

# Parameterize Like a Pro



## Parameterize Like a Pro

### Handling Parameterized RTL in your UVM Testbench

Jeff Montesano

Paul Marriott



1

Welcome to our Short Workshop, presented at DVCon United States in March 2020.

This short workshop delivers techniques, tricks, skills and insight for engineers facing the challenges of verifying a highly configurable parameterized design using the Universal Verification Methodology (UVM).

#### Who will benefit?

To get the most out of this workshop, you should have sound basic knowledge of SystemVerilog and the UVM, and experience of their use on at least one verification project. It's tailored for verification engineers and technical leads who are, or soon will be, working on a project where the RTL device under test (DUT) is highly configurable (parameterized). Such configurability is commonly found in generic design IP such as bus fabric, memory controllers and high-speed interfaces.

Although the focus is on the UVM, many of the ideas presented will also be of value to those using other verification techniques such as formal property verification.

This tutorial workshop brings together a collection of powerful techniques and good practice learnt through experience on many large-scale projects. Drawing on hard-earned wisdom of Verilab's experienced team, highlights of published papers, and new material created especially for the event, the workshop's recommendations, guidance and examples aim to be immediately useful as well as having continuing value in your future work.

#### Additional materials

Supporting this presentation, a curated and annotated collection of papers written by various Verilab consultants is available at <https://www.verilab.com/resources>

# Agenda

- Introduction
- Connecting to Parameterized RTL
- Extracting RTL Parameters
- Tuning Testbenches with RTL Parameters
- RTL Parameter Generation & Coverage
- Parameter Selection Optimization
- Other Topics
- Conclusion

## Agenda

The range of topics we have chosen to discuss are all very practical in nature and are based on experience from the Verilab consultant community. Parameters are powerful things, and anytime code would otherwise need to be manually edited, or “`ifdef MACRO”d like crazy, thankfully we can say “there is a better way forward”, and we will be happy to share that better way with you.



## Introduction

- In SystemVerilog parameters enable flexibility
  - Compile-time specialization of the code-base
    - RTL module with variable FIFO depth, bus width, instance count, etc.
    - Verification component with variable channels, etc.
- Parameterization enables horizontal reuse
  - Same module RTL code in different project derivatives
    - Automatic address space allocation based on number of masters, slaves
  - Same verification code adapting to different projects
- Parameterized code trades flexibility with complexity



RTL designers like to use parameters, and with good reason. There is really no downside to using them in an RTL design context – everything works seamlessly.

# Introduction

- Verifying highly parameterized RTL designs is a major challenge
  - Verification planning
  - Regression runs and analysis
  - UVM testbench code

Verification's relationship with RTL parameters has never been easy. Before knowing these techniques, I would sometimes recreate a "verification version" of the RTL parameters in a file and compile it separately from the design code – very error prone and involving manual work. Over time better and better ideas have emerged with respect to the handling RTL parameters by verifiers, and thankfully so.

# Introduction

- Parameter "ripple effect" is a major problem in the class world

```
x_env #(parameter int X_BUS_WIDTH=10) extends uvm_env;
  param_cov_wrapper #(.X_BUS_WIDTH(X_BUS_WIDTH)) cov_wrapper;
  x_env_sequencer    #(.X_BUS_WIDTH(X_BUS_WIDTH)) sequencer;
  if_agent          #(.X_BUS_WIDTH(X_BUS_WIDTH)) agent;

  `uvm_component_param_utils(x_env#(X_BUS_WIDTH))

function void build_phase(uvm_phase phase);
  sequencer = x_env_sequencer#(X_BUS_WIDTH)::type_id::create...
  cov_wrapper = param_cov_wrapper#(X_BUS_WIDTH) ::type_id::create...
```

Verbose and error-prone code

```
x_env_sequencer #(parameter int X_BUS_WIDTH) extends uvm_sequencer;

if_agent #(parameter int X_BUS_WIDTH) extends uvm_agent;
  x_monitor #(.X_BUS_WIDTH(X_BUS_WIDTH)) monitor;
  x_driver  #(.X_BUS_WIDTH(X_BUS_WIDTH)) driver;

x_monitor #(parameter int X_BUS_WIDTH) extends uvm_monitor;
  x_driver #(parameter int X_BUS_WIDTH) extends uvm_driver;
```

Imagine when there are many parameters!

This slide illustrates why we want to avoid parameterized classes in our UVM environment. It seems natural on the surface to want to create a parameterized UVM environment to echo a parameterized RTL module, but there are many reasons to avoid doing this. I have worked on many successful UVM projects over the years and have never needed to resort to parameterized classes. If you find yourself in a situation where you think you need them, definitely make an effort to find another way. In my experience there always is one. Each parameterization creates a new type, and this becomes very tedious to deal with as the number of parameters increases.

# Introduction

- Techniques presented today will allow you to:
  - Avoid the “parameter ripple” issue
  - Keep TB in perfect sync with RTL parameters
  - Automatically adjust TB to different RTL parameterizations
  - Make more effective use of limited verification time
- All techniques have been ‘battle tested’ in real projects

In this presentation we aim to provide solutions that will be clean (no parameter ripple), robust (no possible RTL/TB differences in parameterization value), hands-off (TB automatically adjusts itself to RTL parameters), and efficient. The techniques demonstrated have been developed and deployed on real projects, so we have high confidence in their applicability.

# Connecting to Parameterized RTL



## Traditional DUT to TB Connection

```
module tb;
  b_if_max_type b_if_1();
  b #(...) //p1 parameterization
    b_inst_1(.data(b_if_1.data[7:0]), ...);
  ...
  b_if_max_type b_if_2();
  b #(...) //p2 parameterization
    b_inst_2(.data(b_if_2.data[11:0]), ...);
  ...
  initial begin
    uvm_config_db#(virtual b_if_max_type)::set(null,
      "*.b1_agent", "b_vif", b_if_1);

    uvm_config_db#(virtual b_if_max_type)::set(null,
      "*.b2_agent", "b_vif", b_if_2);
  end
endmodule
```

Each RTL parameterization connects to max-size interface

Works ok - But we can do better...

Publish to config\_db

```
class b_agent extends uvm_agent;
  virtual b_if_max_type vif;
  function void build_phase(...);
    uvm_config_db#(virtual b_if_max_type)::get(this, "", "b_vif", vif);
  ...
endclass
```

Get from config\_db; avoids parameterized classes

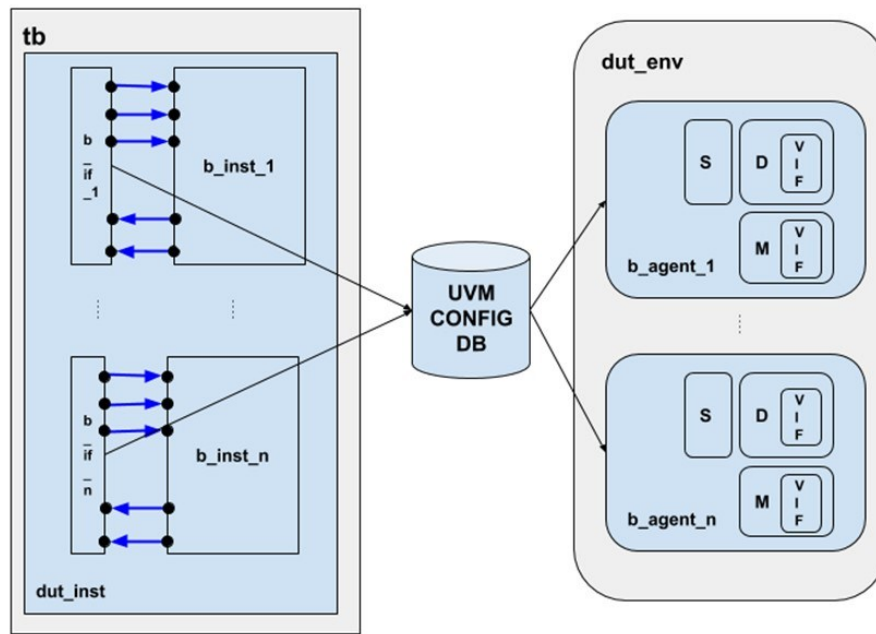


9

## Traditional DUT to TB Connection

This example shows a single RTL module with two different bus-width parameterizations; because we want to avoid parameterized classes (a parameterized uvm\_agent, to be specific to this example), we select the “max-bus-width” virtual interface to communicate between the class-based and module-based worlds. (Note: Parameterized classes in a UVM testbench are problematic because of all the code necessary to maintain them, and how error prone that code is. – as shown in the introduction) This is a fine technique – way better than having parameterized classes to deal with the different bus-widths. There is some pain in having to specify what part of the max-width interface connects to the actual RTL bus (which is usually smaller than max-width). Also, this example is a bit contrived for illustrative purposes - usually a tb module will instantiate just one RTL module as a DUT, and that DUT will in turn instantiate multiple instances of a particular RTL block. We use the multiple instances directly in the tb module to make our point as clearly as possible.

# Traditional DUT to TB Connection



## Traditional DUT to TB Connection

This is what it looks like visually when virtual interfaces are published to the config\_db and UVM agents get those virtual interfaces. Recall that the interfaces are defined as max-width interfaces.



# UVM Harness

- What is the UVM Harness?
  - A technique to connect SV/UVM TB and RTL module
- Scalable solution for DUTs in which the number of instances of an RTL module is parameterized
- Handles bus-width parameterized RTL
- Avoids messy assign statements and wires in testbench module file
- Encapsulates SV interface-RTL module connectivity in a reusable file
- Low maintenance from a code perspective
- Error-proof from a functionality perspective

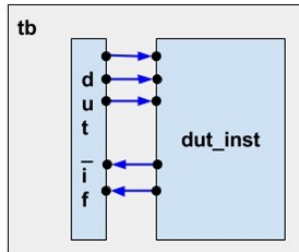
## UVM Harness

The UVM Harness technique was first introduced to the presenters in 2011 in a conference paper written by David Larson (“UVM Harness Whitepaper: The missing link in interface connectivity”), and it has proven to be very useful on several projects since then. It’s the kind of technique that you didn’t know you were missing until you see it, as connecting RTL modules and TB just seems rudimentary and a necessary evil. But this technique is a huge improvement over the old way of doing things and anyone who grasps it will likely never do it the old way ever again.

# UVM Harness: Interface Placement

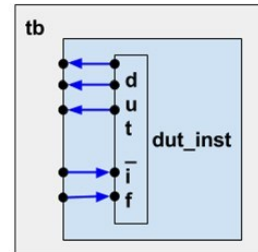
## Traditional placement

- Interface is outside design module



## UVM Harness placement

- Interface is inside design module

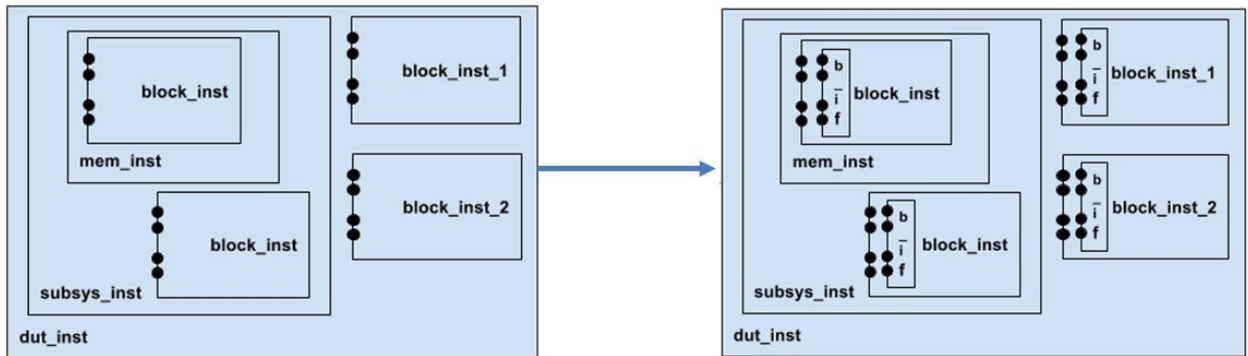


## UVM Harness: Interface Placement

The UVM harness technique for connecting the RTL module and the UVM world requires a few 'paradigm shift' ideas. The first one being that the virtual interface is inside the RTL module rather than outside it.

# UVM Harness: Interface Inside DUT

- SystemVerilog bind directive can amend a module definition
  - Syntax: **bind** <module\_type> <interface\_type> <interface\_name>;
- Amending module definition naturally affects all instances



## UVM Harness: Interface Inside DUT

The next 'paradigm shift' idea is to remember that the SystemVerilog bind statement can do more than copy-paste code into an instance of a module: it can change the module definition itself, so that all instances are always affected. Here we show how a particular syntax of the bind statement results in an interface being placed inside a particular module type by amending that module's definition to always have the interface. This is a powerful technique when a design has multiple instances of a particular module, because so much gets accomplished in an error-resilient manner with very little code.

# UVM Harness Step-by-Step

- Step 1: Define an **RTL module interface** with input ports of type "wire" and matching parameters

```
module #(int IN_WIDTH = 6, int OUT_WIDTH = 8)
  b_type(input logic clk,
         input logic rst,
         input logic data_in[IN_WIDTH-1:0],
         output logic data_out[OUT_WIDTH-1:0]);
endmodule
```

```
interface #(int IN_WIDTH = 6, int OUT_WIDTH = 8)
  b_if_type (input clk,
             input rst,
             input data_in[IN_WIDTH-1:0],
             input data_out[OUT_WIDTH-1:0]);
endinterface
```



Interface is all inputs,  
uses SV port coercion

Port coercion: IEEE 1800-2017  
(SystemVerilog LRM) section  
23.3.3.1

## UVM Harness Step-by-Step

We now present the steps involved in the UVM harness technique. Step 1 is to create an RTL module interface where all of the interface ports are of type wire (in the example we leave out the type, as it defaults to wire – alternatively we would have wrote “input wire clk, etc.”), and matching parameters. Of note is that the RTL module interface has all ports defined as “input”, even if their corresponding RTL module port is output. This is because we are relying on a little-known SystemVerilog technique called port coercion, whereby the simulator will coerce the direction of the port to the proper direction. For example, should there be a driver to that port, the compiler will know to treat it as an output; if there’s no driver, it will remain an input. This is extremely powerful when it comes to reusing interfaces in highly configurable testbenches (as detailed in [1] and [3] – see references). Also note that port-coercion is in the SV LRM and therefore is supported by all simulators. Also, the question sometimes gets raised “instead of doing this port-coercion stuff, why not use inout ports?” The answer is that inout require port widths to be identical when making connections, and so that doesn’t work in this max-width technique.

# UVM Harness Step-by-Step

- Step 2: Define a **max-width interface** with ports of type "wire"

```
interface
  b_max_if_type (input clk,
                  input rst,
                  input data_in [`MAX_IN_WIDTH-1:0],
                  input data_out[`MAX_OUT_WIDTH-1:0]);
endinterface
```



## UVM Harness Step-by-Step

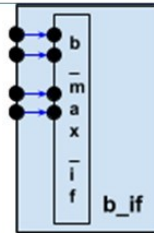
The max-width interface concept is still applied in the UVM harness technique, because we still want to avoid having parameterized classes. A max-width virtual interface will allow for connection to any instance of the RTL module.

# UVM Harness Step-by-Step

- Step 3: Instantiate and connect max-width interface inside RTL module interface

```
interface #(int IN_WIDTH = 6, int OUT_WIDTH = 8)
    b_if_type (input clk,
               input rst,
               input data_in [IN_WIDTH-1:0],
               input data_out[OUT_WIDTH-1:0]);

    // Step 3
    b_max_if_type b_max_if(.*) ;
    ...
endinterface: b_if_type
```



DUTs with ports of type  
"logic" are problematic  
for port coercion

Solution: 2-levels of  
interface with ports of  
type "wire"

## UVM Harness Step-by-Step

This is the newest addition to our understanding of how to best do the UVM Harness technique. We instantiate the max-width interface inside a non-max-width RTL module interface... this is counterintuitive – a larger thing shouldn't be able to fit inside a smaller thing! But in software it all works. The reason for not just making the RTL module interface itself max-width is to immunize ourselves against an RTL module that has ports of a type other than wire (e.g. logic). It turns out that for port-coercion to work, there needs to be a wire-to-wire connection. By placing the max-width interface with ports of type wire inside the RTL module interface (also with ports of type wire), the port-coercion will work regardless of the types on the ports of the actual RTL module. In the past I have asked RTL designers to change their port types from logic to wire, to accommodate this technique. No longer will I need to do this.

# UVM Harness Step-by-Step

- Step 4: Tie off unused bits of max-width interface with weak tie-off

```
interface #(int IN_WIDTH = 6, int OUT_WIDTH = 8)
  b_if_type (input clk,
             input rst,
             input data_in [IN_WIDTH-1:0],
             input data_out [OUT_WIDTH-1:0]);

  // Step 3
  b_max_if_type b_max_if(.);

  // Step 4
  if(IN_WIDTH < `MAX_IN_WIDTH) begin: unused_data_in_pull-down
    assign (pull1,pull0) b_max_if.data_in[`MAX_IN_WIDTH-1:IN_WIDTH] = '0;
  end
  if(OUT_WIDTH < `MAX_OUT_WIDTH) begin: unused_data_out_pull-down
    assign (pull1,pull0) b_max_if.data_out[`MAX_OUT_WIDTH-1:OUT_WIDTH] = '0;
  end
endinterface: b_if_type
```

## UVM Harness Step-by-Step

Here we tie off the unused bits of the max-width interface. We contemplated what would happen if we just assigned the entire bus to a weak pulldown rather than being precise about the bits to affect – after all if the pulldown is really “weak”, it should be overridden when a strong driver is connected to it. After some thought, it turns out that this is bad verification practice: As a verifier, you don't want to tie-off any bits that the RTL module is really responsible for: If the RTL module doesn't drive a net it's meant to drive, you don't want the testbench silently cleaning up the mess through a testbench's pullup/pulldown.

# UVM Harness Step-by-Step

- Step 5: Create parameterized harness module and Instantiate RTL module interface inside

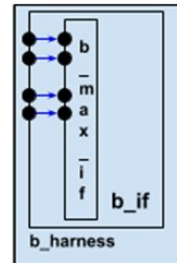
```

module b_harness (filename b_harness.sv)
  #(IN_WIDTH  = `MAX_IN_WIDTH,
    OUT_WIDTH = `MAX_OUT_WIDTH) ();

  b_if_type #(.IN_WIDTH(IN_WIDTH), .OUT_WIDTH(OUT_WIDTH))
    b_if(.clk      (      ),
         .rst      (      ),
         .data_in  (      ),
         .data_out (      )
        );
endmodule

```

Default of max-width;  
bind statement will right-size them  
for each RTL module instance later



## UVM Harness Step-by-Step

This is also a change from our original understanding of the UVM harness: we now believe a module is a better type for the harness than an interface is. A module is more versatile because it can instantiate other modules. An interface cannot. Some projects might want to reuse existing assertion modules inside their harness, and making the harness a module will allow that. Some would argue “an interface can be virtual, a module cannot” – which is true. However, we were unable to come up with any compelling use cases for a virtual harness in the class world.



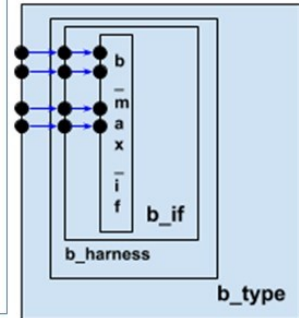
# UVM Harness Step-by-Step

- Step 6: Connect RTL module interface ports to RTL module using upward reference to module type

Hierarchical references: IEEE 1800-2017 (SystemVerilog LRM) sections 23.6 – 23.8

```
module b_harness
  #(IN_WIDTH  = `MAX_IN_WIDTH,
    OUT_WIDTH = `MAX_OUT_WIDTH) ();

  b_if_type #(.IN_WIDTH(IN_WIDTH), .OUT_WIDTH(OUT_WIDTH))
    b_if(.clk      (b_type.clk ),
         .rst      (b_type.rst ),
         .data_in  (b_type.data_in ),
         .data_out (b_type.data_out )
        );
endmodule
```



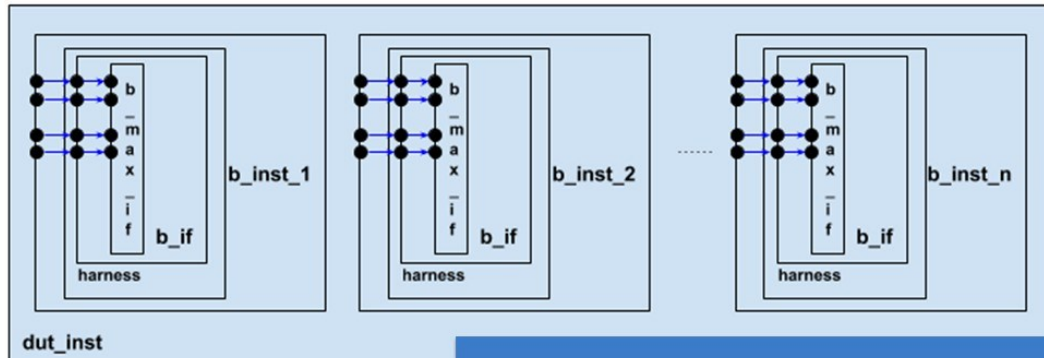
## UVM Harness Step-by-Step

This step is another ‘paradigm shift’ one. The interface ports are being connected to `module_type.port_name` of the RTL module, as opposed to `module_instance.port_name`. The fact that SystemVerilog supports such syntax was a surprise to me, and it turns out to be powerful. What this is saying is, “anywhere this type of module is visible within the scope of this harness, connect the ports of that module to the ports of this interface”. This, combined with the “module-amending” use of `bind`, become a super-powerful technique for RTL-TB connections, as we’ll see in the next slides.

# UVM Harness Step-by-Step

- Step 7: Bind harness to RTL module definition

```
// bind <module_type> <interface_type> #(parameter mapping) <interface_name>;
bind b b_harness #(.IN_WIDTH(IN_WIDTH), .OUT_WIDTH(OUT_WIDTH)) harness();
```



Each harness takes parameters from each DUT instance:  
The interface bus widths 'auto-size' themselves

## UVM Harness Step-by-Step

Here's the power of the technique – with one line of code, we go from having an RTL DUT, to a fully-connected DUT-TB for all instances of a particular module; no possibility of missing a module, no possibility that one module instance is connected differently than another module instance; complete certainty that there will be no further TB bugs due to connections.

# UVM Harness Step-by-Step

- Step 8: Add a `set_vif()` function to harness

```
module b_harness(...);  
  b_if_type b_if(...);  
  function void set_vif(string path);  
    uvm_config_db#(virtual b_max_if_type)::set(null, path, "b_vif", b_if.b_max_if);  
  endfunction
```

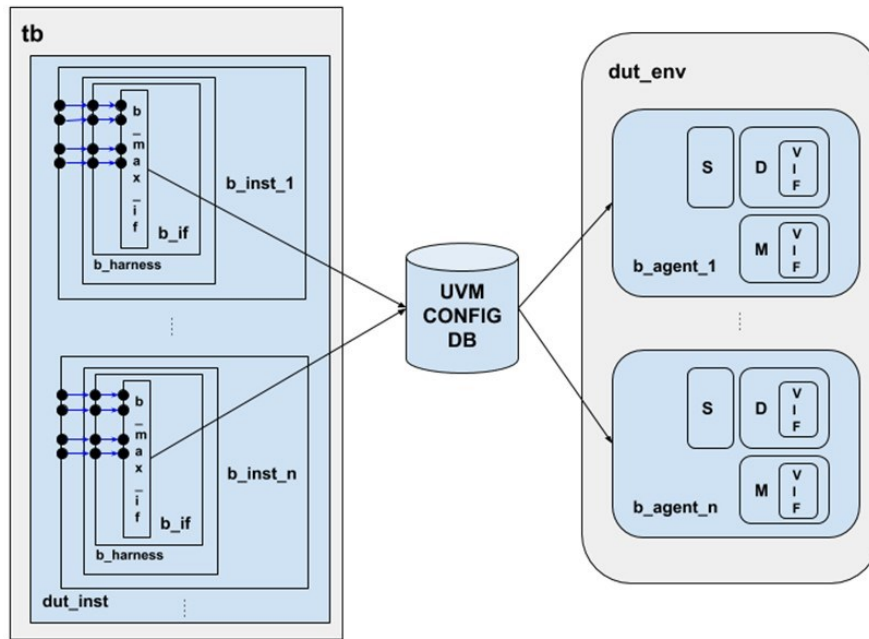
- Step 9: Call `set_vif()` from top testbench module

```
module tb;  
  dut dut_inst();  
  ...  
  initial begin  
    dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.driver");  
    dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.monitor");  
    ...  
    dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.driver");  
    dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.monitor");  
    run_test();  
  end
```

## UVM Harness Step-by-Step

We need to get the virtual interfaces into the UVM `config_db`. Adding a “`set_vif`” function in the harness, that takes a variable string for path, allows the virtual interface to be set wherever it is needed in the verification environment. It is important to call `set_vif` before calling `run_test`, to ensure that the virtual interfaces are in the `config_db` before the environment gets built. Having them both in the same initial block assures that this is the case.

# UVM Harness: Result



## UVM Harness: Result

What it looks like visually when virtual interfaces are published to the config\_db and UVM agents get those virtual interfaces, using the UVM harness technique. Notice how this is similar to the picture in the traditional approach except that we have done something a lot more powerful, using a lot less code.



## Extracting RTL Parameters

- Automatically extract RTL parameters for testbench use
  - Avoids proliferation of ``ifdef` macros in testbench code
  - Simplifies merging of functional coverage over different parameter sets
- Use extracted RTL parameters to make testbench flexible
  - Less DV engineer time/effort
  - More readable testbench
- Technique:
  - Use harness module to collect RTL parameters
  - Use UVM config DB to publish collected parameters



Wouldn't it be nice for the testbench to be able to make use of RTL parameters in a robust and tidy fashion? I've struggled in the past with non-robust and ugly ways of accomplishing this, but was very pleased to learn about this approach, which I now use religiously. The idea is to bind an interface (or module – let's stick with interface for the sake of readability) to an RTL module, capture the values of the parameters inside that interface (since it has module scope after the bind) and then publish the parameter values to the config\_db so that the class-world can make use of them. I like to think of this as a kind of trojan-horse technique, with the bound interface being the horse, and instead of releasing soldiers (parameters) it is capturing them to send off to far away places.

# Extracting RTL Parameters

```
module dut_harness #(
    PARAM1 = 8,
    ...
    PARAMN = 1)
) ();

    rtl_info_struct_t  rtl_info;

function void capture_rtl_info(string path);
    rtl_info.dut_param1 = PARAM1;
    rtl_info.dut_param2 = PARAM2;
    rtl_info.dut_paramN = PARAMN;
    uvm_config_db#(rtl_info_struct_t)::set(null,{path, "env.*"},
                                             "rtl_info_struct", rtl_info);
endfunction

endmodule
```

Corresponding names of  
DUT's RTL parameters

Struct reflecting RTL  
parameter values

Method capturing values  
into struct and put into  
config\_db

## Extracting RTL Parameters

We start by defining a harness that has matching parameters as the RTL module we wish to extract from. Next, we define a data structure (of type struct) that has fields which correspond to each RTL parameter. Finally we define a method that can be called by a bound harness to capture the values of the RTL parameters in which it is bound, and publish them to the UVM config\_db.

# Extracting RTL Parameters

- Interface bound to the DUT via the SystemVerilog bind statement

```
bind dut dut_harness #(
    .PARAM1 (DUT_PARAM1),
    .PARAM2 (DUT_PARAM2),
    [...]
    harness_inst (.*)
```

Bind statement will capture  
DUT RTL parameters

Bind statement can go anywhere  
(e.g. dut\_harness.sv)

Results in:

```
module dut #(DUT_PARAM1 = 4, ...) (...)
    dut_harness
    #(.PARAM1 (DUT_PARAM1)
      .PARAM2 (DUT_PARAM2)
      .PARAM3 (DUT_PARAM3))
    harness_inst (
        ...
    endmodule
```

```
module tb_top;
    dut dut_inst
    # (DUT_PARAM1 ( 2 ),
      DUT_PARAM2 ( 0 ),
      DUT_PARAM3 ( 8 )
      ...
    );
```

## Extracting RTL Parameters

The bind statement creates an instance of “dut\_harness” inside the DUT module, and therefore the harness’ parameter values are equal to the DUT parameters after that bind statement executes. This is because the bind statement says “connect DUT\_PARAM1 to the PARAM1 parameter of the harness”, which is a way of assigning DUT\_PARAM1 to PARAM1; after the bind, DUT\_PARAM1 is the value of the RTL parameter, and so PARAM1 gets the RTL parameter value. Tricky the first time you see it.

# Extracting RTL Parameters

- Capture method call made before testbench is created

```
module tb_top;
  initial
  begin
    dut.harness_inst.capture_rtl_info("*.env");
    run_test();
  end
endmodule
```

## Extracting RTL Parameters

Before creating the UVM environment (via `run_test`), the RTL parameters are captured (via the `capture_rtl_info` call) and published to the UVM environment (in this case the set is to `*.tb_env`). Again it's important to keep the call to `capture_rtl_info` in the same initial block as `run_test`, to ensure that the `config_db` has the parameter values when the `build_phase` of the verification environment executes.



# Extracting RTL Parameters

```
// file b_harness.sv
module b_harness #(IN_WIDTH = `MAX_IN_WIDTH, ..., PARAM1 = 8, ... ) ();

    b_if_type #(.IN_WIDTH(IN_WIDTH), .OUT_WIDTH(OUT_WIDTH))
        b_if(.clk      (b_type.clk      ),
            .rst      (b_type.rst      ),
            .data_in   (b_type.data_in  ),
            .data_out  (b_type.data_out )
        );

    rtl_info_struct_t rtl_info;
    function void capture_rtl_info(string path);
        rtl_info.dut_param1 = PARAM1;
        ...
    endfunction
endmodule

bind dut dut_harness #( .IN_WIDTH(DUT_IN_WIDTH), ..., .PARAM1(DUT_PARAM_1))
    harness_inst(.*)
```

Harness file combining TB hookup  
and RTL parameter extraction

## Extracting RTL Parameters

Here's the final result of what a harness file looks like with interface connection, RTL parameter extraction, and bind statement.

# Extracting RTL Parameters

TB module combining TB hookup  
and RTL parameter extraction

```
module tb;
  dut dut_inst();
  ...
  initial begin
    dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.driver");
    dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.monitor");
    ...
    dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.driver");
    dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.monitor");
    dut.harness_inst.capture_rtl_info("*.env");
    run_test();
  end
end
```

## Extracting RTL Parameters

And what the testbench module file looks like with harnesses publishing virtual interfaces and the RTL parameters being captured and published – all before run\_test creates the UVM environment. Notice how clean this is compared to the “pollution” we normally see in a testbench module file.

# Tuning Testbenches with RTL Parameters



## Tuning Testbenches with RTL Parameters

- Classes use RTL parameters from the config\_db for randomization

```
class config_obj extends uvm_object;
  `uvm_component_utils(config_obj)

  rtl_info_struct_t dut_rtl_param_info;
  rand int          port_number;

  constraint port_num_constr {
    (port_number >= 0) && (port_number < dut_rtl_param_info.number_of_ports);
  }

  function new(string name);
    if (!uvm_config_db#(rtl_info_struct_t)::get(null, "",
        "rtl_info_struct", dut_rtl_param_info))
      `uvm_fatal(get_name(), "Failed to get ...")
  endfunction
endclass
```

Use RTL parameter for  
randomization constraint

Retrieve rtl\_info\_struct  
reflecting captured RTL  
parameter values



32

Because the extracted RTL parameters are available before the UVM environment is constructed, constraint blocks can safely make use of the RTL parameters to guide randomization anywhere in the UVM environment. In this instance the RTL parameter specifies the number of ports present in the DUT, and the randomization limits itself to a port number inside that valid range. Another situation where I have seen this is to randomize values to be programmed into DUT registers (via a random configuration object), where the RTL parameters specify the bus widths of the register fields.

# Tuning Testbenches with RTL Parameters

- Classes use RTL parameters from config\_db for coverage

```
class param_cov_wrapper extends uvm_component;
  covergroup x_bus_vals_cg (int BUS_WIDTH);
    coverpoint x_bus {
      bins all_values[] = {0:2**BUS_WIDTH};
    }
  endgroup
  rtl_info_struct_t dut_rtl_param_info;
  ...
  function new(...
    if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
      "rtl_info_struct", dut_rtl_param_info))
      `uvm_fatal(get_name(), "Failed to get ...")

    x_bus_vals_cg = new(.BUS_WIDTH(dut_rtl_param_info.BUS_WIDTH));
```

SV covergroup constructor  
can be passed arguments

Arguments can tune  
coverage bins

RTL parameters are known  
at the time of all UVM  
class construction

## Tuning Testbenches with RTL Parameters

Covergroups can be passed arguments at the time of their creation, and these arguments can tune coverage bins. Fortunately, the RTL parameter extraction technique shown earlier makes the values available before the UVM environment is built, so any class can make use of them in their constructor. Note that this isn't usually the case when it comes to tuning coverage: if you wanted to tune coverage bins to a configuration object that is randomized within the UVM environment, that configuration object would not be ready in time for the constructor of the coverage class (1800-2017 section 19.4 states that "An embedded covergroup variable may only be assigned in the new method."). Solutions do exist for this situation and are described in [5] and [6].

# Tuning Testbenches with RTL Parameters

- SVA can use RTL parameters from the config\_db

```
interface dut_if_assertions(input clk, ...)
  int rtl_param_max_value;
  class checker_phaser extends uvm_component
    function void end_of_elaboration_phase(...)
      if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
        "rtl_info_struct", dut_rtl_param_info))
        `uvm_fatal(get_name(), "Failed to get ...")
        rtl_param_max_value = dut_rtl_param_info.max_value;
    endfunction
  endclass
  checker_phaser m_phase = new("checker_phaser");

  property p_check_data_range;
    @ (posedge clk)
    data_in < rtl_param_max_value;
  endproperty
  a_check_data_range: assert property(p_check_data_range);
endinterface
```

Redeclare local variable  
for RTL parameter

Embed class to get RTL  
parameters from  
config\_db

Use parameter value in  
SVA

## Tuning Testbenches with RTL Parameters

There are different approaches available to use RTL parameters in assertions. One simple way would be to parameterize the encapsulating module/interface and just pass the RTL parameter values that way. Here we show a more complex method that was adapted from a technique to use UVM configuration values in SVA. Both techniques are valid, though the one presented here is slightly less error-prone because it uses the extract RTL parameters as opposed to relying on manually entering their value upon instantiation of the interface/module containing the SVA (though defining the parameters as `define macros would alleviate this issue too).

You can find more on this approach in Mark Litterick's Advanced UVM Tutorial from DVCon-Europe 2015 (see references at the end of this presentation).

# Tuning Testbenches with RTL Parameters

- RAL uses RTL parameters for register reset values

```
rtl_info_struct_t dut_rtl_param_info;  
uvm_reg_fields    dut_reg_fields[$];  
[...]  
if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",  
                                             "rtl_info_struct", dut_rtl_param_info))  
    `uvm_fatal(...)  
  
dut_regblock.featureA_reg.field1.set_reset(  
    dut_rtl_param_info.mydut_featA_regfield1_resetval);
```

Get RTL parameter struct  
as usual via config db

Struct contains register  
reset value to reflect  
RTL value

## Tuning Testbenches with RTL Parameters

The reset value of a register can be specified in an RTL parameter. The slide shows how a UVM RAL implementation would make use of that parameter when defining the reset value of that field (via `reg_field.set_reset()`). A user of that register would later apply the reset to that field (perhaps in response to detecting a hard reset in the system) by calling `reg_field.reset()`, and get the correct value according to the current RTL parameterization.

# Tuning Testbenches with RTL Parameters

- RAL uses RTL parameters for register inclusion/removal

```
if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
                                           "rtl_info_struct", dut_rtl_param_info))
    uvm_fatal(...)

if (dut_rtl_param_info.mydut_featureA_enable == 1'b0) begin
    dut_regblock.myreg_featureA_reg1.get_fields(dut_reg_fields);
    foreach(dut_reg_fields[r]) begin
        dut_reg_fields[r].set_access("RO");
        dut_reg_fields[r].set_reset(0);
        dut_reg_fields[r].reset();
    end
end
end
```

Check if parameter  
includes feature

Conditionally set the  
fields as read-only

- Need to verify that a removed register/field really was removed
- UVM built-in register tests adapt automatically

## Tuning Testbenches with RTL Parameters

The presence/absence of a register in a DUT version can be specified in an RTL parameter. This slide shows how a UVM RAL implementation would make use of that parameter when defining the access type of that field (via `reg_field.set_access()`). A nice feature of the UVM RAL is that the built-in register test sequences will automatically adapt to the access types of the registers. However, additional testing is required to check if a register which was present in one DUT version, truly is no longer present in another DUT version. This would be done by attempting to write to the register and checking that the write value was not stored (or checking for a slave error on the register block's port)



# Tuning Testbenches with RTL Parameters

- RAL uses RTL parameters for field sizes
  - Could not be changed dynamically in reg model
- Note: this changed in UVM 1800.02; our solution was developed before then
- Clever use of factory overrides provides a solution

```
class myreg_featA_channelen_reg extends uvm_reg;
  `uvm_object_utils(myreg_featA_channelen_reg)
  uvm_reg_field channel_en;

  function void build_phase(uvm_phase phase);
    channel_en = uvm_reg_field::type_id::create("channel_en");
    channel_en.configure(
      .parent  (this),
      .size    (8),
      .lsb_pos (0),
      .access  ("RW"),
      [...]
    )
  endfunction
```

Same size regardless of  
parameters

## Tuning Testbenches with RTL Parameters

The width of a register field can be specified RTL parameter. We build this example over the course of a few slides, so here we just show the most basic approach of hardcoding the size field. Note that this technique was developed prior to UVM 1800.02, which allows a register model to be unlocked and modified, and so there may be better ways to do this going forward. 1800.02 is also called UVM 1.2



# Tuning Testbenches with RTL Parameters

- RAL uses RTL parameters for field sizes
  - Use virtual method to return the size parameter

```
class myreg_featA_channelen_reg extends uvm_reg;
  `uvm_object_utils(myreg_featA_channelen_reg)
  uvm_reg_field channel_en;

  function void build_phase(uvm_phase phase);
    channel_en = uvm_reg_field::type_id::create("channel_en");
    channel_en.configure(
      .parent  (this),
      .size    (get_channel_en_size()),
      .lsb_pos (0),
      .access  ("RW"),
      [...]
    )
  endfunction

  virtual function int get_channel_en_size();
    return 8;
  endfunction
```

Use method to  
return value

'virtual' so method body  
can be overridden

## Tuning Testbenches with RTL Parameters

An improvement over hardcoding the "size" field is to use a function to return a value. The 'base' version of the function returns a fixed number, so there's no value being added just yet. But by making the function virtual (as we have above), a child of this class could override the function to return a different size.

# Tuning Testbenches with RTL Parameters

- RAL uses RTL parameters for field sizes

```
Class myreg_featA_channelen_extended_reg extends myreg_featA_channelen_reg
  uvm_object_utils(myreg_featA_channelen_extended_reg)

  virtual function int get_channel_en_size();
    rtl_info_struct_t dut_rtl_param_info;
    if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
        "rtl_info_struct", dut_rtl_param_info))
      `uvm_fatal(...)
    return dut_rtl_param_info.featureA_reg_width;
  endfunction
endclass
```

Get parameter value  
from the config db

```
class my_env extends uvm_env;
  ...
  set_type_override_by_type(myreg_featureA_channelen_reg::get_type(),
    myreg_featureA_channelen_extended_reg::get_type());
  ...
endclass
```

Factory override register  
type

## Tuning Testbenches with RTL Parameters

This is where the payoff is: by extracting the RTL parameters, implementing a child class and overriding the virtual function `get_channel_en_size()`, we can have the register model specify the 'size' field based on the RTL parameters of the DUT – very powerful! The final step is to use the factory to override the base class with the child class.

# RTL Parameter Generation & Coverage



## RTL Parameter Generation & Coverage

- Need to specify what RTL parameters will be simulated
- Solution: autogenerate a SystemVerilog file that assigns all parameter values using ``define's`
  - Avoids error-prone manually maintained “regression list” files
- Generation script can be coded in a scripting language
  - But best done in self-contained SystemVerilog module to take advantage of randomization engine



There are several different ways we can use to generate the actual RTL parameters used in a simulation, but all of them require this to be done in advance. We want to avoid manually created sets of parameters as these are error prone and difficult to maintain. We could use any scripting language but it's better to use SystemVerilog as we can take advantage of the built in constrained random engine

# RTL Parameter Generation & Coverage

```
// RTL parameter generation script
// implemented in SystemVerilog
// instead of Python/Perl/etc. to
// take advantage of randomization
// engine
module generate_rtl_params_file;
  initial
  begin
    // SV randomization & file I/O
  end
endmodule
```

generates

```
//auto-generated RTL parameters file
`define X_BUS_WIDTH 13
`define X_ADDR_WIDTH 4
...
```

```
// Testbench file
module tb();
  X #(.BUS_WIDTH (`X_BUS_WIDTH),
    .ADDR_WIDTH(`X_ADDR_WIDTH) x_inst (
  ...
```

## RTL Parameter Generation & Coverage

Whether we use SV to generate the parameters, or any other scripting language, the generated parameters need to be written out to a file. One way to do this is to have the file contain a set of `define values, and then refer to these in our test environment.

# RTL Parameter Generation & Coverage

- Random parameter generation requires coverage!
- RTL parameter information can be sampled in SV covergroups
- Data can be merged and analyzed after regression runs
  - Even across different RTL snapshots which vary over parameter value permutations
- TB can get all RTL parameters from config\_db to perform coverage
- Very important to cross with functional coverage to avoid false positives
  - Functionality which is parameter-dependent must be considered for crossing

## RTL Parameter Generation & Coverage

Once we have generated the parameter values, we need to know what was actually generated and SystemVerilog coverage is ideally suited to this task. This allows us to ensure that all the relevant cases are covered. As the testbench automatically adapts to the parameters (it retrieves them from the config\_db), theoretically we only need to compile the testbench once, even though the RTL will be compiled for each set of parameters. The testbench can extract all the parameter values used from the config db and use these for the coverage. For the coverage, we cover the parameter values themselves but should also consider using the parameters as cross items in functionality which is dependent on the value of the parameters. Note that code-coverage probably cannot be merged across different parameter sets.

# RTL Parameter Generation & Coverage

```
class my_tb_covergroups extends uvm_object;
  `uvm_object_utils(my_tb_covergroups)
  rtl_info_struct_t dut_rtl_param_info;

  covergroup my_dut_rtl_params_cg;
    cp_rtl_params_num_ports: coverpoint dut_rtl_param_info.num_ports;
    ...
  endgroup

  function new(string name);
    //... extract into dut_rtl_param_info via uvm_config_db like shown previously
    ...
    my_dut_rtl_params_cg = new();
    my_dut_rtl_params_cg.sample();
  endfunction
endclass
```

Add a coverpoint for each  
RTL parameter

Sample covergroup during  
construction as parameters  
are constant for a given run

## RTL Parameter Generation & Coverage

As we saw earlier, we have an `rtl_info_struct` type that encapsulates the RTL parameters used. This can be sampled at construction time of the covergroup as all the RTL parameters are constant (and known) at that point for a given simulation of a particular set.



## Parameter Selection Optimization

- How do we decide which parameterizations to simulate?
- Number of permutations for even a simple IP block can be prohibitive
- Usual solutions:
  - Test "common parameterizations" exhaustively, "random parameterizations" lightly
  - Test "random parameterizations" exhaustively

"random parameterizations" inadequately verified

"common parameterizations" inadequately verified



Deciding what to test in a complex system is part of the art of verification. But it's too risky to leave such a choice to chance, and we verification engineers can do better than that. The two extremes of doing just full random, or just targeted parameterizations, leave important holes in what's not covered by verification. Often projects will do a combination of the two to get the benefits of both worlds - but there's still opportunity for improvement from doing that too.



# Parameter Selection Optimization

- Software testing has used the *pairwise* technique for many years
  - An attempt to focus our finite testing effort as effectively as possible
  - Complements constrained-random (does not replace it)
- Basic principle: don't try to test all combinations of variables, but instead:

**For every pair of variables, test every  
combination of that pair**

## Parameter Selection Optimization

The pairwise testing technique comes from the software world and is simple in practice: choose every pair of variables in the system and run tests with every possible value combination of that pair.



# Parameter Selection Optimization

- Why does pairwise work?

Biggest "bang for the buck"

Bug caused by a specific value on	Probability of bug existing	Difficulty of uncovering bug
One parameter	High	Easy
Two parameters	High	Hard
Three or more parameters	Low	Very Hard

- Blind application of pairwise is not recommended
  - Random parameterizations should still be run for 'bug hunting'

## Parameter Selection Optimization

The pairwise approach is effective at finding bugs in many systems because bugs are often present when 2 variables have a certain value pair, and covering all possible value-pairs is doable within a reasonable time frame. Bugs can and do exist for higher-order combinations (e.g. 3 or more variables having a set of values), but covering all possibilities at these higher levels is prohibitive in terms of time (we're talking thousands of years of testing time in some instances!). Often such bugs are not as prevalent as those that can be found with pairwise, and so it's a good approach to cover all pairwise parameterizations. The authors still think constrained-random test should supplement pairwise testing, to uncover bugs that the pairwise didn't catch.

# Parameter Selection Optimization

- Number of tests for full pairwise coverage lower than you might expect  
– 5 RTL parameters for a simple DUT

0-3    0-3    0-2    0-1    0-1

P1    P2    P3    P4    P5

- **192** combinations to be tested for exhaustive verification! (4x4x3x2x2)

- Test **every** combination of **pairs**

P1×P1	Total of 88 value-pairs			P4×P4
P1×P2 16	P2×P2 16	P3×P2 12	P4×P2 12	P5×P2 12
P1×P3 12	P2×P3 12	P3×P3 6	P4×P3 6	P5×P3 6
P1×P4 8	P2×P4 8	P3×P4 6	P4×P4 4	P5×P4 4
P1×P5 8	P2×P5 8	P3×P5 6	P4×P5 4	P5×P5 4

parameter pair P2,P3:  
12 possible values

parameter pair P4,P5:  
4 possible values

## Parameter Selection Optimization

We present an example of how pairwise testing would work for a DUT with 5 RTL parameters. Notice that although there are 192 different combinations to achieve exhaustive coverage, there are only 88 different value-pairs to be checked to achieve pairwise coverage.

# Parameter Selection Optimization

	P1	P2	P3	P4	P5
1	0	0	0	0	0
2	1	0	1	1	1
3	2	0	2	1	0
4	3	0	1	0	0
5	0	1	2	0	1
6	1	1	0	1	0
7	2	1	1	0	1
8	3	1	0	1	1
9	0	2	1	1	0
10	1	2	2	0	1
11	2	2	0	0	1
12	3	2	2	0	0
13	0	3	0	0	0
14	1	3	1	1	1
15	2	3	2	0	0
16	3	3	1	1	1

192 possible parameterizations

Pairwise: thorough testing with only 16 parameterizations

complete coverage of P4xP5

complete coverage of P2xP3

### Larger example

- 20 parameters, each with 10 values
- $10^{20}$  possible parameterizations
- Pairwise covered by **230** tests
- Generator runtime ~10 seconds

## Parameter Selection Optimization

The 88 value pairs to be checked require less than 88 tests because more than one value pair can be covered in a single test. This example reached pairwise coverage with just 16 tests. To compare, our attempts with pure randomization required 3x as many random seeds to achieve pairwise coverage.

# Parameter Selection Optimization

```

`include "nwise.svh"
typedef enum {P_NONE, P_ODD, P_EVEN} param_t;

class Params extends Nwise_base#(uvm_sequence_item);
...
  NWISE_BEGIN(Params)
  NWISE_VAR_INT( int, P1, {[0:3]} )
  NWISE_VAR_INT( bit[1:0], P2 )
  NWISE_VAR_ENUM( param_t, P3 )
  NWISE_VAR_INT( bit, P4 )
  NWISE_VAR_INT( bit, P5 )
  ...

  string name;
  function new(...);
  ...
endclass

rand int P1;
constraint c_NWISE_P1 {
  P1 inside {[0:3]};
  P1 = __nwise_value_proxy[...];
}
  
```

www.verilab.com

Mixin supports UVM base classes if required

Value-set required for wide types

Other user code OK

Macro-generated code (simplified)

P1 P2 P3 P4 P5

## Parameter Selection Optimization

The Verilab nwise.svh library was designed with general configurations in mind, as opposed to RTL parameters, but it works equally well for generating RTL parameters. This example shows how a series of variables can make use of the package, to enable pairwise generation.

# Parameter Selection Optimization

- Call methods of Nwise\_base to get patterns

```
class Params extends Nwise_base#(...)
```

```
...  
  NWISE_BEGIN(Params)
```

```
  N  
  N  
  Params prms = new(...);  
  int num_patterns = prms.Nwise_generate_patterns(2);  
  for (int p = 0; p < num_patterns; p++) begin  
    prms.Nwise_render_pattern(p);  
    $display("Params %2d = %b,%b,%s,%b,%b",  
            p, prms.P1, prms.P2, prms.P3.name, prms.P4,  
            prms.P5);  
  end
```

Generation order:  
2 for pairwise

```
Params 0 = 00,00,P_NONE,0,0  
Params 1 = 01,00,P_ODD,1,1  
Params 2 = 10,00,P_EVEN,1,0
```

## Parameter Selection Optimization

The nwise.svh library has a simple API for generating pairwise value pairs. The value pairs can then be printed out to a file for use by a regression management script (or define a macro file, similar to what's shown in the previous section "RTL Parameter Generation & Coverage". For more details on this technique, see

<https://www.verilab.com/resources/papers-and-presentations/#dvcon2015nwise>.

# Additional Topics

# Generated/Templated RTL Code

- RTL code generators can be tricky to deal with
  - Often used by IP vendors
  - May require parameter/configuration info published to config\_db
- Generated code not always expressible as parameters
  - Requires corresponding TB code generator
- How to handle this will depend on your project's circumstances



# ***`define* Rather Than SV Parameters**

- RTL code parameterization can be implemented as *`define* macros instead of module parameters
  - Techniques presented here are overkill
  - Faster/simpler to capture the *`define* values into TB config object

```
//DUT PARAMS FILE  
`define DUT_PARAM1 = 5  
// etc.
```

```
class config_obj extends uvm_object;  
  `uvm_component_utils(config_obj)  
  
  int dut_param1 = `DUT_PARAM1;  
  // etc.  
  ...  
endclass
```

## *`define* Rather Than SV Parameters

Instead of using parameterized modules, we can directly use *`define* macros to refer to the parameter values. In this case, we can directly use these in our usual testbench configuration object and store that in the config\_db, rather than the extracted parameters struct we saw earlier

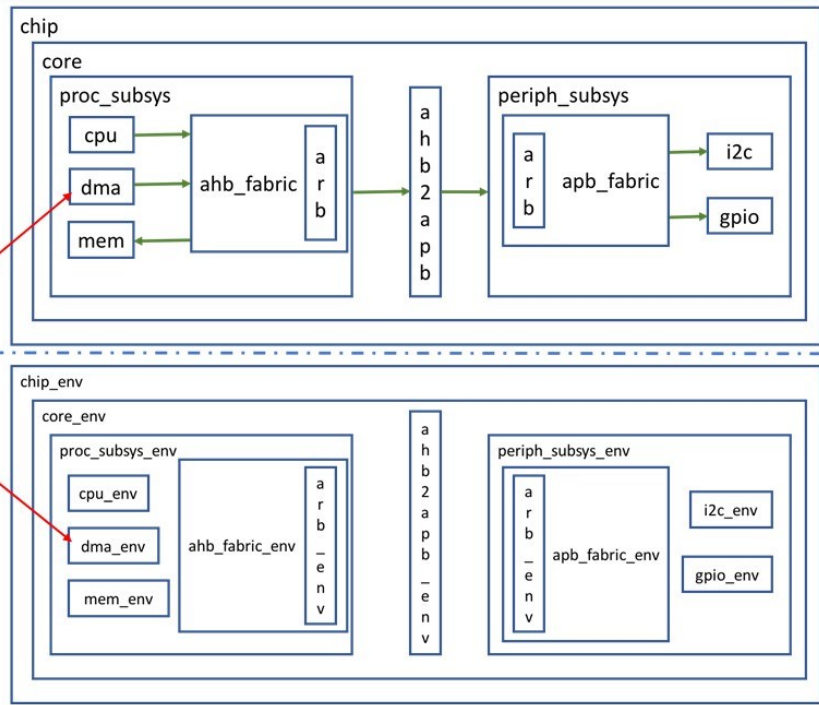
# Configuration-Aware Testbenches

- Testbench can auto-adapt to different DUT configurations
- Powerful technique when:
  - Many DUT configurations needed
  - Many stubbing configurations needed
  - A single testbench for block/system-level is desired
- Requires the following techniques
  - UVM harness with port coercion (already discussed)
  - UVM environment mirroring of design hierarchy
  - Auto-publishing of virtual interfaces

**Env component  
names match  
module instances**

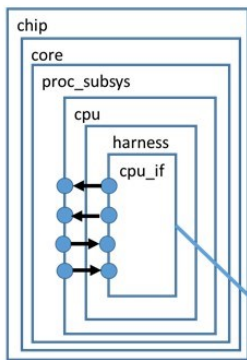
RTL

TB



## Mirrored Hierarchies

The key takeaway to this slide is that the names and hierarchy of the system under test is exactly mirrored by the testbench environment.



```

interface cpu_harness();
    cpu_if_type cpu_if(.clk(cpu.clk),
                      .rst(cpu.rst),
                      ...
                      );

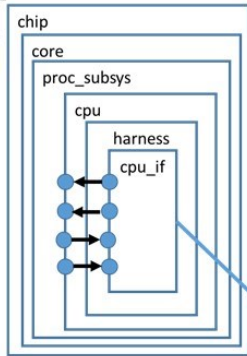
    initial begin
        uvm_config_db#(virtual cpu_if_type)::set(null, $sformatf("%m"),
        "cpu_if", cpu_if)
    end
endinterface
    
```

Use %m to define scope  
for set()



## Self-publishing Interfaces Using Module Path Name

We can use the module path format specifier **%m** to auto-publish the scope of the bound `cpu_if`.

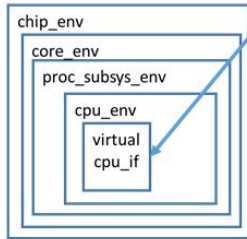


```
interface cpu_harness();
  cpu_if_type cpu_if(.clk(cpu.clk),
                    .rst(cpu.rst),
                    ...
                    );

  initial begin
    uvm_config_db#(virtual cpu_if_type)::set(null, $sformatf("%m"),
                                             "cpu_if", cpu_if)
  end
endinterface
```



Scope for get() will be identical to that of set()

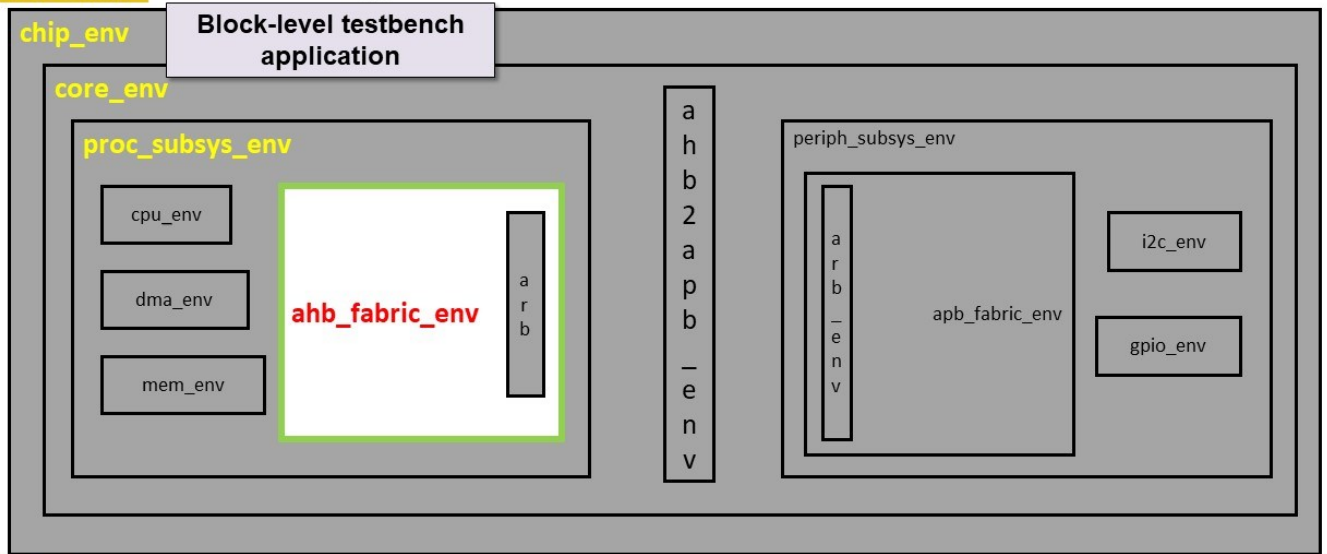


```
class cpu_env extends uvm_env;
  virtual cpu_if_type vif;
  ...
  function void build_phase(uvm_phase_phase);
    if(!uvm_config_db#(virtual cpu_if_type)::get(this, "",
                                                  "cpu_if", vif))
      `uvm_fatal("NOVIF", "No cpu_if in uvm_config_db")
  end
endclass
```

## Automatic Retrieval of the Correct Virtual Interface

Now when we “get” the virtual interface from the config\_db it automatically has the correct scope

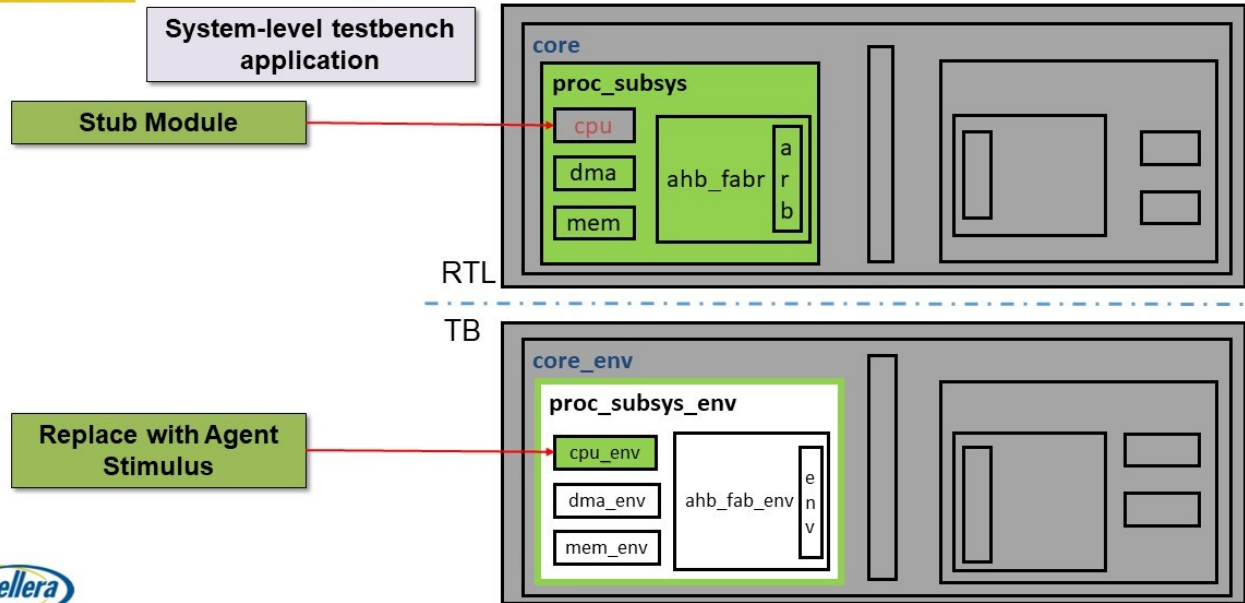
# Configuration-Aware Testbenches



## Configuration-Aware Testbenches

This illustrates a portion of the verification environment of a block we wish to reuse at system level

# Configuration-Aware Testbenches



## Configuration-Aware Testbenches

We can use parameters to stub-out the actual CPU RTL in the system-level RTL and then use an agent in our testbench to drive the actual stimulus in the system, via the embedded virtual interface we saw in the earlier slide



# Polymorphic Harnesses

- Same as regular UVM Harness plus you can publish to config\_db
  - Avoids use of SV struct to hold RTL parameters
    - Classes “get” the harness, harness fields hold RTL parameters
  - Allows classes to call methods defined inside harness
    - e.g. preload memories, force signals for error-injection
- Syntax not straightforward, beyond workshop scope
  - Benefits are worth the added complexity

## Polymorphic Harnesses

This is an advanced technique that’s particularly powerful where an ability to force or examine internal signals in the RTL is required. SystemVerilog does not allow a virtual interface handle to be created to an interface which has cross-module references. However, these can be quite handy for forcing stimulus into the RTL to simulate difficult-to-hit error conditions. This abstract BFM technique was first described by Bromley & Rich in 2008. Our paper “Verification Prowess with the UVM Harness” builds on this technique

# When is Verification Finished?

- Exhaustive coverage for the one set of parameters corresponding to key customer requirements
- Pairwise coverage of parameters
- Whatever random coverage of parameters time/resources permit after that
- Being vigilant about what *doesn't* need to be verified

## When is Verification Finished?

Customer requirements are always top of the list for determining verification completion. Following that, it's important to focus on what does need to be verified and guard against verifying sets of parameters that, while theoretically possible, will never be used in practice. As always, a good specification should be the driver.

## Conclusion

- Parameterized RTL can be handled in SV/UVM in a clean and powerful fashion
  - UVM Harness can handle parameterized instance counts and bus widths
  - RTL parameters can be captured and used to tune the TB
  - Pairwise can be applied to RTL parameter generation
  - Coverage can be applied to RTL parameters
  - Auto-publishing virtual interfaces and hierarchy mirroring allows seamless TB adaptation to RTL parameters

# Thanks to Our Contributors & Reviewers

Jonathan Bromley

Kevin Johnston

Mark Litterick

David Long  
(Doulos)

Alex Melikian

Jeff Montesano

Paul Marriott

Tamas Simon

Dan Slocombe

Jason Sprott

Jeff Vance

Kevin Vasconcellos

Thanks to Our Contributors & Reviewers

# References

- |   |                  |
|---|------------------|
| 1. My Testbench Used to Break,                  | DVCon USA 2018   |
| 2. UVM Audit Tutorial,                          | DVCon EU 2018    |
| 3. Verification Prowess with the UVM Harness,   | SNUG Austin 2017 |
| 4. Perplexing Parameter Permutation Problems,   | SNUG Canada 2017 |
| 5. Effective SystemVerilog Functional Coverage, | SNUG UK 2016     |
| 6. Advanced UVM Tutorial,                       | DVCon EU 2015    |
| 7. Is Your Testing N-wise or Unwise?,           | DVCon EU 2015    |
| 8. SVA encapsulation in UVM,                    | DVCon USA        |
| 2013  |                  |



Available at [www.verilab.com](http://www.verilab.com) under "papers and resources"

## References

[Jeff.Montesano@verilab.com](mailto:Jeff.Montesano@verilab.com)

[paul.Marriott@verilab.com](mailto:paul.Marriott@verilab.com)