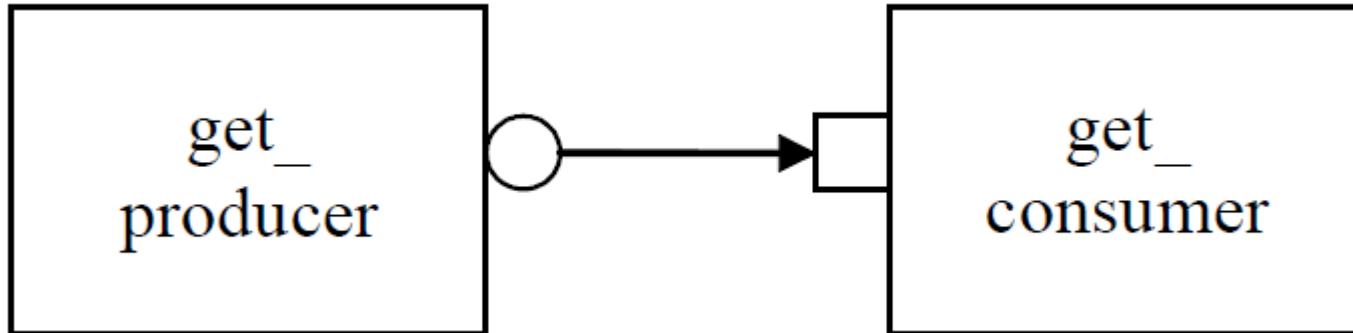
```
class consumer extends uvm_component;
  uvm_blocking_put_imp #(simple_trans, consumer) put_export; // 2 parameters
  ...
  task put(simple_trans t);
    case(t.kind)
      READ: // Do read.
      WRITE: // Do write.
    endcase
  endtask

endclass
```

- Direktno povezivanje može napraviti blokirajuće probleme. Elastična veza uvođenjem FIFO kanala po imenu uvm_tlm_fifo razdvaja izvor i prijemnik.

- Videli smo komunikaciju upotrebom put_port i export-a. Izvor informacije preko svog put_port-a poziva put task sa consumer strane.
- Na sličan način može se uspostaviti kanal komunikacije u kom će pokretač biti consumer uotrebom get taska kojeg će pozivati.

Consumer poziva task get()



- Sada je kvadratić na strani consumer-a, pošto on inicira preuzimanje transakcije od producer-a.

- Consumer zahteva transakciju od producer-a preko svog get port-a.

```
class get_consumer extends uvm_component;
    uvm_blocking_get_port #(simple_trans) get_port;
    function new( string name, uvm_component parent);
        get_port = new("get_port", this);
        ...
    endfunction
    virtual task run();
        simple_trans t;
        for(int i = 0; i < N; i++) begin
            // Generate t.
            get_port.get(t);
        end
    endtask
```

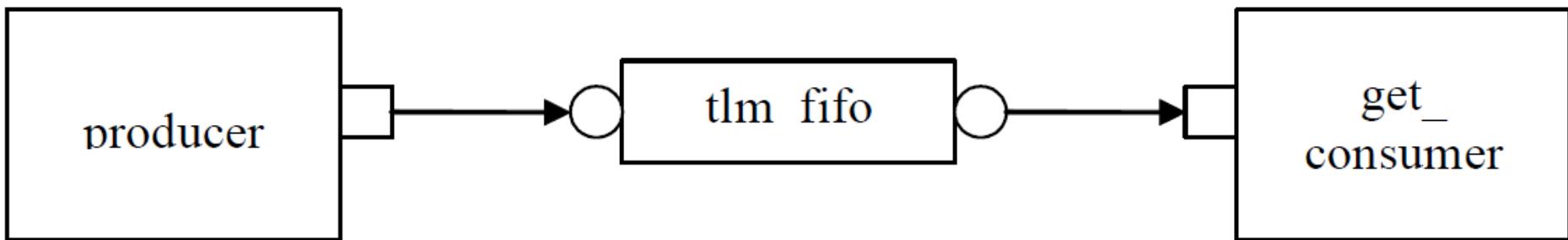
- Implementacija get() metode na strani producer-a.

```
class get_producer extends uvm_component;
  uvm_blocking_get_imp #(simple_trans, get_producer) get_export;
  ...
  task get(output simple_trans t);
    simple_trans tmp = new();
    // Assign values to tmp.
    t = tmp;
  endtask
endclass
```

Komunikacija korišćenjem uvm_tlm_fifo

- Često može biti potrebno da komponente rade nezavisno, pri čemu producer generiše transakcije u jednom procesu, dok consumer treba da barata sa tim transakcijama u drugom.
- UVM pruža kanal uvm_tlm_fifo za olakšavanje takve komunikacije.
- uvm_tlm_fifo implementira sve metode TLM interfejsa, tako da producer postavlja transakciju u uvm_tlm_fifo, dok consumer nezavisno preuzima transakciju iz FIFO.
- uvm_tlm_fifo ima dubinu od 1 transakcije

- Kada producer postavi transakciju u FIFO, blokiraće ako je FIFO pun, u protivnom postaviće transakciju u FIFO i odmah nastaviti.
- Operacija get će se odmah obaviti ako je transakcija dostupna (i zatim će biti uklonjena iz FIFO), u suprotnom će blokirati sve dok se transakcija ne pojavi u FIFO.
- Ukoliko uzastopno dva puta poziva get () consumer će dobiti dve različite transakcije.
- Ukoliko isto pokuša sa pozivom peek () metode, dobiće dvostruku kopiju dostupne transakcije bez njenog uklanjanja.



Blokirajuća i neblokirajuća komunikacija

- Interfejsi koje smo do sada posmatrali blokiraju izvršavanje taska dok se ne završe; nije im dozvoljen neuspeh.
 - Ne postoji mehanizam za bilo koji blokirajući poziv da se prekine.
 - Oni jednostavno čekaju da se zahtev izvrši.
 - U vremenskom smislu, ovo znači da može proći vreme između trenutka kada je poziv pokrenut i vremena kada se poziv vrati.
- Nasuprot tome, neblokirajući poziv se odmah vraća.
 - Semantika neblokirajućeg poziva garantuje da se poziv vraća u istom delta ciklusu u koem je pozvan, tj. bez utroška vremena.
 - U UVM-u, neblokirajući pozivi su modelovani kao funkcije.

Neblokirajuće metode

- Ako postoji transakcija, ona će biti vraćena u argument, dok će poziv funkcije će vratiti TRUE.
- Ako transakcija nepostoji, funkcija će vratiti FALSE.
- Slično je i sa try_peek () metodom.
- try_put () metoda vraća TRUE ako je transakcija poslata.

```
class consumer extends uvm_component;
    uvm_get_port #(simple_trans) get_port;
task run;
    ...
    for(int i=0; i<10; i++)
        if(get_port.try_get(t))
            //Do something with t.
    ...
endtask
endclass
```

Povezivanje TL komponenti

- Kada su port-ovi i export-ovi definisani za komponente na nivou transakcije, stvarna veza između njih se ostvaruje putem metode connect() u hijerarhijski višoj komponenti ili env, s argumentom koji je objekat (port ili export) na koji će biti povezan.
- U verifikacionom environment-u, niz connect () poziva između port-ova i export-ova uspostavlja netlistu peer-to-peer i hijerarhijskih veza.
- Dakle kad komponente pozove
 - put_port.put(t);
- Zapravo se konekcijom poziva
 - target.put_export.put(t);
- Gde je target konektovana komponenta

- Pri povezivanju komponenti na istom nivou hijerarhije, port je uvek povezani na export. Svi connect () pozivi između komponenti se obavljaju u roditeljskoj metodi connect ().

```
function void connect_phase(uvm_phase phase);
    producer_h.put_port_h.connect(fifo_h.put_export);
    consumer_h.get_port_h.connect(fifo_h.get_export);
endfunction : connect_phase
```

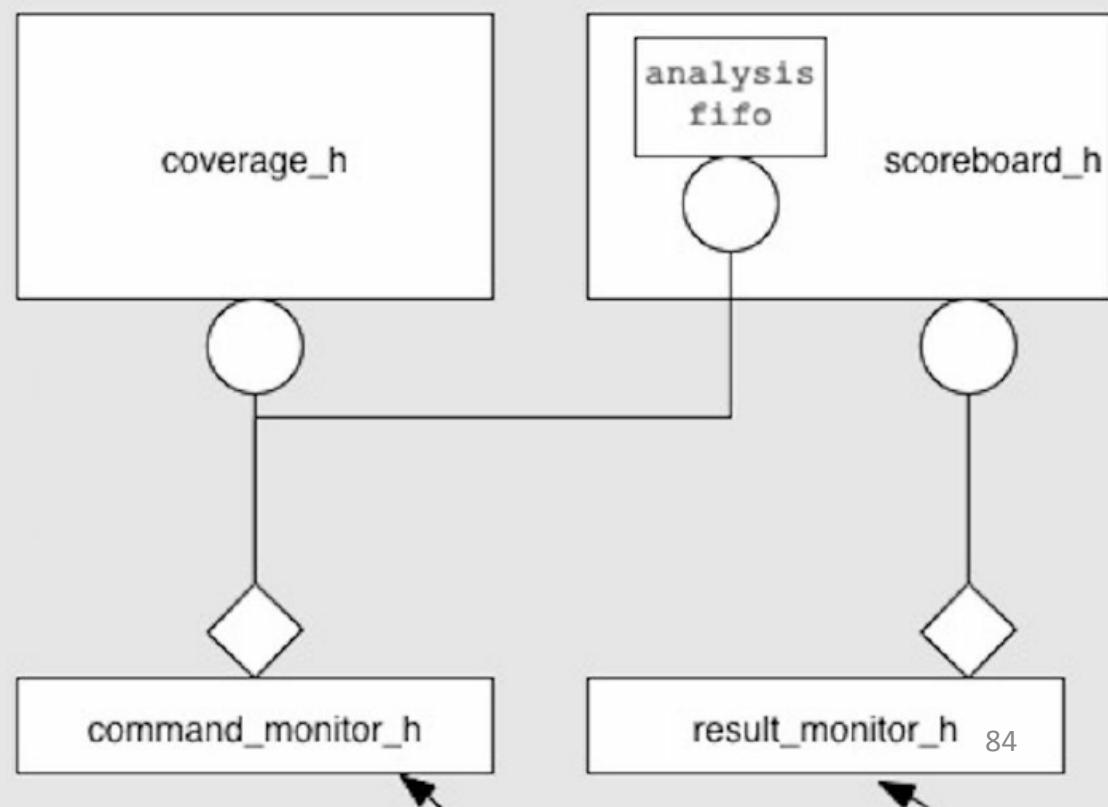
- Dodatna prednost TLM komunikacije u UVM-u je ta da su kompatibilnosti svih TLM konekcija proverene pre pokretanja testa.
- Da bi konekcija bila validna, export mora da obezbedi implementacije za najmanje skup metoda koje definiše port i naravno tip transakcije kao parametar mora biti identičan u oba slučaja.
- Na primer, `blocking_put_port`, koji zahteva implementaciju `put()` može biti povezan na `blocking_put_export` ili `put_ekport`. Oba export-a obezbeđuju implementaciju metode `put()`, takođe `put_export` obezbeđuje implementacije `try_put()` i `can_put()` metoda.

Komunikacija u procesu analize i verifikacije

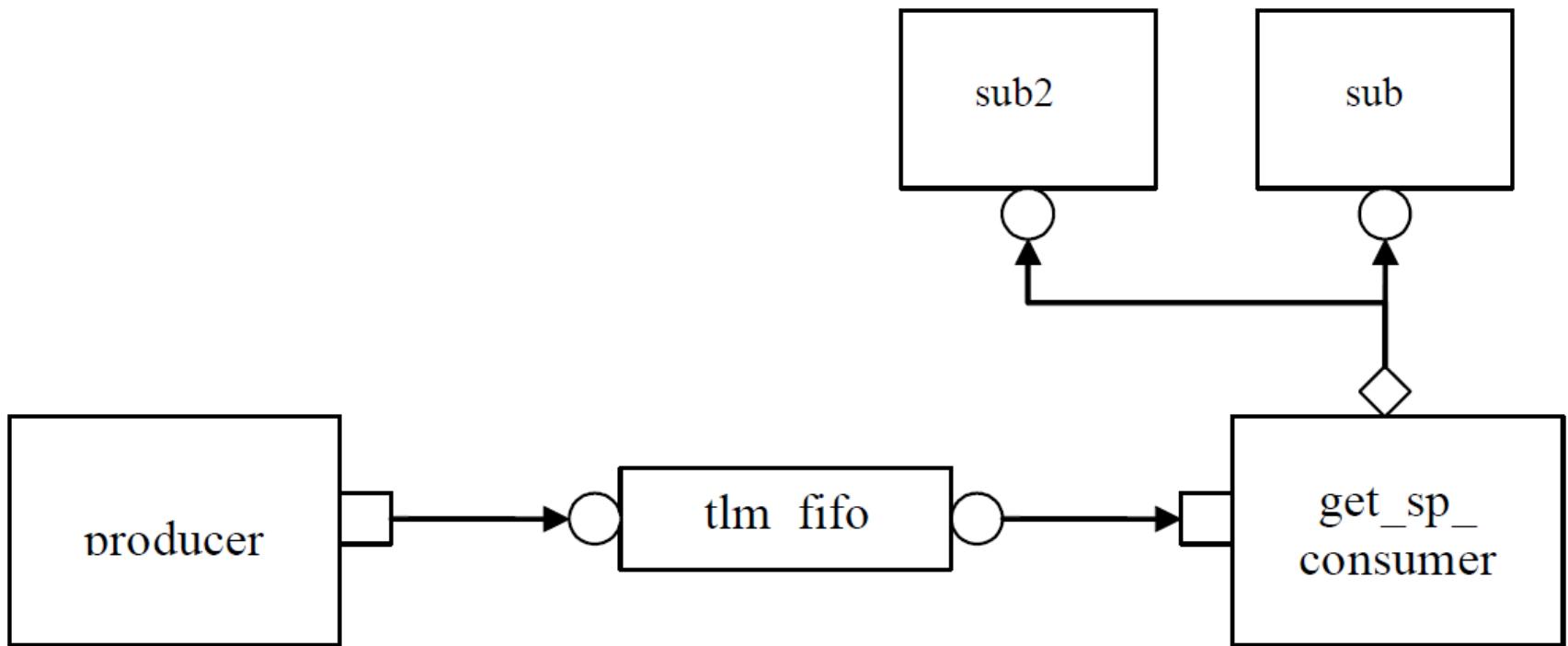
- put/get komunikacija kao što je prethodno opisano omogućava da se kreiraju verifikacione komponente koje modeluju operativno ponašanje sistema.
- Svaka komponenta je odgovorna za komunikaciju kroz TLM interfejs sa drugim komponentama u sistemu kako bi se stimulisala aktivnost DUT-a, odnosno odgovorila na njegovo ponašanje.
- Međutim, u složenijim okruženjima za verifikaciju, posebno prilikom korišćenja randomizacije, prikupljene transakcije potrebno je distribuirati okolini za finalnu proveru (scoreboard), ili za proveru pokrivenosti.
- Ključna razlika između dva tipa TLM komunikacije je u tome što portovi put/get obično zahtevaju odgovarajući export.
- Za analizu, međutim, fokus je na određenim komponentama, tipično to je monitor, koji generiše tok transakcija, bez obzira da li postoje komponente zakačene na taj tok transakcija ili ne.
- Tok transakcija se preko analysis_port-a povezuje na prateće komponente, takozvane komponente modularne analize. To povezivanje se obavlja preko analysis_port-ova.

Analysis port-ovi

- uvm_analysis_port (predstavljen kao dijamant na monitoru) je specijalizovani TLM port čiji se interfejs sastoji od jedne funkcije, write(). Analysis port sadrži listu analysis_export-a koji su povezani sa njim.



analysis_port analysis_export primer



- Kada komponenta pozove analysis_port.write(), analysis_port kruži kroz listu i poziva metodu write() svakog povezanog export-a.
- Ukoliko ništa nije povezano, write() poziv se jednostavno vraća.
- Stoga, analysis port može biti povezan na nulu, jedan ili više analysis export-a, ali rad komponente koja piše u analysis port ne zavisi od broja povezanih export-a.
- Pošto je write() void funkcija, poziv će uvek biti završen u istom delta ciklusu, bez obzira na to koliko komponenti je povezano na dati analysis port.

Mehanizam pisanja transakcije na analysis_port

```
class command_monitor extends uvm_component;
  `uvm_component_utils(command_monitor);

  uvm_analysis_port #(command_s) ap;

  function void build_phase(uvm_phase phase);
    virtual tinyalu_bfm bfm;

    if(!uvm_config_db #(virtual tinyalu_bfm)::get(null, "*", "bfm", bfm))
      $fatal("Failed to get BFM");

    bfm.command_monitor_h = this;
    ap = new("ap",this);

  endfunction : build_phase

  function void write_to_monitor(byte A, byte B, bit[2:0] op);
    command_s cmd;
    cmd.A = A;
    cmd.B = B;
    cmd.op = op2enum(op);
    $display("COMMAND MONITOR: A:0x%2h B:0x%2h op: %s", A, B, cmd.op.name());
    ap.write(cmd);
  endfunction : write_to_monitor
```

analysis_export mehanizam

- Kao i kod ostalih TLM konekcija, svaka komponenta koja je povezana sa analysis_port-om obezbeđuje implementaciju write() funkcije preko analysis_export-a.
- Osnovna komponenta uvm_subscriber je namenjena da obezbedi jednostavnu realizaciju ove operacije, tako da tipična komponenta za analizu upravo nasleđuje uvm_subscriber

```
class sub1 #(type T = simple_trans) extends uvm_subscriber #(T);  
...  
function void write(T t);  
    // Call desired functionality in parent.  
endfunction  
endclass
```

- Kao i kod put() i get() opisanih u delu vezanom za port i export konekcije, TLM veza između analysis_port-a i analysis_export-a, omogućava analysis_export-u da implementira svoju write() funkciju.
- Ako je višestruki analysis_export povezan sa analysis_port-om, analysis_port će pozvati write() svakog analysis_export-a, u nizu.
- U svim implementacijama write() mora biti funkcija, ona se izvršava trenutno.

- Ilustracija povezivanja izvora **g** sa dva pratioca **s1** i **s2** preko `analysis_port` / `analysis_export` konekcija u okviru `env` objekta

```
class my_env extends uvm_env;
    get_component_with_ap g;
    sub1 s1;
    sub2 s2;
    function new(string name, uvm_component parent) ;
        super.new(name,parent);
        s1 = new("s1");
        s1.env = this ;
        s2 = new("s2");
        s2.env = this;
    endfunction
    function void connect_phase(uvm_phase phase);
        g.ap.connect(s1.analysis_export);
        // to illustrate analysis port can be connected to multiple
        // subscribers; usually the subscribers are in separate components
        g.ap.connect(s2.analysis_export);
        ...
    endfunction
endclass
```

- Kada je više pratilaca povezano sa analysis_port-om, svakom se prenosi pokazivač na isti objekat transakcije, kao argument za njegovu write() funkciju.
- Svaka write() implementacija pravi lokalnu kopiju transakcije, a zatim radi nad kopijom da bi se izbeglo menjanje sadržaja transakcije za bilo kog drugog pratioca.
- Veza između analysis_port-a i exporta inherentno uključuje uvm_tlm_analysis_fifo.
- Dubina uvm_tlm_analysis_fifo objekta je neograničena, što obezbeđuje nesmetani upis transakcija, dok pratioci koji vrše analizu mogu da čitaju samo ukoliko fifo nije prazan..

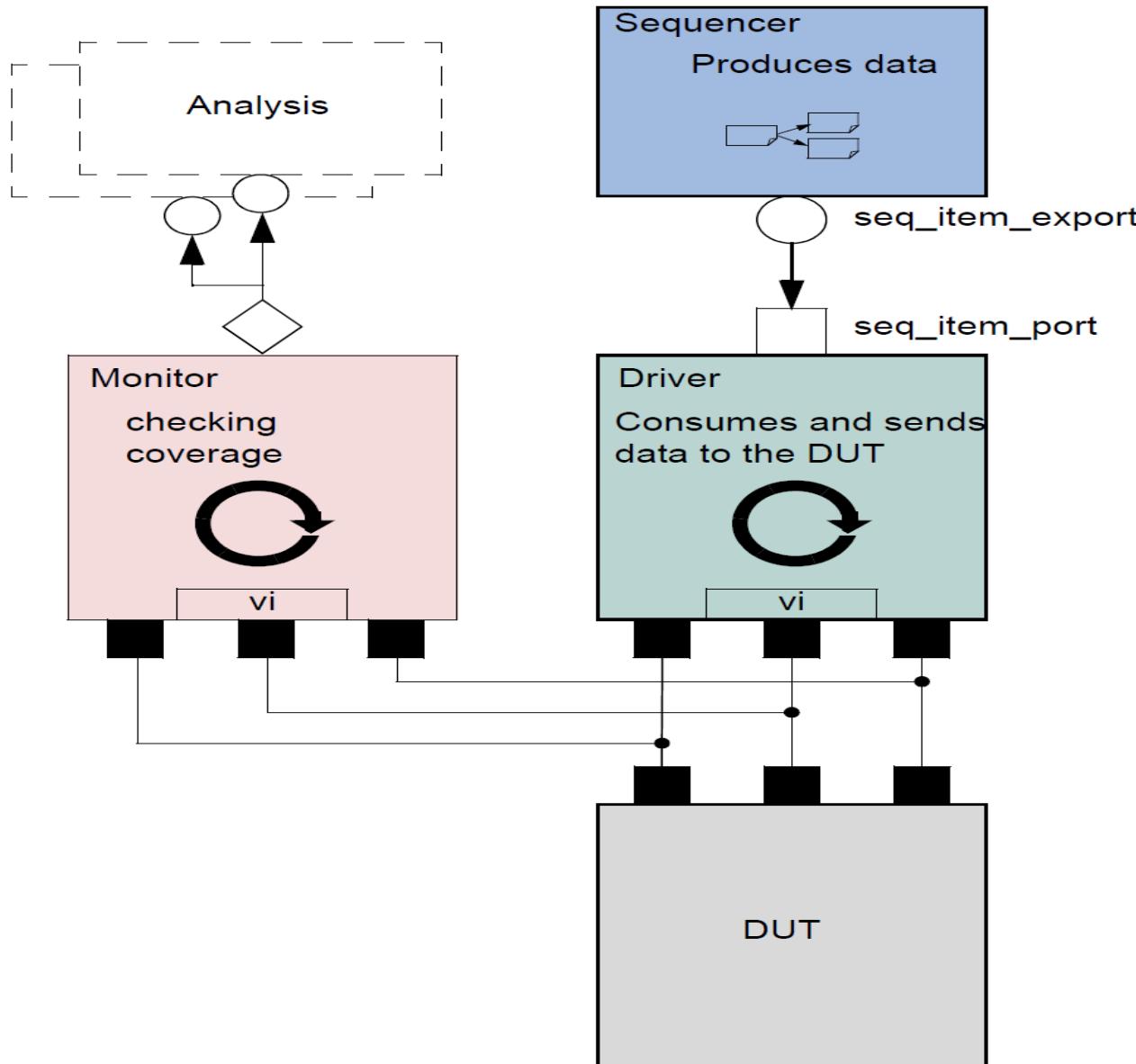
Sekvence

- uvm_sequence predstavlja proširenje/ naslednika uvm_transaction klase
- uvm_sequence_item sadrži dodatne varijable koje dozvoljavaju objektu da se koristi na sekvenceru / driver-u.
- uvm_sequence_item osnovna klasa ima polje m_sequence_id koje je neophodno u dvosmernim protokolima tako da sekvencer može da vrati odgovor nazad na izvornu sekvencu
- Preporuka je da se stoga za šablonski kreirane UVM testbenčeve koriste sekvence kao osnovni apstraktni izvori transakcija
- Više o sekvencama, nakon detaljnijeg pregleda TL komponenti.

Komponente na transakcionom nivou

- TLM interfejsi u UVM-u obezbeđuju konzistentan skup metoda komunikacije za slanje i primanje transakcija između komponenti.
- Sami delovi su instancirani i povezani u testbenču, da bi obavili različite operacije potrebne za verifikaciju DUT-a.
- Pojednostavljeni testbench je prikazan na sledećem slajdu (zbog jednostavnosti tipični blokovi kao što su environment i agenti nisu prikazano ovde).

Tipične komponente na transakcionom nivou

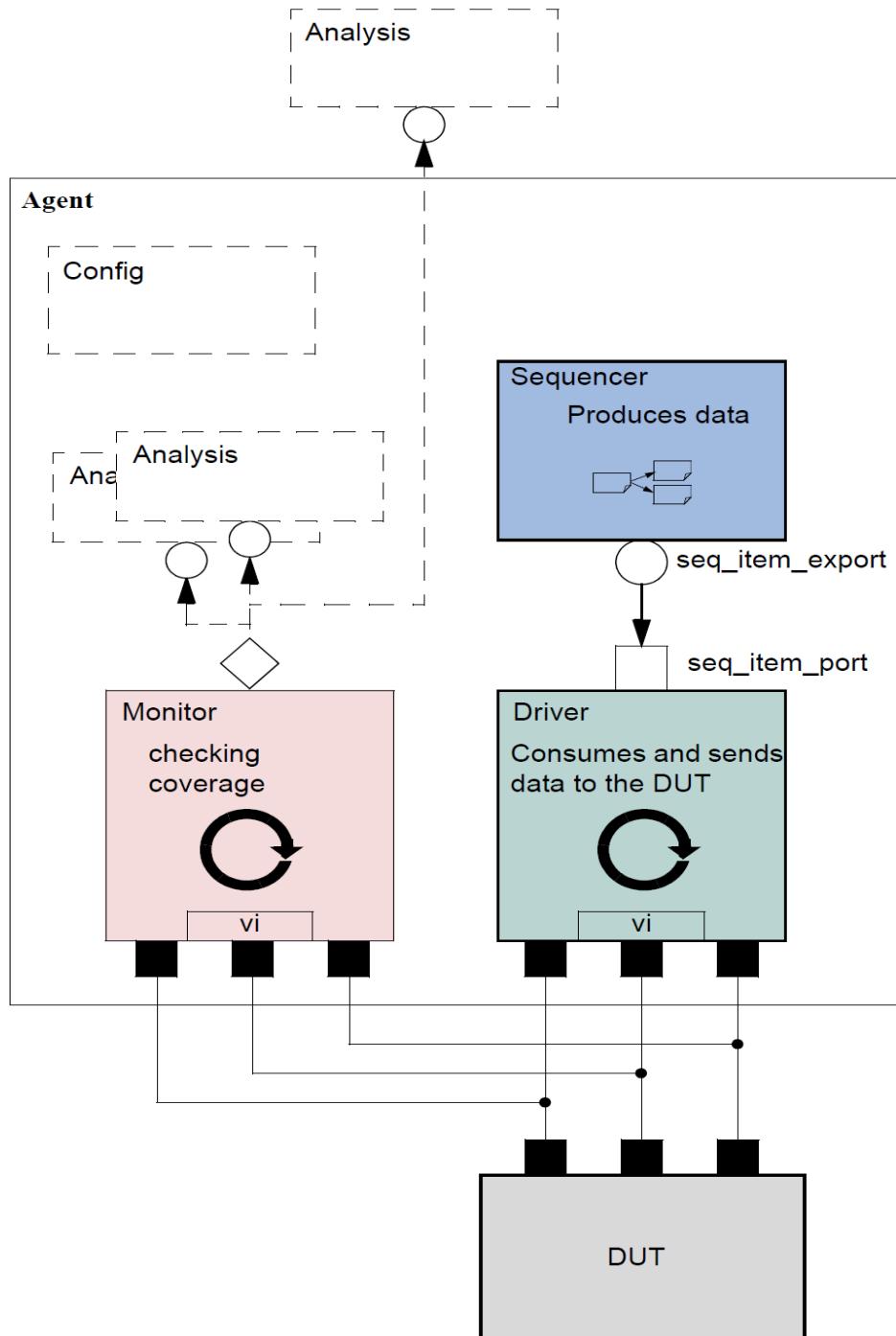


Osnovne komponente jednostavnog okruženja za verifikaciju na nivou transakcije

- a) Generator stimulusa (sekvencer) za stvaranje saobraćaja na nivou transakcije prema DUT-u.
- b) Driver za pretvaranje tih transakcija u stimulus na nivou signala na DUT interfejsu.
- c) Monitor koji prepoznaje aktivnosti nivoa signala na DUT interfejsu i pretvara ih u transakcije.
- d) Komponente za analizu, kao što su scoreboard ili coverage, za analizu transakcija.

- Konzistentnost i modularnost TLM interfejsa u UVM-u dozvoljava da komponente budu ponovo korišćene, zamenjene ili enkapsulirane.
- Svaku komponentu karakterišu njeni interfejsi, bez obzira na njihovu internu implementaciju.
- Enkapsuliranje ovih tipova komponenti u standardizovanu šablonsku UVM arhitekturu, poboljšava iskoristivost i efikasnost.

Arhitektura agenta



- Preporučljivo je grupisanje pojedinačnih komponenti na viši nivo.
- Umesto ponovne upotrebe klasa niskog nivoa, kreira se komponenta koja obuhvata pod-klase na konzistentan način.

Kreiranje Driver-a

- Uloga driver-a je da postavlja podatke na adekvatnu magistralu preko interfejsa. Driver dobija podatke iz sekvencera.
- UVM biblioteka klasa obezbeđuje osnovnu klasu `uvm_driver`, iz koje sve driver klase treba da budu izvedene, bilo direktno ili indirektno.
- Driver ima TLM port preko kog komunicira sa sekvencerom.
- Driver takođe može da koristi jednu ili više faza izvršavanja (`run`, `pre_reset`, `post_shutdown`...) da bi se unapredila njegova funkcionalanost.

Koraci u kreiranju Driver-a

- Izvodi se iz osnovne klase `uvm_driver`.
- Po potrebi, dodaju mu se UVM infrastrukturni makroi za implementaciju metoda za štampanje, kopiranje, komparaciju...
- Priprema mu se naredni podatak iz sekvencera i predaje mu se na izvršenje.
- Potrebno je deklarisati virtualni interfejs na Driver-u putem kog će biti povezan sa DUT-om

- Klasa `simple_driver` u narednom primeru definiše klasu željenog driver-a.
- `simple_driver` klasa je izvedena iz `uvm_driver` klase (parametrizovana tako da koristi tip transakcije `simple_item`) i koristi metode u objektu `seq_item_port` da komunicira sa sekvencerom.
- Uključuje se konstruktor i ``uvm_component_utils` makro da registruju tip driver-a pri fabrici.

```
1 class simple_driver extends uvm_driver #(simple_item);
2   simple_item s_item;
3   virtual dut_if vif;
4   // UVM automation macros for general components
5   `uvm_component_utils(simple_driver)
6   // Constructor
7   function new (string name = "simple_driver", uvm_component parent);
8     super.new(name, parent);
9   endfunction : new
10  function void build_phase(uvm_phase phase);
11    string inst_name;
12    super.build_phase(phase);
13    if(!uvm_config_db#(virtual dut_if)::get(this,
14          "", "vif", vif))
15      `uvm_fatal("NOVIF",
16                  {"virtual interface must be set for: ",
17                   get_full_name(),".vif"});
18  endfunction : build_phase
19  task run_phase(uvm_phase phase);
20    forever begin
21      // Get the next data item from sequencer (may block).
22      seq_item_port.get_next_item(s_item);
23      // Execute the item.
24      drive_item(s_item);
25      seq_item_port.item_done(); // Consume the request.
26    end
27  endtask : run
28
29  task drive_item (input simple_item item);
30    ... // Add your logic here.
31  endtask : drive_item
32 endclass : simple_driver
```

Tumačenje primera po linijama

- 1 simple_driver nasleđuje uvm_driver
- 5 Dodaje se UVM infrastrukturni makro.
- 13 preuzima se hendler na virtualni interfejs
- 22 Poziva se get_next_item() da bi se dobila sledeća transakcija (sequence_item) za izvršenje od sekvencera.
- 25 Signalizira sekvenceru da je izvršenje trenutne transakcije obavljenog.
- 30 Realizuje logiku specifičnu za dati komunikacioni protokol.

Kreiranje Sequencer-a

- Sekvencer generiše podatke stimulusa i prosleđuje ih Driver-u za izvršenje.
- Biblioteka UVM klasa nudi osnovnu klasu `uvm_sequencer`, koja je parametrizovana `request` i `response` tipovima stavki-itema.
- Osnovna klasa `uvm_sequencer` sadrži sve osnovne funkcionalnosti koje su potrebne da bi se sekvenci omogućila komunikacija sa driver-om.
- `Uvm_sequencer` se instancira direktno, sa odgovarajućim parametrima.
- U definiciji klase, podrazumevano, tip odgovora je isti kao i tip zahteva.
- Ukoliko je potreban drugačiji tip odgovora, opcioni drugi parametar koji ga opisuje mora biti naveden u okviru osnovnog tipa `uvm_sequencer`-a:

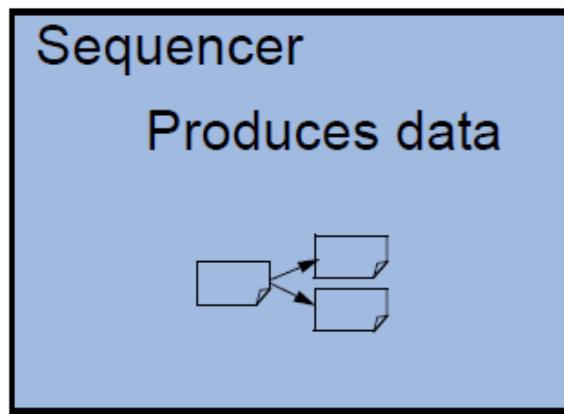
```
uvm_sequencer #(simple_item, simple_rsp) sequencer;
```

Povezivanje Driver-a i Sequencer-a

- Driver i sekvencer su povezani preko TLM-a, pri čemu je seq_item_port driver-a povezan sa sequencer-ovim seq_item_export.
- Sequencer generiše item-e podataka koji se postavljaju na njegov export.
- Driver preuzima data item-e preko svog seq_item_port-a i opcionalno, vraća odgovore.
- Komponenta koja sadrži instance driver-a i sequencera obezbeđuje vezu između njih.

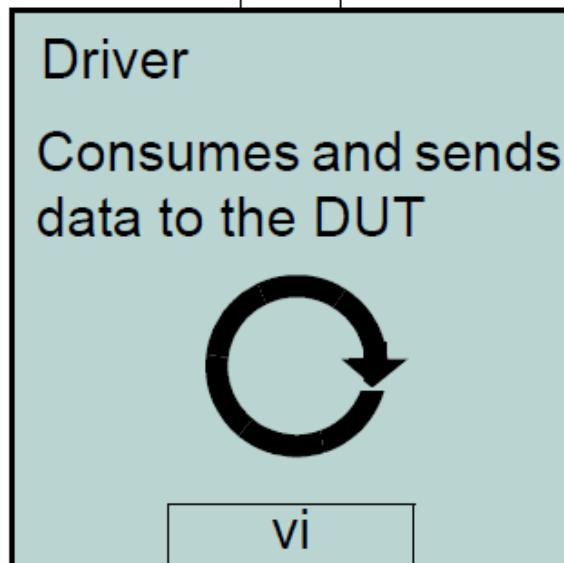
Veza

Driver - Sequencer



uvm_seq_item_pull_export
seq_item_export

uvm_seq_item_pull_port
seq_item_port



- Seq_item_port u uvm_driver-u definiše skup metoda koje koristi driver za dobijanje sledećeg sequence item-a.
- Važan deo ove interakcije je sposobnost driver-a da se sinhronizuje sa magistralom i za interakciju sa sequencerom za generisanje data item-a u odgovarajuće vreme.
- Sequencer implementira skup metoda koje omogućavaju fleksibilnu i modularnu interakciju između driver-a i sequencera.

Osnovna interakcija Sekvencer - Driver

- Osnovna interakcija između driver-a i sequencera se vrši pomoću taskova `get_next_item()` i `item_done()`.
- Driver koristi `get_next_item()` da dobije sledeći randomizovani data item za slanje.
- Nakon slanja na DUT, driver signalizira sequenceru da je data item obrađen pomoću `item_done()`.

Tipično, glavna petlja unutar driver-a liči na sledeći pseudo kod.

```
forever begin
    get_next_item(req);
    // Send item following the protocol.
    item_done();
end
```

Podsetnik:

`get_next_item()` je blokirajući poziv. Blokada traje sve dok sequencer ne obezbedi naredni data item / transakciju iz sekvence koja se na njemu izvršava.

Neblokirajući zahtev ka sequencer-u

- Pored taska `get_next_item()`, klasa `uvm_seq_item_pull_port` nudi i drugi task, `try_next_item()`. Ovaj taske se vraća u istom koraku simulacije ako nema dostupnih podataka za izvršenje.
- Ovaj task se može koristiti da bi driver izvršio neke idle transakcije, dakle stimulaciju DUT-a u periodu kada nema smislenih podataka za prenos.
- Naredni primer pokazuje upravo ovaj scenario:

```
task run_phase(uvm_phase phase);
    forever begin
        // Try the next data item from sequencer (does not block).
        seq_item_port.try_next_item(s_item);
        if (s_item == null) begin
            // No data item to execute, send an idle transaction.
            ...
        end
        else begin
            // Got a valid item from the sequencer, execute it.
            ...
            // Signal the sequencer; we are done.
            seq_item_port.item_done();
        end
    end
endtask: run
```

Slanje odgovora - obrađenih podataka nazad sequencer-u

- U nekim sekvencama, generisana vrednost item-a podataka zavisi od odgovora na prethodno generisane podatke.
- Podrazumevano, data item-i između driver-a i sequencera kopiraju se referencom, što znači da ukoliko driver izvrši promenu nad podacima to će biti vidljivo unutar sequencera.
- U slučajevima kada je data item između driver-a i sequencera kopiran po vrednosti, driver mora da vrati obrađen odgovor nazad u sequencer.
Primer je dat na sledećoj strani:

- `seq_item_port.item_done(rsp);`

Ili korišćenjem `put_response()` metode

- `seq_item_port.put_response(rsp);`
- Pre postavljanja odgovora, sekvenca odgovora i identifikacija transakcije moraju biti podešeni tako da odgovaraju inicijalnoj transakciji.
- Za to se koristi `rsp.set_id_info(req)`.
- `put_response()` je blokirajuća metoda, tako da sequence mora uraditi odgovarajući `get_response(rsp)`

Kreiranje Monitor-a

- Monitor je odgovoran za ekstrakciju informacija sa magistrale i prevođenje te informacije u transakciju odnosno data item.
- Ove informacije u obliku transakcije postaju dostupne drugim UVM komponentama putem TLM interfejsa i kanala.
- Monitor ne bi trebao da se oslanja na informacije koje su prikupile druge komponente, kao što je recimo driver, ali će možda morati da se oslanjaju na informacije o ID-ju specifičnom za zahtev date sekvence koja se izvršava i na podatke o transakciji za odgovor.
- Funkcionalnost monitora treba da bude ograničena na osnovni nadzor magistrale koji je uvek potreban.
- Ovo može da uključi proveru protokola.
- Dodatna funkcionalnost na visokom nivou, što podrazumeva scoreboard, treba da bude implementirana odvojeno na vrhu hijerarhije monitora.
- Ako treba verifikovati apstraktни model ili ubrzati funkcionalnost pin-nivoa, trebalo bi razdvojiti detekciju signalnog nivoa, pokrivenost od provera i aktivnosti na nivou transakcija.
- Analysis port omogućava komunikaciju između komponenti preko TLM-a.

Primer monitora

- Sledeći primer prikazuje jednostavan monitor koji ima sledeće funkcije:
- Monitor prikuplja informacije o magistrali preko virtuelnog interfejsa (xmi).
- Prikupljeni podaci se koriste u prikupljanju i proveri pokrivenosti.
- Prikupljeni podaci se export-uju na analysis port (item_collected_port).

```

class master_monitor extends uvm_monitor;
    virtual bus_if xmi; // SystemVerilog virtual interface
    bit checks_enable = 1; // Control checking in monitor and interface.
    bit coverage_enable = 1; // Control coverage in monitor and interface.
    uvm_analysis_port #(simple_item) item_collected_port;
    event cov_transaction; // Events needed to trigger covergroups
protected simple_item trans_collected;
`uvm_component_utils_begin(master_monitor)
    `uvm_field_int(checks_enable, UVM_ALL_ON)
    `uvm_field_int(coverage_enable, UVM_ALL_ON)
`uvm_component_utils_end
covergroup cov_trans @cov_transaction;
    option.per_instance = 1;
    ... // Coverage bins definition
endgroup : cov_trans
function new (string name, uvm_component parent);
    super.new(name, parent);
    cov_trans = new();
    cov_trans.set_inst_name({get_full_name(), ".cov_trans"});
    trans_collected = new();
    item_collected_port = new("item_collected_port", this);
endfunction : new

```

```
virtual task run_phase(uvm_phase phase);
    collect_transactions(); // collector task.
endtask : run
virtual protected task collect_transactions();
forever begin
    @ (posedge xmi.sig_clock);
    ...// Collect the data from the bus into trans_collected.
    if (checks_enable)
        perform_transfer_checks();
    if (coverage_enable)
        perform_transfer_coverage();
    item_collected_port.write(trans_collected);
end
endtask : collect_transactions
virtual protected function void perform_transfer_coverage();
    -> cov_transaction;
endfunction : perform_transfer_coverage
virtual protected function void perform_transfer_checks();
    ... // Perform data checks on trans_collected.
endfunction : perform_transfer_checks
endclass : master_monitor
```

- Prikupljane transakcija se vrši u tasku (`collect_transactions`) koji se postavlja na početku `run()` phase.
- Task radi u beskonačnoj petlji i prikuplja podatke čim se signali pojave na magistrali.
- Prikupljene transakcije se šalju na analysis port (`item_collected_port`) odakle će ih preuzeti druge komponente koje ih čekaju.
- Prikupljanje i provera pokrivenosti su uslovni jer mogu uticati na performanse simulacije.
- Ako nije potrebno, mogu se isključiti postavljanjem `coverage_enable` ili `checks_enable` na 0, korišćenjem mehanizma konfiguracije. Na primer:

```
uvm_config_int::set(this,"*.master0.monitor", "checks_enable", 0);
```

- Ako je provera omogućena, task poziva funkciju `perform_transfer_checks`, koja izvršava potrebne provere prikupljenih podataka (`trans_collected`).
- Ako je omogućeno prikupljanje pokrivenosti task trigeruje event semplovanja pokrivenosti (`cov_transaction`) koji rezultuje prikupljanjem trenutnih vrednosti.
- SV ne dozvoljava istovremene assertion-e (tvrđnje) u klasama, tako da se provere protokola mogu raditi pomoću assertion-a (tvrđnji) u SV interfejsu.

Instanciranje komponenti

- Izolacija koju pruža OO tehnika i TLM interfejsi između komponenti olakšava reusability (ponovnu upotrebu) u UVM-u što daje veliku fleksibilnost u formiranju okruženja.
- Obzirom na nezavisnost komponenti među sobom, svaka se može zamjeniti novom komponentom sa istim interfejsom bez potrebe za promenom roditeljskog metoda connect () .
- Ova fleksibilnost se postiže kroz upotrebu Factory koncepta (fabrike) u UVM-u.

```

class my_component extends uvm_component;
    my_driver driver;
    ...
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        driver = new("driver",this);
        ...
    endfunction
endclass

```

- Instanciranje pozivanjem konstruktora – gore
- Instanciranje pozivanjem create() metode -dole

```

class my_component extends uvm_component;
    my_driver driver;
    ...
    virtual function void
        build_phase(uvm_phase phase);
        super.build_phase(phase);
        driver = my_driver::type_id::create("driver",this);
        ...
    endfunction
endclass

```

Preporučljiviji metodinstanciranje je korišćenje Factory-ja (fabrike)

- UVM obezbeđuje ugrađen Factory (fabriku) koja omogućava komponentama da kreiraju objekte bez određivanja tačne klase objekat koji se generiše.
- Mehanizam koji se koristi označava se kao override (prepisivanje), a ono može biti po instanci ili po tipu.
- Ova funkcionalnost je korisna za variranje funkcionalnosti sekvene ili za promenu jedne verzije komponente drugom.
- Sve komponente koje se zamenjuju moraju biti polimorfno kompatibilne.
- Ovo podrazumeva da svi novi hendleri TLM interfejsa i TLM objekti moraju biti kreirani od strane nove prepisane komponente.
- Fabrika obezbeđuje ovu mogućnost sa statickom funkcijom dodele koja se može koristiti umesto postojeće new funkcije.
- Funkcija koju nudi fabrika je:

type_name::type_id::create(string name, uvm_component parent)

- Obzirom da je metoda create () automatski vezana za tip, može se koristiti za kreiranje komponenti ili objekata.
- Kada se kreiraju objekti, drugi argument, prethodnik, je opcionalan.
- Komponenta koja koristi fabriku za kreiranje objekata podataka izvršava kod koji sledi:

```

task mycomponent::run_phase(uvm_phase phase);
    mytype data; // Data must be mytype or derivative.
    data = mytype::type_id::create("data");
    $display("type of object is: %0s", data.get_type_name());
    ...
endtask

```

- U datom kodu, komponenta zahteva objekat iz fabrike koji je tipa mytype sa nazivom instance data.
- Kada fabrika kreira ovaj objekat, prvo će tragati za instancom, koja odgovara punom imenu instance objekta. Ako se ne pronađe, fabrika će tragati za tipom mytype. Ako se pronađe bilo instance ili tip uradiće se prpisivanje (override).
- Ako se ne pronađe ni taj tip, biće kreiran objekat tipa mytype.

Registracija pri Factory (pri fabrici)

- Fabrici se mora definisati kako da generiše objekte određenih tipova. U UVM-u postoji više načina da se to uradi.
- Preporučen metod je da se koristi `uvm_object_utils (T) ili `uvm_component_utils (T) makro nad izvedenoj uvm_object ili uvm_component deklaraciji klase.
Ovi makroi sadrže kod koji će registrovati dati tip pri fabrici.
- Ako je T parametrizovanog tipa, koristi se `uvm_object_param_utils ili `uvm_component_param_utils,
Na primer:
`uvm_object_utils (packet)
`uvm_component_param_utils (my_driver)
- Može se koristiti `uvm_object_registry (T, S) ili
`uvm_component_registry (T, S) makro za registraciju.
Ovi makroi se mogu pojaviti bilo gde u prostoru deklaracije klase T, gde će povezati string S sa objektom tipa T.

Factory override (prepisivanje komponenti, tipova i instanci)

- Globalna fabrika nam omogućava zamenu tipa unapred definisane komponente nekom drugom vrstom koja je specijalizovani za navedene namene, bez potrebe za ikakvim izmenama u kodu koji sadrži date komponente.
- Fabrika može da zameni komponentu unutar hijerarhije komponenti bez promene bilo koje druge komponente u hijerarhiji.
- Potrebno je znati kako koristiti fabriku, a ne i kako fabrika radi.
- Prepisivanje treba raditi nad hijerarhijskim nivoom kojiinstancija date tipove, odnosno instance.

Prepisivanje tipa

- Prepisivanje komponenti zamenjuje sve komponente specificiranog tipa novim specificiranim tipom.
- Prototip je.
`<orig_type> :: type_id :: set_type_override (<override_type> :: get_type(), bit replace = 1);`
- Metoda `set_type_override()` je statička metoda statičke članice `type_id` klase `orig_type`.
- Ovi elementi su definisani u makroima `uvm_component_utils` i `uvm_object_utils`.
- Prvi argument (`override_type :: get_type()`) je tip koji fabrika vraća na mesto originalnog tipa.
- Drugi argument, `replace`, određuje da li se vrši nastavak postojećeg prepisivanja (`replace = 1`). Ako je ovaj bit 0 i nema više prepisivanja datog tipa.

Prepisivanje instance

- Druga mogućnost prepisivanja zamenjuje ciljane komponente sa odgovarajuće putanje instance sa novim navedenim tipom.
- Prototip za uvm_component je
`<orig_type> :: type_id :: set_inst_override (<override_type> :: get_type (), string inst_path);`
- set_inst_override () je statička metoda članica tipa type_id od orig_type.
- Prvi argument je tip koji će biti vraćen na mesto originalnog tipa prilikom kreiranja komponente na relativnoj putanji kao što je navedeno u drugom argumentu.
- Drugi argument predstavlja cilj, odnosno instancu koja se prepisuje.

Primer:

Prepostavimo da je ubus_new_slave_monitor već definisan. Kada se sledeći kod izvrši, okruženje će kreirati novi tip, ubus_new_slave_monitor, za sve instance koji odgovaraju navedenoj putanji instance.

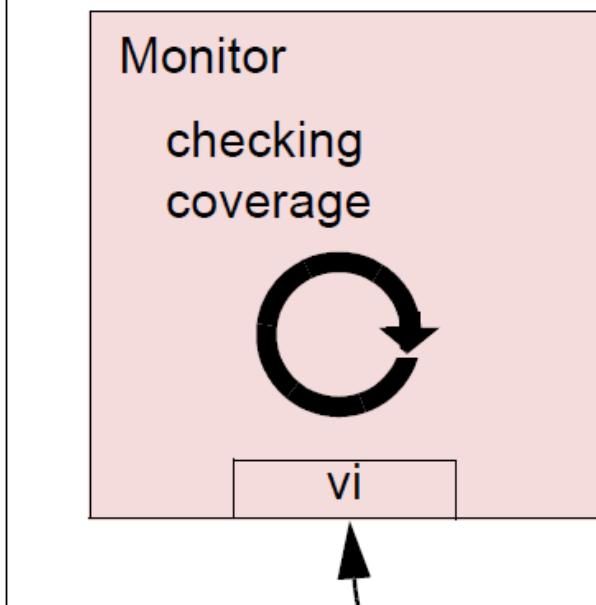
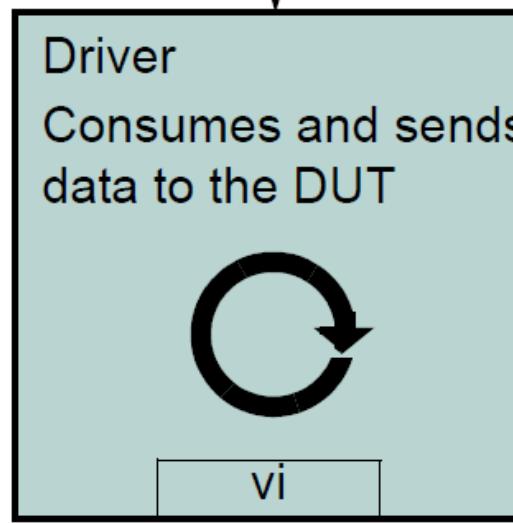
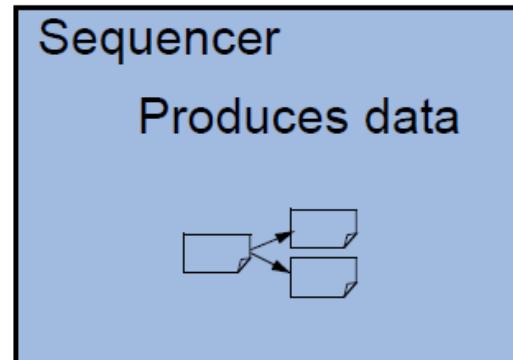
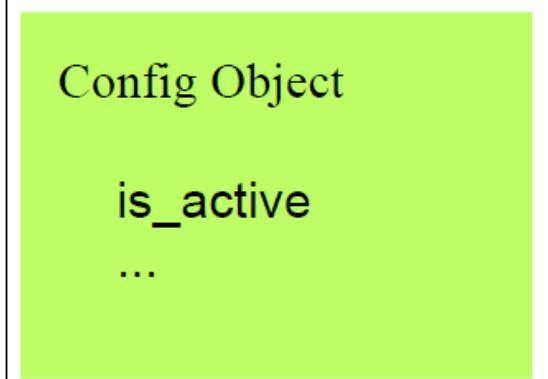
```
ubus_slave_monitor :: type_id :: set_inst_override (ibus_new_slave_monitor :: get_type(),  
"slaves[0].monitor");
```

U ovom slučaju, tip je prepisan samo za instancu slaves[0].monitor instance koja odgovara dатој putanji.

Kreiranje Agent-a

- Agent instancira i povezuje driver, monitor i sequencer koristeći TLM konekcije.
- Da bi se osigurala veća fleksibilnost, agent sadrži i informacije o konfiguraciji i druge parametre.
- U okruženju baziranom na magistralama agent obično modeluje vodeći (master), prateći (slave) uređaj ili arbitar na magistrali.

Agent



interface

Modovi rada Agent-a

- Agent ima dva osnovna načina rada:
- Aktivni način rada, gde agent emulira uređaj u sistemu i pokreće DUT signale. Ovaj način rada zahteva da agent instancira driver i sequencer. Monitor se takođe instancira za proveravanje i pokrivenost.
- Pasivni način rada, u kom agent ne instancira driver ili sekvenser i radi pasivno. Instancira se i konfiguriše samo monitor. Ovaj način rada se koristi samo ukoliko je potrebno vršiti pregled i pokrivanje.

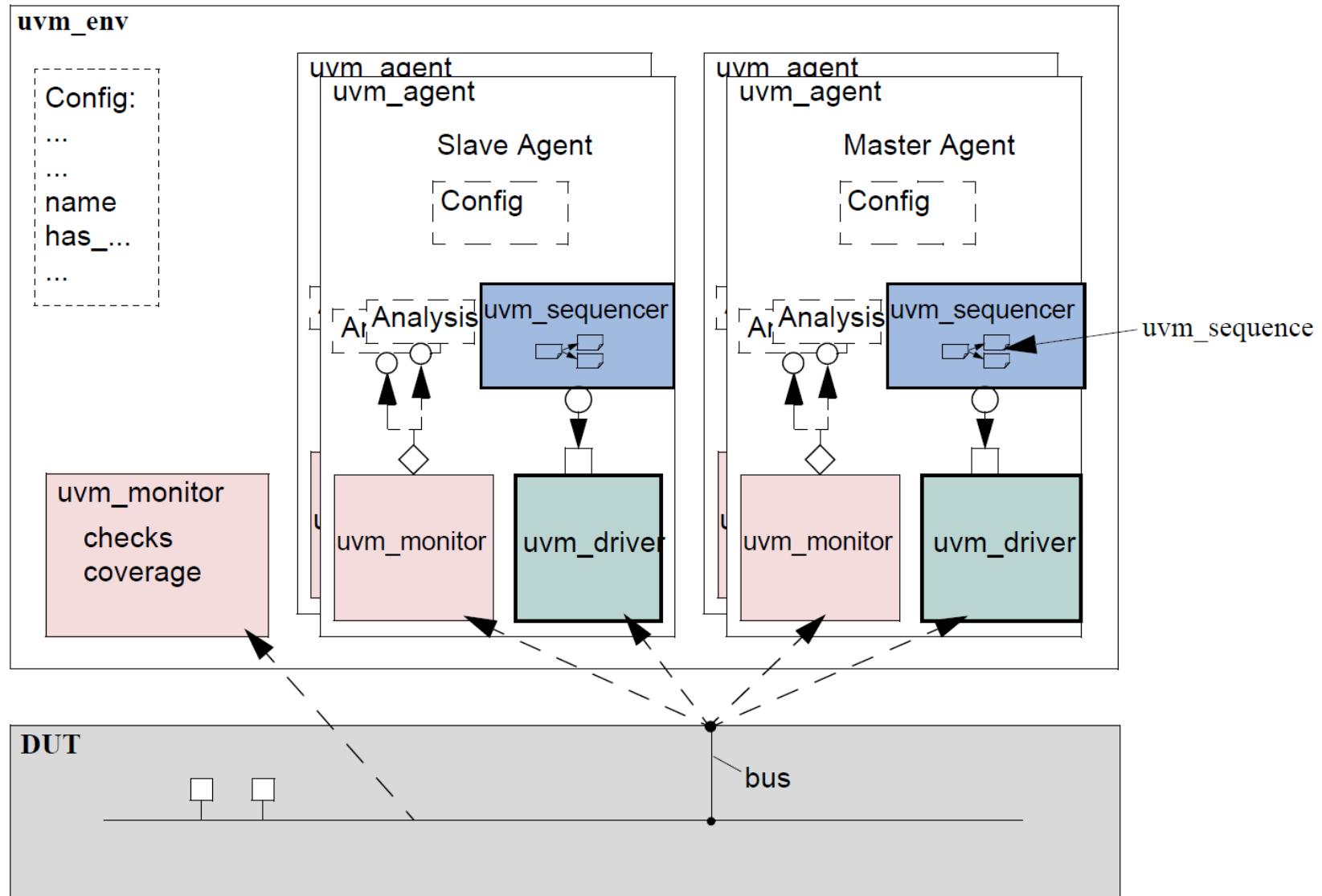
- Klasa `simple_agent` u narednom primeru instancira sekvencer, driver i monitor. Umesto korišćenja konstruktora, UVM build faza se koristi za konfigurisanje i konstrukciju podkomponenti agenta. Za razliku od konstruktora, ova virtualna funkcija može biti prepisana bez posebnih ograničenja.
- Alokacija `type_id :: create()` instancira podkomponente.

```
1 class simple_agent extends uvm_agent;
2   ... // Constructor and UVM automation macros
3   uvm_sequencer #(simple_item) sequencer;
4   simple_driver driver;
5   simple_monitor monitor;
6   // Use build_phase to create agents's subcomponents.
7   virtual function void build_phase(uvm_phase phase);
8     super.build_phase(phase)
9     monitor = simple_monitor::type_id::create("monitor",this);
10    if (is_active == UVM_ACTIVE) begin
11      // Build the sequencer and driver.
12      sequencer =
13        uvm_sequencer#(simple_item)::type_id::create("sequencer",this);
14      driver = simple_driver::type_id::create("driver",this);
15    end
16  endfunction : build_phase
17  virtual function void connect_phase(uvm_phase phase);
18    if(is_active == UVM_ACTIVE) begin
19      driver.seq_item_port.connect(sequencer.seq_item_export);
20    end
21  endfunction : connect_phase
22 endclass : simple_agent
```

Tumačenje primera

- Linija 8. omogućava automatsku konfiguraciju UVM polja deklarisanih putem `uvm_field_*` makroa tokom build faze.
- Linija 9. Monitor se instancira pomoću `create()` metode.
- Linije 10 – 15 ispituju uslov `is_active` i zavisno od njega kreiraju se sequencer i driver.
- I sekvencer i driver kreirani su istom tehnikom kao i monitor.
- Primer pokazuje flag `is_active` kao konfiguraciono svojstvo agenta. Moguće je definisati kontrolne flag-ove koji opredeljuju topologiju komponente. Na nivou okruženja, to mogu biti flag-ovi kao na primer `num_masters celobrojni`, `num_slaves celobrojni`, ili flag `has_bus_monitor`.
- NAPOMENA: metodu `create()` treba uvek pozivati iz metode `build_phase()` da bi se stvorila multi-hijerarhijska komponenta.
- Linije 18 – 20. Veza između driver-a i sequencer-a postavljena u `connect_phase` se postavlja ukoliko je agent aktivan.

Kreiranje Environment-a (okruženja)



Environment klasa

- Klasa environment predstavlja vrh hijerarhije komponenti za višekratnu upotrebu. Ona instancira i konfiguriše sve svoje podkomponente.
- Najčešće ponovna upotreba se koristi na nivou environmenta pri čemu korisnik instancira environment klasu i konfiguriše je sa agentima za specifične verifikacione aktivnosti.

Environment klasa - primer

```
class ahb_env extends uvm_env;
    int num_masters;
    ahb_master_agent masters[];
`uvm_component_utils_begin(ahb_env)
    `uvm_field_int(num_masters, UVM_ALL_ON)
`uvm_component_utils_end
virtual function void build_phase(phase);
    string inst_name;
    super.build_phase(phase);
    if(num_masters ==0)
        `uvm_fatal("NONUM", {"'num_masters' must be set");
masters = new[num_masters];
    for(int i = 0; i < num_masters; i++) begin
        $sformat(inst_name, "masters[%0d]", i);
        masters[i] = ahb_master_agent::type_id::create(inst_name,this);
    end
    // Build slaves and other components.
endfunction
function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction : new
endclass
```

Formiranje scenarija

- Korisnik Environment-a mora da kreira tipično veći broj testnih scenarija za ispitivanje datog DUT-a.
- Verifikacioni inženjer je prvenstveno fokusiran na DUT protokol, a ne na same blokove od kojih se DUT sastoji, stoga je potrebno:
 - Postaviti kontrole u data item klasu da bi se pojednostavila deklarativnu kontrolu testa.
 - Formirati biblioteku značajnih sekvenci koje se mogu ponovno koristiti.

Korisnik Environment-a kontroliše obrasce koje taj Environment generiše konfigurišući njegove sekvencere. Korisnik može da:

- Definiše nove sekvence koje generišu nove transakcije.
- Definiše nove sekvence koje sadrže postojeće sekvence.
- Prepisuje zadate kontrole nad data item-ima radi modifikacije driver-a i celokupnog ponašanja Environment-a.

Deklarisanje korisnički definisanih Sequences (sekvenci)

- Sekvence se sastoje od više data item-a, koji zajedno sačinjavaju potreban scenario ili data pattern.
- Komponente za proveru mogu uključiti biblioteku osnovnih sekvenci (umesto pojedinačnih data item-a).
- Ovaj pristup povećava višekratnu upotrebljivost stimulacionih obrazaca pobude i smanjuje dužinu testova.
- Takođe sekvenci je omogućeno pozivanje druge sekvence čime se stvaraju složeniji scenariji.

Koraci pri kreiranju korisnički definisane Sequence

- Sequence treba izvesti iz osnovne klase uvm_sequence uz navođenje request i response item tipa.
- Pomoću makroa `uvm_object_utils sekvenca se registruje pri fabrici.
- Ako sequence zahteva pristup izvedenoj funkcionalnosti njoj pridruženog sekvencera, to treba dodati u kodu ili iskoristiti makro 'uvm_declare_p_sequencer za deklaraciju i postavljanje željenog pokazivača na sekvencer.
- Treba implementirati body task sequence sa specifičnim scenariom koji je potrebno izvršiti. U body tasku, mogu se izvršavati data item-i i druge sequence.

Primer korisničke Sequence

```
class simple_seq_do extends uvm_sequence #(simple_item);
    rand int count;
    constraint c1 { count >0; count <50; }
    // Constructor
    function new(string name="simple_seq_do");
        super.new(name);
    endfunction
    //Register with the factory
    `uvm_object_utils(simple_seq_do)
    // The body() task is the actual logic of the sequence.
    virtual task body();
        repeat(count)
            // Example of using convenience macro to execute the item
            `uvm_do(req)
    endtask : body
endclass : simple_seq_do

class simple_sequencer extends uvm_sequencer #(simple_item) ;
    // same parameter as simple_seq_do
    `uvm_component_utils(simple_sequencer)
    function new (string name="simple_sequencer", uvm_component parent) ;
        super.new(name,parent) ;
    endfunction
endclass
```

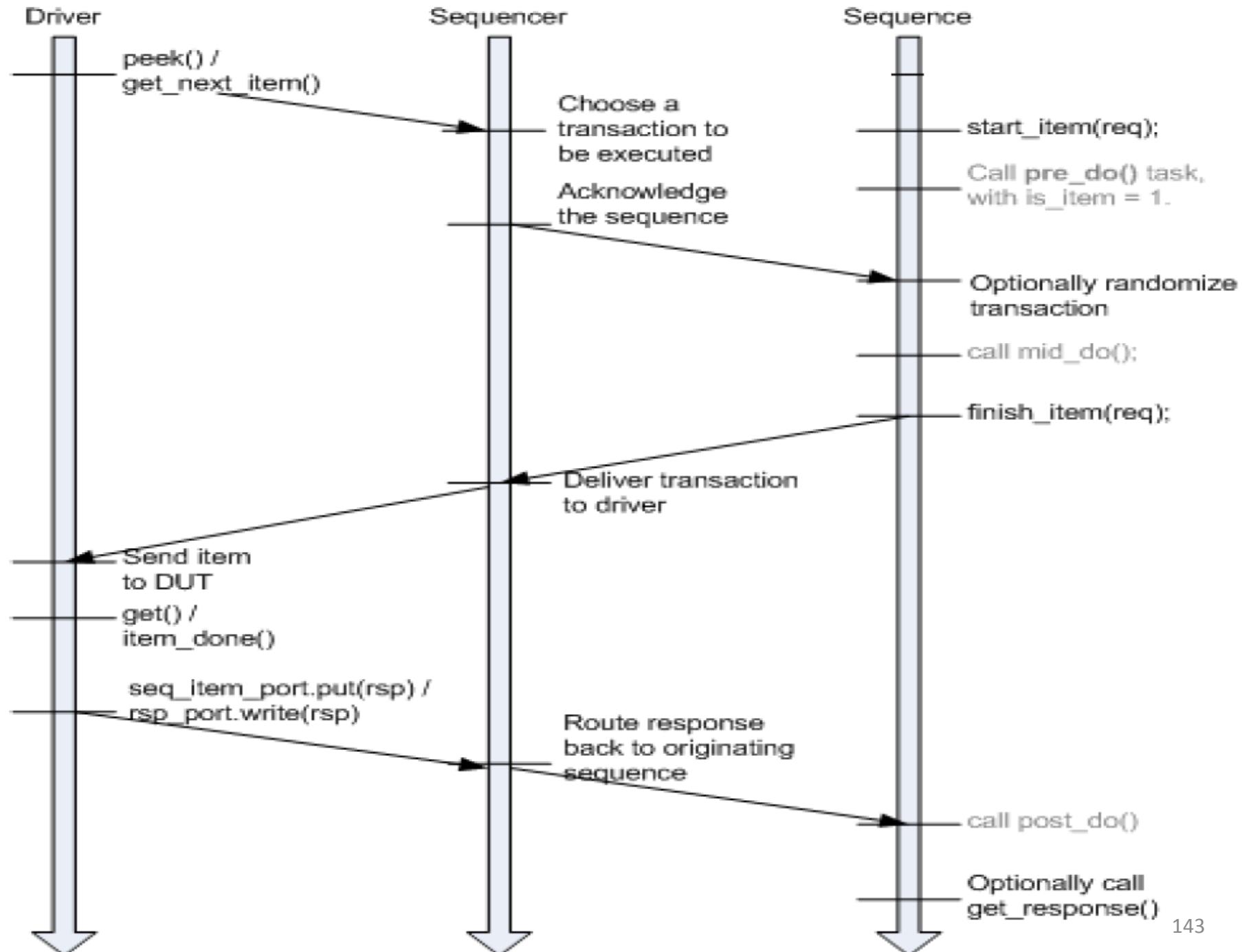
Šta nam Sequence omogućava?

- Postavljanje toka podataka koje se šalju DUT-u.
- Postavljanje toka akcija koje se izvršavaju na DUT interfejsu.

Moguće je naravno formirati statičku listu čistih podataka bez potrebe za konekcijom kad DUT interfejsu.

Osnovni tok Sequence Item-a

- Da bi se poslao sequence item, body() task iz sequence treba da kreira item korišćenjem fabrike, zatim poziva start_item () na item-u, a opcionalno može da ga randomizuje i na kraju poziva finish_item().
- Za slanje subsequence , task body () iz sequece prethodnika treba da kreira subsequence, opcionalno može da ga randomizuje i konačno poziva start() za datu subsequence. Ako subsequence ima pridruženi response (odgovor), izvorna sequence-a ga može pozvati sa get_response().
- Celokupan tok uključuje alokaciju objekata na osnovu fabričkog podešavanja datih registrovanih tipova, to ukratko zovemo stvaranje. Nakon stvaranja sledi inicijalizacija svojstva klase. Tehnika obrade zavisi da li je u pitanju sequence item ili sequence-a: pre_do(), mid_do(), i post_do() pozivi sequence i randomizacija objekta se pozivaju u različitim tačkama obrade za svaki tip objekta.
- Metode pre_body() i post_body() se ne pozivaju za subsequence.



UVM reporting

- **UVM** poseduje **reporting** makroe koji nam omogućavaju da kontrolišemo i filtriramo broj poruka koje dobijamo iz **testbench**-a.
- Vrlo često postavljamo razne izveštaje o greškama unutar **testbench**-a u toku razvoja projekta.
- Kako projekat raste, tipičan problem koji se može pojaviti jeste generisanje ogromnog broja informacija koje generišu naši testovi.
- Sa povećanjem složenosti projekta i sa brojem inženjera koji rade na projektu količina informacija koje nas mogu zapljušnuti u okviru svakog testa može nekontrolisano rasti.
- Klasičan način rada sa debug i ostalim informacijama koje se generišu tokom razvoja je da ih ručno iskomentarišemo ili obrišemo pošto izgube na aktuelnosti.

- UVM makroi nam omogućavaju dobru kontrolu reportinga
- Potrebno je uključiti ih u svoj fajl da bi se mogli koristiti

```
import uvm_pkg::*;  
`include "uvm_macros.svh"
```

- Postoje 4 tipa poruka koje možemo generisati ovim makroima

```
`uvm_info(<Message ID String>, <Message String>, <Verbosity>)  
`uvm_warning(<Message ID String>, <Message String>)  
`uvm_error(<Message ID String>, <Message String>)  
`uvm_fatal(<Message ID String>,<Message String>)
```

- Veoma je pogodan makro ispisa poruka ``uvm_info` jer bez generisanja grešaka koje imaju veću težinu omogućava preciznu kontrolu nad ispisom u toku debagovanja projekta.
- U okviru njega za svaku poruku definiše se nivo značaja, odnosno **Verbosity**. Direktan prevod izraza bio bi govorljiv, što je viši nivo govorljivosti, više poruka se ispisuje...

- Najprostiji mehanizam kojim doziramo količinu ispisanih poruka u simulaciji realizovan je globalnim argumentom koji se može opredeliti na nivou cele simulacije dodavanjem njega kao argumenta u skripti.
- Na primer `+UVM_VERBOSITY=UVM_HIGH` znači da će na nivou simulacije biti ispisivane sve poruke u kojima je verbosity nivo jednak ili niži od `UVM_HIGH`. Još kažemo da smo definisali verbosity plafon.

- Tehnički gledano UVM_VERBOSITY se definiše kao enumerisani tip

```
typedef enum
{
    UVM_NONE = 0,
    UVM_LOW = 100,
    UVM_MEDIUM = 200,
    UVM_HIGH = 300,
    UVM_FULL = 400,
    UVM_DEBUG = 500
} uvm_verbosity;
```

- Podrazumevani verbosity “plafon” nivo ukoliko nismo ništa definisali jeste ***UVM_MEDIUM*** što znači da će sve ***uvm_info*** poruke sa nivoom preko ovoga biti ignorisane.
- Omogućeno je i formiranje različitih Verbosity nivoa po blokovima sistema
- Na primer na nivou scoreboarda možemo povećati govorljivost:

scoreboard_h.set_report_verbosity_level(UVM_HIGH)

Materijali korišćeni u okviru kursa

- Materijal sa IMEC kursa “Essential verification with UVM”
<http://mtc.imec.be/courses/EssentialVerificationWithSystemVerilogAndUVM.pdf?c=1>
- IEEE Standard for SystemVerilog (IEEE Std 1800™-2017)
<https://ieeexplore.ieee.org/document/8299595>
- Universal Verification Methodology (UVM) 1.2 User's Guide – Accellera
https://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf
- The UVM Primer, Ray Salemi book
https://books.google.rs/books/about/The_Uvm_Primer.html?id=h7MLNgEACAAJ&redir_esc=y
- <https://github.com/rdsalemi/uvmprimer>