Effective SystemVerilog Functional Coverage

design and coding recommendations

Jonathan Bromley
Verilab Ltd

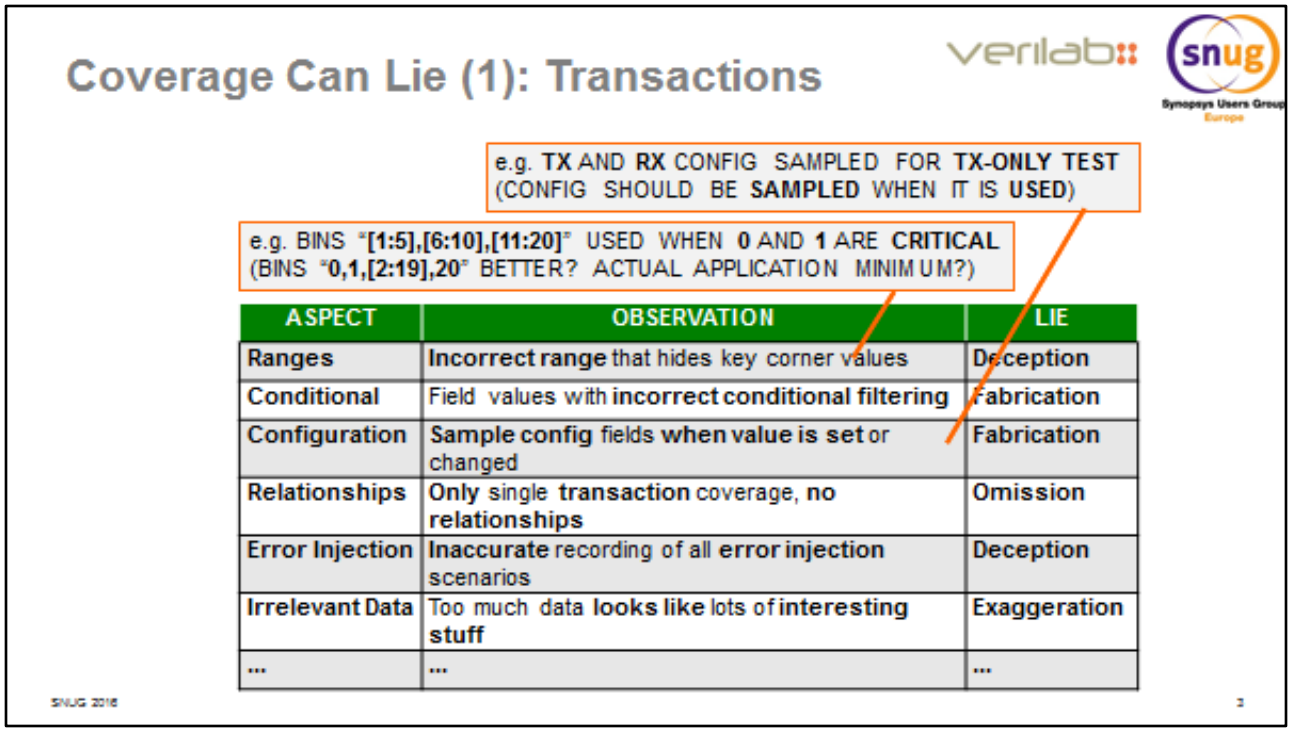
30 June 2016
SNUG Reading, England

Agenda

Coverage aims – honesty, precision, completeness, value

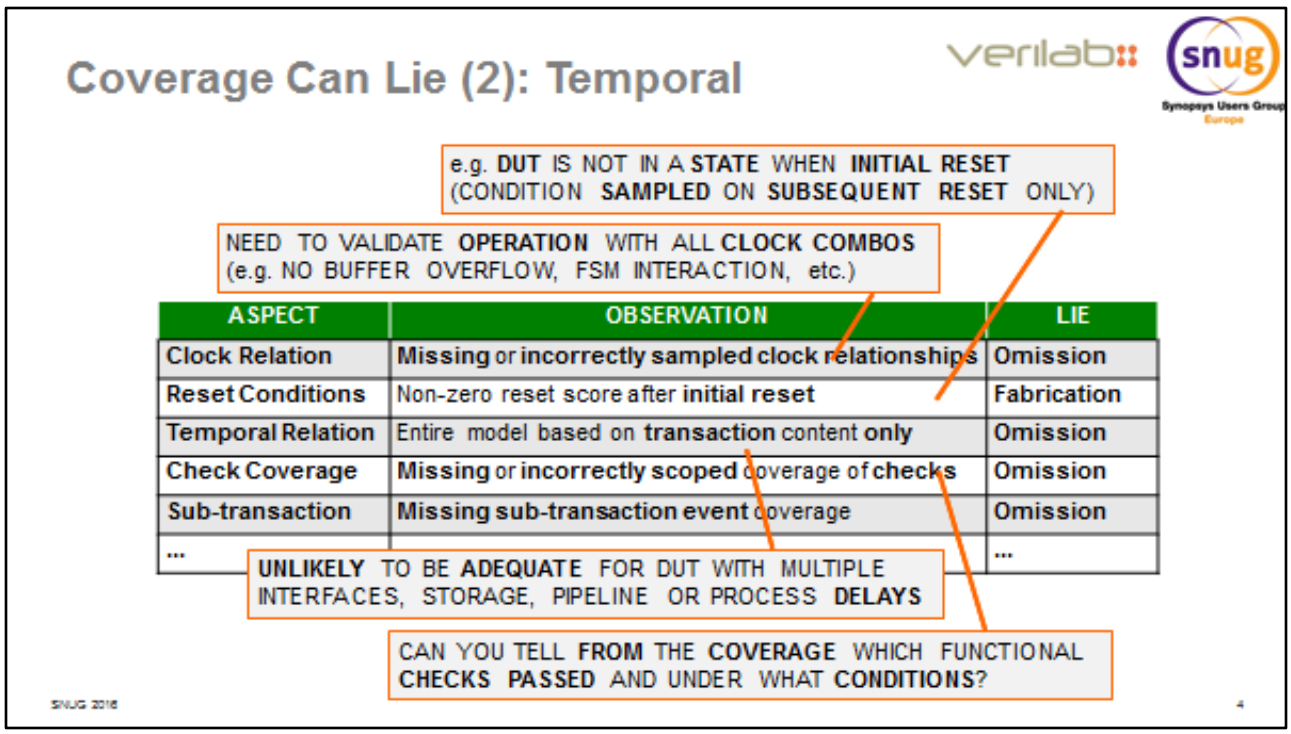
Strategic and structural concerns

Coding for flexibility, including new SV-2012 features



This slide, and the following two, are lifted directly from another Verilab paper from DVCon 2015 in which Mark Litterick described many of the pitfalls caused by careless, thoughtless or even dishonest coding of functional coverage. We won't go into it in any detail, but several important concerns are highlighted. The big problem here is that coverage is often thought of as the verification metric that's used for signoff, but – as these examples indicate – it can be deeply flawed and rarely gets the detailed analysis and review that it deserves.

Coverage Can Lie (2): Temporal



ASPECT	OBSERVATION	LIE
Clock Relation	Missing or incorrectly sampled clock relationships	Omission
Reset Conditions	Non-zero reset score after initial reset	Fabrication
Temporal Relation	Entire model based on transaction content only	Omission
Check Coverage	Missing or incorrectly scoped coverage of checks	Omission
Sub-transaction	Missing sub-transaction event coverage	Omission
...		...

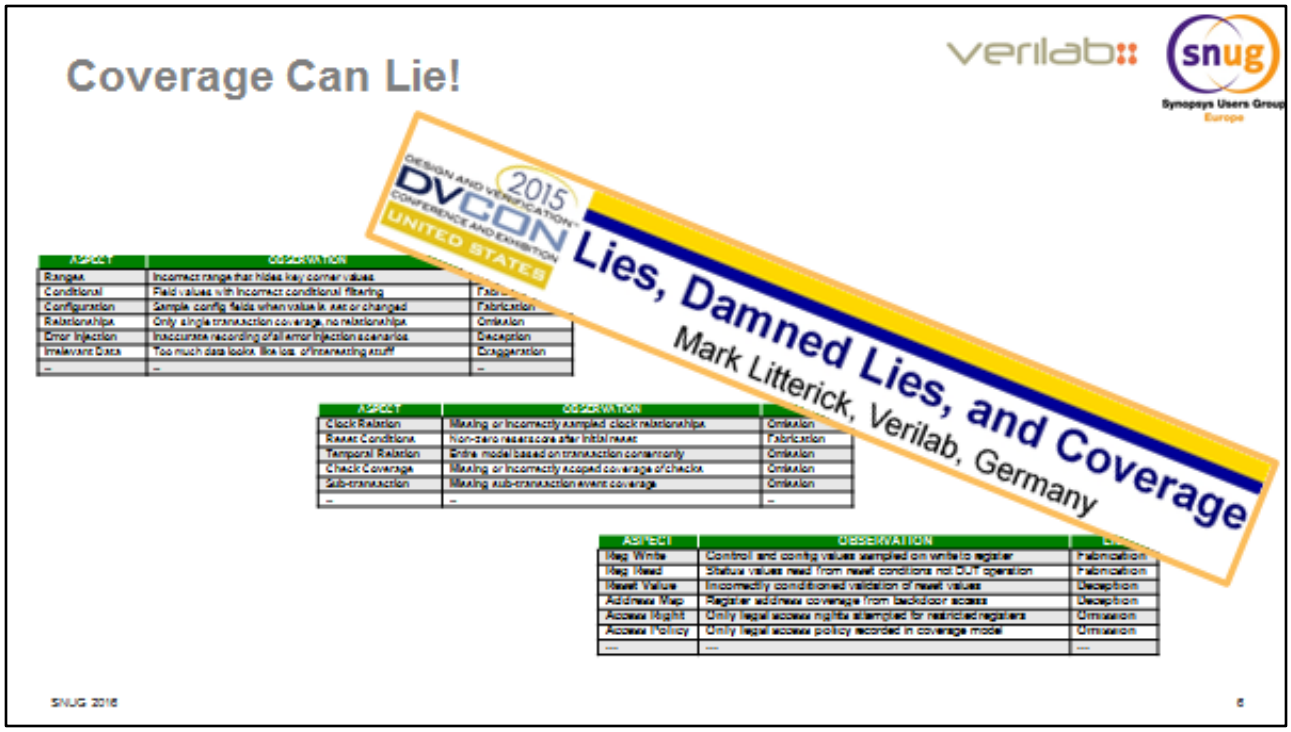
Coverage Can Lie (3): Registers

BACKDOOR DOES **NOT VALIDATE** ADDRESS **DECODE**
(EXCLUDE BACKDOOR ACCESS FROM ADDRESS COV)

EASY TO GET **100%** COVER ON **MULTIPLE WRITES**
BUT **MISLEADING** SINCE VALUES **NOT USED** BY DUT

ASPECT	OBSERVATION	LIE
Reg Write	**Control** and **config** values **sampled on write** to register	Fabrication
Reg Read	**Status** values **read** from **reset** conditions not DUT operation	Fabrication
Reset Value	**Incorrectly conditioned** validation of **reset** values	Deception
Address Map	Register **address coverage** from **backdoor** access	Deception
Access Right	**Only legal access rights** attempted for restricted registers	Omission
Access Policy	**Only legal access policy** recorded in coverage model	Omission
...	...	...

NEED TO **ALSO COVER** ALL RELEVANT **ACCESS ATTEMPTS**
e.g. **WRITE 0** AND 1 FOR **W1C**, **WRITE** AND READ FOR **RO**



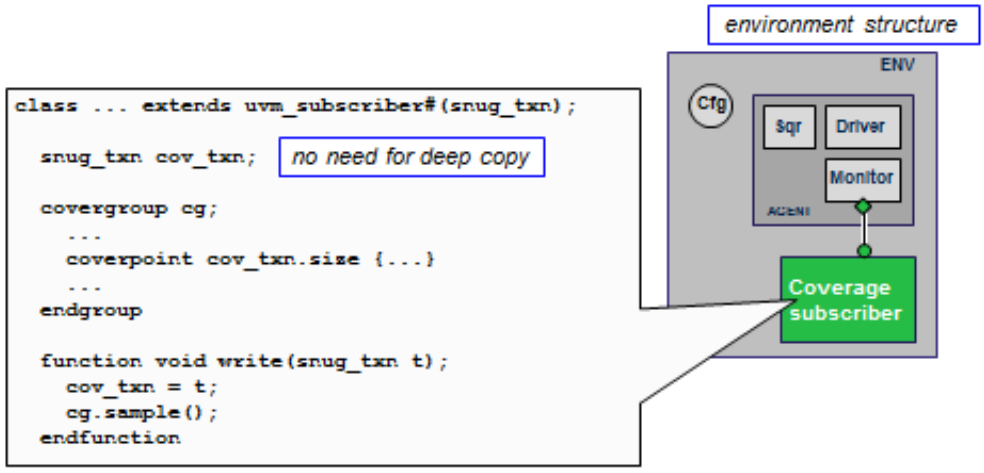
Here's the reference.

Time to cut to the chase: How do we set about writing coverage code in SystemVerilog? Simple tutorial examples can make it look easy, but the reality is often very different.

Coverage code often seems to have a distinct character of its own. It can be very verbose and repetitive, and of course it has no influence on the actual workings of a testbench – it doesn't affect stimulus, and (with the single exception of illegal bins) it doesn't even help with correctness checking. It seems sensible, therefore, to seek ways of keeping coverage code isolated so it can be reviewed independently, and doesn't clutter the rest of the testbench.

Implementation Options in UVM-SV (1)

Coverage in a Subscriber Class



SNUG 2016 8

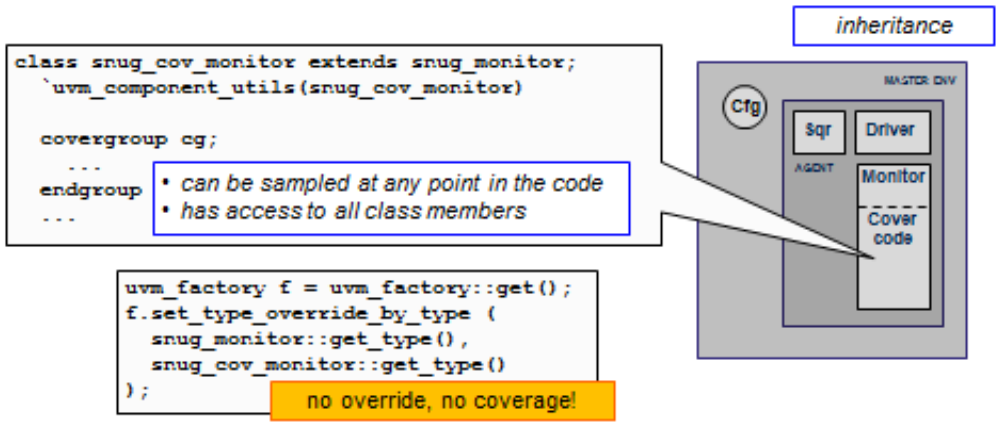
Given that coverage is a strictly passive function, the uvm_subscriber component seems an ideal candidate for encapsulating coverage code. You can add covergroups and other coverage code, and coverage can be triggered simply when a transaction or other data item is provided through the subscriber's analysis_export by calling the subscriber's write() method.

It's useful to note here that, unlike many subscriber components, a coverage subscriber usually doesn't need to take a copy of the transaction object presented to it by write(). The entire job of coverage can be done in zero time within the write() method. The transaction is neither modified nor stored. Consequently, it can simply be observed during the write() method's execution and there is no reason to make a copy of it.

This pattern, however, is limited to coverage that can be gathered on single data items (typically, transaction objects) presented on a single analysis port. It is less useful when coverage needs to examine various items gathered from various parts of a testbench.

Implementation Options in UVM-SV (2)

Extend a component to add coverage



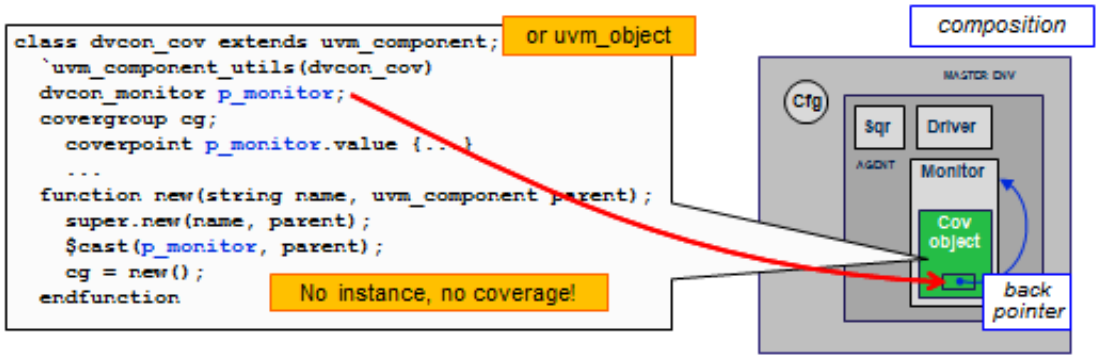
Another useful approach to creating coverage code is to write an extension to an existing class. Structurally, the coverage is now an integral part of the component in question – but the coverage code, textually, is nicely isolated in the class extension.

An especially useful feature of this approach is that the coverage code now has access to all data members (even protected ones) and methods of the base class. For simplicity and clarity our diagram shows the coverage extending a monitor class, which would restrict it to observing only that monitor's protocol – but the same approach can be taken, for example, with a scoreboard (which sees transactions from many parts of the DUT) or even the whole environment.

Factory override allows us easily to substitute the coverage-enabled extended class in place of the original base class. However, if you fail to do the factory override, you have a nasty problem on your hands: The testbench will continue to operate apparently correctly, and won't yield any errors, but the relevant coverage will not be collected! Avoiding this situation – and detecting it, if it should occur through an oversight – calls for careful review of the coverage results against your verification plan.

Implementation Options in UVM-SV (3)

Instantiate a Coverage Class in a Component



A third option for structural layout takes us back to the idea of encapsulating coverage code in a distinct component, but in this case we add an instance of that component to the testbench component that it will observe. This has all the advantages of the "extension" approach, except that protected members of the enclosing object are not available through the back pointer. It usefully isolates the coverage code in its own component, but has the drawback of that component's design being very tightly coupled to that of the enclosing component.

Implementation Options - review

- Coverage in a subscriber class *environment structure*
  - restricted to contents of a single object (transaction, transaction-set)
  - but provides good isolation
- Extend monitor class to add coverage *inheritance*
  - flexible, but limits TB structure
  - instantiate correct monitor type, else coverage missed!
- Instantiate coverage class in another component *composition*
  - coverage class has handle to component, sees all contents
  - allows all coverage implementation including control/timing
  - most flexible solution

Best choice depends on application!

A brief comparison of the three structuring approaches we've just discussed. No simple "best choice" – you must decide based on your application, TB architecture and many other factors.

Coverage Is Distributed

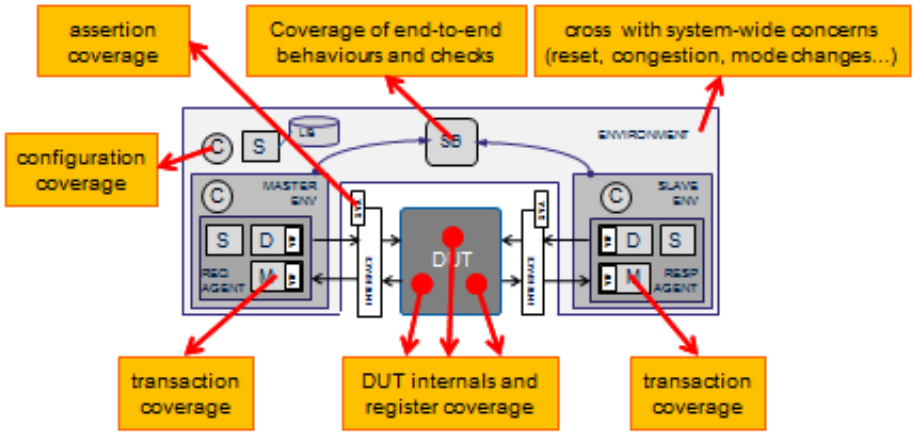
- Coverage of individual transactions: easy but insufficient
  - cover each txn from monitor – *transaction coverage*
- DUT behaviour coverage is also important, but….
  sampling and data gathering likely to be distributed across….
  - parts of verif env
    - activity on other interfaces
    - register state and changes
    - end-to-end matching
  - time
    - what else happened during the life of this instruction/transaction/… ?
    - DUT state at start, end, other key points in txn lifetime?

An important concern that we've mentioned briefly but in practice gives rise to a lot of bother: Coverage needs collecting from many parts of a testbench. In typical designs, an individual transaction, instruction, video frame or other unit of activity will pass through many different parts of a DUT, and it's important to collect coverage on that transaction's effect on each of those parts. Inevitably this means we need to probe various parts of the environment, often collecting related information over considerable periods of simulation time.

End-to-end matching or scoreboard components are a fruitful source of coverage information. They typically have extensive knowledge of the state of the DUT and the life-cycle of data as it flows through the DUT, and they can distinguish successful matches (on which coverage should of course be collected) from other activity (which usually should not contribute to coverage at all).

Coverage Is Distributed

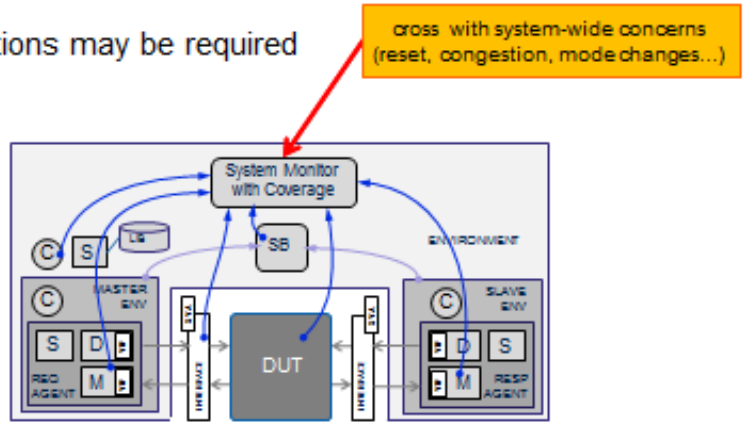
- Coverage from *many places* contributes to verif plan



Just a diagrammatic representation of what we said on the previous slide.

Coverage Is Distributed

- Coverage from *many places* contributes to verif plan
- *Encapsulate!*
- Many connections may be required



The sprawling nature of coverage data collection calls for some care and discipline if the resulting connections are not to become a mess. One very useful way to keep things tidy is to encapsulate many related coverage components in a single “system monitor” component which can then be given analysis connections (or any other appropriate mechanisms) to the rest of the testbench and even to probes bound into parts of the DUT.

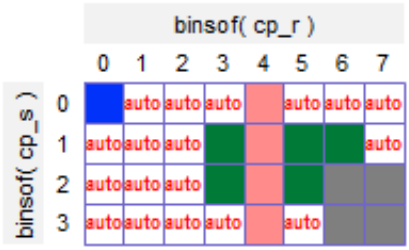
We'll round off this presentation with some ideas, coding patterns and other suggestions for making your coverage code as flexible and re-usable as possible, despite its verbose and highly specialized character.

Making Coverage Flexible

- Traditional approaches to configurable points/bins
  - `sample()` wrapper – map sampled data to coverage-friendly value
  - coverpoint expression is the result of a function
  - `sample()` wrapper – multiple `sample()` can be useful
  - covergroup constructor args used to shape bins
  - module parameters used to shape bins
- SV gotchas:
  - `iff` doesn't remove a bin, just suppresses a sample
  - Effects of `illegal`, `ignore`, `default` – often misunderstood
  - No `default` specification for cross bins

As we'll see later, recent enhancements in the SV language and available tools have made some things much easier than they used to be. First, though, let's take a look at some traditional techniques for making coverage code more flexible – and a couple of traditional mistakes. We'll look at these in the next few slides.

Cross Bins in SV




```
bit [2:0] r;
bit [1:0] s;
covergroup cg;
  cp_r: coverpoint r { illegal_bins bad_r4 = {4}; }
  cp_s: coverpoint s;
  cp_rXs: cross cp_r, cp_s {
    bins zero =    binsof(cp_r) intersect {0}
                && binsof(cp_s) intersect {0};
    bins mid  =    binsof(cp_r) intersect {[3:6]}
                && binsof(cp_s) intersect {[1:2]};
    ignore_bins brhc =   binsof(cp_s) intersect {[2:3]}
                      && binsof(cp_r) intersect {[6:7]};
  }
endgroup
```

No `default` for cross bins!

SNUG 2016 17

SLIDE STARTS: Here's a simple covergroup looking at the value of two variables 'r' (three bits, 8 values) and 's' (two bits, 4 values). We know that the value of 'r' should never be 4, so we've made that an illegal bin on the coverpoint cp_r. We're going to show you some of the awkwardness of coding cross coverage in SV, and highlight an important "gotcha".

First we describe the cross coverage bins that particularly interest us. First we'll pick the bin in which 'r' and 's' are both zero – it's highlighted in blue. There will probably be a few more special-purpose bins like this. Note the somewhat bizarre "binsof" syntax. The argument to binsof() must be a *coverpoint name*, not a variable name – but the values specified in the "intersect" ranges are values of the *expression*. You can't use the names of bins in those ranges.

Next we create a bin describing central values of the two coverpoints, highlighted in green. Two points worthy of note: you can use ranges in the intersect-expressions, and any value-pairs that fall into an illegal or ignore bin are automatically removed from the bin. It doesn't matter whether an illegal or ignore bin is declared before or after other bins; any illegal or ignore values are completely removed. It's important to be aware of the risk that this can leave some of your bins completely un-hittable, if you're not careful.

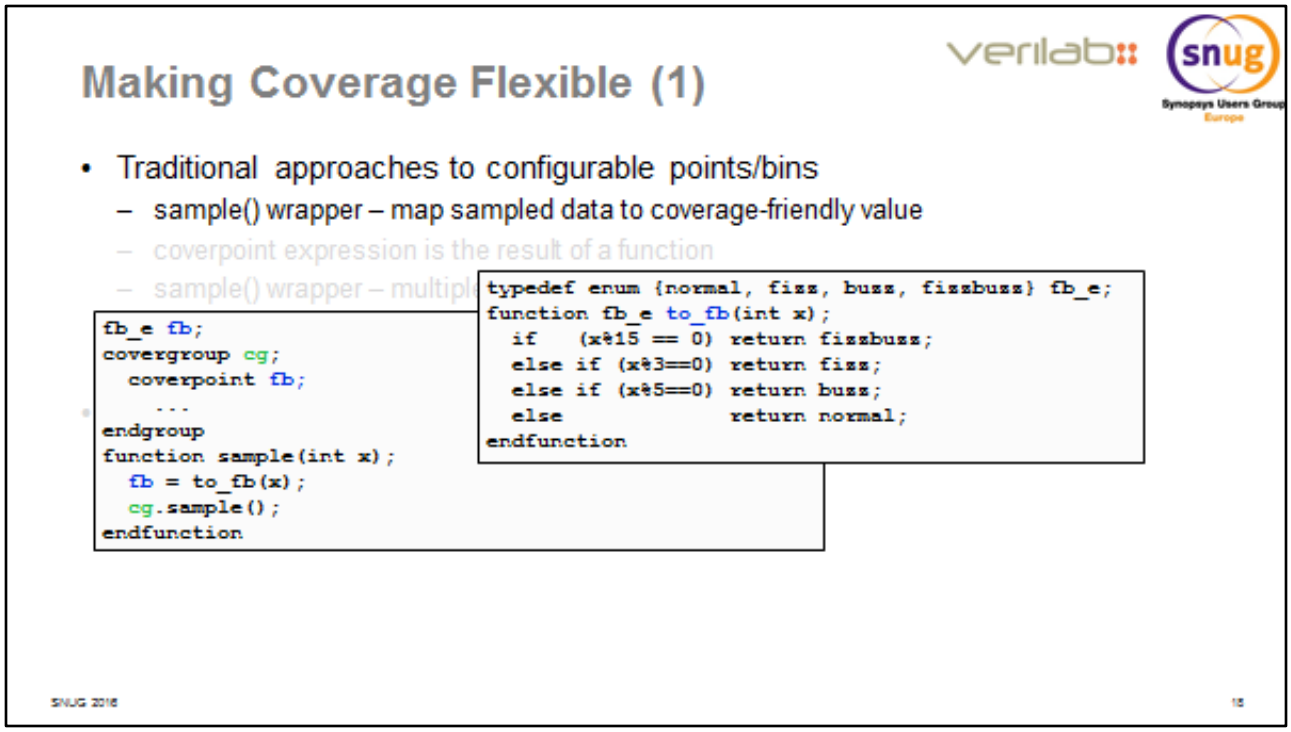
Sometimes there are values that may occur during normal DUT operation, but are not

relevant for coverage. It makes sense to specify such values in an ignore-bin. Once again, note how the ignore-bin (in gray) slices some values out of other bins.

OK, we specified all the bins we care about in our cross (in practice of course there would be more than our three) and now we're left with a weird-shaped set of holes in the map. We can soak those up in a default bin that we ignore, right? Gotcha! No, you can't… because…

…there is NO default specification available for cross bins! *Any values you don't specify become automatically created bins that fill up the remainder of the cross* ("auto" red bins). And, of course, it is exceedingly tiresome to create a bins-of-specification that will describe that set of values. You can't specify an ignore-bin covering the whole range, because it would obliterate all the bins you actually want.

There really is no truly convenient solution to this problem in traditional SystemVerilog coverage. Over the next few slides we'll look at possible fixes, and then introduce some recent language enhancements that make the problem much easier to handle.



Introductory discussion: The "fizzbuzz" challenge is a notorious interview question designed to weed out the incompetent applicants for a programming job. The deal is that you're asked to count through a bunch of numbers, printing "fizz" when you encounter a number divisible by 3, "buzz" for a number divisible by 5, and "fizzbuzz" for a number divisible by both 3 and 5. Here we've turned it into a coverage problem: if we sample some number 'x', cover how many times we get fizzbuzz, fizz, buzz or "normal" numbers.

Clearly we'd like our coverpoint to reflect those names. An enum sounds like a good start – type "fb_e" in the upper right code box. But how to map the numbers on to that enum? Function to_fb does that for us (no, we're not applying for a job here), taking any number and returning the appropriate enum value. A very simple wrapper around the covergroup's sample() method allows us to massage the raw input value 'x' and extract from it the enum that we really want to cover. Don't underestimate this technique – it's versatile, highly adaptable and very easy to understand. It also emphasises the idea that covergroups sampled automatically on an event probably aren't such a good idea as you would guess from reading introductory SV texts and training material. Use the sample() method, and don't be afraid to provide your own wrapper for it to beat your raw values into a more coverage-friendly shape.

Making Coverage Flexible (2)

- Traditional approaches to configurable points/bins
  - sample() wrapper – map sampled data to coverage-friendly value
  - coverpoint expression is the result of a function
  - sample() wrapper – multipl

```
int x;
covergroup cg;
  cp_fb: coverpoint to_fb(x);
  ...
endgroup
```

```
typedef enum {normal, fizz, buzz, fizzbuzz} fb_e;
function fb_e to_fb(int x);
  if    (x%15 == 0) return fizzbuzz;
  else if (x%3==0)  return fizz;
  else if (x%5==0)  return buzz;
  else              return normal;
endfunction
```


Another way to achieve a similar result is to have the coverpoint expression be a function call. This is more concise than the sample-wrapper method, but probably less versatile and harder to review. Note that the function is unchanged from the previous slide.

Making Coverage Flexible (3)

- Traditional approaches to configurable points/bins
  - sample() wrapper – map sampled data to coverage-friendly value
  - coverpoint expression is the result of a function
  - sample() wrapper – multiple sample() can be useful
  - covergroup
  - module pa
- SV gotchas
  - "iff" doesn't
  - Effects of i

```
bit this_bit;
int bit_num;
covergroup cg;
  coverpoint this_bit;
  coverpoint bit_num;
  cross this_bit, bit_num;
endgroup
function sample (bit [31:0] x);
  for (bit_num = 0; bit_num < 32; bit_num++) begin
    this_bit = x[bit_num];
    cg.sample();
  end
endfunction
```

SNUG 2016 20

Finally, note that there's no reason why you can't call a covergroup's sample() method multiple times for a given data set. The example on this slide answers the FAQ "how can you get toggle coverage on the bits of a vector using a SV covergroup?". The trick is to have two coverpoints, one for the bit NUMBER (index) and one for its value. Each time you sample the vector, iterate over all its bits in your sample() wrapper, invoking the covergroup's sample() once per bit. The *cross* of the bit number and bit value points provides value coverage on each individual bit of the vector. OK, this is a party trick, but the underlying idea – multiple covergroup samples, managed by a single call to the sample() wrapper – is powerful and useful.

Note, even this "party trick" has some usefulness. Standard toggle coverage simply checks that every bit has taken both 0 and 1 values *at some time*. By contrast, functional coverage like this considers the value *only when it's known to be relevant*. The distinction is important!

Recent Language Enhancements

- SV-2012 options available in VCS:
  - bin `with()` function (value filter)
  - cross bin `with()` (tuple filter)
- Many scope visibility gotchas
  - good examples are hard to find
  - meanwhile, read LRM carefully and try small test examples

Everything we've looked at in the last few slides has been possible for years in all serious SV tools. Now, though, we move on to look at some recent language enhancements. There were some big improvements in coverage features in the 2012 release of the SV language standard. (If you want the gory details, take a look at https://accellera.mantishub.io/view.php?id=2506)

VCS now supports filtering functions that can be used to describe bins and cross bins in an algorithmic way, without the awkwardness of the binsof syntax. There's an alternative approach, using functions that can return queues of bin values or cross bin value-tuples, that is probably even more powerful – but at the time of writing it's not supported by all simulators.

There isn't yet much in the way of published materials or examples on this, so you will have to do some homework on the LRM and a few simple examples.

Bin Specification Improvements

- Illegal cross bins example taken from a real-world problem:
  - Illegal if `valid==0` or if `valid==ready`
  - Illegal if `valid` has any bits set that are not set in `ready`
- `binsof` expressions: messy, inflexible

illegal example

example for Nbits=2 only!

```
covergroup cg_valid_ready(int Nbits);
  cp_valid: coverpoint valid;
  cp_ready: coverpoint ready;
  cp_validXready: cross cp_valid, cp_ready {
    illegal_bins no_valid = binsof(cp_valid) intersect {0};
    illegal_bins valid_eq_ready =
      (binsof(cp_valid) intersect {1} && binsof(cp_ready) intersect {1}) ||
      (binsof(cp_valid) intersect {2} && binsof(cp_ready) intersect {2}) ||
      (binsof(cp_valid) intersect {3} && binsof(cp_ready) intersect {3});
    ...
  }
endgroup
```

SNUG 2016 22

Here's an example where the new capabilities make your life MUCH better. The example is taken (with some simplification) from the author's project experience. We have two vectors **ready** and **valid**. Each bit position in those vectors represents activity on some part of the DUT, and there are legality rules about the possible values, as described on the slide. Even the very simple illegal-condition "valid==ready" is extremely wearisome to describe using the binsof syntax. If the number of bits were to be configurable – which it was, of course – then this approach is unworkable. We leave it to you as a thought-experiment to code up the illegal bin "valid has some bits set that are not set in ready".

Bin Specification Improvements

- Illegal cross bins example taken from a real-world problem:
  - Illegal if `valid==0` or if `valid==ready`
  - Illegal if `valid` has any bits set that are not set in `ready`
- Easily solved in SV-2012

illegal example

self-tuning for Nbits

```
covergroup cg_valid_ready(int Nbits);
  cp_valid: coverpoint valid;
  cp_ready: coverpoint ready;
  cp_validXready: cross cp_valid, cp_ready {
    illegal_bins no_valid = binsof(cp_valid) intersect {0};

    illegal_bins valid_eq_ready = cp_validXready with (cp_valid == cp_ready);

    illegal_bins valid_without_ready = cp_validXready with
                ((cp_valid & (cp_ready ^ {Nbits{1'b1}})) != 0);
  }
endgroup
```

SNUG 2016 23

Here's the same example, fully coded using the SV-2012 improvements and fully parameterized for any bit width. All that's necessary is for us to code an expression or function that returns "1" for value-pairs that should be in the bin, and "0" for value-pairs that don't belong in the bin. (There's a very similar mechanism available for coverpoints, but it's cross points for which it's most desperately needed.)

Note the slightly clumsy coding of the valid_without_ready bin. A minor VCS restriction means that the obvious formulation of the with-expression:
… with ((cp_valid & ~cp_ready) != 0)
yields a compilation error. The workaround given here compiles and operates correctly, and has the same effect.

It's interesting to note that the number of bits in the vectors is given as an argument to the covergroup, and therefore does not have to be an elaboration-time constant (parameter). It can, if necessary, be computed dynamically provided it's known at the time the covergroup is constructed. More on this issue later.

Note, too, that this is nothing to do with the "iff" qualifier that you can put on to a coverpoint or coverage bin. "iff" means "don't sample into this point/bin if the iff condition is false", but the bin continues to exist and will give you a coverage hole for any values for which the iff-condition is false. With-filters on the bins expressions, on the other hand, control the set of values that contribute to the bin – they affect the

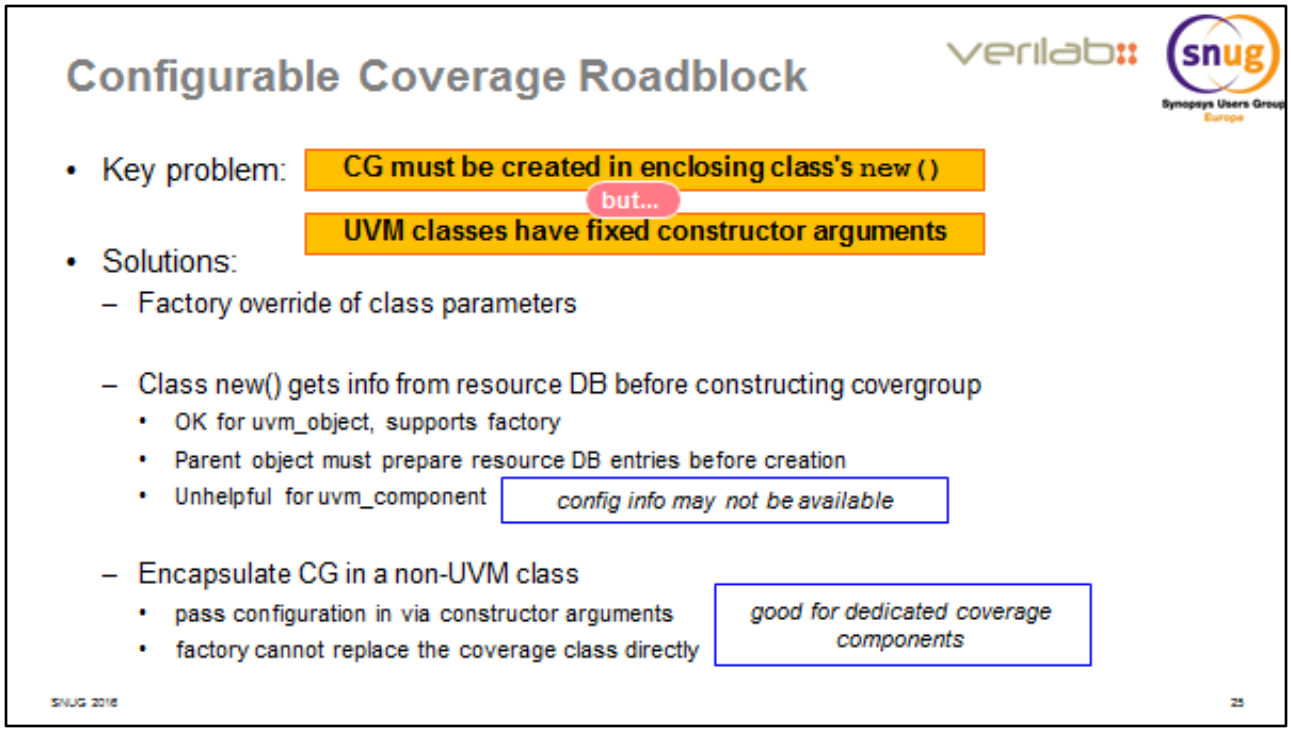
bin’s shape.

Configurable Coverage Roadblock

- Any good reusable VC or env has a **configuration object**
- Can we use this config object to tune coverage?
- YES, but there are challenges:
  - covergroup must be created in a class's constructor
  - this is too early in a UVM component; config is not yet known

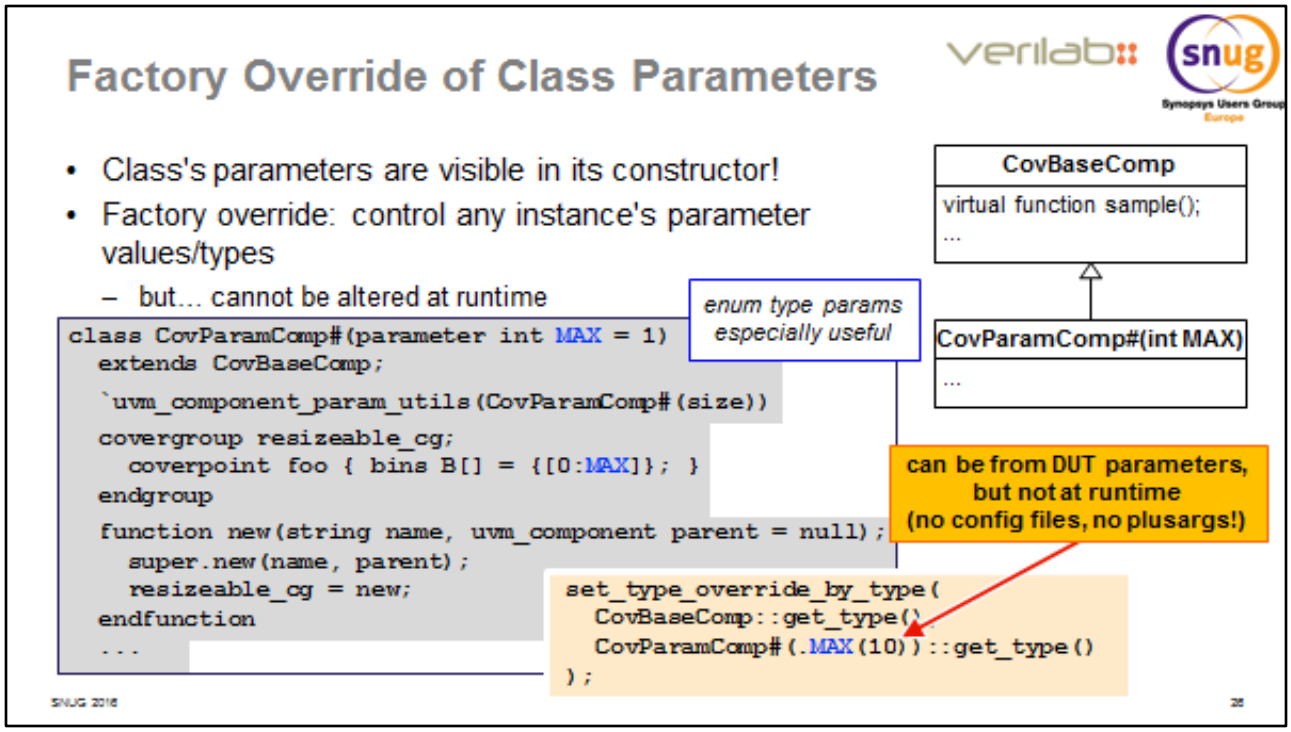
- Following slides offer 3 solutions:
  1. factory override of class parameters
  2. wrap covergroup in uvm_object, get config from UVM config DB
  3. embedded class constructed much later, after config is known

The bit-width concern on the previous two slides highlights a major difficulty in creating re-usable coverage code: The code must automatically self-adapt to configuration values such as bit-width parameters, sizes and value-ranges of data items, the presence or absence of certain options, and so on. In a UVM testench, values controlling such configuration are usually to be found in a configuration object that has been set up in preparation for running any given test. It's not difficult to pick relevant values out of such a configuration object and apply them to construction of a covergroup so that the coverage is automatically tailored to suit the test's and environment's configuration. However, there is a very awkward interaction between UVM's phasing mechanism and the SV language rule that says a covergroup within a class MUST be created by the class's constructor. On the remaining slides we offer three distinct solutions to this problem. You can adapt one or more of these solutions to your own needs.



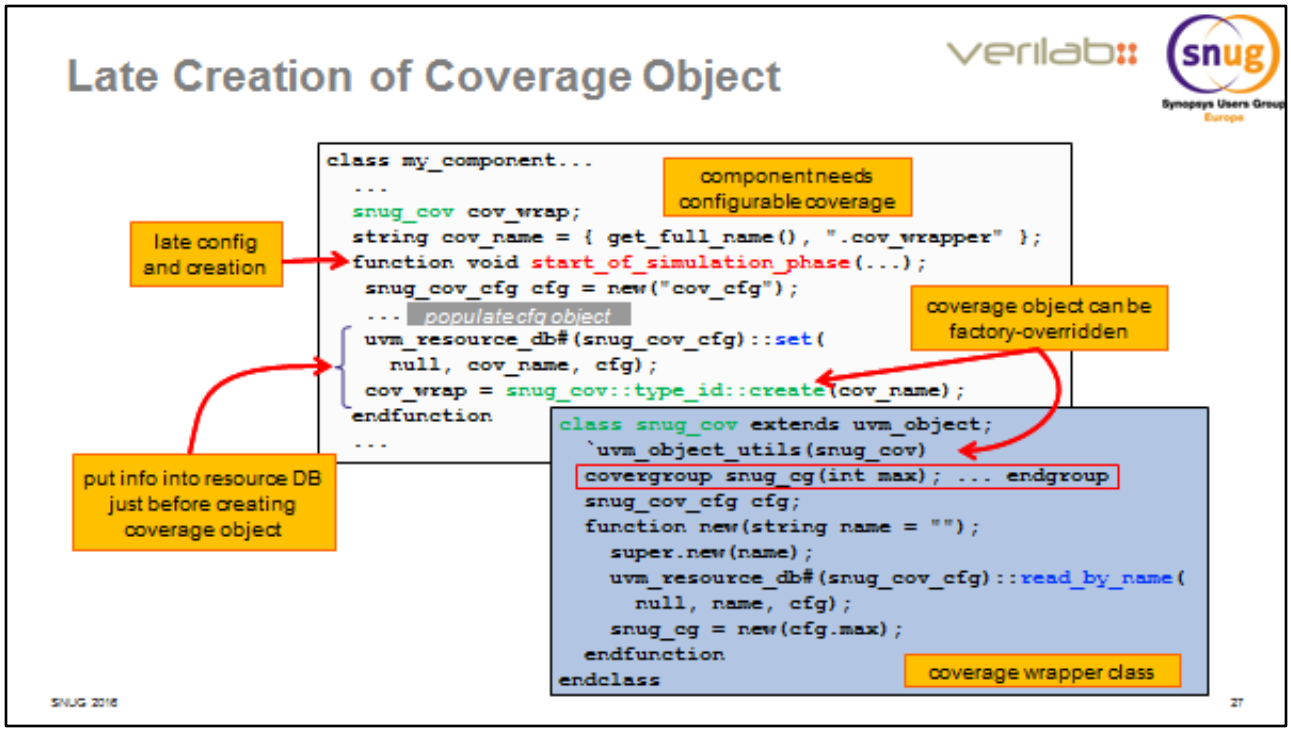
Here, in outline, is the problem. A covergroup *must* be constructed by its enclosing class's constructor, and at that moment the shape of the CG's bins is established – it can't be altered later. However, all UVM classes have a fixed argument list because of the requirements of the UVM factory. It's therefore very difficult to get any custom configuration information into a class's constructor. Later, in build_phase, it's straightforward to get the config information – but that's too late; the class's covergroups have already been built and their shapes frozen.

All three of our solutions require interesting and fairly sophisticated coding techniques. Luckily, in each case the actual code pattern is quite straightforward and is easily reproduced for new situations.



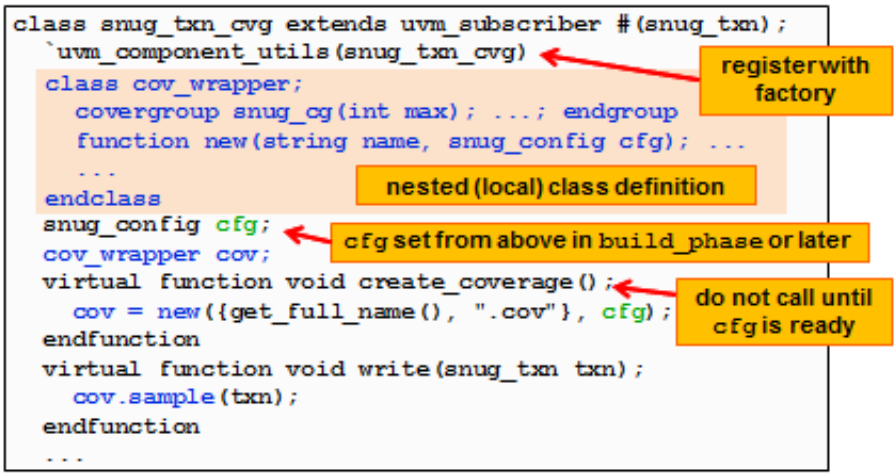
The first solution was provided, during review of this paper, by Dr Dave Long of Doulos – thanks!

Any UVM class that can be registered with the factory (which, basically, means all of them unless you're bending the rules) has a fixed constructor argument list. You can't add arguments to that list without completely breaking factory registration. So, how do you get custom information into the object's constructor so that its covergroups can be shaped? Well… parameters of the class are of course constants, and therefore are visible in the class's constructor and everywhere else too. If we have a base class *without* parameters, and then create a parameterized *derived* class, we can instantiate the base class in our testbench and then use factory override to replace it with a specialization of the derived class having any arbitrary parameter values. Because the parameterization (even in the factory override call) is part of a data type, it must be an elaboration time constant; so this doesn't work if you want the config to come from randomization, or the command line, or from a file. But it is ideal for capturing truly static constant configuration values such as the bit-width of some port in the DUT.



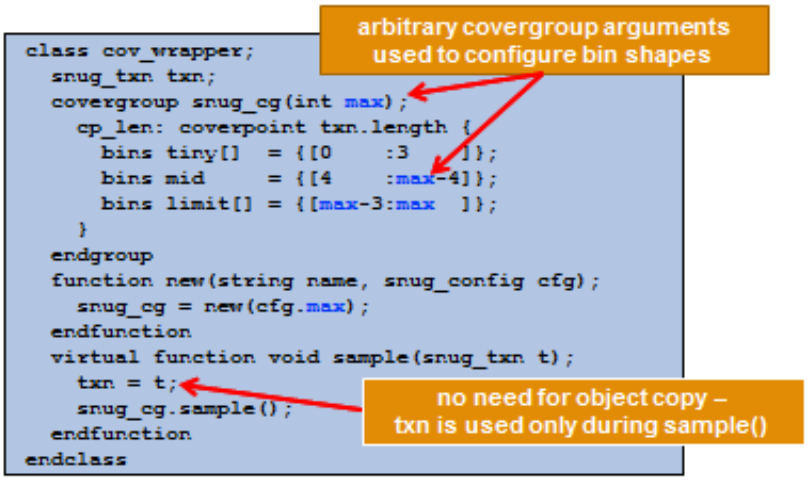
A more dynamic and flexible approach is based on putting the coverage into a UVM object (*not* a component). Construction of this object can be delayed as long as you wish in the life of the testbench, so there's no problem about getting access to the required configuration values. The only problem is how to get them into the coverage class's constructor so they can be used to shape the covergroup. Our example shows how this can be done by creative use of the UVM resource database, putting an entry into the resource DB just before constructing the object. The object's constructor can then look into the resource DB to find the custom configuration information it needs. Note we take quite a lot of care to make the resource's string name completely unambiguous and unique; there's no place for wildcard resource pathnames here.

Embedded Class in Component



Finally, accepting that it is often very helpful to have your coverage inside a component rather than an object, here's a way to achieve that. We accept that no UVM class can get custom config information into its constructor – so we bend the rules a little and create a non-UVM class. But its definition is embedded (nested) within a genuine uvm_component, and we use it strictly within that component class. This non-UVM class can have arbitrary constructor arguments without defeating the ability to register its enclosing component with the factory – so we can easily factory-reokace replace not the inner coverage class, but the whole component. The component can defer creating an instance of the inner coverage class until it has all the necessary config information available. Note that we pass the config information packaged as a configuration object – it's tidier and more flexible.

Embedded Class – Details



Here's some detail of what the embedded (nested) coverage class might look like. Its constructor unpicks the config object it's been given, and uses those config values to provide arguments to its covergroup's constructor and shape the bins appropriately.

Configurable Coverage Component
Summary

- nested (wrapper) class contains covergroup(s)
- not a uvm_object – arbitrary constructor signature OK
- nested-class object can be constructed any time
  - postpone until config is fully known
- component encapsulates responsibility for:
  - understanding and preparing configuration
  - constructing nested-class object
  - data collection and sampling
- prepare for extension, factory applicability

A final summary of that last technique.

An afterthought: Any component gets a handle to its parent as one of its constructor arguments. So it could easily enough look back into its parent and pick out a config object owned by its parent. This is actually a really neat solution *provided* all the necessary config is known by the time of the parent's build_phase. The embedded-class technique on the previous few slides is a tad more flexible, because there's no limit at all on how long you can postpone construction of the embedded class instance, so you're not restricted to config information that's available at build_phase.

verilab

snug
Synopsys Users Group
Europe

Thank You