Effective SystemVerilog Functional Coverage: design and coding recommendations

Jonathan Bromley[1], Mark Litterick[2]

(1) Verilab Ltd, Oxford, England

(2) Verilab GmbH, Munich, Germany

www.verilab.com

ABSTRACT

This paper gives practical recommendations from our experience of design, planning and coding of SystemVerilog functional coverage on many projects. Its guiding principles are that coverage should yield valuable and accurate information, and should be both configurable and reusable.

The design and coding of SystemVerilog covergroups can be laborious and error-prone, partly because of the inherent difficulty of the task, but also because the language's features have traditionally provided limited support for configurable and reusable coverage. Fortunately, support is now available in VCS and other simulators for many of the interesting coverage features introduced in the 2012 revision of SystemVerilog. We give examples, guidelines and recommendations showing how these features can make your code more concise, expressive and versatile. The Universal Verification Methodology brings its own special considerations, so the paper also offers specific coding patterns for configurable and reusable coverage within UVM testbench classes.

Table of Contents

Table of Code Examples

1. Introduction

Our experience on many client projects suggests that the design and coding of functional coverage is often troublesome. Furthermore, there are some specific coding concerns that repeatedly seem to cause trouble. In this paper aimed at intermediate-to-advanced SystemVerilog users, we combine a discussion of good coverage design with specific guidelines for the coding of configurable and reusable covergroups. Particular attention is given to effective use of the powerful new language features introduced in the 2012 revision of the SystemVerilog standard. These features have only recently begun to enjoy wide tool support, and therefore there is a shortage of good examples and other guidelines. The paper also includes specific guidance on writing configurable and reusable coverage in a UVM environment, where the specific requirements of UVM component classes give rise to some interesting challenges.

2. The Problem of Coverage Quality

Functional coverage is widely (and correctly) understood to be a cornerstone of effective constrained-random verification. Despite this broad agreement, our experience on real projects suggests that verification teams often underestimate the challenge of providing meaningful, rigorous and sufficient functional coverage in their verification environment. For the sake of clarity, and to avoid excessive dependence on a specific application, introductory texts on verification typically use examples based on stimulus or protocol coverage. Such coverage is comparatively easy to define, since the set of available inputs to a DUT or the set of possible transactions in a bus protocol are well-specified and somewhat independent of DUT operation. Real projects, though, demand wider-ranging consideration of the DUT's operation to ensure that all specified use cases have been exercised, important interactions among activities on various DUT interfaces have been considered, and there has been sufficient verification of internal DUT behaviors that are known to be of particular interest.

Reference [1] is a rare example of a text that takes a highly rigorous approach to the definition of coverage on more subtle aspects of a DUT's operation. One of the present authors has previously published an extended discussion [2] of common coverage design errors and how they can be avoided. This paper summarizes those concerns, and offers coding patterns that allow flexible and effective implementation of high-quality coverage in SystemVerilog [3] and the UVM [4].

2.1 Steps to Coverage Implementation

Implementing functional coverage in a verification environment is notoriously laborious and demands input from many contributors.

First the required coverage must be defined, usually by manual analysis of functional and architectural specification documents along with expert consideration of the DUT's architecture. RTL implementers are likely to be aware of the relationships between specified activity and the operation of internal blocks, and will be able to suggest important coverage scenarios that are not necessarily evident from a high-level functional spec.

Next, the verification team must identify how to capture the necessary information – an easy task for activity on a DUT interface, but often much more challenging for coverage that captures DUT internal state, or timing relationships across multiple interfaces. At this stage it is also important to identify the triggering and filtering criteria that will be used to determine whether coverage information should or should not be sampled. This is an aspect of coverage that is often overlooked, and we urge teams to consider reference [2] which discusses numerous common errors in sampling strategy that can lead to untruthful, over-optimistic coverage results.

The coverage code itself must then be implemented, typically using SystemVerilog covergroups. Well-known limitations of the SystemVerilog language can make this surprisingly troublesome, especially if the coverage must be tailored to respond to various DUT configurations. The UVM helps considerably with some aspects of this configurability problem – in particular, the factory mechanism makes it easy to replace a base coverage class with a more specialized derived class – but it also introduces some implementation difficulties. We discuss this problem of configurability in more detail later in the paper.

Finally, coverage results must be analyzed and used to measure progress against a verification plan. This paper does not consider that problem, as it is strongly influenced by the team's choice of verification planning and tracking tools.

2.2 Re-use of Existing Coverage Implementation

Verification components, particularly if they are sourced from a reputable third party and support some well-defined interface protocol, are likely to have extensive built-in coverage. While it is useful and convenient, predefined coverage of this kind can never be sufficient. Coverage in a generic verification component, however thorough, cannot capture the relationships among activity on different interfaces that are critical to effective coverage of the DUT's operation.

3. Overview of Recommendations

In subsequent sections we present a number of guidelines, suggestions and coding idioms that have proved to be helpful in our practical experience of numerous projects. This section briefly summarizes their content.

3.1 Structural concerns

Coverage code has a distinct character of its own. It is often quite verbose and highly specialized to a specific DUT, and may even be auto-generated. It should be controllable (enabled, disabled, configured) independently of other parts of the testbench, and it may need connections to many diverse parts of the verification environment in order to gather all the information it needs. All these concerns make it important for coverage code to be kept separate from other verification code such as scoreboards and monitors.

3.2 Favour covergroups over assertion-based (SVA) coverage

Tools generally support an integrated view of coverage including functional (covergroups), assertion-based (SVA) and automatic code coverage. Assertion-based coverage can be very useful in some situations, but cannot appear in classes and is therefore less flexible than covergroup-based coverage. This concern is addressed in section 4.

3.3 Consider your packaging options

Coverage can be packaged or encapsulated in many different ways. Section 5 discusses three basic approaches, all of which can be helpful in some situations: using a subscriber class, extending an existing component, and adding coverage objects by composition.

3.4 Encapsulate coverage that needs widely distributed connections

Some coverage inevitably requires information from many different parts of the verification environment. Providing a system-monitor component to encapsulate such coverage does not help with the ugly problem of multiple connections, but it helps to keep the coverage code separate from the rest of the environment, and eases the problem of crossing coverage with broader system-wide

concerns such as resets, power management and major changes of DUT operating mode. Section 6 describes these concerns in more detail.

3.5 Adopt a consistent methodology for coverage tuning or parameterization

The SystemVerilog language imposes some significant restrictions on the way covergroups interact with their host classes. These restrictions make it a little tricky to design covergroups that adapt automatically to DUT configuration and parameterization, and the UVM's fixed signature for class constructors provides yet another difficulty. Sections 7 to 10 offer coding patterns that provide a consistent and re-usable methodology for dealing with these concerns.

4. Assertion-based Coverage

SVA coverpoints can be very useful, but they have the drawback that they cannot appear in classes, and therefore cannot be configured by typical configuration-object mechanisms. Similarly, SVA coverage cannot be extended using class inheritance and factory mechanisms. Consequently we favour covergroup-based coverage wherever possible.

Despite this recommendation, there are many situations where SVA-based coverage can be valuable. Reference [5] discusses techniques for integrating SVA (both coverage and assertions) into a UVM testbench in a flexible manner. It is also useful to note that SVA code encapsulated in an interface or checker[1] can be injected into any place in the statically-instantiated Verilog hierarchy using the `bind` construct, offering a convenient way to collect white-box coverage from deep within the RTL design.

5. Packaging Options

As already noted, coverage code tends to have a character all of its own, and consequently it is important to keep it packaged independently of other code in the environment. The usual best-practice rules of code organization apply, of course – keep each class in a separate file, isolate concerns into classes, don't allow classes to become too big – but the need for coverage code to extract information from various sources can make such organization more difficult than it might first appear.

5.1 Coverage in a subscriber component

Coverage is an inherently passive function – it observes activity, but does not affect what's happening elsewhere in the environment. Consequently, the UVM *subscriber* and *analysis_port* paradigm is well suited to the packaging of coverage, especially when coverage information is to be taken from a single source or from a rather small number of sources. The `uvm_subscriber` component base class is pre-equipped with an analysis export, so that the developer needs only to create a `write()` method implementation in their derived class. This method can massage the observed transaction data in any way that is appropriate before calling a covergroup's `sample()` method. For situations where information must be gathered from multiple sources, standard techniques (commonly using the `uvm_analysis_imp_decl` macro, or multiple `tlm_analysis_fifo` instances) readily support creation of subscriber components with more than one analysis export.

[1] We use "checker" here to mean specifically the SystemVerilog `checker` construct. Checkers were introduced in the 2009 revision of the SystemVerilog standard, and provide a flexible way to encapsulate SVA code so that it can be instantiated as needed. For more detail on checkers, see reference [7] or any other good text on SVA.

5.2 Coverage in an extended class

When more flexibility is required than can be achieved with TLM analysis connections, it may be helpful to have your coverage code embedded directly within another UVM object or component. This can easily be done by creating a derived class containing all the coverage code. Because that code forms an integral part of the original base class, it has direct access to all the base's data members and methods; but because the derived class is coded in a separate file, it isolates your coverage code in a convenient way. This approach may be particularly useful when adding coverage to a scoreboard or some other end-to-end checking component. The original base component typically contains modeling code that captures a convenient representation of the state of interesting parts of the DUT, making it easier to make accurate decisions about whether a given transaction (or other activity) should or should not contribute to each coverage point.

5.3 Instantiate a coverage object within another object

Finally, it may be appropriate to create a customized architecture based on instantiating a coverage object within some other component (using composition, rather than the inheritance proposed in section 5.2). The coverage object can have a reference back to its enclosing component, either by using the `get_parent()` method (if the coverage object is a component itself) or by having the enclosing component push a self-reference into a data member of the coverage object. Through this back reference, the coverage component can reach all public data members and methods of the enclosing component.

This approach is more flexible for re-use and extension than the inheritance approach, because the coverage is now in a completely separate class that can be factory-replaced in its own right.

6. Encapsulation

A recurring theme in coverage code is the need to capture information from multiple locations in the test environment in order to determine the real coverage that has been hit. For example, effective coverage of a register value requires not only that the register has been written with that value, but also that the DUT is in such a state that it actually makes use of the written value. Without extensive knowledge of DUT state and activity, coverage of individual signal values such as registers is almost useless, and gives false confidence that the requisite functional coverage has been achieved. Reference [2] gives much more detail on this kind of concern.

Consequently, some coverage code needs to know a great deal about the state of the DUT. Some of this information can be inferred from the state of testbench end-to-end checking components, which need some understanding of DUT operation in order to do their checking. Sometimes, though, it can only be obtained from within the DUT itself.

Regardless of the exact details, the highly connected nature of such coverage code is always likely to be troublesome. Multiple, disparate connections are a fact of life in this situation, but at least it is possible to keep the testbench environment reasonably tidy by encapsulating the coverage code in some kind of system monitoring component, as indicated in Figure 6-1 below. Such encapsulation makes it much easier to modify and configure the coverage code at a later stage.

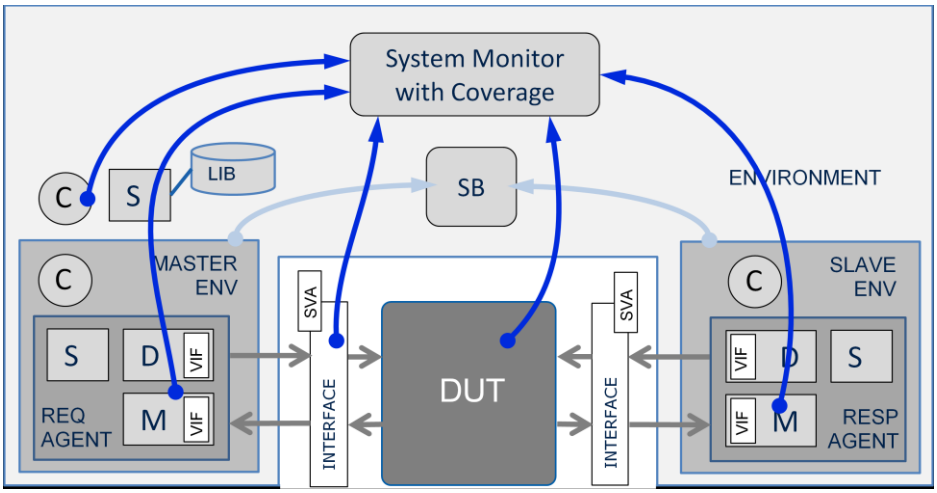

Figure 6-1: System monitor component collects information from many parts of the environment

7. Configurable or Self-tuning Coverage

If coverage is to be collected on a configurable DUT, covergroups must be tailored to suit the requirements. The authors routinely hear clients asking questions such as "why can't you use a generate loop to construct bins in a coverpoint?" and it is clear that the customization of covergroups gives rise to considerable effort and frustration.

7.1 Bins and coverpoints are not extensible

For better or worse it is a feature of SystemVerilog that the set of coverpoints of a covergroup, and the set of bins of a coverpoint, are hard-coded and cannot be altered.

The situation is not quite as bad as our simple headline statement. A bin can be specified as an automatically sized array bin. Combined with parameterized bounds for the bin, this is a useful way to specify a set of bins whose size (number of bins in the set) can be configured.

```
covergroup example1;
  coverpoint x {
    bins small_values[] = {[SMALL_MIN:SMALL_MAX]};
  }
endgroup
```

Code Example 7-1

The "parameters" `SMALL_MIN` and `SMALL_MAX` might indeed be true Verilog parameters or localparams. More usefully, they could be numeric arguments passed to the covergroup at its creation:

```
covergroup example1(int SMALL_MIN, int SMALL_MAX);
...
// Create covergroup
example1 = new(.SMALL_MIN(3), .SMALL_MAX(10));
```

Code Example 7-2

As we shall see later, this ability to parameterize a covergroup through its constructor is powerful and important, but needs handling with some care especially in the UVM context.

7.2 Bins cannot be repurposed unless the code is rewritten

A major limitation of the bin parameterization outlined above is that a bin definition cannot be reassigned from a regular bin to an illegal or ignored bin. This can be very irksome for DUTs with highly parameterized topologies and address maps. For example, consider a coverpoint designed to record accesses from a set of bus masters to a set of slaves. Depending on device configuration, some combinations of master-to-slave access may be forbidden, and should appear as illegal bins. There is no way in SystemVerilog to flip a coverage bin between an illegal and a regular bin; it must simply be redefined appropriately in a new version of the covergroup.

8. Traditional Tricks for Greater Flexibility

There are a few coding tricks that do not directly solve the concerns outlined in section 7, but nevertheless have been found useful in practice. Although they are not novel, they are offered here because the authors have not seen them widely documented elsewhere.

8.1 Covering the result of a function call or other expression

The general syntax of a coverpoint in SystemVerilog (taken from [3], clause 19.5) is:

cover_point ::=
[[data_type_or_implicit] cover_point_identifier ***:*** *]* ***coverpoint*** *expression [* ***iff (*** *expression* ***)*** *] bins_or_empty*

The simplest example of this syntax is a coverpoint covering all values of a single variable:

```
coverpoint my_variable;
```

Code Example 8-1

The coverpoint can be labeled, allowing its name to be decoupled from the name of the variable to be sampled and providing a meaningful name if the coverpoint expression is complicated:

```
cp_example_covpoint: coverpoint (my_variable+3) % 4;
```

Code Example 8-2

The coverpoint is not restricted to a simple variable or signal name. It can also cover any expression, as in the following example where only the count of bits set in a vector is considered.

```
bit cp_parity: coverpoint $countones(serial_word) {
  bins set_bits_count[] = {[0:$bits(serial_word)]};
}
```

Code Example 8-3

In the example above, the set of bins has been parameterized to match the number of bits in the word being counted. This is possible because the system function `$bits` acts as a *constant expression*; like a parameter, it is an elaboration-time constant and can be used to parameterize bins and other features of the environment.

Using a function as the coverpoint expression can provide a great deal of flexibility by conditioning your coverage data to fit into a meaningful set of bins. The function is typically defined elsewhere in the class that contains the covergroup, where it can be a virtual function allowing overriding if future needs change. This technique is especially useful when the function returns a value of

enumerated type, so that the bins are labeled by the enumeration[2]. The function can then perform arbitrarily complicated analysis on the source data, yielding a small number of meaningful classifications each of which is represented by one value of the enumeration type.

In the following example, we have two vectors `valid` and `ready` for the handshake signals on an arbitrary number of channels. We wish to categorize these signals in what appears to be a simple way:

- QUIET: there are no VALIDs
- BUSY: one or more VALIDs also have a READY, and there is no VALID without READY
- STALL_ALL: all VALIDs are set, but no READYs
- STALL: one or more VALIDs are waiting for READY, and no VALID has READY
- STALL_BUSY: two or more VALIDs, some of which have READY and some do not

Creating bins for this specification is extremely laborious and verbose, especially if the number of bits is parameterized. However, a preprocessing function can deal with it rather elegantly, with bins created automatically for each enumeration label:

```
parameter CHANS…; // Number of channels, width of vectors
typedef enum {QUIET, BUSY, STALL. STALL_BUSY, STALL_ALL} hs_e;
function hs_e handshake_state(bit [CHANS-1:0] valid, ready);
  if (valid == 0) return QUIET;
  if ((valid & ~ready) == 0) return BUSY;
  if ((valid == {CHANS{1'b1}}) && (ready == 0)) return STALL_ALL;
  if ((valid & ready) == 0) return STALL;
  return STALL_BUSY;
endfunction
...
covergroup ...
  cp_hs_state: coverpoint handshake_state(valid, ready);
endgroup
```

Code Example 8-4: Function classifies numerous values into a few meaningful scenarios

This same technique is also valuable in creating illegal bins. Rather than constructing the illegal bins directly from the raw data, a function can determine illegal conditions from the sampled data and these conditions can be binned. As an example, suppose it is illegal for more than one READY to be set without a corresponding VALID. This could be reported meaningfully by the following code, which provides one bin for each possible count of unused READYs. Without a function to calculate the number of READYs, the bins in this covergroup would be tiresomely clumsy and impossible to configure for different word widths.

```
covergroup ...
  cp_unused_ready: coverpoint $countones(ready & ~valid) {
    illegal_bins too_many_unused_ready[] = {[2:CHANS]};
    bins no_unused_ready = {0};  // Normal, expected behavior
    ignore_bins one_unused_ready = {1}; // Within spec, but won't occur
  }
endgroup
```

Code Example 8-5: Coverpoint with a non-trivial expression

[2] Not for the first time, we encounter a situation in which the ability to extend enumeration types in SystemVerilog would be very welcome. Regrettably it is not possible today, and it is unlikely to become possible any time soon.

8.2 Customized invocation of the `sample` method

Introductory texts often describe covergroups with a clock providing automatic sampling:

```
covergroup auto_sampled_cg @(posedge SYSCLOCK);
```

Code Example 8-6: Automatic sampling

While this form of sampling can be useful for some straightforward situations such as transaction coverage on a bus, it is usually too inflexible for serious usage. We favor the use of covergroups with no sampling clock, and explicit code elsewhere in the class that calls the covergroup's `sample` method. This is especially important when coverage makes use of information that has been gathered from multiple sources, possibly over a significant span of simulation time, and it is important that the whole set of data has been collected before the covergroup's `sample` method is called.

The SystemVerilog language standard's 2009 revision introduced a new feature for covergroups, allowing coverage data to be passed as an argument to the `sample` method rather than having to be placed in a variable in the scope that encloses the covergroup. While this feature is never essential (it's always possible to place data into a variable and then call the covergroup's `sample` to capture that variable's newly updated value), it can be very convenient. It is particularly useful for coverage preprocessing that requires a covergroup's `sample` method to be called repeatedly, as in the following example which checks that both 0 and 1 have been written to every bit of a word.

```
covergroup reg_toggle_cg with function sample(int bit_num, bit bit_val);
  coverpoint bit_val {
    bins value[] = {0,1};
    type_option.weight = 0;
  }
  coverpoint bit_num {
    bins bit_number[] = {[0:NBITS-1]};
    type_option.weight = 0;
  }
  cp_bitXval : cross bit_num, bit_val;
endgroup
function void cover_new_value(bit [NBITS-1:0] value);
  for (int b=0; b<NBITS; b++) begin
    // Re-sample the covergroup for each bit of the value
    reg_toggle_cg.sample(b, value[b]);
  end
endfunction
```

Code Example 8-7: Passing arguments to the sample() method

In this example, each call to `sample` covers the index number of one of a vector's bits, and the value of the corresponding bit of that vector. By doing this once for each bit in the vector, the cross point `cp_bitXval` builds up coverage of 0 and 1 values in each bit position. The class method `cover_new_value` should be called once whenever there is a new value to be covered.

9. Fine control of coverage bin structure and configuration

There are at least two situations in which we need programmable or configurable control over the bin structures in our coverpoints:

- Coverage, especially cross coverage, where the required bin structure is complicated or difficult to describe using the standard bins constructs
- Parameterized coverage where the bin structure must be modified to reflect the configuration currently in force in DUT and/or testbench

Sections 8.1and 8.2discussed how coverage data can be massaged, either before being presented to a covergroup, or within the covergroup using a coverpoint expression. However, the bins of a coverpoint are fixed at the time the covergroup is constructed, and cannot be adjusted using these techniques. This section presents various techniques that are available in the VCS SystemVerilog simulator for helping with this problem.

9.1 Covergroup constructor arguments

Bins of a coverpoint can of course be tailored using SystemVerilog parameters or other constant expressions, as in the following example:

```
parameter MIN = 3;
parameter MAX = 100;
parameter NUM_BINS = 5;
...
int x;
...
covergroup param_bins_cg;
  coverpoint x {
    bins lowest = {MIN};
    bins highest = {MAX};
    bins middle[NUM_BINS] = {[MIN+1:MAX-1]};
  }
endgroup
```

Code Example 9-1: Reshaping coverage bins using parameters

However, module parameters are generally too inflexible for use in a class-based verification environment. Greater flexibility can be achieved by adding constructor arguments to the covergroup. These arguments are then populated at the moment the covergroup is constructed, and are used at that time to configure the covergroup's bins. In this way, values computed at runtime can be used to shape coverage bins. Below we show the same example, reworked to use covergroup constructor arguments and showing how the constructor may be called. It is important to note that any covergroup in a SystemVerilog class *must* be constructed by the class's own constructor; it is impossible to construct the covergroup at any other time. In a later section we will explore the interaction between this rule and use of covergroups in the UVM. For the purposes of the present example, though, we assume that the necessary information can be passed directly to the class constructor through its argument list.

```
class configurable_coverage_class;
  int x;
  covergroup constructed_bins_cg(input int min, max, num_bins);
    coverpoint x {
      bins lowest = {min};
      bins highest = {max};
      bins middle[num_bins] = {[min+1:max-1]};
    }
  endgroup
  function new(int min, int span);
    int max = min + span - 1;
    int n_bins = (span-2)/5; // aim for 5 values per bin
    constructed_bins_cg = new(min, max, n_bins);
  endfunction
endclass
```

Code Example 9-2

9.2 Bin filter expressions

The 2012 revision of the SystemVerilog LRM introduced many new features to improve the expressive power of coverpoint bins. At the time of writing VCS fully supports bin filter expressions using the `with` construct, making it much easier to write meaningful coverage bins in some situations. Our next code fragment revisits the READY/VALID example of Code Example 8-4 but adds a new challenge. Suppose our specification makes all the following conditions illegal:

- VALID==0 (no VALID bits asserted)
- any bit is set in VALID for which the corresponding bit in READY is false
- VALID==READY

Using the traditional `binsof` constructs to create an `illegal_bins` specification for this, especially when the width of the READY/VALID words is parameterized, would be extremely laborious. However, we can rather easily implement it using the new bin filter expressions:

```
cp_valid: coverpoint valid ...
cp_ready: coverpoint ready ...
cp_validXready: cross cp_valid, cp_ready {
  illegal_bins bad_valid = cp_validXready with (
    cp_valid == 0 || (cp_valid & ~cp_ready) != 0 || cp_valid == cp_ready
  );
}
```

Code Example 9-3: Bin filter expression

Note that the `with`-expression automatically respects the vectors' widths, and therefore is self-tuning for any number of channels.

There are some non-obvious features in Code Example 9-3. First, the name of the coverpoint itself (`cp_validXready`) is used as the basis of the bin expression, before the `with` keyword. Second, it is the names of coverpoints (`cp_valid`, `cp_ready`) that participate in the filter expression, *not* the variables that are being sampled. Nevertheless, filter expressions using `with` provide a powerful and convenient new way to define the bins of your covergroups. They are especially useful for cross points, where the language rules specify that bins are automatically created for any values that are not otherwise mentioned in the cross point definition. This means that, in all realistic situations, it is essential for your cross point definition to specify bins (some of which may

be `ignore_bins` or `illegal_bins`) that cover every possible combination of values. If you do not do this, you will find the cross contains unwanted auto-generated bins that will distort your coverage figures. In a regular coverpoint it is possible to use the `default` specification to create an ignored bin that collects all otherwise unspecified values of the coverpoint, but there is no such `default` bin specification for cross points.

10. Self-configuring Coverage and the UVM

A typical UVM testbench is likely to be highly reusable and configurable. DUT parameters, together with (possibly randomized) test configuration features, are used to populate one or more configuration objects which are then propagated to appropriate parts of the test environment using the UVM configuration database or other appropriate mechanisms. During the UVM build phase this configuration information is then used to control the UVM environment's structure, and to set the values of class members that will subsequently control progress of the simulation.

This configurability can lead to quite drastic differences in testbench structure and behavior from one run to another. From our current perspective it is important to note that it can have a large effect on the required coverage. Changes in DUT parameterization can, for example:

- change the widths of vectors
- control the existence of significant DUT features, numbers of ports, etc
- determine whether certain operations are illegal

All these changes may require corresponding adjustment of coverage collection arrangements, and of course it is essential that any such adjustments should be fully automatic based on the contents of the UVM configuration objects we mentioned earlier.

10.1 Configuration Phasing

Immediately we encounter a significant difficulty. As previously mentioned,

- covergroups in a class must be constructed within the class's constructor (`new`).

For UVM components, this is a troublesome limitation. All components must be constructed during the build phase – indeed, components are normally brought into existence by a factory `create` call in their parent's `build_phase` method. During the build phase, configuration information is readily available. In particular, if UVM field automation macros are used, automatic configuration of those fields does not take place until the component's `build_phase` method calls `super.build_phase`. **This is too late to construct covergroups based on the configuration**, because the component's constructor (`new`) has already finished.

It seems, then, that coverage should perhaps be encapsulated in UVM objects rather than in components. Classes that are not components can be constructed at any time, allowing the execution of an object's constructor to be postponed until all necessary configuration is readily available. However, even this is not as satisfactory as we might hope, because **the constructor of a UVM object has a fixed argument list** and therefore there is no obvious way to pass configuration information into the object's constructor.

In the following sub-sections we outline some coding patterns that we have found useful in addressing these concerns, allowing configuration information to be passed in to the constructor of an object (and then used in construction of a covergroup) with no artificial restrictions imposed by the UVM's phasing and constructor prototypes.

10.2 Factory override of class parameters[3]

If a class has parameters (which, of course, may be either value or type parameters) then those parameters are fixed for any given specialization of the class. The class's parameters are available in its constructor without needing to be passed as constructor arguments, and therefore can be used not only to parameterize a covergroup in the class, but also to compute argument values for the covergroup's constructor. Ordinarily it is not recommended to use parameters to specialize a class in a UVM testbench, because there is then a "parameter ripple" effect causing any objects that use a parameterized class to be parameterized themselves, so that they can specify the parameters of the class they use.

This unwanted outward propagation of class parameters can be sidestepped by using the UVM factory to replace a base class object with a parameter-specialized derived class, as shown in the following example. We begin with a base class that will act as a placeholder for your coverage component. It may have (for example) one or more analysis exports, or other features allowing it to hook into the environment. Note that it is not parameterized, and typically has no real functionality of its own.

```
class cov_base extends uvm_component;
  `uvm_component_utils(cov_base)  // factory registration
  ... other details not shown
```

Code Example 10-1: Base class for parameterizable coverage component

Next, we derive a class from this one. Our derived class contains the coverage constructs that we require, and has some parameters that shape its covergroups. As a simple example of how this might be done, we have parameters specifying the upper and lower limit of the legal value of one variable that we are to cover, and an type parameter that should be specialized with an enumerated type, providing the set of names (and associated values) for another variable:

[3] The authors are indebted to Dr David Long (Doulos Ltd) for bringing this intriguing technique to their attention.

```
class param_cov_base
    #(parameter int MIN=1, MAX=2, parameter type T = int)
    extends cov_base;
  `uvm_component_param_utils(param_cov_base#(MIN,MAX))
  int x;  // These are the variables that
  int y;  //   will be covered by this class
  // Covergroup shaped by class parameters
  covergroup shaped_cg;
    cp_int_x: coverpoint x {
      bins legal[] = {[MIN:MAX]};
      illegal_bins too_low = {[$:MIN-1]};
      illegal_bins too_big = {[MAX+1:$]};
    }
    cp_T_y: coverpoint T'(y); // gets its bin names from the enum type T
  endgroup
  // Constructor also constructs the CG
  function new(string name, uvm_component parent = null);
    super.new(name, parent);
    shaped_cg = new;
  endfunction
  ... other details not shown
```

Code Example 10-2: Parameterized derived coverage class

Note that these parameters do not affect the way the component fits into its enclosing environment. They affect only the way its covergroup is built and used.

Finally, we can use a factory override to set the parameters on any given instance of this component, by overriding its type to have the appropriate specialization. As a simple example, this by-type override will cause *all* instances of the component to be parameterized in the same way:

```
typedef enum {ONE=1, THREE=3, FIVE=5} small_odd_e;
...
set_type_override_by_type(
  cov_base::get_type(),
  param_cov_base#(.MIN(50), .MAX(100), .T(small_odd_e))::get_type()
);
```

Code Example 10-3: Factory override to establish a specialization

The coverpoint `cp_T_y` will now have three bins, labeled to match the enumerated type's names.[4]

The type override must be executed before the components in question are built. In practice it is likely to be part of the setup code that plants entries in the UVM configuration database just before invoking the `run_test()` method. Consequently, this code will be in the test runner module where it has direct access to parameters of the RTL design. In this way, covergroups can readily be configured to correspond to RTL parameter values.

A possible drawback of this technique is that all the parameters and type parameters used in the factory override must be established at SystemVerilog elaboration time; they must all be constant

[4] The complete code example, available on the authors' website, also shows how it is possible to use the `bins…with` construct to create an illegal or ignore bin for values that are not in the enumerated type's set of names.

expressions. This precludes the possibility of randomization, or setting the parameters from a configuration file.

10.3 Solution based on UVM objects and the resource database

As noted in section 10.1 above, a UVM object that is not a component can be constructed at any time. This means that any necessary configuration information can be gathered before the object is constructed. However, there remains the problem that the UVM factory mechanism requires an object's constructor to conform to the standard prototype:

```
function new ( string name = "" );
```

Code Example 10-4: Mandatory UVM object constructor prototype

How, then, are we to pass configuration information into the object's constructor so that it can be used to create a covergroup with appropriate parameterization? Fortunately it is possible to use the UVM's resource database. The configuration database is less appropriate in this case, because objects are not located in the component hierarchy and so the configuration database's hierarchy features are not useful.

All that is necessary is for the creating component (or sequence, or object) to plant an appropriate entry in the resource database before creating the object. (One entry should be sufficient, as the entry should probably itself be an object that encapsulates all relevant values.) Once this has been done, the object's constructor can interrogate the resource database to retrieve the entry, and then can use the retrieved object to set up its embedded covergroup appropriately.

This approach is straightforward and widely applicable. It has the small drawback that the code to set up the resource database entries in the creating class, and to retrieve them in the object's constructor, is a little clumsy and not entirely obvious. Much more important, though, is the problem that a suitable name (key) must be chosen for the resource database entry.

It is likely that the creating class is in fact a UVM component. If so, the resource key can be chosen automatically and without ambiguity in the following simple way:

- choose a short name to identify the object (`".cov_object"` in the example below)
- construct a complete name string by concatenating the creating component's full name with the short name
- use that complete name both as the resource key, and as the name of the object
- in the object's constructor, use `get_name()` as the resource key to interrogate

```
string cov_object_name = {get_full_name(), ".cov_object"};
cov_config_class cov_cfg;  // object containing coverage config info
... set up cov_cfg appropriately, then prepare resource DB entry:
uvm_resource_db#(cov_config_class)::set(null, cov_object_name, cov_cfg);
cov_object = cov_object_class::type_id::create(cov_object_name);
```

Code Example 10-5: Component creates a unique name for both the coverage object and its resource DB entry

```
class cov_object_class extends uvm_object;
  `uvm_object_utils(cov_object_class)
  cov_config_class cfg;
  covergroup configurable_cg...
    ...
  endgroup
  function new(string name = "");
    super.new(name);
    bit cfg_OK = uvm_resource_db#(cov_config_class)::read_by_name(
                                        null, name, cfg);
    ... if cfg_OK is false, throw an error because no config available
    configurable_cg = new(...); // construct covergroup using config info
  endfunction
```

Code Example 10-6: Coverage object uses its own name as a resource DB key

Solution based on an embedded non-UVM wrapper class

An interesting alternative solution is based on accepting the need for a coverage class's constructor to have custom arguments so that it can pass on those values directly to the covergroup it constructs. The coverage class then cannot be a UVM object or component. However, it *can* be a local class embedded within a UVM component class. In this way we get the best of both worlds. We have all the features of a UVM component including hierarchy, phasing, factory replacement, field automation and so on. We also have the means to construct our covergroup at any time of our choosing, with whatever constructor arguments it needs.

To keep this tutorial example compact, we consider an artificial example where a transaction contains just one `int` value, and the covergroup's configuration is an object containing only the upper and lower limit values for coverage of that transaction `int`. Despite this radical oversimplification, our example shows the key features of the approach. The code is presented in fragments here to allow explanations to be interspersed, but it is listed in full at the end of the paper. A more realistic example, which extends the UVM "UBUS" sample code, is available on the authors' website.

First we define the transaction object, and a configuration object containing the two `int` limits:

```
class int_txn extends uvm_object;
  int value;
  // and all the usual UVM stuff such as constructor, registration, ...
endclass

class int_txn_cov_cfg extends uvm_object;
  int lower_limit;
  int upper_limit;
  // etc ...
endclass
```

Code Example 10-7: Transaction and configuration classes

Next we create an observer component that will collect coverage on these transactions. It is coded here as a `uvm_subscriber` that would be connected to a monitor's analysis port, but of course there are many other possible arrangements, as discussed in section 5.

```
class int_txn_observer extends uvm_subscriber #(int_txn);

  // Configuration object handle:
  int_txn_cov_cfg cov_cfg;
  // Ensure the config object can be picked up by automatic configuration
  `uvm_component_utils_begin(int_txn_observer)
    `uvm_field_object (cov_cfg, UVM_DEFAULT)
  `uvm_component_utils_end
  ...
```

Code Example 10-8: Component class that will implement some coverage

It is very important to note that we do *not* directly create a covergroup in this component. Instead, the covergroup will be created in a class that is nested within the `int_txn_observer` class. The covergroup accepts a configuration object as a constructor argument, and has a custom sample method allowing a transaction object to be passed in to it. It contains a coverpoint that samples the transaction's data, and whose bins are configured by the configuration object that was passed as a constructor argument. This covergroup is constructed by the embedded class's constructor.

```
  // Class embedded within int_txn_observer, not visible outside it
  class coverage_container; // NOT a uvm_object
    covergroup cg (int_txn_cov_cfg cfg)
        with function sample(int_txn txn);
      cp_value: coverpoint txn.value {
        bins minimum = {cfg.lower_limit};
        bins maximum = {cfg.upper_limit};
        bins others[] = {[cfg.lower_limit+1 : cfg.upper_limit-1]};
      }
    endgroup
    function new(int_txn_cov_cfg cfg);
      cg = new(cfg);
    endfunction
  endclass
```

Code Example 10-9: Coverage wrapper class nested within the component class

It is interesting to note that the embedded class is very small indeed, being no more than a thin wrapper around the covergroup. This means that there is no significant obstacle to creating alternative or additional coverage in a class that is derived from the component and instantiated using factory override.

Back in the scope of the main class, we now need to construct an instance of the embedded class at a time when all necessary configuration information is known. It may be useful to postpone this until the very end of UVM elaboration so that we can be sure that all configuration updates and propagation have taken place (the configuration object was most likely obtained from the configuration database during the build phase, but might be determined somewhat later):

```
coverage_container cvg_inst;
function void end_of_elaboration_phase(uvm_phase phase);
  if (cov_cfg == NULL) begin
    `uvm_error("NO CONFIG for coverage here!!!!")
  end
  cvg_inst = new(cov_cfg);
endfunction
```

Code Example 10-10: Instantiating an instance of the embedded coverage class within the subscriber component

As a `uvm_subscriber`, this component implements a `write` method that accepts transactions from the connected analysis port during the run phase. This write method is straightforward, simply appealing to the `sample` method of the embedded class's covergroup. There is no need to take a copy of the transaction, as it is used only within the body of this function and it is not stored.

```
function void write(int_txn t);
  cvg_inst.cg.sample(t);
endfunction
```

Code Example 10-11: Subscriber class's `write` method

We have now achieved our goals. The covergroup (wrapped in the thin skin of its embedded class) can be constructed at any time, allowing all necessary configuration information to be gathered before its creation. Coverage is within a UVM component, and so has all the benefits of UVM hierarchy, naming, utilities, and factory override capability.

11. Conclusion

Coverage code has a distinct character and brings its own special challenges. A verification engineer developing coverage code must continually pay attention to many considerations: ensuring the trustworthiness of their coverage, designing for extension and reuse, achieving configurability for a parameterized DUT or for different verification scenarios, and producing maintainable code that is easy for colleagues and reviewers to understand. This paper offers a variety of general guidance, methodology recommendations and specific coding patterns that may help to achieve those aims.

12. References

[1] A. Piziali, Functional Verification Coverage Measurement and Analysis, New York: Springer, 2008.

[2] M. Litterick, "Lies, Damned Lies and Coverage," in *DVCon*, San Jose, 2015.

[3] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.

[4] Accellera Systems Initiative, "UVM (Standard Universal Verification Methodology)," June 2014. [Online]. Available: http://www.accellera.org/downloads/standards/uvm. [Accessed April 2015].

[5] M. Litterick, "SVA Encapsulation in UVM," in *DVCon*, San Jose, 2013.

[6] J. Sprott, P. Marriott and M. Graham, "Navigating the Functional Coverage Black Hole," in *DVCon*, San Jose, 2015.

[7] E. Cerny, S. Dudani, J. Havlicek and D. Korchemny, SVA: The Power of Assertions in SystemVerilog (2nd edition), Springer, 2014.