Navigating The Functional Coverage Black Hole: Be More Effective At Functional Coverage Modeling

Jason Sprott
Verilab Inc.

Paul Marriott
Verilab Inc.
Email:
paul.marriott@verilab.com

Matt Graham
Cadence Design Systems Inc.

***Abstract*- Coverage modeling is an essential component of today's verification methodologies, but it's often badly planned and poorly executed. The topics discussed in this paper are based on the experiences of the authors, working on SoC and IP projects for more than a decade. The aim of this paper is to enable readers to do a better job of developing their coverage models, by understanding the objectives and requirements more clearly, using some best practice styling, and taking reviews of implementation more seriously.**

INTRODUCTION

Despite domain specific language support, advanced analysis tools and over 15 years of functional coverage history in HVLs, many engineers still find it difficult to develop functional coverage models effectively for today's projects. Coverage models are often badly planned, with poorly defined objectives and priorities, dubious estimates of effort, and unrealistic expectations of returns. Even using methodologies such as the UVM, and the built-in SystemVerilog features for functional coverage, implementation can be inconsistent and error-prone. Sometimes, little or no consideration is given to the downstream results analysis, nor peer code reviews. The result can be unexpected project slippage, in the coverage closure stage with a risk for false or misleading results.

The solution is not simply to add, or fix, functional coverage features in whatever language is being used. Nor is it to develop new tools. The real solution is for our developers to better understand the actual requirements and techniques needed for the correct modeling of functional coverage. They must recognize and understand the different kinds of functional coverage, from basic coverage of operational states, through complex use-cases, all the way to the mapping of results to a plan/specification.

At the end of the day, functional coverage modeling, sometimes with complex state and temporal concerns, is essentially a software task. As such, models will likely have bugs and will have to be reviewed and validated for correctness. Given the trust we put in functional coverage results for tapeout decisions, this is an oddly overlooked requirement. An essential component of modeling is to make the review and accurate analysis of results as painless as possible.

This paper aims to address the most important aspects of coverage model development, during planning, implementation (in SystemVerilog), through to review. The experience of the authors in developing, debugging and using functional coverage for verification closure, over many types of projects, will be captured in a set of practical guidelines and advice.

For planning: the different kinds of functional coverage will be examined, looking at how these affect objectives, priorities and implementation. The authors will share how coverage collection concerns (state and temporal), can affect estimates and accuracy of results, with a view to ensuring the best effort versus returns tradeoffs.

For implementation: structure and coding styles will be discussed and the authors' best practice recommendations will be shared. The impact of the newest features in IEEE1800-2012 (SystemVerilog) [1] on implementation style, maintenance and debug (of the coverage model itself), will also be mentioned, though many of these have yet to be implemented in the major simulators at the time of writing.

For review: the focus will be on a practical audit of functionality, not cosmetics. The authors target areas where serious problems commonly occur, often causing misleading or false results. The aim is to bring more consistency and rigor to the review and validation process for functional coverage.

COVERAGE PLANNING

Although functional coverage is just one criterion used in the front-end verification sign-off process, its importance and scope are increasing. Coverage spans scopes from block to chip, and is soaking up adjacent concerns, such as low power, mixed signal and software. We have coverage domain-specific language features, tools and techniques. Coverage statistics are often requested by customers, as proof of verification quality and even live project development status.

The verification plan itself (amongst other things), acts as a bridge for traceability between the multitude of requirements defined in specifications and use-modeling, and the results of the simulations.

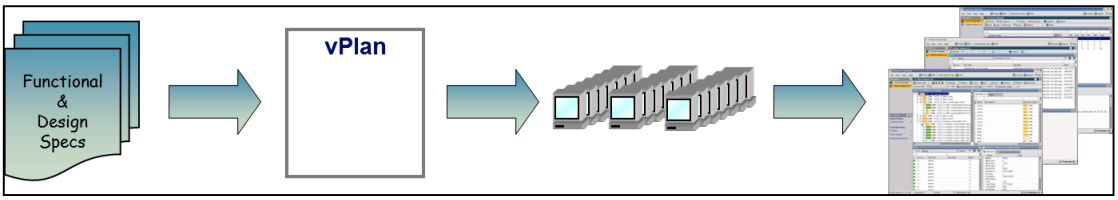

Figure 1: Traceability via Verification Plan

Implementing coverage has probably never been more straightforward. Covergroups allow a high degree of automated bin generation, and we're even offered essentially *free* coverage from register packages and the like. The reality, though, is that behind every coverage definition there is a cost. It takes time and effort to design coverage collection, reach coverage goals, as well as to check and analyze the results.

Coverage is always part of a larger picture in verification. One of the easiest things to get wrong is to forget that. When planning functional coverage, the corresponding checks must also be planned. They are tightly coupled. This sounds obvious, but it is often the case that someone who has no knowledge of the checking capabilities of the testbench has to develop the coverage. Or, coverage can be inherited as part of a VIP, where coverage items are absorbed into the greater coverage model, but don't quite map to relevant checks or goals anymore.

If we consider that functional coverage collection is a statistics gathering exercise, we can examine some simple statistics gathering exercises to illustrate the criticality of joined-up thinking. If a group of 100 athletes were to run a 100 metre course, and officials wished to rank them by time required to complete the course, it's obvious the statistic to be collected would be the time from beginning to end of the course for each athlete. What is equally obvious in this case, but often overlooked in verification, is that checks must be in-place to ensure the athletes all start and stop in the correct location, cover the entire course, and so on. This same principle must be applied for functional coverage, for which appropriate checks of DUT behavior must be implemented to validate that the sign-off statistics (coverage) gathered are valid.

Despite this obvious connection between functional coverage modeling and planning, we often make fundamental mistakes from the outset. The following sections aim to give the reader a better idea of the mindset and actual concerns when planning coverage.

A. The three most important things when planning coverage

However functional coverage is implemented, the mechanics of defining the points themselves is mostly a manual process. Tough luck! Until a machine can interpret a well-meaning, but often vague specification, not specifically written with verification in mind, this won't change much. Despite this, with help from language features and tools, we manage to accumulate a potentially huge amount of coverage data. These results all have to be analyzed and *trusted*.

To this end, the top three most important criteria used when planning coverage are that it be:

- Accurate
- Representative

• Complete

Accurate coverage means that the trigger event(s) and sample(s) capture the desired states or transitions in a way that matches expectations. Our aim is to avoid false positives, hiding multiple paths to the same outcome, or creating results that can be misunderstood.

Representative coverage falls broadly into two categories: functional and priority. It's typically impossible (or at least impractical), to create coverage for every scenario possible in a design. A representative subset of the functionality must be identified that allows us prove we've hit enough to satisfy verification goals. In order to achieve this, quite ruthless prioritization is required. We need to determine the core functionality that's important to the project's business goals.

Complete verification is determined based on scope and a representative subset of functionality. A collectively exhaustive list of features for that scope should be identified. It may not be possible to achieve, or have sufficient time to implement everything that's identified, but without careful analysis subtle and important things will be missed.

B. Kinds of Functional Coverage

In order to plan development of a functional coverage model, it's necessary to understand the different kinds of functional coverage [2,3]. By *kind*, we mean the functionality, or behavior, to be observed, not the implementation style of the coverage code itself. Some kinds of coverage are easier to collect than others. This usually comes down to how much work is involved in triggering and sampling the coverage accurately. The following are typical kinds of functional coverage:

Use-cases: This is one of the most difficult kinds of coverage to collect and decide upon. Use-cases are typically a top-level view of how a design will actually be used (at block or chip level). This may be in relation to software control and interaction, or discrete signal control. This kind of coverage is difficult to collect, as use-cases are often quite abstract descriptions involving multiple operations, performed during various starting and transitioning states. It can be difficult to decide on the most interesting and relevant order of events. It's easy to come up with many different valid variants on the same theme.

Interesting scenarios: We split this out as it represents a case for making the verification environment flexible and able to handle unforeseen extensions. An interesting scenario can be something outside the normal coverage already being collected, but could be related, or an extension of it. For example: a specific cross between coverpoints, or sequence of events, that sensitizes a particular area of the design.

Register Coverage: This is related to the correct connectivity, access policies, reset states, etc. of the registers (control and status). Register coverage is rarely useful in relation to the functionality actually connected to those registers. We will talk about this a bit more in section F.

Protocol: One of the most common and easily understood kinds of coverage, this relates to the various aspects of a given protocol, e.g. in AMBA AXI4 the various read modes, or response types. While this can usually be treated in isolation and is generally thought of as "portable", some care needs to be taken. It's definitely useful to collect observed behavior on the physical interface of a protocol, but how relevant that coverage actually is depends on what else is going on. For example, observing a transaction that never reaches an endpoint (and is never checked), due to some system-level behavior, may invalidate the coverage collection on that operation.

Modes of operation: Probably the little brother of the use-case, these are the various modes of operation on the design. This may involve programming particular register values or observing conditions. It's most decidedly not simply register coverage. Here we must be very careful to ensure the observed operations actually happen and are functionally meaningful.

Responses: Typically this is a cause and effect concern; e.g. we perform some operation and expect to see a particular response. This can actually be quite tricky to collect for a number of reasons. The operations may be split temporally with responses occurring out of order, or not at all. The particular response may be a result of multiple different conditions, e.g. a slave error on a bus, where the actual root cause of the error is important to us, but buried in a status register somewhere in a slave. It's a checker's job to ensure the cause and effect behavior is correct, but if we want to collect meaningful and accurate coverage, we do need to ensure the coverage measurement matches with the originating operation.

Temporal proximity or pattern-based: We've lumped these together, as they are sort of the same. The concern here is that a specific type of operation is required to sensitize the design. Temporal proximity might be something happening within a specific time of another operation, such as a cache entry operation, or arbitration condition. Pattern-based conditions are things that require specific data, or series of data to be present, e.g. a write-combine buffer [4], where the patterns of values written to a register affect the follow-on behavior. These sound like corner cases, but in fact there are many cases where temporal and pattern sensitivity are relevant in normal operation. This kind of coverage can present some challenges.

State (machines): It's often thought (and sometimes true), that FSM coverage can be done using code coverage tools. The problem (amongst other things) is that in order to do that, there needs to be a guarantee that observing a given state using code coverage means that something interesting is actually happening. That's outside the scope of regular code coverage. It's easy to implement basic state coverage in a covergroup, but it's important when doing that, to extend thinking to what else is going on around the FSM. There may be subtle correlations between a given FSM and its inputs, or effects of the outputs to the machine. For example, there may be multiple conditions driving a single input to the FSM; or multiple connections to a single output from an FSM, which react differently depending on their state. Interesting scenarios can often occur during normal operation.

Cross-cutting concerns: Good examples of a cross-cutting concerns are reset and changing power modes. A cross-cutting concern is something that can happen in parallel (often randomly), affecting other things. The problem is that due to the many possible scenarios, it's necessary to pick a handful of the most interesting. It's rarely trivial to collect this type of coverage, due to the parallel and apparently disconnected nature of the samples. We might collect power state information in a completely separate way from the enabling of a particular mode of operation; but somehow we need to correlate them to collect coverage.

Error injection: It's typically best to observe the *result* of an injected error, but when that result could have happened due to different conditions, it's sometimes necessary to correlate the cause and effect. This can be tricky. Some considered but pragmatic thinking is usually required.

Illegal conditions: These are things that really can't, or shouldn't happen in a design. They are special cases from a coverage point of view. If you can identify them, state-space and correspondingly functional coverage implementation can be reduced. Lumping illegal stuff together and being able to ignore it for analysis is very useful. It is possible to handle illegal conditions using covergroups in a way that highlights if you've got your assumptions wrong, by issuing an error. There are pros and cons to this approach, but it's worth considering. The assessment is of the risk missing a mode that is actually possible in a design, versus the effort to implement an illegal bin set. It's certainly not effort free, and can be fragile, so a blanket rule is typically not appropriate, especially as there is a subtle difference between "can't" and "shouldn't".

Analog - mixed signal: There are many useful coverage points to be collected related to the interaction between analog and digital domains. What's actually possible depends on the models being used (accuracy and capability), and the useful values. As described in [5] section 2.6, the objectives and scope of the coverage have to be well understood and planned. Understanding how to organize continuous real values and when to sample them is a specialized concern and should be planned as such.

C. *The Verification Plan & MECE*

Coverage models can be very complex. The act of interpreting a specification and turning it into a set of collectable, traceable coverage points is a manual process and can be error-prone. At the very least, it's variable in terms of quality depending upon the knowledge and experience of the team doing the work. The scope of a previously developed coverage model (for an imported VIP for example), can change depending on where it's being used. Something that's relevant at block-level, may not be at chip-level, or could be relevant only at a specific time. It's important to have a clear picture of the coverage objectives, relevant to the content being worked on.

One useful tool when thinking about coverage planning is to use the *Mutually Exclusive, Collectively Exhaustive* (MECE) mindset [6]. Using this model (illustrated in Figure 2), we break down a requirement (functionality or use-case), into independent definitions. Our aim is to make these definitions independent from one another, so as to avoid repetition, but at the same time cover the whole intent, without gaps. Mutual exclusivity is not just a matter of efficiency, as situations where there's repetition in different areas of a coverage model can be indicative of a poor structure, or even a source of misinterpretation during analysis. Something may look covered, but is actually only sensitive to a specific set of conditions.

Mutual exclusivity between definitions for a given level isn't actually necessary; it's more a case of independence. For example, it may be possible for one definition state to occur in parallel with another, but it should always be the case that they are independent features. An example of that might be a response to a given command/instruction, where the response itself is an independent feature, but may, or may not happen as a result of that feature.

The most important part of MECE from a verification point of view is to be *collectively exhaustive*. This means that when all the definitions are taken together, they completely address the coverage space of interest. If something is missing from the model, it does not appear as a coverage hole, it's simply invisible. That's a serious flaw, as a bug may slip through as a result of functionality not being hit. It's important to note this is not about addressing the complete functional space exhaustively, as that may be impossible. Our aim is to specify interesting cases and build as complete a model as possible, accounting for scope and priorities. To review a coverage model implementation properly, collective exhaustiveness must be checked.

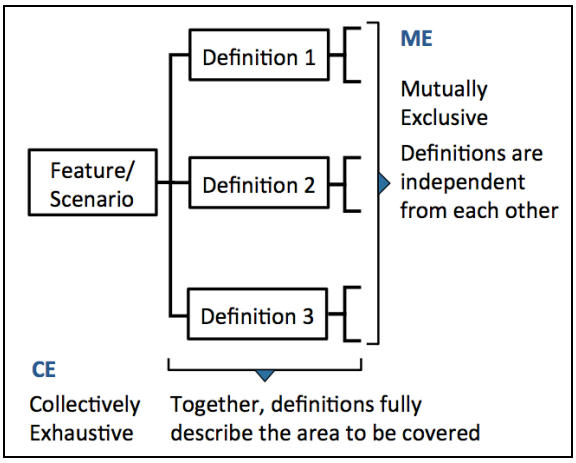

Figure 2: MECE for Coverage

So, how do we plan if we can't address the whole functional space, even if it's possible to build a coverage model that can represent it? One of the most important aspects to answering this question are the business goals of the project. In most projects, *ruthless prioritization* is required. Our experience has shown that we need to get the highest priority, highest risk, things done early. The 80/20 rule shown in Figure 3, illustrates that the biggest bang for the buck is in the early stages of development. Spending too much time trying to perfect a model can be a false economy. Worse still would be working on low priority items before completing the high priority ones. This happens a lot on projects, especially when the low priority

items are easy to implement. As mentioned earlier, writing the coverage definition itself is only part of the story.

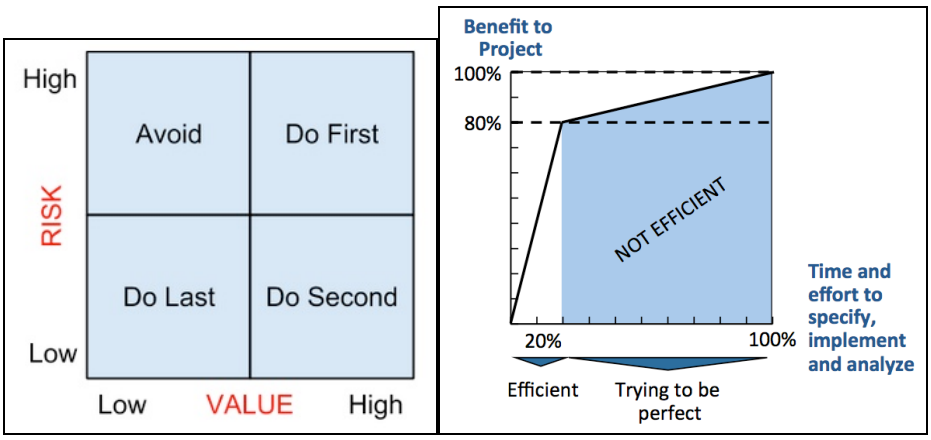

Figure 3: Risk/Value and 80/20 Illustration

D. AXI protocol example

In some cases grouping features for MECE style thinking is quite straightforward. Looking at the AXI4 bus protocol, we can quite easily group by independent transaction type, and then break down according to the various items within that transaction (as illustrated in Figure 4). If we only concern ourselves with the protocol from the perspective of the bus, we can identify interesting values and relationships such as the various components of a transaction: length, byte lanes and address alignment. Another example might be the various response types for a given transaction. We should be able to come up with a collectively exhaustive set for each independent transaction type.

However, it doesn't take much effort to think beyond that bus-level perspective. An example might be the slave error response (highlighted as implementation-specific). The AXI protocol doesn't define what might cause this response, but in our system, we have various types of slaves, each with different conditions, that might cause a slave error. This is an application-specific concern, but in fact it could also be a system concern. From a reuse point of view we'd want to group the generic features separately from the application specific ones, which only apply to a specific block.

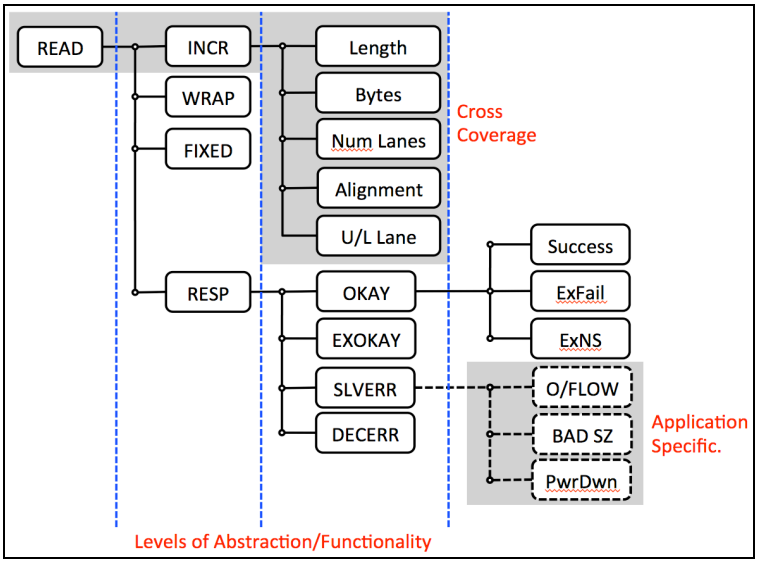

Figure 4: AXI breakdown

Also, even at the protocol level it's easy to mix up levels of abstraction or functionality that affect the validity of the coverage model. Two examples of this are:

- *Odd man out*: this is where a feature doesn't belong in the model at all. For example, in the AXI example, having low power concerns as part of the raw protocol coverage, at the same level as READ would probably not be appropriate.
- *Abstraction blur*: this is where a function or requirement is at the wrong level. For example it would be strange to have SLVERR at the same level as READ in the AXI example.

Abstraction blur may seem unlikely, but in fact the way we build over coverage models, it can easily happen. In covergroups, we typically build cross coverage from independent coverpoints. In some cases the coverpoints themselves have no real relevance, other than as part of a cross. Even in our AXI example, this might be true. *Length* on its own is really only interesting in the context of more than one (or even all) of the other modes. At best the result could seem misleading and waste time. Worst case someone decides to merge results, hiding reachable and important functionality.

E. Audio Hub example

While protocol examples work very well for demonstrating coverage, real life is often very different. Even in a simple system, there can be a jumble of related concerns that are loosely defined and difficult to list.

Figure 5 illustrates a greatly simplified example of an audio hub (e.g. [7]). It illustrates some system-level thinking, which is less structured and more difficult to identify the relevant coverage concerns. In this audio hub there are a lot of mixing options from various inputs to outputs. Various filters, sample rate converters (SRC), and dynamic range control (DRC) components can be incorporated.

In this example we show a two data paths, where we dynamically switch in an incoming voice call, while listening to music from an MP3. This is one of many data path scenarios possible, and itself has a large number of factors that can affect behavior, e.g. volume adjustment, different sample rates for mixed sources. There can be many different initial and exit states to the scenario, such as kicking off from a power down state, or returning to the music once the call is over. What seems like a simple use-case can be quite challenging to define a collectively exhaustive coverage set. A deep understanding of expected use and system limitations is typically required.

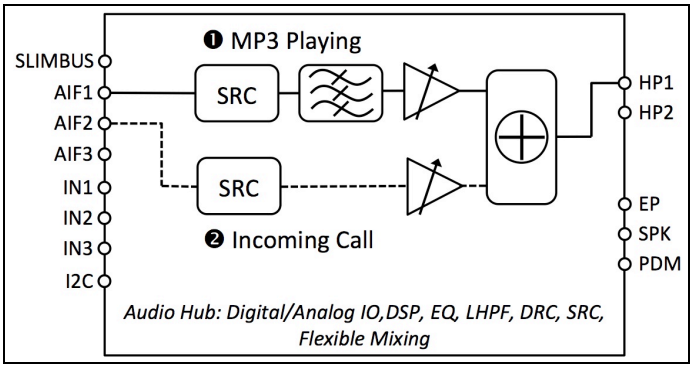

Figure 5: Use-case - Audio Hub (simplified)

The discipline of MECE thinking and analysis can help identify the interesting coverage, allowing prioritization and implementation planning. However, some thought has to go into the starting point – the feature that identifies its independence from the others. For example, would the starting coverage feature be everything related to interrupting events while playing MP3s, or would it be a voice call interrupting other activity?

The audio hub is a good example of the difficulties of sampling the coverage data correctly. In the diagram above, there are two audio paths so one might think that coverage should be placed on the volume levels of each channel, to ensure they are all accounted for. Just covering the register settings can be insufficient to account for all the behaviors. For example, one of the audio paths may have to be muted when the other one is activated, but the switch from one to the other may also include a volume ramp function. The muting may be atomically covered as well as the volume ramp, but it is the sequence of

operations that are executed that is important as there are behaviors which, though valid operations, but lead to unwanted artifacts. The sampling of values out of the context of these sequences of operations is an example of oversampling as it gives the impression that coverage has been obtained, but functionality is actually overlooked. The order in which configuration settings take place can be as important as the collection of settings themselves, especially when there are multiple possible paths to get to the same end result. The correlation of system behavior and the sampled values is an important consideration.

At the end of the day what matters is the MECE thinking where independent features (whatever they may be), are broken down to a collectively exhaustive set, with respect to project priorities. This can be a difficult and iterative process, but it's important for both implementation and review of the coverage model.

F. Register coverage – a blessing and a curse

In register model VIPs, there's often the built-in capability for "register coverage". This can be both a blessing and a curse. Some would classify this type of coverage as akin to code toggle coverage and it does feel like that sometimes; however, it can be more useful than that. Access policies, memory mapping, reset states, connectivity and even some power-aware behavior can be verified, with corresponding functional coverage. It's this author's experience that this can often lead to a very false sense of security.

Register coverage is typically not related to the functionality of the blocks the registers themselves are connected to. Take our example in Figure 5, where a system-level use-case is performed, primarily by programming the chip via registers to do certain things. The sequence and values in registers, related to one another, is an important part of the coverage definition of that use-case. Specific coverage goals have to be identified for that. This type of functionality is very unlikely to be addressed by register coverage.

You would think this obvious, but it can be confusing when a set of results tells you that all the values for a given control register have been covered. In fact the act of programming these values is meaningless unless the corresponding functionality is activated *and* has a corresponding check. As mentioned in the audio hub example, the order of programming may also be important and any other sequences of operations in the system which effectively form a functional aspect that is not directly visible by the settings of the configuration registers themselves.

G. Coverage collection concerns

The concerns related to coverage collection are captured quite nicely in [8], in particular the section *"Covergroup based coverage considerations"*. Whether the implementation is done using covergroups, or some other way, it's important to focus on collecting meaningful (functionally), accurate and useful (to the project) coverage. Especially in the case of covergroups, it's quite easy to sample values and generate coverage for a large dataset with a very small amount of coding. Cool, but not always helpful.

A useful process to is to ensure the following areas are specifically addressed as part of the planning:

Identify important values: this sounds obvious, but it can be quite difficult, especially when the relevance of a value depends on something else, e.g. a filter setting, relying on a previous stage, or input frequency of the stimulus. In its simplest form we're looking at discrete values, or ranges of values we care about, but even these might need to be grouped in various degrees of granularity to ensure downstream analysis is practical. Normal cases, handled exceptions, and corner cases need to be identified. It is often the case that the sampled values have no functional significance. At the end of the day we're looking to thin out a massive state space to things that are most important to us.

Relationship between values: this usually falls into three categories: direct correlation between values in the current sample, relationships that transition between states, cause and effect. Often thinking about these relationships can create a lot of interesting coverage scenarios. For example, simply thinking about an enable related to a particular setting can lead to thinking about the entering or leaving the enabled state, when that might allowed to happen, if the setting can be changed dynamically, or other adjacent behavior that might affect operation. It's easy to get carried away, but in fact these observations are sometimes very relevant.

Illegal conditions: these are values that are not allowed to happen. There are various ways to handle this type of thing, but identifying these values can help to reduce the state space to be covered considerably. Whether it's useful to trap these as errors, or just ignore them is implementation specific. One objective of identifying these conditions is to avoid them ending up in coverage definitions as areas we'll never be able to hit. That can waste a lot of time and effort in downstream analysis.

When to sample: here our aim is to understand when best to sample valid and stable data. Many false positive coverage problems occur due to greedy or sloppy sampling. By that we mean that the sample may be valid in some, but not all of the cases, or apply to some but not all of the data. False positives are horrible and to be avoided at all costs. Just like checks, we wouldn't know anything was wrong until a bug popped up in an area we *thought* we'd verified, but had actually failed to hit. It's also important to avoid oversampling, so use the lowest frequency sampling event that makes sense and ensure it is related to the functionality being verified. Under sampling is also problematic, but oversampling gives a false sense of security. Spend some time on this, especially the cases that affect coverage measurements that span multiple samples.

When to ignore: this is related to all of the previous point, but it's worth addressing specifically. Where the previously mentioned areas are mainly concerned with looking for events and states to cover, this specifically looks at things that we want to ignore. It's a subtle, but important distinction from a mindset point of view. We might decide to ignore something for functional reasons, e.g. through particular modes such as reset, or power transitions/changing clock gearing. Alternatively, we might just decide that something is lower priority, or too difficult to handle reliably. This might affect sampling events or sampled data equally.

Conditionals: another form of false positive coverage count is a conditional treated as one fused value. A simple example might be a sequence such as *(A or B) implies C*. Here if we don't cover the *A* and *B* cases separately, we'll never know if all combinations of both were taken. This is quite often seen in cover properties, but also applies to sampling events and covergroups. Conditions that hide paths and state combinations come in many shapes and forms.

Post-sample behavior: this is another thing worth thinking about on its own. For any given sample it's useful to apply some thought to events that may change the validity of that sample after the fact. Since we're often observing a setup-and-go scenario and relying on checking to pick up errors, it's often easy to overlook legal behavior where an operation may get dropped (for example). Checkers typically have to handle this kind of thing otherwise they end up failing every 5 minutes. Coverage, on the other hand, might not as nothing falls over by incrementing a bin without fully validating that the operation made it all the way through.

Accuracy: it's always useful to think about how accurate any item in a coverage model must be. Covergroups, in particular, tend to group data together in transactions, based on a sensible sample point, which may or may not coincide with when the associated data appeared on a net somewhere. Modeling takes a lot of effort and sometimes a tradeoff needs to be made between accuracy and effort. It can be acceptable if a certain behavior is assumed, not observed. However, it's *never* acceptable to assume something is acceptable. Once an inaccuracy is built-in, it can be very difficult to spot, much as turning off a check can be difficult to spot. If a tradeoff needs to be made that may compromise the results, requiring downstream review, make sure it's obvious.

These areas are also, not surprisingly, used for reviewing functional coverage implementation. If a reviewer has a list of project priorities and targets for a coverage model, it usually doesn't take long to find some issues based on this list.

REVIEW OF SYSTEMVERILOG 2012 ADDITIONS

There are some noteworthy improvements in SystemVerilog 1800-2012 [9] related to functional coverage (covergroups specifically). When implementing a coverage model in a covergroup, syntax for describing bins and crosses can become quite obfuscated, especially when expressing non-trivial selections. The following improvements will help with this, though, at the time of writing, the major simulators do not support all of them:

Coverpoint variables: The coverpoint label is now a variable name for that coverpoint and can be defined as a specific integral type. This may be particularly useful when carving out a specific subset of values to be covered, allowing for more meaningful crosses and less complex *with* expressions.

Coverage bin ... with expressions: This provides the option to refine coverage bins more easily and precisely. Each value of the variable to be covered can be qualified by an expression, with binning only created for the values returning true. This is distinctly different from the *iff* expression, which only provides the option disable coverage collection on a coverpoint, but doesn't affect the bins created.

Functions in covergroups: This is a valuable addition to the expressiveness of covergroups. In many cases a function can provide a much clearer and easier to review/debug definition than the typical *binsof* and *intersect* equivalent, which can be very difficult to understand.

COVERAGE IMPLEMENTATION BEST PRACTICE CHECKLIST

1. Ensure any reused coverage is still accurate and meaningful, e.g. observing accesses modes on a bus interface may need to be tuned, when using at higher levels, based on target destination, or slave responses.
2. Name/Comment coverage based on coverage *intent* not simply the feature name. Thinking about how someone should interpret the code can speed up both analysis and review.
3. Group multiple conditions/threads appropriately, i.e. don't potentially hide untaken paths.
4. Use bin ranges to help rationalize the amount of data to analyze, but ensure the ranges are both meaningful and don't hide important corner cases.
5. Coverage should be focused on functionality, not simply bit toggling (which can be observed using code coverage tools). If bit values, in a register for example, are being covered any correlation with other modes should be accounted for in the sampling, conditional triggering, or crossing.
6. Reduce coverage, especially complex crosses, to the useful values.
7. Use illegal bins in covergroups sparingly, for situations where generating an error is useful to debug assumptions, or stimulus, rather than functional correctness checking (better done explicitly in a checker). Provide a mechanism to disable.
8. Utilize coverage in register packages for access policy and generic register related behaviors, but full toggle and design feature functionality are typically better handled differently.
9. Split covergroup and SVA cover statement implementation according to their strengths, e.g. SVA can be more practical when collecting temporal behavior on a physical interface.
10. Include sequence/scenario coverage. Some simple functional coverage to determine what "abstracted" sequences or use cases have been exercised can be very helpful. Don't assume the scenarios get executed simply because they are written.

COVERAGE IMPLEMENTATION REVIEW

There is often some confusion when it comes to coverage reviews. Are we reviewing verification closure against a plan (which might also include a bit of poking about in the code), or are we doing a code review

of coverage implementation? Well of course we need both, but the latter is rarely done properly. In this section we address only the review of functional coverage implementation.

There are a number of *bad smells* in coverage implementation, not all of which are necessarily errors, but may just make reviewing, or maintenance difficult or inefficient. The review points fall broadly into the following categories:

- Style: for inspection, maintenance, and reuse
- Sampling events: errors, over/under sampling
- Bin triggering: errors, relevance, missing

When project realities and schedule pressures necessitate taking short cuts with coverage development, the resultant coverage may be neither optimal nor ideal, but still needs review. We must scrutinize the value and reliability of the statistics gathered. If short cuts result in data that is "mostly reliable" but still gives reasonable value in terms of an indication of the functionality exercised, it may be considered good coverage. Conversely, if the coverage will result in misleading, confusing, or otherwise worthless data, it will only cost more than it returns in the long run, and should be either reworked or discarded.

When performing a general review of the implementation the following process can be useful:

1. Assumptions:
    a. The verification plan of record has been reviewed
    b. Any priorities are valid
    c. Access to checkers is available (required to validate if coverage is meaningful)
2. Look at imported coverage validity (e.g. block-to-chip)
    a. Is coverage still meaningful?
    b. Is coverage still accurate?
3. Deep dive (normal operation)
    a. Are the coverage concerns addressed (listed in Section G)
    b. Has the coverage developer reviewed the associated checks? If not, this is typically a red flag.
    c. MECE analysis spot-check to get a feel of the quality
4. Deep dive (corner cases)

During a review, our aim is to pick up actual coding errors that would result in false positives, or hide important conditions that we need to ensure are covered. It's entirely possible at the time of writing a complete specification, or plan, wasn't available. Priorities may have been different, checks not written yet, or work may have changed hands a number of times. The review is the best opportunity to assess status and highlight issues. Even serious issues can usually be addressed pragmatically if they are discovered. Undiscovered issues are bugs in silicon waiting to happen.

1. Review the verification plan and consult it when reviewing the coverage. Coverage is part of the signoff criteria for the device, and should be given equal scrutiny to the RTL and verification environment code.
2. Review the coverage that is being analyzed. There may be a great deal of coverage that is not part of the verification plan, or that is not being analyzed. Don't review its implementation in detail if it's not part of the signoff criteria.
3. Coverage that is implemented or collected that is NOT part of the verification plan should be reviewed in so far as understanding WHY it's not part of the verification plan. There are many legitimate reasons why implemented and collected coverage is not part of the verification plan, but they should all be scrutinized and confirmed.

4. Each piece of functional coverage should have an associated check. Capturing statistics about features exercised in the device is useless if there is no corresponding check to determine if the device operated correctly in response to the stimulus. Both the check and the coverage should be part of the verification plan section for the given feature.
5. When reviewing the verification plan, scrutinize whether all relevant coverage has been mapped to the specific section. Are there portions of code coverage, other covergroups, additional assertions, etc. that can and should be included to determine completeness of the verification of a specific feature.
6. Sampling of functional coverage should be reviewed to ensure that coverage is not oversampled, or sampled incorrectly. Oversampling can lead to false positive results, indicating scenarios as being covered when they have not been appropriately exercised.
7. Binning and illegal conditions in coverpoints should be reviewed to ensure the values are correct and meaningful. Bins need not be too large or too small to capture actual interesting corners of the design or implementation.
8. Sections of the coverage plan, and their mapped coverage, should be reviewed to ensure alignment with functional specification and business goals. If a particular section of the verification plan is difficult to connect to either of these two sources, it may be poorly defined, or redundant.
9. Prioritization or weighting of specific coverage elements and sections of the verification plan should be aligned with priorities of technical and business goals. This should include reviewing which sections of the device have the greatest percentage of "new code" compared to previous revisions, as well as which are the most critical to the end customer.
10. The verification plan and coverage implementation represents one of the key signoff criteria for ASIC development. Involving all stakeholders, including SW, HW, system, etc. in the review of the plan and potentially the coverage implementation is critical.

CONCLUSIONS

The task of implementing coherent functional coverage is one that is very often underestimated. It can be deceptively easy to get high functional coverage figures, but being able to thoroughly trust these is the key to success when it is used a sign-off criterion.

We have described the different kinds of functional coverage and how to plan, implement and review a coverage model. At the time of writing we had hoped to include more details of the 1800-2012 extensions to SystemVerilog for coverage, but their implementation in the main simulators is not yet mature enough.

REFERENCES

[1] IEEE 1800-2012 http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[2] Paul Marriott and Steven Bailey, *Functional Coverage Using SystemVerilog,* DVCon 2006

[3] Andrew Piziali, *Functional coverage measurement and analysis*, ISBN 1-4020-8025-5

[4] http://en.wikipedia.org/wiki/Write_combining

[5] Bishnupriya Bhattacharya et al, *Advanced verification topics*, ISBN 9781105113758

[6] Barbara Minto, *The pyramid principle – logic in writing and thinking*, Third Edition, pp96, ISBN 978-0-273-71051-6

[7] Wolfson Microelectronics, WM5102 Audio Hub CODEC Product Brief

[8] Gordon Allan et al, *Coverage Cookbook*, Mentor Graphics, available on-line https://verificationacademy.com

[9] Stuart Sutherland and Tom Fitzpatrick, *Keeping Up with Chip – the Proposed SystemVerilog 2012 Standard Makes Verifying Ever-increasing Deign Complexity More Efficient* http://www.sutherland-hdl.com/papers/2012-DVCon_SystemVerilog-2012_paper.pdf