UVM Architecture For Performance: Go Hierarchical!

Paul Lungu, Zygmunt Pasturczyk, James Strober, Ciena, Ottawa
Christopher Taylor, Coveloz, Ottawa
Chris Thompson, Synopsys, Ottawa

April 21, 2017
Canada

Agenda

What problem do we solve?

Why hierarchical architecture?

Reusable set of parameterized base classes

Build your architecture

Challenges

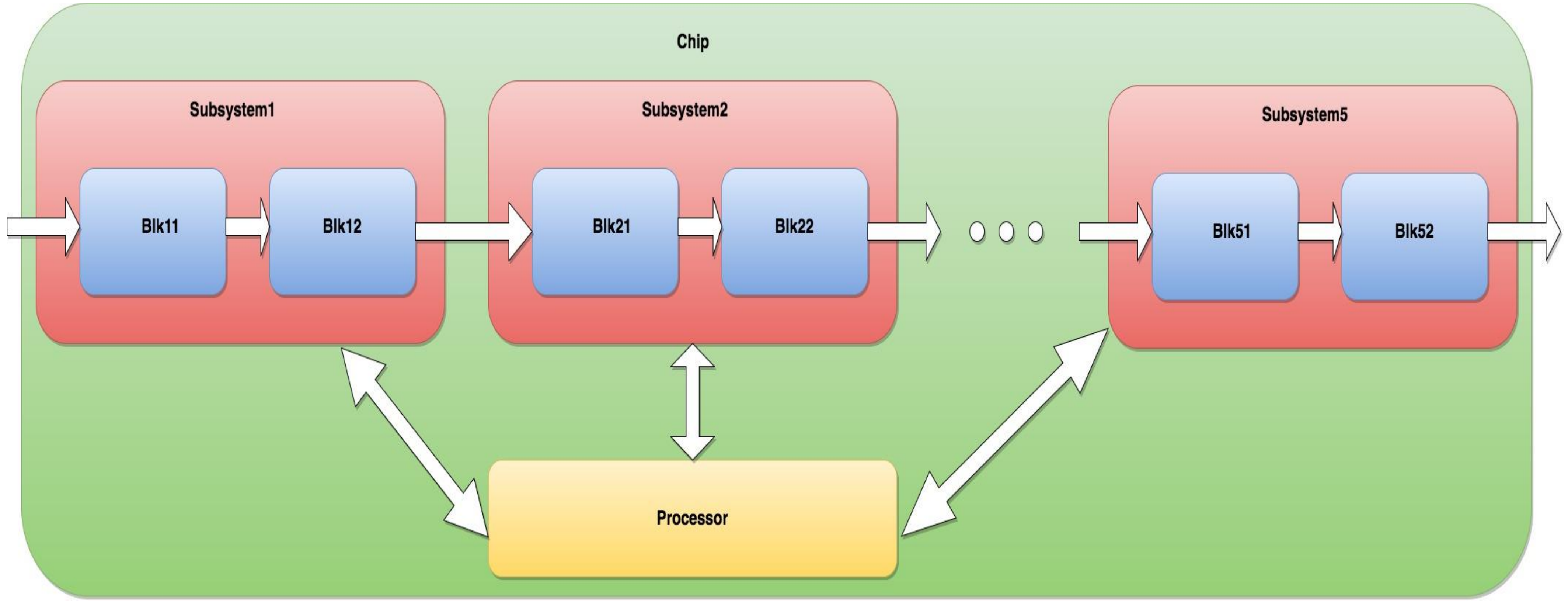
Conclusions and future work

What problem do we solve?

What problem do we solve?

- ASIC made of several subsystems
- A subsystem is made of two or more blocks
- Verification organization to mimic design hierarchy at all levels
- Different verification teams for each subsystem
- Designers/verification people working on verification at block level
- Work to be done independently in each subsystem
- Tight schedule requires toplevel sanity before subsystems are fully verified

What problem do we solve?



Why hierarchical architecture?

Why hierarchical architecture?

- Maximize the use of resources
- Allows each team to completely focus on a reduced verification scope
- Minimize the amount of interaction between people from different teams
- Chip level verification to reuse lower level components
- Chip level verification almost fully independent
- This presentation to focus on higher level hierarchy – low level see paper

Why hierarchical architecture ?

- **Subsystem config to constraint block level config variables**
- **Toplevel config to constraint subsystem level variables**

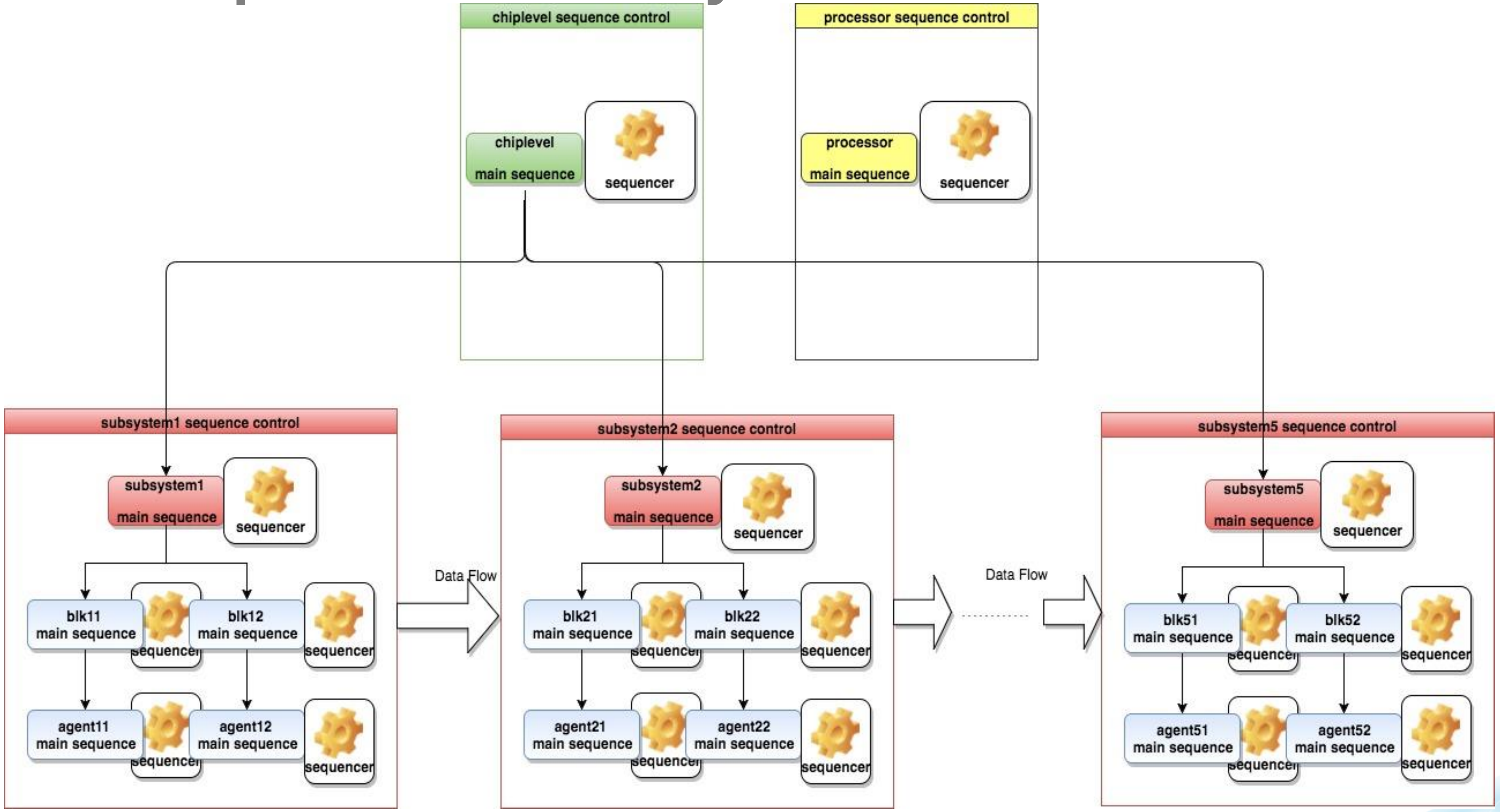
- **Subsystem level sequences to start block level sequences**
- **Toplevel sequences to start subsystem level sequences**

- **Subsystem interfaces to reuse block level interfaces**
- **Toplevel interface to reuse subsystem level interface**

- **Subsystem to reuse block level interface registration via config_db**
- **Toplevel to reuse subsystem interface registration via config_db**

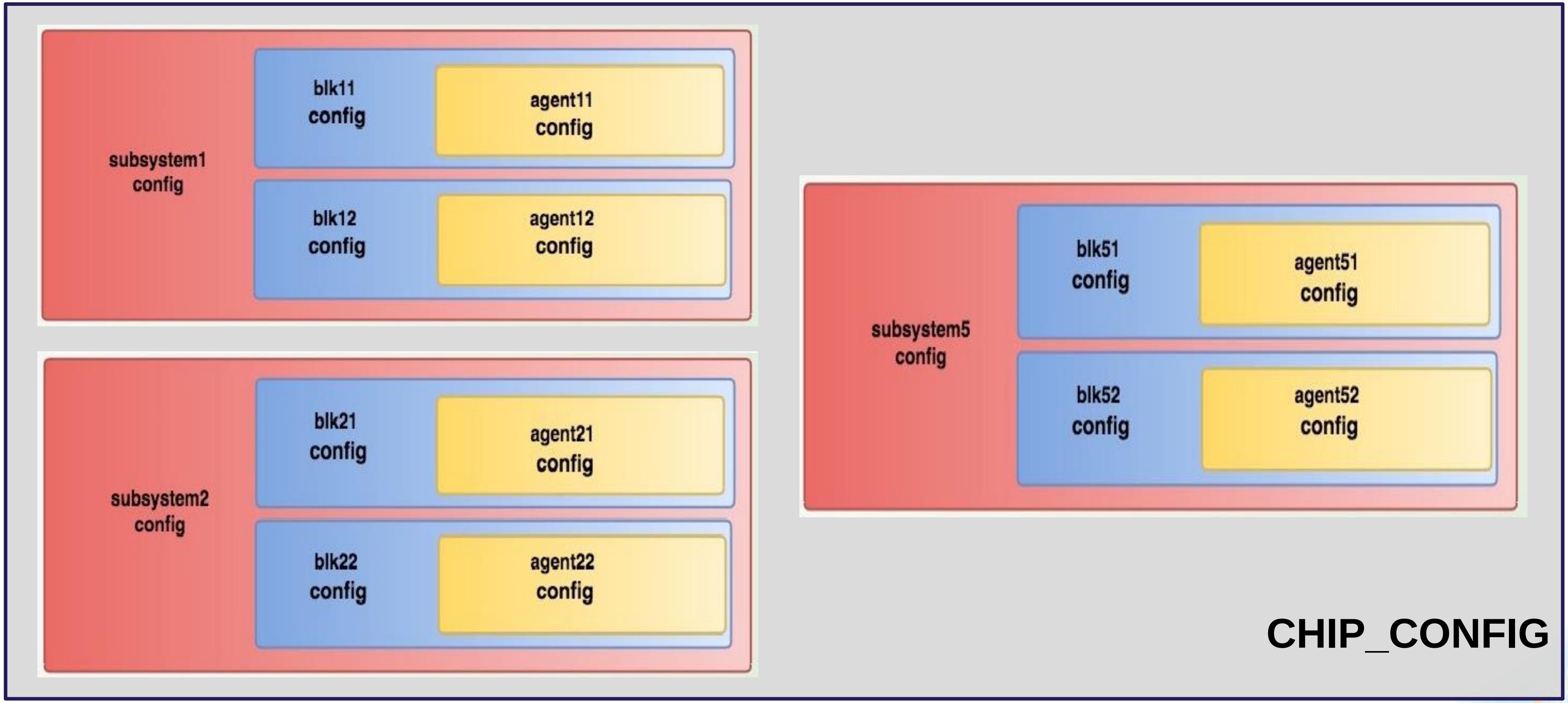
Why hierarchical architecture?

- Sequence hierarchy



Why hierarchical architecture?

- Config hierarchy

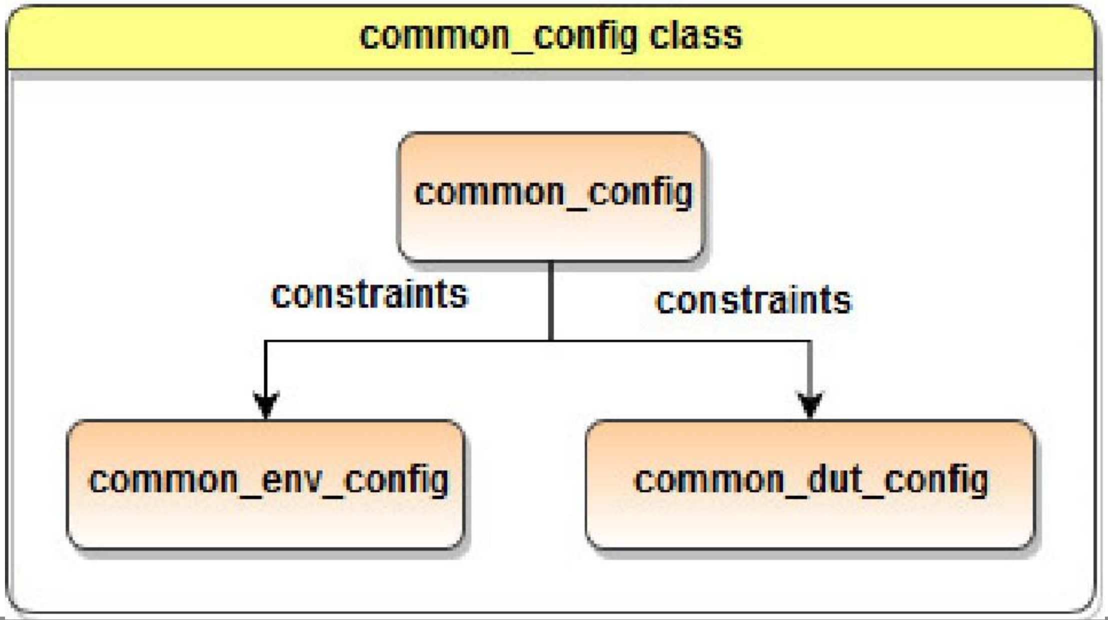


Reusable set of parameterized base classes

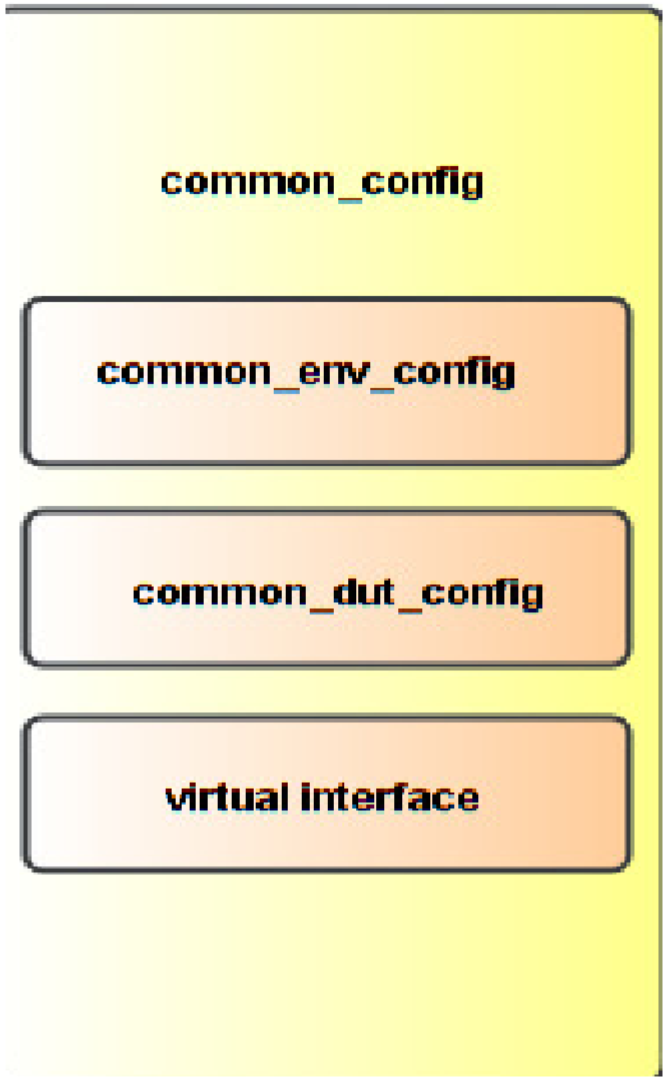
Reusable set of parameterized base classes

- common_config class
- common_sequence base class
- common_sequencer base class
- common_env base class
- common_base_uvm_test base class

common_config base class

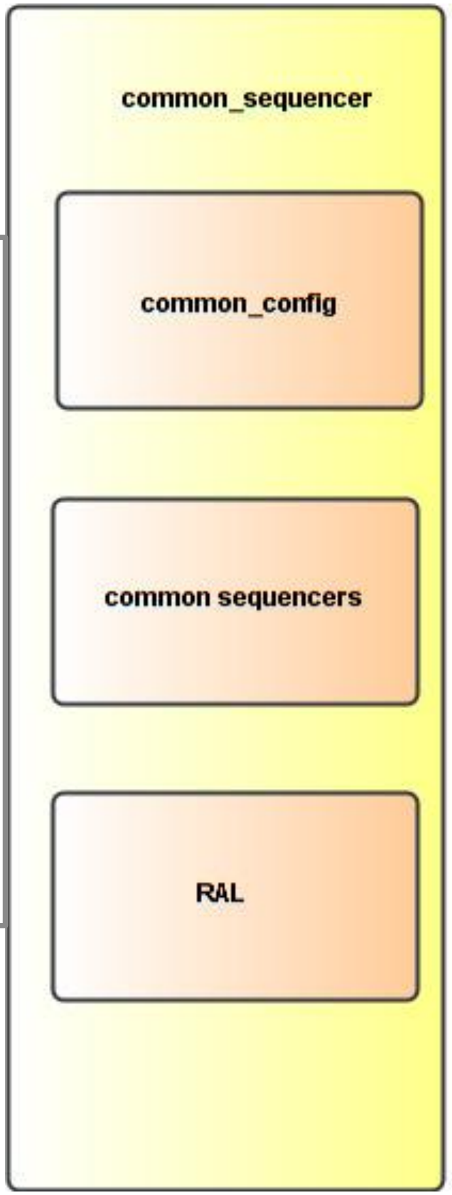


```
class common_config #(type ENV_CONFIG_T = uvm_object,
                      type DUT_CONFIG_T = uvm_object,
                      type VIF_T = virtual dummy_if
                     ) extends uvm_object;

  rand ENV_CONFIG_T m_env_config;
  rand DUT_CONFIG_T m_dut_config;
  VIF_T itf;
...
endclass: common_config
```



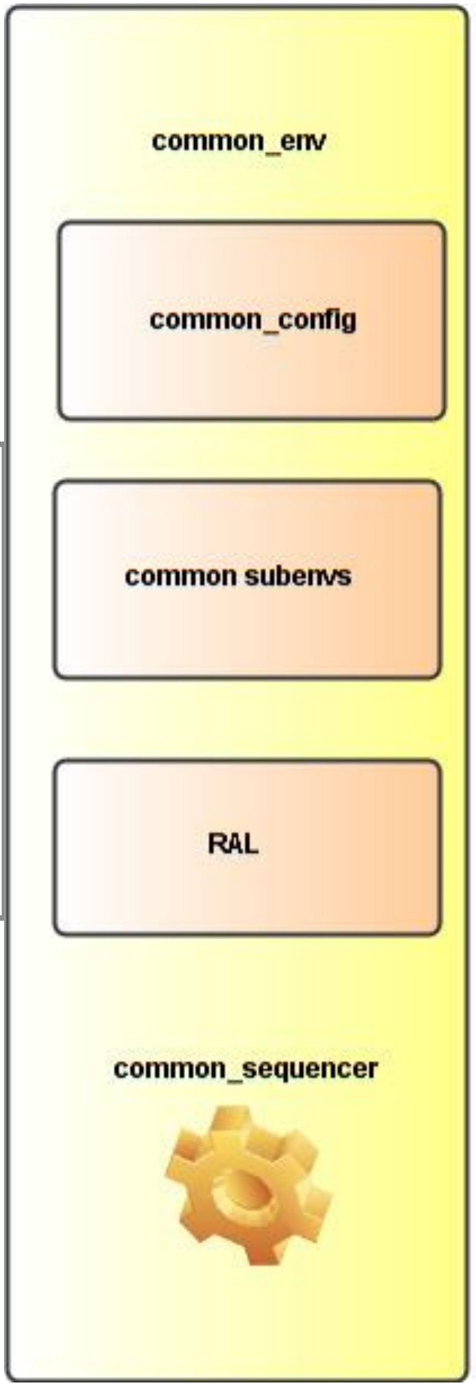
common_sequencer base class

```
class common_base_sequencer #(type CONFIG_T = uvm_object ,
                              type RAL_T = uvm_reg_block,
                              int  RST_NUM_ENB  = 0,
                              int  EBUS_NUM_ENB = 0,
                              int  GMS_NUM_ENB  = 0
                              ) extends uvm_sequencer;
..
  rst_sequencer            m_rst_seqr[RST_NUM_ENB];
  ebus_sequencer           m_ebus_seqr[EBUS_NUM_ENB];
  gms_master_sequencer     m_gms_seqr[GMS_NUM_ENB];
  CONFIG_T                 m_config;
  RAL_T                    RAL; ...
endclass : common_base_sequencer
```

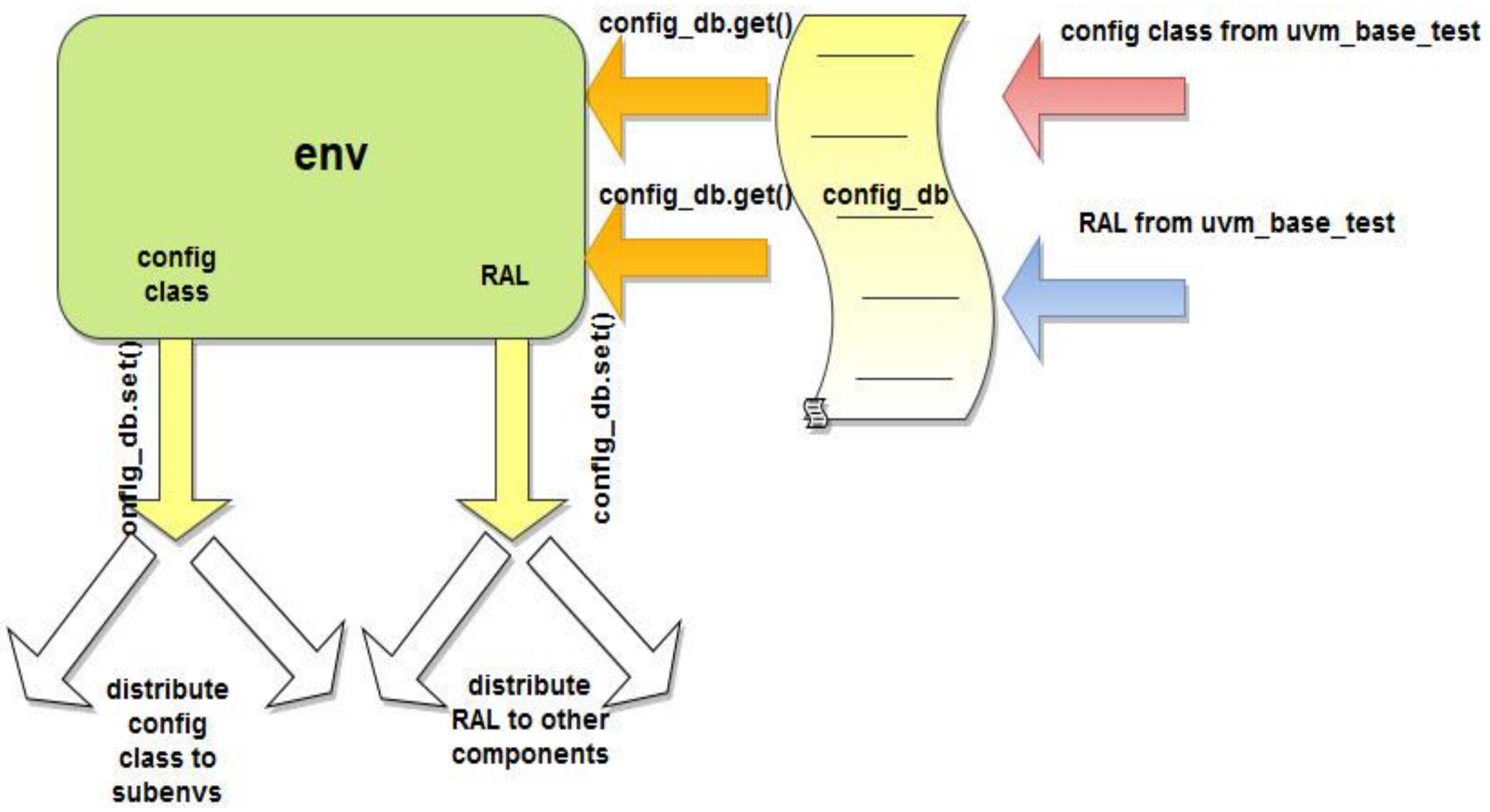


common_env base class

```
class common_env#(type CONFIG_T= uvm_object,
                 type BASE_SEQR_T= uvm_sequencer,
                 type RAL_T = uvm_reg_block,
                 string CFG_STR= "",
                 string RAL_STR= "",
                 string BASE_SEQR_STR= ""
                 ) extends uvm_env;
...
```

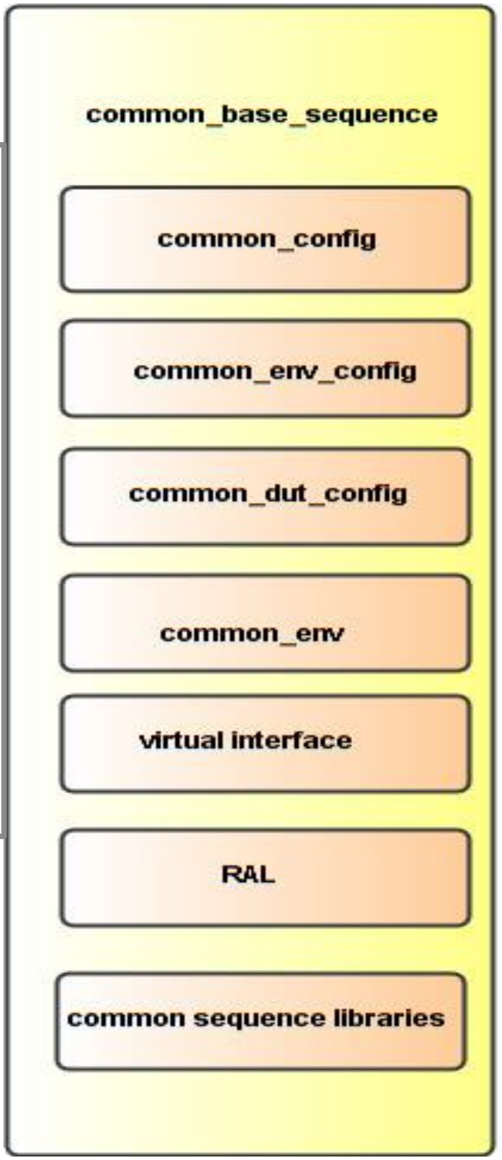


common_env base class (cont'd)



common_sequence base class

```
class common_base_phase_seq #(type ENV_T = uvm_env,
                              type CONFIG_T = uvm_object,
                              type ENV_CONFIG_T = uvm_object,
                              type DUT_CONFIG_T = uvm_object,
                              type API_T = uvm_object,
                              type BASE_SEQR_T = uvm_sequencer,
                              type RAL_T = uvm_reg_block,
                              type VIF_T = virtual dummy_if,
                              int  RST_NUM_ENB = 0)
                            extends uvm_sequence;
```

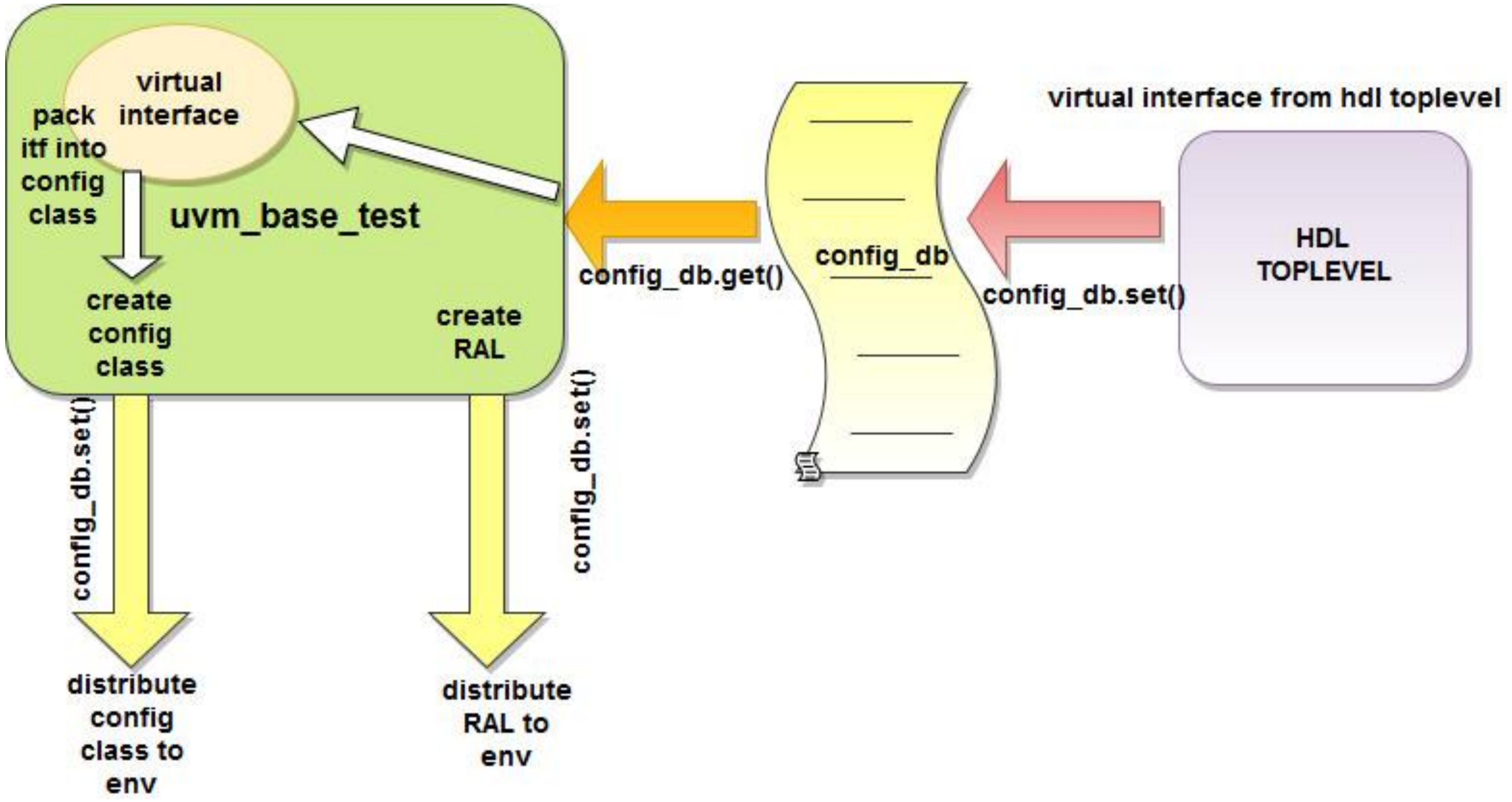


common_sequence base class (cont'd)

- It offers access to hierarchical interface, RAL, all config classes
- It instantiates predefined sequence libraries (ie. reset sequence)

```
    ENV_T           m_env;
   CONFIG_T         m_config;
   DUT_CONFIG_T     m_dut_config;
   ENV_CONFIG_T     m_env_config;
   RAL_T            RAL;
   VIF_T            itf;

   // Example of a functional Sequence Library
   rst_seq_lib      rst_lib[RST_NUM_ENB];
...
endclass : common_base_phase_seq
```

common_uvm_base_test base class



Build your architecture

Build your architecture

Generic high level env class

Generic high level sequencer class

Generic high level sequence class

Generic uvm base test for any level

blk_env class

```
typedef common_env #(blk_config, blk_sequencer, dut_sys_ral,
                     "blk_config", "ral_sys_dut", "blk_sequencer")
blk_env_base_t;

class blk_env extends blk_env_base_t;
  `uvm_component_utils(onramp_env)

  blk1_env       m_blk1_env[];
  blk2_env       m_blk2_env[];
  blk_scbd       m_scbd; // optional

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

  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void connect_phase(uvm_phase phase);
 ...
endclass: blk_env
```

blk_sequence class

- Design hierarchy is followed at all sequence levels

```
class blk_main_seq extends blk_base_phase_seq_t;
  `uvm_object_utils(blk_main_seq)

  // add all block level sequences as members of this class
  blk1_main_seq        m_blk1_main_seq;
  blk2_main_seq        m_blk2_main_seq;

  function new(string name="blk_main_seq");
    super.new(name);
    m_blk1_main_seq = new;
    m_blk2_main_seq = new;
  endfunction : new

  task pre_body();
    super.pre_body();
    m_blk1_main_seq=blk1_main_seq::type_id::create("m_blk1_main_seq");
    m_blk2_main_seq=blk2_main_seq::type_id::create("m_blk2_main_seq");
  endtask ...
```

blk_sequence class (cont'd)

- Sequences are started in parallel at all levels
- Any lower hierarchy level sequence can be started at higher levels

```
task body();
     super.body();
    // start in paralell sub-block level sequences
    fork
    begin
      if(m_env_config.m_enable_blk1_in)
        m_blk1_main_seq.start(m_env.m_base_seqr.m_blk1_seqr);
    end
    begin
      if(m_env_config.m_enable_blk2_in)
        m_blk2_main_seq.start(m_env.m_base_seqr.m_blk2_seqr);
    end
    join_none
   wait fork;
 endtask
endclass : blk_main_seq
```

uvm_blk_test class

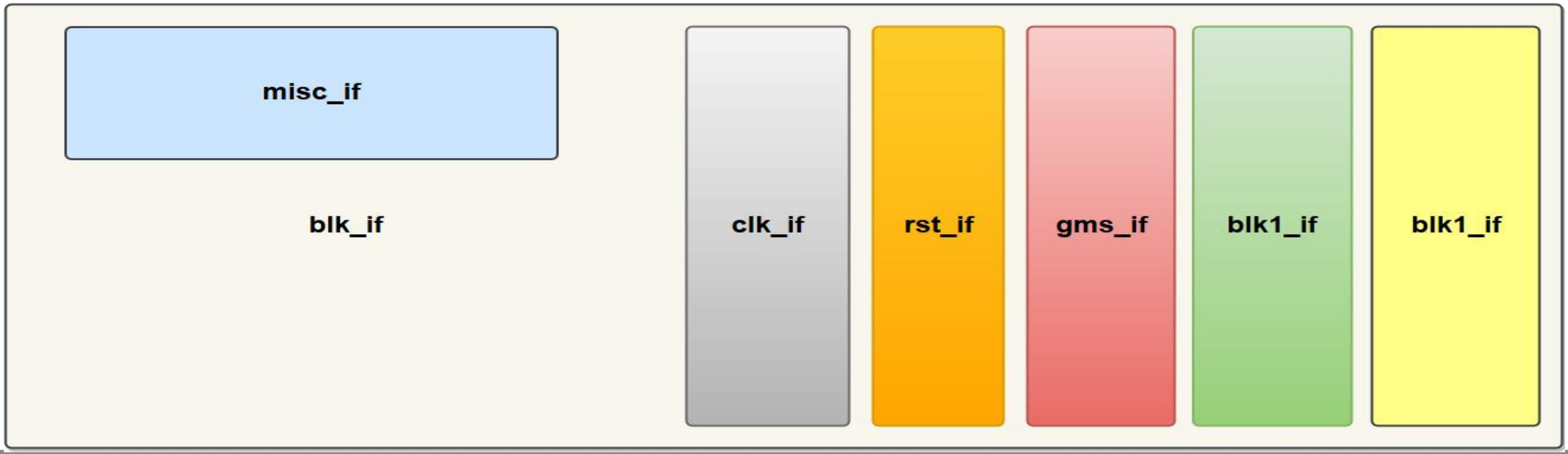
```
typedef  common_test_base#(blk_env,  blk_config, blk_env_config,
                           blk_dut_config, blk_env_api,  blk_reset_seq,
                           blk_config_seq, blk_main_seq, blk_shutdown_seq,
                           dut_sys_ral,  virtual blk_if,  "blk_config",
                           "ral_sys_dut",  "blk_itf")  blk_base_uvm_test_t;

class blk_base_uvm_test extends blk_base_uvm_test_t;
  `uvm_component_utils(blk_base_uvm_test)

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

  virtual function void test_config();
    // override any variable specific for this particulat level of hierarchy
    // this is a place to override constraints if needed
  endfunction
…
```

uvm_blk_test class (cont'd)

- Initialize lower level hierarchical interfaces inside the build_phase() of the base_uvm_test

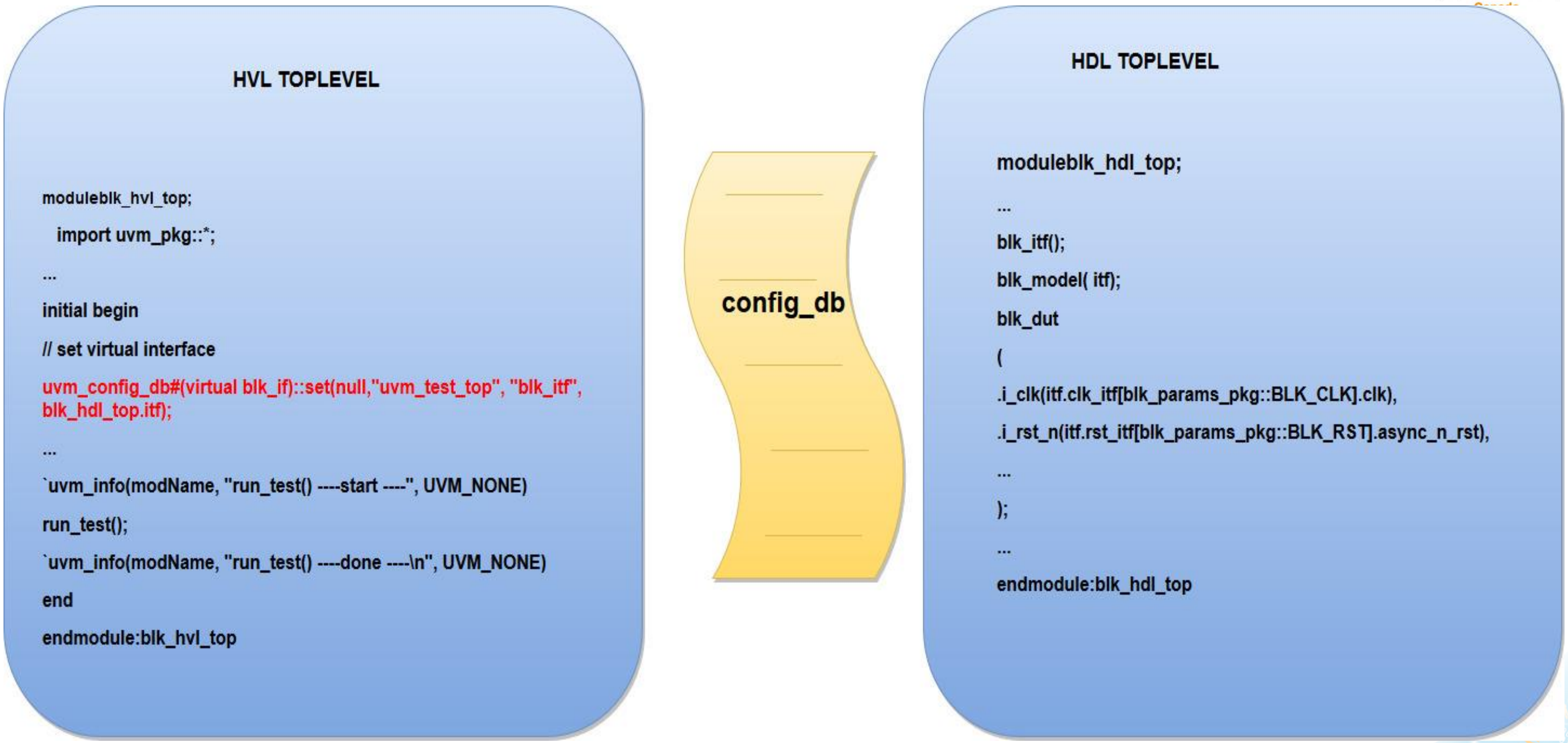
```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_config.m_blk1_config.itf = m_config.itf.sblk1_itf;
    m_config.m_blk2_config.itf = m_config.itf.sblk2_itf;
    m_config.itf.RAL = this.RAL;
endfunction : build_phase
endclass: blk_base_uvm_test
```

Hierarchical interfaces



```
interface blk_if();
   import blk_params_pkg::*; ...

  clk_if    clk_itf[NUM_CLK_IF](); // clock interface not reusable at higher levels
  rst_if    rst_itf[NUM_RST_IF](1'b0);  // reset interface not reusable at higher levels
  blk1_if   blk1_itf[NUM_SBLK1_IF]();  // fully reusable at higher levels
  blk2_if   blk2_itf[NUM_SBLK2_IF]();  // fully reusable at higher levels
  gms_if    gms_itf[NUM_GMS_IF]();     // processor interface not reusable at highe
rlevels
  blk_misc_if   misc_itf(.clk(clk_itf[BLK_CLK].clk)); ...
endinterface
```

Toplevels



HVL TOPLEVEL

```
moduleblk_hvl_top;
  import uvm_pkg::*;
...
initial begin
// set virtual interface
uvm_config_db#(virtual blk_if)::set(null,"uvm_test_top", "blk_itf",
blk_hdl_top.itf);
...
`uvm_info(modName, "run_test() ----start ----", UVM_NONE)
run_test();
`uvm_info(modName, "run_test() ----done ----\n", UVM_NONE)
end
endmodule:blk_hvl_top
```

HDL TOPLEVEL

```
moduleblk_hdl_top;
...
blk_itf();
blk_model( itf);
blk_dut
(
.i_clk(itf.clk_itf[blk_params_pkg::BLK_CLK].clk),
.i_rst_n(itf.rst_itf[blk_params_pkg::BLK_RST].async_n_rst),
...
);
...
endmodule:blk_hdl_top
```

Challenges

Challenges

Explain why this architectural choice?

The number of files/classes needed to maintain grows with hierarchy level

Maintaining non class based components (ie. interfaces) to follow hierarchy

Subsystems have to be compliant with all architectural rules

Blocks and subsystems have to be UVM compliant

Conclusions and future work

Conclusions and future work

1. The methodology considerably reduced the time to verify a complex ASIC
2. It reduced the interaction between groups
3. It allowed blocks to be independently verified in short time
4. It enabled toplevel verification with basic block sanity
5. Reduced project risk
6. Future work: make the architecture common for FPGAs and multiple ASICs
7. Add an extra layer of common classes which is project specific

Thank You