2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

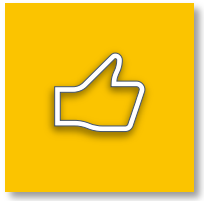
Plug and Play Verification: Leveraging Block-Level Components for Seamless Top-Level Integration

Tchiya Dayan

mobileye™

accellera SYSTEMS INITIATIVE

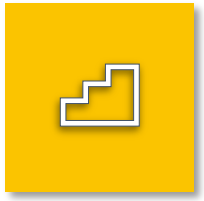
Agenda



Introduction

Challenges in Top-Level Test Creation

Proposed Solution



Implementation Steps

Results

Summary

Introduction

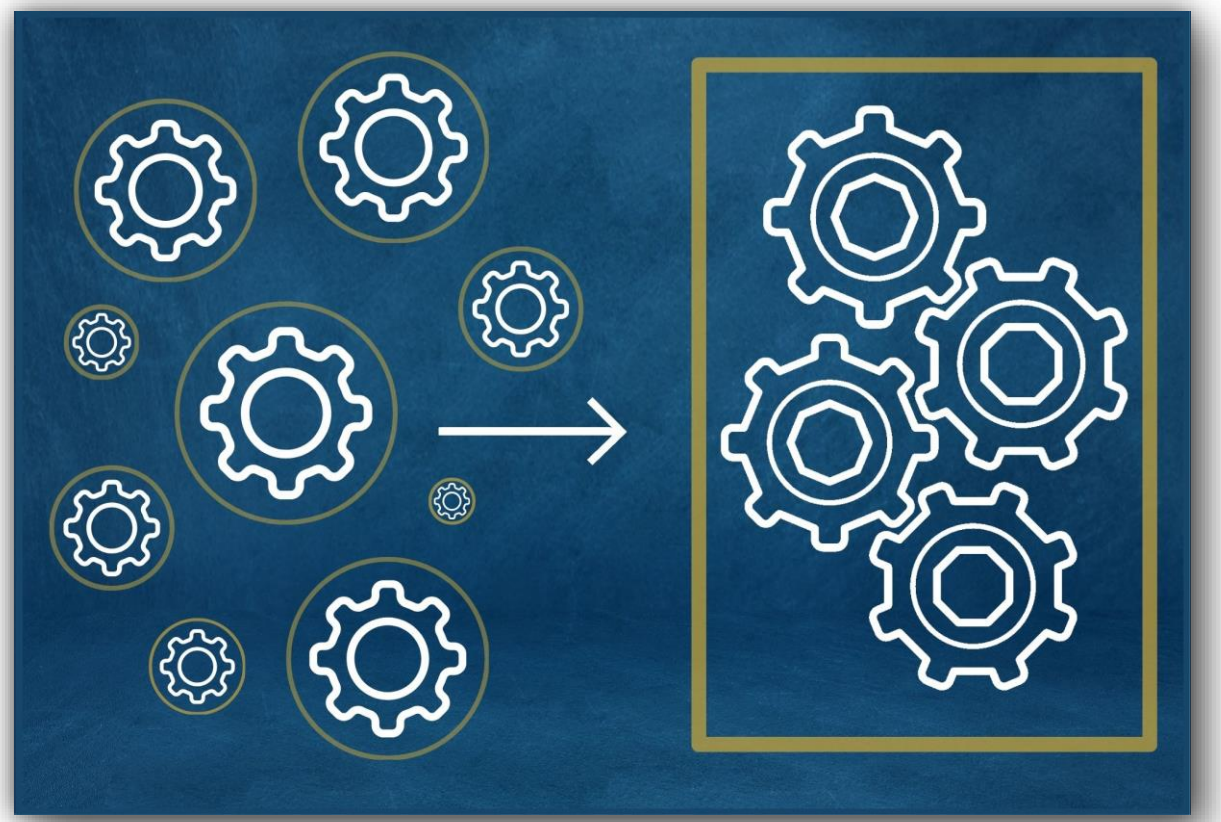
- Growing design complexity demands both block-level and top-level verification
- Time constraints often initiate top-level work before block-level completion
- Many options for reusing the static parts of the testbench, but few for the dynamic parts

Challenges in Top-Level Test Creation

- Precise selection of test sequences from block-level sets is crucial
- Accurate placement within the flow for multi blocks scenarios
- Deep understanding of block-level elements for correct inputs and settings
- Non-reusable "copy-paste" methods often lead to misalignment issues

Seamless integration between block and top-level simulations is essential for efficiency and error reduction.

Proposed Solution

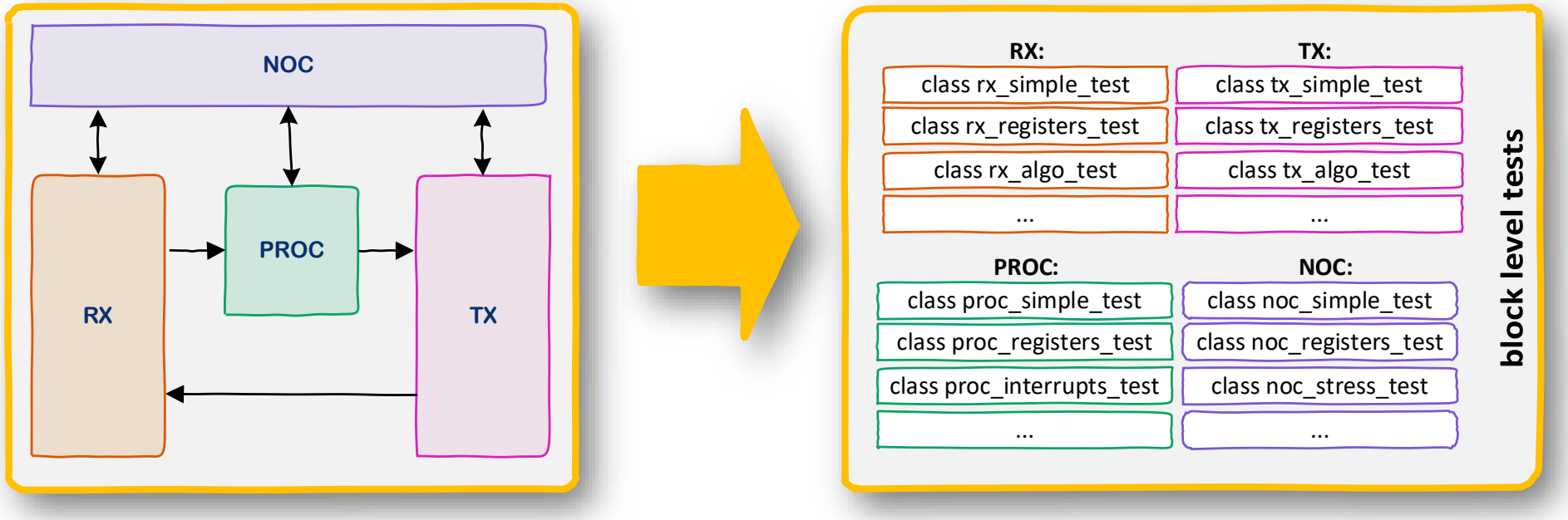


Proposed Solution

- Block-level test component instantiated in the top-level test
- Parametric test to reuse the top-level test code
- Common base-class for block-level cfg and template pattern for test elements
- Synchronization between block-level test components flow
- Adding additional logic reuse for top functions/phases
- Randomization of test content – replacing inline constraints

Proposed Solution

DUT example



Implementation Steps

A. Block-level test component instantiated in the top-level test

Eliminating the need for importing and maintaining individual block-level test sub-elements

- Using entire block-level tests as UVM components in top-level test
- Adjusting environment settings through accessing relevant test components

Implementation Steps

A. Block-level test component instantiated in the top-level test


```
class top_rx_simple_tx_algo_test extends uvm_test;
   rx_simple_test m_rx_test;
   tx_algo_test     m_tx_test;

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      m_rx_test = rx_simple_test::type_id::create::(“m_rx_test”);
      m_tx_test = tx_algo_test::type_id::create::(“m_tx_test”);
      m_rx_test.cfg.tx_agent_cfg.m_is_active = UVM_PASSIVE;
   endfunction
   …
endclass
```

Implementation Steps

B. Parametric test to reuse top-level test code

- Enabling efficient reuse of top-level test logic and code
- Simplifying the creation of new top tests for various block-level test scenarios

Design top-level test as a superset using parametric test, incorporating sub test-components types as parameters

Implementation Steps

B. Parametric test to reuse top-level test code


```
class top_base_test #(   parameter type RX_TEST_TYPE, parameter type TX_TEST_TYPE, parameter type PROC_TEST_TYPE, ... ,
                         parameter string TEST_NAME = “”) extends uvm_test;
  `uvm_component_registry(top_base_test#( RX_TEST_TYPE ,TX_TEST_TYPE ,PROC_TEST_TYPE , … ,TEST_NAME ) , TEST_NAME)
  RX_TEST_TYPE        m_rx_test;
  TX_TEST_TYPE        m_tx_test;
  PROC_TEST_TYPE      m_proc_test;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_rx_test         = RX_TEST_TYPE::type_id::create::(“m_rx_test”);
    m_tx_test         = TX_TEST_TYPE::type_id::create::(“m_tx_test”);
    m_proc_test       = PROC_TEST_TYPE::type_id::create::(“m_proc_test”);
    ...
  endfunction
  …
endclass
```

Implementation Steps

B. Parametric test to reuse top-level test code

New top level test creation:

typedef <generic_top_level_test_T>#(<block_level_A_T>,<block_level_B_T>, … , <top_level_test_name_s>) top_level_test_name;

typedef **top_base_test**#(rx_simple_test,tx_simple_test,proc_simple_test, … ,"**rx_tx_proc_simple_test**") **rx_tx_proc_simple_test**;

Implementation Steps

C. Common base-class for block-level config/env and template pattern for test elements

- Ensuring uniformity and consistency across diverse test components
- Enhancing test flow modularity and organization

- Developing a common base-class for configuration, environment, etc.
- Utilizing a template pattern to create an array of test components with phase-specific time control

Implementation Steps

C. Common base-class for block-level config/env and template pattern for test elements


```
class base_cfg extends uvm_object;
   axi_cfg       m_axi_cfg_arr[];
   bit[31:0]     m_num_of_axi_cfg;

   virtual function void build_sub_config();
      m_axi_cfg_arr = new[m_num_of_axi_cfg];
      foreach(m_axi_cfg_arr[ii])
         m_axi_cfg_arr[ii] = axi_cfg::type_id::create(...);
   endfunction
endclass
```

Block-level base config

```
class rx_cfg extends base_cfg;
   virtual function void build_sub_config();
      m_num_of_axi_cfg = 3;
      ...
   endfunction
endclass
```

```
class template_test extends uvm_test;
   base_cfg m_cfg; //inheirted test is creating

   //build phase splitting
   extern virtual function setup_env_structure();
   extern virtual function build_config();
   extern virtual function post_randomize_config();
   extern virtual function build_env();

   //run phase splitting
   extern virtual task reset_dut();
   extern virtual task system_start();
   extern virtual task config_blocks();
   extern virtual task run_test();
endclass
```

Block-level base test

Implementation Steps

C. Common base-class for block-level config/env and template pattern for test elements


```
class top_base_test #(   parameter type RX_TEST_TYPE, parameter type TX_TEST_TYPE, … ,
                         parameter string TEST_NAME = “”) extends uvm_test;
  …
  template_test m_block_test[NUM_OF_BLOCKS];

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_block_test[RX]     = RX_TEST_TYPE::type_id::create::(“m_rx_test”);
    m_block_test[TX]     = TX_TEST_TYPE::type_id::create::(“m_tx_test”);
    m_block_test[PROC] = PROC_TEST_TYPE::type_id::create::(“m_proc_test”);
    foreach(m_block_test[ii])
      foreach(m_block_test[ii].m_cfg.m_axi_cfg_arr[jj])
        m_block_test[ii].m_cfg.m_axi_cfg_arr[jj].m_enable_burst_mode = 0;
    …
  endfunction
  …
endclass
```

Top-level base test

Implementation Steps

D. Synchronization between block-level test components flow

Achieving seamless synchronization of internal phases across block-level for harmonized top-level simulation flow

- Utilizing UVM phases' automatic scheduling only in block-level mode
- Employing top-level test as a scheduling layer for synchronizing internal phases execution

Implementation Steps

D. Synchronization between block-level test components flow

template_test

```
class base_test extends template_test;

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if(m_cfg.m_top_test==0)
         setup_env_structure();
         build_config();
         post_randomize_config();
         build_env();
      end
   endfunction : build_phase

//internal phases implementation
endclass
```

```
virtual task run_phase(uvm_phase phase);
      super.run_phase(phase);
      if(m_cfg.m_top_test==0)
         reset_dut();
         system_start();
         config_blocks();
         run_test();
      end
   endtask : run_phase
```

Block-level base test

Implementation Steps

D. Synchronization between block-level test components flow

Empty block-level UVM phases in top-level mode:

```
class base_test extends template_test;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(m_cfg.m_top_test==0)
      setup_env_structure();
      build_config();
      post_randomize_config();
      build_env();
    end
  endfunction : build_phase

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    if(m_cfg.m_top_test==0)
      reset_dut();
      system_start();
      config_blocks();
      run_test();
    end
  endtask : run_phase

//internal phases implementation
endclass
```

Implementation Steps

D. Synchronization between block-level test components flow

Block-level phases synchronization in top-level test:

```
class top_base_test #(   parameter type RX_TEST_TYPE, parameter type TX_TEST_TYPE, … , parameter string TEST_NAME = “”) extends uvm_test;
   base_test m_block_test[NUM_OF_BLOCKS];

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      m_block_test[RX]      = RX_TEST_TYPE::type_id::create::(“m_rx_test”);
      m_block_test[TX]      = TX_TEST_TYPE::type_id::create::(“m_tx_test”);
      m_block_test[PROC] = PROC_TEST_TYPE::type_id::create::(“m_proc_test”);

      foreach(m_block_test[ii])
         m_block_test[ii].setup_env_structure();

      foreach(m_block_test[ii])
         m_block_test[ii].build_config();
      …
   endfunction : build_phase

   virtual task run_phase(uvm_phase phase);
      super.run_phase(phase);
      foreach(m_block_test[ii])
         fork
            m_block_test[ii].reset_dut();
         join_none; wait fork;

      foreach(m_block_test[ii])
         fork
            m_block_test[ii].system_start();
         join_none; wait fork;
      …
   endtask : run_phase

endclass
```

Implementation Steps

E. Adding additional logic reuse for top functions/phases

Enabling cross block-level adjustments in the top-level by inserting logic at predefined stages

Creating template tasks in top-level tests after each block-level phase executing for efficient customization

Implementation Steps

E. Adding additional logic reuse for top functions/phases


```
class top_base_test #(   parameter type RX_TEST_TYPE, parameter type TX_TEST_TYPE, … , parameter string TEST_NAME = “”) extends uvm_test;

  virtual function void build_phase(uvm_phase phase);
    ...
    foreach(m_block_test[ii])
      m_block_test[ii].setup_env_structure();
    top_setup_env_structure();

    foreach(m_block_test[ii])
      m_block_test[ii].build_config();
    top_build_config();
    ...
  endfunction : build_phase

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    foreach(m_block_test[ii])
      fork
        m_block_test[ii].reset_dut();
      join_none; wait fork;
    top_reset_dut();
    foreach(m_block_test[ii])
      fork
        m_block_test[ii].system_start();
      join_none; wait fork;
    top_system_start();
    …
  endtask : run_phase

endclass
```

Implementation Steps

F. Randomization of test content – replacing inline constraints

- Ensuring consistent constraint rules in inherited tests without rewriting constraints
- Enabling cross block-level variable control by randomizing all test elements as a whole

- Converting inline constraints to be simple constraint in the test body
- Instantiating test-component array as a rand member and randomizing the test instead of cfg by “this.randomize()”
- Excluding block-level randomization in top mode, and randomizing entire top-level test within its context

Implementation Steps

F. Randomization of test content – replacing inline constraints

Block-level base test

```
class base_test extends template_test;
   base_cfg m_cfg;

   constraint clock_en_c {
      foreach(m_cfg.clock_en[ii]){
         soft m_cfg.clock_en[ii] == 1;
      }
   }

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      <internal phases pre random cfg>
      if(m_cfg.top_level == 0)
         randomize_configuration();
      end
      <internal phases post random cfg>
   endfunction : build_phase

   function void randomize_configuration();
      this.randomize();
   endfunction : randomize_configuration
endclass
```

Top-level base test

```
class top_base_test #(   parameter type RX_TEST_TYPE,
                         parameter type TX_TEST_TYPE, … ,
                         parameter string TEST_NAME = “”) extends uvm_test;

   rand base_test m_block_test[NUM_OF_BLOCKS];

   constraint top_clock_en_c {
      m_block_test[RX].m_cfg.m_clock_en[0] == 0;
      m_block_test[TX].m_cfg.m_clock_en[2] == 0;
   }

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      <internal phases pre random cfg>
      top_randomize_configuration(); // replacing the BL’s randomize_configuration call
      <internal phases post random cfg>
   endfunction : build_phase

   function void top_randomize_configuration();
      this.randomize();
   endfunction : top_randomize_configuration
endclass
```

Implementation Steps

F. Randomization of test content – replacing inline constraints

Block-level base test

```
class base_test extends template_test;
   base_cfg m_cfg;

   constraint clock_en_c {
      foreach(m_cfg.clock_en[ii]){
         soft m_cfg.clock_en[ii] == 1;
      }
   }

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      <internal phases pre random cfg>
      if(m_cfg.top_level == 0)
         randomize_configuration();
      end
      <internal phases post random cfg>
   endfunction : build_phase

   function void randomize_configuration();
      this.randomize();
   endfunction : randomize_configuration
endclass
```

Top-level base test

```
class top_base_test #(   parameter type RX_TEST_TYPE,
                         parameter type TX_TEST_TYPE, … ,
                         parameter string TEST_NAME = “”) extends uvm_test;

   rand base_test m_block_test[NUM_OF_BLOCKS];

   constraint top_clock_en_c {
      m_block_test[RX].m_cfg.m_clock_en[0] == 0;
      m_block_test[TX].m_cfg.m_clock_en[2] == 0;
   }

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      <internal phases pre random cfg>
      top_randomize_configuration(); // replacing the BL’s randomize_configuration call
      <internal phases post random cfg>
   endfunction : build_phase

   function void top_randomize_configuration();
      this.randomize();
   endfunction : top_randomize_configuration
endclass
```

Implementation Steps

F. Randomization of test content – replacing inline constraints

Block-level base test

```
class base_test extends template_test;
   base_cfg m_cfg;

   constraint clock_en_c {
      foreach(m_cfg.clock_en[ii]){
         soft m_cfg.clock_en[ii] == 1;
      }
   }

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      <internal phases pre random cfg>
      if(m_cfg.top_level == 0)
         randomize_configuration();
      end
      <internal phases post random cfg>
   endfunction : build_phase

   function void randomize_configuration();
      this.randomize();
   endfunction : randomize_configuration
endclass
```

Top-level base test

```
class top_base_test #(   parameter type RX_TEST_TYPE,
                         parameter type TX_TEST_TYPE, … ,
                         parameter string TEST_NAME = “”) extends uvm_test;

   rand base_test m_block_test[NUM_OF_BLOCKS];

   constraint top_clock_en_c {
      m_block_test[RX].m_cfg.m_clock_en[0] == 0;
      m_block_test[TX].m_cfg.m_clock_en[2] == 0;
   }

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      <internal phases pre random cfg>
      top_randomize_configuration(); // replacing the BL’s randomize_configuration call
      <internal phases post random cfg>
   endfunction : build_phase

   function void top_randomize_configuration();
      this.randomize();
   endfunction : top_randomize_configuration
endclass
```

Implementation Steps

Top-level tests creation


```
typedef top_base_test#(rx_registers_test,tx_registers_test,proc_registers_test, ,"rx_tx_proc_registers_test") rx_tx_proc_registers_test;
```

```
class top_main_master_test #(  parameter type RX_TEST_TYPE, parameter type TX_TEST_TYPE, parameter type PROC_TEST_TYPE, … ,
                                parameter string TEST_NAME = "")

   virtual function void top_adjust_configuration();  // implemented after adjust_configuration internal phase
      super.top_adjust_configuration();
      foreach(reg_blocks[ii]) begin
         reg_blocks[ii].default_map = reg_blocks[ii].get_map_by_name("PROC_MAIN_MASTER");
         reg_blocks[ii].default_map.set_sequencer(m_block_test[PROC].m_regmodel.get_map_by_name("PROC_MAIN_MASTER").get_sequencer());
      end
   endfunction : top_adjust_configuration
endclass
```

Top-level test package

```
typedef top_main_master_test # (RX_random_test, , PROC_fw_mode_test, ...)  RX_PROC_fw_test;
typedef top_main_master_test # ( , TX_POR_test ,  PROC_POR_test , ...)  TX_PROC_POR_main_master_test;
typedef top_base_test # (RX_POR_test,TX_POR_test,PROC_POR_test, ...)  RX_TX_PROC_POR_test;
...
```

Results

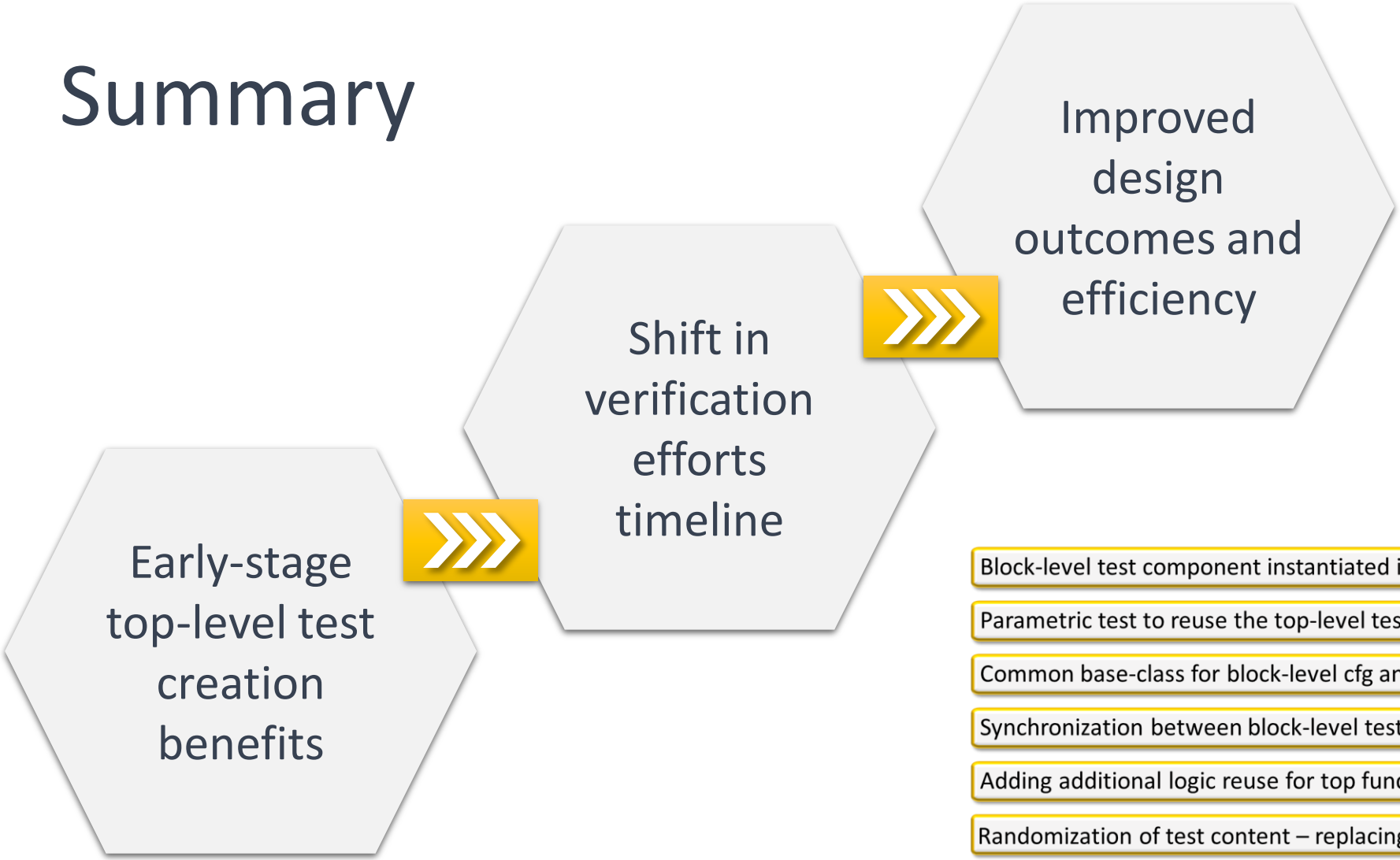
Modular top-level test tree from block-level framework

Highly efficient top-level test creation, often just one line of code

Hierarchical extension for intricate tests and phase integration

Alternative to inline constraints, emphasizing reusability

Summary



- Block-level test component instantiated in the top-level test
- Parametric test to reuse the top-level test code
- Common base-class for block-level cfg and template pattern for test elements
- Synchronization between block-level test components flow
- Adding additional logic reuse for top functions/phases
- Randomization of test content – replacing inline constraints

Questions

Tchiya.dayan@mobileye.com