February 28 – March 1, 2012

Configuring Your Resources the UVM Way!

by

Parag Goel; Amit Sharma;

Rajiv Hasija

Agenda

- Need for Configurability in Test-benches
- Understanding the UVM Configuration Mechanism
- Addressing Verification Requirements Using UVM Resources
- Improving Verification Throughput with Resources
- Verifying RTL configurations
- Summary

Testbench Configuration

Need for configurability

- Multiple configurations of multiple IPs
  - Power modes, memory accesses, optimizations etc.
- Responsiveness to DUT behavior
  - Reconfiguration of testbench attributes
- Horizontal & Vertical reuse requirements
  - Establish signal communication

Configuration Classes – UVM Style!

```
class vc_cfg extends uvm_object;
  rand int number_of_trans;
  rand int mode_of_operatio
  constraint  valid_mode {
    mode_of_operation inside
                        {MAST, SLV };
  }

  `uvm_object_utils_begin(vc_cfg)
    `uvm_field_int(number_of_trans,
                   UVM_PRINT | UVM_COPY)
    `uvm_field_int(mode_of_operation,
                   UVM_PRINT | UVM_COPY)
  `uvm_object_utils_end
  function new(string name = "vc_cfg");
    super.new(name);
  endfunction : new
endclass : vc_cfg
```

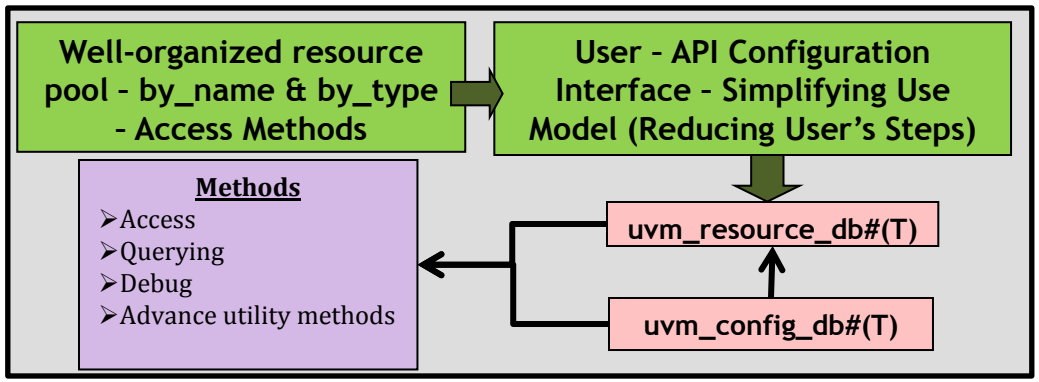
Elements of configuration

Set of valid constraints

Automation macros

UVM Resources and Configuration

- Enable sharing of information across a testbench, i.e. "Universal Configuration"
- Managed through a centralized global database
  - no type restrictions, accessible by name/type
- Rich APIs provided
  - Access : across testbench
  - Querying : user-defined scopes/global
- Provides audit trail for debug
  - What & when?



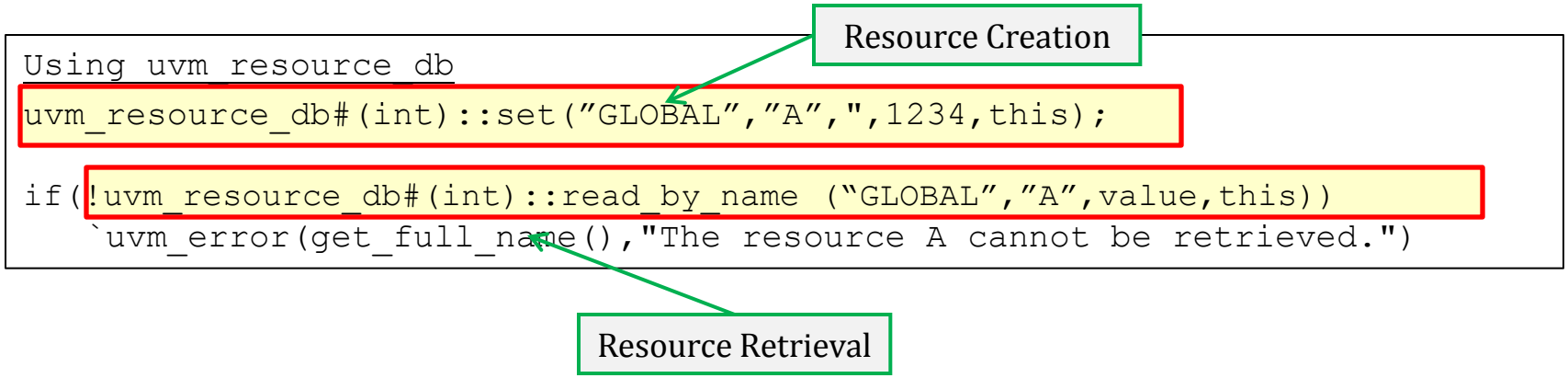
UVM Configuration Mechanism

User API Interfaces

- Resource syntax & Usage

```
uvm_resource_db #(<type>)::set("<scope>","<key>",value,<accessor>);
```

```
uvm_resource_db#(<type>)::read_by_name("<scope>","<key>",value,<accessor>);
uvm_resource_db #(<type>)::read_by_type("<scope>",value, <accessor>);
```



Guideline: Used for general testbench wide resources

UVM Configuration Mechanism

User API Interfaces

- Configuration syntax & Usage

```
uvm_config_db #(<type>)::set(this, "<inst>","<field>", value);
uvm_config_db #(<type>)::get(this, "<inst>","<field>", value);
```

Configuration Handle

```
class my_drv extends uvm_driver;
 drv_cfg cfg;
 `uvm_object_utils_begin(my_drv)
   `uvm_field_object(cfg, UVM_DEFAULT)
 `uvm_object_utils_end
 function void build_phase(uvm_phase. .
   super.build_phase(phase);
   uvm_config_db #(drv_cfg)::get(this, "","cfg", cfg);
   endfunction
endclass
```

Retrieving Configuration Handle

Setting Configuration Handle from agent/env/test case

Hierarchical Context

```
uvm_config_db #(drv_cfg)::set(this,".*drv*", "cfg", my_cfg);
```

Guideline: Used when hierarchical context is important to configuration settings

Propagating Interfaces

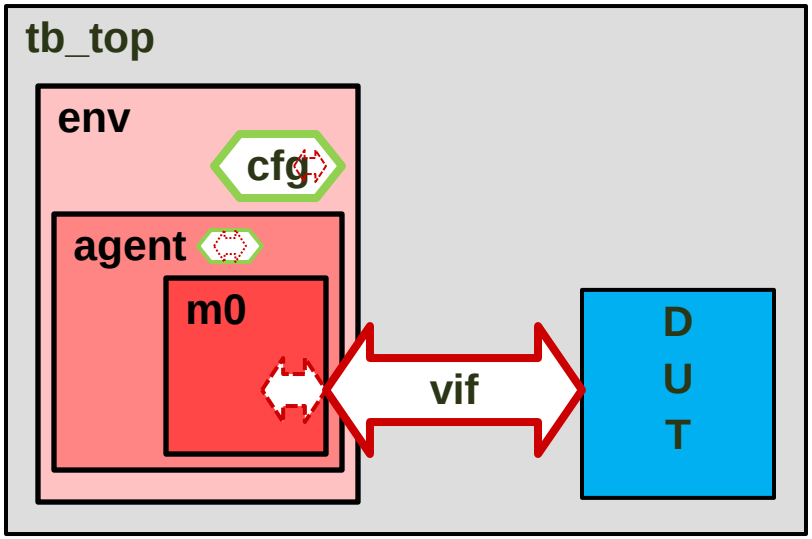
- Virtual interface connection achieved using resources.
  - ✓ Avoids creating object wrappers
  - ✓ Reusable – avoids hierarchical method call/ XMR's
  - ✓ Ease-of-use – avoid package approach

Interface passing across the testbench hierarchy

```
class my_agent extends uvm_agent;
 virtual my_if vif;
 function void build_phase(uvm_phase. .
   if(!uvm_config_db#(virtual my_if)::get
                  (this, "", "vif", vif)))
     `uvm_fatal(. . .)
   uvm_config_db#(virtual my_if)::set
              (this, "drv", "vif", vif);
   endfunction
endclass
```

Interface set via. testbench

```
module tb_top;
 my_if my_if0(...), my_if1(...);
 initial begin
  uvm_config_db#(virtual my_if)::set
    (null, "uvm_test_top.env.agt0",
                    "vif", my_if0);
  uvm_config_db#(virtual my_if)::set
    (uvm_root::get(),"*.env.agt1",
                    "vif", my_if1);
 end endmodule
```



Configurability in stimulus generation and execution

- Creating Reconfigurable Sequences
  - Responds to TB changes
- Executing sequences in different phases
  - Override the 'default_sequence' string in relevant phase

Reference configuration field through parent sequencer

```
class pkt_sequence extends uvm_sequence;
 int item_count = 10;
 task body();
  uvm_config_db#(int)::get(m_sequencer,
            "","item_count",item_count);
  repeat(item_count) `uvm_do(req)
 endtask
endclass
```

Set call from the test case

```
class test_20_items extends uvm_test;
 function void build_phase(uvm_phase. .
  uvm_config_db#(int)::set(this,
   "env.agent.seqr", "item_count", 20);
 endfunction
endclass
```

```
class phase_test extends test_base;
 typedef uvm_config_db
    #(uvm_object_wrapper) seq_phase;
virtual function void build_phase(..
  super.build_phase(phase);
  seq_phase::set(this,
           "env.seqr.reset_phase",
           "default_sequence",
           simple_seq_RST::get_type());
  seq_phase::set(this,
           "env.seqr.main_phase",
           "default_sequence",
           simple_seq_MAIN::get_type());
  endfunction
endclass
```

Setting phase-specific sequences

Improving Verification Throughput

Using the Command Line Manager

- Allows setting UVM variables from command line

```
+uvm_set_config_int=<comp>,<field>,<value>
+uvm_set_config_string=<comp>,<field>,<value>
```

  - Example:

    +uvm_set_config_int=uvm_test_top..drv,delay,10*

  - Requires registration of the field using the FA macros.
- Avoids testcase creation/recompilations for minor changes
- Can be used for sequence control/debug
- Factory Overrides from Command Line

```
+uvm_set_inst_override=<req_type>,<override_type>,<inst_path>
+uvm_set_type_override=<req_type>,<override_type>
```

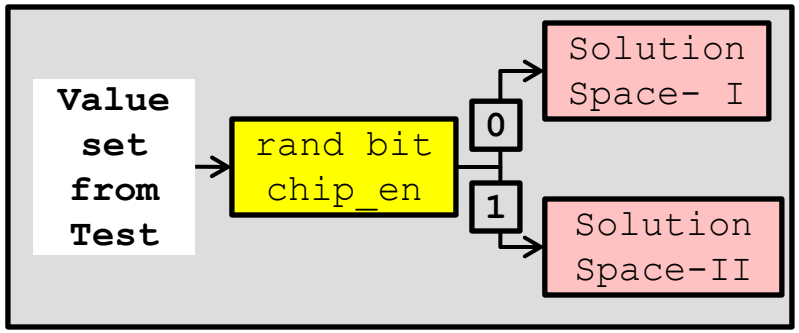
Improving Verification Throughput

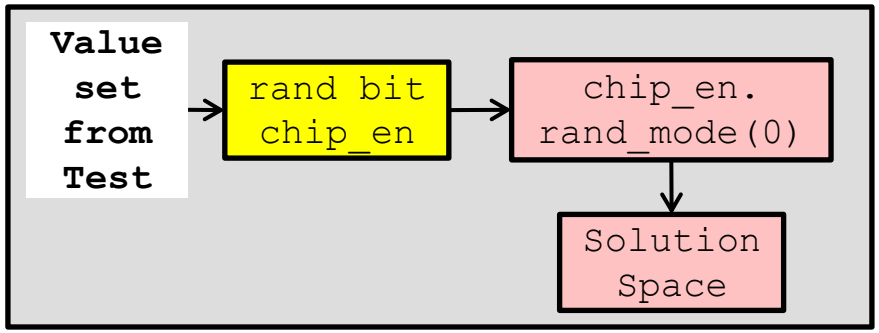
Randomization Control and Performance

- Avoid unwanted variables in class randomization
- Help reduce solution space exponentially for the solver
- Enabled through UVM Resources…
  - Turn off rand_mode if variable overridden through the resource mechanism.

```
if(uvm_resource_db#  int)::read_by_name("*",”A”, value,this))
      A.rand_mode(0);
```

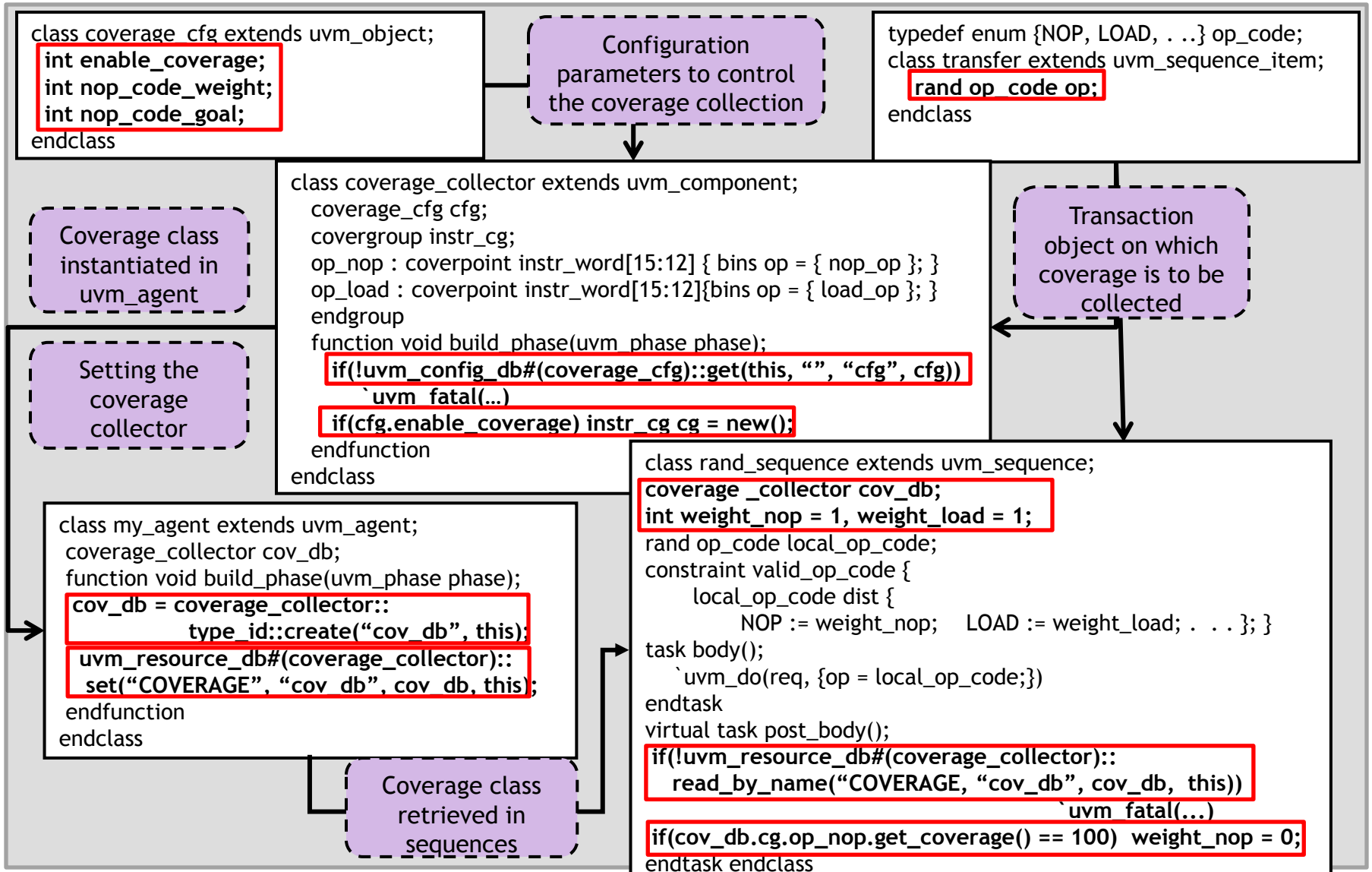
  - Modify constraints
    - Enable ‘solve before’ semantics





Improving Verification Throughput

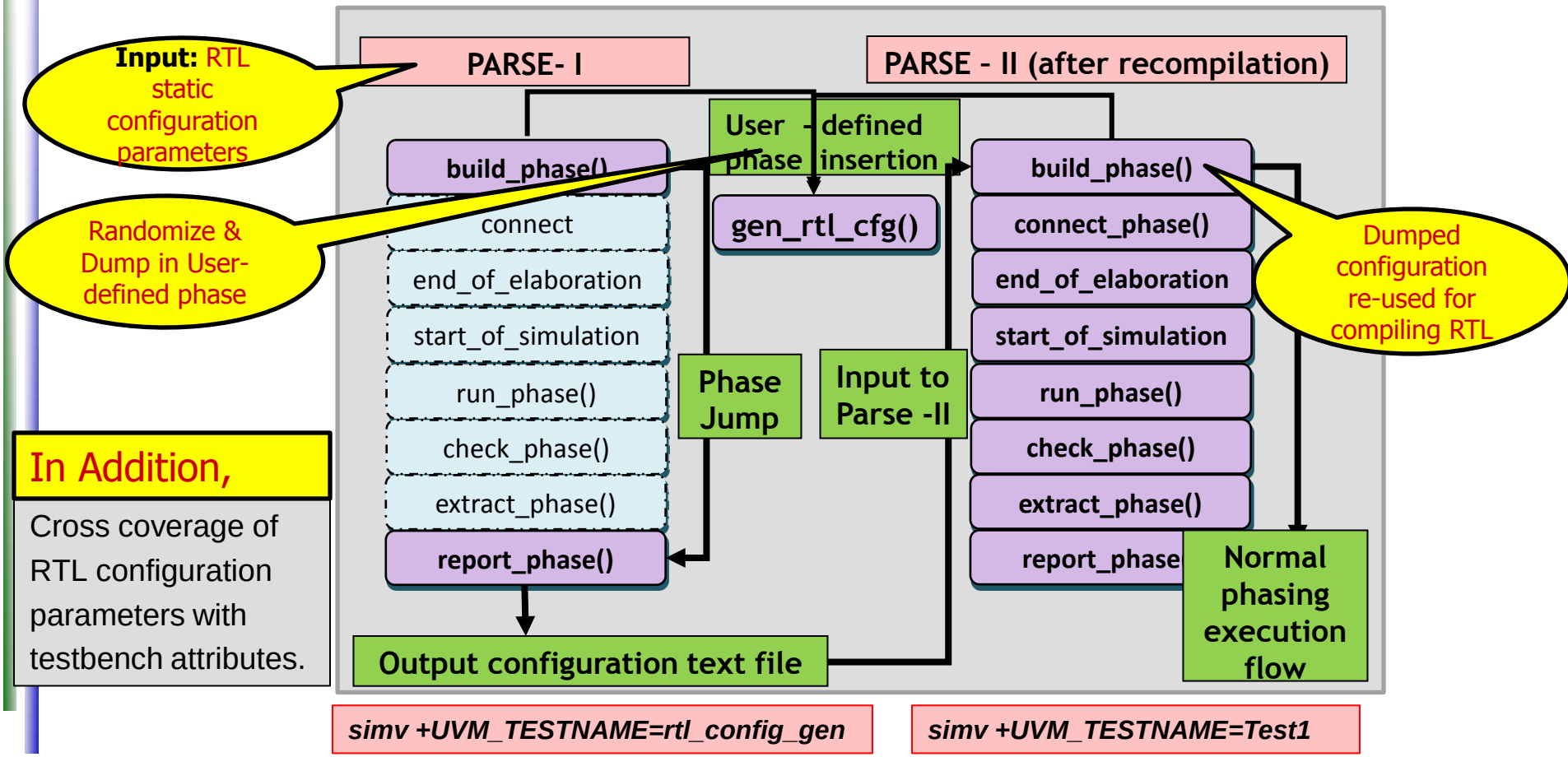
Dynamic Feedback &Coverage Convergence



Verifying RTL Configurations

Challenge : Verification of multiple RTL configurations
(static parameter changes)

Solution : RTL configuration class managed through UVM resources



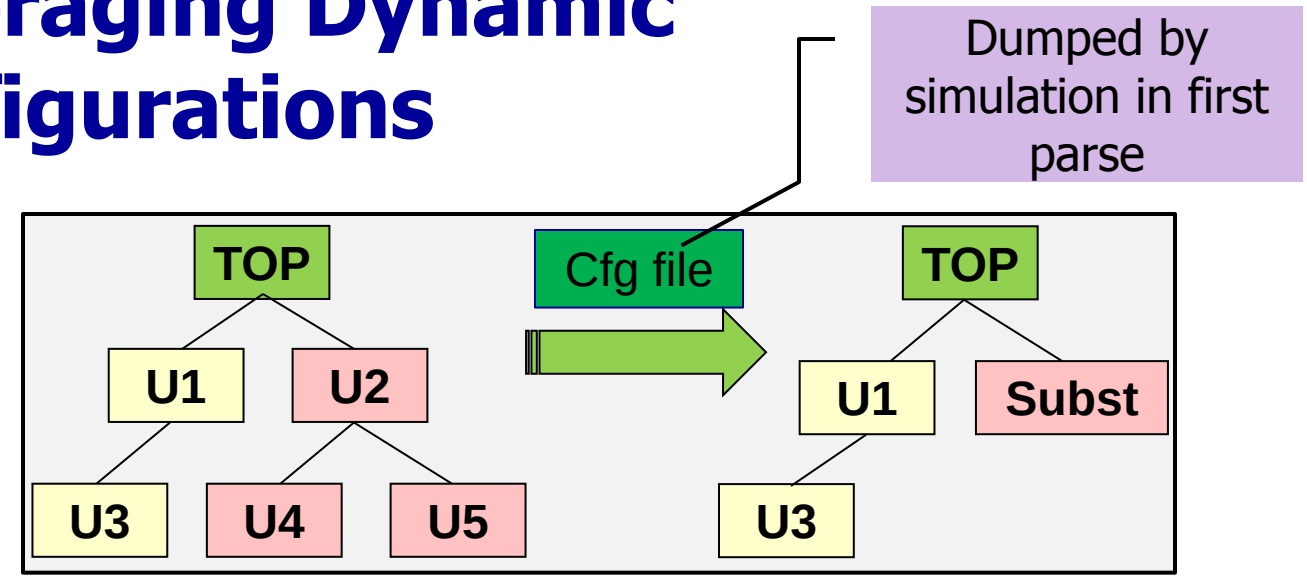
In Addition,

Cross coverage of RTL configuration parameters with testbench attributes.

Leveraging Dynamic reconfiguration

- Verifying a 100 IPs in a SOC?
- Increasingly difficult to accommodate large designs
  - simulation overhead/disk space constraints
  - Not all IPs required for all tests
- Use Dynamic reconfigurations
  - flexible mechanism to replace portions of the design hierarchy at runtime
  - requires passing in **config_file** at compile times
    - specifying hierarchies which can be black boxed
  - run Time **config_file** tells simulator to pick up required hierarchies

Leveraging Dynamic Configurations



- Reuse 2 parse flow
- Dump out of **“runtime configuration”** file in first parse
  - TB aware of the actual design hierarchy
  - TB can generate different permutations through“runtime configuration” file
    - Use SV distribution constraints

Configuration Aware Debug

- Debugging UVM resources:
  - use run-time switch available to trace all config sets and gets

```
+UVM_CONFIG_DB_TRACE, +UVM_RESOURCE_DB_TRACE
```

  - use tracing controls in test code

```
uvm_config_db_options::
turn_tracing_on(), turn_tracing_off() ,is_tracing()
```

  - Dump all resources in the resource pool using,

```
uvm_resources.dump(.audit(1))
uvm_resources.dump_get_records();
```

  - Find unused reources

```
uvm_resources.find_unused_resources();
```

Summary

- Creating Configurable Testbenches is key to tackle today's verification challenges
- Powerful Configuration Mechanism available through UVM
- Leveraged to go beyond regular configuration needs
  - Enhancing simulation throughout
  - Coverage convergence,
  - Verifying configurable RTL
- Configuration Aware Debug made possible

More Powerful and efficient verification Environments through use of UVM Resources!!!