Method for Generating Unique Coverage Classes to Enable Meaningful Covergroup Merges Across Testbenches

Eldon Nelson, MS, PE (eldon_nelson@ieee.org)
Verification Engineer, Micron Technology, Inc.

***Abstract*- The SystemVerilog covergroup is a valuable tool for determining what conditions have been met in a design. Coverage-driven testplans support two reference possibilities when using covergroups to satisfy verification requirements: the covergroup type or the specific covergroup instance. Some coverage-driven testplans may even enable a pattern of covergroup instances to be listed. Unfortunately, covergroup type and covergroup instance references each have their own limitations that are only amplified when more than one testbench is part of the coverage-driven testplan. This paper provides an improved method of implementing embedded covergroups by adding capability that can better meet verification requirements.**

I. Introduction

A SystemVerilog covergroup can be defined by either including it or importing it into a component. In the first case, the covergroup will have a type path in the simulator hierarchy that is equal to that of the verification component. The type path will be different for each testbench, thus preventing a merge from occurring. This paper defines this type of covergroup as a simple covergroup.

If the covergroup is defined outside of the verification component through a coverage class and made available with a SystemVerilog import, a different problem arises. The covergroup type becomes static but may not provide the required specificity to meet certain verification requirements. This paper defines this type of covergroup as an embedded covergroup [1]. A covergroup type with contributions from many unrelated covergroup instances yields a merge with little value.

Solving the covergroup problem requires determining what combination of covergroup instances across multiple verification environments creates a useful merged result. One possible solution is to perform manual pattern matching of covergroups outside the simulator and code base in the coverage-driven testplan to meet the verification requirements, but this method is not optimal because it is prone to mistakes and fragile to design changes. The verification environment should generate these relevant covergroup types.

Source Code

The ideas presented in this paper are demonstrated through the use of example verification environments. To run the examples and to see the full context of the file excerpts presented, the source code may be obtained from a Git repository hosted on GitHub [2] at the URL provided below. No login is necessary and the source may be viewed with a standard web browser. The website includes directions on how to run the examples in a SystemVerilog simulator. The examples have been tested with Mentor Questa and Cadence Incisive. The source code is released under the GNU General Public License Version 2 [3].

```
https://github.com/tenthousandfailures/uniquecoverage
```

DUT Design

In order to explain the results and approach, a description of the example environment is needed. The testbench is a collection of first-in first-out (FIFO) designs under test (DUTs) that receive commands and forward them to another identical DUT. The entire DUT source is shown in Figure 1. The testbenches vary in terms of the scope of where in this FIFO chain the testbench starts and ends, but they include the same components and connections that are in that scope. This example is used to mimic what happens in verification environments. Small testbenches

encapsulate a portion of the DUT and then larger and larger collections of DUT components are included in progressively higher-order testbenches.

```
module dut (
            dut_if.slave slave,  // input
            dut_if.master master // output
            );

    // assign inputs to outputs
    always @(posedge slave.clk) begin
        master.cmd  <= slave.cmd;
        master.adr  <= slave.adr;
        master.data <= slave.data;
    end
endmodule
```

FIGURE 1. DUT.SV

TESTBENCH DESIGN

Figure 2, shows testbench zero (`TB0`), which includes three copies of the module `dut` named `duta`, `dutb`, and `dutc`. These modules are connected by SystemVerilog interface instantiations of `dut_if` named with suffixes that represent its connections. For example, `dut_if_t_a` is a connection from the testbench (t) to `duta` (a).

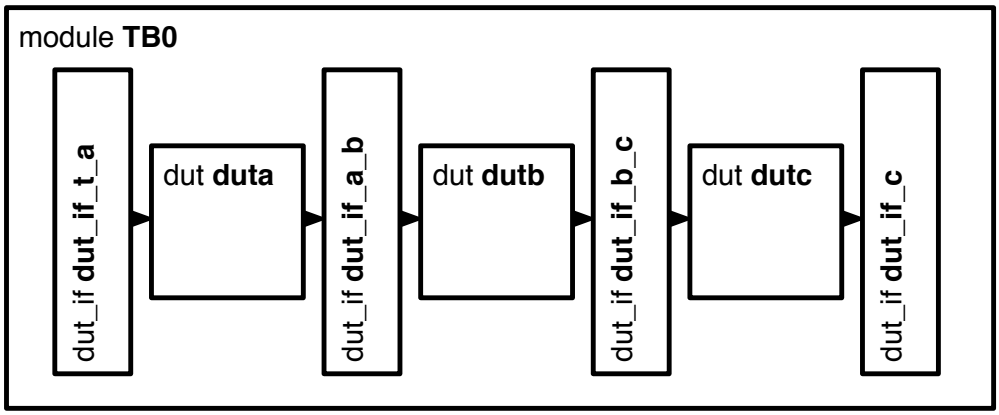

FIGURE 2. TESTBENCH TB0

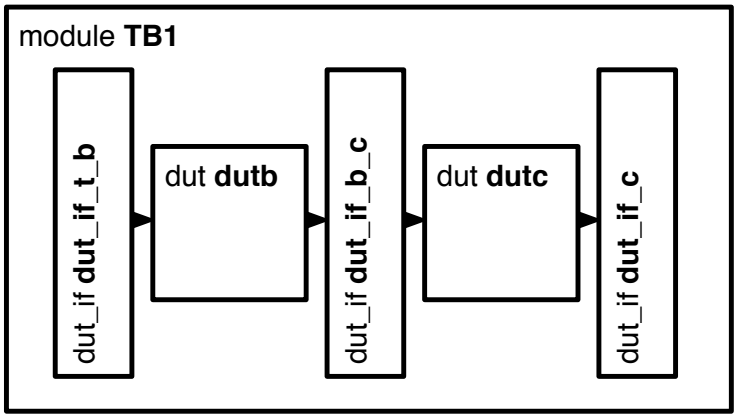

FIGURE 3. TESTBENCH TB1

Figure 3 shows `TB1`, which is a subset of `TB0` and includes `dutb` and `dutc`. `dut_if_t_b` connects testbench (t) to `dutb` (b).

COVERGROUP REFERENCES

A covergroup needs a purpose to give it meaning and value. A coverage-driven testplan gives the covergroup purpose. A coverage-driven testplan is limited to two possible methods of referencing a covergroup: instance or type. Choosing the appropriate reference to use in the testplan makes a significant difference in what the covergroup's coverage will mean. In order to explain the difference and the proposed improvement to these reference methods, a comparison of instance coverage versus type coverage is provided in the following section.

COVERGROUP INSTANCES AND THE SIMPLE COVERGROUP

Instance coverage will give coverage of a particular covergroup in a single environment. Instance coverage is defined by the hierarchal path to where the covergroup was instantiated plus the instance name. If only one testbench is being used, a great degree of specificity is achieved by being able to specify one particular covergroup instance. It is exact and it is simple.

The covergroup `dut_if_cg`, shown in Figure 4, has three inputs and has default settings for how coverage should be merged. Contained within the covergroup is a single coverpoint with four possible bins in the `cmd` field. This covergroup definition is used throughout all the examples in this paper and is included (`` `include ``) in the source when the covergroup is required.

```
covergroup dut_if_cg (ref logic [3:0] cmd,
                      input string    inst_name,
                      input string    comment);

    type_option.merge_instances = 1;

    option.name = {name, ".", inst_name};
    option.per_instance = 1;

    coverpoint cmd {
        bins _null  = {'h0};
        bins _read  = {'h1};
        bins _write = {'h2};
        bins _cfg   = {'h3};
    }

endgroup
```

FIGURE 4. DUT_IF_CG.SVH

The covergroup `dut_if_cg` is included in the `dut_if` interface, as shown in the initial block in Figure 5. This pattern is a simple covergroup because the covergroup is defined and instantiated within the component that will be sampling it. When this covergroup is instantiated with new, the type path for this covergroup is defined to have the same path as the component it was defined in. In this example, the type path is the hierarchical path to `dut_if` within the design. When the simulation is run, the example covergroup is visible in Mentor Questa from the covergroup window [4], as shown in Figure 6.

```
interface dut_if (input logic clk);

    string      name = "simple";

    logic [3:0] adr, cmd, data;
    `include "dut_if_cg.svh" // included covergroup definition

    dut_if_cg simple_inst;   // simple covergroup

    always @(posedge clk) begin
        simple_inst.sample();
    end

    initial begin
        $sformat(inst_name, "%m");

        simple_inst = new(.cmd(cmd),
                          .inst_name(inst_name),
                          .comment(simple_comment)
                          );
    end

endinterface
```

FIGURE 5. DUT_IF.SV

Every simulator has a different way of showing the active covergroups within a design. It is useful to analyze the graphic that is based on the Mentor Questa output (Figure 6) from this example. The first line `/TB0/dut_if_c` is the hierarchical path of the instantiated covergroup. Within the simulator, the type path for this covergroup is the hierarchical path plus the TYPE:

```
/TB0/dut_if_c/dut_if_cg
```

which is an important property of the covergroup. The type path determines what can be merged with this covergroup. In order to merge a covergroup with this one, the corresponding covergroup must have the exact same type path. A simple covergroup can never be merged with an instance of a covergroup other than its own because its type path is specific to its component.

▼ Name	Class Type	Coverage
/TB0/dut_if_c		
TYPE dut_if_cg		50.0%
CVP dut_if_cg::cmd		50.0%
bin _null		1
bin _read		0
bin _write		0
bin _cfg		96
INST simple.TB0_dut_if_c		50.0%
bin _null		1
bin _read		0
bin _write		0
bin _cfg		96

FIGURE 6. SIMPLE COVERGROUP

The coverpoint (`CVP`) is defined within the covergroup. `INST` stands for instance and defines the name and details of the instance coverage. The name of this instance is custom-defined for clarity by defining the `option.name` attribute of the covergroup. The coverage column defines the number of hits each bin had or the percentage coverage for the line.

References to a specific covergroup occur within the coverage-driven testplan. Table 1 shows an example of a coverage-driven testplan in the Mentor Questa XML [5] format. The value for the "Link" field is the type path to the covergroup, followed by a comma, and then followed by the instance name.

TABLE 1. TESTPLAN FOR SIMPLE COVERGROUP USING INSTANCE COVERAGE

Section	Title	Link	Type
1	Simple Covergroup		
1.1	cmd coverpoint for interface _c on TB0	/TB0/dut_if_c/dut_if_cg,simple.TB0.dut_if_c:cmd	CoverPoint
1.2	cmd coverpoint for interface _c on TB1	/TB1/dut_if_c/dut_if_cg,simple.TB1.dut_if_c:cmd	CoverPoint
1.3	cmd coverpoint for all interfaces of _c with wildcard	/TB*/dut_if_c/dut_if_cg,simple.TB*.dut_if_c:cmd	CoverPoint

Specifying a covergroup instance may meet a verification requirement. However, if there are multiple verification environments, a single covergroup instance may not be sufficient. If, for example, the requirement is that this covergroup must reach a certain level of completeness in every verification environment, the testplan line must be repeated with new covergroup instance paths and instance names for each verification environment. Repeating lines in a testplan goes against the don't-repeat-yourself (DRY) [6] principle of software engineering. If a line is repeated in the testplan multiple times only because the instance paths are changing for each verification environment, something is wrong. Another solution for the "Link" field in the coverage-driven testplan is to use wildcard (*) pattern matching, which can work successfully in some applications, but it requires that certain naming structures are always followed, making this approach fragile to design changes.

COVERGROUP TYPE AND THE EMBEDDED COVERGROUP

Another way to reference a covergroup is by the covergroup type. The covergroup type is defined by where the covergroup is declared. A covergroup type reference makes the most sense when the covergroup is defined within a class. Figure 7 shows an example from the source that defines an embedded covergroup. The class is defined within a package so it can be imported into the necessary components. Figure 8 shows how the interface instantiates the embedded covergroup.

```
package emb_pkg;

class cov;

     string      name = "emb_pkg";

    `include "dut_if_cg.svh" // included covergroup definition

    function new(ref logic [3:0] cmd, ref string inst_name);
        dut_if_cg = new(.cmd(cmd),
                        .inst_name(inst_name),
                        .comment("embedded covergroup")
                        );
    endfunction

    function void sample();
        dut_if_cg.sample();
    endfunction

endclass

endpackage
```

FIGURE 7. EMB_PKG.SV

```
interface dut_if (input logic clk);

    string inst_name = "";
    logic [3:0] cmd, adr, data;

    emb_pkg::cov emb_inst; // embedded covergroup

    always @(posedge clk) begin
        emb_inst.sample();
    end

    initial begin
        $sformat(inst_name, "%m");

        emb_inst = new(.cmd(cmd),
                       .inst_name(inst_name)
                       );
    end

endinterface
```

FIGURE 8. DUT_IF.SV

Figure 9 shows the results of the embedded covergroup simulation in the Mentor Questa covergroup viewer. Unlike the simple covergroup results, the "Class Type" field contains a value in the embedded covergroup results. The simple covergroups had no class to contain the covergroup. The type path is freed from the component hierarchy with embedded covergroups. The type path is based off of the package and class, which is ideal because it enables new merging possibilities. Figure 9 shows that the `/emb_pkg/cov` covergroup type has contributions from nine interfaces across three testbenches. While no single testbench or interface can reach more than fifty percent, together, the covergroup type reaches one hundred percent because the union-merged contributions meet the requirements of the covergroup type's bins.

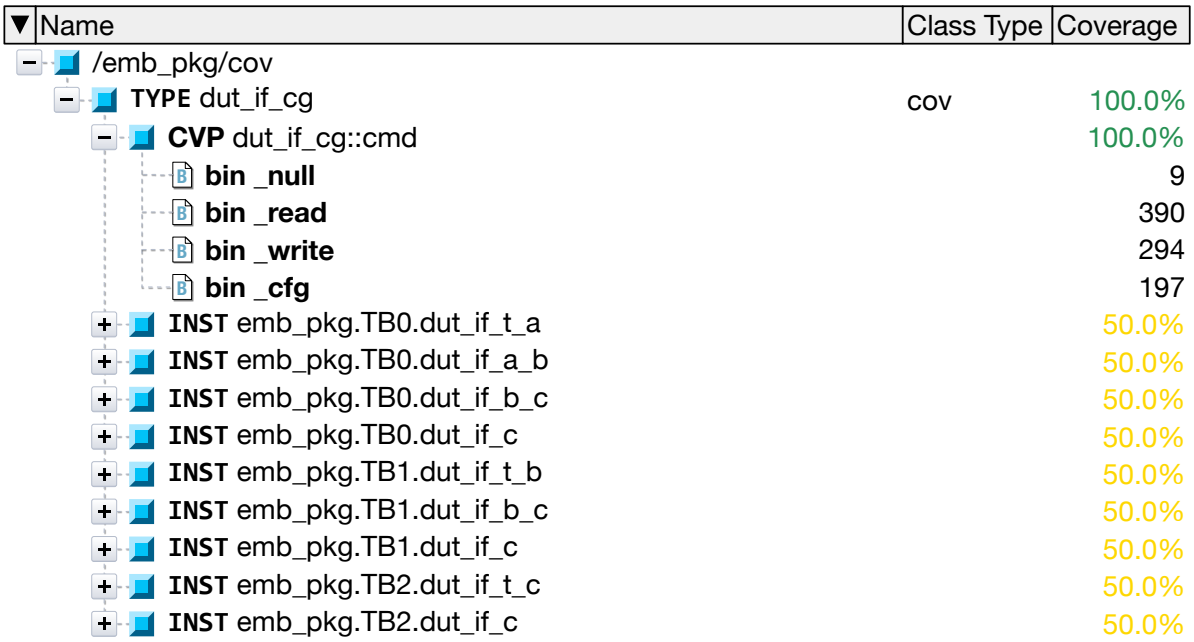

FIGURE 9 EMBEDDED COVERGROUP

Instance coverage is still available when using embedded covergroups if the covergroup `option.get_int_coverage` attribute is true. The simple embedded covergroup was only able to show contributions to its type from one instance, but since the type path is identical for this embedded covergroup all of the instances of the embedded covergroup can be merged which is why all of the instances are shown. The coverage-driven testplan entry containing this covergroup is shown in Table 2. The "Link" field format includes the covergroup type followed by the coverpoint name.

TABLE 2. TESTPLAN FOR EMBEDDED COVERGROUP USING TYPE COVERAGE

Section	Title	Link	Type
2	Embedded Covergroup Method		
2.1	cmd coverpoint for covergroup type /emb_pkg/cov	/emb_pkg/cov/dut_if_cg:cmd	CoverPoint

The covergroup type can help efficiently prove that a certain condition is met with weighted-average merges or with union-merged contributions from several instances. By changing the value of `type_option.merge_instances` in the covergroup, its meaning is easily changed between weighted-average merge and union merge. In some testplan tools, wildcards can be used to meet the weighted-average merge requirement with simple covergroups. However, a union merged requirement cannot be met with wildcards. Wildcard matching of simple covergroups cannot be used to perform a union merge of covergroups.

Using the covergroup type reference in the coverage-driven testplan is appropriate if the requirement involves every instance of that covergroup across every possible verification environment. However, if the covergroup type is used too broadly or encompasses instances that don't make sense, it may not be specific enough.

SOLUTION

The covergroup instance and covergroup type references both have useful applications in coverage-driven testplans; however, these covergroup references have drawbacks as well. The absolute path of a covergroup instance implemented with a simple covergroup is inflexible and invites replication into the coverage-driven testplan. The global nature of covergroup types implemented with embedded covergroups is sometimes too broad to capture exactly what is needed. This paper proposes implementing a covergroup type with unique embedded

covergroups that can be specific enough to meet verification requirements and provide additional hierarchy that can be used to create more powerful covergroup structures.

II. Methodology

The proposed solution is to create unique covergroup types that encapsulate particular verification goals. Figure 10 uses a partial unified modeling language (UML) [7] class diagram to illustrate the proposal. The package `uniq_pkg` is created to hold class definitions whose ultimate parent is an abstract (`virtual` in terms of SystemVerilog) class that defines common variables and the functions `new` and `sample`.

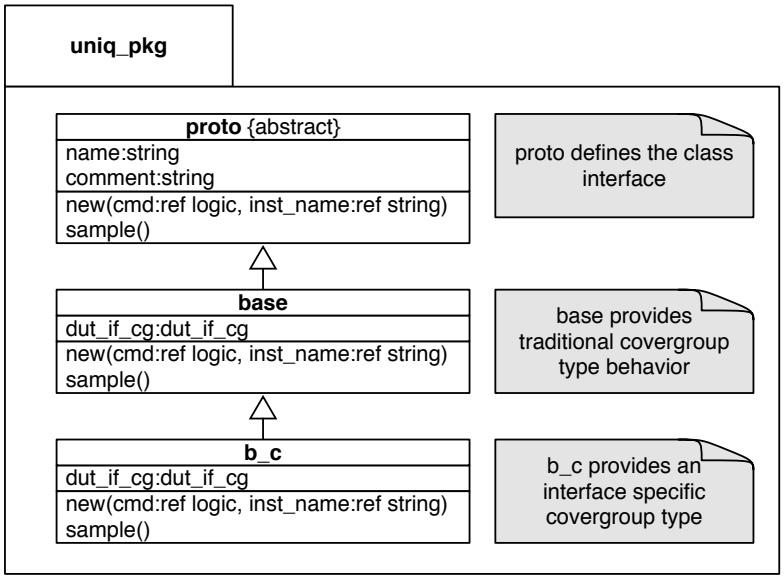

Figure 10. Partial UML Class Diagram Solution

The base class is the first concrete class, which provides the same functionality as embedded covergroups. This feature is useful because it makes the application of this method a superset of the existing and already accepted embedded covergroup. The child class `b_c` represents internal communication between `dutb` and `dutc` that exists in both `TB0` and `TB1`. The `b_c` class can represent a verification requirement that dictates that a combination of, or exhaustively, the communication between `dutb` and `dutc` must be exercised to a certain goal across all testbenches.

UML Class Diagram of Solution

A unique covergroup type allows for merging of covergroups from different testbenches that exist in the present and also ones that will come to exist in the future. The covergroup type denoted by the type path: `/uniq_pkg/b_c` is available to be sampled and the meaning of this covergroup type is known and specific. It is a specific container to hold coverage from a specific physical interface across all testbenches.

The simplified UML class diagram, shown in Figure 11, matches what is implemented in the source example. Its six separate covergroup types each have their own meaning. Each covergroup type can be added to a testplan to satisfy the particular requirements of the verification environment and operation of the DUT under external stimulus. For example, if the requirement is that the `t_c` (testbench to `dut_c`) coverage is set to a particular goal, the `/uniq_pkg/t_c` covergroup type can be used. If the requirement is to observe coverage between `dutb` and `dutc,` the covergroup type `/uniq_pkg/b_c` can be used and will have contributions from any testbenches with this connection present.

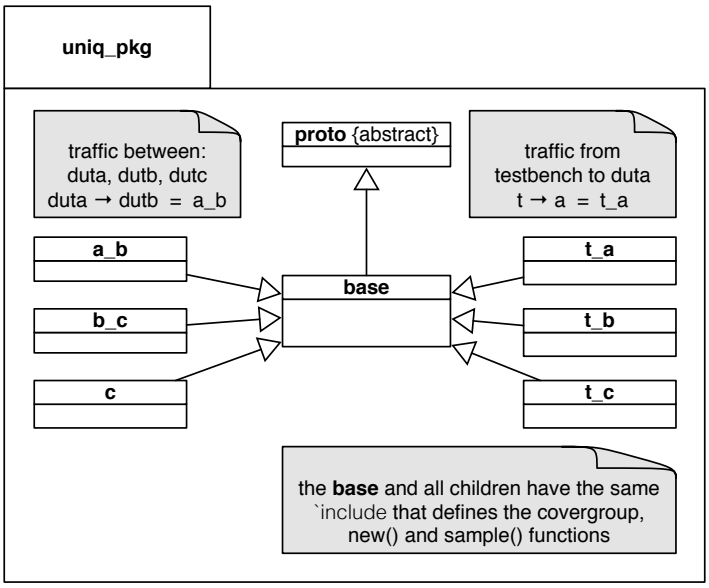

FIGURE 11. COMPLETE UML CLASS DIAGRAM SOLUTION

Figure 11 shows one example of a UML class diagram; additional levels of children can be added if further categorization is needed to match verification requirements. Even non-symmetrical levels of hierarchy could be added to encompass groups of covergroup instances that together yield a meaningful result.

CLASS IMPLEMENTATION OF SOLUTION

The SystemVerilog code template shown in Figure 12 is included (`include) in the unique embedded covergroups. This template defines the new function that calls super.new and also instantiates the covergroup dut_if_cg. It also defines the sample function that samples the parent class as well as the covergroup defined in the local class itself. Sampling the parent is important because a child can trigger samples in all of its parents. This provides compatibility with embedded covergroup behavior and enables aggregate parent covergroup types to be sampled as well.

It is useful to note that the class passes the sampled arguments to the covergroup by reference. If an argument is not passed in by reference to a covergroup, it is only sampled with the value given to it when it is created with the new operator [8]. Therefore, passing by reference is crucial to passing live values into the covergroup.

```
function new(ref logic [3:0] cmd, ref string inst_name);
    super.new(.cmd(cmd),
              .inst_name(inst_name)
              );
    dut_if_cg = new(.cmd(cmd),
                    .inst_name(inst_name),
                    .comment(comment)
                    );

endfunction

function void sample();
    super.sample();
    dut_if_cg.sample();
endfunction
```

FIGURE 12. UNIQ_PKG_FCN.SVH

```
package uniq_pkg;

    string pkg_prefix = "uniq_pkg::";

virtual class proto;

    string name = "proto";
    string comment = "proto comment";

    function new(ref logic [3:0] cmd, ref string inst_name);
    endfunction

    function void sample();
    endfunction

endclass

class base extends proto;

    string name = {pkg_prefix, "base"};
    string comment = "uniq::base default comment";

   `include "dut_if_cg.svh"
   `include "uniq_pkg_fcn.svh"

endclass

class b_c extends base;

    string name = {pkg_prefix, "b_c"};
    string comment = "b_c comment";

    `include "dut_if_cg.svh"
    `include "uniq_pkg_fcn.svh"

endclass

endpackage
```

FIGURE 13. UNIQ_PKG.SV

The SystemVerilog shown in Figure 13 implements the partial UML class diagram of Figure 10. As shown in the embedded covergroup and simple covergroup, the covergroup definition itself is defined once in `dut_if_cg.svh` (Figure 4) and included only when needed (`` `include "dut_if_cg.svh" ``), replicating the covergroup definition in each class. As children of the abstract class `proto` are defined, each concrete child includes the covergroup definition in the class, enabling the creation a new covergroup type. This is done because the type path is created for a covergroup from where the covergroup is defined and not where it is instantiated. The `` `include `` of the functions for `new` and `sample` are required because if these functions are not overwritten at each child class, it would instead call the parent version of those commands and not sample or instantiate the embedded covergroup within the local class.

INTERFACE DUT_IF

Figure 14 shows the parameterized SystemVerilog interface `dut_if`. For simplicity in this example, the type passed in is called `T`; however, in a production environment, a more descriptive name can be used. This parameterization is an important distinction between an embedded covergroup and the proposed unique embedded covergroup. An embedded covergroup would be statically instantiated in every instance of the interface. However, in this interface, a unique embedded covergroup is parameterized in, which enables the interface to be customized in terms of what covergroup and what parent covergroups are sampled.

```
interface dut_if #(type T = uniq_pkg::base) (input logic clk);

    string inst_name = "";
    logic [3:0] cmd, adr, data;

    T cov_inst; // parameterized unique class containing covergroup

    always @(posedge clk) begin
        cov_inst.sample();
    end

    initial begin
        $sformat(inst_name, "%m");
        cov_inst = new(.cmd(cmd),
                       .inst_name(inst_name)
                       );
    end

endinterface
```

FIGURE 14. DUT_IF.SV

TESTBENCH CONNECTIONS

Figure 15 shows the implementation of testbench `TB0`. SystemVerilog interfaces of `dut_if` connect the design components together in a way that is familiar to the typical application of interfaces. The salient feature of this testbench is that the interfaces are parameterized with context to what they will be connecting. For example, the interface instantiation of `dut_if_b_c` is given a unique embedded covergroup of `uniq_pkg::b_c,` which determines which series of covergroup types define the `dutb` to `dutc` interface.

```
module TB0 ();

    dut_if #(uniq_pkg::t_a) dut_if_t_a(clk);
    dut_if #(uniq_pkg::a_b) dut_if_a_b(clk);
    dut_if #(uniq_pkg::b_c) dut_if_b_c(clk);
    dut_if #(uniq_pkg::c)   dut_if_c(clk);

    dut duta(
             .slave(dut_if_t_a), // input
             .master(dut_if_a_b) // output
             );
    dut dutb(
             .slave(dut_if_a_b),
             .master(dut_if_b_c)
             );
    dut dutc(
             .slave(dut_if_b_c),
             .master(dut_if_c)
             );

endmodule
```

FIGURE 15. TB0.SV

UVM SOLUTION

The previous SystemVerilog solution uses unique embedded covergroups within interfaces that live within a conventional SystemVerilog testbench. Many verification environments today are built on the Universal Verification Methodology (UVM) framework. The UVM implementation in Figure 16 shows a solution using unique embedded covergroups.

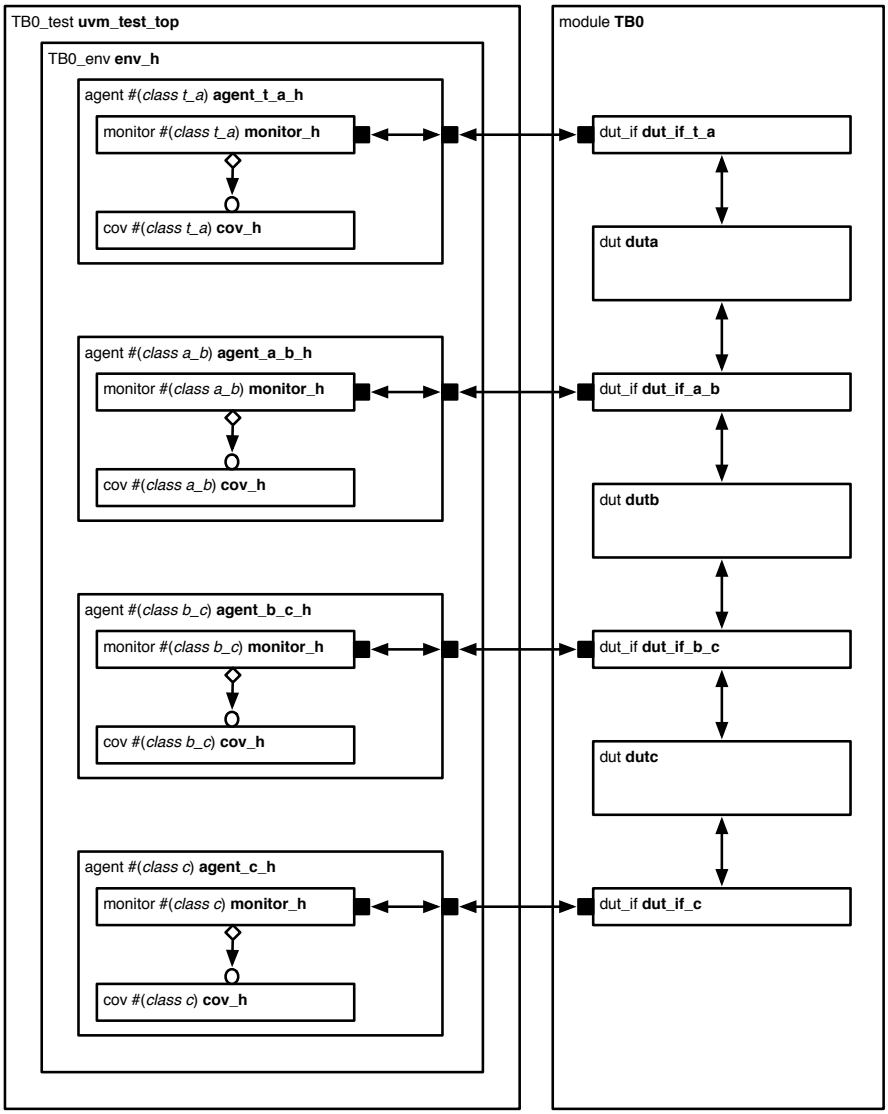

FIGURE 16. UVM SOLUTION DIAGRAM

The block diagram of the UVM verification environment shows how the UVM agent passes down its unique embedded covergroup parameterization to its subcomponents. Because the agent instantiates its children, including the `cov` class, it is necessary to parameterize the UVM agent so that it can pass down the correct parameterization at build time. Where the coverage objects are instantiated in a UVM environment varies based on the verification requirements. In this case, the goal is to be able to gather interface coverage between DUT components. Therefore, having the coverage class live within the agent gives context, and UVM hierarchy, to the interface being covered.

UVM Implementation of Solution

Implementing the UVM solution is very similar to the SystemVerilog solution. In the package `TB0_pkg` shown in Figure 17, `typedefs` are defined to make the parameterized agents easier to update in the future. The `_t` suffix is added to signify type. Instead of parameterizing the SystemVerilog interfaces (`dut_if`) with the unique embedded covergroups, like in the SystemVerilog solution, the UVM agents are parameterized.

```
package TB0_pkg;

    import uvm_pkg::*;
    `include "uvm_macros.svh"
    `include "TB_common.svh"

    typedef agent #(uniq_pkg::t_a) agent_t_a_t;
    typedef agent #(uniq_pkg::a_b) agent_a_b_t;
    typedef agent #(uniq_pkg::b_c) agent_b_c_t;
    typedef agent #(uniq_pkg::c)   agent_c_t;

    `include "TB0_env.sv"
    `include "TB0_test.sv"

endpackage
```

Figure 17. TB0_pkg.sv

The agents are instantiated during the build phase from these types and are given handles to their interfaces in the DUT in the UVM environment `TB0_env`, as shown in Figure 18.

```
class TB0_env extends uvm_env;
    `uvm_component_utils(TB0_env)

    dut_if_t_a_t dut_if_t_a;
    dut_if_a_b_t dut_if_a_b;
    dut_if_b_c_t dut_if_b_c;
    dut_if_c_t   dut_if_c;

    agent_t_a_t agent_t_a_h;
    agent_a_b_t agent_a_b_h;
    agent_b_c_t agent_b_c_h;
    agent_c_t   agent_c_h;

    function void build_phase(uvm_phase phase);

        agent_t_a_h = agent_t_a_t::type_id::create("agent_t_a_h", this);
        agent_a_b_h = agent_a_b_t::type_id::create("agent_a_b_h", this);
        agent_b_c_h = agent_b_c_t::type_id::create("agent_b_c_h", this);
        agent_c_h   = agent_c_t::type_id::create("agent_c_h", this);

        agent_t_a_h.intf = dut_if_t_a;
        agent_a_b_h.intf = dut_if_a_b;
        agent_b_c_h.intf = dut_if_b_c;
        agent_c_h.intf   = dut_if_c;

    endfunction

endclass
```

Figure 18. TB0_env.sv

III. Results

The proposal is to improve embedded covergroups by adding context that makes the covergroup type meaningful and unique. Just as the capabilities of embedded covergroups are an improvement compared to simple covergroups, this too is an incremental improvement. Applying this method preserves the same functionality of embedded covergroups. Unique embedded covergroups are, therefore, a superset of embedded covergroups.

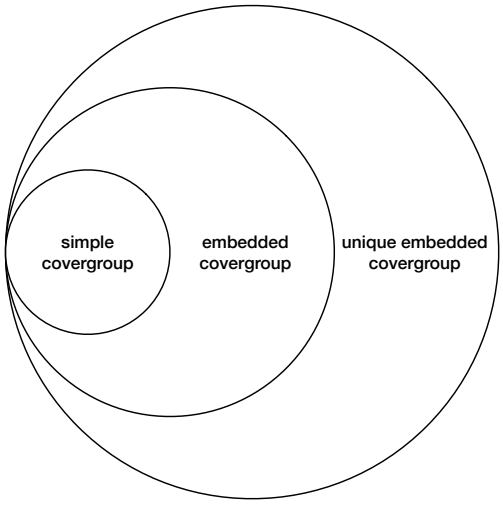

Figure 19. Progression of Features through Covergroup Methods

The example code used in this paper implements all three methods of gathering coverage simultaneously, which enables a comparison of the results and a clear illustration of the methods available to implement covergroups. The proposed unique embedded covergroup has the same capabilities as embedded covergroups, as shown in Figure 20; an embedded covergroup with its singular class `cov` is shown on the left, and the proposed unique embedded covergroup showing its `base` class type is shown on the right. The coverage is exactly the same.

Name	Class Type	Coverage
/emb_pkg/cov		
TYPE dut_if_cg	cov	100.0%
CVP dut_if_cg::cmd		100.0%
bin _null		9
bin _read		390
bin _write		294
bin _cfg		197
INST emb_pkg.TB0.dut_if_t_a		50.0%
INST emb_pkg.TB0.dut_if_a_b		50.0%
INST emb_pkg.TB0.dut_if_b_c		50.0%
INST emb_pkg.TB0.dut_if_c		50.0%
INST emb_pkg.TB1.dut_if_t_b		50.0%
INST emb_pkg.TB1.dut_if_b_c		50.0%
INST emb_pkg.TB1.dut_if_c		50.0%
INST emb_pkg.TB2.dut_if_t_c		50.0%
INST emb_pkg.TB2.dut_if_c		50.0%

Name	Class Type	Coverage
/uniq_pkg/cov		
TYPE dut_if_cg	base	100.0%
CVP dut_if_cg::cmd		100.0%
bin _null		9
bin _read		390
bin _write		294
bin _cfg		197
INST uniq_pkg::base.dut_if_t_a		50.0%
INST uniq_pkg::base.TB0.dut_if_a_b		50.0%
INST uniq_pkg::base.TB0.dut_if_b_c		50.0%
INST uniq_pkg::base.TB0.dut_if_c		50.0%
INST uniq_pkg::base.TB1.dut_if_t_b		50.0%
INST uniq_pkg::base.TB1.dut_if_b_c		50.0%
INST uniq_pkg::base.TB1.dut_if_c		50.0%
INST uniq_pkg::base.TB2.dut_if_t_c		50.0%
INST uniq_pkg::base.TB2.dut_if_c		50.0%

Figure 20. Comparison of Embedded Covergroup
versus Unique Embedded Covergroup

Having unique embedded covergroups inherit from their `base` class replicates the functionality of embedded covergroups. This feature is useful for providing legacy functionality that SystemVerilog users are accustomed to and is a useful data point in itself. Figure 20 also demonstrates the problem with embedded covergroups: the lack of specificity. The comparison shown in Figure 20 includes contributions from nine interfaces across three testbenches, yet, the result is a union-merged result across all of them. If a verification requirement is to know what is the union-merged coverage between `dutb` and `dutc` from all testbenches, this information cannot be easily determined from a single covergroup.

Applying this proposed method results in unique covergroup types that can answer specific questions. Figure 21 is created from the identical simulations but shows the combined coverage between `dutb` and `dutc`, which is available at the same time as the `base` coverage and uses the same shared definition of the covergroup. The "Class Type" can be seen as `b_c`, which represents the coverage between `dutb` and `dutc`. There were two testbenches that had a DUT connection between `dutb` and `dutc`, which is shown in Figure 21.

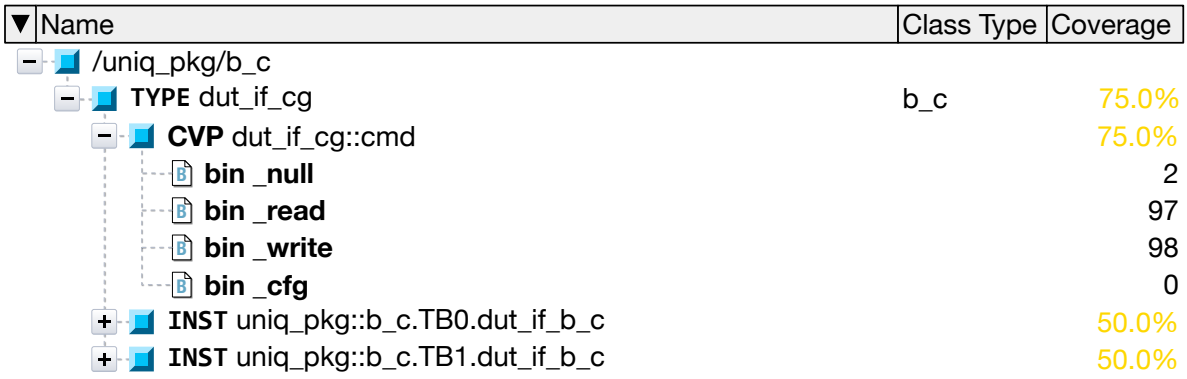

FIGURE 21. UNIQUE EMBEDDED COVERGROUP AND INTERFACE SPECIFIC COVERAGE B_C

As shown in Table 3, the testplan entry for this covergroup reference is simple and works across all verification environments. Because the unique covergroup types can be created so quickly and even defined before the components exist, they can be linked into a testplan early, freeing the testplan from dependence on covergroup instances that rely on hierarchical path names. No wildcards or changes to the "Link" field depending on which testbench is used are necessary; this is an incredible improvement compared to simple covergroups. Unique embedded covergroups provide specificity that embedded covergroups cannot offer.

TABLE 3. TESTPLAN FOR UNIQUE EMBEDDED COVERGROUP

Section	Title	Link	Type
3	Proposed Unique Embedded Covergroup		
3.1	Coverage between `dutb` and `dutc`	/uniq_pkg/b_c/dut_if_cg:cmd	CoverPoint

IV. SUMMARY

Using covergroup instances or covergroup type references has its tradeoffs. A lingering issue in either case is that only certain covergroups can be merged depending on the covergroup type path. This paper proposes a method to improve embedded covergroups by overcoming the limitations of covergroup instances and covergroup type references.

UML class diagrams show how to layer covergroups into a class structure that gives context to each child covergroup. The embedded covergroup behavior remains in the `base` class of the implementation while children of the `base` class instantiate their own copies of the included covergroup definition. This paper describes how to realize the proposal both in SystemVerilog and UVM in order to demonstrate the idea with common verification environments. The full source code of the examples is provided, and important sections of the source implementation decisions are explained.

Each method of using covergroups expands on its parent's functionality, from simple covergroups, to embedded covergroups, and onto unique embedded covergroups. The enhancements to embedded covergroups allow for all of the standard behavior that was present before, but also adds the ability to merge covergroups across testbenches that are specific to a particular instantiated interface or component, the ability to sample multiple covergroups simultaneously in a hierarchical way depending on their context, and a consistent structure that makes it easier to reference unique covergroups by type in coverage-driven testplans without the use of wildcards.

The added abilities of unique embedded covergroups is a compelling improvement to embedded covergroups. The ability to be more specific about how covergroups are merged using type coverage provided by unique embedded covergroups is a valuable verification tool. Coverage-driven testplans crave type coverage because of the

independence they provide without having to deal with design hierarchy. Unique embedded covergroups can provide useful and specific covergroup type references earlier in the design cycle, enabling coverage metrics to be tracked through every verification environment.

V. Future Work

A SystemVerilog macro can be created to reduce replication in the uniq_pkg.sv file (Figure 13); the macro is not discussed in this paper to simplify the examples.

It might seem that the concrete classes could be parameterized in uniq_pkg.sv file (Figure 13) since then each class would then be unique and the same benefit could be reached as achieved in unique embedded covergroups. Unfortunately with current tools parameterized classes are not suitable for use in coverage-driven testplans. The coverage-driven testplan needs a name of a covergroup for its reference, but a parameterized class name is unpredictable. In the simulator each parameterized class will have its own class name that is currently an internal auto-numbered one that is not related to the parameters used to create it. Not being able to predict the class name in the coverage-driven testplan for parameterized classes forces the use of defined non-parameterized classes.

Acknowledgment

Thanks to Mentor Application Engineer Josh Rensch for encouragement and feedback on the paper and Mentor Technical Writer Geoff Koch for editing help. Thanks to Cadence Application Engineer Brent Carlson for bringing up the example design with the Cadence toolset. Thanks to my employer Micron Technology for sponsoring my attendance at DVCon to share this paper.

References

1 "IEEE Standard for System Verilog: IEEE Std 1800-2012, Using covergroup in classes" IEEE Computer Society and IEEE Standards Association Corporate Advisory Group. Web. 21 Feb. 2013.

2. "Build Software Better, Together." Github. Web. 14 Dec. 2014.

3. "GNU General Public License, version 2." Free Software Foundation. Web. 14 Dec. 2014.

4. "Questa Sim User's Manual." Mentor. 2013.

5. "Questa Verification Management User Guide." Mentor. 2013

6. Hunt, Andy and Thomas, Dave. *The Pragmatic Programmer*. Indianapolis, IN: Addison-Wesley Professional. 1999.

7. "Unified Modeling Language." Wikipedia. Web. 14 Dec. 2014.

8. "IEEE Standard for System Verilog: IEEE Std 1800-2012, Defining the coverage model: covergroup" IEEE Computer Society and IEEE Standards Association Corporate Advisory Group. Web. 21 Feb. 2013.