# Funkcionalna verifikacija hardvera

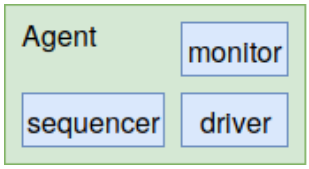
## Vežba 9
## Hijerarhija UVM verifikacionog okruženja

# Sadržaj

Deveta vežba je posvećena hijerarhiji UVM verifikacionog okruženja. Dat je pregled UVM agent klase, često korišćene klase za konfiguraciju kao i UVM environment klase. Objašnjen je *factory override* mehanizam i dat primer upotrebe.

# 1 Agent

Iako se drajver, sekvencer i monitor mogu koristiti nezavisno to iziskuje poznavanje imena, uloge i načina povezivanja svake komponente. Agent obuhvata ove tri komponente, predstavlja apstraktniji pristup datom interfejsu i olakšava ponovno korišćenje ovih komponenti. Može biti aktivni ili pasivni. Aktivni agenti sadrže komponente koje generišu stimulus na interfejsu kao i one koje ga nadgledaju, dok pasivni agenti samo nadgledaju interfejs odnosno pasivni agenti sadrže samo monitor zbog čega je veoma bitno da su drajver i monitor potpuno odvojeni iako im je deo funkcionalnosti sličan. Pasivni agenti se uglavnom koriste kao mali deo nekog većeg okruženja kada se stimulus generiše iz neke druge komponente.




Slika 1: Agent

## 1.1 Struktura

Kao što je već rečeno agent vrši povezivanje monitora, sekvencera i drajvera koristeći TLM konekcije. Kako bi se olakšala upotreba i fleksibilnost agenta, on sadrži i konfiguracione informacije smeštene u posebnom objektu.

Agent ima dva režima rada:

- Aktivni režim - agent generiše DUT signale; u ovom režimu agent instancira drajver i sekvencer, kao i monitor
- Pasivni režim - agent samo nadgleda signale; u ovom režimu agent ne instancira drajver i sekvencer, već samo monitor

U nastavku je dat primer agenta. Implementacija se vrši nasleđivanjem *uvm_agent* klase, sadrži kod za *factory* registraciju i konstruktor koji prate klasični UVM mehanizam.

```
class calc_agent extends uvm_agent;

  // components
  calc_driver drv;
  calc_sequencer seqr;
  calc_monitor mon;

  // configuration
  calc_config cfg;

  `uvm_component_utils_begin (calc_agent)
    `uvm_field_object(cfg, UVM_DEFAULT)
  `uvm_component_utils_end

  function new(string name = "calc_agent", uvm_component parent = null);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(calc_config)::get(this, "", "calc_config", cfg))
      `uvm_fatal("NO_CFG",{"Config object must be set for: ",get_full_name(),".cfg"})
```

```
        mon = calc_monitor::type_id::create("mon", this);
        if(cfg.is_active == UVM_ACTIVE) begin
            drv = calc_driver::type_id::create("drv", this);
            seqr = calc_sequencer::type_id::create("seqr", this);
        end

    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        if(cfg.is_active == UVM_ACTIVE) begin
            drv.seq_item_port.connect(seqr.seq_item_export);
        end
    endfunction : connect_phase

endclass : calc_agent
```

Kod 1: Kostur agenta

Agent u *build* fazi uvek kreira monitor, a kreira drajver i sekvencer jedino ukoliko je odabran aktivni režim rada. Sve komponente se kreiraju koristeći preporučeni metod, a ne direktno pozivanjem konstruktora *new*. Takođe će se drajver i sekvencer u *connect* fazi povezati jedino u aktivnom režimu rada.

## 1.2 Konfiguracija

Odabir režima rada, kao i sve opcije vezane za konfiguraciju agenta, obično se nalaze u posebnom objektu koji se prosleđuje koristeći *uvm_config_db*. Preuzimanje iz baze se vrši u *build* fazi:

```
if (!uvm_config_db#(calc_config)::get(this, "", "calc_config", cfg))
        `uvm_fatal("NO_CFG",{"Config object must be set for: ",get_full_name(),".cfg"})
```

Sama konfiguraciona klasa se implementira nasleđivanjem *uvm_object* klase, sadrži *factory* registraciju i konstruktor pisani po već objašnjenim pravilima i može sadržati proizvoljna polja u zavisnosti od potreba datog agenta. U nastavku je dat primer:

```
class calc_config extends uvm_object;

    uvm_active_passive_enum is_active = UVM_ACTIVE;

    `uvm_object_utils_begin (calc_config)
        `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_DEFAULT)
    `uvm_object_utils_end

    function new(string name = "calc_config");
        super.new(name);
    endfunction

endclass : calc_config
```

Kod 2: Primer konfiguracije agenta

Polje koje se skoro uvek sreće u konfiguraciji agenta je za odabir režima rada. Zbog ovoga postoji predefinisani nabrojivi tip *uvm_active_passive_enum* i konvencija je da se koristi za ovu funkcionalnost. Može imati vrednosti UVM_ACTIVE ili UVM_PASSIVE.

```
// Enum: uvm_active_passive_enum
//
// Convenience value to define whether a component, usually an agent,
// is in "active" mode or "passive" mode.
typedef enum bit { UVM_PASSIVE=0, UVM_ACTIVE=1 } uvm_active_passive_enum;
```

Pored ovog polja ova klasa može sadržati bilo koja polja potrebna za podešavanje konfiguracije agenta - npr. da li da se sakupljaju podaci o pokrivenosti ili ne, da li da se vrši čekiranje ili ne, *baud*

*rate*, odabir *parity* režima, CRC režima i sl.

Konfiguracioni objekat se kreira i podešava na višem nivou hijerarhije (npr. u testu), a zatim se preko baze prosleđuje agentu. Npr:

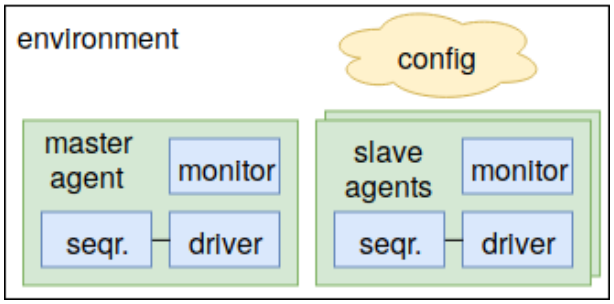
```
calc_config cfg;
// ...
function void build_phase(uvm_phase phase);
  // ...
  cfg = calc_config::type_id::create("cfg");
  uvm_config_db#(calc_config)::set(this, "*", "calc_config", cfg);
endfunction : build_phas
```

# 2 Hijerarhija okruženja

UVM verifikaciona okruženja se grade iz klasa koje nasleđuju *uvm_ component* klasu. Hijerarhija je određena nizom *has-a* klasnih veza gde jedna komponeta instancira druge, podkomponente. Na najvišem nivou se nalazi sam test u kome se vrši odabir konfiguracije okruženja, kreiranje svih podkomponenti, startovanje sekvenci, ... Sama arhitektura okruženja je uglavnom konstantna i modularna kako bi se omogućila laka ponovna upotreba verifikacionih komponenti (bilo horizontalna, bilo vertikalna). U UVM-u postoje dve komponente koje služe za grupisanje i olakšavanje ponovne upotrebe. Njihova uloga je jedino organizacija okruženja i definisanje jasne hijerarhije. Ove dve klase su agenti tj. *uvm_ agent* (opisani u prethodnom poglavlju) i okruženja tj. *uvm_ env*.

## 2.1 Blok nivo

Ovaj nivo zapravo predstavlja *environment* klasu koja grupiše sve komponente za ponovnu upotrebu. U njoj se vrši konfiguracija svih podkomponenti. Većina ponovne upotrebe se vrši na ovom nivou odnosno korisnik instancira datu *uvm_ env* klasu i konfiguriše agente prema svojim potrebama. *Env* klasa obično sadrži konfiguracionu klasu i željeni broj agenata, prikazano na slici 2.




Slika 2: UVM env

Umesto pojedinačnog instanciranja svih podkomponenti i podešavanja režima rada svake od njih, korišćenjem komponenti za grupisanje (agent i *env*) omogućava se podizanje nivoa hijerarhije i olakšava upotreba, ukoliko su sve komponente kreirane prateći UVM metodologiju. Upotreba od strane korisnika mora biti jasno definisana tj. opcije konfiguracije treba da su dobro dokumentovane.

Klasičan primer upotrebe bile bi komponente za protokole npr. AXI, UART, SPI, ... Prilikom verifikacije dizajna koji implementira neki protokol, korisnik bi instancirao datu klasu i promoću konfiguracije kontrolisao strukturu - da li je potreban master ili *slave* agent, broj agenata, da li su aktivni ili pasivni itd. Ovim se eleminiše potreba za detaljnim poznavanjem implementacije samih komponenti i omogućava brz i jednostavan način kreiranja kompleksnih okruženja. Ovakve komponente se obično nazivaju protokol UVC (UVM Verification Component) i najčešći su primer ponovne upotrebe koda u verifikacionim okruženjima.

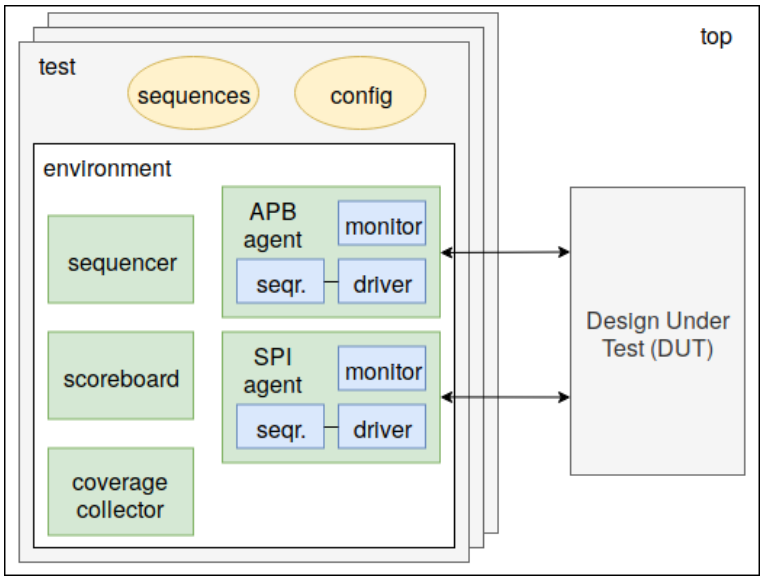
## 2.2 *Top-level* okruženje

Kao što je već rečeno na najvišem nivou hijerarhije se nalazi test klasa. Međutim, radi preglednosti i jasne organizacije koda, obično postoji i jedno okruženje koje obuhvata sve podkomponente i koje se instancira u samom testu. Tada se test fokusira samo na kreiranje sekvenci i podešavanje konfiguracija, a sva instanciranja i povezivanja podkomponenti se vrše u *env* klasi. Ovo *top-level* okruženje se često naziva i testbenč.

*Top-level* okruženje uglavnom sadrži određeni broj agenata (direktno ili preko okruženja opisanih u prethodnom poglavlju), ali sadrži i sve ostale komponente potrebne za proveru rada DUT-a npr.

*scoreboard*-e, globalne monitore, prikupljače pokrivenosti, ... Obično sadrži i konfiguracionu klasu preko koje se mogu podešavati određene osobine npr. konfiguracije samih agenata.

Na slici 3 je prikazan blok dijagram klasičnog UVM verifikacionog okruženja. *Env* klasa sadrži SPI i APB agent, *scoreboard* povezan sa oba monitora preko TLM interfejsa, i monitor za prikupljanje pokrivenosti povezan sa APB monitorom. *Env* takođe sadrži i konfiguracionu klasu preko koje je moguće pristupati konfiguracijama APB i SPI agenata. U testu je jedino potrebno instancirati *env* klasu, eventualno podesiti konfiguraciju i zatim pokretati sekvence na APB i SPI sekvencerima.




Slika 3: Primer okruženja

# 3 *Factory override*

UVM *factory* omogućava da se jedna klasa zameni drugom, izvedenom klasnom kada se konstruiše. Ovaj mehanizam može biti veoma korisan za promenu ponašanja testbenča bez potrebe da se modifikuje ili rekompajlira kod. Kako bi ovo bilo moguće potrebno je pratiti sve konvencije kodovanja opisane u petoj vežbi.

UVM *factory* se može posmatrati kao *lookup* tabela. Kada se komponente konstruišu koristeći *<type>::type_id::create("<name>", <parent>)* pristup, tada se *type_id* koristi kako bi se odabrao *wrapper* za klasu, izvršila konstrukcija i vratio pokazivač. Korišćenjem *override* mehanizma se originalni *type_id* zamenjuje drugim i time se vraća pokazivač na drugi objekat. Čitava tehnika je zasnovana na polimorfizmu tj. mogućnosti da se izvedeni tipovi referenciraju koristeći bazni pokazivač. Ovo znači da *override* može da se koristi jedino da se *parent* klasa zameni sa nekom njenom *child* klasom.

U UVM-u se *override* može koristiti i za komponente i za objekte, što je opisano u nastavku.

Komponente se mogu zameniti na dva načina - preko tipa ili preko instance. Zamena preko tipa znači da će se prilikom svakog kreiranja date komponente zapravo koristiti druga klasa, odnosno zamena se odnosi na sve instance date komponente u okruženju. Takođe je moguće zameniti samo jednu instancu, a ne sve. Ovo je često korisno prilikom korišćenja parametrizovanih klasa kako bi se jednostavno promenila vrednost parametra.

Postoje dve UVM funkcije koje služe za *override*:

```
<original_type>::type_id::set_type_override(<substitute_type>::get_type(), bit replace = 1);
<original_type>::type_id::set_inst_override(<substitute_type>::get_type(),string <path_string>);
```

Gde je *<original_type>* tip komponente koju želimo da zamenimo sa *<substitute_type>*, a *replace* je bit koji dozvoljava zamenu već postojeće zamene ukoliko ima vrednost 1, a koristi se već postojeća zamena ukoliko ima vrednost 0. *<path_string>* prilikom zamene instance predstavlja putanju do instance koju treba zameniti.

U nastavku je dat jednostavan primer:

```
class colour extends uvm_component;
  `uvm_component_utils(colour)
  // etc
endclass: colour

class red extends colour;
  `uvm_component_utils(red)
  //etc
endclass: red

// zamena svih instanci tipa "colour" sa "red":
colour::type_id::set_type_override(red::get_type(), 1);

// odnosno svaki poziv naredne linije vraca "red", a ne "colour"
pixel = colour::type_id::create("pixel", this);

// druga mogucnost - zamena samo jedne instance
// koja se nalazi na putanji "uvm_test_top.env.spot"
colour::type_id::set_inst_override(red::get_type(), "uvm_test_top.env.spot");
```

Za zamenu objekata se generalno koristi jedino zamena tipa, pošto za objekte ne postoji hijerarhija kao za komponente na kojoj se zasniva mehanizam zamene instanci. Kod je isti kao i za komponente.

# 4 Zadaci

**Zadatak** U pratećim materijalima za vežbu je dat kostur okruženja za "Calc1" dizajn. Korišćene su *uvm_ agent* i *uvm_ env* klase. Proučiti strukturu okruženja. Gde su i kako kreirane podkomponente? Šta može sadržati *calc_ config* klasa?

**Zadatak** U pratećim materijalima za vežbu je dat fajl "factory_override_.sv" u kome se nalazi primer korišćenja mehanizma zamene za objekte i komponente. Analizirati dati kod. Koji tipovi će se koristiti u *env* klasi? Modifikovati test tako da se jedino za a2 komponentu u *env*-u koristi *A_ ovr* tip, dok za sve ostale komponente (*a1* u ovom slučaju) koristi *A* klasa.

**Zadatak** Za razvijeno okruženje za "Calc1" implementirati novu klasu za *sequence_ item*, koja nasleđuje već razvijenu klasu. Nova transakcija treba da sadrži ograničenje da se nikad ne koristi operacija oduzimanja. Napisati test koji će u čitavom okruženju koristiti novu klasu.

# 5 Appendix

```
import uvm_pkg::*;              // import the UVM library
`include "uvm_macros.svh"     // Include the UVM macros

// components
// uvm_component -> A -> A_ovr
class A extends uvm_component;

   `uvm_component_utils(A)

   function new (string name = "A", uvm_component parent = null);
      super.new(name, parent);
   endfunction : new

endclass : A

class A_ovr extends A;

   `uvm_component_utils(A_ovr)

   function new (string name = "A_ovr", uvm_component parent = null);
      super.new(name, parent);
   endfunction : new

endclass : A_ovr

// objects
// uvm_object -> B -> B_ovr
class B extends uvm_object;

   `uvm_object_utils(B)

   function new (string name = "B");
      super.new(name);
   endfunction : new

endclass : B

class B_ovr extends B;

   `uvm_object_utils(B_ovr)

   function new (string name = "B_ovr");
      super.new(name);
   endfunction : new

endclass : B_ovr

// environment using A and B
class environment extends uvm_env;

   `uvm_component_utils(environment)

   A a1;
   A a2;
   B b1;

   function new(string name="environment", uvm_component parent = null);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      a1 = A::type_id::create("a1", this);
      a2 = A::type_id::create("a2", this);
      b1 = B::type_id::create("b1");
```

```
    endfunction : build_phase

    task run_phase(uvm_phase phase);
        `uvm_info(get_type_name(), "Env run_phase", UVM_LOW);
        `uvm_info(get_type_name(), $sformatf("Using component type for a1: %s", a1.get_type_name()),
       UVM_LOW)
        `uvm_info(get_type_name(), $sformatf("Using component type for a2: %s", a2.get_type_name()),
       UVM_LOW)
        `uvm_info(get_type_name(), $sformatf("Using object type for b1: %s", b1.get_type_name()), UVM_LOW
       )
    endtask : run_phase

endclass : environment

// test
class test extends uvm_test;

    `uvm_component_utils(test)

    environment env;

    function new(string name = "test", uvm_component parent = null);
        super.new(name, parent);
    endfunction : new

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env = environment::type_id::create("env", this);

        `uvm_info(get_type_name(), "Example set_type_override", UVM_LOW);
        A::type_id::set_type_override(A_ovr::get_type(), 1);
        B::type_id::set_type_override(B_ovr::get_type(), 1);
    endfunction : build_phase

endclass : test

// top module
module top();
     initial begin
        run_test("test");
    end
endmodule : top
```

Kod 3: v9_factory_override

```
`ifndef CALC_IF_SV
 `define CALC_IF_SV

interface calc_if (input clk, logic [6 : 0] rst);

    parameter DATA_WIDTH = 32;
    parameter RESP_WIDTH = 2;
    parameter CMD_WIDTH = 4;

    logic [DATA_WIDTH - 1 : 0] out_data1;
    logic [DATA_WIDTH - 1 : 0] out_data2;
    logic [DATA_WIDTH - 1 : 0] out_data3;
    logic [DATA_WIDTH - 1 : 0] out_data4;
    logic [RESP_WIDTH - 1 : 0] out_resp1;
    logic [RESP_WIDTH - 1 : 0] out_resp2;
    logic [RESP_WIDTH - 1 : 0] out_resp3;
    logic [RESP_WIDTH - 1 : 0] out_resp4;
    logic [CMD_WIDTH - 1 : 0]  req1_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req1_data_in;
    logic [CMD_WIDTH - 1 : 0]  req2_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req2_data_in;
    logic [CMD_WIDTH - 1 : 0]  req3_cmd_in;
```

```
    logic [DATA_WIDTH - 1 : 0] req3_data_in;
    logic [CMD_WIDTH - 1 : 0] req4_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req4_data_in;

endinterface : calc_if

`endif
```

Kod 4: calc_if

```
class calc_agent extends uvm_agent;

    // components
    calc_driver drv;
    calc_sequencer seqr;
    calc_monitor mon;

    // configuration
    calc_config cfg;

    `uvm_component_utils_begin (calc_agent)
        `uvm_field_object(cfg, UVM_DEFAULT)
    `uvm_component_utils_end

    function new(string name = "calc_agent", uvm_component parent = null);
        super.new(name,parent);
    endfunction

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(calc_config)::get(this, "", "calc_config", cfg))
            `uvm_fatal("NO_CFG",{"Config object must be set for: ",get_full_name(),".cfg"})

        mon = calc_monitor::type_id::create("mon", this);
        if(cfg.is_active == UVM_ACTIVE) begin
            drv = calc_driver::type_id::create("drv", this);
            seqr = calc_sequencer::type_id::create("seqr", this);
        end

    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        if(cfg.is_active == UVM_ACTIVE) begin
            drv.seq_item_port.connect(seqr.seq_item_export);
        end
    endfunction : connect_phase

endclass : calc_agent
```

Kod 5: v9_calc_agent

```
class calc_config extends uvm_object;

    uvm_active_passive_enum is_active = UVM_ACTIVE;

    `uvm_object_utils_begin (calc_config)
        `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_DEFAULT)
    `uvm_object_utils_end

    function new(string name = "calc_config");
        super.new(name);
    endfunction

endclass : calc_config
```

Kod 6: v9_calc_config

```
`ifndef CALC_DRIVER_SV
 `define CALC_DRIVER_SV

class calc_driver extends uvm_driver#(calc_seq_item);

   `uvm_component_utils(calc_driver)

   // The virtual interface used to drive and view HDL signals.
   virtual interface calc_if vif;

   function new(string name = "calc_driver", uvm_component parent = null);
      super.new(name,parent);
   endfunction

   function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      if (!uvm_config_db#(virtual calc_if)::get(this, "*", "calc_if", vif))
        `uvm_fatal("NO_IF",{"virtual interface must be set for: ",get_full_name(),".vif"})
   endfunction : connect_phase

   task main_phase(uvm_phase phase);
      forever begin
         seq_item_port.get_next_item(req);
         `uvm_info(get_type_name(),
                   $sformatf("Driver sending...\n%s", req.sprint()),
                   UVM_HIGH)
         // do actual driving here
         /* TODO */
         seq_item_port.item_done();
      end
   endtask : main_phase

endclass : calc_driver

`endif
```

Kod 7: v9_calc_driver

```
`ifndef CALC_ENV_SV
 `define CALC_ENV_SV

class calc_env extends uvm_env;

   calc_agent agent;

   `uvm_component_utils (calc_env)

   function new(string name = "calc_env", uvm_component parent = null);
      super.new(name,parent);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      agent = calc_agent::type_id::create("agent", this);
   endfunction : build_phase

endclass : calc_env

`endif
```

Kod 8: v9_calc_env

```
`ifndef CALC_SEQ_ITEM_SV
 `define CALC_SEQ_ITEM_SV

class calc_seq_item extends uvm_sequence_item;
```

```
    /* TODO add fields and methods here */

    `uvm_object_utils_begin(calc_seq_item)
    `uvm_object_utils_end

    function new(string name = "calc_seq_item");
        super.new(name);
    endfunction

endclass : calc_seq_item

`endif
```

Kod 9: v9_calc_seq_item

```
class calc_monitor extends uvm_monitor;

    // control fileds
    bit checks_enable = 1;
    bit coverage_enable = 1;

    uvm_analysis_port #(calc_seq_item) item_collected_port;

    `uvm_component_utils_begin(calc_monitor)
        `uvm_field_int(checks_enable, UVM_DEFAULT)
        `uvm_field_int(coverage_enable, UVM_DEFAULT)
    `uvm_component_utils_end

    // The virtual interface used to drive and view HDL signals.
    virtual interface calc_if vif;

    // current transaction
    calc_seq_item curr_it;

    // coverage can go here
    // ...

    function new(string name = "calc_monitor", uvm_component parent = null);
        super.new(name,parent);
        item_collected_port = new("item_collected_port", this);
    endfunction

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        if (!uvm_config_db#(virtual calc_if)::get(this, "*", "calc_if", vif))
          `uvm_fatal("NO_IF",{"virtual interface must be set:",get_full_name(),".vif"})
    endfunction : connect_phase

    task main_phase(uvm_phase phase);
        // forever begin
        // curr_it = calc_seq_item::type_id::create("curr_it", this);
        // ...
        // collect transactions
        // ...
        // item_collected_port.write(curr_it);
        // end
    endtask : main_phase

endclass : calc_monitor
```

Kod 10: v9_calc_monitor

```
`ifndef CALC_SEQUENCER_SV
 `define CALC_SEQUENCER_SV

class calc_sequencer extends uvm_sequencer#(calc_seq_item);
```

```
    `uvm_component_utils(calc_sequencer)

    function new(string name = "calc_sequencer", uvm_component parent = null);
        super.new(name,parent);
    endfunction

endclass : calc_sequencer

`endif
```

Kod 11: v9_calc_sequencer

```
`ifndef TEST_BASE_SV
 `define TEST_BASE_SV

class test_base extends uvm_test;

    calc_env env;
    calc_config cfg;

    `uvm_component_utils(test_base)

    function new(string name = "test_base", uvm_component parent = null);
        super.new(name,parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env = calc_env::type_id::create("env", this);
        cfg = calc_config::type_id::create("cfg");
        uvm_config_db#(calc_config)::set(this, "*", "calc_config", cfg);
    endfunction : build_phase

    function void end_of_elaboration_phase(uvm_phase phase);
        super.end_of_elaboration_phase(phase);
        uvm_top.print_topology();
    endfunction : end_of_elaboration_phase

endclass : test_base

`endif
```

Kod 12: v9_test_base

```
`ifndef TEST_SIMPLE_SV
 `define TEST_SIMPLE_SV

class test_simple extends test_base;

    `uvm_component_utils(test_simple)

    calc_simple_seq simple_seq;

    function new(string name = "test_simple", uvm_component parent = null);
        super.new(name,parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        simple_seq = calc_simple_seq::type_id::create("simple_seq");
    endfunction : build_phase

    task main_phase(uvm_phase phase);
        phase.raise_objection(this);
        simple_seq.start(env.agent.seqr);
        phase.drop_objection(this);
    endtask : main_phase
```

```
endclass

`endif
```

Kod 13: v9_test_simple

```
`ifndef TEST_SIMPLE_2_SV
 `define TEST_SIMPLE_2_SV

class test_simple_2 extends test_base;

   `uvm_component_utils(test_simple_2)

   function new(string name = "test_simple_2", uvm_component parent = null);
      super.new(name,parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      uvm_config_db#(uvm_object_wrapper)::set(this, "seqr.main_phase","default_sequence",calc_simple_seq
     ::type_id::get());
   endfunction : build_phase

endclass

`endif
```

Kod 14: v9_test_simple_2

```
`ifndef TEST_LIB_SV
 `define TEST_LIB_SV

 `include "tests/v9_test_base.sv"
 `include "tests/v9_test_simple.sv"
 `include "tests/v9_test_simple_2.sv"

`endif
```

Kod 15: v9_test_lib

```
`ifndef CALC_BASE_SEQ_SV
 `define CALC_BASE_SEQ_SV

class calc_base_seq extends uvm_sequence#(calc_seq_item);

   `uvm_object_utils(calc_base_seq)
   `uvm_declare_p_sequencer(calc_sequencer)

   function new(string name = "calc_base_seq");
      super.new(name);
   endfunction

   // objections are raised in pre_body
   virtual task pre_body();
      uvm_phase phase = get_starting_phase();
      if (phase != null)
        phase.raise_objection(this, {"Running sequence '", get_full_name(), "'"});
   endtask : pre_body

   // objections are dropped in post_body
   virtual task post_body();
      uvm_phase phase = get_starting_phase();
      if (phase != null)
        phase.drop_objection(this, {"Completed sequence '", get_full_name(), "'"});
```

```
    endtask : post_body

endclass : calc_base_seq

`endif
```

Kod 16: v9_calc_base_seq

```
`ifndef CALC_SIMPLE_SEQ_SV
 `define CALC_SIMPLE_SEQ_SV

class calc_simple_seq extends calc_base_seq;

    `uvm_object_utils (calc_simple_seq)

    function new(string name = "calc_simple_seq");
        super.new(name);
    endfunction

    virtual task body();
        // simple example - just send one item
        `uvm_do(req);
    endtask : body

endclass : calc_simple_seq

`endif
```

Kod 17: v9_calc_simple_seq

```
`ifndef CALC_SEQ_LIB_SV
 `define CALC_SEQ_LIB_SV

 `include "sequences/v9_calc_base_seq.sv"
 `include "sequences/v9_calc_simple_seq.sv"

`endif
```

Kod 18: v9_calc_seq_lib

```
`ifndef CALC_VERIF_PKG_SV
 `define CALC_VERIF_PKG_SV

package calc_verif_pkg;

    import uvm_pkg::*;      // import the UVM library
 `include "uvm_macros.svh" // Include the UVM macros

 `include "v9_calc_config.sv"

 `include "v9_calc_seq_item.sv"
 `include "v9_calc_driver.sv"
 `include "v9_calc_sequencer.sv"
 `include "v9_calc_monitor.sv"
 `include "v9_calc_agent.sv"

 `include "v9_calc_env.sv"

 `include "sequences/v9_calc_seq_lib.sv"
 `include "tests/v9_test_lib.sv"

endpackage : calc_verif_pkg

 `include "calc_if.sv"
```

```
`endif
```

Kod 19: v9_calc_verif_pkg

```
module calc_verif_top;

   import uvm_pkg::*;     // import the UVM library
`include "uvm_macros.svh" // Include the UVM macros

   import calc_verif_pkg::*;

   logic clk;
   logic [6 : 0] rst;

   // interface
   calc_if calc_vif(clk, rst);

   // DUT
   calc_top DUT(
                .c_clk         ( clk ),
                .reset         ( rst ),
                .out_data1     ( calc_vif.out_data1 ),
                .out_data2     ( calc_vif.out_data2 ),
                .out_data3     ( calc_vif.out_data3 ),
                .out_data4     ( calc_vif.out_data4 ),
                .out_resp1     ( calc_vif.out_resp1 ),
                .out_resp2     ( calc_vif.out_resp2 ),
                .out_resp3     ( calc_vif.out_resp3 ),
                .out_resp4     ( calc_vif.out_resp4 ),
                .req1_cmd_in   ( calc_vif.req1_cmd_in ),
                .req1_data_in  ( calc_vif.req1_data_in ),
                .req2_cmd_in   ( calc_vif.req2_cmd_in ),
                .req2_data_in  ( calc_vif.req2_data_in ),
                .req3_cmd_in   ( calc_vif.req3_cmd_in ),
                .req3_data_in  ( calc_vif.req3_data_in ),
                .req4_cmd_in   ( calc_vif.req4_cmd_in ),
                .req4_data_in  ( calc_vif.req4_data_in )
                );

    initial  begin
       uvm_config_db#(virtual calc_if)::set(null, "*", "calc_if", calc_vif);
       run_test();
   end

   // clock and reset  init .
    initial  begin
       clk <= 0;
       rst <= 1;
       #50 rst <= 0;
   end

   // clock generation
   always #50 clk = ~clk;

endmodule : calc_verif_top
```

Kod 20: v9_calc_verif_top

```
# Create the library
if [ file exists work] {
    vdel -all
}
vlib work

# compile DUT
vlog +incdir+../dut \
    ../dut/calc_top.v
```

```
# compile testbench
vlog +acc -sv \
    +incdir+$env(UVM_HOME) \
    ./*calc_verif_pkg.sv \
    ./*calc_verif_top.sv

# run simulation
vsim calc_verif_top +UVM_TESTNAME=test_simple +UVM_VERBOSITY=UVM_HIGH -sv_seed random
     -do "run -all"
```

Kod 21: calc_run