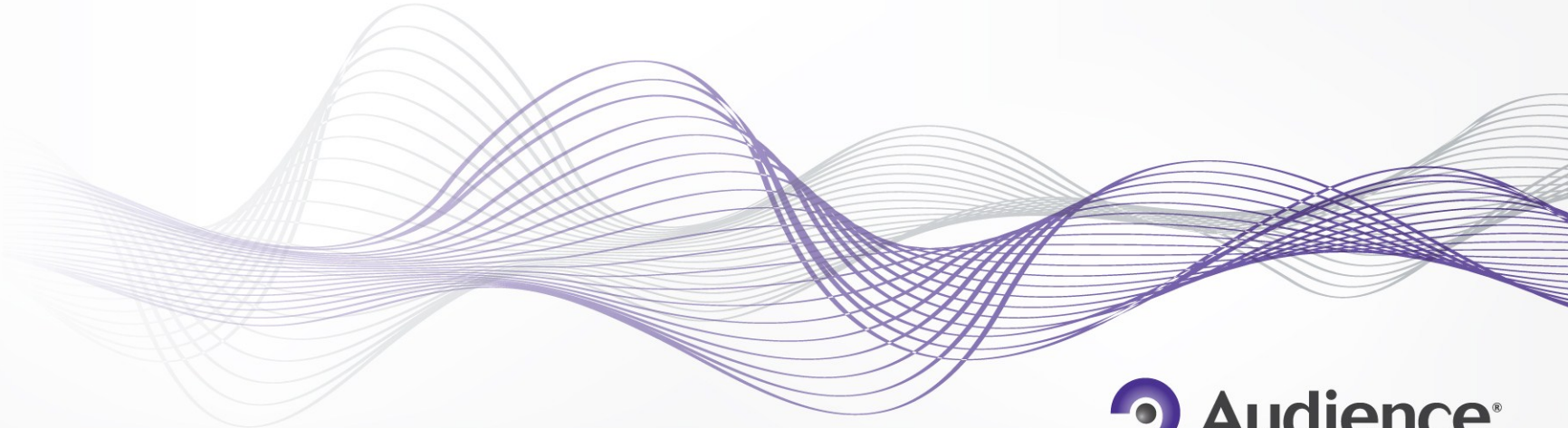
Do More With UVM

Rambabu Maddali

Audience Communications Systems India Pvt Ltd

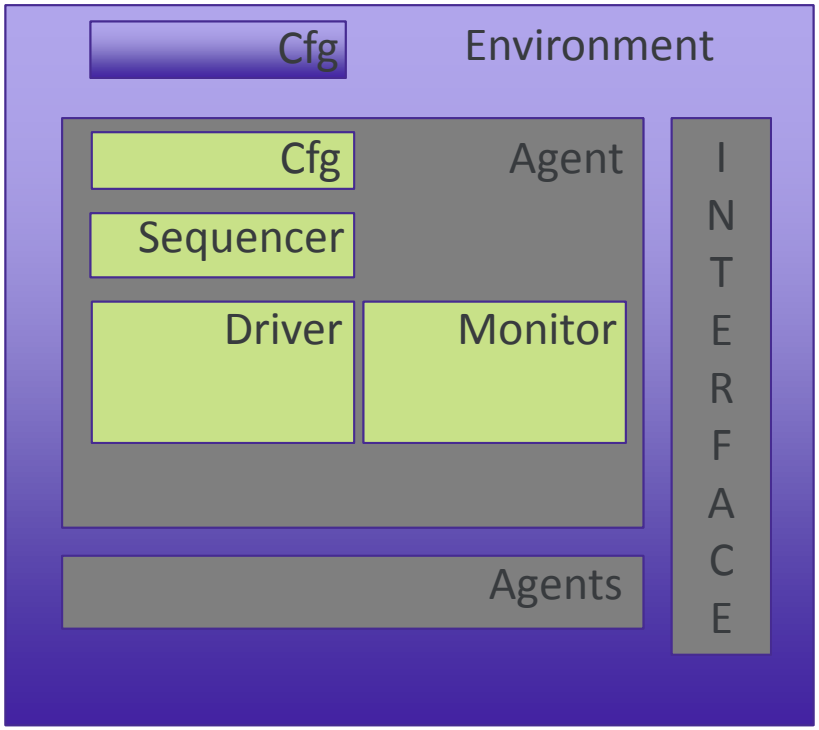


Audience®

Why UVM?

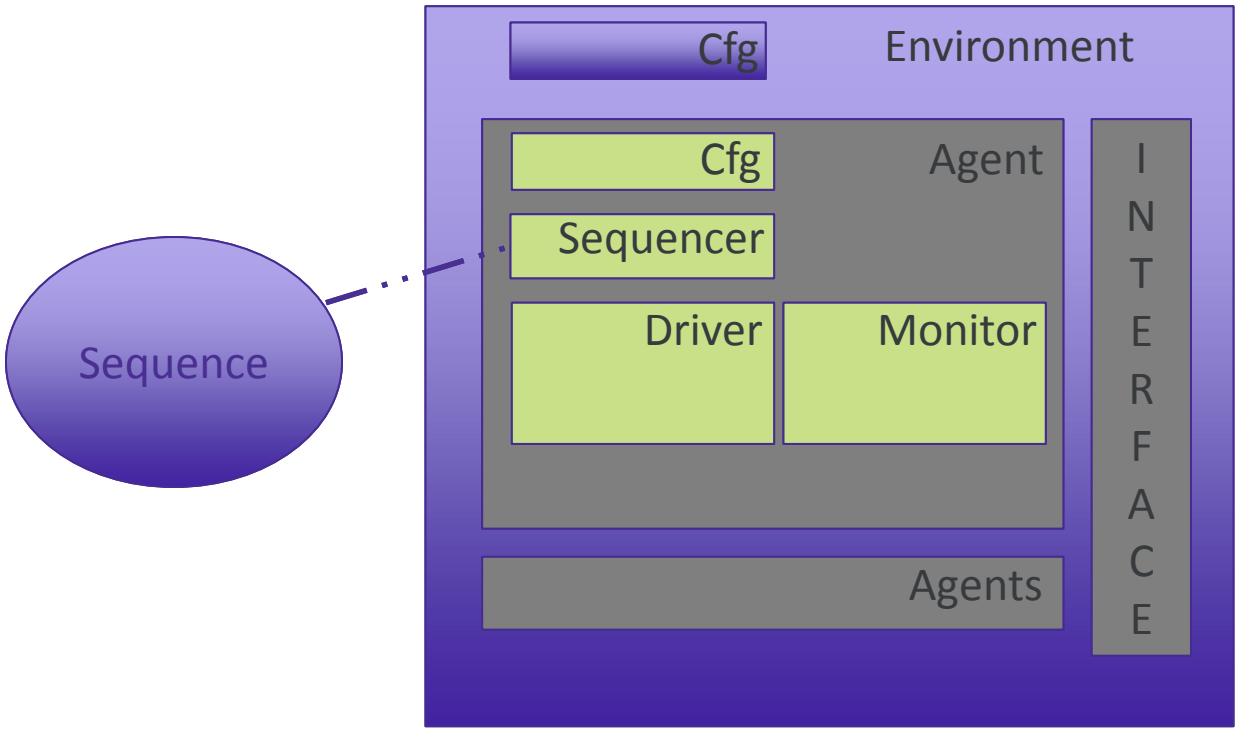
- **Scalability and Reusability**
- **Has the best features of OVM and VMM**
- **UVM config db and factory are very powerful features**
- **Easy porting from module level to top level**
- **Easy to understand environments**
- **Better control of debug information, testcase writing**
- **Structured communication mechanism between different modules in TB**
  - Abstracted communication
- **Creating error tests is made easy**
  - No need to touch working tests
- **Rich portfolio of verification components (UVCs)**

UVC structure



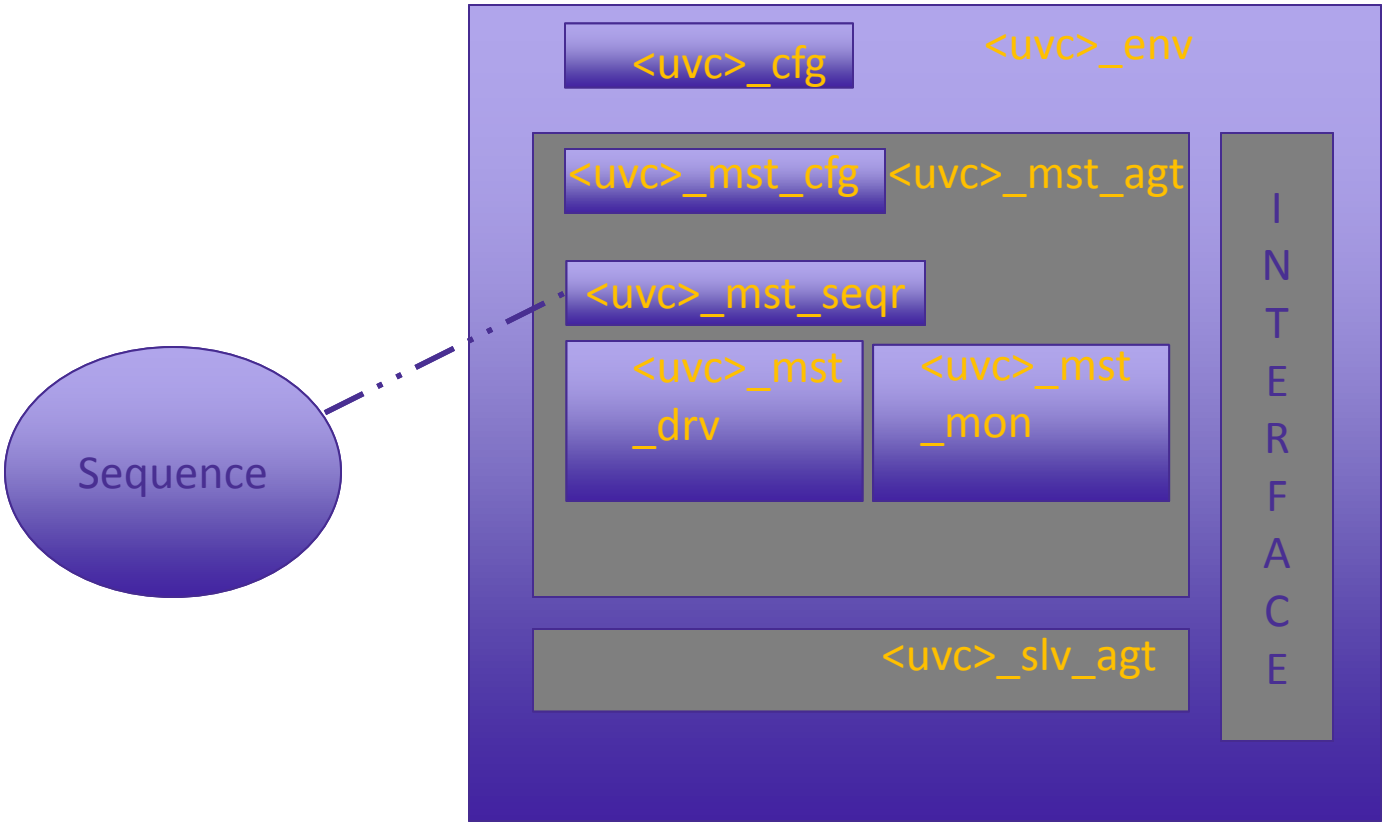
Generic UVM structure only defines the structure, no naming conventions

UVC structure



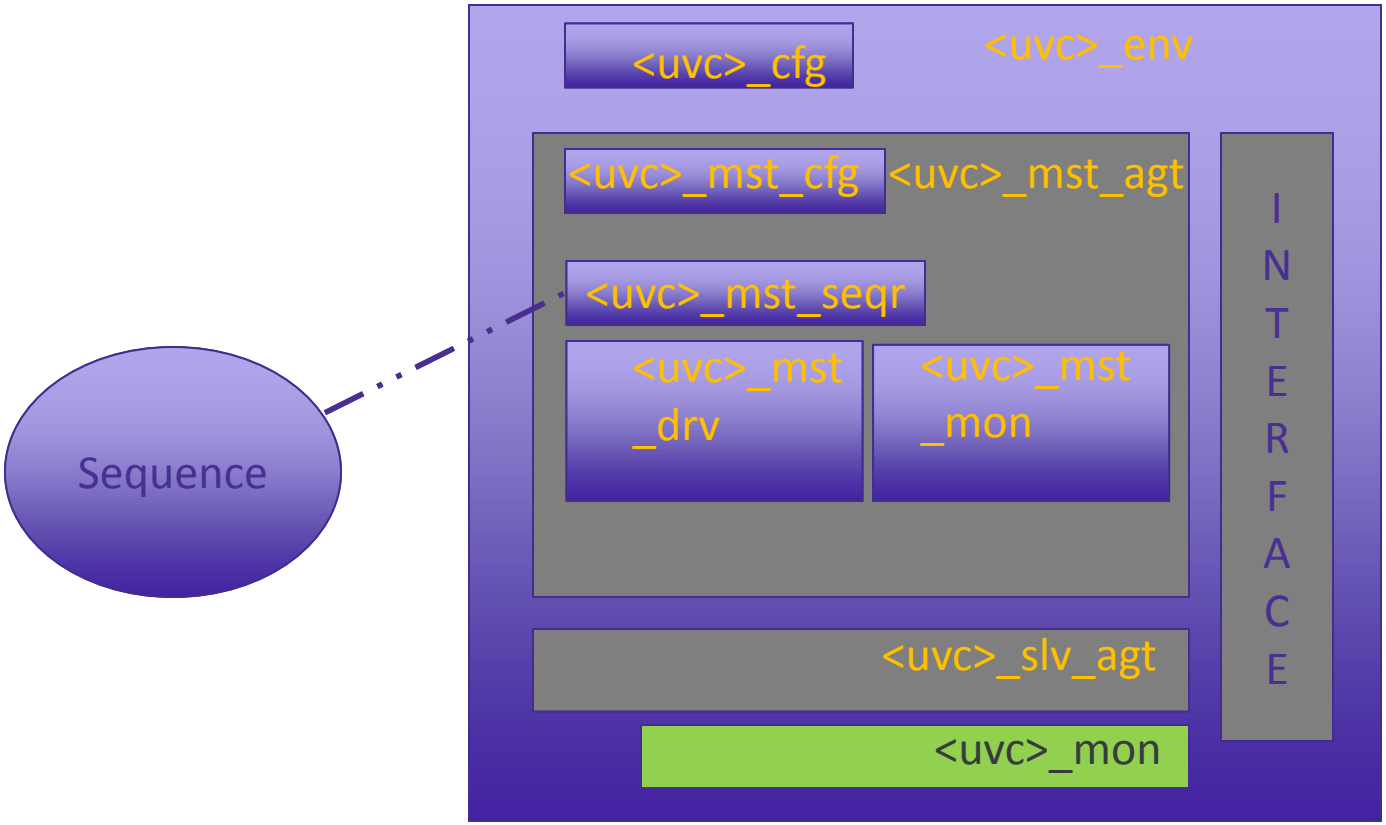
Generic UVM structure only defines the structure, no naming conventions

Audience UVC structure



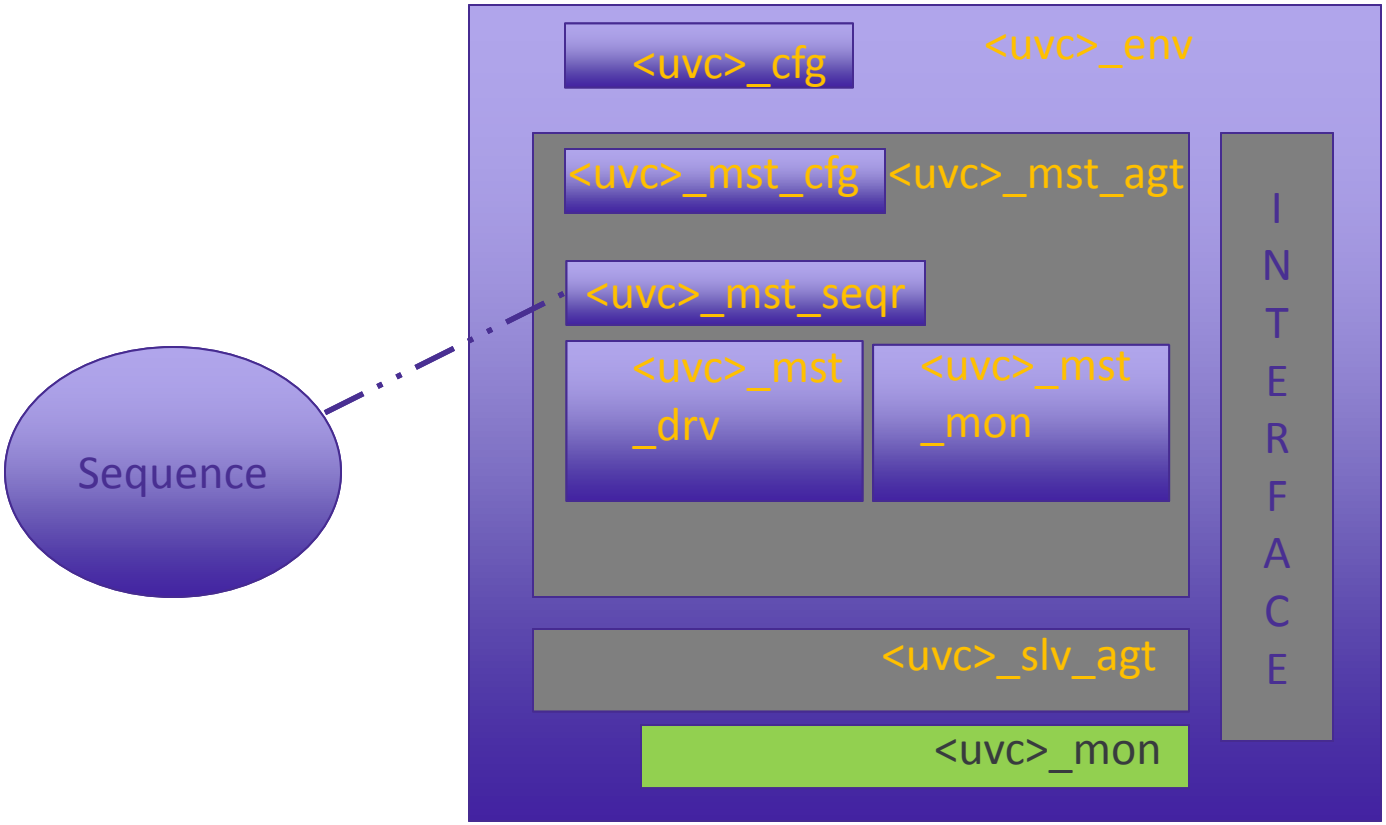
To make it same look and feel, we define the names of components of UVCs, instance names

Audience UVC structure



Adding Bus Monitor at UVC environment level and making the agent specific monitors having only master/slave sepcifc checkers helps in reducing the number of active monitors
Otherwise, UVC configuration at top level would need changes to module level environments

Audience UVC structure



To make it same look and feel, we define the names of components of UVCs, instance names

We even fix the filenames of the UVC sv directory & UVC template is script generated

Clocks and Resets for UVC

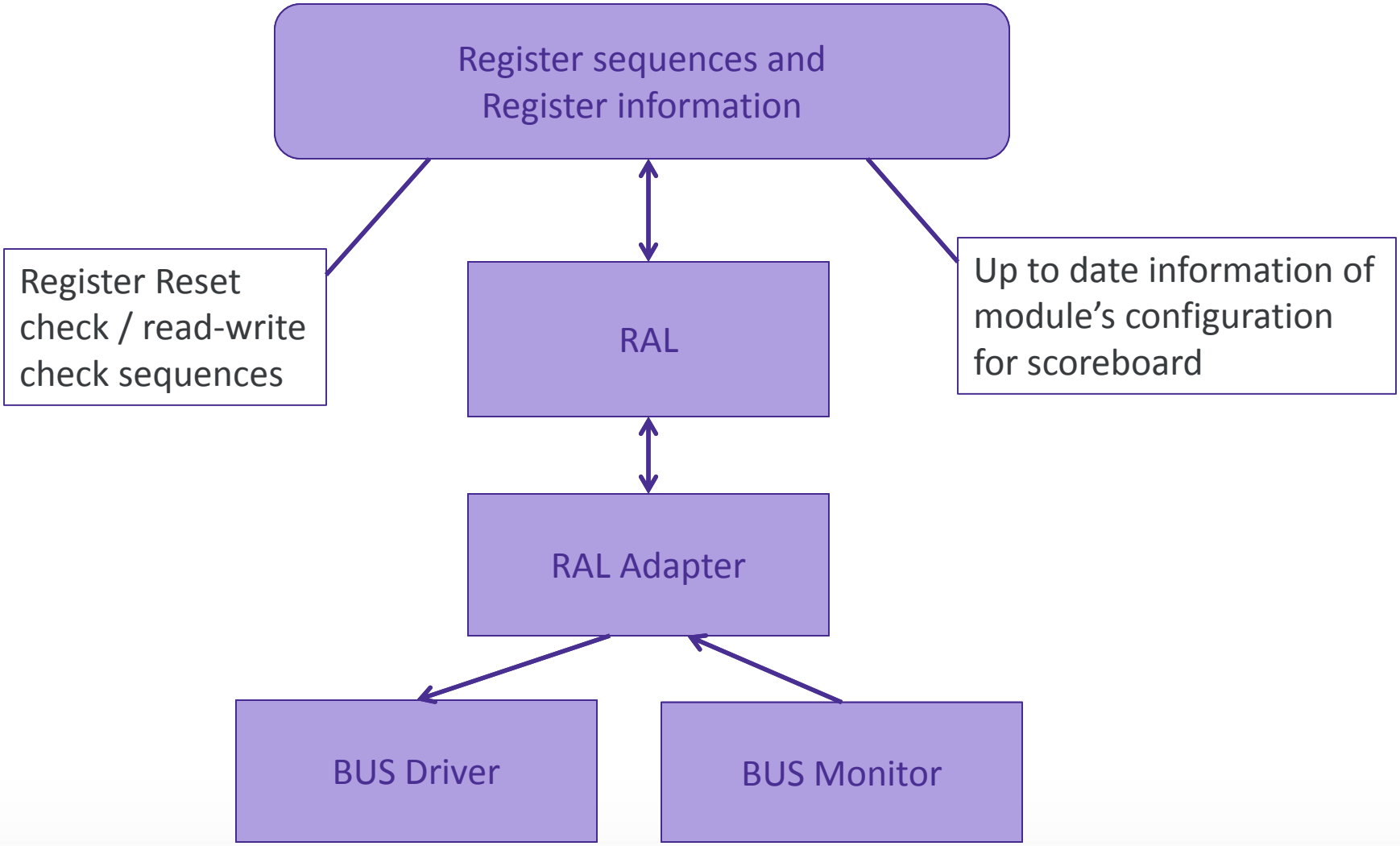
- **Adding reset related code in the initial UVC development makes the UVC usable in environments where Reset is applied more than once**
- **Always define clocking block inside interface and use those clocking block events, instead of posedge of clock / negedge of clock**
  - Later if we decide to move the sampling of signals from posedge to negedge, adaptation would be easy

```
task wait_for_req();
   while(vif.req == 0)
      @(posedge vif.clk);
   $display("req asserted");
   ....
endtask
```

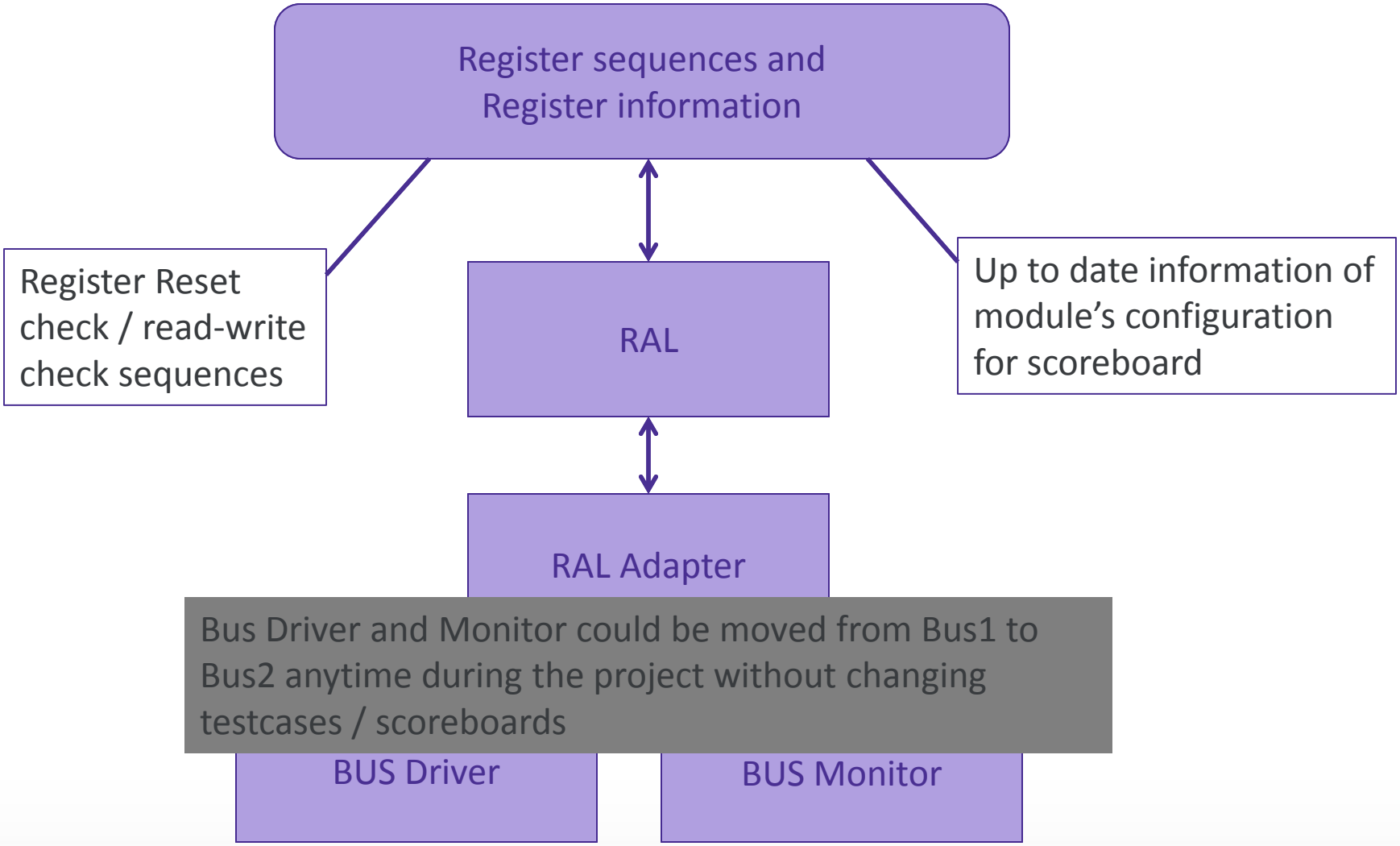
→

```
task wait_for_req();
   while(vif.req == 0)
      @vif.cb;
   `uvm_info(...);
   ....
endtask
```

Integration of RAL



Integration of RAL



RAL - TB & DUT Synchronization

- **Synchronizing DUT configuration with TB to start a sequence is similar to an approach based on adhoc delays or based on internal signals**
  - when RAL is integrated in TB, it is different

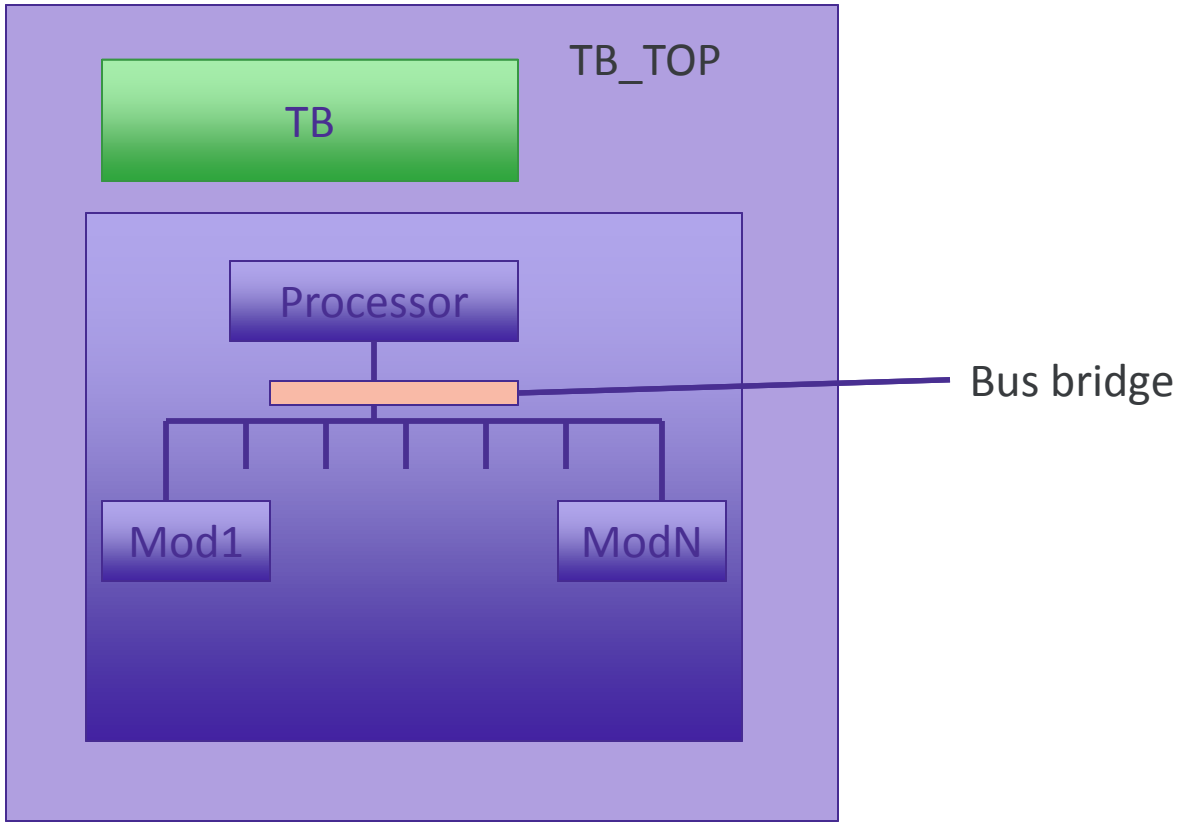
Without RAL

```
task body();
  #10000;
  `uvm_do(seq1);
endtask
```

With RAL integrated

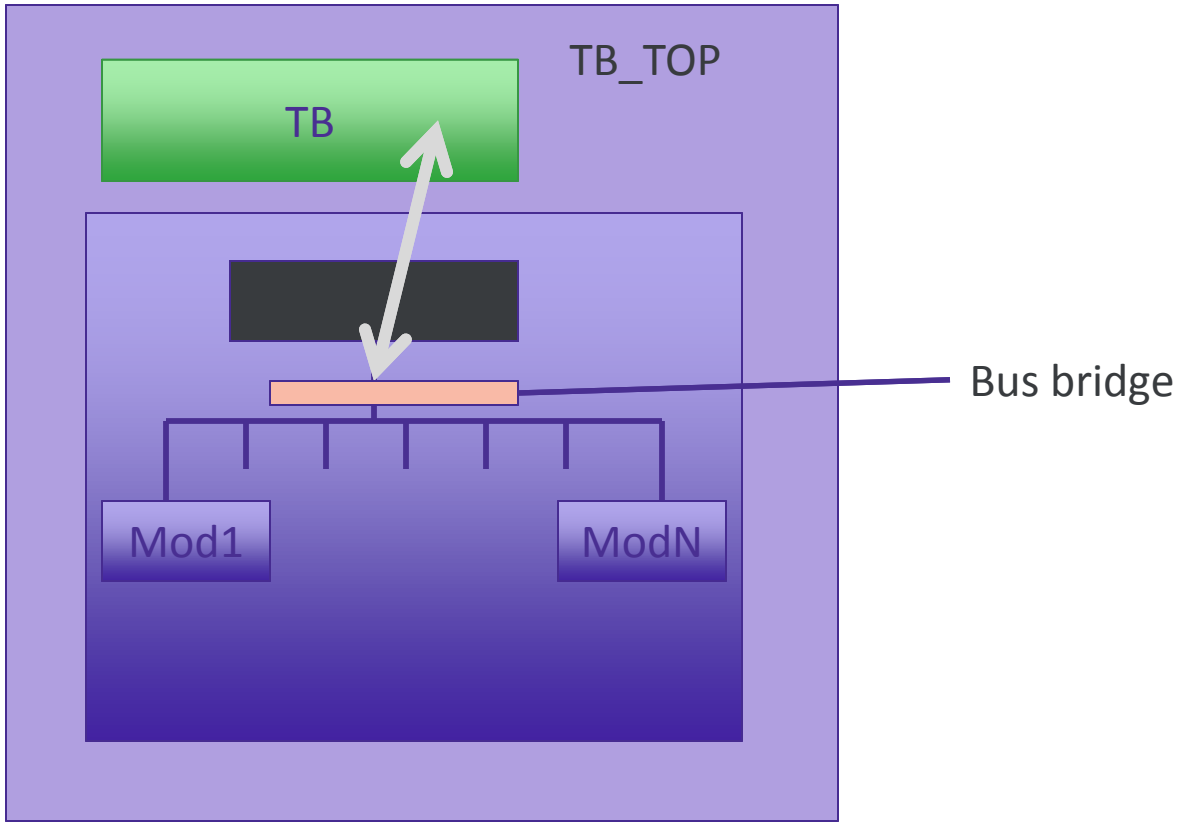
```
task body();
  do begin
    rdata = reg_model.reg.get_mirrored_value();
  while(rdata == 1);
  `uvm_do(seq1);
endtask
```

RAL - Replacing Processors



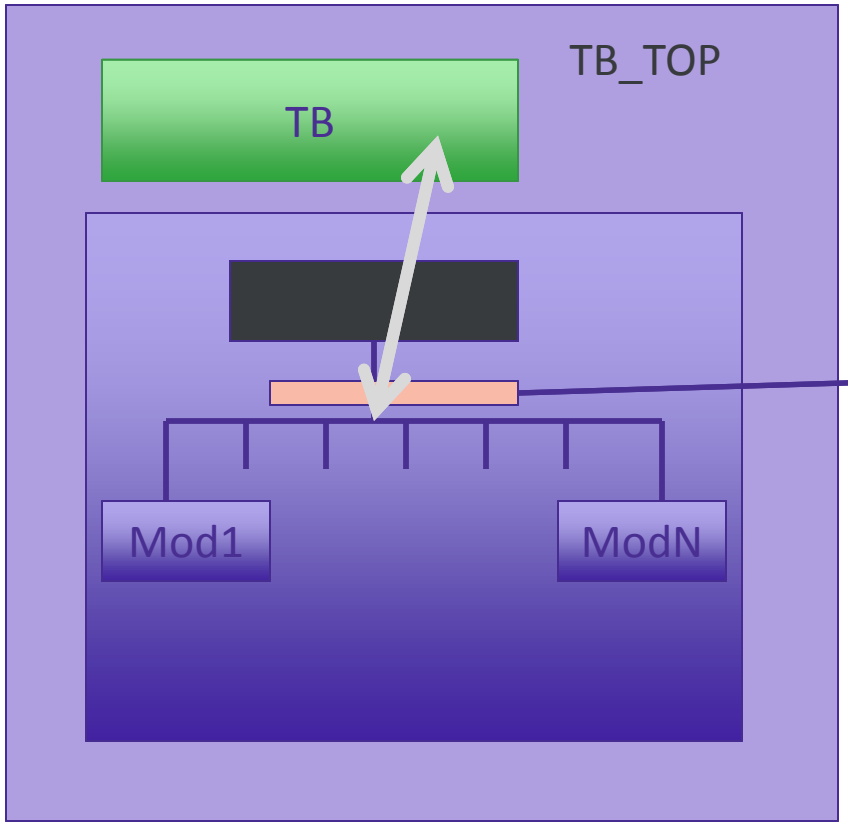
General testcase, processor code works as FW

RAL - Replacing Processors



RAL based sequences can be used to drive the traffic using UVM sequences
Few scenarios could be easily created

RAL - Replacing Processors



Bus bridge

RAL can be configured to drive traffic after the bridge as well

RAL based sequences can be used to drive the traffic using UVM sequences
Few scenarios could be easily created

Layered Sequences

- **Layered sequences help in scalability of tests**
  - To make it more scalable, have common testcase sequences with dummy body() which will be extended for final test sequences
    - Raise/drop objections
    - This way, any pre_body / post_body() need to be added to a class of tests, can be easily achieved
    - Code to wait for C-code to finish for SoC tests can be put in post_body()

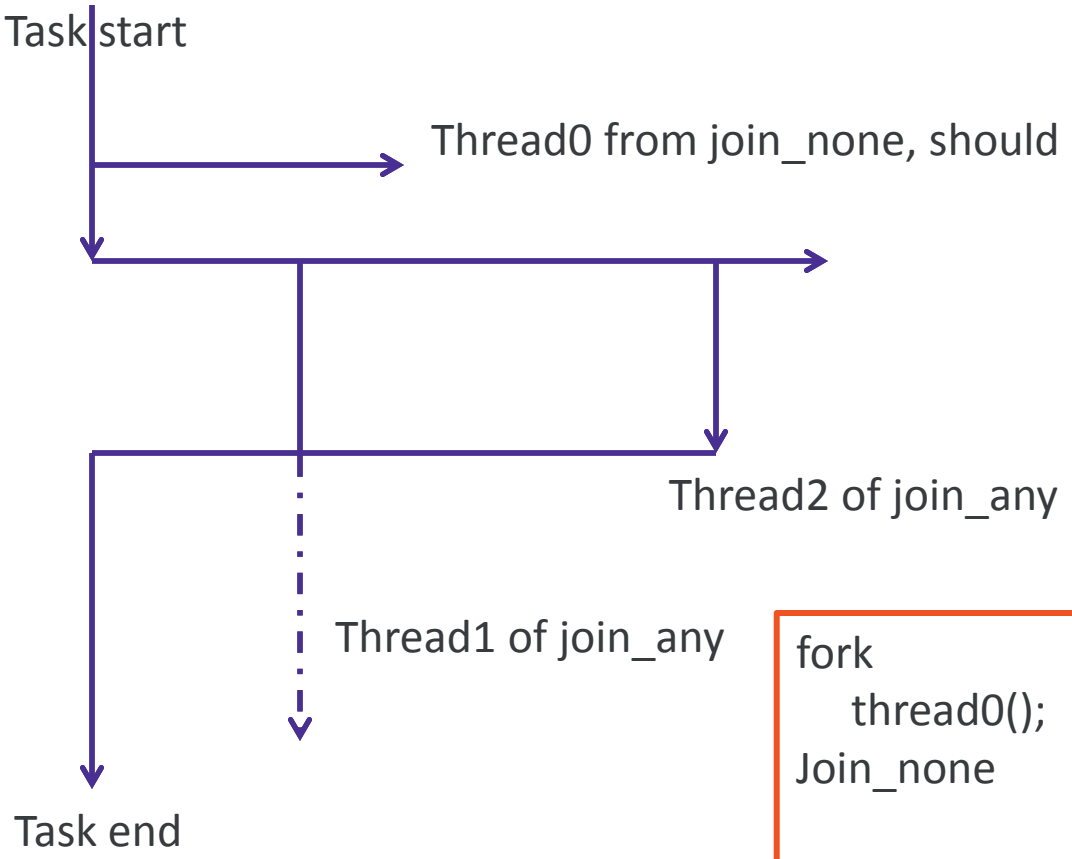
```
class chip_test extends uvm_test;
   ….
endclass
class c_chip_test extends uvm_test;
   …. ( defines pre_body() & post_body() )
endlcass
class sv_chip_test extends uvm_test;
   …. ( defines pre_body() & post_body() )
endlcass
```

```
class test1 extends c_chip_test;
   ….
endclass
class test20 extends sv_chip_test;
   ….
endlcass
```

Handling join_any

- **Fork – join_any + disable**
  - Caution is needed while using fork-join_any
  - After fork-join_any is done, only one thread would have completed execution and other threads started inside the fork are still running
  - Sometime, need to stop the other threads
  - If other threads need to be stopped, disable fork can be used
  - However, disable fork will terminate all child threads in a thread, where it is called
    - Affect might be at a place one don't expect
  - EXAMPLE diagram in next slide
    - Code in red box would terminate thread0() as well once thread2() is completed
    - Where as code in green box works as intended, kills thread1() once thread2() is over

Example of join_any




```
fork
    thread0();
Join_none

fork
    thread1();
    thread2();
join_any
disable fork;
```

```
fork
    thread0();
Join_none

fork
  begin
   fork
    thread1();
    thread2();
   join_any
   disable fork;
   end
join
```

Extendibility

- **Add hookup tasks inside monitor / sequence library code**
  - So that, if user wants to add some specific code, one can extend the class and define this tasks
  - With this, one need not copy the whole code from monitor / sequence and edit that
    - Otherwise, any updates to common UVC are not ported to user's environment
  - Especially after capturing the data from the interface signals into monitor item / before driving the data item onto interface
- **Use RAL based sequences to configure the DUT, instead of using system bus UVC sequences**
  - This will make porting tests from one environment to other environment easy, where system bus is not same

Controlling the LOG

- **No hard and fast rules to determine the Verbosity for UVM_INFO messages**
- **After adapting the UVM, now we have few guidelines on how we can determine on the verbosity for messages**
  - During development of UVM code, recommend to use UVM_NONE
  - Once the UVC / scoreboard / Monitor is working properly, then increase the level of verbosity level for all UVM_INFO messages
  - UVM_LOW for messages which occur once in a while (like interrupts, exception)
  - UVM_MEDIUM for messages which are important to track the flow of testcase
  - Any other messages UVM_HIGH, UVM_FULL
- **Avoid using $display as UVM_INFO is available**

Performance - Reduce usage of system tasks

- **Avoid using system tasks inside active threads of monitors / sequences**
  - Use them in new() or similar tasks which are called once, so that system task is not called too many times

```
class monitor;
   task run();
      forever
      begin
         @vif.cb;
          if(vif.req ==1 && vif.resp == 0
             && $test$plusargs("check_resp"))
              `uvm_error("ERR","err with resp");
      end
   endtask : run
endclass
```

```
class monitor;
   bit check_resp;
   task new();
       check_resp =
             $test$plusargs("check_resp");
   endtask
   task run();
      forever
      begin
         @vif.cb;
          if(vif.req ==1 && vif.resp == 0
             && check_resp == 1)
          `uvm_error(…);
      end
   endtask : run
endclass
```

Performance with UVM

- **While adding checkers / scoreboards at top level, for design correctness, ensure proper knobs are added to enable/disable them**
  - Otherwise, simulation performance is observed to be low with all checkers/scoreboards at top level
  - Make the knobs controllable through run-time arguments to avoid multiple compilations
- **At top level, tests can be written with SV by using sequences to drive transactions at bus level for system bus instead of processor C / ASM code**
- **Any checks that are done for every clock, will reduce the performance**

Constraints

- **Avoid writing unnecessary constraints in sequence items, especially for UVCs**
- **Add constraints only on need basis**
  - Ex: delay, byte_en

Constraints in seq_item

```
class seq_item;
  ....
  int dly;
  bit [3:0] byte_en;
  constraint c_dly { dly < 10; }
  constraint c_byte_en {
      soft byte_en == 4'hf; }
endclass
```

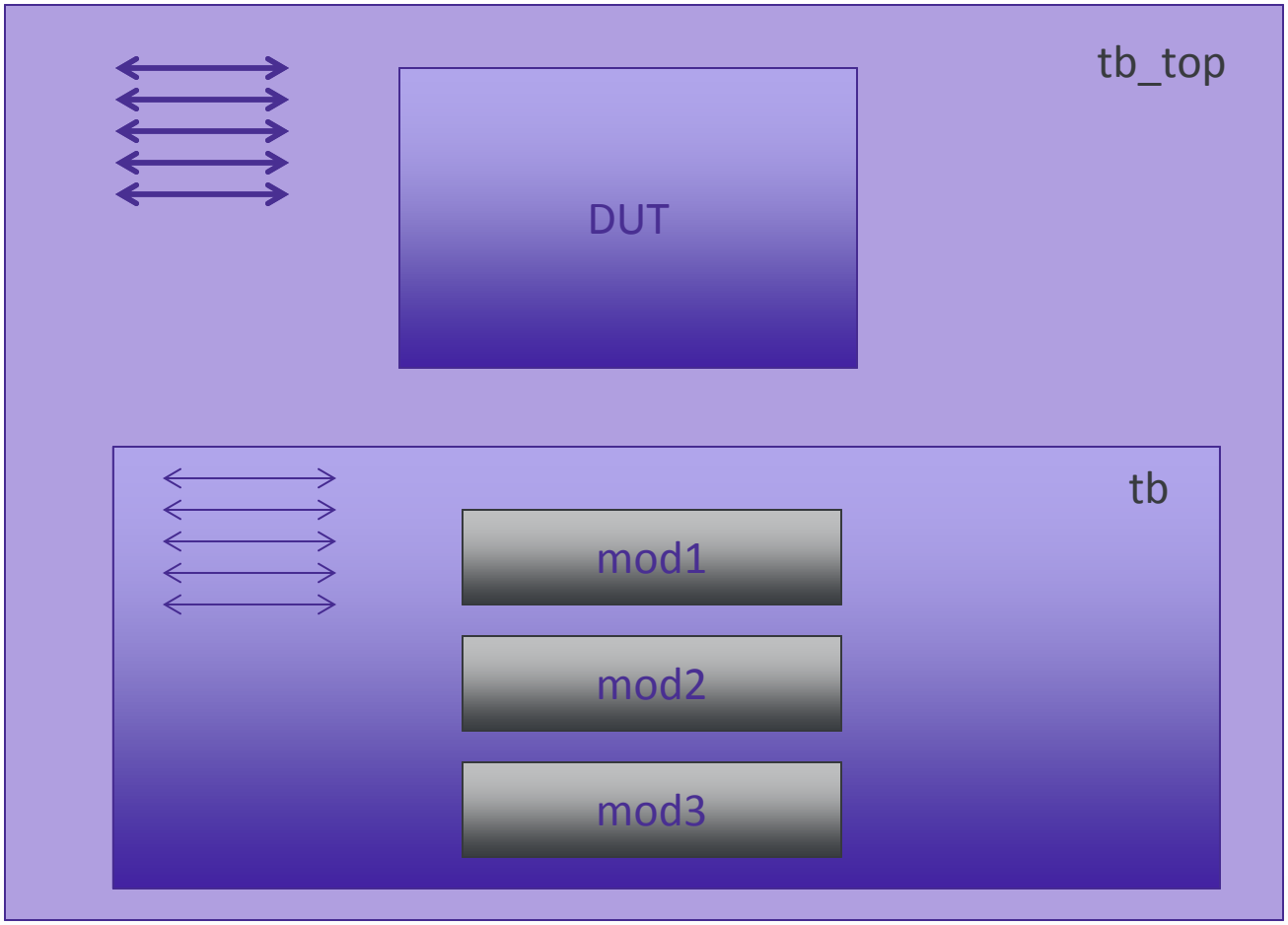
This constraint make user extend the item and modify the constraint for achieving dly >= 10

For byte_en, even if user wants all possible values, unless .byte_en != 4'hf is given all the time, byte_en would be 4'hf

Few Techniques we followed @ Audience Inc

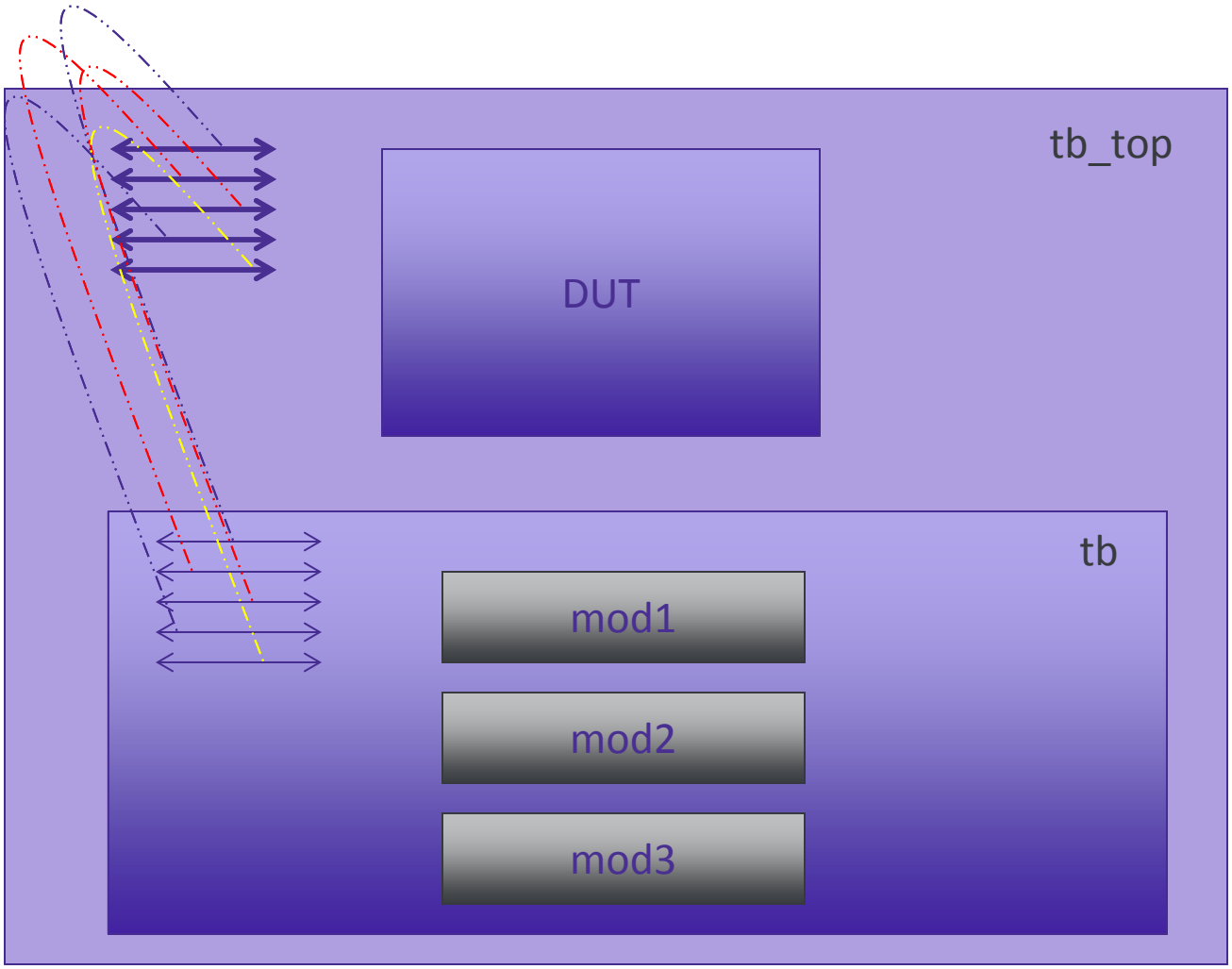
- **Interafces**
  - Multiple interfaces of same type
    - Ex: If 20 ahb interfaces are needed, tough to manage without array of interfaces
    - Use scripting to create the SV code with portno appended
- **Assertions**
  - Add assertions in the UVC protocol checkers / design configuration checks / scoreboards so that coverage collection of such assertions is made easy

Passing Interfaces



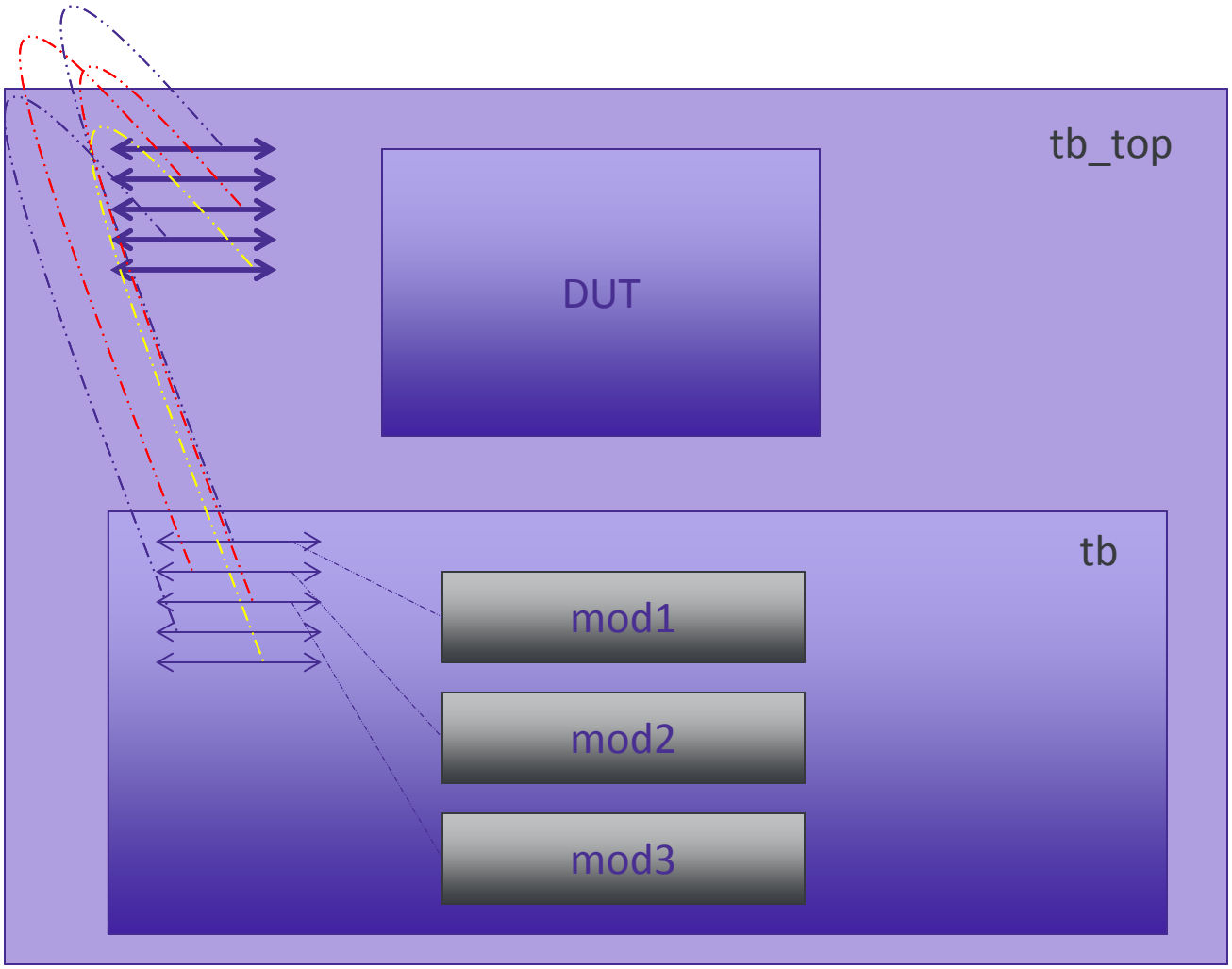
Instaniate interfaces at tb_top and virtual interfaces at tb

Passing Interfaces



Connect tb_top interfaces to tb virtual interfaces using config_db

Passing Interfaces



Now tb can connect the virtual interfaces to respective mod envs/ components

Useful techniques for quick results

- **Code development**
  - Use plusargs / plusargs based variables to try more than one variant of implementation / option in scoreboard / sequences
    - Otherwise, multiple compilations will lead to more time to close on the implementation
- **Code debugging**
  - When a debug is targeted for a specific monitor / scoreboard, better enable the full verbosity only for those modules instead of making +UVM_VERBOSITY=UVM_FULL
    - Following cmd argument will set the verbosity of specified instance to UVM_FULL, keeping whole other environment's verbosity to default
      - +uvm_set_verbosity="uvm_test_top.tb0.inst1.inst2.inst3*",_ALL_,UVM_FULL,run
- **UVC hand-off**
  - Develop sequence library containing sequences which will be needed by the UVC user (write/read/write-and-read/wait-readdata)
- **Testcase development**
  - Develop random tests and constrain those tests to create directed testcases
  - Try to keep number of tests to minimal to avoid more changes for a new requirement / constraint in design configuration

Acknowledgements

- **Thanks to Kartik Raju and other members of the VLSI team at Audience Inc, who are very objective oriented and always open for new ideas and methodologies**

Thank You