Relieving the Parameterized Coverage Headache

Christine Lovett, Bryan Ramirez, Stacey Secatch
Xilinx, Inc.
3100 Logic Dr.
Longmont, CO 80503
cristi.lovett@xilinx.com , byran.ramirez@xilinx.com,
stacey.secatch@xilinx.com

Michael Horn
Mentor Graphics, Corp.
1811 Pike Rd.
Longmont, CO 80501
mike_horn@mentor.com

***Abstract*— Modern FPGA and ASIC verification environments use coverage metrics to help determine how thorough the verification effort has been. Practices for creating, collecting, merging and analyzing this coverage information are well documented for designs that operate in a single configuration only. However, complications arise when parameters are introduced into the design, especially when creating customizable IP.**

This paper will discuss the coverage-related pitfalls and solutions when dealing with parameterized designs.

Keywords UVM, OVM, SystemVerilog, parameters, coverage, coverage closure, verification methodology, testbench, intellectual property (IP) cores

I. INTRODUCTION

Code reuse is increasingly common, mostly because it helps to make engineers more productive. Through reuse it is possible to use a single code base tuned to specific requirements in lieu of maintaining multiple code bases. A standard ASIC technique is to use strap signals along with register programming at the start of operation to generate a customized configuration. Designs on programmable fabric can take this one step further. The downloadable netlist can be configured by the use of parameters, which optimizes away unneeded logic at synthesis time. IP providers use this technique to generate optimized netlists for every customer with a single infrastructure.

The addition of parameters increases complexity compared to a non-parameterized design, but working from one code base instead of multiple customized solutions generally makes this complexity well worth it. However, compared to working on a single static configuration, verifying designs over a range of parameterizations can greatly increase the coverage space and, consequently, the effort required to reach coverage closure. The problem is compounded as the number of modifiable parameters increases. In this paper, functional coverage of the parameter space will be referred to as "parameter coverage" and coverage of variable constants with functional coverage will be referred to as "parameterized coverage." Parameter coverage appears to be just another coverage metric, except that it needs to be closed in the compile/optimization phase of simulation, instead of the run-time phase where normal functional coverage is closed.

The Verilog Language Reference Manual (LRM) [1] describes in straightforward terms how to write parameterized Register Transfer Level (RTL). The addition of SystemVerilog onto the Verilog standard has made the language more powerful, allowing more advanced verification techniques to be used in developing testbenches.

Gaps still exist when trying to verify a parameterizable design within a SystemVerilog testbench, especially during the coverage merge step. The SystemVerilog standard doesn't specify in detail how coverage is to be merged. For a design with a single parameterization, normal verification techniques generally lead to more or less normal merging of coverage. However, for multiple parameterizations it's not clear what should be done when "constant" values in *covergroups* are no longer so constant. Likewise, classes that are parameterized with different values are now different types, and don't appear to be part of the same merge hierarchy.

Along with the new requirement of parameter coverage, standard functional coverage metrics including coverage groups, coverage properties, assertions, and code coverage are still used. However, workarounds are needed to handle the non-"constant" constant problem that parameters add to functional coverage. This paper will focus on ways to make coverage useable for a parameterized design.

A. Coverage for Reusable Code Bases

The parameterized coverage techniques described in this paper developed as a result of Xilinx's verification of its Serial RapidIO (SRIO) Gen2 IP core. The Serial RapidIO protocol is a serial standard supporting numerous line rates and multiple link widths. The Xilinx SRIO Gen2 IP is highly configurable, built to provide customers with flexibility while limiting the resources required for each specific application. This configuration is done with parameters, thus allowing the synthesis tools to provide an optimized solution for every design permutation. There are currently 77 parameters in the design, which control everything from register defaults to the existence of code to the changes in the protocol behavior or bus widths.

The issues described in this paper may plague anyone creating reusable designs. It was particularly important during the development of the SRIO Gen2 IP to ensure the core was tested to some degree across the various supported permutations. However, this could also apply to anyone trying to future-proof their designs by improving maintainability. Any

design that uses a single code base for multiple synthesized netlists will find these techniques useful. The two most common issues with coverage that arise from using parameters are varying bus widths and generated code. As bus widths vary, it makes it difficult to create meaningful coverage for the current parameterization being targeted. Generated code is an issue because code may or may not exist.

One of the problems with parameterized coverage is determining when to merge a certain set of coverage and when to keep it separate. Unfortunately, there are no clear rules dictating when to merge and when not to. Merging coverage is usually preferred as it enables quicker coverage closure. However, before completing the merge, the user must evaluate the goal of the coverage and determine if it must be kept separate. Specific examples of this will be shown later in the paper.

1) Parameter Coverage

In an ideal world, all parameters would be fully crossed with each other to create parameter coverage, and then that would also be completely crossed with all functional coverage. This would ensure that each functional coverage item was tested against each specific parameterization. As the coverage space grows exponentially, and additional parameters or parameter values are added, this comprehensive approach quickly becomes infeasible. For example, if a design has two parameters, each with two different values, crossing the parameters would result in four different parameterizations, which translates to four times the amount of functional coverage. What happens if the design has more parameters? Assume now the design has ten parameters, again with only two possible values each. This would translate to 2^10 (1024) different parameterizations. Closing on functional coverage is often difficult enough, but it would take an unreasonable amount of time if it had to be closed over 1000 times. Imagine the problem if a design has 50 to 100 parameters with more than two values for each parameter. The same techniques to close functional coverage can be used to close parameter coverage. The verification engineer crosses parameters that interrelate just like regular functional coverage, leaving parameters that don't interfere with each other as stand-alone *coverpoints.*

During coverage closure on one of the SRIO Gen2 testbenches, there were a set of parameters that were particularly difficult to close by pure constrained random techniques. The parameter coverage controlled what types of transactions could be initiated and targeted on the user interfaces and resulted in 512 unique parameterization crosses. Relying solely on constrained random would have taken weeks to close this coverage. Realizing this amount of time was unacceptable, randomizations were directed to generate all 512 parameterizations, enabling coverage to be closed in a weekend. Closing holes in parameter coverage is generally not a difficult problem.

While closing the parameter coverage space is a straightforward problem, crossing each point in that space with the complete functional coverage space is not realistic. Therefore, a different approach is needed. A couple of options exist to ensure functional coverage is adequately achieved relative to the various parameterizations. The first step is to cover all parameter settings, but without crossing them. This verifies that all parameter values have been tested. To further validate the parameter coverage, only important parameters or parameters that have a direct effect on another should be crossed. Finally, specific functional coverage items should be crossed with any parameters that affect that functionality.

2) Formal Verification with Parameterized IP

Formal verification is another popular technique that can be used to close coverage on a design. Formal excels at arbitration and other state-deep problems that are difficult to cover with functional coverage. Additionally, most vendors provide static checks that check for code liveness by locating dead code or unhittable states in state machines that simplify code coverage analysis.

However, parameterizable code is not currently a good fit for this method. Normally, any design is constrained into a specific configuration, and then functionality and code liveness are mathematically proven for that mode. The process can then be repeated if the constraints need to be changed for a different configuration. However, for parameterized designs, every possible permutation needs to be proven in order to get the same level of confidence as is possible with a non-parameterized design. Like parameter coverage, a subset of parameter crosses can be made here, but with 77 parameters, covering all the parameter interactions still expand the number of configurations into a completely infeasible number.

While formal property checking is a poor fit for parameterized code, formal liveness checks aren't even possible. One of the purposes of parameters is to have a common code base for different functionalities. This means for any given parameterization there will be dead code as illustrated by the two examples in Fig. 1 below.

```
// Example 1: Explicitly declaring dead code
// using a generate statement
generate if (MODE1[1]) begin
  // Active when MODE1 is 1, dead code when
  // MODE1 is 0
end else begin
  // Active when MODE1 is 0, dead code when
  // MODE1 is 1
end

// Example 2: Implicit dead code using an
// if statement
always @(posedge clk) begin
  if (MODE1) begin
    // Dead code when parameter MODE1 is 0
  end else begin
    // Dead code when parameter MODE1 is 1
  end
end
```

Figure 1. Parameterized Dead Code

[1] *Design parameters will be written in all capital letters throughout this paper.*

II. Parameters and Functional Coverage

Even in the case of a single design configuration, functional coverage requires a lot of time and effort to do correctly in the case of a single design configuration. When parameters and a parameterized design are factored into the equation, additional considerations arise.

A. Parameters and SystemVerilog covergroups within classes

Any value in a coverage bin definition must be constant and defined when the coverage group is constructed. Occasionally these are literal values, but parameterized designs also require parameter values in these definitions. Therefore, there needs to be some way to get parameters down into coverage groups.

One way would be to declare a covergroup inside a parameterized class and use the parameters of the parent class. The biggest issue when dealing with parameterized classes is that, as far as SystemVerilog is concerned, every different parameterization of a class results in a different type. Unlike class extensions, multiple parameterizations of a parameterized class (specializations) are not compatible through polymorphism[5]. The original class with default parameters can't be used as a handle as it still creates a unique default specialization. As a result, coverage merging tools see different types when they are trying to merge, and are unable to do so. If a covergroup is defined within a parameterized class and different simulations are run with coverage results saved out, the coverage results contained within each parameterized class instance will not be able to merge together.

Our preferred solution is to not define covergroups within a parameterized class but rather in a non-parameterized class, which is then instantiated within the parameterized parent class. Two variations exist for passing parameter values to the non-parameterized class:

1) Generalized solution for getting parameters into covergroups

The first option, which works regardless of the methodology used, is to pass parameter values from the parameterized class as constructor arguments. This exposes the information needed to collect useful coverage, but without affecting post-simulation coverage merging. A simple example of what this would look like in code is shown in Fig. 2

```
// The coverage_collector class contains all
// the coverage to be sampled from the
// my_agent class
class coverage_collector;
  int full_count; // Indicates fifo fullness

  covergroup cg_fifo_count(int fifo_depth[2]);
    coverpoint (full_count) {
      bins empty  = {0};
      bins in_use = {[1:fifo_depth-2]};
      bins full   = {fifo_depth-1};
    }
  endgroup

  function new(int fifo_depth = 16);
    cg_fifo_count = new(fifo_depth);
  endfunction : new
  ...
endclass : coverage_collector
```

Figure 2. Parameter Passing Using New

In this example, arguments are added to the constructor (one per parameter). Once this class is defined, then it could be used in a parameterized class as shown in Fig. 3.

```
// my_agent instantiates the
// coverage_collector
class my_agent #(int FIFO_DEPTH = 16);
  coverage_collector  coverage_col;
  ...

  function void build();
    coverage_col = new(.fifo_depth
                       (FIFO_DEPTH));
  endfunction : build
  ...
endclass : my_agent
```

Figure 3. Parameterized Class Providing Parameters to Coverage Class

Fig. 4 below shows this example where my_agent is a parameterized class passing FIFO_DEPTH through the constructor to the coverage_collector. The coverage_collector then passes this on to the *covergroup* cg_fifo_count.

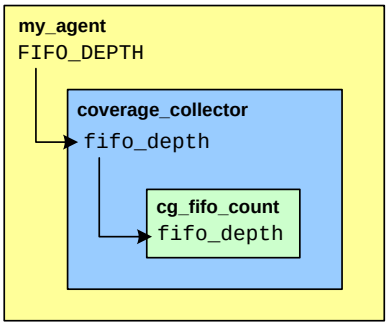

Figure 4. Parameters passed through the heirarchy

By extending the constructor in the non-parameterized class, parameters can be used to collect and affect coverage. This

[2] *Parameters which are passed as arguments to children or covergroups will be denoted with the same parameter name, but in lowercase letters.*

technique is useful for simple cases and can easily be extended to deal with multiple parameters. However, consider a real world design such as SRIO where 77 parameters are used. These parameters would have to be passed down through every constructor, a tedious requirement that would make change difficult.

2) Getting parameters into covergroups for a OVM/UVM methodology

Our preferred solution for any OVM[6] or UVM[7] based environment uses a variant of this technique that is easier to maintain for a highly parameterizable design. An object should be created containing all of the possible parameters for a design. The object can also contain the parameter coverage for the design, since all variables are present in one location. The object is then placed into the configuration space, making it available to any object anywhere in the object hierarchy. Note that this means that some unnecessary parameters are in the object; but this is overshadowed by the simplification of having all parameters available in a common location. In the case where large numbers of parameters are used, such as in SRIO, using the parameter object provides an easy, reusable way to get the parameter information to coverage classes. *Covergroups* must be created in the constructor of their parent class. However, since the parent class isn't constructed until the build phase, the configuration class is already available to the covergroup when it is constructed inside the parent. This is not a problem since the build phase is top down and has already completed for the top level test which places the parameter object into the configuration space. Using this variation, a configuration object is created to hold the parameter values. An example container is shown in Fig. 5 below.

```
// The param_containter class hold the values
// for all parameters in the design
class param_container extends uvm_object;
  `uvm_object_utils(param_container)

  // Data members to hold parameter values
  // Each parameter is set to a default value
  // these values are overwritten in the
  // parameterized test base when the object
  // is created.
  int fifo_depth = 16;
  // other parameter values
  ...

  // constructor, etc.
endclass : param_container
```

Figure 5. Parameter Container Class

The coverage_collector class then gets the parameter container prior to constructing the *covergrou*p as shown in Fig. 6. Note that the coverage object must still pass in every parameter that is required.

```
// The coverage_collector class contains all
// the coverage to be sampled from the
// my_agent class
class coverage_collector
  extends uvm_subscriber #(transaction_class);

  function new(string        name,
               uvm_component parent);
    param_container params;
    ...

    // Call into the configuration space
    // to get the FIFO_DEPTH value
    void'(uvm_config_db #(param_container)::
                        get(this, "",
                            "param_container",
                            params));

    cg_fifo_count = new(params.fifo_depth);
  endfunction : new
  ...
endclass : coverage_collector
```

Figure 6. Parameter Passing Using the Configuration Space

This method for parameter access is shown in Fig. 7 below.

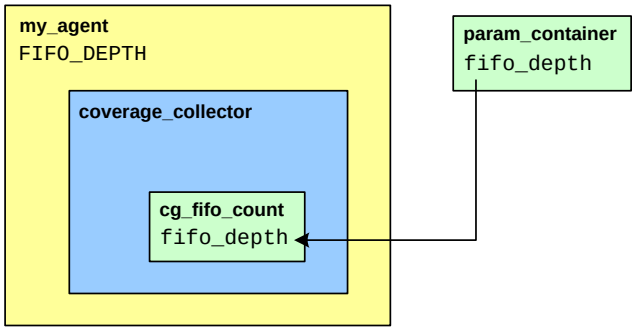

Figure 7. Parameter Accesses Using the Configuration Space

A single instance of the *param_container* class is created, has its parameter values set and is then placed into the configuration space in the UVM test. Using a configuration coverage object to encapsulate parameters is described in the DVCon 2011 paper "Parameters and OVM: Can't They Just Get Along" [3].

B. Class- versus Module- Based Coverage

When writing functional coverage, it is important to determine where it should live within the test hierarchy. Two options to consider are class- and module-based coverage. Class-based coverage is when *covergroups* are defined within a class such as a UVM subscriber. *Module*-based coverage has *covergroups* defined within *modules*. The *modules* could be standard RTL code, or SystemVerilog *interfaces* or *modules* that are instantiated using *bind* statements. Note that SystemVerilog requires cover properties and assert properties to be located within *modules*. Regardless of where the coverage is placed, tradeoffs must be considered when parameters are involved.

1) Class-based Coverage

Protocol level coverage should be isolated within a transaction class that is used across interfaces in a design. Coverage for the protocol can then be reused across all interfaces handling that transaction. Some examples of protocol coverage for SRIO include packet types, valid byte sizes, packet priorities, and other applicable packet fields. Fortunately, protocol coverage should have no need for parameterized code. All parameters should be isolated within the layers around the coverage class as described above. In the event the coverage class does require a parameter value, workarounds are required in order to access those parameters, such as the previously shown configuration object.

In order to write a *covergroup* covering a signal with a parameterized width in a class, the minimum and maximum values should be considered based on all possible configurations and the bins must be appropriate for all configurations. Some coverage is not possible for a given configuration, which will result in run-time warnings. However, all coverage will be closed over a full regression run. The following example covers a counter which iterates though packet identifiers (IDs) and verifies all IDs are used. The ID_SIZE parameter is used to set the number of available IDs to the 0 to 31 range when ID_SIZE=SMALL or the 0 to 63 range when ID_SIZE=LARGE. To save resources, the width of the counter is parameterized to 5 or 6, respectively. If this coverage was written in a single *coverpoint*, all bins could not be covered in only one configuration. Once multiple ID_SIZE values are merged, coverage could still be closed over the complete solution space. In this case, two *covergroups* are required in order to make sure all the values are seen in both configurations since coverage cannot be disabled on a bin basis due to restrictions in SystemVerilog. This could also be done using cross coverage, but then many illegal bins would have to be specified for ID_SIZE=SMALL and a value of 31 cannot be grouped into multiple bin types. The cleanest solution is shown in Fig. 8.

```
// Covering packet ID values within a class
// when packet_ids is a parameterized width
covergroup cg_packet_ids;
  cp_small_pid: coverpoint (packet_ids)
                      iff (id_size == small) {
    bins min = {0};
    bins mid = {[1:30]};
    bins max = {31};
  }

  cp_large_pid: coverpoint (packet_ids)
                      iff (id_size == large) {
    bins min = {0};
    bins mid = {[1:62]};
    bins max = {63};
  }
endgroup
```

Figure 8. Covering Packet IDs in a Class

2) Module-based Coverage

Verification of behavior that affects internal logic and is not visible on an external bus level should be covered within a *module*. The reason: higher level classes do not know or care about these details. An example of this, which is seen frequently in SRIO, is handshaking between internal *modules* or across clock domains. When block A asserts a VALID flag, VALID cannot deassert until a READY response is asserted from block B back to block A. One required coverage point is that a stall condition is seen where VALID is asserted without the assertion of READY. This functionality is not visible on a transaction level and should be written in the *module* to verify this expected functionality occurs.

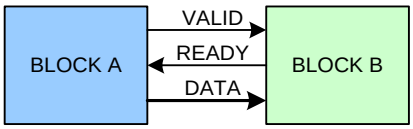

Figure 9. Handshaking Between Two Modules

Writing coverage within a *module* provides access to all parameters in the *module*. This provides the benefit of being able to write only valid coverage based on the current configuration and the ability to merge coverage across *modules* depending on vendor support. Merging across *modules* is possible because a union of all coverage pieces may be available (again, the merge algorithm is not defined in SystemVerilog). In this case, when covering a parameterized bus the width is known and the coverage can be written using the parameters such that only valid bins are created. This eliminates unnecessary warnings. The example in Fig. 10 below shows the same packet ID coverage described above, but where the *covergroup* only contains valid bins based on the ID_SIZE parameter. Since it is within a *module*, the *generate* keyword is available to exploit.

```
// Covering packet ID values within a
// module when packet_ids is a parameterized
// width
generate if (ID_SIZE == SMALL) begin
  covergroup cg_packet_ids;
    cp_small_pid: coverpoint (packet_ids){
      bins min = {0};
      bins mid = {[1:30]};
      bins max = {31};
    }
  endgroup

end else if (ID_SIZE == LARGE) begin
  covergroup cg_packet_ids;
    cp_large_pid: coverpoint (packet_ids){
      bins min = {0};
      bins mid = {[1:62]};
      bins max = {63};
    }
  endgroup
end endgenerate
```

Figure 10. Covering Packet IDs in a Module

3) Placement Recommendation for coverage

Understanding the issues with both *module* and class coverage can help determine where coverage should be located. When covering protocol requirements, it is best to cover in a class because this enables easy reuse of the coverage. The

reason: parameter workarounds aren't required. Coverage of implementation-specific design choices, independent of the testbench, should be located in a *module* where it has access to parameters and does not need to be reusable.

C. Use Cases for parameterized coverage classes

During development of the SRIO core, a range of problems and limitations were seen with SystemVerilog when *covergroups* needed to be parameter aware. The next group of examples illustrates the basic set of problems and shows the associated solutions.

1) Parameterized Number of Bins

One common deficiency is how to use parameterized values as bins within a *covergroup*. In general, if a single maximum value is used, a parameter can be inserted into the bin definition. However, there is no clean way to code the coverage if the parameter affects the number of desired bins. For example, for a buffer structure, it is easy enough to specify full and empty as coverage values. This can even be generalized to any number of specific values, such as empty, quarter, half, three-quarters and full. If the number of values to cover based on the parameterized depth needs to be specified, there is no clean way to do this within a single *covergroup* because the number of values to cover is not known ahead of time. One possible solution is to create a series of *covergroups* within a *generate* block (for *modules*) or within the new function (for *classes*) as demonstrated in Fig. 11.

```
// Break a collection of values into single
// values. Determine which values are
// active and new the correct groups
covergroup watermark(int value);
  coverpoint (curr_watermark) {
    bins hit = {value};
  }
endcovergroup

// Create a placeholder for the maximum
// possible values
watermark cg_watermark[MAX_DEPTH/16];

// Create a covergroup for each value
function new(string        name,
             uvm_component parent);
  ...
  // Determine the depth from the
  // configuration object
  ...
  for (int ii[3] = 0; ii < depth/16; ii++) begin
    cg_watermark[ii] = new(ii*16);
  end
endfunction : new
```

Figure 11. Creating an Array of Covergroups for a Range of Values

[3] *Note the use of " ii" for loop variables rather than "i". The amount of loops is very high in parameterized code. This coding guideline simplifies searching for loop variable use within the code since patterns like "ii", "jj", etc, are unlikely to be found within the text. At the same time, variable patterns like "ii" keep it very clear that the variable is loop-based rather than a signal-based.*

Note that the solution to this problem (determining which *covergroups* to construct), is very similar to the solution for the ID_SIZE parameter example depicted above. Whenever parameters dictate which bins are active, this solution can be used.

2) Parameterized Bin Values

The SRIO core uses the AXI4-Streaming protocol for ports and internal connections. A verification component is used for every interface for improved reuse. While SRIO only has a single bus width, the verification component is designed to be valid for any possible configuration; bus widths may vary from 8 bits to 4096 bits. There is not a clean way to cover a range of specific literal values when dealing with a variable width bus. One requirement is to cover all possible combinations of the byte enables where the enable bits are packed to the right. These bins are easily coded for a constant width signal. For example, a 32-bit bus would have bins of {4'b1111, 4'b0111, 4'b0011, 4'b0001} as valid byte_enables. Figures 12-15 show the valid data bits in green for each of the valid byte_enable values.

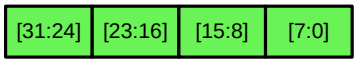

Figure 12. Valid Data Bits When byte_enables is 4'b1111

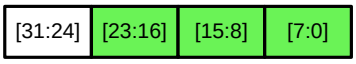

Figure 13. Valid Data Bits When byte_enables is 4'b0111

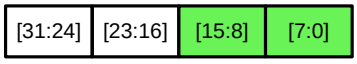

Figure 14. Valid Data Bits When byte_enables is 4'b0011

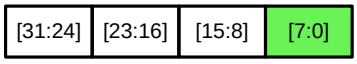

Figure 15. Valid Data Bits When byte_enables is 4'b0001

Coverage within the verification component needs to handle up to 4096-bit data widths. The solution used for this is to directly declare every possible value as shown in Fig. 16. The tools will throw an elaboration warning for bins that are too large and cover the rest normally. Unlike the packet ID example above, the widths that are too large will never be covered for a merged run since the bus width is known in advance. This is not a coverage hole since the excess bins are automatically removed because the literals are not possible given the signal width being covered.

```
// cg_byte_enable_cov contains all the
// coverage to sample from the byte_enable
// signal
covergroup cg_byte_enable_cov;
  coverpoint (byte_enable) {
    bins align[width] = bins{2'b01,
                             2'b11,
                             4'b0111,
                             4'b1111,
                             ...
                             512'b1111_1111…};
  }
  ...
endgroup
```

Figure 16. Variable Width Byte Enable Pattern Coverage

a) SystemVerilog Enhancements

Mantis 2506[4], which is targeted for IEEE 1800-2012, will allow algorithmic bin generation once it is supported by vendors. The current proposal includes an algorithmic function on the left side of the bins statement. This will remove much of the headache and overhead seen with this kind of coverage need. Note that additional work will be needed to ensure that coverage has been written correctly using this new method. Rather than a list of easy to inspect literals, the bins are now the output of a potentially complex function. This feature isn't only useful for parameterized designs. It also allows large or complex *covergroups* to be coded short and succinctly as shown in Fig. 17.

```
// cg_byte_enable_cov contains all the
// coverage to sample from the byte_enable
// signal
covergroup cg_byte_enable_cov(int width);
  coverpoint (byte_enable) {
    // Return type must be specified if the
    // bins returned are larger than an
    // integer. Since the return type is
    // passed on the configuration, the max
    // possible width must be used. For this
    // protocol, MAX_WIDTH = 4096/8
    function [MAX_WIDTH-1:0]
                        left_align(int width);
      for (int ii = 0; ii < width; ii++) begin
        left_align.push_back(
                    MAX_WIDTH'({ii{1'b1}}));
      end
    endfunction

    // byte_enable is the signal being
    // sampled
    bins align[] = byte_enable with
                          (left_align(width));
  }
  ...
endgroup
```

Figure 17. Variable Width Byte Enable Using Mantis 2506

In the actual code for this verification component, this *covergroup* has 102 lines of hand-typed code, with error-prone literals. Using the new syntax, there are only 10 lines of code to maintain. Additionally, only the correct bins are created removing all of the warnings generated by the work-around version.

D. Parameterized Behavior

Protocol coverage is best written in a class which allows for reuse throughout the testbench. This will almost always be the case but complications can arise when protocol functionality is parameterized.

Consider the case of an arbiter where eight different packet types can be issued on each input port and the number of ports is parameterized by value N_PORTS. This is a simplified version of a problem encountered in the SRIO design. These packets can also have variable lengths. In the common use case, a shared class would exist on all N_PORTS and cover all packet types, then cross this with all the valid packet lengths to guarantee all expected functionality is transmitted. Now consider that each of the eight packet types is enabled or disabled through a parameter independently for each port defined. For example, if N_PORTS is one, port A can have 2 to the power of 8 combinations of supported packet types (8 packet types by 2 modes (on and off)). For SRIO, this arbiter has the added complexity that some ports are not allowed to send certain packet types. If packet types were not restricted to certain ports, all types would be covered over a merged run and a shared *covergroup* could be used. With type restrictions on certain ports, sharing a *covergroup* would result in coverage gaps on restricted ports even with merged coverage.

Given the limitations of SystemVerilog, it's difficult to find a clean solution to this problem since *covergroups* cannot be generated inside of a class. One solution is adding configuration settings for the coverage class. Based on these settings, the appropriate coverage is created that only covers the supported transactions. The shared class is still reusable across ports. Note this is similar to the FIFO example where the number of *covergroups* was dynamically selected. The downside to this is that multiple *covergroups* now have to exist, one for each possible configuration. For SRIO, packet type coverage needed to be crossed with various other metrics like packet length and spacing between packets. Combining all these types in one covergroup would result in gaps when a packet type did not exist on a port. To account for this problem, each packet type was defined in its own c*overgroup* and the configuration setting determined which covergroups exist.

This quickly becomes a burden as N_PORTS increases and additional requirements are added to each port. The complexity can be seen in the example below where, based on the additional configuration setting on the coverage class, only the appropriate coverage should be constructed. Fig. 18 shows an example of one such SRIO configuration where N_PORTS=3. Port A can only send READS, port B can only send WRITES and port C has to handle the remaining six packet types. When port A is created in the testbench environment, the coverage configuration information will need to indicate READs are supported and disable all the other packet types. Port B will need to enable only WRITE support and disable all other types and port C disables READs and WRITEs with all other types enabled.

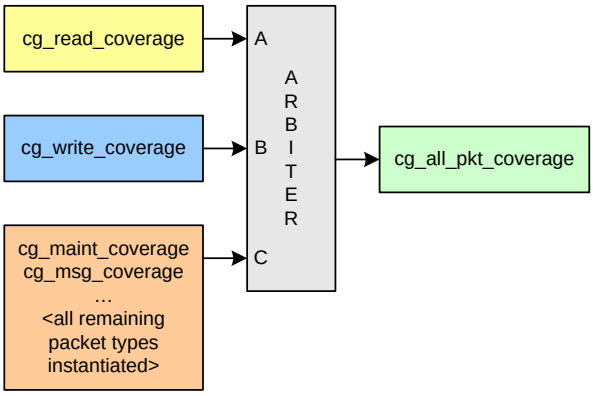

Figure 18. Arbiter with N_PORTS=3

Fig. 19 shows the logic used to create only valid packet coverage based on the configuration settings.

```
// Construct the appropriate covergroups
// within the new function
function new(string        name,
             uvm_component parent);
  ...
  // configured_types will be set for
  // interfaces which disable some packet
  // types. In this case, check what is
  // enabled and only new those covergroups

  if (configured_types) begin
    if (read_support) begin
      cg_read_coverage = new;
    end
    if (write_support) begin
      cg_write_coverage = new;
    end
    ...
    // One check for each packet type

  // If no types are configured, create
  // coverage including all packet types
  end else begin
    cg_all_pkt_coverage = new;
  end
endfunction : new
```

Figure 19. Conditional Covergroup Creation

Originally, the SRIO transaction coverage was ~190 lines of code prior to packet configuration requirements. After parameterization allowing disabled packet types was added, coverage alone was ~600 lines of code. This number was increased even more to configure the coverage class for each port and select only appropriate coverage for a total of ~1200 lines of code.

E. Property Coverage Considerations

Cover directives and assertions (properties), according to SystemVerilog, must be located within a *module* or *interface*. The *module* or *interface* provides direct access to parameters as static values. Although parameter workarounds are not required, additional work is needed up front in order to correctly write property coverage. This affects how RTL is written. *Generate*/*if* blocks that exist in parameterized designs must be written correctly to avoid bogus assertions and false coverage.

If a bind file is used, the RTL must be written such that the bind has access to the signals being covered. *Generates* will often be coded with local variables. This is great for coding but hides access to these signals from the bind file because the *generate* statement makes a new scope. The bind is creating a new instance of a *module* or *interface* with the *module* that had the *generate* statements. This means there are now two parallel scopes which can't see each other. If these signals are moved outside the *generate* block and thus made global variables, they are now visible to the bind file and can be covered. Fig. 20 below shows module logic_block with logic_block_bind bound to it. The bind file is able to access the send_ccomp signal located in logic_block. When localized to *generate* ccomp_logic_gen the bind cannot access it since it is in a parallel scope .

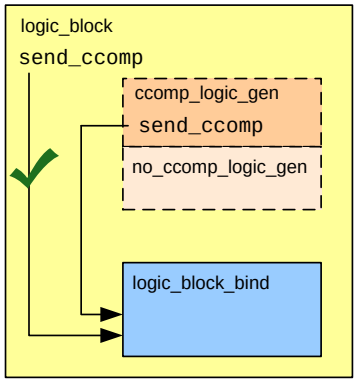

Figure 20. Signal Access Allowed From A Bind File

Care must be taken to ensure signals are not accidentally covered in an invalid configuration. Undriven signals lead to assertion failures and can trigger false or illegal coverage. This can be fixed by either initializing these signals in the else condition or disabling the cover property based on the generating parameter as done in the examples in Fig. 21 below.

```
// Example 1: Clear undriven signals
// for invalid configurations.
// A clock compensation is only sent if
// CCOMP_EN is 1
wire send_ccomp;

generate if (CCOMP_EN == 1) begin:
                                ccomp_logic_gen
  assign send_ccomp =
                (ccomp_ctr == MAX_CCOMP_CNT);

end else begin: no_ccomp_logic_gen
  // zero out when CCOMP_EN is 0 so the
  // bind file only samples valid values
  // and assertions on this signal do not
  // fire incorrectly
  assign send_ccomp = 0;
end endgenerate

// Example 2: Disable properties for
// invalid configurations.
cover property
  (@(posedge clk) disable iff (!CCOMP_EN)
     (send_ccomp));
```

Figure 21. Global Variables for Coverage

F. Generate Block Considerations

Another problem with generated code is that merging coverage for a signal across all configurations of a design is not always wanted. For example, consider the case where a FIFO has a parameterized depth which can be set to 8 or 32 and a full condition needs to be covered. Covering that the full signal was asserted might not be sufficient because it will be merged whether full is seen with a depth of 8 or 32; it is possible full might have only ever occurred when the depth was 8. If the parameterization affects the implementation, it is important to see both settings to ensure full correct functionality. These decisions must be made by someone familiar with the design architecture. A simple solution for this example is to cross the values of FIFO depth with the full flag set to 1. This is shown below in Fig. 22, which is also an example of replicating parameter coverage simply to provide access to a cross.

```
// cg_fifo_full covers the full signal for
// each valid value of FIFO_DEPTH
covergroup cg_fifo_full;
  cp_fifo_full : coverpoint (fifo_full);

  // Coverage on FIFO_DEPTH is already
  // managed within parameter coverage but
  // is duplicated here to allow for use
  // in the cp_fifo_full cross.
  cp_fifo_depth: coverpoint (FIFO_DEPTH) {
    bins min = {8};
    bins max = {32};
  }

  cross cp_fifo_full, cp_fifo_depth {
    ignore_bins ignore =
       binsof(cp_fifo_full) intersect {0};
  }
endgroup
```

Figure 22. Crossing Parameters with Functional Coverage

Writing properties within *modules* covering parameterized code can be greatly simplified by the use of *generate* statements, which are also available. Note that generated code creates implicit hierarchy. This leads to unique coverage spaces that will not be merged. For busses where each bit needs to be covered independently, only one cover property needs to be written in a *generate* statement that can be reused on all bits. The example in Fig. 23 shows a signal with a width of BUF_DEPTH which indicates free locations in a buffer. Each location needs to be detected as full and empty.

```
// Based on the buffer depth, see each
// buffer location as full and empty
reg [BUF_DEPTH-1:0] free_locations;
...

generate
  for (int ii=0; ii < BUF_DEPTH; ii++) begin
    covergroup cg_location;
      coverpoint (free_locations[ii]);
    endgroup
  end
endgenerate
```

Figure 23. Generate Loop for Per Bit Bus Coverage

Another use case for a *generate* statement is for multidimensional arrays. The SRIO core can have a serial link width generated based on a user's needs. For this, the LINK_WIDTH parameter indicates how many lanes are valid. In this example, each lane needs to maintain a count of how many bit errors were detected on the link and thus a multidimensional array of counters is used. A cover property can be generated for only valid links to cover the error count reaches its max value as done in Fig. 24.

```
// Check the counter reaches its max value
// on each lane.
reg [3:0] error_cnt [LINK_WIDTH-1:0];
...

generate
  for (int ii=0; ii < LINK_WIDTH; ii++) begin
    cover property
      (@(posedge clk)
         (error_cnt[ii] == MAX_ERRORS));

  end
endgenerate
```

Figure 24. Generate Loop for Multi-Dimensional Signal Coverage

For both examples described, it is desired to not merge coverage. By using *generate* loops, hierarchy will be provided and results in not merging the coverage.

III. PARAMETERS & CODE COVERAGE

Normally, code coverage is used as a cross-check for functional coverage to ensure that all coverage points have been defined. Gaps in code coverage without any relevant functional coverage can also identify dead code. As mentioned earlier, while formal can detect this condition early in the design process for standard designs, this is not possible for non-parameterized designs. The only way to locate dead code or unhittable states in a parameterized code base is to run code

coverage and review the output. This makes the code coverage review phase even more critical for a parameterized design.

One additional problem for code coverage is that it is additional functionality provided by the vendors and not specified by the SystemVerilog standard. Vendors are free to specify code coverage as they see fit including any parameterized merging behavior. Users must examine all merged code coverage results in order to ensure that results provide the information they need for their design.

We suggest code coverage tools merge gathered coverage metrics across *generate/for* loops. For most situations, the code's functionality is what is being tested; multiple copies don't provide more information about the correctness of the code. If it is the case that specific behavior matters per loop, embedded functional coverage can be bound in to track this. Note that this requires that *generate* signals be declared as global variables as described above.

For *generate/if* or *generate/case* constructs, we suggest that vendors don't merge the branches, even when the *generate* labels are identical for each branch. Because custom code appears in each branch, every branch needs to be fully exercised in order to ensure that the code is well tested.

IV. Conclusion

Parameters are well integrated into the RTL design subset of SystemVerilog. This enables a clean methodology for designing and maintaining a single code base, while still delivering multiple custom netlists. The addition of keywords like *generate* have eased the burden of designing with parameters.

The solutions are not as straightforward within the verification space of SystemVerilog. The good news is that testbenches can still completely verify these parameterized designs; the bad news is that solutions still require a lot of overhead. The availability of the configuration space in OVM/UVM environments has simplified parameter passing and access. Using a container object to collect all parameter values makes it easy to access these values anywhere in the design, including functional coverage groups.

However, there is still work to be done, particularly for functional coverage, where handling variable code causes additional overhead, as shown previously. When verifying a parameterized design, coverage constructs grow in size. Additionally, work must be duplicated in order to correctly ensure that every variant is covered within a configurable environment.

New additions to the SystemVerilog standard will greatly help with the problem. Until then, the tips and techniques outlined above will reduce the headache from a migraine to a minor annoyance.

Acknowledgments

We would like to thank Adam Rose from Mentor Graphics for some excellent technical guidance. We would also like to thank Geoff Koch for helping to edit this paper.

References

[1] IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, IEEE Std. 1800-2009.

[2] RapidIO™ Interconnect Specification Revision 2.2, 2011, www.rapidio.org

[3] B. Ramirez, M. Horn, OVM & Parameters: Why Can't They Just Get Along?, http://go.mentor.com/parameters_and_ovm

[4] Mantis 2506, http://www.eda.org/svdb/view.php?id=2506

[5] D. Rich, A. Erickson, Using Parameterized Classes and Factories: The Yin and Yang of Object-Oriented Verification, http://go.mentor.com/yin-and-yang

[6] Open Verification Methodology (OVM), http://ovmworld.org

[7] Universal Verification Methodology (UVM), http://www.accellera.org/activities/committees/vip/